buildout.coredev Documentation

Release 4.3

The Plone Foundation

Jun 21, 2017

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.

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!

CHAPTER 2

Contents

Guidelines for contributing to Plone Core

You probably came here by clicking one of the 'guidelines for contributing' links on GitHub. So you probably have an issue to report or you want to create a pull request. Thanks a lot! Let's bring you up to speed with the minimum you need to know to start contributing.

Create an issue

- If you know the issue is for a specific package, you can add an issue there. When in doubt, create one in the CMFPlone issue tracker.
- Please specify a few things:
  - What steps reproduce the problem?
  - What do you expect when you do that?
  - What happens instead?
  - Which Plone version are you using?
- If it is a visual issue, can you add a screen shot?
- If there is an error in the Site Setup error log, please include it. Especially a traceback is helpful. Click on the 'Display traceback as text' link if you see it in the error log.

Create a pull request

- 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.
- Add a changelog entry in `CHANGES.rst`. Make it clear if it is a fix or a new feature. When in doubt, don't worry about it.

- For new features, an addition to README.rst is probably needed. A package may include other documentation that needs updating.
- All text that can be shown in a browser must be translatable. Please mark all such strings as translatable.
- Be nice and use code quality checkers like flake8 and jshint.
- See if you can use git to squash multiple commits into one where this makes sense. If you are not comfortable with git, never mind.
- If after reading this you become hesitant: don't worry. You can always create a pull request, mark it as WIP (work in progress), and improve the above points later.

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 git repositories 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 repositories, 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 below 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, 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.

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.

Getting started with development

This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, 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 2.7 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 5 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 5.0 https://github.com/plone/buildout.coredev ./plone5devel
> virtualenv --no-site-packages plone5devpy
> cd plone5devel
> ../plone5devpy/bin/pip install -r requirements.txt
> ../plone5devpy/bin/buildout bootstrap
> 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!

Jenkins / mr.roboto

Plone has a Continuous Integration setup and follows CI rules.

When you push a change to any Plone core package, our testing/CI middleware `mr.roboto` starts running all the tests that are needed to make sure that you don't break anything. For each Plone and Python version we run two jobs, one for the package itself (which will give you a fast feedback, within 10 minutes) and one on the full coredev build (which can take up until an hour, but makes sure no other packages are effected by your change.

For more information you can check *Mr. Roboto workflow* or our Jenkins machine.

The CI system at jenkins.plone.org is a shared resource for Plone core developers to notify them of regressions in Plone core code. Build breakages are a normal and expected part of the development process. Our aim is to find errors and eliminate them as quickly as possible, without expecting perfection and zero errors. Though, there are some essential practices that needs to be followed in order to achieve a stable build:

1. Don't Check In on a Broken Build - check Jenkins before
2. Always Run All Commit Tests Locally before Committing
3. Wait for Commit Tests to Pass before Moving On
4. Never Go Home on a Broken Build
5. Always Be Prepared to Revert to the Previous Revision
6. Time-Box Fixing before Reverting
7. Don't Comment Out Failing Tests
8. Take Responsibility for All Breakages That Result from Your Changes

See *Essential Continuous Integration Practices* for more information.

Since it can be burdensome to check this manually, install yourself the tools to always see the current state of the Plone CI Server:

- For (Ubuntu?) Linux there is BuildNotify.
- For Mac there is CCMenu.
- For windows there is CCTray.
- For Firefox there is CruiseControl Monitor and many other jenkins specific plugins.

These tools were built to parse a specific file that CruiseControl another CI tool generated. Jenkins generates this file too. You want to configure your notifier of choice with this url: `http://jenkins.plone.org/cc.xml`

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
```

Alternatively, 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

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! Now 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.rst`.

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 separator 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 /develop/styleguide/index?
- 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 4.2.x, 4.3.x and master, which is the implied 5.0. This may change faster than this documentation, so check the branch dropdown on GitHub.

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!

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.rst` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone

The comprehensive resource for Plone documentation is http://docs.plone.org/. The following sections among others are included:

- User Manual
- Installing Plone
- Theme Reference
- Developer Manual

The documentation is hosted on GitHub and information how to contribute can be found at Contributing to the documentation.

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.rst` 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.rst` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

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 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. If the change you are proposing is not in the scope of a PLIP, a GitHub pull-request or issue is the right format. The key point here is that each change needs documentation so the change can be tracked and understood.

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 implementers 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 implementer 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 one of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the GitHub issue page for Products.CMFPlone for issues tagged with "03 type: feature (plip)". For reference to old PLIPs see the deprecated Trac tracker. 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 Framework Team mailing list.

A PLIP is just a GitHub issue on Products.CMFPlone with a special template and a specific tag. To get started, open a new issue by using the issuetemplate web service. You have to login with your GitHub account to use the issuetemplate service. Fill in all applicable fields. After submitting, select the tag "03 type: feature (plip)" for the issues.

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.

After your PLIP is written, solicit feedback on your idea on the Plone Community Forum. 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.

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 implementer.

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 issue itself so please keep your eyes and inbox open for changes.

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

- 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 *Reviewing PLIPs* 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

- Follow our style guides. For convenience and better code quality use Python, JavaScript and other code linting plugins in your editor.
- 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. The PLIP number is your PLIPs issue number. This file will define the branches you're working with in your PLIP along with other buildout configuration. 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-widget-frobbing

[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.

Working on a PLIP

To work on a PLIP, you bootstrap buildout and then invoke buildout with your PLIP config:

```
$ virtualenv .
$ ./bin/pip install -U zc.buildout setuptools pip
$ ./bin/buildout -c plips/plip-1234-widget-frobbing.cfg
```

If you are using a `local.cfg` to extend your plip file with some changes that you do not want to commit accidentally, be aware that you need to override some settings from `plipbase.cfg` to avoid some files being created in the `plips` directory or in the directory above the buildout directory. Like this:

```
[buildout]
extends = plips/plip-1234-widget-frobbing.cfg
develop-eggs-directory = ./develop-eggs
bin-directory = ./bin
parts-directory = ./parts
sources-dir = ./src
installed = .installed.cfg
```

```
[instance]
var = ./var
```

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

Once you have finished, please update your PLIP issue 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 *Reviewing PLIPs*.

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).

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 556, in resolve
    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.buildout-1.5.1-py2.6.egg), Requirement.parse('zc.buildout==1.5.2'))
```

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. Go 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 (:file:`.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

The Plone release process

Release process for Plone packages

To keep the Plone software stack maintainable, the Python egg release process must be automated to high degree. This happens by enforcing Python packaging best practices and then making automated releases using the zest.releaser tool. This is extended with Plone coredev specific features by *plone.releaser <https://github.com/plone/plone.releaser>*.

- Anyone with necessary PyPi permissions must be able to make a new release by running the `fullrelease` command

... which includes ...

- All releases must be hosted on PyPi
- All versions must be tagged at version control
- Each package must have README.rst with links to the version control repository and issue tracker
- CHANGES.rst (docs/HISTORY.rst respectively .txt in some packages) must be always up-to-date and must contain list of functional changes which may affect package users.
- CHANGES.rst must contain release dates
- README.rst and CHANGES.rst must be visible on PyPI
- Released eggs must contain generated gettext .mo files, but these files must not be committed to the repository. The .mo files can be created with the `zest.pocompile` add-on, which should be installed together with `zest.releaser`.
- `.gitignore` and `MANIFEST.in` must reflect the files going to egg (must include page template, po files)

More information:

- High quality automated package releases for Python with zest.releaser.

Special packages

The Plone Release Team releases the core Plone packages. Several others also have the rights to release individual packages on https://pypi.python.org. If you have those rights on your account, you should feel free to make releases.

Some packages need special care or should be done only by specific people as they know what they are doing. These are:

`Products.CMFPlone`, `Plone`, and `plone.app.upgrade`: Please leave these to the release manager, Eric Steele.

`plone.app.locales`: Please leave this to the i18n team lead, Vincent Fretin.

Plone core release process checklist

1. Check Jenkins Status

Check latest Plone coredev job on jenkins.plone.org, it should be green, if it is not, fix the problem first.

2. Check out buildout.coredev

```
git clone git@github.com:plone/buildout.coredev.git
cd buildout.coredev
git checkout 5.0
python bootstrap.py
bin/buildout -c release.cfg
```

Note that `release.cfg` installs `plone.releaser` which hooks into `zest.releaser` and adds a `bin/manage` script.

3. Check Packages for Updates

Check all packages for updates, add to/remove from checkouts.cfg accordingly.

This script may help:

```
bin/manage report --interactive
```

This step should not be needed, because we do the check for every single commit, but people may still have forgotten to add a package to the `checkouts.cfg` file.

4. Check packages individually

   Use the `bin/fullrelease` script from the core development buildout. This includes extra checks that we have added in `plone.releaser`. It guides you through all the next steps.

   1. Check changelog

      Check if CHANGES.rst is up-to-date, all changes since the last release should be included, a Fixes and/or New header should be included, with the relevant changes under it. Upgrade notes are best placed here as well. Compare `git log HEAD...<LAST_RELEASE_TAG>` with `CHANGES.rst`. Or from zest.releaser: `lasttaglog <optional tag if not latest>`.

   2. Run pyroma
   3. Run check-manifest
   4. Check package "best practices" (README.rst, CHANGES.rst, src directory)
   - Check if version in setup.py is correct and follows our versioning best practice
   5. Make a release (zest.releaser: `bin/fullrelease`)
   6. Remove packages from auto-checkout section in `checkouts.cfg` and update `versions.cfg`.

5. Make sure `plone.app.upgrade` contains an upgrade step for the future Plone release.
6. Update CMFPlone version in `profiles/default/metadata.xml`
7. Write an email to the translation team, asking them to do a plone.app.locales release.
8. Ask Rok to make a plone.app.widgets release (TODO!)
9. Create a pending release (directory) on dist.plone.org
1. Copy all core packages there.
2. Possibly make an alpha/beta release of CMFPlone.
3. Copy the versions.cfg file from coredev there.
10. Write an email to the Plone developers list announcing a pending release.
11. Inform the QA team about a new pending release.
12. Update plone.app.locales version.

13. Make final release on dist.plone.org (remove "-pending")
14. Update the "-latest" link on dist.plone.org
15. Create a unified changelog

```
bin/manage changelog
```

16. Create new release on launchpad (https://launchpad.net/plone/)
17. Create release page on http://plone.org/products/plone/releases
18. Send links to installers list (plone-installers@lists.sourceforge.net <mailto:plone-installers@lists.sourceforge.net>)
19. Wait for installers to be uploaded to Launchpad, link on plone.org release page
20. Mark release page as "final" (launchpad?)
20. Update PloneSoftwareCenter pointer to the newest release, so that it's linked from the homepage
21. Send out announcement to plone-announce
22. Update #plone topic

Managing Issues and Pull Requests

Introduction

We are using the Github issue trackers of the repositories at github.com/plone/* Here our issues for all Plone core repositories are handled.

Pull Requests are issue-like at Github. They share the set of labels and milestones.

New issues should be created in Products.CMFPlone. If the submitter is sure the issue belongs to a specific package, the issue can be created there too.

Labels explained

Main Labels

We have a well defined set of labels for all Plone core packages and some related packages. We tried to find a label system that is not too complex but covers several use cases.

01 type: bug Something that were supposed to work does not work.

02 type :regression Something that used to be available or worked in a former version, is not available any more or does not work anymore.

03 type: feature (plip) Really new stuff or major change that needs to go through the Plone Improvement Proposal process.

04 type: enhancement Makes existing features, code or documentation better. Needs to be backward compatible.

05 type: question A general question or request for comments to be answered or discussed.

11 prio: blocker Need to be solved immediatly.

12 prio: high High Priority, needs a solution.

13 prio: normal Normal priority.

14 prio: low Low priority.

21 status: confirmed Issue or pull request was read and got confirmed.

22 status: in-progress Somebody works on that issue or pull request.

23 status: testing Pull request specific: test or similar is running, waiting for results or if failed for a solution. Note: Jenkins pull request testing does report the state back to Github.

24 status: ready Pull request specific: Tests are passing, to be merged as soon as possible. Look for a reason in the comments why it was not merged already.

25 status: deferred Work on issue or pull request is scheduled for future. Look for a reason in the comments why it was deferred.

31: needs: help Anybody please help with this issue. Some issues or pull request need somebody with different skills to get solved. Others are new and unresolved or orphaned. Some times its an good entry point for newbies. Look if the level was set to get a first impression.

32: needs: review This issue needs a review, mostly pull requests. Often its good if more than one person looks after it and comment the opinion.

33: needs: docs Documentation is needed. This label is used for general documentation, but also to indicate missing change log entries. Look for a reason in the comments why exactly.

34: needs: tests This issue or pull request implies missing tests.

35: needs: rebase Pull request specific: needs a git rebase. This means the master or version-series branch diverged from the branch to merge. Github can no merge automatically into the requested main-line branch. The submitter is expected rebase the branch.

36: needs: cla Pull request specific: Submitter has not signed the Plone contributor lienses agreement. For legal reasons its not possible to merge.

41 lvl: easy Beginner skills needed. Perfect to make your hands dirty and start contributing to Plone.

42 lvl: moderate Fair plone insight needed. If you develop with Plone for some time and know the common bits and pieces, thats kind is for you.

43 lvl: complex For wizards. If you know Plone in depth please help with this kind.

Colors codes are used as shown in the image.

Special Labels

One can define arbitary labels using the scheme: "99 tag: short custom text"

We have a bunch of custom tags already around, and this makes sense. They get their own namespace and can be labled free after the prefix

Anatomy of a Label

number-prefix We need this to have a sanely sorted list of the issues at Github. The widgets are sorting them alphabetically. Otherwise selecting and viewing them is a large headache.

group-prefix It makes things more clear to prefix, similar to a namespace

color while the main types should be easily to differenciate, the other groups are each one color, eventually using a gradient

open/closed This is handled by Github already and does not need some extra label!

Issue vs. Pull Request Dont make a difference here, Github does it already. A PR is usally a 01-04.

State vs. Status For the non-native-english folks: http://english.stackexchange.com/questions/12958/status-vs-state

How to get this on all issue trackers

There is already a script plone.github that takes care of it. Also migration from old labels to new labels happens automatically. For new repositories the script just need to be re-run. Github-API FTW!

Working with Git and GitHub

The Plone Git workflow & branching model

Our repository on GitHub has the following layout:

- **feature branches**: all development for new features must be done in dedicated branches, normally one branch per feature,
- **master branch**: when features get completed they are merged into the master branch; bugfixes are commited directly on the master branch,
- **tags**: whenever we create a new release we tag the repository so we can later re-trace our steps, re-release versions, etc.

Git Basics

Some introductory definitions and concepts, if you are already familiar enough with Git, head to next section: *General guidelines*.

Mental working model

With Git (as well as all modern DVCS), distributing changes to others is a two steps process (contrary to traditional VCS like `svn`).

This way what on svn is a single `svn ci` in Git is two commands: `git commit` and `git push`.

This may seem to be a drawback, but instead it's a feature.

You are working locally until you decide to push your changes.

Not a single commit anymore, but a series of them, meaning that all those fears, concerns, doubts are taken away!

You can freely fix/change/remove/rework/update/... your commits afterwards.

Just push your changes whenever you are sure they are what you, and others, expect them to be.

Concepts

In Git there are:

commits A patch made out of changes (additions, removals) on files tracked by Git.

branches Series of commits that have a name.

tags A name attached to a single commit.

`HEAD` A pointer that always tells you where you are (extremely useful when doing some operations).

The index A temporal staging storage with changes on files that are pending to be added to a commit. If your Git output is colored, green filenames are those in the index.

Working tree Your current modified files. This is the only place where you can loose your changes. If your Git output is colored, red filenames are those in the working tree.

Stash Temporal storage for changes, again, extremely useful in some scenarios, see further below for examples.

Branches

Another great feature of DVCS is cheap branching, i.e. branching in Git is effortless and really useful. As it's no longer too much effort to branch, there is no need to always work on the master branch. A developer can branch easily for each fix/feature.

Branches allow you to tinker with your changes while keeping the master branch clean.

Not only that, it also allows you to keep modifying your changes until you and your peers are fine with them.

Further documentation: Introduction to branching.

Commands

Some of the most useful/common commands (note that most of them have switches that enhance/*completely twist* their functionality):

Just append `--help` on all of them to get their full definitions and options, i.e. `git add --help`.

clone Download a repository from a given remote URL.

add Add the given files to the index.

> **Note:** **pro tip**: once a file is add via `git add` your changes will never be lost! As long as you don't remove the `.git` folder, even if you remove the file you just added, the changes you made before doing `git add` are still there ready to be recovered at any time!

status Get an overview of the repository status.

If there are files on the index, or files not tracked by Git, or the status of your local branch with regards to the remote, etc.

diff See the current changes made to the files already tracked by Git.

> **Note:** Fear not, if you used `git add SOME_FILE` and then `git diff` doesn't output anything you haven't lost your changes!
>
> Just try `git diff --cached`. Now you know how to see the working tree changes (`git diff`) and index changes (`git diff --cached`).

commit Create/record changes to the repository (locally only, nothing is sent over the wire).

push Send your changes, either commits or a complete new branch, to the configured remote repository.

show Display the given commit(s) details.

log Shows the repository history. Sorted by date (last commit at the top), and like all other commands, extremely versatile with all its switches.

See further below for an example of a powerful combination of switches.

branch Create a branch.

fetch Download changes from the remote repository.

Without changing the current `HEAD` (see rebase and pull commands).

pull Fetch and integrate changes from remote repository.

Internally that means to do a `git fetch` plus either `git merge` or `git rebase`.

Note: Used careless most probably adds extra superfluous commits. See further down.

---

merge Join two, or more, branches together.

rebase Forward-port your current local commits (or branch) to be based on top of another commit.

An image is worth 1000 words: http://git-scm.com/docs/git-rebase

checkout Change to the given branch or get the given file to its latest committed version.

Note: If Git is criticized for being complex, this command is one of the main sources of complains.

You can compare it with `svn switch` if you happen to know it.

Fear not though, two main use cases are: change branches and reset a file to its last committed version. Still, the syntax for both cases is really simple.

---

cherry-pick Apply a commit(s) to the current working branch.

stash Use a temporal storage to save/restore current changes still not meant to be used on a commit.

Note: Seems a bit not so useful on a first look, but it is indeed.

Think about this scenario: you are working on your branch coding away. All of the sudden you notice a small fix that should be done directly on master. Thanks to `git stash` you can save your changes quickly and safely, move to master branch, do the quick fix, commit and push it, move back to your branch and `git stash pop` to recover your changes and continue hacking away.

---

reflog When things go bad you will **love** this command.

It effectively shows you a histogram of what happened on the repository, allowing you to rollback you repository to a previous stage.

Extremely useful once a bad interactive rebase has happened.

General guidelines

Pulling code

Let's compare this two histories:

```
*   3333333 (HEAD, master) Merge branch 'feature-branch' into master
|\
| * 2222222 (feature-branch) Last changes on feature-branch
| *   1111111 Merge branch 'master' into feature-branch
| |\
| * | 0000000 More changes on feature branch
```

```
| * |   fffffff Merge branch 'master' into feature-branch
| |\ \
* | | | eeeeeee master keeps rocking
| |_|/
|/| |
* | | ddddddd master goes and goes
| |/
|/|
* | ccccccc master evolves
| * bbbbbbb First commit on feature-branch
|/
* aaaaaaa commit on master  # this is where feature-branch was created
```

With:

```
* 3333333 (HEAD, master) Merge branch 'feature-branch' into master
|\
| * 2222221 (feature-branch) Last changes on feature-branch
| * 0000001 More changes on feature branch
| * bbbbbb1 First commit on feature-branch
|/
* eeeeeee master keeps rocking
* ddddddd master goes and goes
* ccccccc master evolves
* aaaaaaa commit on master
```

What do we see above? Actually and contrary to what it seems, exactly the same **result** (as how the files and its content look like on commit `333333`).

The second version is far more easy to understand what happened and removes two superfluous commits (the two partial merges with master (`fffffff` and `1111111`).

This happens if you have not properly configured `git pull`. By default it does a `merge` meaning that an extra commit is always added, tangling the history and making this more complex when looking back for what happened there.

How to solve it?

ALWAYS do a **`git pull --rebase`** when fetching new code, configure Git to do always so with:

```
git config branch.autosetuprebase always # add the --global switch to make it default␣
↪everywhere
```

This way you do not introduce new extra commits and the Git history is kept as simple as possible.

This is especially important when trying to understand why some changes were made, or who did actually change that line, etc.

A couple of further explanations: http://stevenharman.net/git-pull-with-automatic-rebase

http://www.slideshare.net/michalczyzcs3b/git-merge-vs-rebase-miksturait-4

Just search for `git merge vs rebase`, you will find plenty of literature.

Reviewing your changes

After hacking for some minutes/hours/days you are finished and about to commit your changes, great!

BUT, please, do so with **`git add --patch`**.

The `--patch` (also `-p`) switch allows you to select which hunks you want to add on a commit.

This is not only great to split changes into different commits, but is also the time when you actually **review** your code before anyone else sees it.

This is the time when you spot typos, pep8 errors, misaligned code, lack of docstrings in methods, that a permission is not defined on Generic Setup, that an upgrade should be needed...

Remember that the first code review is the one you do on your own. Some inspiration/better phrasing: http://ada.mbecker.cc/2012/11/22/be-your-own-code-review/

And please, do remember the gold metric about reviewing code: http://www.osnews.com/story/19266/WTFs_m

One commit does one thing

Repeat with me: *One commit does one thing*. Period.

When someone else needs to review your code, most probably she will give up or just skim over your code if there are too many (unrelated) changes.

Reviewing commits with +20 files doing all sorts of changes on them (maybe even unrelated) is no fun and adds complexity and cognitive load.

Something that should mostly be a verification of a checklist like:

- the browser view is registered on ZCML?
- is there an interface for that form?
- the pt and py are there?
- ...

Turns instead into a list of questions:

- why is this interface renamed here if it has nothing to do with this adapter?
- all this removal of deprecated code while adding new features just mixes the diff, am I missing something?
- *others*

If you can not express what has been changed within 50 characters (suggested length of a commit message subject), or you say it like "it does XXX and YYY", you most probably need to split that commit into, at least, two or more commits.

That doesn't mean that a +20 files or +100 lines of code changes are bad per se, you may be doing a simple refactoring across lots of files, that's fine and good actually.

As long as a commit is just and only about a specific purpose, and not a mixed selection of the following:

- refactoring code
- moving things around
- fixing some bugs while at it
- adding some docs
- a new cool feature
- fixing typos on documentation
- pep8 fixes

It is absolutely fine to refactor.

And this is actually to help both your present self and your +5 years from now that will have to refactor that code of yours, and maybe is struggling to understand what was going on there.

Following this advice will:

- keep things simple where there's no gain in adding complexity
- make your changes easy to be reviewed
- make later on lookups on those changes easy to follow

Making commits

For commit messages see: git_commit_message_style_guide.

Adding references to issues

Always add the full URL to the issue/pull request you are fixing/referring to.

Maybe within the Git repository it makes sense, but as soon as you are outside of it, it will not.

Take into account mr.roboto automatic commits to buildout.coredev for example, if your commit message goes like *Fix for #33*, which issue/pull request is that fixing? The one in buildout.coredev itself? On another issue tracker? Somewhere else?

It would be far better if the commit goes instead like:

```
Brief description

Further explanation.

Fixes: https://github.com/plone/plone.app.discussion/issue/999
```

Bad examples

Some bad examples of commit messages: https://github.com/plone/plone.app.content/commit/0f3a6c65b2018e0ecc65d0ad1581e345f17e531b

Commit messages goes like *"Make note about how this interface is now for BBB only"*.

Question: if it's BBB only, where is the new place to look for that interface now?

The problem is that, in this case Martin, wrote that in 2009, so most probably once a refactor of that package is done later on 2015, Martin is no longer around, and if he was, most probably he would not remember something from +6 years ago.

Ask yourself a question: if someone comes to you asking details about a random commit done by you +5 years ago, what will you reply?

Try that, get one project that you worked 5 years ago, get a random commit and:

See if, just by reading the commit message, you are given enough information of what changes have been made, when comparing the commit message and the actual code. Does the commit message match the code changed?

Before pushing commits

Code is reviewed, spread into nice isolated commits, descriptive enough commit messages are written, so, *what's left?*

A final overview of what you are about to push!

To do so, you can get an idea with the following Git alias (to be added on your `~/.gitconfig`):

```
[alias]
    fulllog = log --graph --decorate --pretty=oneline --abbrev-commit --all
```

Now run **`git fulllog`** on your Git repository, you will see a nice graph showing you the current situation.

Maybe it makes you realize that commits need to be reordered, commit messages could get some improvements, that you forgot to add a reference to an issue, etc.

Pull requests

Some specific tips and best practices for pull requests.

Always rebase

Always rebase on top of the branch you want your changes to be merged before sending a pull request, and as your pull request is still pending to be merged and the master branch evolves, keep rebasing it.

To do so:

```
git checkout <your branch>
git rebase master # or the branch you are targeting to integrate your changes to
# done!
# or if there are conflicts,
# fix them and follow instructions from git itself
```

The principle here is: if you do merges with master, you are actually spreading your pull request into more commits, and at the end making it more difficult to track what was changed.

On top of that, the commit history is more complex to follow.

See the history example above: *General guidelines*.

Unfortunately the flat view from GitHub prevents us from seeing that, which is a shame.

One line one commit

On a series of commits make sure the same code line is not changed twice, the worst thing you can do to the one reviewing your changes, is to make him/her spend time reviewing some code changes that one the next commit are changed again to do something else.

It will not only make your commits smaller, but it will also make it easy to do atomic commits.

No cleanup commits please

On the context of a pull request

Ask yourself: What relation does a cleanup commit, say pep8 fixes or other code analysis fixes, have with your pull request?

Couldn't that pep8 fixes commit or small refactoring go straight into master branch?

Or even if you send a pull request for it, chances are that it will be merged right away. As long as it is a cleanup commit, there's not much to argue with it.

The same goes with commits that improve or actually fix previous commits (within the same pull request). A series of commits like this:

```
* 11ba28c Last fix, finally
* 11ba28c Fix tests, again
* 11ba28c Fix tests
* 11ba28c Do something fancy
* 11ba28c Failing test, we are doing TDD right?
```

Only tells you that the author did not take care at all about the one who will review it, and specially about the person that in +5 years will try to understand that test. Specially because now the test is not only spread between 4 commits, but most probably during those 5 years it has already been refactored, so maybe a **`git blame`** will report that within that test method, there are +5 related current commits to check, not nice right?

Squashing commits

To fix the previous example, run the following command:

```
git rebase ---interactive <base> # which mostly is usually master
```

This allows you to rewrite the story of your branch. See a more elaborate description with examples.

Note: Be careful on not to run that on master itself! Please take your time to really understand it.

It's a really powerful tool, and as Stan Lee says, it comes with great responsibility.

To actually make it easier you can do commits like this:

```
git commit --fixup HASH
```

Where `HASH` is the commit hash you want the changes you are about to commit be merged with.

This way, when running **`git rebase --interactive`**, Git will already reorder the commits as you already want.

No side changes

That's an extension to the previous point.

Keeping pull requests simple and to the point, without changes not related to the pull request itself, will make your changes easier to understand and easier to follow.

Again this applies: http://www.osnews.com/story/19266/WTFs_m

Recipes

Assorted list of tips and tricks.

Change branches with uncommitted changes

Situation: you are working on a pull request and while working on it founds that some cleanups are needed, how to proceed forward?

Solution: `git stash` or `git commit --amend -m"TMP"`.

The basic idea here is: store your current changes safely (either on a Git stash commit or directly on a commit on the branch, whichever you prefer), move to the canonical branch (`master` usually), do the fixes/cleanups/refactorings there, commit those changes, rebase your branch on top of the changes you made, hack away.

Command line version:

```
git stash # or git commit --amend -m"TMP"
git checkout master # or whatever happens to be the canonical branch name (i.e. 5.0␣
↪on buildout.coredev)
# do the cleanups && push them
git checkout your-branch # get back to your branch
git rebase master # again the canonical branch where you made the changes
git stash pop # or git reset HEAD^ if you did a git commit --amend -m"TMP"
# if needed, fix the conflicts, with patience and practise that's a piece of cake␣
↪once you are used to
```

Git visual applications

Not everyone is a fan of the command line, for them there is a list of GUI clients on the official Git website:

http://git-scm.com/downloads/guis

Enhanced Git prompt

Do one (or more) of the following:

- http://clalance.blogspot.com/2011/10/git-bash-prompts-and-tab-completion.html
- http://en.newinstance.it/2010/05/23/git-autocompletion-and-enhanced-bash-prompt/
- http://gitready.com/advanced/2009/02/05/bash-auto-completion.html

Git dotfiles

Plone developers have dotfiles similar to these: https://github.com/plone/plone.dotfiles.

Learn more

What's here is just the tip of the iceberg, there's plenty of Git knowledge on the web.

A few good further resources are listed here (contributions welcome):

- official online Git book: Pro Git
- PyCon 2015 talk: Advanced Git by David Baumgold

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 `5.0`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 5.0
```

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 **`git cherry-pick`**.

CHAPTER 3

Additional Material

These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained

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

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

About the Plone Contributor Agreement

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

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

Questions & Answers

What does the contributor's agreement cover?

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

What rights will I continue to have for my contributions?

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

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

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

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

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

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

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

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

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

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

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

Why would anyone want a non-GPL Plone?

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

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

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source

provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Essential Continuous Integration Practices

The CI system at jenkins.plone.org is a shared resource for Plone core developers to notify them of regressions in Plone core code. Build breakages are a normal and expected part of the development process. Our aim is to find errors and eliminate them as quickly as possible, without expecting perfection and zero errors. Though, there are some essential rules that needs to be followed in order to achieve a stable build.

1) Don't Check In on a Broken Build

Do not make things more complicated for the developer who is responsible for breaking the build. If the build breaks, the developer has to identify the cause of the breakage as soon as possible and should fix it.

If we adopt this strategy, we will always be in the best position to find out what caused the breakage and fix it immediately. If one of the developers has made a check-in and broken the build as a result, we have the best chance of fixing the build if we have a clear look at the problem. Checking in further changes and triggering new builds will just lead to more problems.

If the build is broken over a longer period of time (more than a couple of hours) you should either notify the developer who is responsible for the breakage, fix the problem yourself, or just revert the commit in order to be able to continue to work.

Note: There is one exception to this rule. Sometimes there are changes or tests that depend on changes in other packages. If this is the case, there is no way around breaking a single build for a certain period of time. In this case run the all tests locally with all the changes and commit them within a time frame of 10 minutes.

---

2) Always Run All Commit Tests Locally before Committing

Following this practice ensures the build stays green and other developers can continue to work without breaking the first rule.

There might be changes that have been checked in before your last update from the version control that might lead to a build failure in Jenkins in combination with your changes. Therefore it is essential that you check out (`git pull`) and run the tests again before you push your changes to GitHub.

Furthermore, a common source of errors on check-in is to forget to add some files to the repository. If you follow this rule and your local build passes, you can be sure that this is because someone else checked in in the meantime, or because you forgot to add a new class or configuration file that you have been working on into the version control system.

3) Wait for Commit Tests to Pass before Moving On

Always monitor the build's progress and fix the problem right away if it fails. You have a far better chance of fixing the build, if you just introduced a regression than later. Also another developer might have committed in the meantime (by breaking rule 1) making things more complicated for your.

4) Never Go Home on a Broken Build

Taking into account the first rule of CI ("Don't check in on a broken build"), breaking the build essentially stops all other developers from working on it. Therefore going home on a broken build (or even on a build that has not finished yet) is not acceptable, because it will prevent all the other developers to stop working on the build or fixing the errors that you introduced.

5) Always Be Prepared to Revert to the Previous Revision

In order for the other developers to be able to work on the build, you should always be prepared to revert to the previous (passing) revision.

6) Time-Box Fixing before Reverting

When the build breaks on check-in, try to fix it for ten minutes. If, after ten minutes, you aren't finished with the solution, revert to the previous version from your version control system. This way you will allow other developers to continue to work.

7) Don't Comment Out Failing Tests

Once you begin to enforce the previous rule, the result is often that developers start commenting out failing tests in order to get the build passing again as quick as possible. While this impulse is understandable, it is wrong. The tests have been passing for a while and then start to fail. This means that we either caused a regression, made assumptions that are no longer valid, or the application has changed the functionality being tested for a valid reason.

You should always either fix the code (if a regression has been found), modify the test (if one of the assumptions has changed), or delete it (if the functionality under test no longer exists).

8) Take Responsibility for All Breakages That Result from Your Changes

If you commit a change and all the tests you wrote pass, but others break, the build is still broken. This also applies to tests that fail in `buildout.coredev` and don't belong directly to the package you worked on. Usually this means that you have introduced a regression bug into the application. It is your responsibility — because you made the change — to fix all tests that are not passing as a result of your changes.

There are some tests in Plone that fail randomly, we are always working on fixing those. If you think you hit such a test, try to fix it (better) or re-run the Jenkins job to see if it passes again. In any case the developer who made the commit is responsible to make it pass.

Further Reading

Those rules were taken from the excellent book "Continuous Delivery" by Jez Humble and David Farley (Addison Wesley), and have been adopted and rewritten for the Plone community. If you want to learn more about Continuous Integration and Continuous Delivery, I'd recommend that you buy this book.

Mr. Roboto

GitHub push

When a push happens on GitHub, `mr.roboto` is triggered so starts to analyze the push.

- If it's on `buildout-coredev` it starts the job of the branch that has been pushed. In this case we send to plone-cvs the commit to keep track of the commits on that list.
- If it's on a package that's on the `sources.cfg` of a `buildout-coredev` it starts the coredev jobs that are linked to that package and a kgs job with that package. This kgs job is a snapshot of the last working version of the `buildout.coredev` with the newest version of the package that is involved on the push. This jobs are really fast as we only test the package applied to the kgs plone/python version `coredev` buildout.
- If it's on a plip specification it runs the job that is configured Through The Web on `mr.roboto` interface. (http://jenkins.plone.org/roboto/plips)

Job finishes

When jenkins finish a job it does a callback to `mr.roboto` in order to :

- If it comes from a `coredev` job, all the `coredev` jobs related to that push are finished writes a comment on the GitHub commit with all the information (once and with all the information so no more empty mails from gh notification system)
- If it comes from a kgs job and all the kgs jobs are finished, (that may take max 10 min) and some has failed we send a mail to testbot mailing list saying that a commit failed on kgs job. We also send a mail to plone-cvs with the information to keep track of all the commits.
- If it comes from a kgs job and all the kgs jobs are finished, and all are working we send a mail to plone-cvs with the information to keep track of all the commits.

For all kgs jobs jenkins sends a mail to the author with the results when is finished.

All the notifications has a url like : http://jenkins.plone.org/roboto/get_info?push=9a183de85b3f48abb363fa8286928a10

on this url there is the commit, who, the diff, the files and the result for each jenkins job.

So ...

- plone-testbot mailing list is receiving only when a test fails on kgs environment and may take max 10 min from the push.
- plone-cvs always has the commit there with the diff and the information and may take 10 min to get there after the push.
- author receives the results of tests failing against kgs on 10 min

---

Note: In case of integration errors with other packages that may fail because of the push kgs will not be aware of that, so it's important that at the end (and after the 50' that takes the `coredev` jobs you also check the latest version of `coredev` with your push)

Mr. Developer

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

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

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

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

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

```
> bin/develop st
```

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

```
> bin/develop purge
```

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

Reviewing PLIPs

Expectations

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

Setting up the environment

Follow the instructions on setup-development-environment. You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that PLIP:

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

Functionality Review

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

General

- Does the PLIP actually do what the implementers proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?

- Are there any performance considerations? Has the implementer addressed them if so?

Bugs

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

Usability Issues

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

Documentation Issues

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

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

Code Review

Python

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

JavaScript

- Does the JavaScript meet our set of JavaScript standards? See our section about JavaScript and the JavaScript styleguide.
- Does the JavaScript work in all currently supported browsers? Is it performant?

ME/TAL

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

Our coding style guides are located at the Plone Style Guide section.