ubuntu

Ubuntu Developer Guide

Ubuntu Developer Team

Table of Contents

Quickly

Overview

Quickly is an application that helps you create software or command line applications and python games quickly, it can also be used to create other things. You can select from a set of templates and use some simple Quickly commands to create, edit code and GUI, and publish your software for others to use.

Quickly's templates are easy to write. So if you are a fan of language foo, you can create a foo-project template. Or if you want to help people making plugins for your killer app, you can make a killer-app-plugin template. You can even create a template for managing corporate documents, creating your awesome LaTeX helpers, the sky is the limit!

Quickly is a command line based application which makes programming easy and fun by bringing *opinionated* choices about how to write different kinds of programs to developers. There are several commands that you will need to know about which we will explain in this section.

Get Quickly

It's free and easy to get everything you need to start programming installed, you just need to install Quickly and everything will be set up for you. To install Quickly, open a Terminal window, and type:

```
$ sudo apt-get install quickly
```

Hit Enter and you will be prompted to supply your user password. Quickly will be downloaded and installed. Quickly contains many elements so it may take a while to get everything downloaded and installed.

Start A Project

You start a new Quickly project using Quickly's **create** command. In this guide, you will be starting and enhancing a Feed Reader for identi.ca, a free service similar to Twitter. The application looks roughly like this:

Figure 1. TODO:screenshot of FeedReader

For the demonstration purposes, we can name this application "Feed Reader". I know, not very original, but it does the job. Get started by typing:

```
quickly create ubuntu-application feed-reader
```

Let's look briefly at each part of this command.

- **quickly** - tells the terminal to use the program *quickly*. Note that capitalization is important. "Quickly" with a capital "Q" is the name of the project, but "quickly" with a lower case "q" is the name of the program. Terminal won't know what "Quickly" with a capital "Q" is.

- **create** - the Quickly command to use. Quickly has a set of commands you can use. Besides **create** , there is **edit**, **design** , **share**, **release**, **license**, **add** , and even a few others. This guide will explain commands as they are used.

- **ubuntu-application** - the name of the template to pass to the **create** command. There are other templates, such as a template for creating a command line application, and one for creating a game. But this guide only covers using the ubuntu-application template.

- **feed-reader** - the name of the project you are creating. The way naming works, is that you always name your project with lower case words, and if you want to have more than one word for the name, separate the words with a dash, not a space. Quickly will take care of converting that input correctly in titles, library names, modules, etc...

First Look

After running the create command, Quickly will output some information to the Terminal, and then will finish by actually running your application. Note that the window is named correctly. Also, there are menus and an About Box and even a preference dialog, that all work.

Figure 2. TODO:screenshot of first run application, with About Box showing

After you quit the application, you need to move the terminal into the directory for your project. Quickly created a directory called "feed-reader" for this purpose. Use the **cd** command to "change directory" into this newly created directory:

```
$cd feed-reader}
```

Run the Project

Once you are in the project directory, you can run your application again by issuing the run command

```
$quickly run
```

Use the command **quickly run** after you make changes to your GUI or to your code to test them out.

Explore the Code

To take a look at the code that Quickly created for you, use the **edit** command.

```
$ quickly edit
```

This will open all of your code in Gedit. Lots of code was created for you, this is called the "boiler plate"(standard code created from a template) code. You can edit any of the files to make your application work the way you want to. In practice you will mostly be editing the main bin file, and other files that you may add, such as dialogs for example. You can switch to the feed-reader file, and take a look at the generated class "FeedReaderWindow".

In Gedit, the bin file is always named the same as the project, and is stored in the bin directory. This is because it is a special file that Ubuntu runs when you ask Ubuntu to run the application. All the other files are considered part of your projects library or data, so those files are in different directories.

Look at the GUI

If you want to start modifying the GUI, you can open Glade. You use the "design" command to do this. Don't try to open Glade and then open your project from in there, it won't work. So, just use this:

```
$ quickly design
```

The program Glade will open with your project loaded.

Figure 3. TODO:screenshot of glade

Quickly Tutorial

To learn more about Quickly use the command **quickly tutorial** to launch a more comprehensive introduction to the application.

Revision History

Revision History
Revision   Creating   an2011
application with Quickly V1.0

Glade

Rick Spencer

`<ubuntu-developer-manual@lists.launchpad.net>`

Revision History

Revision Glade V1.0   2011

Overview

This section will describe how to layout and use widgets to make a functional user interface for Ubuntu. We'll do this by laying out a user interface for a feed reader application using Glade. The application will look like this:

Figure 1. TODO:feedreader application build using Glade

Boxes

The Gtk library positions widgets in relation to each other and their parent containers. This works much like a grid bag layout manager in Java's awt library, HTML, or XAML Windows Presentation Foundation. The essential container elements are the VBox and the HBox. The VBox orients child widgets vertically, and the HBox orients child elements horizontally. By combining HBoxes and VBoxes, you can create virtually any layout desire.

To follow along, create a Quickly application:

```
quickly create ubuntu-application feed-reader
```

To run Glade for your Quickly application, you must use the command **quickly design**. If you try to run Glade directly Glade will not work because it requires certain configuration options to work correctly with Quickly.

So, cd into your project directory and type:

```
quickly design
```

Your Quickly application will open. You can use the tabs to select between the different windows and dialogs in your application. If it's not already selected, choose "FeedReaderWindow" from the project menu.

Figure 2. TODO:feedreader window open in Glade

Notice that there are some default widgets in the Window. You should start by deleting those. Click on the big Ubuntu logo. When it is selected, press the delete key on your keyboard to delete it. Do the same for labels on the form.

Now your Window has a VBox ready for filling with widgets. In a Quickly application, "vbox1" is the name of the primary container. The window we are designing is laid out in a more or

less top to bottom format, in four rows of controls, so we need four empty rows in vbox1. The first row is already taken by the menu bar, and the bottom row by the status bar, so we need to add two rows. You can do this by selecting vbox1 in the inspector. Then in the editor, select the General tab and change "Number of Items" fro 4 to 6. A new row will appear.

Figure 3. TODO:Glade set up with vbox1 set to have 5 sections

Adding Widgets

We'll add the widgets from top to bottom. Start by adding the tool bar to the top. In the Toolbox window, click on the Tool Bar icon. Then, click in the first empty spot in VBox. Since the tool bar is empty, the spot will immediately collapse down to have no height. Click on the pencil in Glade's tool bar to bring up the Tool Bar Editor.

Figure 4. TODO:Toolbar editor with collapsed tool bar behind

Switch to the Hierarchy tab of the Tool Bar Editor. Then click the Add button to a new widget to the tool bar. To make a refresh button, set the Stock Id to refresh.

Stock widgets in Gtk are incredibly useful. When you set a widget to a stock widget, Gtk will apply the whatever theme desktop theme the user has chosen to the widget, including the right icon if required. It will also supply the right word, so that the widget will show up translated into whatever language the user is using on their desktop with no translating.

So to create the Refresh button, choose gtk-refresh as the stock id, and you'll get the right icon and translations, and everything for free.

Figure 5. TODO:Tool Bar Editor with the stock icon set

To make it a bit easier to program later, rename the button from |toolbutton1| to | refresh_button|. You could easily add more widgets to the tool bar, but for now, we'll only need the refresh button.

The next step is to add a label at the top to display your last dent. Select Label from Controls and Display and add that to the next open area. If you want, you can quickly change the label. Click on the label to select it, then select the General tab in the editor. Then set whatever you want in the ``Label'' field.

Figure 6. TODO:insert screen shot of changing label text

Next, we'll add the text entry and button fields at the bottom. These are two widgets, laid out horizontally, so we'll use an HBox with two items. Click on HBox in the containers section of the palette, and then click in the empty space at the bottom of the Vbox you just created.

Figure 7. TODO:changing label text picture of Glade with the HBox added

To add the entry field, click on TextEntry under the Control and Display section of the toolbox, the click in the left hand pane of the HBox you just added.

To add the button, click on ``Button'' (also in the Control and Display section of the palette), and put a button in the right hand box.

Figure 8. TODO:changing label text picture of Glade with the entry button

Notice that the widgets don't look like what we're designing. For instance, the button is huge and just says ``button,'' and the spacing above the entry looks odd. We'll fix the sizing problems next, but first, it's easy to change the label on the button. Simply select the button, then on the general tab, change the Label property to "Dent".

Figure 9. TODO: picture of Glade with button text set

The middle section of window will be taken up by all the dents, and we'll use a VBox for as a container. However, we want the VBox to be scrollable, so we'll add a scrollable region. Under Containers in the toolbox, click on Scrolled Window, and then add it to the VBox.

Next, we want to add a VBox to the scrolledwindow1. However, it won't work if we try to add it directly, because VBoxes and HBoxes don't support scrolling. So, add a View Port (also under Containers in the toolbox) to scrolledwindow1, and then add a VBox to the Viewport. You don't much have to worry about the number of items in the VBox, it will grow as you add items from code.

Since we will be accessing the ScrolledWindow, The Label. The TextEndtry, and the Button from code, it is a good idea to give them names that will make them easier to remember. For example, after adding the scrolled window, in the editor, under General, change the name to something like "dent_vbox". Do the same for the other widgets, such as "dent_entry", "dent _button", "dent_label".

Figure 10. TODO: Glade showing renamed vbox

Packing

Make sure to save your file in Glade so the changes that you made take effect before running. Then switch back to your terminal and use:

```
quickly run
```

to test the application.

As you maximize or resize the application, you'll notice:

1. The button at the bottom is big, and stretches to fill the space no matter how big or small we make the window.

2. The labels for the previous dent does the same thing.

3. The space between the controls and the outside of the window aren't right.

4. The button just says ``button.''

We'll fix the first three problems using the `Packing' tab of the editor. We'll get to item 4 a little later.

Expand and Fill

Fill is a property that tells a widget that all of it's children should fill up it's space. Expand tells a widget to expand to fill all the space allotted to it. You can use these two properties to control the size of widgets in relation to each other.

To get started fixing the buttons, choose the HBox in the inspector that contains the buttons. Choose the Packing tab of the editor. Notice that Expand and Fill are both set to ``Yes''. Click the Fill button to set it to ``No''. Notice that the buttons no longer fill up the whole Height of the Hbox. That's because you just told it not to tell it's children to fill up the space alloted to it.

Now click the ``Expand'' button to set that one to ``No''. Notice that the Hbox is not only as high as it's child widgets. That's because you just told it not to expand to fill the space alloted to it.

Now we need to apply the same logic to the widgets in that HBox.. Click on the button to select it, and then set expand and fill to ``No''.

The entry for the dents to be posted, should expand to fill all the space alloted to it. Therefore, there is no need to change the packing for dent_entry.

Change the Expand and Fill for the dent_label to "No" as well. If you run the application again, you'll see that except for the missing dents, which won't be added until the next chapter, the window is working properly.

The spacing of the controls could be nicer. We'll use the padding and border properties to fix this. Padding tells a widget how much space to put between itself and it's sibling widgets (and also the side of it's parent), while border tells a widget how much space to surround itself with.

These two properties are additive, so that if you set a border property, it tells the widget to include that space in addition to any space that is already taken up by padding.

To create the right spacing between all the other widgets, select each and set their padding to 5. Padding is in the Packing tab of the editor.

Menus

Quickly provide a set of default menus for FeedReader. If you want to adjust those menus, you can do that by selecting the menu in the inspector, and then clicking the "Edit" button on Glade's toolbar. This will bring up the editor dialog window. You can switch to the Hierarchy tab to add, remove, and move menu items.

Responding to Signals

Your code will need to respond to three of the widgets that you've added, but buttons, and also the user should be able to just use the enter key to send their dent.

We'll write just enough code to prove to ourselves that we are able to respond to signals from these widgets. To start writing code, we need to open it in an editor. You can use the edit command for this.

```
quickly edit &
```

This will open all of your code in Gedit. The file called "feed-reader" is your main file, so choose that for editing. We'll use the auto-signals feature to wire up the signals. Add the following methods to the FeedReaderWindow class:

```
[firstnumber=last]
def refresh_button_clicked_event(self, widget, data=None):
    print "REFRESH"

def dent_button_clicked_event(self, widget, data=None):
    print "DENT"

def dent_entry_activate_event(self, widget, data=None):
    print "DENT"
```

Now you can run the project using the run command, and when you click the buttons, notice that the terminal prints the correct message.

```
 quickly run
```

How did this work? Your Quickly project used the auto-signals feature to connect the button to the event. To use auto-sginals, simple follow this pattern when you create a signal handler:

```
[firstnumber=last]
def widgetname_eventname_event(self, widget, data=None):
```

Sometimes a signal handler will require a different signature, but (self, widget, data=None) is the most common. We'll start to make the feed reader really work in the next chapter.

Using Bazaar for Revision Control

Overview

Bazaar is the Ubuntu and Launchpad revision control system. It is usually referred to as 'bzr'.

Bzr is similar to other revision control systems, such asi git , svn, and cvs. Unlike those systems, bzr leverages Launchpad's capabilities and has native support for *distributed development*. That is, unlike systems that lock down a file if someone needs to modify it, bzr allows multiple people to work simultaneously on the same resource, and then bzr helps merge the results together.

With bzr you can:

- Enable a directory for revision control: **bzr init**

  A *branch* is a directory under bzr revision control.

- Add the files you want: **bzr add**

- Commit changes with a log message: **bzr commit**

- See the bzr commit log: **bzr log**

- See uncommitted changes in a file: **bzr diff <file>**

- Push changes to a central trunk: **bzr push**

  A *trunk* is the central branch for a project on Launchpad.

- Carry trunk changes into local: **bzr merge** and **bzr pull**

- Roll back changes to previous states: **bzr revert**

- Display bzr help: **bzr help**

And much more. You can use bzr in any directory on your system without any connection to Launchpad. Bzr can be used on Linux, Mac and Windows systems. For the complete guide, see the project's homepage at: http://bazaar.canonical.com.

Using Bazaar Locally

Let's start with an example of using bzr in a local system to track changes to some files. Then, we will dive into using bzr with others to develop a common *trunk* branch.

In this example, you create a bzr branch, add some files to it, commit them and add a log message. Then, you change a file, use **bzr diff** to see what you changed, and use **bzr revert** to undo the change.

1. Create a directory named 'bzrdemo' and move into it:


```
mkdir bzrdemo
```

```
cd bzrdemo
```

2. Add some files to the directory:

```
touch file1 file2
```

3. Initialize the directory as a bzr branch:

```
bzr init
```

4. Add all files in the directory to the branch:

```
bzr add
```

Note

You can add single files by naming them: **bzr add file1 file 2** or a subdirectory by specifying it: **bzr add mydirectory**

5. Commit all file changes. You will be prompted to enter a commit message that goes into the bzr log:

```
bzr commit
```

Note

You can commit changes to specific files by naming them: **bzr commit file1 file2**

6. Display the bzr log to see your commit message:

Note

It is useful to pipe the log through **| less** so the latest log entries do not scroll out of sight.

```
bzr log | less
```

7. Modify file1 and display status, which indicates the files that have changed and have not been committed:

```
bzr status
```

8. Display a diff of file1 that shows the changes that are not yet committed:

```
bzr diff file1
```

Note

You can display the bzr diff for all files with **bzr diff** with no file argument.

9. Let's say you don't want to keep the changes to file1 and want to revert file1 to the last commit state:

```
bzr revert file1
```

Note

You can revert all files to their last commit state with **bzr revert** with no file argument.

With the commands just covered you can do most of what is needed to work with an isolated branch.

When developing Ubuntu Applications, there is a trunk branch on Launchpad from which people create local branches. These local branches are modified and pushed to trunk. But what happens if trunk has changed in the meantime? Let's take a look at how Bazaar handles simultaneous changes from multiple people. This is covered next.

Distributed Development

While some revision control systems lock down a file when someone wants to modify it, bzr takes another path. Bzr allow people to simultaneously modify the same resource. In bzr terms, this is called 'distributed development.'

Distributed development creates a situation where the trunk branch can be changed by someone else after you made a local branch from it and before you try to push you changes

to it. Naturally, you are not able to push your branch to trunk if trunk has changed without taking steps to resolve the conflicts.i

The next sections covers how to handle this.

First:

- We cover a simple case where trunk gets ahead of your local branch.

- Then, we cover how to resolve differences when the two branches have *diverged*, that is, when someone modifies trunk after you have branched from it and have committed changes locally.

When Trunk Gets Ahead of Local

Let's say you pushed some changes to trunk a while ago. Now you want to make some more changes. You still have your local branch. But, someone may have made changes to trunk, in other words, the trunk may be *ahead* of your local branch.

Figure 1. Trunk Branch is Ahead of Local Branch
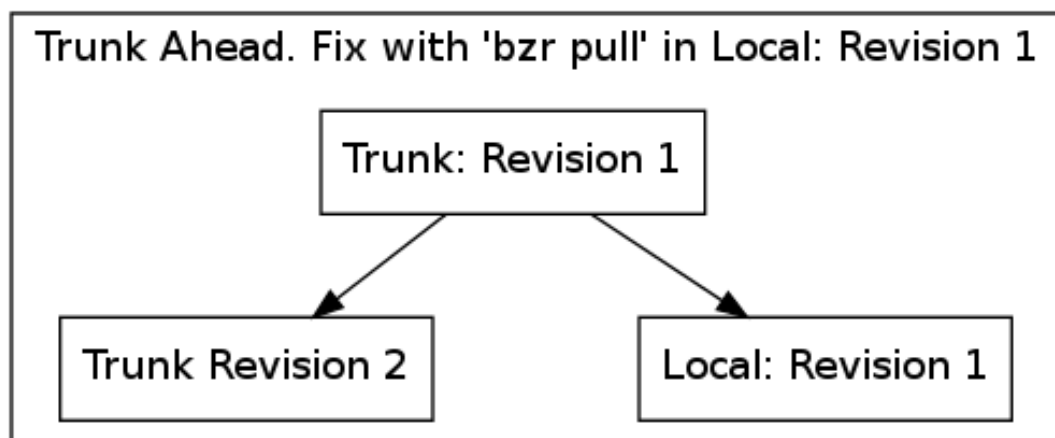
Before making any changes to your locale branch, bring your local branch up-to-date with the trunk branch with the **bzr pull** command.
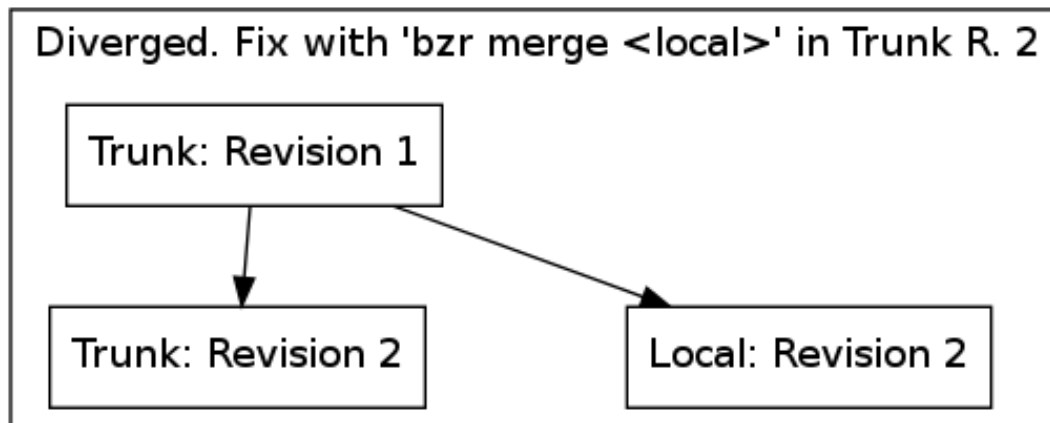
Note

You cannot use **bzr pull** when your branch has uncommitted changes. Use **bzr status** to show whether a branch has uncommitted changes.

After that, use **bzr status** to see new log messages pulled in from trunk, if any.

When Trunk and Local 'Diverge'

If trunk is ahead of your local branch *and* you have local committed changes, then the two branches are *diverged*. In this case you cannot push your changes to trunk until you first use **bzr merge**.

Figure 2. Trunk branch is diverged from local branch

The best approach is to get a fresh trunk and merge your local branch *into trunk* from the trunk directory.

Note

You could merge trunk changes into your local branch, but that is not a good practice because the log message of the branch being pulled in are *collapsed*. Because you do not want to collapse log messages already pushed to trunk, it is a best-practice to allow your local log messages to be collapsed. You can do this by getting a fresh branch of trunk and *from inside trunk* executing **bzr merge <path-to-local-branch>**.

As an example, let's say the project is named *blackbird*. Because you branched from blackbird's trunk branch, your local branch directory is named *blackbird*. After trying to **bzr push**, you found the two branches have diverged. Here is how you can resolve this situation.

1. Move to your local branch's parent directory, and rename your local 'blackbird' directory to 'blackbird-local':

```
mv blackbird blackbird-local
```

2. Get a fresh branch of *blackbird* trunk:

```
bzr branch lp:blackbird
```

3. Move into the fresh trunk branch directory, *blackbird*:

```
cd blackbird
```

4. Merge your local branch into the trunk branch:

```
bzr merge ../blackbird-local
```

In many cases, bzr merges diverged files automatically. If so, you can simply **bzr commit**.

Sometimes, bzr needs your help to merge diverged files. If so, bzr reports the files that need your attention.

Note

Use **bzr conflicts** to list files that need your attention.

In this common case, you need to edit the files manually. bzr inserts labels into the text that show the parts of the files that need your attention. You must edit the files until they are correct.

Then, execute **bzr resolve** to notify bzr that you have the files in their correct state.

Lastly, commit the files with **bzr commit**

Now, this trunk branch is:

- The up-to-date trunk branch

- With your local changes merged in

You can push your merged trunk to Launchpad with **bzr push**.

Revision History

Revision History
Revision Using Bazaar for2011
Revision Control V1.0

Using Launchpad to Manage Your Project

Overview

Launchpad is a free web site that hosts open source development projects. The home page for Launchpad can be found at: http://launchpad.net.

The first project hosted on Launchpad was Ubuntu: http://launchpad.net/ubuntu. you can open a project on Launchpad by adding the project name to the URL: http://launchpad.net/unity .

Launchpad now hosts thousands of projects. Each project benefits from a dedicated bug tracking system, a translation web portal, bzr revision control, *teams* for permissions, and Personal Package Archives (PPAs), which automatically build source packages into binary (installable) packages and publish them.

This section will quickly cover the points necessary to use Launchpad for quick success for opportunistic Ubuntu application development. A more extensive help system for Launchpad can be found at: http://help.launchpad.net/FrontPage.

Getting Started

Naturally, before you can do anything in Launchpad, you need to create a user account. If you do not already have an account you can create your user account from the launchpad home page: http://launchpad.net.

Let's suppose that you have an idea for a new Ubuntu application. This means a new Debian source package, developed, built, translated, and published. You have started with *Quickly* locally but you also need bug tracking, a development team with source file revision control, and life cycle management tools.

This is where Launchpad *projects* comes in.

Note

You can experiment with Launchpad, including creating projects and teams, with http://staging.launchpad.net. Note that work on staging is automatically deleted periodically, so it is strictly for temporary experimentation.

Creating a Launchpad Project

Now that your user account is created, you can create a Launchpad project here: https://launchpad.net/projects/+new where new is the name of your test project.

There are a few fields you need to fill in that are self-explanatory.

Your new project home page is: https://launchpad.net/mynewproject.

For example, let's say I created a project named *blackbird*. Its home page would be: https://launchpad.net/blackbird . Or, if you created the blackbird project on staging, its URL would be: https://staging.launchpad.net/blackbird .

Creating a Launchpad Team

Although you may be ready to push your local *Quickly* project branch to the Launchpad project, you know other people who want to help out with development. You probably want to limit who can push changes to just these people; for this you will need a Launchpad *team*.

Create your new team here: https://launchpad.net/people/+mynewteam. Your new team's home page is: https://launchpad.net/~mynewteam . Note that Teams always start with a tilde ~ in Launchpad URLs. This will help differentiate between Team and project URLs.

For example, let's say you create a team named *blackbird-team*. Its home page would be: https://launchpad.net/~blackbird-team . *Teams* also host Launchpad PPAs.

Enabling Project Code Hosting

Now that your team is in place, you can enable your project to host bzr branches.

Enable code hosting with the project's *Code* tab and then *Configure code hosting*.

Use the displayed options to:

- Create a new, empty branch

- Set the branch name to *trunk*

- Set the branch owner to your team

Projects host branches. Teams maintain branches. So you can see branches for a project or for a team, and the URLs reflect this.

- *Project's* code is here: https://code.launchpad.net/theprojectname

- *Team's* code is here: https://code.launchpad.net/~theteamname

- *Team's* project code is here: https://code.launchpad.net/~theprojectteam>/projectname

But, we haven't yet actually *pushed* any branches yet, so let's take a look at how to do that.

Pushing to Trunk

Now that code hosting is enabled, you can push your branch to the project's trunk as follows:

```
bzr push lp:projectname
```

Note that because you configured the *trunk* capability when setting up code hosting, you can use the "lp:project" style of URL to push. For example, if the project is named ``blackbird'', you could push to trunk with:

```
bzr push lp:blackbird
```

Note: It may be necessary on the very first push to use the ``--use-existing'' argument:

```
bzr push --use-existing lp:blackbird
```

Now that you have pushed an actual branch to trunk, you can view information about it, such as commit log messages, from its Launchpad page: https://code.launchpad.net/~projectteam/projectname/trunk .

To push *non-trunk* branches to you project, use a *complete* bzr-style URL that specifies the team, the project and the branch: **lp:\~teamname/projectname/branch**. For example, you might want to push an experimental branch to blackbird as follows: **bzr push lp:~blackbird-team/blackbird/experimental**

For the imaginary blackbird project and blackbird-team, the URL would be: https://code.launchpad.net/~blackbird-team/blackbird/trunk .

From any such Launchpad branch page, you can display all files in the branch and see what has changed with each commit (bzr log messages and color-coded diff visualizations).

Note

Tip: **bzr merge proposals** let people propose commits with a review process and other helpful work flow.

Now that you and your team can push directly to trunk, what next?

Other Launchpad Features

In addition to basic code hosting Launchpad also contains bug tracking and ways to manage your development process. Let's take a quick look at some of these features.

Creating Bug Tracking

Before you release and distribute your project, you are likely to go through a period of development in which bugs are found and need to be tracked to resolution (to ``fix-committed'' status). Launchpad provides project bug tracking for this. From the project's Launchpad page you will see a tab named "Bugs". First, you need to configure bug tracking for your project by clicking the "Bugs" tab of your project home page and using "Configure bug tracking".

Simply provide the few bits of required information, and bug tracking is enabled for the project and available on the project's "Bugs" tab. You can also open the "Bugs" page with a URL: http://bugs.launchpad.net/projectname

Bug tracking allows you and others to create bugs, assign them, change their status, add tags to them, and target bugs for any "milestones" you create for your project.

Managing Development to Milestones

Milestones help coordinate development activities. For example, you might want an alpha phase (proof of concept) and one or more beta phases (to test and find and fix bugs). The "finish line" is the release, but to get there, you may want to go through a few phases or "series". "Series" enable differentiation of key branches. For example, the *trunk* series is the current development focus. You may use other series for other Ubuntu releases, for example 10.10 (Maverick).

You can create milestones for each "series" and then add those milestones from a drop-down list to individual bugs. For example, you can target a set of bugs to a "beta-2" trunk milestone. You can also view just those bugs that have been targeted to any milestone.

To create a milestone:

1. Navigate to your project's code page: https://code.launchpad.net/projectname

2. Find the trunk branch \textbf{Series: trunk}, and click on the trunk link there. This takes you to a page like this: https://launchpad.net/projectname/trunk

3. Use "Create milestone" to create the milestone name. Fill in any other fields you find useful.

Now, you can target a bug to a milestone from the bug web page (open the bug) with \textbf{Target to milestone}.

Milestones are displayed in various places. If you click one, you can see bugs that are targeted to it.

What good are bugs if your application is not built and distributed? For that, you will need a PPA.

Creating your Team PPA

You can use Quickly to build your project locally (for testing). But, use Launchpad builds for distribution (at least during development \dash other distribution option are available for wide-spread distribution.)

That is what PPAs are for: predictable, uniform source package builds into binaries (for target CPU architectures) and hosting them for distribution. PPAs distribute both *source* packages and their built *binary* packages.

To create a PPA for your team:

1. Navigate to your team's code page: https://code.launchpad.net/\~teamname

2. Use "Create a new PPA"

You can also use the **dput** command to push a source package to a PPA.

A few key points about PPAs:

1. PPAs host and build packages for multiple different Ubuntu series

2. PPAs build packages for multiple different CPU architectures

3. You can display details about packages with the \textbf{View package details} link. From the details page, you can download the package (source and binary) and see the package build log.

4. You can install from and push to the PPA using the instructions at the top of the page

We have covered some (but by no means all) Launchpad features. Did you know that you can help develop Launchpad?

Launchpad is open source. That means you can help out, by finding and fixing bugs, and even by adding new features. See https://dev.launchpad.net/.

Revision History

Revision History
Revision   Using   Bazaar   for2011
Revision Control V1.0

Distributing Your Work

Micahel Terry

`<ubuntu-developer-manual@lists.launchpad.net>`

Distribution overview

This section covers how to share and distribute your program with other people. From just a few friends to the entire Ubuntu community sharing your work not only enhances the community but lets users and other developers test, use and even help you make your work better. By the end of this section, you should be able to publish packages into Personal Package Archives (PPAs) and submit applications to the Ubuntu Software Center.

When users install your application, they'll be installing a file called a `binary package'; this is a file that ends with the .deb extension. This is transparent to your users but knowing how to generate packages and publish them is important for testing and releasing your software.

Users will either install your application from a PPA or the Ubuntu Software Center. A PPA is a channel for delivering application updates to users that you have full control over. It's most useful for pre-release versions or to quickly and easily share new versions. But to make your application available to a wider audience, you should submit your application to the Ubuntu Software Center. We'll cover that process at the end of this section.

Getting Set Up

One thing you should probably do if you're publishing software is have a project web page. See https://help.launchpad.net/Projects/Registering for more information on registering a project on Launchpad.

Another good thing to do before actually sharing your application is to make sure that certain information about your application is filled out. This includes project contact information, a description of what your application does and any other helpful information.

Run the following command from your Quickly project directory:

```
$ gedit setup.py
```

This file contains the code to package your software. Most of this is not intended for direct editing. But near the bottom you will see a code block that looks like the following:

```
DistUtilsExtra.auto.setup(
name='feed-reader',
version='0.1',
#license='GPL-3',
#author='Your Name',
#author_email='email@ubuntu.com',
#description='UI for managing ...',
```

```
#long_description='Here a longer description',
#url='https://launchpad.net/feed-reader',
cmdclass={'install': InstallAndUpdateDataDirectory}
)
```

Change these fields to something more suitable for your project. Make sure the line isn't commented out (\ie, make sure to remove the `\#' character at the beginning of the lines you change). You should end up with something like the following:

```
DistUtilsExtra.auto.setup(
name='feed-reader',
version='0.1',
license='GPL-3',
author='Oliver Twist',
author_email='twist@example.com',
description='Example feed reader',
long_description='This test program lets you read all your favorite RSS feeds'
url='https://launchpad.net/feed-reader',
cmdclass={'install': InstallAndUpdateDataDirectory}
)
```

Testing Locally

It is important to make sure that your application installs correctly before you publish your work. You will need to generate a binary package file and install it manually for testing.

Starting in your Quickly project directory, run the following command:

```
$ userinput{quickly package}
```

This will generate a feed-reader_0.1_all.deb in the directory above your current one. Install it with the following command:

```
sudo dpkg -i ../feed-reader\_0.1\_all.deb
```

Your application is installed! You can run it by going to the Applications menu or running the **feed-reader** command.

Publishing Development Releases

To share an in-progress version of your application, you can publish into a PPA. See https://help.launchpad.net/Packaging/PPA [https://help.launchpad.net/Packaging/PPA]for more information about creating a PPA. Let's say you have a ppa called `testing.'

```
quickly share --ppa testing
```

Now you can go to https://launchpad.net/people/+me/+archive/testing [https://launchpad.net/people/+me/+archive/testing] and see that the package is building. Wait 30~minutes for it to finish and then follow the instructions on the page to add the PPA to your system.

Publishing Stable Releases

To publish a new stable version of your application, again you'll publish into a PPA. It's recommended you use a separate PPA from your development releases. For this example you will use a PPA called ``stable'' to release version 1.0:

```
quickly release --ppa stable 1.0
```

If you don't specify a version, a default version of year.month.release will be used. For example, your first release during July 2011 will be versioned 11.07. Your next release that month will be 11.07.1.

Now you can go to "https://launchpad.net/people/+me/+archive/stable [https://launchpad.net/people/+me/+archive/stable] and see that the package is building. Wait 30~minutes for it to finish and then follow the instructions on the page to add the PPA to your system.

Submitting to the Ubuntu Software Center

When you're ready to go the final step and publish your software in the Ubuntu Software Center, you should follow the steps outlined on the Ubuntu Wiki: https://wiki.ubuntu.com/AppReviews [https://wiki.ubuntu.com/AppReviews].

The process includes:

1. Preparing your application for assessment

2. Submitting your application to the Application Review Board via Launchpad

3. Incorporating feedback from the Board

When the application is approved by the Board it will be published in the Ubuntu Software Center.

Example Glossary

This is not a real glossary, it's just an example.

E

Extensible Markup Language	Some reasonable definition here. See Also Standard Generalized Markup Language.

S

SGML	See Standard Generalized Markup Language.
Standard Generalized Markup Language	Some reasonable definition here. See Also Extensible Markup Language.

Credits

Ubuntu Developer Documentation Team
`<ubuntu-developer-manual@lists.launchpad.net>`

Copyright © 2011 Ubuntu Developer Documentation Team

Revision History

Revision Credits V1.0	2011

People

This resource wouldn't have been possible without the efforts and contributions from the following people:

- Rick Spencer - Team Founder and Author

- Kyle Nitzsche - Author, Toolchain Developer, Translation Coordinator

- Kevin Godby

- Ryan Macnish

- Chris Woollard

- Didier Roche - author

- Owais Lone - author

- Michael Terry - author

- Stuart Landgridge - author

- Belinda Lopez - editor