buildout.coredev Documentation

Release 4.2.beta2

The Plone Foundation

June 16, 2015

Contents

CHAPTER 1

Introduction

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

1.1 STOP!

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

1.2 Table of Awesome

1.2.1 Contributing to Plone Core

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

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

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

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

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

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

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

Welcome to the Plone community!

Dealing with pull requests on GitHub

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

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

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

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

1.2.2 Plone Developer Culture

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

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

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

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

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

1.2.3 How To Commit Fixes to Plone Core

This document assumes you want to fix a bug and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy

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

Dependencies

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

Setting up Your Development Environment

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

To set up a plone 4.2 development environment:

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

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

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

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

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

Switching Branches

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

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

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

```
> git checkout 4.1
```

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

```
> git branch
```

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

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

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

Checking out Packages for Fixing

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

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

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

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

Then rerun buildout to get the source packages:

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

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

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

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

Testing Locally

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

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

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

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

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

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

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

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

Updating CHANGES.rst and checkouts.cfg

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

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

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

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

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

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

Committing and Pull Requests

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

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

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

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

Committing to Products.CMFPlone

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

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

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

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

First check out the branch:

```
> git checkout 4.2
```

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

> git cherry-pick b6ff4309

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

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

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

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

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

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

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

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

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

Branches and Forks and Direct Commits - Oh My!

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

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

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

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

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

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

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

```
> git push origin my_descriptive_branch_name
```

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

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

Jenkins

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

Finalizing Tickets

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

FAQ

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

validate that yours is present.

1.2.4 Writing documentation

Documentation of Plone

As a community, Plone maintains several types of documentation:

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

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

Documenting a package

The basics

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

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

`README.rst` should include:

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

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

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

It includes topics like:

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

- Common gotchas

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

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

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

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

Reference (a.k.a. API documentation)

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

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

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

A changelog looks something like:

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

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

* Documented changelogs.
  [davisagli]
```

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

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

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

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

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

1.2.5 Style Guide

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

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

Naming Conventions

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

File Conventions

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

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

compare that to:

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

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

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

compare that to:

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

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

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

Concrete Rules

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

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

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

1.2.6 Implementing PLIPS

All about PLIPS

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

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

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

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

A champion should:

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

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

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

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

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

PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

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

Process Overview

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

How to Submit a PLIP

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

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

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

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

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

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

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

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

Evaluating PLIPs

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

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

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

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

See the `plipreview` page for more information.

Implementing Your PLIP

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

General Rules

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

Creating a New PLIP Branch

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

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

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

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

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

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

Finishing Up

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

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

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

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

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

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

1.2.7 Troubleshooting

Buildout Issues

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

Errors Running bootstrap.py

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

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

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

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

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

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

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

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

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

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

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

Hooray!

When Mr. Developer is Unhappy

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

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

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

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

You can also get such fun errors as:

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

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

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

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

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

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

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

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

Hooray!

mr.developer Path Errors

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

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

To fix, simply do:

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

Other Random Issues

Dirty Packages

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

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

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

No module named zope 2

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

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

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

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

Missing PIL

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

Modified Egg Issues

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

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

1.2.8 How to Update these Docs

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

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

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

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

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

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

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

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

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

1.3 Translations available

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

- `Spanish version`.
- `Portuguese version`.