Launchpad Documentation

Release dev

The Launchpad Developers

May 14, 2015

Contents

Welcome to the Launchpad developer documentation. This documentation is for people who want to hack on Launchpad.

CHAPTER 1

Overview

1.1 README for Launchpad

Launchpad is an open source suite of tools that help people and teams to work together on software projects. Unlike many open source projects, Launchpad isn't something you install and run yourself (although you are welcome to do so), instead, contributors help make <https://launchpad.net> better.

Launchpad is a project of Canonical <http://www.canonical.com> and has received many contributions from many wonderful people <https://dev.launchpad.net/Contributions>.

If you want help using Launchpad, then please visit our help wiki at:

https://help.launchpad.net

If you'd like to contribute to Launchpad, have a look at:

https://dev.launchpad.net

Alternatively, have a poke around in the code, which you probably already know how to get if you are reading this file.

1.1.1 Getting started

There's a full guide for getting up-and-running with a development Launchpad environment at <https://dev.launchpad.net/Getting>. When you are ready to submit a patch, please consult <https://dev.launchpad.net/PatchSubmission>.

Our bug tracker is at <https://bugs.launchpad.net/launchpad/> and you can get the source code any time by doing:

```
$ bzr branch lp:launchpad
```

Navigating the tree

The Launchpad tree is big, messy and changing. Sorry about that. Don't panic though, it can sense fear. Keep a firm grip on *grep* and pay attention to these important top-level folders:

bin/, utilities/ Where you will find scripts intended for developers and admins. There's no rhyme or reason to what goes in bin/ and what goes in utilities/, so take a look in both. bin/ will be empty in a fresh checkout, the actual content lives in 'buildout-templates'.

configs/ Configuration files for various kinds of Launchpad instances. 'development' and 'testrunner' are of particular interest to developers.

cronscripts/ Scripts that are run on actual production instances of Launchpad as cronjobs.

daemons/ Entry points for various daemons that form part of Launchpad

database/ Our database schema, our sample data, and some other stuff that causes fear.

doc/ General system-wide documentation. You can also find documentation on <https://dev.launchpad.net>, in docstrings and in doctests.

lib/ Where the vast majority of the code lives, along with our templates, tests and the bits of our documentation that are written as doctests. 'lp' and 'canonical' are the two most interesting packages. Note that 'canonical' is deprecated in favour of 'lp'. To learn more about how the 'lp' package is laid out, take a look at its docstring.

Makefile Ahh, bliss. The Makefile has all sorts of goodies. If you spend any length of time hacking on Launchpad, you'll use it often. The most important targets are 'make clean', 'make compile', 'make schema', 'make run' and 'make run_all'.

scripts/ Scripts that are run on actual production instances of Launchpad, generally triggered by some automatic process.

You can spend years hacking on Launchpad full-time and not know what all of the files in the top-level directory are for. However, here's a guide to some of the ones that come up from time to time.

buildout-templates/ Templates that are generated into actual files, normally bin/ scripts, when buildout is run. If you want to change the behaviour of bin/test, look here.

bzrplugins/ Bazaar plugins used in running Launchpad.

sourcecode/ A directory into which we symlink branches of some of Launchpad's dependencies. Don't ask.

zcml/ Various configuration files for the Zope services. Angels fear to tread.

1.2 Launchpad Strategy

We want to make Ubuntu the world's best operating system. To do this, we need to give Canonical an edge in productivity over and above other Linux vendors and, just as importantly, help make the development of open source software faster, more efficient and more innovative than its proprietary rivals.

Launchpad does this by helping software developers share their work and plans, not just within a project but also **between** projects.

1.2.1 Introduction

This document tries to answer two big questions:

1. *why* are we making Launchpad?
2. *who* is Launchpad for?

This is not our strategy for the year or the scope of Launchpad development for the next six months. Rather, it is our answer to these fundamental questions.

When you are finished reading this document, you should know what problems we want to solve, what we hope to gain from solving these problems and how we know if Launchpad is doing well.

Audience

This document is for everyone who cares about improving Launchpad. Primarily, we've written it for Launchpad's stakeholders within Canonical and for the developers of Launchpad, whether they are Canonical employees or not.

1.2.2 Why are we making Launchpad?

The world we live in

Open source software is bigger than you think. It is much more than simply writing the code. Code has to be packaged, integrated and delivered to users who can then give feedback and file bugs. Distributions made up of tens of thousands of different software packages need to be released to meet a deadline. Translations must be made into hundreds of different languages and accumulated from a variety of sources. Everywhere bugs need to be tracked, fixed and checked. Plans must be made and kept. Distributions have to be made to work on a wide variety of hardware platforms with varying degrees of openness.

Those who make open source software and wish to profit commercially also face unique challenges. Contributors are scattered across the world, making coordination, communication and alignment just that little bit more difficult. Many contributors are volunteers, and so decisions must often be made by consensus, deadlines enforced without the leverage of an employment contract and quality maintained without formal training. Users of open source software use a widely heterogeneous stack of software and hardware, thus increasing the burden of compatibility work. All of these things make open source software development more difficult, thus increasing the need for tools to aid collaboration.

The Ubuntu community, together with Canonical, are dedicated to making the very best open source operating system possible, one that far excels any proprietary operating system. To do this, we need to ensure that the process of making Ubuntu is as effective as possible. Moreover, we need to make the process of making open source software as effective as possible, and then make it easy, quick and desirable to get that software into Ubuntu.

Secondarily, Canonical's main business is the provision of premium services built around Ubuntu. Many of these services are based on proprietary software, which Canonical must be able to make more quickly and at less cost than any rival.

The word "effective" covers a multitude of concepts. Here we mean doing the *right* work with the highest possible *quality* as *quickly* and with as little *waste* as possible.

Business goals

Launchpad exists to give Canonical a competitive advantage over other operating system vendors and service providers, both proprietary and open source.

To gain an advantage over *open source* operating system vendors, Canonical is relying on Launchpad to:

- increase Canonical's effectiveness in making software
- grow and accelerate contributions to Ubuntu

To gain an advantage over proprietary operating system vendors, Canonical needs Launchpad to do both of the above and:

- improve and accelerate open source software development in general beyond that of proprietary software so that the software in Ubuntu is better than the software in any rival proprietary operating system

The value flow of Launchpad can be summed up in this diagram:

1.2.3 Who is Launchpad for?

Launchpad is aimed at many different groups of users. They can be roughly described as follows:

Software developers These are people who make or contribute to free and open source software. They are made up of both paid professionals and volunteers working in their spare time. They vary widely in expertise and patience. Any given software developer might be working on both open source software and proprietary software.

Expert users of software The sort of people who file bugs, try new releases, run the bleeding edge snapshot, are interested in following development plans, who help other people on mailing lists. Note that software developers are frequently but not always expert users of software.

End users of software People who download and install software and then use it. These people have little understanding about what software actually is or how it is made. They use it, sometimes without noticing, sometimes enjoying it, sometimes hating it.

Translators A special class of software developer who is normally a native speaker of a language other than English. They contribute to open source software projects not by submitting code, but by translating strings to new languages.

Managers These are managers in the broad sense of people who are responsible for the completion of a task and so need to know what many other people are doing towards that goal. This includes release managers, project leads and traditional corporate project managers. It does not necessarily mean people who are employed as managers.

User needs

The people who use Launchpad, in whatever role, share one broad goal: "make great software and get it to its users". To do this, they need:

- tools to facilitate collaboration on their proprietary and open source software projects
- a place to host and publish their open source software projects
- as little overhead as possible in maintaining these projects
- more contributors to their projects
- to be able to easily contribute to existing software projects

Some of our users have particular needs:

- managers need to be able to quickly get an overview of activity and progress for their teams and their projects
- expert users of software need to be able to give high quality feedback to the software developers

Further, we believe that providing tools for cross-project collaboration, we can benefit our users by:

- giving them feedback from groups of their own users that they couldn't reach before
- reducing the time and effort required to publish software to actual end users
- pointing them to knowledge and fixes from other projects in their network
- helping OS-driven improvements reach them code faster, and their improvements reach the OS faster

Conflicts between business goals & user needs

Canonical is primarily interested in open source software that runs on Linux or lives within the Linux ecosystem. Thus, even though Launchpad could be an excellent, general platform for Windows, OS X, iOS and Android based software, our main area of focus is software that is aimed to run natively on Linux.

Canonical is much more interested in quality assurance and release management than many open source and even proprietary projects.

1.2.4 References

- What is Launchpad?
- Launchpad Values
- Feature checklist

1.3 What is Launchpad?

Launchpad is a complete system for gathering changes from different types of sources and collaboratively organizing them into packaged software for the end user, delivered as part of an operating system that can be downloaded or that comes already installed on purchased hardware.

If you start by thinking of Launchpad as a traditional software "forge" – a web service that provides bug tracking, code hosting and other related services – then you are not too far off understanding what Launchpad is. However, Launchpad has many distinctive traits that combine to put it into an entirely different category of software.

But at its core, it is best to think of Launchpad as a service that meshes together two important networks:

1. Networks of people making software
2. The network of dependencies between software

But first, a story.

1.3.1 The Story of a Bug

Arnold writes software for a living, and he runs Ubuntu on his desktop. He wishes he could contribute to open source, but he doesn't have much spare time, and when he gets home from his job the last thing he wants to do is program. However, the spirit of willingness is there.

One day, Arnold notices that Tomboy loses his formatting if he alt-tabs at the wrong time. Arnold knows that a well-filed bug report is a thing of beauty to most programmers, so he decides to spend a few moments to file a bug against Tomboy.

Rather than search the net to find where Tomboy tracks its bugs, Arnold uses Ubuntu's built-in bug filing mechanism. It asks him a bunch of questions, invites Arnold to write his bug report and then files the bug on Launchpad against the Tomboy package in Ubuntu.

Becca has been dabbling in Ubuntu contribution for a while, mostly by helping new users solve their problems or turn them into good bug reports. She notices Arnold's bug, sees that it's well written and thinks that it would be easy enough to test against trunk. She opens up the Tomboy source package in Ubuntu and sees that there is a known-good daily build of Tomboy's trunk hosted in a trusted user archive. Becca installs Tomboy from this archive and tests to see if the bug is still there in the latest version of the code. It is. Becca sees this, opens up the original bug report and clicks a button to forward the bug to the upstream bug tracker.

Carlos is one of the Tomboy developers. He sees the bug in the tracker, sees that it has been tested against trunk, realizes that it's an annoying bug that's easy to fix and decides to fix it. He does the fix, applies it to the Tomboy trunk and marks the bug as fixed.

At this point, both Arnold and Becca are notified that the bug is fixed in Tomboy trunk, and that they can try a version of Tomboy that has the fix by using the known-good daily build archive for Tomboy. They are warned that this is dangerous and may cause data loss, but they are also told how they can try the bug fix for free using a cloud-based

Ubuntu desktop. They both try the bug, see that it's fixed, and are happy, albeit a little impatient for the fix to be actually released and part of stock Ubuntu.

Meanwhile, *Dalia* is an Ubuntu developer who takes a special interest in desktop productivity applications like Tomboy. She checks on the Ubuntu source package for Tomboy from time to time. The last time she checked, she saw that quite a few bugs have been fixed in trunk but not yet released. Since she knows the Tomboy release manager from long hours of IRC chat, she contacts him and gently suggests that he do a release.

Edmund, the Tomboy release manager, takes Dalia's hint well and realizes that a release is way overdue. He makes a release of Tomboy following his normal procedure.

Launchpad detects that Tomboy has a new, official release and alerts interested distribution maintainers that the release has been made and now would be a good time to package a new version. Dalia packages up a new version, requests that an Ubuntu core developer sponsor the change and then waits for the new version to be uploaded. Dalia also uploads the fixed version to one of her personal archives so that others can easily get it without waiting for the next release of Ubuntu.

Fiona the Ubuntu core developer sees Dalia's patch in the sponsorship queue on Launchpad, notes that it's all good and then uploads the patch to the official Ubuntu archive. (Fiona might also choose to upload the patch to Debian).

Launchpad sees that this upload fixes a number of bugs, including the one originally filed by Arnold, and automatically includes those bugs in the list of bugs that will be fixed by the next release of Ubuntu.

Two months later, the next release of Ubuntu is actually released. Arnold upgrades on release day, and tries out Tomboy to see if his bug was really, actually fixed. It is, and all is right with the world.

1.3.2 Distinctive traits

Launchpad is different from other "forges" in a few important ways:

Cross-project collaboration

No project lives in isolation. Each project is part of an ecosystem of software. Projects must be able to interact with each other, share bugs, teams, goals and code with each other.

Launchpad takes every chance it gets to show the connections between projects and to bring the opportunities created by those connections to light.

By encompassing the entire process, all the way to operating system delivery, Launchpad can provide a unique service: enable each contributor to focus on the work they care about, while giving them an ambient awareness of how their work fits into a larger picture, and providing a path by which they can participate in other parts of that picture when they feel the need.

Front-end to open source

Launchpad aims to be a front-end to open source. Whether or not a project chooses to host on Launchpad, opportunistic developers can use Launchpad to navigate bugs, get code and send patches. Likewise, we aim to present a uniform interface to the projects we have.

Centralized service

Because Launchpad emphasises cross-project collaboration, and because Launchpad aims to be a front-end to all of open source, it necessarily has to be a centralized service rather than a product that users deploy on their own servers.

Networks of collaborators

Launchpad understands that much of the human interaction around open source is not primarily social, but rather collaborative: many people working together in different ways toward the same goals.

As such, Launchpad highlights actions and opportunities rather than conversations and status. It answers questions like, "what can I do for you?", "who could help me do this?", "who is waiting on me in order to get their thing done?", "can I rely on the advice offered by this person?" and so forth.

Distributions are projects too

Launchpad hosts Linux distributions in much the same way as it hosts projects, allowing for developers to feel at home when interacting with distributions.

Gated development

Sometimes, secrets are necessary. Launchpad understands that sometimes development needs to be done privately, and the results only later shared with the world. Security fixes, OEM development for new hardware, proprietary services with open source clients are all examples of these.

Hardware matters

Many software developers like to pretend that hardware does not really exist. When people distribute software as part of an operating system, they don't have the luxury of forgetting. Launchpad understands that developers often need to acknowledge and work around differences thrown up by hardware.

We don't care if you use Launchpad, sort of

Many other forges define their success by how many users they have. Although we love our users and welcome every new user, Launchpad does not judge its success by the number of users. If one project wishes to host its development on another platform, Launchpad acts as a front-end to that platform.

One project, many communities

Any given project can have many distinct communities interested in it. These communities have different interests and different motivations, but all work in the same project space so that they can easily benefit from each others' efforts.

1.3.3 Scope

Launchpad has many major components. These can be broken up into four major areas of functionality:

1. where work is done; developers interact with other developers
2. where plans are made and reviewed; expert users interact with expert users and developers
3. where projects engage with their communities; developers interact with end users and other developers, and vice-versa
4. major supporting features

Work

At the core of every software project is the actual code that makes up that project. Here "code" is a broad term that also includes the project's documentation, the translatable and translated strings that make up its user interface, the packaging and integration scripts required to get the software installed on end user's systems and so forth.

Launchpad is built to be able to take contributions from anybody, regardless of how involved they are in a project. For packages, translations and code proper we provide mechanisms to allow people to review changes from others and then merge them into the official parts of the project.

Launchpad pulls in changes that happen in the upstreams and downstreams of a project, whether those changes are patches to code, new translations or packaging updates. It makes contributors to a project aware of the work that's going on upstream and downstream and helps them take advantage of that work.

And, of course, all work is for nothing if it does not get to the people who might want to actually use its results. As such, project maintainers can publish released versions of their code, any contributor can publish Ubuntu packages to unofficial archives or even set up Launchpad to automatically build and publish packages of latest snapshots of code.

Plans

People who are interested in doing something great will need to coordinate their work, keep track of the defects in the things they have already done and describe the things that they aren't doing yet but wish they could.

Every software product in the world has bugs. For some projects, the rate of incoming bugs is fairly low, and each bug can expect to receive some attention from a core developer. For other projects, the rate of new bugs filed is so high that the core development team can never hope to keep up with it. Launchpad supports both kinds of projects.

If every software product has bugs, every software user has great ideas about how a product can be improved. Project maintainers need to get at these ideas, evaluate them, and develop them into workable concepts.

Often, a problem is so tricky that those concerned need to have a detailed, managed discussion about what exactly the problem is. At other times, the problem is easy enough to define, but there are so many solutions with difficult trade-offs or difficult implementations that it is better to talk about them and plan them out before proceeding with any of them. Launchpad acknowledges that this can happen on any project, and that becoming clear on a problem or becoming clear on the "best" solution can be helped a great deal using tools.

Crucially, all of these different types of "plans" – bugs, specifications, blueprints, ideas – can span more than one code base and more than one conceptual project. These plans need to be drafted, discussed, clarified and reviewed before work starts, monitored, evaluated and changed as work progresses, and then the results of that work need to be checked against the plan when the work is finished.

Community

Not everything that's done on a project is work toward a particular outcome, or plans for how to get there. Every project needs to have some things that are more general and stable.

Projects need to be able to present themselves to the world, confident in their identity and communicating exactly what they are about. Project maintainers need to be able to announce important news, such as releases, license changes or new practices. Contributors need to get a sense of who is working on which parts of the project. Users need to be able to ask questions, get support and give feedback.

Contributors also need to share documentation about the project and how the project runs. They need to be able to discuss general topics about the project.

Launchpad supports all of these things, and also makes it clear how any project fits into the broader ecosystem of projects. It shows which projects are upstreams or downstreams, which projects are affiliated with other projects, which projects share contributors with other projects and so forth.

Supporting features

Launchpad has many major areas of functionality that are best considered as "supporting features": APIs, migration services, privacy, the mail UI, synchronizing with external systems.

1.3.4 New World

When Launchpad is really doing all of these things and doing them well, the world of open source software will be significantly changed.

Patches will no longer lie decaying in someone else's bug tracker, waiting to be noticed. Instead, they will all be synced into a central code review system and queued for review and approval.

Instead of a distribution tracking one set of bugs and upstream projects tracking their own set of sometimes duplicated bugs, both upstream and downstream developers can seamlessly accesses both sets of bugs.

1.3.5 Glossary

Upstream A software project itself, as opposed to the packaged version of a software project that is included in a distribution. Note, can also be used as a relative term, e.g. "Debian is upstream of Ubuntu".

Downstream The opposite of an upstream. Can be used to refer either to a packaged version of a specific software project, or the entire distribution where that package occurs.

1.3.6 References

- Launchpad Strategy
- Launchpad Values
- Feature checklist

1.4 Launchpad Values

Whenever we are thinking about what Launchpad should be, or how we should implement a change, or whether something is a good idea or not, we have recourse to three distinct sets of guidelines.

The first is Launchpad Strategy, which reminds us why we are making Launchpad and helps us answer questions such as "How does this help Launchpad meet its goals?". The second is What is Launchpad?, which helps us answer questions like "Is this in scope?". Together, they sort out the 'matter' of Launchpad.

The third is this document, the Launchpad Values. It tries to address the 'manner' of Launchpad. It, perhaps, will not answer specific questions for you. Rather, it will rule out certain options and decisions before they are even considered.

Like the Launchpad Strategy, this document is living: it should be changed and improved as we learn more about how to make Launchpad. It is also aspirational: not all of Launchpad lives up to these values.

1.4.1 Better tools help

Launchpad is fundamentally based on the principle that improving the tools that we use to make things will help us make better things than we could before and indeed make better things more cheaply than we could before.

Launchpad should be that "better tool" and be always aiming to smooth and accelerate the process of making software.

1.4.2 Invisible. If not, fun.

Launchpad is a tool to help busy people get important stuff done. It should stay out of the way where possible. Bugs, OOPSes, downtime and slowness all draw attention to Launchpad and away from the interesting problems that our users are trying to solve.

Where it is not possible to stay out of the way, Launchpad should be fun to use. We make next actions obvious and draw attention to users' achievements.

Example

When a branch is merged into the trunk of a project, that's generally the end of its story. Launchpad quietly and silently detects that it has been merged and marks the branch and any merge proposals as such. The branch then no longer appears on default listings.

1.4.3 Reveal opportunities

One of the grand things about open source software is that it is open to contributions from total strangers.

Launchpad makes those contributions possible by removing as many barriers as possible to contribution, and highlights areas where contributions would be especially welcome.

Example

When you click the Translations tab of a project that's translatable, you see a list of all of the languages you speak, together with a progress bar telling you how much translation work is available. You can click a language and start translating right away.

1.4.4 Not our data

The data in Launchpad does not really belong to us, we are merely its stewards. We make sure that users can get their data easily and that they can change it as they see fit.

Example

You can access almost all of the data in Launchpad through the RESTful APIs.

1.4.5 Not just their data

The data people store in Launchpad doesn't just belong to them, though. It also belongs to the wider open source community. The data needs to be used to link between other projects, and to allow Launchpad to act as a front-end of open source.

Example

Someone who is not actually a maintainer of a project might register that project on Launchpad so they can import code or synchronize bugs or so forth. If the project's actual maintainer comes along and wishes to take it down, we will *not* do so.

1.4.6 Cross-project consistency

If you know how to contribute to one project on Launchpad, you ought to be able to quickly and painlessly contribute to any other project on Launchpad. Or, if you can't, it won't be Launchpad's fault.

Example

You can get the trunk branch for any project with 'bzr branch lp:project', or the source for any Ubuntu package using 'bzr branch lp:ubuntu/project'.

1.4.7 All of open source

Any open source project ought to be able to host itself on Launchpad. We do not enforce workflows, rather we allow people to fit their project's existing workflow into Launchpad.

However, when we can, we encourage people toward best practices. After all, we want to make open source software better.

Example

Launchpad separates whether or not a reviewer approves of code from whether or not a branch is approved to land.

1.4.8 All users are potential contributors

One of the glories of open source is that any user is a potential contributor. Launchpad guides new users toward the ways where they can begin to contribute.

1.4.9 Close the loop

Something magical happens when a feature or a workflow reaches all the way back to where it began. The feature begins to re-inforce itself, and make things possible that weren't possible before.

Another way of thinking about this is that the value is in the output, and Launchpad is always concerned with the output of all of its features.

Example

These are examples of where we aren't there yet.

Being able to attach patches to bugs is great, but it's not good enough until developers can easily *find* those patches. Finding the patches is only good enough when you can merge, comment and reject those patches.

Likewise, importing translations from an upstream is great, but it becomes much, much better when those translations can be improved in Launchpad and then sent back to the upstream.

1.4.10 References

- Launchpad Strategy
- What is Launchpad?

CHAPTER 2

Technical

2.1 Launchpad Buildout

Launchpad uses the buildout (or "zc.buildout") build system. Buildout's biggest strength is managing Python packages. That is also our focus for it.

We have at least two other ways of managing dependencies. Apt manages our Python language installation, as well as many of our non-Python system-level dependencies, such as PostgreSQL. The sourcecode directory is our other way of managing dependencies. It is supposed to only contain dependencies that are not standard Python packages. bzr plugins and Javascript libraries are existing examples.

If you are not interested in our *Motivations* or in an *Introduction to zc.buildout*, all developers will at least want to read the very brief sections about the *Set Up* and the *Everyday Usage*.

Developers who manage source dependencies probably should read the general information about *Managing Dependencies and Scripts*, but will also find detailed instructions to *Add a Package*, to *Upgrade a Package*, to *Add a Script*, to *Add a File Modified By Buildout*, and to *Work with Unreleased or Forked Packages*.

2.1.1 Motivations

These motivations are labeled as "[INTERNAL]" or "[EXTERNAL]" to indicate whether it is pertinent for internal dependencies, which we on the Launchpad team create and release ourselves; or external dependencies, which other parties, in and out of Canonical, create and release.

- We want more careful specification of our dependencies across branches. [INTERNAL] [EXTERNAL]

  This is a real concern pertinent both for our "trunks" (devel, stable, db-devel, db-stable) and for our development boxes. For instance, before incorporating buildout, in our trunks, when we want to update a dependency, we needed to make sure that *all* the current Launchpad trunks work with the dependency initially; then submit a new Launchpad branch that uses the change dependency. A mistake can even potentially break one or both of the db-* trunks, since PQM only tests against one branch (usually devel), and sourcecode changes affect all branches at once. For simplicity, speed, and safety, we want to be able to submit a single branch that incorporates the source dependencies and the associated changes at once.

  This is also true, if less critical and easier to work around, on developer boxes. Without care, changes to sourcecode when working on dependencies will affect all a developer's branches, polluting test results with false negatives or false positives.

- We want to default to using released versions of our software dependencies. [EXTERNAL]

  A significant number of projects do not always have a pristine trunk, and many also spend extra effort on polish, bug fixes, and compatibility before a release. If we do not desperately need a new feature on trunk, using a

release is generally regarded as a safer, better practice. Our earlier usage of bzr branches of the development trunks does not encourage this practice.

- We want to be encouraged to make the effort to interact with upstream projects to have our patches integrated. [EXTERNAL]

  Interacting and negotiating with upstream is undeniably more time-consuming than our previous practice of maintaining local bzr branches with our patches, especially short-term. But our previous use of bzr branches is not good open-source community behaviour–an ironic characteristic for a project like Launchpad. It also can cause problems down the road, for instance, if the patch becomes stale and we want to migrate to new releases.

- We want to be protected from changes and differences in our operating system. [INTERNAL] [EXTERNAL]

  This is a concern both over time and across different Launchpad environments.

  First, our operating system, Ubuntu, is driven by many needs and goals. Launchpad is among them, but generally Launchpad serves Ubuntu, not the reverse. For instance, Jaunty dropped Launchpad's Python version. The Ubuntu developers had good reason–Python 2.4 has not been supported by the Python developers for some time–but it caused a significant inconvenience to the Launchpad team. Managing our dependencies, particularly the Python library dependencies, can help alleviate these problems.

  Second, Launchpad developers run a significantly different version of the operating system than that run in production. Maintaining our Python library dependencies ourselves can also help alleviate these concerns.

- We want to be able to easily use standard packages of our primary programming language, Python. [EXTERNAL]

  Our Python library dependencies are distributed for many operating systems– Windows, Mac, and other flavors of Linux–in a unified location and format: PyPI, using distutils. Using Python library dependencies in their standard distributions makes it easier for us to reuse code.

- We want to be encouraged to release Python packages of our open-source code. [INTERNAL]

  We are beginning to realize our aspirations of abstracting and releasing some of our code. Much of that code is in Python. Consuming standard Python packages encourages us to follow good practice in releasing our own Python packages. Dogfooding can help keep us honest, and good official releases can help us support and encourage a community of users.

Meanwhile, we want to be able to keep certain aspects of the legacy story.

- We need deployment to not need network access.
- We need to be able to use custom releases of our Python dependencies, if absolutely necessary.
- We would like to be able to follow security releases in our operating system.

We will be able to support the first two of these aspects entirely.

As to the third, we will continue to follow operating system security releases for most or all of the dependencies that are not Python packages–a very similar situation to the one before Buildout integration.

2.1.2 Introduction to zc.buildout

Buildout is a relatively simple system that increases in complexity as it is extended via "recipes". It works on top of distutils and setuptools. It uses declarative ini-style files to direct its work. The Buildout site points to a variety of documents describing and documenting its use.

For Launchpad, Buildout is hidden behind a Makefile as of this writing. If the Makefile is removed, developers will typically run `bootstrap.py` in the branch, and then run `bin/buildout`. After that, the `bin` directory will have the commands to start, stop, test, and debug the software. If you are interested in example direct usage of Buildout, you may want to read the "Hacking" document in the Launchpad wiki that describes the usage patterns in `lazr.*` packages.

Launchpad's Buildout usage is roughly of medium complexity. It is more complex than that needed by a package with few dependencies and simple usage (see lazr.uri, for instance), but less complex than that of other large applications that use the buildout system. More complexity can come by building more non-Python tools and by having multiple configuration variations, for instance.

The documentation below will focus on using Launchpad's buildout. See the links given above for a more thorough general review.

2.1.3 Set Up

If you use the `rocketfuel-get` script, run that, and you will be done.

If you don't, you have just a bit more initial set up. I'll assume you maintain a pattern similar to what the `rocketfuel-*` scripts use: you have a local, pristine branch of trunk from which you make your other branches. You manually update the trunk and rsync sourcecode when necessary. When you make a branch, you use `utilities/link-external-sourcecode`.

Developers that take this approach should do the following, where `trunk` is the trunk branch from which you make local branches.

```
bzr co lp:lp-source-dependencies download-cache
```

Then run `make` in the trunk.

See the *Everyday Usage: Manual* section for further instructions on how to work without the `rocketfuel-*` scripts.

2.1.4 Everyday Usage

`rocketfuel` Scripts

If you typically use `rocketfuel-get`, and you don't change source dependencies, you should not have any further changes, except that `bin/test` has replaced `test.py`. `rocketfuel-branch` and `link-external-dependencies` will Do the Right Thing.

Manual

If you don't use the rocketfuel scripts, you will still use `link-external-dependencies` as before. When a buildout complains that it cannot find a version of a dependency, do the following, from within the branch:

```
bzr up download-cache
```

After this, retry your make (or run `bin/buildout` from the branch).

That's it for everyday usage.

2.1.5 Managing Dependencies and Scripts

What if you need to change or add dependencies or scripts? As you might expect, you need to know a bit more about what's going on, although we can still keep this at a fairly high level.

First let's talk a little about the anatomy of what we have set up. To be clear, much of this is based on our own decisions of what to do. If you see something problematic, bring it up with the Foundations team. Maybe together we can come up with another approach that meets our needs better.

If you saw the top-level Launchpad directory before we started using Buildout, you might notice seven new items in the checkout.

bootstrap.py This is the standard bootstrapping file provided by the Buildout distribution. As of this writing, the Makefile uses this file, and you do not have to modify it or call it directly.

Without a Makefile, the first step of developing a source tree that uses Buildout is to run this file in the top level directory with the Python executable that you wish to use for the source tree. It creates a `bin` directory, if it does not already exist, and puts a `buildout` executable there. The next step is to run `bin/buildout`, which, unless you supply a `-c` option to specify a configuration file, will look for a buildout.cfg file by default to discover what to do.

Again, as of this writing, the Makefile uses this file, and it does not need to be modified, so you need not concern yourself with it further at this time.

ez_setup.py This is another standard file from another project. In this case, it is the file provided by the setuptools project to install setuptools. It is used by the `setup.py` file, described below. It does not need to be modified or called directly.

setup.py This is the file that uses `distutils`, extended by `setuptools`, to specify direct dependencies, scripts, and other elements of the local source tree.

Buildout uses it, but the reverse is not true: `setup.py` does not know about Buildout. This means that packages that use Buildout for development do not have to require it when they are being installed in other software as a dependency.

Describing this file in full is well beyond the scope of this document. We will give recipes for modifying it for certain tasks below. For more information beyond these recipes, see the setuptools and distutils documentation.

buildout.cfg This is the default configuration file that `bin/buildout` will look to for instructions.

Describing it in full is well beyond the scope of this document. However, we will give an overview here.

Configuration files for Buildout are comprised of sections with key-value pairs.

The key-value pairs are separated with new lines, when the subsequent line is not indented. The key and value are each separated with an equals sign.

```
foo = bar
baz = bing
      bah
      boo
sha = zam
```

That example shows three keys, 'foo', 'baz', and 'sha'. The 'baz' key has a string with two new lines (which might be interpreted one several ways, as defined for that key).

The `[buildout]` section is the starting point for Buildout to determine what to do. It looks for an `extends` key to find any additional files to merge in; we use this for `versions.cfg`, discussed below.

In addition to general configuration and initialization such as this, it looks in the `develop` key to find source trees to develop as part of the buildout. In the standard Launchpad configuration, we develop only Launchpad itself (the current directory, or '.'). This means that the local `setup.py` will be run. If you want to develop Launchpad while you develop another dependency, you can link another source tree in, and specify an additional `develop` directory in another line:

```
[buildout]
develop = .
          lazr_uri_branch
```

See *Developing a Dependent Library In Parallel* for more on this.

The other basic key in the `[buildout]` section that we'll highlight here is `parts`. This key identifies the other sections in buildout.cfg that will be processed. A section that is not identified in the `[buildout]` sections `parts` key will usually be ignored (unless chosen for another role by another key elsewhere).

Sections other than `[buildout]` that are specified as parts always must specify a `recipe`: an identifier that determines what code should process that section. You'll see a variety of recipes in Launchpad's buildout.cfg, including `z3c.recipe.filetemplate`, `zc.recipe.egg`, and others.

versions.cfg As mentioned above, `buildout.cfg` extends `versions.cfg` by specifying it in the `extends` key of the `[buildout]` section. Versions.cfg specifies the precise versions of the dependencies we use. This means that we can have several versions of a dependency available locally, but we only build the precise one we specify. We give an example of its use below. To read about the mechanism used, see the zc.buildout documentation of the `versions` option in the `[buildout]` section.

eggs The `eggs` directory holds the eggs built from the downloaded distributions. Unless you set it up differently yourself, this directory is shared by all your branches. This directory is local to your system–we do not manage it in a branch. One reason for this is that eggs are often platform-specific.

download-cache The `download-cache` directory is a set of downloaded distributions–that is, exact copies of the items that would typically be obtained from the Python Package Index ("PyPI"), or another download source. We manage the download cache as a shared resource across all of our developers with a bzr branch in a Launchpad project called `lp-source-dependencies`.

When we run buildout, Buildout reads a special key and value in the `[buildout]` section: `install-from-cache = true`. This means that, by default, Buildout will *not* use network access to find packages, but *only* look in the download cache. This has many advantages.

- First, it helps us keep our deployment boxes from needing network access out to PyPI and other download sites.
- Second, it makes the buildout much faster, because it does not have to look out on the net for every dependency.
- Third, it makes the buildout more repeatable, because we are more insulated from outages at download sites such as PyPI, and poor release management.
- Fourth, it makes our deployments more auditable, because we can tell exactly what we are deploying.
- Fifth, it gives us a single obvious place to put custom package distributions, as we'll discuss below.

The downside is that adding and upgrading packages takes a small additional step, as we'll see below.

buildout-templates The last additional item in the checkout is the `buildout-templates` directory. This is used to hold templates that are used by the section in buildout.cfg that uses the `z3c.recipe.filetemplate` recipe. This can be used for many things, but we are using it as an alternate way for producing scripts when the zc.recipe.egg approach is insufficient.

In addition to these seven listings, after you have run the Makefile (or `bin/buildout`), you will see an additional listing:

bin The `bin` directory has already been discussed many times. After running the bootstrap.py, it holds the `buildout` script which can be used to process Buildout configuration files. In Launchpad's case, after running the buildout, it also holds many executables, including scripts to test Launchpad; to run it; to run Python or IPython with Launchpad's sourcetree and dependencies available; to run harness or iharness (with IPython) with the sourcetree, dependencies, and database connections; or to perform several other tasks. For now, the Makefile provides aliases for many of these.

Now that you have an introduction to the pertinent files and directories, we'll move on to trying to perform tasks in the buildout. We'll discuss adding a dependency, upgrading a dependency, adding a script, adding an arbitrary file, and working with unreleased packages.

Add a Package

Let's suppose that we want to add the "lazr.foo" package as a dependency.

1. Add the new package to the `setup.py` file in the `install_requires` list.

   Generally, our policy is to only set minimum version numbers in this file, or none at all. It doesn't really matter for an application like Launchpad, but it a good rule for library packages, so we follow it for consistency. Therefore, we might simply add `'lazr.foo'` to install_requires, or `'lazr.foo >= 1.1'` if we know that we are depending on features introduced in version 1.1 of lazr.foo.

2. [OPTIONAL] If you know it, add the desired version number to versions.cfg now.

   For instance, if you know you want lazr.foo 1.1.2, add this line to the `[versions]` section of `versions.cfg`:

```
lazr.foo = 1.1.2
```

3. [OPTIONAL] Add the desired distribution of lazr.foo 1.1.2 to the `download-cache/dist` directory.

4. Run the following command (or your variation):

```
./bin/buildout -v buildout:install-from-cache=false | tee out.txt | grep 'Picked'
```

   The first part (`./bin/buildout -v buildout:install-from-cache=false`) will run buildout, allowing it to download source packages from the Internet to `download-cache/dist`. The second part (`tee out.txt`) will dump the full output to the `out.txt` file in case you need to debug a problem. The last part (`grep 'Picked'`) will filter the output so that only additional packages–dependencies of your dependency–will be listed. You need to see if it means that you have dependencies, some of which may be indirect dependencies. We'll see how to do this with an example. Here's an imaginary output:

```
Picked: ipython = 0.9.1
Picked: lazr.foom = 1.4
Picked: zope.bar = 3.6.1
Picked: z3c.shazam = 2.0.1
```

   In our example, this means that these packages have been added to your `download-cache/dist` directory. You now need to add those versions to the `versions.cfg` file:

```
ipython = 0.9.1
lazr.foom = 1.4
zope.bar = 3.6.1
z3c.shazam = 2.0.1
```

   Note that the output might include at least one, and possibly more, spurious "Picked:" listings. ipython, in particular, has shown up in the past incorrectly–that is, when you try to add the file to the download-cache/dist directory, you'll discover that it is already there; and you'll see that versions.cfg already specifies the version. As long as the test passes (see step 5, below), you can ignore this.

5. Test.

   Note that you can tell `ec2 test` to include all uncommitted distributions from the local download-cache in its tests with the `--include-download-cache-changes` flag (or `-c`). If you do this, you cannot use the ec2 test feature to submit on test success. Also, if you have uncommitted distributions and you do *not* explicitly tell ec2 test to include or ignore the uncommitted distributions, it will refuse to start an instance.

6. Check old versions in the download-cache. If you are sure that they are not in use any more, *anywhere*, then remove them to save checkout space. More explicitly, check with the LOSAs to see if they are in use in production and send an email to launchpad-dev@lists.launchpad.net before deleting anything if you are unsure. A rule of thumb is that it's worth starting this investigation if the replacement has already been in use by the

Launchpad tree for more than a month. You can approximate this information by using `bzr log` on the newer (replacement) download-cache/dist file for the particular package.

7. Now you need to share your package changes with the rest of the team. You must do this before submitting your Launchpad branch to PQM or else your branch will not build properly anywhere else, including buildbot. Commit the changes (`cd download-cache`, bzr add the needed files, `bzr up`, `bzr commit -m 'Add lazr.foom 1.1.2 and depdendencies to the download cache'`) to the shared download cache when you are sure it is what you want.

Never modify a package in the download-cache. A change in code must mean a change in version number, or else very bad inconsistencies and confusion across build environments will happen.

Upgrade a Package

Sometimes you need to upgrade a dependency. This may require additional dependency additions or upgrades.

If you already know what version you want, the simplest thing to try is to modify versions.cfg to specify the new version and run steps 4, 5, and 6 of the *Add a Package* instructions.

If you don't know what version you want, but just want to see what happens when you upgrade to the most recent revision, you can clear out the versions of the packages for upgrade and give it a try (that is, run steps 4, 5, and 6 of the *Add a Package* instructions). Note that, when not given an explicit version number, our buildout is set to prefer final releases over alpha and beta releases. If you want to temporarily override this behaviour, include `buildout:prefer-final=false` as another argument to `bin/buildout`.

Add a Script

We often need scripts that are run in a certain environment defined by Python dependencies, and sometimes even different Python executables. Several of the scripts we have are specified using the setuptools-based spelling that the `zc.recipe.egg` recipe supports.

For the common case, in `setup.py`, add a string in the `console_scripts` list of the `entry_points` argument. Here's an example string:

```
'run = lp.scripts.runlaunchpad:start_launchpad'
```

This will create a script named `run` in the `bin` directory that calls the `start_launchpad` function in the `lp.scripts.runlaunchpad` module.

See the zc.recipe.egg documentation for more information on how to add scripts using this method.

Add a File Modified By Buildout

Sometimes we need more control for the way our scripts are generated, or we need other files processed during a buildout. Writing a custom zc.buildout recipe is one way, but well out of the scope of this document. Read the zc.buildout documentation for direction.

A much easier, and more limited approach is to use z3c.recipe.filetemplate to build the file. The recipe uses the `buildout-templates` directory, which is a mirror of the Launchpad source tree. The recipe searches the tree for files ending in '.in', performs variable substitution on them, and then be copies them into the Launchpad source tree.

To add a file using the recipe, simply create mirrors of the source tree directories that you need under `buildout-templates/`, and create a .in file template at the desired location. Take a look at `buildout-templates/bin/` for examples of what is possible.

Work with Unreleased or Forked Packages

Sometimes you need to work with unreleased or forked packages. FeedValidator, for instance, makes nightly zip releases but other than that only offers svn access. Similarly, we may require a patched or unreleased version of a package for some purpose. Hopefully, these situations will be rare, but they do occur.

While other answers are available for Buildout, our solution is to use the download-cache. Basically, make a custom source distribution with a unique suffix in the name, and use it (and its version name) for the normal process of adding or updating a package, as described above. Because the custom package is in the download-cache, it will be found and used.

Here's an example of making a custom distribution of FeedValidator.

FeedValidator is a Subversion project. We check it out:

```
svn co http://feedvalidator.googlecode.com/svn/trunk/feedvalidator/src feedvalidator
```

Next, we `cd feedvalidator`, and, using a Python that has setuptools installed, we run the following command:

```
python setup.py egg_info -r -bDEV sdist
```

For this example, imagine that the current revision of the repository is 1049. Because setuptools has built-in Subversion support, the command above will create a tar.gz in the `dist` directory named `feedvalidator-0.0.0DEV-r1049.tar.gz`. The -r option specifies that the subversion revision should be part of the package name. The -bDEV option specifies that the 'DEV' suffix should be added to the version number.

We could then put the tar.gz file in Launchpad's `download-cache/dist` directory, specify `feedvalidator = 0.0.0DEV-r1049` in the `versions.cfg` file, and proceed with the usual steps to update or add a new package.

If you use a bzr branch, you might use the `-d` option instead of the `-r` option when you create the distribution. This will add the date instead of the revision:

```
python setup.py egg_info -d -bDEV sdist
```

For instance, this might produce a distribution for the `lazr.restful` project with a name like this: `lazr.restful-0.9.1DEV-20090512.tar.gz`.

See the setuptools documentation for more information about the egg_info command.

It is also possible that setup.py does not support the egg_info command and it is sufficient to just run the sdist command. It adds the current revision of the bzr branch to the version number:

```
python setup.py sdist
```

For instance, this might produce a distribution for the `testtools` project with a name like this: `testtools-0.9.12-r228.tar.gz`.

Developing a Dependent Library In Parallel

Sometimes you need to iterate on change to a library used by Launchpad that is managed by buildout as an egg. You could just edit what it is in the `eggs` directory, but it is harder to produce a patch while doing this. You could instead grab a branch of the libarary and produce an egg everytime you make a change and make buildout use the new egg, but this is slow.

buildout defaults to only using the current directory as code that will be used without creating a distribution. We can arrange for it to use other paths as well, so we can use a checkout of any code we like, with changes being picked up instantly without us having to create a distribution.

To do this add the extra paths to the `develop` key in the `[buildout]` section of `buildout.cfg`:

```
[buildout]
develop = .
          path/to/branch
```

and re-run `make`.

Now any changes you make in that path will be picked up, and you are free to make the changes you need and test them in the Launchpad environment.

Once you are finished you can produce a distribution as above for inclusion in to Launchpad, as well as sending your patch upstream. At that point you are free to revert the configuration to only develop Launchpad. You should not submit your branch with this change in the configuration as it will not work for others.

Be aware that you may have to change the version for the package in `versions.cfg` if there is a difference between the version in the `eggs` directory and the one in the `setup.py` that you pointed to in the `develop` key.

One thing to be wary of is that setuptools treats "develop eggs" created by this process with the same precedence as system packages. That means that if the version in the `setup.py` at the path that you put in the `develop` key is the same as the version installed system wide, setuptools may pick the wrong one. If that happens then increase the version in setup.py and setuptools will take it.

2.1.6 Possible Future Goals

- No longer use system site-packages.
- No longer use make.
- Get rid of the sourcecode directory.

2.2 Possibly out-of-date

2.2.1 Security Policy in Launchpad

Zope 3 is a security-aware framework that makes it possible to develop complex applications with security policies that closely resemble the reality that the system is trying to model.

This document is about security policy in Launchpad.

Defining Permissions in Launchpad

NOTE: A new permission should only be defined if absolutely necessary, and it should be considered thoroughly in a code review.

Occassionally, you'll find yourself in a situation where the existing permissions in Launchpad aren't enough for what you want. For example, as I was writing this document I needed a permission I could attach to things to provide policy for who can view a thing. That is, I wanted a permission called launchpad.View. A new permission (see the NOTE above) is defined in Launchpad in the file lib/canonical/launchpad/permissions.zcml. So, to define the permission launchpad.View, we'd add a line like this to that file:

<permission id="launchpad.View" title="Viewing something" access_level="read" />

Defining Authorization Policies for Permissions

Once you've defined a permission, you'll probably want to define some logic somewhere to express the authorization policy for that permission on a certain interface.

In Launchpad, an authorization policy is expressed through a security adapter. To define a security adapter for a given permission on an interface:

1. Define the adapter in lib/canonical/launchpad/security.py. Here's a simple example of an adapter that authorizes only an object owner for the launchpad.Edit permission on objects that implement the IHasOwner interface:

```
class EditByOwner(AuthorizationBase):
    permission = 'launchpad.Edit'
    usedfor = IHasOwner

    def checkAuthenticated(self, person):
        """Authorize the object owner."""
        if person.id == self.obj.owner.id:
            return True
```

Read the IAuthorization interface to ensure that you've defined the adapter appropriately.

2. Declare the permission on a given interface in a zcml file. So, for the above adapter, here's how it's hooked up to IProduct, where IProduct is protected with the launchpad.Edit permission:

```
<class
    class="lp.registry.model.product.Product">
    <allow
      interface="lp.registry.interfaces.product.IProductPublic"/>
    <require
      permission="launchpad.Edit"
      interface="lp.registry.interfaces.product.IProductEditRestricted"/>
    <require
      permission="launchpad.Edit"
      set_attributes="commercial_subscription description"/>
</class>
```

In this example, the EditByOwner adapter's checkAuthenticated method will be called to determine if the currently authenticated user is authorized to access whatever is protected by launchpad.Edit on an IProduct.

2.2.2 Launchpad and Email

Quicker interactive testing

There is a script `process-one-mail.py` which reads a single mail message from a file (or stdin), processes it as if it had been received by Launchpad, and then prints out any mail generated in response. For quasi-interactive testing of email processing this may be your best bet.

Quickstart

Otherwise, you can configure Launchpad with an incoming mailbox and an outgoing mailer, in a way somewhat similar to what is used in production. This lets you catch mail sent other than in response to an incoming message.

Create the file override-includes/+mail-configure.zcml with contents similar to the following:

```
<configure
    xmlns="http://namespaces.zope.org/zope"
    xmlns:mail="http://namespaces.zope.org/mail"
    i18n_domain="zope">

    <!-- delete the username and password attributes if you don't use
        SMTP auth -->
    <mail:smtpMailer
        name="smtp"
        hostname="localhost"
        port="25"
        username="biggus"
        password="dickus"
        />

    <mail:stubMailer
        name="stub"
        from_addr="stuart@stuartbishop.net"
        to_addr="stuart@stuartbishop.net"
        mailer="smtp"
        rewrite="false"
        />

    <mail:testMailer name="test-mailer" />

    <mail:mboxMailer
        name="mbox"
        filename="/tmp/launchpad.mbox"
        overwrite="true"
        mailer="test-mailer"
        />

<!--

    Uncomment me if you want to get copies of emails in your normal inbox,
    via the stubMailer above.  However, tests will fail because they
    depend on using a directDelivery mailer.  See below.

    <mail:queuedDelivery
        mailer="stub"
        permission="zope.SendMail"
        queuePath="/var/tmp/launchpad_mailqueue" />

-->
<!--

    Uncomment me if you want to get test emails in a Unix mbox file, via
    the mboxMailer above.

    <mail:directDelivery
        mailer="mbox"
        permission="zope.SendMail"
        />
-->

</configure>
```

Options

Zope3 provides two defined mailers out of the box (*smtp* and *sendmail*) so most people won't actually need the *mail:smtpMailer* tag because the defaults will usually just work. However, several additional mailers are available for you to use, depending on what you're trying to do.

The *mail:stubMailer* can be used to forward all emails to your normal inbox via some other mailer. Think of it as a proxy mailer that can be used to specify explicit MAIL FROM and RCTP TO envelope addresses. The *rewrite* attribute of the *mail:stubMailer* specifies whether the RFC 2822 headers should also be rewritten. You and your spam filters might prefer this set to *true*.

The *mail:mboxMailer* stores messages in a Unix mbox file and then forwards the message on to another mailer. You can use this if you want a record on disk of all the messages sent, or if you'd rather not clutter up your inbox with all your Launchpad test email. The *overwrite* attribute says whether to truncate the mbox file when Launchpad starts up (i.e. opens the file once in 'w' mode before appending all new messages to the file).

The *mail:testMailer* is necessary for the Launchpad tests to work. You must use a *mail:directDelivery* mailer for the tests, otherwise you'll get lots of failures. Basically, the testMailer stores the messages in a list in memory.

For both *mail:mboxMailer* and *mail:stubMailer* the *mailer* attribute specifies the next mailer in the chain that the message will get sent to. Thus if *mailer* is set to *smtp*, you'll get the messages in your inbox, but if it's *test-mailer*, the unit tests will work.

Finally, these are all hooked up at the top with either a *mail:queuedDelivery* section or a *mail:directDelivery* tag. You must use a *mail:directDelivery* tag if you want the unit tests to work because otherwise, the in-memory list of the *mail:testMailer* won't be updated by the time the unit test checks it.

If you just want the unit tests to work normally, don't include a *mail:queuedDelivery* or a *mail:directDelivery* section at all. Launchpad will DTRT internally. However, if you want copies in an mbox file or in your inbox, set the *mailer* attribute to the appropriate mailer, chaining that to a *mail:testMailer* for the unit tests or a *mail:smtpMailer* for development.

API

Launchpad code should use the methods defined in lp.services.mail.sendmail to send emails (*simple_sendmail*, *sendmail* or possibly *raw_sendmail*)

Functional Tests

The functional test harness is configured to allow easy testing of emails. See *lp/services/mail/tests/test_stub.py* for example code.

Details

To send email from Zope3, you use an *IMailDelivery* Utility, which defines a single *send* method. There are two standard *IMailDelivery* implementations:

1. *QueuedDelivery* – email is delivered in a separate thread. We use this for production.
2. *DirectDelivery* – email is send synchronously during transaction commit. We use this for tests.

Both implementations will send no email if the transaction is aborted. Both implementations use events to notify anything that cares to subscribe if delivery succeeded or failed. Both implementations look up an *IMailer* utility by name to do the actual delivery, as specified in the *mailer* attribute of the *queuedDelivery* and *directDelivery* ZCML tags.

Zope3 provides two *IMailer* implementations out of the box:

1. *SMTPMailer* – sends email using SMTP
2. *SendmailMailer1 – Uses the 'sendmail* program to send email.

In addition to these two, there are three more *IMailer* implementations for use with Launchpad development (production instances will just use *SMTPMailer* or *SendmailMailer*):

3. *StubMailer* – rewrites the envelope headers and optionally the RFC 2822 To and From headers before handing on to a different *IMailer*.
4. *TestMailer* – stores the email in memory in a Python list object called *lp.services.mail.stub.test_email* for easy access by unit tests.
5. *MboxMailer* – stores the email in a Unix mbox file before optionally handing the message off to another *IMailer*.

Developers (and production and dogfood server, until we are confident messaging is working fine) should use the StubMailer, like in the quickstart example. This causes all emails to be redirected to the specified destination address to you can test to your hearts content without spamming other developers or innocent civilians. Or you can use the MboxMailer.

The functional test suite is already configured to use the TestMailer. However, if you use a StubMailer or MboxMailer and want the test suite to work, you must hook it up to a TestMailer explicitly. See *lp/services/mail/tests/test_stub.py* for an example showing how functional tests can check that notifications are being sent correctly.

CHAPTER 3

Other

3.1 Documents of historical interest

The following documents don't really represent current thinking or development practices, but are useful to have around to get a sense of where we've been and how we've progressed.

Contents:

3.1.1 Running Launchpad with Chameleon Template Engine

- Need to pull the following dependencies into `sourcecode`:
  - lp:sourcecodegen/trunk
  - lp:chameleon.core/trunk
  - lp:chameleon.zpt/trunk
  - lp:z3c.pt/trunk
  - lp:z3c.ptcompat/trunk
- Run launchpad with `PREFER_Z3C_PT=true make run` to enable `z3c.pt`. Omitting it or setting to `false` will disable `z3c.pt` and use `zope.pagetemplate` instead. Yes, it's that simple. This is possible thanks to `z3c.ptcompat`.

Other useful environment options for `z3c.pt`:

```
# in debug-mode, templates on disk are reloaded if they're modified
CHAMELEON_DEBUG (default: false)

# disable disk-cache to prevent the compiler from caching on disk
CHAMELEON_CACHE (default: true)

# if eager parsing is enabled, templates are parsed upon
# instantiation, rather than when first called upon; this mode is
# useful for verifying validity of templates across a project
CHAMELEON_EAGER (default: false)

# in strict mode, filled macro slots must exist in the macro that's
# being used.
CHAMELEON_STRICT (default: false)

# when validation is enabled, dynamically inserted content is
```

```
# validated against the XHTML standard
CHAMELEON_VALIDATE (default: false)
```

3.1.2 About Malone

The world has many excellent bug tracking tools already. It would not make sense to create another bugtracker unless the vision behind that software was substantially different to anything that had gone before it. This document outlines that vision, explaining what it is that I hope Malone will do for the open source community.

The Vision behind Malone

Malone is a unified bug tracker for the entire open source world. It is designed to allow the whole open source community to collaborate on software defect management, especially when a single piece of code is being used across many projects. Malone presents a single page which gathers together the combined wisdom and knowledge of the open source world regarding a specific software defect.

Upstream and Distributions

A unique feature of Malone is that it understands the structure of the open source community:

```
Software is developed by individuals or groups with a common interest in a
specific problem. We call this group "upstream". That software is
distributed in its pristine state ("tarballs", usually) and is usually
designed to be compiled and run on a variety of platforms.

However, most people who use that software will not get it directly from
upstream, build it and install it locally. They will install a package
that has already been prepared for the specific platform they are running
on. For example, on Gentoo, they will type "emerge foo". On Ubuntu, they
would type "apt-get install foo". And on RedHat they would install a
custom RPM. So the same software code is being repackaged many times, for
Gentoo, Ubuntu, RedHat, and many other platforms.
```

A natural consequence of this repackaging is that a bug in that software might be detected and/or fixed by a variety of different people, without upstream being aware of either the bug or the fix. In many cases, the people doing the repackaging have entirely separate bug tracking tools to upstream, and it is difficult for them to pass their information and patches to upstream directly.

Malone explicitly tracks the status of a bug both upstream and in any distributions registered in Launchpad. This makes it possible, for example, to see immediately if a fix has been found for a given bug by any of the participating distributions, or upstream. The bug page shows this information very prominently.

Watches

It's unlikely that the whole world will shift to Malone. Many larger projects have their own bug tracking tools (Bugzilla, Sourceforge and Roundup are commonly used) and some have even created custom tools for this purpose. For that reason, Malone supports BugWatches. A BugWatch is a reference to a different bugtracker that is tracking the same bug. Of course it will have a different bug number in that system, and the integration between Malone and that remote bug system is possibly limited, compared to the richness of the Malone data model, but this still allows us to keep track of a bug in a different bug tracker. For example, a bug in the Firefox package on Ubuntu would be tracked in Malone. If the same bug has been identified upstream, it would be recorded in bugzilla.mozilla.org, and we would create a BugWatch in the Malone bug pointing at that upstream bug.

Email Integration

It's important that Malone be usable entirely in email. Many open source developers use their email to track work that needs to be done. So all of Malone's features should be accessible via email, including changing the status of a bug, adding and updating watches, and possibly also requesting reports of bugs on a product or distrbution.

Distribution Bugs

Malone is designed to track bugs upstream, and in distributions. The requirements for a distribution bugtracker are somewhat specialised. A distribution consists of many source packages and binary packages, and it must be possible to track bugs at a fine level of granularity such as at the source/binary package level.

Malone allows us to create bugs that belong only to a distribution, or to a sourcepackage in a distribution if we have that information. Bugs that are not associated with a sourcepackage can be thought of as "untriaged" bugs. In some cases, we should be able to know not only which source package, but also the precise binary package that manifests the bug.

Milestones and DistroSeries

In addition, it's important to be able to know which bugs need to be fix for a given release of the distribution, or a given milestone upstream. Malone allows us to specify a milestone or a distroseries by which a bug needs to be fixed, which allows QA teams to keep track of the progress they are making towards a release.

Version Tracking

One very difficult problem faced by support teams in the open source world is that users may not all be running the latest version of a piece of code. In fact, that's pretty much guaranteed. So Malone needs to be able to say whether a bug is found in a particular version of a package or not.

Future

Bazaar Integration

Malone is part of Launchpad, a web based portal for open source developers. Another component of that portal is the Bazaar, a repository of data and metadata about code stored in the Bazaar revision control system. We hope that Bazaar will be embraced by the open source world, as it solves a number of problems with traditional centralised revision control systems and is again designed to support distributed disconnected operation.

Once more people start keeping their code in Bazaar, it should become possible to streamline the cooperation process even further. For example, if the fix for a particular Malone bug can be found in a Bazaar changeset, then it should be possible for upstream and other distributions to merge in that fix to their codebase automatically and easily. The integration could even be bidirectional - once a fix had been merged in, Bazaar could possibly detect that and mark the bug fixed in that codebase automatically.

3.1.3 How to go about writing a web application

– Steve Alexander <steve@z3u.com>

Introduction

This document presents an approach to constructing web applications, emphasising well-designed user interaction.

In summary, we write a web application by

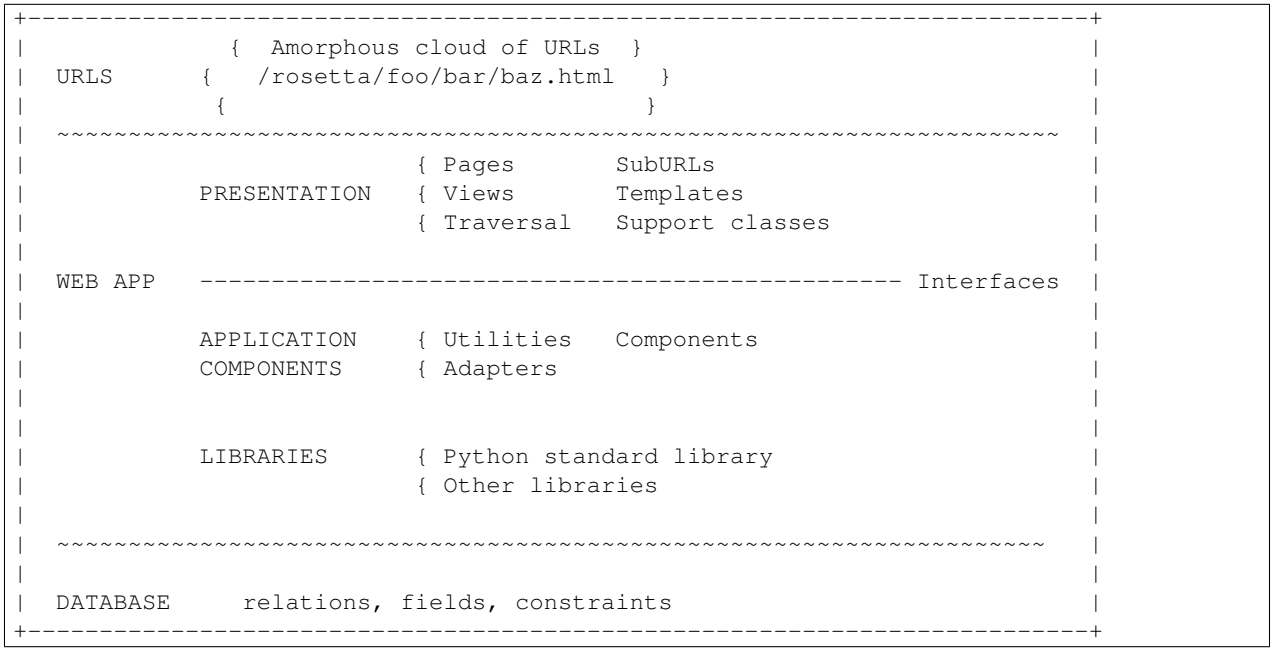
1. Design the database that lies behind the web application - Entity relationship models - SQL statements
2. Design the web application's user interface - Table of URLs (see later) - Page template mock-ups
3. Write the Interfaces that the application will use to access the database - interfaces.py files
4. Write the code and templates of the application - page templates - supporting classes - application components

Of course, this isn't a completely linear process. Steps may be carried out in parallel, and ongoing work in each step will feed into the other steps.

So that we can make rapid progress, and learn about the application as early as possible, we should do steps 2, 3 and 4 focusing on just a few URLs at a time (say, five or so). Then, when those are finished, we can make any necessary changes to the database model, and then repeat steps 2, 3 and 4 with some new URLs and new application functionality.

Web application architecture

For the purposes of this document, a web application is put together as follows:



The boundaries of our web application are URLs at the top and the database at the bottom. So, we must carefully define these things that lie at the boundaries, so that we can work out what must go in the middle.

URLs

URLs are the boundary between the web application and web browsers. When you point a web browser at a URL, you'll get one of the following:

- page
- form

- image or other data
- redirect
- “not found” error
- “unauthorized” error

We’ll be mostly concerned with pages, forms and redirects. We’ll be concerned with image and other data only when they are “managed content” and not just part of the skin of the site. The photograph of a member of the site is an example of managed content: it is stored in the database and may be manipulated through using the application. CSS files, javascript files, icons and logos are typically elements that make up the “skin” of the application, and are not managed through the application.

Another example of image or other data that is managed content is if we want to allow users to download a .po or .pot file of translations.

Forms are rendered in HTML and will typically be “self-posting forms”. That is, the address posted to when the [submit] button is pressed will be the same as the address the page was got from. This allows browsers’ bookmarks to work properly, allows us to straightforwardly validate data entered into the form, and keeps the URL space neat.

The “not found” error is the familiar “404 Not Found” given when someone tries to access a URL that is not known by our system.

An “unauthorized” error means that accessing the given URL requires some authentication. The browser will prompt the user to provide some additional authentication. Alternatively, if we use other login schemes than HTTP Basic or HTTP Digest, then the web application may present a login form to the user.

A “redirect” causes the browser to immediately fetch a different URL. This process is usually not noticed by the user.

The URL table

We need to construct a table describing the URLs used in our application. We won’t bother about the protocol part or any leading path segments of the URL. In the table below, we’ll assume all URLs start http://example.com/rosetta/...

The table has the following columns

- URL: The rest of the URL after /rosetta. For clarity, we’ll start it with a ”./”. So, the rosetta application’s main page might be ”./index.html”. If this table is written in HTML, this may be a link to a mock-up HTML page showing what will be found at that URL.
- Default: For application components, the default page to use. More about this later.
- Type: This is one of “app component”, “page”, “form”, “redirect”, “data”
- Description: A textual description of what is found at this URL. This may be a link to further information about the functioning of that page, form validation constraints, and so on.

When you get an App Component at a particular URL, what does that mean? At certain points in our URL space, we want to expose a set of related functionality under a particular URL. For example, the URL ”./projects/mozilla/show-teams” might show the teams working on the Mozilla project, while ”./projects/mozilla/translations” might show the translations of the mozilla project. Users of the system will come to understand that things related to Mozilla are to be found at URLs starting with ”./projects/mozilla”. We want to present some page at ”./projects/mozilla”. Rather than make a special “no name” page for this, we choose one of the other pages that we want to return. Mozilla is a Project. So, if the default page for a Project is “translations”, then going to ”./projects/mozilla” will return the same page as ”./projects/mozilla/translations”. The usual default page is “index.html”.

Here’s an example of a table for the Rosetta project:

```
URL                 Default      Type            Description

./                  index.html   app component   The rosetta application
```

```
./index.html                      page            Initial navigation page

./intro.html                      page            Introductory page

./signup.html                     form            Allows a new user to
                                                  register with the system.

./projects          index.html    app component   Collection of rosetta
                                                  projects

./projects/index.html             form            Shows ways to search
                                                  through projects

./projects/$PROJECT translations  app component   A particular project
                                                  $PROJECT is the name of
                                                  the project. See key below.

./projects/$PROJECT/translations  page            Shows all translations
                                                  for this project.


Key to $VARs
============
$PROJECT    The name of the project.  This is the name attribute of
            the IProjectGroup interface, or the name field in the Project
            relation.  Case is not significant, and is normalized to
            lower-case in the UI.  Examples: 'mozilla', 'gtk+'.
```

We can use the URL table for simple automated functional testing of the web application, given some suitable $VAR substitutions.

Structure of a web application URL

We need to know what types of things are at a particular URL. Here's an example of a typical URL in a web application. This time, I've included the "rosetta" path segment at the root:

```
/rosetta/projects/$PACKAGENAME/teams/$TEAM/add-member.html
 |       |        |            |     |     |
 |       |        |            |     |     page to add a new member
 |       |        |            |     Name of a particular team "22"
 |       |        |            The teams working on this project
 |       |        A particular project name, such as "mozilla"
 |       Collection of projects that can be queried
The rosetta application
```

Guidelines for URLs

- Make your URLs from lower-case letters, numbers, '-' and '+'.
- Avoid '_', capital letters, other symbols. Using these things makes the URL harder to read out over the phone or write down unambiguously.
- When you have a collection of things, such as people or projects, use a plural noun for that part of the URL. For example, "./projects".

- Consider using "+" as the last URL segment for the URL that adds things to a collection. For example, "./projects/+" to add a new project.
- Where possible, use self-posting forms. So, you would go to the URL "./projects/+" to get a form asking you for the information needed to add a new project. When you submit the form, it POSTs to the same URL. If the provided information is invalid, you'll get the form back with an error message. Otherwise, you'll get a "success" message, or be redirected to the next page in the workflow.

Development iterations

When you're developing a new system, don't try to write the whole table of URLs at once. Instead, we can work in iterative cycles, designing pages and URLs, and making these work in software. That way, we can learn earlier on if the URLs and pages we want will actually work in practice.

Here's the overall process of developing the application.

Overall Process

1. Lay out the total functionality of the system, and divide it into a number of iterations.
2. Pick the next iteration. Go through the Iteration Process described below.
3. Review / refactor the specification for previous iterations based on what we learned during this iteration.
4. Refactor the whole application implemented so far to match the refactored specification.

Each iteration (that is, step 2 above) looks like this.

Iteration Process

1. Write the URLs required for this iteration into the URLs table. Ideally, there should be 3 to 7 URLs in each iteration.
2. Document the functionality required for each page.
3. Produce page template mockups.
4. Implement the functionality, using stub application components rather than real application components.
5. Connect the functionality to the real database, by replacing the stubs with real application components.

I will note that these processes are just guidelines on how to go about writing the software. You might choose to prototype the application in order to learn about what URLs are required for some tricky interaction. Or, you might decide to write two iterations' worth of URLs into the URLs table all at once, but then implement them in two iterations. The important thing is to understand where you are in this process, and why you are doing what you are doing at any particular stage.

Keep the iterations short!

Glossary

Skin: The way the user interface looks in a web browser. The elements of this user interface, including CSS, images and an overall site template.

It is possible to provide multiple skins to the same web application, for example a simple one and a very complex one.

Published: Something that is made available at a particular URL is said to be published.

Presentation component: Some software that interacts with the browser's request and returns information to the browser. This is typically a page template or a page template plus a supporting class.

Other presentation components are traversers, which know what to do when further path segments are given in a URL; and resources, which are CSS files, javascript files, logos, icons, etc.

Application component: An object that represents application functionality, but not presentation functionality. It should have a well-defined interface so that different implementations of a given application component can be presented by the same presentation components.

Component: An object that has clearly defined interfaces.

These interfaces may represent what it offers, or what it requires in order to function.

Utility: A component that is looked up by the interface that it provides.

Adapter: A component that knows how to use a particular interface in order to provide a different interface.

Interface: A software-readable definition of an API provided by some object.

View: A kind of presentation component that provides a representation of some other component.

Browser presentation: Presentation intended for a web browser, as distinct from a presentation intended for XML-RPC or webdav. Or even email.

Non-published {view,resource}: A {view,resource} that is used by other presentation components, but that is not itself addressable by a URL.

Page: An HTML document returned to a browser in response to a GET or POST request to some URL.

Form: A page that contains HTML form elements and at least one "submit" button.

Self-posting form: A form that's "action URL" is the same address that it was loaded from. So, a form that was loaded from "./projects/+" would start:

```
<form action="http://example.com/rosetta/projects/+" method="POST">
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search