Ubuntu Packaging Guide

Release 1.0.0 bzr650 ubuntu14.04.1

Ubuntu Developers

January 19, 2018

Welcome to the Ubuntu Packaging and Development Guide! We are currently developing codename Bionic Beaver, which is to be released in April 2018 as Ubuntu 18.04 LTS.

This is the official place for learning all about Ubuntu Development and packaging. After reading this guide you will have:

- Heard about the most important players, processes and tools in Ubuntu development,
- Your development environment set up correctly,
- A better idea of how to join our community,
- Fixed an actual Ubuntu bug as part of the tutorials.

Ubuntu is not only a free and open source operating system, its platform is also open and developed in a transparent fashion. The source code for every single component can be obtained easily and every single change to the Ubuntu platform can be reviewed.

This means you can actively get involved in improving it and the community of Ubuntu platform developers is always interested in helping peers getting started.

Ubuntu is also a community of great people who believe in free software and that it should be accessible for everyone. Its members are welcoming and want you to be involved as well. We want you to get involved, to ask questions, to make Ubuntu better together with us.

If you run into problems: don't panic! Check out the *communication article* and you will find out how to most easily get in touch with other developers.

The guide is split up into two sections:

- A list of articles based on tasks, things you want to get done.
- A set of knowledge-base articles that dig deeper into specific bits of our tools and workflows.

CHAPTER

ONE

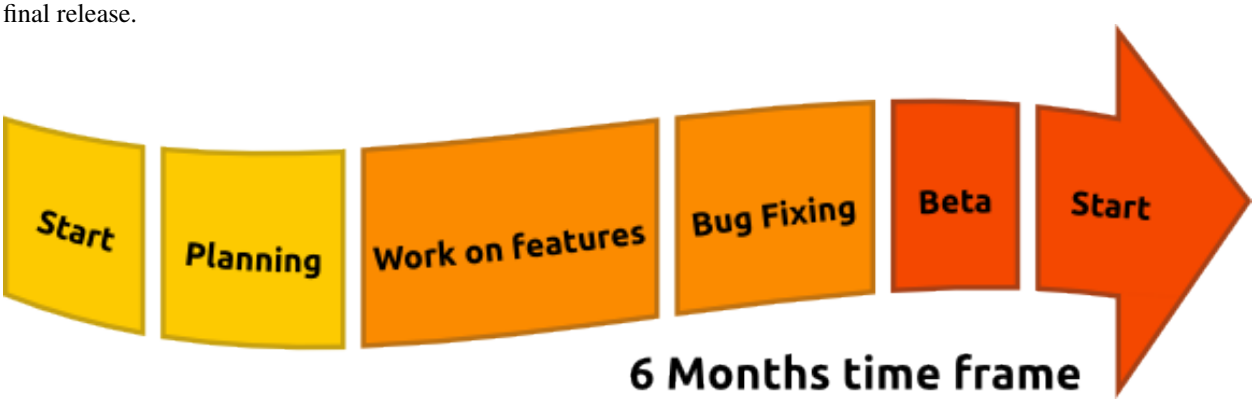
ARTICLES

1.1 Introduction to Ubuntu Development

Ubuntu is made up of thousands of different components, written in many different programming languages. Every component - be it a software library, a tool or a graphical application - is available as a source package. Source packages in most cases consist of two parts: the actual source code and metadata. Metadata includes the dependencies of the package, copyright and licensing information, and instructions on how to build the package. Once this source package is compiled, the build process provides binary packages, which are the .deb files users can install.

Every time a new version of an application is released, or when someone makes a change to the source code that goes into Ubuntu, the source package must be uploaded to Launchpad's build machines to be compiled. The resulting binary packages then are distributed to the archive and its mirrors in different countries. The URLs in `/etc/apt/sources.list` point to an archive or mirror. Every day images are built for a selection of different Ubuntu flavours. They can be used in various circumstances. There are images you can put on a USB key, you can burn them on DVDs, you can use netboot images and there are images suitable for your phone and tablet. Ubuntu Desktop, Ubuntu Server, Kubuntu and others specify a list of required packages that get on the image. These images are then used for installation tests and provide the feedback for further release planning.

Ubuntu's development is very much dependent on the current stage of the release cycle. We release a new version of Ubuntu every six months, which is only possible because we have established strict freeze dates. With every freeze date that is reached developers are expected to make fewer, less intrusive changes. Feature Freeze is the first big freeze date after the first half of the cycle has passed. At this stage features must be largely implemented. The rest of the cycle is supposed to be focused on fixing bugs. After that the user interface, then the documentation, the kernel, etc. are frozen, then the beta release is put out which receives a lot of testing. From the beta release onwards, only critical bugs get fixed and a release candidate release is made and if it does not contain any serious problems, it becomes the final release.



Thousands of source packages, billions of lines of code, hundreds of contributors require a lot of communication and planning to maintain high standards of quality. At the beginning and in the middle of each release cycle we have the Ubuntu Developer Summit where developers and contributors come together to plan the features of the next releases. Every feature is discussed by its stakeholders and a specification is written that contains detailed information about

its assumptions, implementation, the necessary changes in other places, how to test it and so on. This is all done in an open and transparent fashion, so you can participate remotely and listen to a videocast, chat with attendants and subscribe to changes of specifications, so you are always up to date.

Not every single change can be discussed in a meeting though, particularly because Ubuntu relies on changes that are done in other projects. That is why contributors to Ubuntu constantly stay in touch. Most teams or projects use dedicated mailing lists to avoid too much unrelated noise. For more immediate coordination, developers and contributors use Internet Relay Chat (IRC). All discussions are open and public.

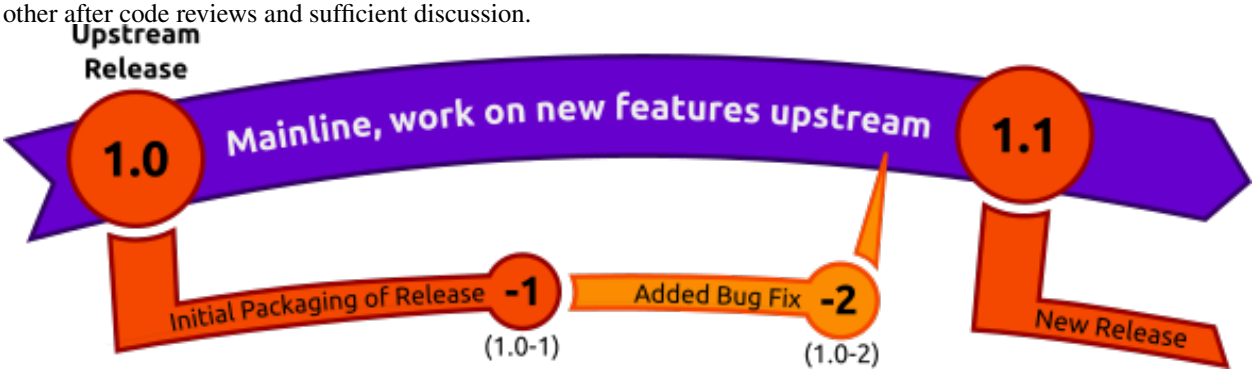
Another important tool regarding communication is bug reports. Whenever a defect is found in a package or piece of infrastructure, a bug report is filed in Launchpad. All information is collected in that report and its importance, status and assignee updated when necessary. This makes it an effective tool to stay on top of bugs in a package or project and organise the workload.

Most of the software available through Ubuntu is not written by Ubuntu developers themselves. Most of it is written by developers of other Open Source projects and then integrated into Ubuntu. These projects are called "Upstreams", because their source code flows into Ubuntu, where we "just" integrate it. The relationship to Upstreams is critically important to Ubuntu. It is not just code that Ubuntu gets from Upstreams, but it is also that Upstreams get users, bug reports and patches from Ubuntu (and other distributions).

The most important Upstream for Ubuntu is Debian. Debian is the distribution that Ubuntu is based on and many of the design decisions regarding the packaging infrastructure are made there. Traditionally, Debian has always had dedicated maintainers for every single package or dedicated maintenance teams. In Ubuntu there are teams that have an interest in a subset of packages too, and naturally every developer has a special area of expertise, but participation (and upload rights) generally is open to everyone who demonstrates ability and willingness.

Getting a change into Ubuntu as a new contributor is not as daunting as it seems and can be a very rewarding experience. It is not only about learning something new and exciting, but also about sharing the solution and solving a problem for millions of users out there.

Open Source Development happens in a distributed world with different goals and different areas of focus. For example there might be the case that a particular Upstream is interested in working on a new big feature while Ubuntu, because of the tight release schedule, is interested in shipping a solid version with just an additional bug fix. That is why we make use of "Distributed Development", where code is being worked on in various branches that are merged with each other after code reviews and sufficient discussion.



In the example mentioned above it would make sense to ship Ubuntu with the existing version of the project, add the bugfix, get it into Upstream for their next release and ship that (if suitable) in the next Ubuntu release. It would be the best possible compromise and a situation where everybody wins.

To fix a bug in Ubuntu, you would first get the source code for the package, then work on the fix, document it so it is easy to understand for other developers and users, then build the package to test it. After you have tested it, you can easily propose the change to be included in the current Ubuntu development release. A developer with upload rights will review it for you and then get it integrated into Ubuntu.

When trying to find a solution it is usually a good idea to check with Upstream and see if the problem (or a possible solution) is known already and, if not, do your best to make the solution a concerted effort.

Additional steps might involve getting the change backported to an older, still supported version of Ubuntu and forwarding it to Upstream.

The most important requirements for success in Ubuntu development are: having a knack for "making things work again," not being afraid to read documentation and ask questions, being a team player and enjoying some detective work.

Good places to ask your questions are `ubuntu-motu@lists.ubuntu.com` and `#ubuntu-motu` on `freenode`.. You will easily find a lot of new friends and people with the same passion that you have: making the world a better place by making better Open Source software.

1.2 Getting Set Up

There are a number of things you need to do to get started developing for Ubuntu. This article is designed to get your computer set up so that you can start working with packages, and upload your packages to Ubuntu's hosting platform, Launchpad. Here's what we'll cover:

- Installing packaging-related software. This includes:
  - Ubuntu-specific packaging utilities
  - Encryption software so your work can be verified as being done by you
  - Additional encryption software so you can securely transfer files
- Creating and configuring your account on Launchpad
- Setting up your development environment to help you do local builds of packages, interact with other developers, and propose your changes on Launchpad.

Note: It is advisable to do packaging work using the current development version of Ubuntu. Doing so will allow you to test changes in the same environment where those changes will actually be applied and used.

Don't want to install the latest development version of Ubuntu? Spin up an LXD container.

1.2.1 Install basic packaging software

There are a number of tools that will make your life as an Ubuntu developer much easier. You will encounter these tools later in this guide. To install most of the tools you will need run this command:

```
$ sudo apt install gnupg pbuilder ubuntu-dev-tools apt-file
```

This command will install the following software:

- `gnupg` – GNU Privacy Guard contains tools you will need to create a cryptographic key with which you will sign files you want to upload to Launchpad.
- `pbuilder` – a tool to do reproducible builds of a package in a clean and isolated environment.

- `ubuntu-dev-tools` (and `devscripts`, a direct dependency) – a collection of tools that make many packaging tasks easier.
- `apt-file` provides an easy way to find the binary package that contains a given file.

Create your GPG key

GPG stands for GNU Privacy Guard and it implements the OpenPGP standard which allows you to sign and encrypt messages and files. This is useful for a number of purposes. In our case it is important that you can sign files with your key so they can be identified as something that you worked on. If you upload a source package to Launchpad, it will only accept the package if it can absolutely determine who uploaded the package.

To generate a new GPG key, run:

```
$ gpg --gen-key
```

GPG will first ask you which kind of key you want to generate. Choosing the default (RSA and DSA) is fine. Next it will ask you about the keysize. The default (currently 2048) is fine, but 4096 is more secure. Afterwards, it will ask you if you want it to expire the key at some stage. It is safe to say "0", which means the key will never expire. The last questions will be about your name and email address. Just pick the ones you are going to use for Ubuntu development here, you can add additional email addresses later on. Adding a comment is not necessary. Then you will have to set a passphrase, choose a safe one (a passphrase is just a password which is allowed to include spaces).

Now GPG will create a key for you, which can take a little bit of time; it needs random bytes, so if you give the system some work to do it will be just fine. Move the cursor around, type some paragraphs of random text, load some web page.

Once this is done, you will get a message similar to this one:

```
pub   4096R/43CDE61D 2010-12-06
      Key fingerprint = 5C28 0144 FB08 91C0 2CF3  37AC 6F0B F90F 43CD E61D
uid                  Daniel Holbach <dh@mailempfang.de>
sub   4096R/51FBE68C 2010-12-06
```

In this case `43CDE61D` is the *key ID*.

Next, you need to upload the public part of your key to a keyserver so the world can identify messages and files as yours. To do so, enter:

```
$ gpg --send-keys --keyserver keyserver.ubuntu.com <KEY ID>
```

This will send your key to the Ubuntu keyserver, but a network of keyservers will automatically sync the key between themselves. Once this syncing is complete, your signed public key will be ready to verify your contributions around the world.

Create your SSH key

SSH stands for *Secure Shell*, and it is a protocol that allows you to exchange data in a secure way over a network. It is common to use SSH to access and open a shell on another computer, and to use it to securely transfer files. For our purposes, we will mainly be using SSH to securely upload source packages to Launchpad.

To generate an SSH key, enter:

```
$ ssh-keygen -t rsa
```

The default file name usually makes sense, so you can just leave it as it is. For security purposes, it is highly recommended that you use a passphrase.

Set up pbuilder

`pbuilder` allows you to build packages locally on your machine. It serves a couple of purposes:

- The build will be done in a minimal and clean environment. This helps you make sure your builds succeed in a reproducible way, but without modifying your local system
- There is no need to install all necessary *build dependencies* locally
- You can set up multiple instances for various Ubuntu and Debian releases

Setting `pbuilder` up is very easy, run:

```
$ pbuilder-dist <release> create
```

where <release> is for example *xenial*, *zesty*, *artful* or in the case of Debian maybe *sid* or *buster*. This will take a while as it will download all the necessary packages for a "minimal installation". These will be cached though.

1.2.2 Get set up to work with Launchpad

With a basic local configuration in place, your next step will be to configure your system to work with Launchpad. This section will focus on the following topics:

- What Launchpad is and creating a Launchpad account
- Uploading your GPG and SSH keys to Launchpad
- Configure your shell to recognize you (for putting your name in changelogs)

About Launchpad

Launchpad is the central piece of infrastructure we use in Ubuntu. It not only stores our packages and our code, but also things like translations, bug reports, and information about the people who work on Ubuntu and their team memberships. You will also use Launchpad to publish your proposed fixes, and get other Ubuntu developers to review and sponsor them.

You will need to register with Launchpad and provide a minimal amount of information. This will allow you to download and upload code, submit bug reports, and more.

Besides hosting Ubuntu, Launchpad can host any Free Software project. For more information see the Launchpad Help wiki.

Get a Launchpad account

If you don't already have a Launchpad account, you can easily create one. If you have a Launchpad account but cannot remember your Launchpad id, you can find this out by going to https://launchpad.net/~ and looking for the part after the ~ in the URL.

Launchpad's registration process will ask you to choose a display name. It is encouraged for you to use your real name here so that your Ubuntu developer colleagues will be able to get to know you better.

When you register a new account, Launchpad will send you an email with a link you need to open in your browser in order to verify your email address. If you don't receive it, check in your spam folder.

The new account help page on Launchpad has more information about the process and additional settings you can change.

Upload your GPG key to Launchpad

First, you will need to get your fingerprint and key ID.

To find about your GPG fingerprint, run:

```
$ gpg --fingerprint email@address.com
```

and it will print out something like:

```
pub   4096R/43CDE61D 2010-12-06
      Key fingerprint = 5C28 0144 FB08 91C0 2CF3  37AC 6F0B F90F 43CD E61D
uid                  Daniel Holbach <dh@mailempfang.de>
sub   4096R/51FBE68C 2010-12-06
```

Then run this command to submit your key to Ubuntu keyserver:

```
$ gpg --keyserver keyserver.ubuntu.com --send-keys 43CDE61D
```

where `43CDE61D` should be replaced by your key ID (which is in the first line of output of the previous command). Now you can import your key to Launchpad.

Head to https://launchpad.net/~/+editpgpkeys and copy the "Key fingerprint" into the text box. In the case above this would be `5C28 0144 FB08 91C0 2CF3 37AC 6F0B F90F 43CD E61D`. Now click on "Import Key".

Launchpad will use the fingerprint to check the Ubuntu key server for your key and, if successful, send you an encrypted email asking you to confirm the key import. Check your email account and read the email that Launchpad sent you. *If your email client supports OpenPGP encryption, it will prompt you for the password you chose for the key when GPG generated it. Enter the password, then click the link to confirm that the key is yours.*

Launchpad encrypts the email, using your public key, so that it can be sure that the key is yours. If you are using Thunderbird, the default Ubuntu email client, you can install the Enigmail plugin to easily decrypt the message. If your email software does not support OpenPGP encryption, copy the encrypted email's contents, type `gpg` in your terminal, then paste the email contents into your terminal window.

Back on the Launchpad website, use the Confirm button and Launchpad will complete the import of your OpenPGP key.

Find more information at https://help.launchpad.net/YourAccount/ImportingYourPGPKey

Upload your SSH key to Launchpad

Open https://launchpad.net/~/+editsshkeys in a web browser, also open `~/.ssh/id_rsa.pub` in a text editor. This is the public part of your SSH key, so it is safe to share it with Launchpad. Copy the contents of the file and paste them into the text box on the web page that says "Add an SSH key". Now click "Import Public Key".

For more information on this process, visit the creating an SSH keypair page on Launchpad.

Configure your shell

The Debian/Ubuntu packaging tools need to learn about you as well in order to properly credit you in the changelog. Simply open your *~/.bashrc* in a text editor and add something like this to the bottom of it:

```
export DEBFULLNAME="Bob Dobbs"
export DEBEMAIL="subgenius@example.com"
```

Now save the file and either restart your terminal or run:


```
$ source ~/.bashrc
```

(If you do not use the default shell, which is *bash*, please edit the configuration file for that shell accordingly.)

1.3 Fixing a bug in Ubuntu

1.3.1 Introduction

If you followed the instructions to *get set up with Ubuntu Development*, you should be all set and ready to go.

As you can see in the image above, there is no surprises in the process of fixing bugs in Ubuntu: you found a problem, you get the code, work on the fix, test it, push your changes to Launchpad and ask for it to be reviewed and merged. In this guide we will go through all the necessary steps one by one.

1.3.2 Finding the problem

There are a lot of different ways to find things to work on. It might be a bug report you are encountering yourself (which gives you a good opportunity to test the fix), or a problem you noted elsewhere, maybe in a bug report.

Take a look at the bitesize bugs in Launchpad, and that might give you an idea of something to work on. It might also interest you to look at the bugs triaged by the Ubuntu One Hundred Papercuts team.

1.3.3 Figuring out what to fix

If you don't know the source package containing the code that has the problem, but you do know the path to the affected program on your system, you can discover the source package that you'll need to work on.

Let's say you've found a bug in Bumprace, a racing game. The Bumprace application can be started by running `/usr/bin/bumprace` on the command line. To find the binary package containing this application, use this command:

```
$ apt-file find /usr/bin/bumprace
```

This would print out:

```
bumprace: /usr/bin/bumprace
```

Note that the part preceding the colon is the binary package name. It's often the case that the source package and binary package will have different names. This is most common when a single source package is used to build multiple different binary packages. To find the source package for a particular binary package, type:


```
$ apt-cache showsrc bumprace | grep ^Package:
Package: bumprace
$ apt-cache showsrc tomboy | grep ^Package:
Package: tomboy
```

`apt-cache` is part of the standard installation of Ubuntu.

1.3.4 Confirming the problem

Once you have figured out which package the problem is in, it's time to confirm that the problem exists.

Let's say the package `bumprace` does not have a homepage in its package description. As a first step you would check if the problem is not solved already. This is easy to check, either take a look at Software Center or run:

```
apt-cache show bumprace
```

The output should be similar to this:

```
Package: bumprace
Priority: optional
Section: universe/games
Installed-Size: 136
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
XNBC-Original-Maintainer: Christian T. Steigies <cts@debian.org>
Architecture: amd64
Version: 1.5.4-1
Depends: bumprace-data, libc6 (>= 2.4), libsdl-image1.2 (>= 1.2.10),
         libsdl-mixer1.2, libsdl1.2debian (>= 1.2.10-1)
Filename: pool/universe/b/bumprace/bumprace_1.5.4-1_amd64.deb
Size: 38122
MD5sum: 48c943863b4207930d4a2228cedc4a5b
SHA1: 73bad0892be471bbc471c7a99d0b72f0d0a4babc
SHA256: 64ef9a45b75651f57dc76aff5b05dd7069db0c942b479c8ab09494e762ae69fc
Description-en: 1 or 2 players race through a multi-level maze
 In BumpRacer, 1 player or 2 players (team or competitive) choose among 4
 vehicles and race through a multi-level maze. The players must acquire
 bonuses and avoid traps and enemy fire in a race against the clock.
 For more info, see the homepage at http://www.linux-games.com/bumprace/
Description-md5: 3225199d614fba85ba2bc66d5578ff15
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Origin: Ubuntu
```

A counter-example would be `gedit`, which has a homepage set:

```
$ apt-cache show gedit | grep ^Homepage
Homepage: http://www.gnome.org/projects/gedit/
```

Sometimes you will find that a particular problem you are looking into is already fixed. To avoid wasting efforts and duplicating work it makes sense to first do some detective work.

1.3.5 Research bug situation

First we should check if a bug for the problem exists in Ubuntu already. Maybe somebody is working on a fix already, or we can contribute to the solution somehow. For Ubuntu we have a quick look at https://bugs.launchpad.net/ubuntu/+source/bumprace and there is no open bug with our problem there.

Note: For Ubuntu the URL `https://bugs.launchpad.net/ubuntu/+source/<package>` should always take to the bug page of the source package in question.

For Debian, which is the major source for Ubuntu's packages, we have a look at http://bugs.debian.org/src:bumprace and can't find a bug report for our problem either.

Note: For Debian the URL `http://bugs.debian.org/src:<package>` should always take to the bug page of the source package in question.

The problem we are working on is special as it only concerns the packaging-related bits of `bumprace`. If it was a problem in the source code it would be helpful to also check the Upstream bug tracker. This is unfortunately often different for every package you have a look at, but if you search the web for it, you should in most cases find it pretty easily.

1.3.6 Offering help

If you found an open bug and it is not assigned to somebody and you are in a position to fix it, you should comment on it with your solution. Be sure to include as much information as you can: Under which circumstances does the bug occur? How did you fix the problem? Did you test your solution?

If no bug report has been filed, you can file a bug for it. What you might want to bear in mind is: Is the issue so small that just asking for somebody to commit it is good enough? Did you manage to only partially fix the issue and you want to at least share your part of it?

It is great if you can offer help and will surely be appreciated.

1.3.7 Getting the code

Once you know the source package to work on, you will want to get a copy of the code on your system, so that you can debug it. The ubuntu-dev-tools package has a tool called `pull-lp-source` that a developer can use to grab the source code for any package. For example, to grab the source code for the tomboy package in `xenial`, you can type this:

```
$ pull-lp-source bumprace xenial
```

If you do not specify a release such as `xenial`, it will automatically get the package from the development version.

Once you've got a local clone of the source package, you can investigate the bug, create a fix, generate a debdiff, and attach your debdiff to a bug report for other developers to review. We'll describe specifics in the next sections.

1.3.8 Work on a fix

There are entire books written about finding bugs, fixing them, testing them, etc. If you are completely new to programming, try to fix easy bugs such as obvious typos first. Try to keep changes as minimal as possible and document your change and assumptions clearly.

Before working on a fix yourself, make sure to investigate if nobody else has fixed it already or is currently working on a fix. Good sources to check are:

- Upstream (and Debian) bug tracker (open and closed bugs),
- Upstream revision history (or newer release) might have fixed the problem,
- bugs or package uploads of Debian or other distributions.

You may want to create a patch which includes the fix. The command `edit-patch` is a simple way to add a patch to a package. Run:

```
$ edit-patch 99-new-patch
```

This will copy the packaging to a temporary directory. You can now edit files with a text editor or apply patches from upstream, for example:

```
$ patch -p1 < ../bugfix.patch
```

After editing the file type `exit` or press `control-d` to quit the temporary shell. The new patch will have been added into `debian/patches`.

You must then add a header to your patch containing meta information so that other developers can know the purpose of the patch and where it came from. To get the template header that you can edit to reflect what the patch does, type this:

```
$ quilt header --dep3 -e
```

This will open the template in a text editor. Follow the template and make sure to be thorough so you get all the details necessary to describe the patch.

In this specific case, if you just want to edit `debian/control`, you do not need a patch. Put `Homepage: http://www.linux-games.com/bumprace/` at the end of the first section and the bug should be fixed.

Documenting the fix

It is very important to document your change sufficiently so developers who look at the code in the future won't have to guess what your reasoning was and what your assumptions were. Every Debian and Ubuntu package source includes `debian/changelog`, where changes of each uploaded package are tracked.

The easiest way to update this is to run:

```
$ dch -i
```

This will add a boilerplate changelog entry for you and launch an editor where you can fill in the blanks. An example of this could be:

```
specialpackage (1.2-3ubuntu4) trusty; urgency=low

  * debian/control: updated description to include frobnicator (LP: #123456)

 -- Emma Adams <emma.adams@isp.com>  Sat, 17 Jul 2010 02:53:39 +0200
```

`dch` should fill out the first and last line of such a changelog entry for you already. Line 1 consists of the source package name, the version number, which Ubuntu release it is uploaded to, the urgency (which almost always is 'low'). The last line always contains the name, email address and timestamp (in **RFC 5322** format) of the change.

With that out of the way, let's focus on the actual changelog entry itself: it is very important to document:

1. Where the change was done.
2. What was changed.
3. Where the discussion of the change happened.

In our (very sparse) example the last point is covered by `(LP: #123456)` which refers to Launchpad bug 123456. Bug reports or mailing list threads or specifications are usually good information to provide as a rationale for a change. As a bonus, if you use the `LP: #<number>` notation for Launchpad bugs, the bug will be automatically closed when the package is uploaded to Ubuntu.

In order to get it sponsored in the next section, you need to file a bug report in Launchpad (if there isn't one already, if there is, use that) and explain why your fix should be included in Ubuntu. For example, for tomboy, you would file a bug here (edit the URL to reflect the package you have a fix for). Once a bug is filed explaining your changes, put that bug number in the changelog.

1.3.9 Testing the fix

To build a test package with your changes, run these commands:

```
$ debuild -S -d -us -uc
$ pbuilder-dist <release> build ../<package>_<version>.dsc
```

This will create a source package from the branch contents (`-us -uc` will just omit the step to sign the source package and `-d` will skip the step where it checks for build dependencies, pbuilder will take care of that) and `pbuilder-dist` will build the package from source for whatever `release` you choose.

Note: If `debuild` errors out with "Version number suggests Ubuntu changes, but Maintainer: does not have Ubuntu address" then run the `update-maintainer` command (from ubuntu-dev-tools) and it will automatically fix this for you. This happens because in Ubuntu, all Ubuntu Developers are responsible for all Ubuntu packages, while in Debian, packages have maintainers.

In this case with bumprace, run this to view the package information:

```
$ dpkg -I ~/pbuilder/*_result/bumprace_*.deb
```

As expected, there should now be a `Homepage:` field.

Note: In a lot of cases you will have to actually install the package to make sure it works as expected. Our case is a lot easier. If the build succeeded, you will find the binary packages in `~/pbuilder/<release>_result`. Install them via `sudo dpkg -i <package>.deb` or by double-clicking on them in your file manager.

1.3.10 Submitting the fix and getting it included

With the changelog entry written and saved, run `debuild` one more time:

```
$ debuild -S -d
```

and this time it will be signed and you are now ready to get your diff to submit to get sponsored.

In a lot of cases, Debian would probably like to have the patch as well (doing this is best practice to make sure a wider audience gets the fix). So, you should submit the patch to Debian, and you can do that by simply running this:

```
$ submittodebian
```

This will take you through a series of steps to make sure the bug ends up in the correct place. Be sure to review the diff again to make sure it does not include random changes you made earlier.

Communication is important, so when you add some more description to it to the inclusion request, be friendly, explain it well.

If everything went well you should get a mail from Debian's bug tracking system with more information. This might sometimes take a few minutes.

It might be beneficial to just get it included in Debian and have it flow down to Ubuntu, in which case you would not follow the below process. But, sometimes in the case of security updates and updates for stable releases, the fix is already in Debian (or ignored for some reason) and you would follow the below process. If you are doing such updates,

please read our *Security and stable release updates* article. Other cases where it is acceptable to wait to submit patches to Debian are Ubuntu-only packages not building correctly, or Ubuntu-specific problems in general.

But if you're going to submit your fix to Ubuntu, now it's time to generate a "debdiff", which shows the difference between two Debian packages. The name of the command used to generate one is also `debdiff`. It is part of the `devscripts` package. See `man debdiff` for all the details. To compare two source packages, pass the two dsc files as arguments:

```
$ debdiff <package_name>_1.0-1.dsc <package_name>_1.0-1ubuntu1.dsc
```

In this case, `debdiff` the dsc you downloaded with `pull-lp-source` and the new dsc file you generated. This will generate a patch that your sponsor can then apply locally (by using `patch -p1 < /path/to/debdiff`). In this case, pipe the output of the debdiff command to a file that you can then attach to the bug report:

```
$ debdiff <package_name>_1.0-1.dsc <package_name>_1.0-1ubuntu1.dsc > 1-1.0-1ubuntu1.debdiff
```

The format shown in `1-1.0-1ubuntu1.debdiff` shows:

1. `1-` tells the sponsor that this is the first revision of your patch. Nobody is perfect, and sometimes follow-up patches need to be provided. This makes sure that if your patch needs work, that you can keep a consistent naming scheme.
2. `1.0-1ubuntu1` shows the new version being used. This makes it easy to see what the new version is.
3. `.debdiff` is an extension that makes it clear that it is a debdiff.

While this format is optional, it works well and you can use this.

Next, go to the bug report, make sure you are logged into Launchpad, and click "Add attachment or patch" under where you would add a new comment. Attach the debdiff, and leave a comment telling your sponsor how this patch can be applied and the testing you have done. An example comment can be:

```
This is a debdiff for Artful applicable to 1.0-1. I built this in pbuilder
and it builds successfully, and I installed it, the patch works as intended.
```

Make sure you mark it as a patch (the Ubuntu Sponsors team will automatically be subscribed) and that you are subscribed to the bug report. You will then receive a review anywhere between several housr from submitting the patch to several weeks. If it takes longer than that, please join `#ubuntu-motu` on `freenode` and mention it there. Stick around until you get an answer from someone, and they can guide you as to what to do next.

Once you have received a review, your patch was either uploaded, your patch needs work, or is rejected for some other reason (possibly the fix is not fit for Ubuntu or should go to Debian instead). If your patch needs work, follow the same steps and submit a follow-up patch on the bug report, otherwise submit to Debian as shown above.

Remember: good places to ask your questions are `ubuntu-motu@lists.ubuntu.com` and `#ubuntu-motu` on freenode. You will easily find a lot of new friends and people with the same passion that you have: making the world a better place by making better Open Source software.

1.3.11 Additional considerations

If you find a package and find that there are a couple of trivial things you can fix at the same time, do it. This will speed up review and inclusion.

If there are multiple big things you want to fix, it might be advisable to send individual patches or merge proposals instead. If there are individual bugs filed for the issues already, this makes it even easier.

1.4 Packaging New Software

While there are thousands of packages in the Ubuntu archive, there are still a lot nobody has gotten to yet. If there is an exciting new piece of software that you feel needs wider exposure, maybe you want to try your hand at creating a package for Ubuntu or a PPA. This guide will take you through the steps of packaging new software.

You will want to read the *Getting Set Up* article first in order to prepare your development environment.

1.4.1 Checking the Program

The first stage in packaging is to get the released tar from upstream (we call the authors of applications "upstream") and check that it compiles and runs.

This guide will take you through packaging a simple application called GNU Hello which has been posted on GNU.org.

Download GNU Hello:

```
$ wget -O hello-2.10.tar.gz "http://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz"
```

Now uncompress it:

```
$ tar xf hello-2.10.tar.gz
$ cd hello-2.10
```

This application uses the autoconf build system so we want to run `./configure` to prepare for compilation.

This will check for the required build dependencies. As `hello` is a simple example, `build-essential` should provide everything we need. For more complex programs, the command will fail if you do not have the needed libraries and development files. Install the needed packages and repeat until the command runs successfully.:

```
$ ./configure
```

Now you can compile the source:

```
$ make
```

If compilation completes successfully you can install and run the program:

```
$ sudo make install
$ hello
```

1.4.2 Starting a Package

`bzr-builddeb` includes a plugin to create a new package from a template. The plugin is a wrapper around the `dh_make` command. Run the command providing the package name, version number, and path to the upstream tarball:

```
$ sudo apt-get install dh-make bzr-builddeb
$ cd ..
$ bzr dh-make hello 2.10 hello-2.10.tar.gz
```

When it asks what type of package type `s` for single binary. This will import the code into a branch and add the `debian/` packaging directory. Have a look at the contents. Most of the files it adds are only needed for specialist packages (such as Emacs modules) so you can start by removing the optional example files:

```
$ cd hello/debian
$ rm *ex *EX
```


You should now customise each of the files.

In `debian/changelog` change the version number to an Ubuntu version: `2.10-0ubuntu1` (upstream version 2.10, Debian version 0, Ubuntu version 1). Also change `unstable` to the current development Ubuntu release such as `trusty`.

Much of the package building work is done by a series of scripts called `debhelper`. The exact behaviour of `debhelper` changes with new major versions, the compat file instructs `debhelper` which version to act as. You will generally want to set this to the most recent version which is `9`.

`control` contains all the metadata of the package. The first paragraph describes the source package. The second and following paragraphs describe the binary packages to be built. We will need to add the packages needed to compile the application to `Build-Depends:`. For `hello`, make sure that it includes at least:

```
Build-Depends: debhelper (>= 9)
```

You will also need to fill in a description of the program in the `Description:` field.

`copyright` needs to be filled in to follow the licence of the upstream source. According to the hello/COPYING file this is GNU GPL 3 or later.

`docs` contains any upstream documentation files you think should be included in the final package.

`README.source` and `README.Debian` are only needed if your package has any non-standard features, we don't so you can delete them.

`source/format` can be left as is, this describes the version format of the source package and should be `3.0 (quilt)`.

`rules` is the most complex file. This is a Makefile which compiles the code and turns it into a binary package. Fortunately most of the work is automatically done these days by `debhelper 7` so the universal `%` Makefile target just runs the `dh` script which will run everything needed.

All of these file are explained in more detail in the *overview of the debian directory* article.

Finally commit the code to your packaging branch:

```
$ bzr add debian/source/format
$ bzr commit -m "Initial commit of Debian packaging."
```

1.4.3 Building the package

Now we need to check that our packaging successfully compiles the package and builds the .deb binary package:

```
$ bzr builddeb -- -us -uc
$ cd ../../
```

`bzr builddeb` is a command to build the package in its current location. The `-us -uc` tell it there is no need to GPG sign the package. The result will be placed in `...`

You can view the contents of the package with:

```
$ lesspipe hello_2.10-0ubuntu1_amd64.deb
```

Install the package and check it works (later you will be able to uninstall it using `sudo apt-get remove hello` if you want):

```
$ sudo dpkg --install hello_2.10-0ubuntu1_amd64.deb
```

You can also install all packages at once using:


```
$ sudo debi
```

1.4.4 Next Steps

Even if it builds the .deb binary package, your packaging may have bugs. Many errors can be automatically detected by our tool `lintian` which can be run on the source .dsc metadata file, .deb binary packages or .changes file:

```
$ lintian hello_2.10-0ubuntu1.dsc
$ lintian hello_2.10-0ubuntu1_amd64.deb
```

To see verbose description of the problems use `--info` lintian flag or `lintian-info` command.

For Python packages, there is also a `lintian4python` tool that provides some additional lintian checks.

After making a fix to the packaging you can rebuild using `-nc` "no clean" without having to build from scratch:

```
$ bzr builddeb -- -nc -us -uc
```

Having checked that the package builds locally you should ensure it builds on a clean system using `pbuilder`. Since we are going to upload to a PPA (Personal Package Archive) shortly, this upload will need to be *signed* to allow Launchpad to verify that the upload comes from you (you can tell the upload will be signed because the `-us` and `-uc` flags are not passed to `bzr builddeb` like they were before). For signing to work you need to have set up GPG. If you haven't set up `pbuilder-dist` or GPG yet, *do so now*:

```
$ bzr builddeb -S
$ cd ../build-area
$ pbuilder-dist trusty build hello_2.10-0ubuntu1.dsc
```

When you are happy with your package you will want others to review it. You can upload the branch to Launchpad for review:

```
$ bzr push lp:~<lp-username>/+junk/hello-package
```

Uploading it to a PPA will ensure it builds and give an easy way for you and others to test the binary packages. You will need to set up a PPA in Launchpad and then upload with `dput`:

```
$ dput ppa:<lp-username>/<ppa-name> hello_2.10-0ubuntu1.changes
```

You can ask for reviews in `#ubuntu-motu` IRC channel, or on the MOTU mailing list. There might also be a more specific team you could ask such as the GNU team for more specific questions.

1.4.5 Submitting for inclusion

There are a number of paths that a package can take to enter Ubuntu. In most cases, going through Debian first can be the best path. This way ensures that your package will reach the largest number of users as it will be available in not just Debian and Ubuntu but all of their derivatives as well. Here are some useful links for submitting new packages to Debian:

- Debian Mentors FAQ - debian-mentors is for the mentoring of new and prospective Debian Developers. It is where you can find a sponsor to upload your package to the archive.
- Work-Needing and Prospective Packages - Information on how to file "Intent to Package" and "Request for Package" bugs as well as list of open ITPs and RFPs.
- Debian Developer's Reference, 5.1. New packages - The entire document is invaluable for both Ubuntu and Debian packagers. This section documents processes for submitting new packages.

In some cases, it might make sense to go directly into Ubuntu first. For instance, Debian might be in a freeze making it unlikely that your package will make it into Ubuntu in time for the next release. This process is documented on the "New Packages" section of the Ubuntu wiki.

1.4.6 Screenshots

Once you have uploaded a package to debian, you should add screenshots to allow propective users to see what the program is like. These should be uploaded to http://screenshots.debian.net/upload .

1.5 Security and Stable Release Updates

1.5.1 Fixing a Security Bug in Ubuntu

Introduction

Fixing security bugs in Ubuntu is not really any different than *fixing a regular bug in Ubuntu*, and it is assumed that you are familiar with patching normal bugs. To demonstrate where things are different, we will be updating the dbus package in Ubuntu 12.04 LTS (Precise Pangolin) for a security update.

Obtaining the source

In this example, we already know we want to fix the dbus package in Ubuntu 12.04 LTS (Precise Pangolin). So first you need to determine the version of the package you want to download. We can use the `rmadison` to help with this:

```
$ rmadison dbus | grep precise
dbus | 1.4.18-1ubuntu1   | precise          | source, amd64, armel, armhf, i386, powerpc
dbus | 1.4.18-1ubuntu1.4 | precise-security | source, amd64, armel, armhf, i386, powerpc
dbus | 1.4.18-1ubuntu1.4 | precise-updates  | source, amd64, armel, armhf, i386, powerpc
```

Typically you will want to choose the highest version for the release you want to patch that is not in -proposed or -backports. Since we are updating Precise's dbus, you'll download 1.4.18-1ubuntu1.4 from precise-updates:

```
$ bzr branch ubuntu:precise-updates/dbus
```

Patching the source

Now that we have the source package, we need to patch it to fix the vulnerability. You may use whatever patch method that is appropriate for the package, but this example will use `edit-patch` (from the ubuntu-dev-tools package). `edit-patch` is the easiest way to patch packages and it is basically a wrapper around every other patch system you can imagine.

To create your patch using `edit-patch`:

```
$ cd dbus
$ edit-patch 99-fix-a-vulnerability
```

This will apply the existing patches and put the packaging in a temporary directory. Now edit the files needed to fix the vulnerability. Often upstream will have provided a patch so you can apply that patch:

```
$ patch -p1 < /home/user/dbus-vulnerability.diff
```

After making the necessary changes, you just hit Ctrl-D or type exit to leave the temporary shell.

Formatting the changelog and patches

After applying your patches you will want to update the changelog. The `dch` command is used to edit the `debian/changelog` file and `edit-patch` will launch `dch` automatically after un-applying all the patches. If you are not using `edit-patch`, you can launch `dch -i` manually. Unlike with regular patches, you should use the following format (note the distribution name uses precise-security since this is a security update for Precise) for security updates:

```
dbus (1.4.18-2ubuntu1.5) precise-security; urgency=low

  * SECURITY UPDATE: [DESCRIBE VULNERABILITY HERE]
    - debian/patches/99-fix-a-vulnerability.patch: [DESCRIBE CHANGES HERE]
    - [CVE IDENTIFIER]
    - [LINK TO UPSTREAM BUG OR SECURITY NOTICE]
    - LP: #[BUG NUMBER]
...
```

Update your patch to use the appropriate patch tags. Your patch should have at a minimum the Origin, Description and Bug-Ubuntu tags. For example, edit debian/patches/99-fix-a-vulnerability.patch to have something like:

```
## Description: [DESCRIBE VULNERABILITY HERE]
## Origin/Author: [COMMIT ID, URL OR EMAIL ADDRESS OF AUTHOR]
## Bug: [UPSTREAM BUG URL]
## Bug-Ubuntu: https://launchpad.net/bugs/[BUG NUMBER]
Index: dbus-1.4.18/dbus/dbus-marshal-validate.c
...
```

Multiple vulnerabilities can be fixed in the same security upload; just be sure to use different patches for different vulnerabilities.

Test and Submit your work

At this point the process is the same as for *fixing a regular bug in Ubuntu*. Specifically, you will want to:

1. Build your package and verify that it compiles without error and without any added compiler warnings
2. Upgrade to the new version of the package from the previous version
3. Test that the new package fixes the vulnerability and does not introduce any regressions
4. Submit your work via a Launchpad merge proposal and file a Launchpad bug being sure to mark the bug as a security bug and to subscribe `ubuntu-security-sponsors`

If the security vulnerability is not yet public then do not file a merge proposal and ensure you mark the bug as private.

The filed bug should include a Test Case, i.e. a comment which clearly shows how to recreate the bug by running the old version then how to ensure the bug no longer exists in the new version.

The bug report should also confirm that the issue is fixed in Ubuntu versions newer than the one with the proposed fix (in the above example newer than Precise). If the issue is not fixed in newer Ubuntu versions you should prepare updates for those versions too.

1.5.2 Stable Release Updates

We also allow updates to releases where a package has a high impact bug such as a severe regression from a previous release or a bug which could cause data loss. Due to the potential for such updates to themselves introduce bugs we only allow this where the change can be easily understood and verified.

The process for Stable Release Updates is just the same as the process for security bugs except you should subscribe `ubuntu-sru` to the bug.

The update will go into the `proposed` archive (for example `precise-proposed`) where it will need to be checked that it fixes the problem and does not introduce new problems. After a week without reported problems it can be moved to `updates`.

See the Stable Release Updates wiki page for more information.

1.6 Patches to Packages

Sometimes, Ubuntu package maintainers have to change the upstream source code in order to make it work properly on Ubuntu. Examples include, patches to upstream that haven't yet made it into a released version, or changes to the upstream's build system needed only for building it on Ubuntu. We could change the upstream source code directly, but doing this makes it more difficult to remove the patches later when upstream has incorporated them, or extract the change to submit to the upstream project. Instead, we keep these changes as separate patches, in the form of diff files.

There are a number of different ways of handling patches in Debian packages, fortunately we are standardizing on one system, Quilt, which is now used by most packages.

Let's look at an example package, `kamoso` in Trusty:

```
$ bzr branch ubuntu:trusty/kamoso
```

The patches are kept in `debian/patches`. This package has one patch `kubuntu_01_fix_qmax_on_armel.diff` to fix a compile failure on ARM. The patch has been given a name to describe what it does, a number to keep the patches in order (two patches can overlap if they change the same file) and in this case the Kubuntu team adds their own prefix to show the patch comes from them rather than from Debian.

The order of patches to apply is kept in `debian/patches/series`.

1.6.1 Patches with Quilt

Before working with Quilt you need to tell it where to find the patches. Add this to your `~/.bashrc`:

```
export QUILT_PATCHES=debian/patches
```

And source the file to apply the new export:

```
$ . ~/.bashrc
```

By default all patches are applied already to UDD checkouts or downloaded packages. You can check this with:

```
$ quilt applied
kubuntu_01_fix_qmax_on_armel.diff
```

If you wanted to remove the patch you would run `pop`:

```
$ quilt pop
Removing patch kubuntu_01_fix_qmax_on_armel.diff
Restoring src/kamoso.cpp

No patches applied
```

And to apply a patch you use `push`:


```
$ quilt push
Applying patch kubuntu_01_fix_qmax_on_armel.diff
patching file src/kamoso.cpp

Now at patch kubuntu_01_fix_qmax_on_armel.diff
```

1.6.2 Adding a New Patch

To add a new patch you need to tell Quilt to create a new patch, tell it which files that patch should change, edit the files then refresh the patch:

```
$ quilt new kubuntu_02_program_description.diff
Patch kubuntu_02_program_description.diff is now on top
$ quilt add src/main.cpp
File src/main.cpp added to patch kubuntu_02_program_description.diff
$ sed -i "s,Webcam picture retriever,Webcam snapshot program,"
src/main.cpp
$ quilt refresh
Refreshed patch kubuntu_02_program_description.diff
```

The `quilt add` step is important, if you forget it the files will not end up in the patch.

The change will now be in `debian/patches/kubuntu_02_program_description.diff` and the `series` file will have had the new patch added to it. You should add the new file to the packaging:

```
$ bzr add debian/patches/kubuntu_02_program_description.diff
$ bzr add .pc/*
$ dch -i "Add patch kubuntu_02_program_description.diff to improve the program description"
$ bzr commit
```

Quilt keeps its metadata in the `.pc/` directory, so currently you need to add that to the packaging too. This should be improved in future.

As a general rule you should be careful adding patches to programs unless they come from upstream, there is often a good reason why that change has not already been made. The above example changes a user interface string for example, so it would break all translations. If in doubt, do ask the upstream author before adding a patch.

1.6.3 Patch Headers

We recommend that you tag every patch with DEP-3 headers by putting them at the top of patch file. Here are some headers that you can use:

Description Description of what the patch does. It is formatted like `Description` field in `debian/control`: first line is short description, starting with lowercase letter, the next lines are long description, indented with a space.

Author Who wrote the patch (i.e. "Jane Doe <packager@example.com>").

Origin Where this patch comes from (i.e. "upstream"), when *Author* is not present.

Bug-Ubuntu A link to Launchpad bug, a short form is preferred (like *https://bugs.launchpad.net/bugs/XXXXXXX*). If there are also bugs in upstream or Debian bugtrackers, add *Bug* or *Bug-Debian* headers.

Forwarded Whether the patch was forwarded upstream. Either "yes", "no" or "not-needed".

Last-Update Date of the last revision (in form "YYYY-MM-DD").

1.6.4 Upgrading to New Upstream Versions

To upgrade to the new version, you can use `bzr merge-upstream` command:

```
$ bzr merge-upstream --version 2.0.2 https://launchpad.net/ubuntu/+archive/primary/+files/kamoso_2.0
```

When you run this command, all patches will be unapplied, because they can become out of date. They might need to be refreshed to match the new upstream source or they might need to be removed altogether. To check for problems, apply the patches one at a time:

```
$ quilt push
Applying patch kubuntu_01_fix_qmax_on_armel.diff
patching file src/kamoso.cpp
Hunk #1 FAILED at 398.
1 out of 1 hunk FAILED -- rejects in file src/kamoso.cpp
Patch kubuntu_01_fix_qmax_on_armel.diff can be reverse-applied
```

If it can be reverse-applied this means the patch has been applied already by upstream, so we can delete the patch:

```
$ quilt delete kubuntu_01_fix_qmax_on_armel
Removed patch kubuntu_01_fix_qmax_on_armel.diff
```

Then carry on:

```
$ quilt push
Applied kubuntu_02_program_description.diff
```

It is a good idea to run refresh, this will update the patch relative to the changed upstream source:

```
$ quilt refresh
Refreshed patch kubuntu_02_program_description.diff
```

Then commit as usual:

```
$ bzr commit -m "new upstream version"
```

1.6.5 Making A Package Use Quilt

Modern packages use Quilt by default, it is built into the packaging format. Check in `debian/source/format` to ensure it says `3.0 (quilt)`.

Older packages using source format 1.0 will need to explicitly use Quilt, usually by including a makefile into `debian/rules`.

1.6.6 Configuring Quilt

You can use `~/.quiltrc` file to configure quilt. Here are some options that can be useful for using quilt with debian/packages:

```
# Set the patches directory
QUILT_PATCHES="debian/patches"
# Remove all useless formatting from the patches
QUILT_REFRESH_ARGS="-p ab --no-timestamps --no-index"
# The same for quilt diff command, and use colored output
QUILT_DIFF_ARGS="-p ab --no-timestamps --no-index --color=auto"
```


1.6.7 Other Patch Systems

Other patch systems used by packages include `dpatch` and `cdbs simple-patchsys`, these work similarly to Quilt by keeping patches in `debian/patches` but have different commands to apply, un-apply or create patches. You can find out which patch system is used by a package by using the `what-patch` command (from the `ubuntu-dev-tools` package). You can use `edit-patch`, shown in *previous chapters*, as a reliable way to work with all systems.

In even older packages changes will be included directly to sources and kept in the `diff.gz` source file. This makes it hard to upgrade to new upstream versions or differentiate between patches and is best avoided.

Do not change a package's patch system without discussing it with the Debian maintainer or relevant Ubuntu team. If there is no existing patch system then feel free to add Quilt.

1.7 Fixing FTBFS packages

Before a package can be used in Ubuntu, it has to build from source. If it fails this, it will probably wait in -proposed and will not be available in the Ubuntu archives. You can find a complete list of packages that are failing to build from source at http://qa.ubuntuwire.org/ftbfs/. There are 5 main categories shown on the page:

- Package failed to build (F): Something actually went wrong with the build process.
- Cancelled build (X): The build has been cancelled for some reason. These should probably be avoided to start with.
- Package is waiting on another package (M): This package is waiting on another package to either build, get updated, or (if the package is in main) one of it's dependancies is in the wrong part of the archive.
- Failure in the chroot (C): Part of the chroot failed, this is most likely fixed by a rebuild. Ask a developer to rebuild the package and that should fix it.
- Failed to upload (U): The package could not upload. This is usually just a case of asking for a rebuild, but check the build log first.

1.7.1 First steps

The first thing you'll want to do is see if you can reproduce the FTBFS yourself. Get the code either by running `bzr branch lp:ubuntu/PACKAGE` and then getting the tarball or running `dget PACKAGE_DSC` on the .dsc file from the launchpad page. Once you have that, build it in a schroot.

You should be able to reproduce the FTBFS. If not, check if the build is downloading a missing dependency, which means you just need to make that a build-dependency in debian/control. Building the package locally can also help find if the issue is caused by a missing, unlisted, dependency (builds locally but fails on a schroot).

1.7.2 Checking Debian

Once you have reproduced the issue, it's time to try and find a solution. If the package is in Debian as well, you can check if the package builds there by going to http://packages.qa.debian.org/PACKAGE. If Debian has a newer version, you should merge it. If not, check the buildlogs and bugs linked on that page for any extra information on the ftbfs or patches. Debian also maintains a list of command FTBFSs and how to fix them which can be found at https://wiki.debian.org/qa.debian.org/FTBFS, you will want to check it for solutions too.

1.7.3 Other causes of a package to FTBFS

If a package is in main and missing a dependency that is not in main, you will have to file a MIR bug. https://wiki.ubuntu.com/MainInclusionProcess explains the procedure.

1.7.4 Fixing the issue

Once you have found a fix to the problem, follow the same process as any other bug. Make a patch, add it to a bzr branch or bug, subscribe ubuntu-sponsors, then try to get it included upstream and/or in Debian.

1.8 Shared Libraries

Shared libraries are compiled code which is intended to be shared among several different programs. They are distributed as `.so` files in `/usr/lib/`.

A library exports symbols which are the compiled versions of functions, classes and variables. A library has a name called an SONAME which includes a version number. This SONAME version does not necessarily match the public release version number. A program gets compiled against a given SONAME version of the library. If any of the symbols is removed or changes then the version number needs to be changed which forces any packages using that library to be recompiled against the new version. Version numbers are usually set by upstream and we follow them in our binary package names called an ABI number, but sometimes upstreams do not use sensible version numbers and packagers have to keep separate version numbers.

Libraries are usually distributed by upstream as standalone releases. Sometimes they are distributed as part of a program. In this case they can be included in the binary package along with the program (this is called bundling) if you do not expect any other programs to use the library, more often they should be split out into separate binary packages.

The libraries themselves are put into a binary package named `libfoo1` where `foo` is the name of the library and `1` is the version from the SONAME. Development files from the package, such as header files, needed to compile programs against the library are put into a package called `libfoo-dev`.

1.8.1 An Example

We will use libnova as an example:

```
$ bzr branch ubuntu:trusty/libnova
$ sudo apt-get install libnova-dev
```

To find the SONAME of the library run:

```
$ readelf -a /usr/lib/libnova-0.12.so.2 | grep SONAME
```

The SONAME is `libnova-0.12.so.2`, which matches the file name (usually the case but not always). Here upstream has put the upstream version number as part of the SONAME and given it an ABI version of `2`. Library package names should follow the SONAME of the library they contain. The library binary package is called `libnova-0.12-2` where `libnova-0.12` is the name of the library and `2` is our ABI number.

If upstream makes incompatible changes to their library they will have to reversion their SONAME and we will have to rename our library. Any other packages using our library package will need to recompiled against the new version, this is called a transition and can take some effort. Hopefully our ABI number will continue to match upstream's SONAME but sometimes they introduce incompatibilities without changing their version number and we will need to change ours.

Looking in debian/libnova-0.12-2.install we see it includes two files:

```
usr/lib/libnova-0.12.so.2
usr/lib/libnova-0.12.so.2.0.0
```

The last one is the actual library, complete with minor and point version number. The first one is a symlink which points to the actual library. The symlink is what programs using the library will look for, the running programs do not care about the minor version number.

`libnova-dev.install` includes all the files needed to compile a program with this library. Header files, a config binary, the `.la` libtool file and `libnova.so` which is another symlink pointing at the library, programs compiling against the library do not care about the major version number (although the binary they compile into will).

`.la` libtool files are needed on some non-Linux systems with poor library support but usually cause more problems than they solve on Debian systems. It is a current Debian goal to remove .la files and we should help with this.

1.8.2 Static Libraries

The -dev package also ships `usr/lib/libnova.a`. This is a static library, an alternative to the shared library. Any program compiled against the static library will include the code directory into itself. This gets round worrying about binary compatibility of the library. However it also means that any bugs, including security issues, will not be updated along with the library until the program is recompiled. For this reason programs using static libraries are discouraged.

1.8.3 Symbol Files

When a package builds against a library the `shlibs` mechanism will add a package dependency on that library. This is why most programs will have `Depends: ${shlibs:Depends}` in `debian/control`. That gets replaced with the library dependencies at build time. However shlibs can only make it depend on the major ABI version number, `2` in our libnova example, so if new symbols get added in libnova 2.1 a program using these symbols could still be installed against libnova ABI 2.0 which would then crash.

To make the library dependencies more precise we keep `.symbols` files that list all the symbols in a library and the version they appeared in.

libnova has no symbols file so we can create one. Start by compiling the package:

```
$ bzr builddeb -- -nc
```

The `-nc` will cause it to finish at the end of compilation without removing the built files. Change to the build and run `dpkg-gensymbols` for the library package:

```
$ cd ../build-area/libnova-0.12.2/
$ dpkg-gensymbols -plibnova-0.12-2 > symbols.diff
```

This makes a diff file which you can self apply:

```
$ patch -p0 < symbols.diff
```

Which will create a file named similar to `dpkg-gensymbolsnY_WWI` that lists all the symbols. It also lists the current package version. We can remove the packaging version from that listed in the symbols file because new symbols are not generally added by new packaging versions, but by the upstream developers:

```
$ sed -i s,-0ubuntu2,, dpkg-gensymbolsnY_WWI
```

Now move the file into its location, commit and do a test build:


```
$ mv dpkg-gensymbolsnY_WWI ../../libnova/debian/libnova-0.12-2.symbols
$ cd ../../libnova
$ bzr add debian/libnova-0.12-2.symbols
$ bzr commit -m "add symbols file"
$ bzr builddeb
```

If it successfully compiles the symbols file is correct. With the next upstream version of libnova you would run dpkg-gensymbols again and it will give a diff to update the symbols file.

1.8.4 C++ Library Symbols Files

C++ has even more exacting standards of binary compatibility than C. The Debian Qt/KDE Team maintain some scripts to handle this, see their Working with symbols files page for how to use them.

1.8.5 Further Reading

Junichi Uekawa's Debian Library Packaging Guide goes into this topic in more detail.

1.9 Backporting software updates

Sometimes you might want to make new functionality available in a stable release which is not connected to a critical bug fix. For these scenarios you have two options: either you upload to a PPA or prepare a backport.

1.9.1 Personal Package Archive (PPA)

Using a PPA has a number of benefits. It is fairly straight-forward, you don't need approval of anyone, but the downside of it is that your users will have to manually enable it. It is a non-standard software source.

The PPA documentation on Launchpad is fairly comprehensive and should get you up and running in no time.

1.9.2 Official Ubuntu Backports

The Backports Project is a means to provide new features to users. Because of the inherent stability risks in backporting packages, users do not get backported packages without some explicit action on their part. This generally makes backports an inappropriate avenue for fixing bugs. If a package in an Ubuntu release has a bug, it should be fixed either through the *Security Update or the Stable Release Update process*, as appropriate.

Once you determined you want a package to be backported to a stable release, you will need to test-build and test it on the given stable release. `pbuilder-dist` (in the `ubuntu-dev-tools` package) is a very handy tool to do this easily.

To report the backport request and get it processed by the Backporters team, you can use the `requestbackport` tool (also in the `ubuntu-dev-tools` package). It will determine the intermediate releases that package needs to be backported to, list all reverse-dependencies, and file the backporting request. Also will it include a testing checklist in the bug.

CHAPTER

TWO

KNOWLEDGE BASE

2.1 Communication in Ubuntu Development

In a project where thousands of lines of code are changed, lots of decisions are made and hundreds of people interact every day, it is important to communicate effectively.

2.1.1 Mailing lists

Mailing lists are a very important tool if you want to communicate ideas to a broader team and make sure that you reach everybody, even across timezones.

In terms of development, these are the most important ones:

- https://lists.ubuntu.com/mailman/listinfo/ubuntu-devel-announce (announce-only, the most important development announcements go here)
- https://lists.ubuntu.com/mailman/listinfo/ubuntu-devel (general Ubuntu development discussion)
- https://lists.ubuntu.com/mailman/listinfo/ubuntu-motu (MOTU Team discussion, get help with packaging)

2.1.2 IRC Channels

For real-time discussions, please connect to irc.freenode.net and join one or any of these channels:

- #ubuntu-devel (for general development discussion)
- #ubuntu-motu (for MOTU team discussion and generally getting help)

2.2 Basic Overview of the `debian/` Directory

This article will briefly explain the different files important to the packaging of Ubuntu packages which are contained in the `debian/` directory. The most important of them are `changelog`, `control`, `copyright`, and `rules`. These are required for all packages. A number of additional files in the `debian/` may be used in order to customize and configure the behavior of the package. Some of these files are discussed in this article, but this is not meant to be a complete list.

2.2.1 The changelog

This file is, as its name implies, a listing of the changes made in each version. It has a specific format that gives the package name, version, distribution, changes, and who made the changes at a given time. If you have a GPG key (see: *Getting set up*), make sure to use the same name and email address in `changelog` as you have in your key. The following is a template `changelog`:

```
package (version) distribution; urgency=urgency

 * change details
  - more change details
 * even more change details

-- maintainer name <email address>[two spaces]  date
```

The format (especially of the date) is important. The date should be in **RFC 5322** format, which can be obtained by using the command `date -R`. For convenience, the command `dch` may be used to edit changelog. It will update the date automatically.

Minor bullet points are indicated by a dash "-", while major points use an asterisk "*".

If you are packaging from scratch, `dch --create` (`dch` is in the `devscripts` package) will create a standard `debian/changelog` for you.

Here is a sample `changelog` file for hello:

```
hello (2.8-0ubuntu1) trusty; urgency=low

  * New upstream release with lots of bug fixes and feature improvements.

-- Jane Doe <packager@example.com>  Thu, 21 Oct 2013 11:12:00 -0400
```

Notice that the version has a `-0ubuntu1` appended to it, this is the distro revision, used so that the packaging can be updated (to fix bugs for example) with new uploads within the same source release version.

Ubuntu and Debian have slightly different package versioning schemes to avoid conflicting packages with the same source version. If a Debian package has been changed in Ubuntu, it has `ubuntuX` (where `X` is the Ubuntu revision number) appended to the end of the Debian version. So if the Debian hello `2.6-1` package was changed by Ubuntu, the version string would be `2.6-1ubuntu1`. If a package for the application does not exist in Debian, then the Debian revision is `0` (e.g. `2.6-0ubuntu1`).

For further information, see the changelog section (Section 4.4) of the Debian Policy Manual.

2.2.2 The control file

The `control` file contains the information that the package manager (such as `apt-get`, `synaptic`, and `adept`) uses, build-time dependencies, maintainer information, and much more.

For the Ubuntu `hello` package, the `control` file looks something like this:

```
Source: hello
Section: devel
Priority: optional
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
XSBC-Original-Maintainer: Jane Doe <packager@example.com>
Standards-Version: 3.9.5
Build-Depends: debhelper (>= 7)
Vcs-Bzr: lp:ubuntu/hello
Homepage: http://www.gnu.org/software/hello/
```



```
Package: hello
Architecture: any
Depends: ${shlibs:Depends}
Description: The classic greeting, and a good example
 The GNU hello program produces a familiar, friendly greeting. It
 allows non-programmers to use a classic computer science tool which
 would otherwise be unavailable to them. Seriously, though: this is
 an example of how to do a Debian package. It is the Debian version of
 the GNU Project's 'hello world' program (which is itself an example
 for the GNU Project).
```

The first paragraph describes the source package including the list of packages required to build the package from source in the `Build-Depends` field. It also contains some meta-information such as the maintainer's name, the version of Debian Policy that the package complies with, the location of the packaging version control repository, and the upstream home page.

Note that in Ubuntu, we set the `Maintainer` field to a general address because anyone can change any package (this differs from Debian where changing packages is usually restricted to an individual or a team). Packages in Ubuntu should generally have the `Maintainer` field set to `Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>`. If the Maintainer field is modified, the old value should be saved in the `XSBC-Original-Maintainer` field. This can be done automatically with the `update-maintainer` script available in the `ubuntu-dev-tools` package. For further information, see the Debian Maintainer Field spec on the Ubuntu wiki.

Each additional paragraph describes a binary package to be built.

For further information, see the control file section (Chapter 5) of the Debian Policy Manual.

2.2.3 The copyright file

This file gives the copyright information for both the upstream source and the packaging. Ubuntu and Debian Policy (Section 12.5) require that each package installs a verbatim copy of its copyright and license information to `/usr/share/doc/$(package_name)/copyright`.

Generally, copyright information is found in the `COPYING` file in the program's source directory. This file should include such information as the names of the author and the packager, the URL from which the source came, a Copyright line with the year and copyright holder, and the text of the copyright itself. An example template would be:

```
Format: http://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: Hello
Source: ftp://ftp.example.com/pub/games

Files: *
Copyright: Copyright 1998 John Doe <jdoe@example.com>
License: GPL-2+

Files: debian/*
Copyright: Copyright 1998 Jane Doe <packager@example.com>
License: GPL-2+

License: GPL-2+
 This program is free software; you can redistribute it
 and/or modify it under the terms of the GNU General Public
 License as published by the Free Software Foundation; either
 version 2 of the License, or (at your option) any later
 version.
 .
```

```
 This program is distributed in the hope that it will be
 useful, but WITHOUT ANY WARRANTY; without even the implied
 warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
 PURPOSE.  See the GNU General Public License for more
 details.
 .
 You should have received a copy of the GNU General Public
 License along with this package; if not, write to the Free
 Software Foundation, Inc., 51 Franklin St, Fifth Floor,
 Boston, MA  02110-1301 USA
 .
 On Debian systems, the full text of the GNU General Public
 License version 2 can be found in the file
 `/usr/share/common-licenses/GPL-2'.
```

This example follows the Machine-readable debian/copyright format. You are encouraged to use this format as well.

2.2.4 The rules file

The last file we need to look at is `rules`. This does all the work for creating our package. It is a Makefile with targets to compile and install the application, then create the `.deb` file from the installed files. It also has a target to clean up all the build files so you end up with just a source package again.

Here is a simplified version of the rules file created by `dh_make` (which can be found in the `dh-make` package):

```
#!/usr/bin/make -f
# -*- makefile -*-

# Uncomment this to turn on verbose mode.
#export DH_VERBOSE=1

%:
	dh  $@
```

Let us go through this file in some detail. What this does is pass every build target that `debian/rules` is called with as an argument to `/usr/bin/dh`, which itself will call all the necessary `dh_*` commands.

`dh` runs a sequence of debhelper commands. The supported sequences correspond to the targets of a `debian/rules` file: "build", "clean", "install", "binary-arch", "binary-indep", and "binary". In order to see what commands are run in each target, run:

```
$ dh binary-arch --no-act
```

Commands in the binary-indep sequence are passed the "-i" option to ensure they only work on binary independent packages, and commands in the binary-arch sequences are passed the "-a" option to ensure they only work on architecture dependent packages.

Each debhelper command will record when it's successfully run in `debian/package.debhelper.log`. (Which dh_clean deletes.) So dh can tell which commands have already been run, for which packages, and skip running those commands again.

Each time `dh` is run, it examines the log, and finds the last logged command that is in the specified sequence. It then continues with the next command in the sequence. The `--until`, `--before`, `--after`, and `--remaining` options can override this behavior.

If `debian/rules` contains a target with a name like `override_dh_command`, then when it gets to that command in the sequence, `dh` will run that target from the rules file, rather than running the actual command. The override target

can then run the command with additional options, or run entirely different commands instead. (Note that to use this feature, you should Build-Depend on debhelper 7.0.50 or above.)

Have a look at /usr/share/doc/debhelper/examples/ and man dh for more examples. Also see the rules section (Section 4.9) of the Debian Policy Manual.

2.2.5 Additional Files

The install file

The install file is used by dh_install to install files into the binary package. It has two standard use cases:

- To install files into your package that are not handled by the upstream build system.
- Splitting a single large source package into multiple binary packages.

In the first case, the install file should have one line per file installed, specifying both the file and the installation directory. For example, the following install file would install the script foo in the source package's root directory to usr/bin and a desktop file in the debian directory to usr/share/applications:

```
foo usr/bin
debian/bar.desktop usr/share/applications
```

When a source package is producing multiple binary packages dh will install the files into debian/tmp rather than directly into debian/<package>. Files installed into debian/tmp can then be moved into separate binary packages using multiple $package_name.install files. This is often done to split large amounts of architecture independent data out of architecture dependent packages and into Architecture: all packages. In this case, only the name of the files (or directories) to be installed are needed without the installation directory. For example, foo.install containing only the architecture dependent files might look like:

```
usr/bin/
usr/lib/foo/*.so
```

While foo-common.install containing only the architecture independent file might look like:

```
/usr/share/doc/
/usr/share/icons/
/usr/share/foo/
/usr/share/locale/
```

This would create two binary packages, foo and foo-common. Both would require their own paragraph in debian/control.

See man dh_install and the install file section (Section 5.11) of the Debian New Maintainers' Guide for additional details.

The watch file

The debian/watch file allows us to check automatically for new upstream versions using the tool uscan found in the devscripts package. The first line of the watch file must be the format version (3, at the time of this writing), while the following lines contain any URLs to parse. For example:

```
version=3

http://ftp.gnu.org/gnu/hello/hello-(.*).tar.gz
```


Running uscan in the root source directory will now compare the upstream version number in debian/changelog with the latest available upstream version. If a new upstream version is found, it will be automatically downloaded. For example:

```
$ uscan
hello: Newer version (2.7) available on remote site:
  http://ftp.gnu.org/gnu/hello/hello-2.7.tar.gz
  (local version is 2.6)
hello: Successfully downloaded updated package hello-2.7.tar.gz
    and symlinked hello_2.7.orig.tar.gz to it
```

If your tarballs live on Launchpad, the debian/watch file is a little more complicated (see Question 21146 and Bug 231797 for why this is). In that case, use something like:

```
version=3
https://launchpad.net/flufl.enum/+download http://launchpad.net/flufl.enum/.*/flufl.enum-(.+).tar.gz
```

For further information, see man uscan and the watch file section (Section 4.11) of the Debian Policy Manual.

For a list of packages where the watch file reports they are not in sync with upstream see Ubuntu External Health Status.

The source/format file

This file indicates the format of the source package. It should contain a single line indicating the desired format:

- 3.0 (native) for Debian native packages (no upstream version)
- 3.0 (quilt) for packages with a separate upstream tarball
- 1.0 for packages wishing to explicitly declare the default format

Currently, the package source format will default to 1.0 if this file does not exist. You can make this explicit in the source/format file. If you choose not to use this file to define the source format, Lintian will warn about the missing file. This warning is informational only and may be safely ignored.

You are encouraged to use the newer 3.0 source format. It provides a number of new features:

- Support for additional compression formats: bzip2, lzma and xz
- Support for multiple upstream tarballs
- Not necessary to repack the upstream tarball to strip the debian directory
- Debian-specific changes are no longer stored in a single .diff.gz but in multiple patches compatible with quilt under debian/patches/

https://wiki.debian.org/Projects/DebSrc3.0 summarizes additional information concerning the switch to the 3.0 source package formats.

See man dpkg-source and the source/format section (Section 5.21) of the Debian New Maintainers' Guide for additional details.

2.2.6 Additional Resources

In addition to the links to the Debian Policy Manual in each section above, the Debian New Maintainers' Guide has more detailed descriptions of each file. Chapter 4, "Required files under the debian directory" further discusses the control, changelog, copyright and rules files. Chapter 5, "Other files under the debian directory" discusses additional files that may be used.

2.3 ubuntu-dev-tools: Tools for Ubuntu developers

`ubuntu-dev-tools` package is a collection of 30 tools created for making packaging work much easier for Ubuntu developers. It's similar in scope to Debian `devscripts` package.

2.3.1 Setting up packaging environment

`setup-packaging-environment` command allows to interactively set up packaging environment, including setting environment variables, installing required packages and ensuring that required repositories are enabled.

2.3.2 Environment variables

Introducing yourself

`ubuntu-dev-tools` configurations can be set using environment variables. It's used for example in changelogs. For example, to set e-mail address (and full name), use `UBUMAIL` variable. It overrides the `DEBEMAIL` and `DEBFULLNAME` variables used by `devscripts`. To learn `ubuntu-dev-tools` about you, open *~/.bashrc* in text editor and add something like this:

```
export UBUMAIL="Marcin Mikołajczak <marcin@example.org>"
```

Now, save this file and restart your terminal or use `source ~/.bashrc`.

Changing preferred builder

Default builder is specified by `UBUNTUTOOLS_BUILDER` variable. To set between *pbuilder* (default), *pbuilder-dist*, and *sbuild*, change this variable. Example:

```
export UBUNTUTOOLS_BUILDER=sbuild
```

Save file and restart terminal.

You can also check whether to update the builder every time before building, by changing `UBUNTUTOOLS_UPDATE_BUILDER` from `no` (default) to `yes`.

2.3.3 Downloading source packages

`ubuntu-dev-tools` comes with `pull-lp-source` command, allowing to download source packages from Launchpad. Its usage is simple. To download latest source package for ubuntu-settings, use:

```
$ pull-lp-source ubuntu-settings
```

You can also specify release from which you want to download source or specify version of source package. `-d` option allows to download source package without extracting. A slightly more complex example would look like this:

```
$ pull-lp-source brisk-menu 0.5.0-1 -d
```

`pull-debian-source` package allows to do the same for Debian source packages. It has similar syntax.

2.3.4 Backporting packages

`ubuntu-dev-tools` provides `backportpackage` allowing us to backport a package from specified release of Ubuntu or Debian. For example, to backport `bzr` package from latest development release for your installed Ubuntu version, simply:

```
$ backportpackage -w . bzr
```

This command allows to use more options. To specify Ubuntu release for which you are going to backport a package, use `-d dest` or `--destination=DEST` parameter, where `DEST` is Ubuntu release, for example `xenial`. You can specify more than one destination. In turn, `-s SOURCE` and `--source=SOURCE` specifies the Ubuntu or Debian release from which you are going to backport a package. `-w DIR` and `--workdir=DIR` specifies directory, where package files will be downloaded, unpacked and built. By default, it will create temporary directory that will be automatically deleted. `-U` or `--update` allows to update build environment before building package. `-u` or `--upload` allows to upload package after building (for example to PPAs) using `dput`.

2.3.5 Requesting backports

`requestbackport` command makes creating backports through Launchpad bugs much easier. It creates testing checklist that will be included in the bug. For example, to request backporting libqt5webkit5 from latest development branch to current stable release (without optional parameters):

```
$ requestbackport libqt5webkit5
```

You should fulfill the checklist if you have already tested the backport.

Additional options allows to specify destination of backport and its source, by using `-d DEST` or `--destination=DEST` and `s SRC` or `--source=SRC`.

2.3.6 Other simple commands

`ubuntu-dev-tools` also includes small utilities allowing to do simple tasks like checking whether .iso file is an Ubuntu installation media.

ubuntu-iso

To do this, use `ubuntu-iso <pathtoiso>`, for example:

```
$ ubuntu-iso ~/Downloads/ubuntu.iso
```

bitesize

“Bitesize” tag is used on Launchpad to describe tasks that are suitable for begineers who want to contribute to one of the projects. `bitesize` command allows to add “bitesize” tag to Launchpad bug with just simple command, by providing its number, like:

```
$ bitesize 1735410
```

404main

`404main` allows to check whether all of package build dependencies are included in main repository of specified Ubuntu distribution. Example:


```
$ 404main libqt5webkit5 xenial
```

If any of the required packages isn't part of Ubuntu main repository, you can check whether the package fulfill Ubuntu main inclusion requirements and request it.

Further reading

`ubuntu-dev-tools` manpages are covering more about usage of this package.

2.4 autopkgtest: Automatic testing for packages

The DEP 8 specification defines how automatic testing can very easily be integrated into packages. To integrate a test into a package, all you need to do is:

- add a file called `debian/tests/control` which specifies the requirements for the testbed,
- add the tests in `debian/tests/`.

2.4.1 Testbed requirements

In `debian/tests/control` you specify what to expect from the testbed. So for example you list all the required packages for the tests, if the testbed gets broken during the build or if `root` permissions are required. The DEP 8 specification lists all available options.

Below we are having a look at the `glib2.0` source package. In a very simple case the file would look like this:

```
Tests: build
Depends: libglib2.0-dev, build-essential
```

For the test in `debian/tests/build` this would ensure that the packages `libglib2.0-dev` and `build-essential` are installed.

Note: You can use `@` in the `Depends` line to indicate that you want all the packages installed which are built by the source package in question.

2.4.2 The actual tests

The accompanying test for the example above might be:

```
#!/bin/sh
# autopkgtest check: Build and run a program against glib, to verify that the
# headers and pkg-config file are installed correctly
# (C) 2012 Canonical Ltd.
# Author: Martin Pitt <martin.pitt@ubuntu.com>

set -e

WORKDIR=$(mktemp -d)
trap "rm -rf $WORKDIR" 0 INT QUIT ABRT PIPE TERM
cd $WORKDIR
cat <<EOF > glibtest.c
#include <glib.h>
```



```
int main()
{
    g_assert_cmpint (g_strcmp0 (NULL, "hello"), ==, -1);
    g_assert_cmpstr (g_find_program_in_path ("bash"), ==, "/bin/bash");
    return 0;
}
EOF

gcc -o glibtest glibtest.c `pkg-config --cflags --libs glib-2.0`
echo "build: OK"
[ -x glibtest ]
./glibtest
echo "run: OK"
```

Here a very simple piece of C code is written to a temporary directory. Then this is compiled with system libraries (using flags and library paths as provided by *pkg-config*). Then the compiled binary, which just exercises some parts of core glib functionality, is run.

While this test is very small and simple, it covers quite a lot: that your -dev package has all necessary dependencies, that your package installs working pkg-config files, headers and libraries are put into the right place, or that the compiler and linker work. This helps to uncover critical issues early on.

2.4.3 Executing the test

While the test script can be easily executed on its own, it is strongly recommended to actually use `autopkgtest` from the `autopkgtest` package for verifying that your test works; otherwise, if it fails in the Ubuntu Continuous Integration (CI) system, it will not land in Ubuntu. This also avoids cluttering your workstation with test packages or test configuration if the test does something more intrusive than the simple example above.

The README.running-tests (online version) documentation explains all available testbeds (schroot, LXD, QEMU, etc.) and the most common scenarios how to run your tests with `autopkgtest`, e. g. with locally built binaries, locally modified tests, etc.

The Ubuntu CI system uses the QEMU runner and runs the tests from the packages in the archive, with `-proposed` enabled. To reproduce the exact same environment, first install the necessary packages:

```
sudo apt-get install autopkgtest qemu-system qemu-utils
```

Now build a testbed with:

```
autopkgtest-buildvm-ubuntu-cloud -v
```

(Please see its manpage and `--help` output for selecting different releases, architectures, output directory, or using proxies). This will build e. g. `adt-trusty-amd64-cloud.img`.

Then run the tests of a source package like `libpng` in that QEMU image:

```
autopkgtest libpng --- qemu adt-trusty-amd64-cloud.img
```

The Ubuntu CI system runs packages with only selected packages from `-proposed` available (the package which caused the test to be run); to enable that, run:

```
autopkgtest libpng -U --apt-pocket=proposed=src:foo --- qemu adt-release-amd64-cloud.img
```

or to run with all packages from `-proposed`:

```
autopkgtest libpng -U --apt-pocket=proposed --- qemu adt-release-amd64-cloud.img
```

The `autopkgtest` manpage has a lot more valuable information on other testing options.

2.4.4 Further examples

This list is not comprehensive, but might help you get a better idea of how automated tests are implemented and used in Ubuntu.

- The libxml2 tests are very similar. They also run a test-build of a simple piece of C code and execute it.
- The gtk+3.0 tests also do a compile/link/run check in the "build" test. There is an additional "python3-gi" test which verifies that the GTK library can also be used through introspection.
- In the ubiquity tests the upstream test-suite is executed.
- The gvfs tests have comprehensive testing of their functionality and are very interesting because they emulate usage of CDs, Samba, DAV and other bits.

2.4.5 Ubuntu infrastructure

Packages which have `autopkgtest` enabled will have their tests run whenever they get uploaded or any of their dependencies change. The output of automatically run autopkgtest tests can be viewed on the web and is regularly updated.

Debian also uses `autopkgtest` to run package tests, although currently only in schroots, so results may vary a bit. Results and logs can be seen on http://ci.debian.net. So please submit any test fixes or new tests to Debian as well.

2.4.6 Getting the test into Ubuntu

The process of submitting an autopkgtest for a package is largely similar to *fixing a bug in Ubuntu*. Essentially you simply:

- run `bzr branch ubuntu:<packagename>`,
- edit `debian/control` to enable the tests,
- add the `debian/tests` directory,
- write the `debian/tests/control` based on the DEP 8 Specification,
- add your test case(s) to `debian/tests`,
- commit your changes, push them to Launchpad, propose a merge and get it reviewed just like any other improvement in a source package.

2.4.7 What you can do

The Ubuntu Engineering team put together a list of required test-cases, where packages which need tests are put into different categories. Here you can find examples of these tests and easily assign them to yourself.

If you should run into any problems, you can join the #ubuntu-quality IRC channel to get in touch with developers who can help you.

2.5 Using Chroots

If you are running one version of Ubuntu but working on packages for another versions you can create the environment of the other version with a `chroot`.

A `chroot` allows you to have a full filesystem from another distribution which you can work in quite normally. It avoids the overhead of running a full virtual machine.

2.5.1 Creating a Chroot

Use the command `debootstrap` to create a new chroot:

```
$ sudo debootstrap trusty trusty/
```

This will create a directory `trusty` and install a minimal trusty system into it.

If your version of `debootstrap` does not know about Trusty you can try upgrading to the version in `backports`.

You can then work inside the chroot:

```
$ sudo chroot trusty
```

Where you can install or remove any package you wish without affecting your main system.

You might want to copy your GPG/ssh keys and Bazaar configuration into the chroot so you can access and sign packages directly:

```
$ sudo mkdir trusty/home/<username>
$ sudo cp -r ~/.gnupg ~/.ssh ~/.bazaar trusty/home/<username>
```

To stop apt and other programs complaining about missing locales you can install your relevant language pack:

```
$ apt-get install language-pack-en
```

If you want to run X programs you will need to bind the /tmp directory into the chroot, from outside the chroot run:

```
$ sudo mount -t none -o bind /tmp trusty/tmp
$ xhost +
```

Some programs may need you to bind /dev or /proc.

For more information on chroots see our Debootstrap Chroot wiki page.

2.5.2 Alternatives

SBuild is a system similar to PBuilder for creating an environment to run test package builds in. It closer matches that used by Launchpad for building packages but takes some more setup compared to PBuilder. See the Security Team Build Environment wiki page for a full explanation.

Full virtual machines can be useful for packaging and testing programs. TestDrive is a program to automate syncing and running daily ISO images, see the TestDrive wiki page for more information.

You can also set up pbuilder to pause when it comes across a build failure. Copy C10shell from /usr/share/doc/pbuilder/examples into a directory and use the `--hookdir=` argument to point to it.

Amazon's EC2 cloud computers allow you to hire a computer paying a few US cents per hour, you can set up Ubuntu machines of any supported version and package on those. This is useful when you want to compile many packages at the same time or to overcome bandwidth restraints.

2.6 Setting up sbuild

`sbuild` simplifies building Debian/Ubuntu binary package from source in clean environment. It allows to try debugging packages in environment similar (as opposed to `pbuild`) to builders used by Launchpad.

It works on different architectures and allows to build packages for other releases. It needs kernel supporting overlayfs.

2.6.1 Installing sbuild

To use sbuild, you need to install sbuild and other required packages and add yourself to the `sbuild` group:

```
$ sudo apt install debhelper sbuild schroot ubuntu-dev-tools
$ sudo adduser $USER sbuild
```

Create `.sbuildrc` in your home directory with following content:

```
# Name to use as override in .changes files for the Maintainer: field
# (mandatory, no default!).
$maintainer_name='Your Name <user@example.org>';

# Default distribution to build.
$distribution = "bionic";
# Build arch-all by default.
$build_arch_all = 1;

# When to purge the build directory afterwards; possible values are "never",
# "successful", and "always".  "always" is the default. It can be helpful
# to preserve failing builds for debugging purposes.  Switch these comments
# if you want to preserve even successful builds, and then use
# "schroot -e --all-sessions" to clean them up manually.
$purge_build_directory = 'successful';
$purge_session = 'successful';
$purge_build_deps = 'successful';
# $purge_build_directory = 'never';
# $purge_session = 'never';
# $purge_build_deps = 'never';

# Directory for writing build logs to
$log_dir=$ENV{HOME}."/ubuntu/logs";

# don't remove this, Perl needs it:
1;
```

Replace "Your Name <user@example.org>" with your name and e-mail address. Change default distribution if you want, but remember that you can specify target distribution when executing command.

If you haven't restarted your session after adding yourself to the `sbuild` group, enter:

```
$ sg sbuild
```

Generate GPG keypair for sbuild and create chroot for specified release:

```
$ sbuild-update --keygen
$ mk-sbuild bionic
```

This will create chroot for your current architecture. You might want to specify another architecture. For this, you can use `--arch` option. Example:

```
$ mk-sbuild xenial --arch=i386
```


2.6.2 Using schroot

Entering schroot

You can use `schroot -c <release>-<architecture> [-u <USER>]` to enter newly created chroot, but that's not exactly the reason why you are using sbuild:

```
$ schroot -c bionic-amd64 -u root
```

Using schroot for package building

To build package using sbuild chroot, we use (surprisingly) the `sbuild` command. For example, to build `hello` package from x86_64 chroot, after applying some changes:

```
apt source hello
cd hello-*
sed -i -- 's/Hello/Goodbye/g' src/hello.c   # some
sed -i -- 's/Hello/Goodbye/g' tests/hello-1 #
dpkg-source --commit
dch -i                                      #
update-maintainer                           # changes
sbuild -d bionic-amd64
```

To build package from source package (`.dsc`), use location of the source package as second parameter:

```
sbuild -d bionic-amd64 ~/packages/goodbye_*.dsc
```

To make use of all power of your CPU, you can specify number of threads used for building using standard `-j<threads>`:

```
sbuild -d bionic-amd64 -j8
```

2.6.3 Maintaining schroots

Listing chroots

To get list of all your sbuild chroots, use `schroot -l`. The `source:` chroots are used as base of new schroots. Changes here aren't recommended, but if you have specific reason, you can open it using something like:

```
$ schroot -c source:bionic-amd64
```

Updating schroots

To upgrade the whole schroot:

```
$ sbuild-update -ubc bionic-amd64
```

Expiring active schroots

If because of any reason, you haven't stopped your schroot, you can expire all active schroots using:

```
$ schroot -e --all-sessions
```


2.6.4 Further reading

There is Debian wiki page covering sbuild usage.

Ubuntu Wiki also has article about basics of sbuild.

`sbuild` manpages are covering details about sbuild usage and available features.

2.7 KDE Packaging

Packaging of KDE programs in Ubuntu is managed by the Kubuntu and MOTU teams. You can contact the Kubuntu team on the Kubuntu mailing list and `#kubuntu-devel` Freenode IRC channel. More information about Kubuntu development is on the Kubuntu wiki page.

Our packaging follows the practices of the Debian Qt/KDE Team and Debian KDE Extras Team. Most of our packages are derived from the packaging of these Debian teams.

2.7.1 Patching Policy

Kubuntu does not add patches to KDE programs unless they come from the upstream authors or submitted upstream with the expectation they will be merged soon or we have consulted the issue with the upstream authors.

Kubuntu does not change the branding of packages except where upstream expects this (such as the top left logo of the Kickoff menu) or to simplify (such as removing splash screens).

2.7.2 debian/rules

Debian packages include some additions to the basic Debhelper usage. These are kept in the `pkg-kde-tools` package.

Packages which use Debhelper 7 should add the `--with=kde` option. This will ensure the correct build flags are used and add options such as handling kdeinit stubs and translations:

```
%:
    dh $@ --with=kde
```

Some newer KDE packages use the `dhmk` system, an alternative to `dh` made by the Debian Qt/KDE team. You can read about it in /usr/share/pkg-kde-tools/qt-kde-team/2/README. Packages using this will `include /usr/share/pkg-kde-tools/qt-kde-team/2/debian-qt-kde.mk` instead of running `dh`.

2.7.3 Translations

Packages in main have their translations imported into Launchpad and exported from Launchpad into Ubuntu's language-packs.

So any KDE package in main must generate translation templates, include or make available upstream translations and handle `.desktop` file translations.

To generate translation templates the package must include a `Messages.sh` file; complain to the upstream if it does not. You can check it works by running `extract-messages.sh` which should produce one or more `.pot` files in `po/`. This will be done automatically during build if you use the `--with=kde` option to `dh`.

Upstream will usually have also put the translation `.po` files into the `po/` directory. If they do not, check if they are in separate upstream language packs such as the KDE SC language packs. If they are in separate language packs Launchpad will need to associate these together manually, contact David Planella to do this.

If a package is moved from universe to main it will need to be re-uploaded before the translations get imported into Launchpad.

`.desktop` files also need translations. We patch KDELibs to read translations out of `.po` files which are pointed to by a line `X-Ubuntu-Gettext-Domain=` added to `.desktop` files at package build time. A .pot file for each package is be generated at build time and .po files need to be downloaded from upstream and included in the package or in our language packs. The list of .po files to be downloaded from KDE's repositories is in `/usr/lib/kubuntu-desktop-i18n/desktop-template-list`.

2.7.4 Library Symbols

Library symbols are tracked in `.symbols` files to ensure none go missing for new releases. KDE uses C++ libraries which act a little differently compared to C libraries. Debian's Qt/KDE Team have scripts to handle this. See Working with symbols files for how to create and keep these files up to date.

CHAPTER

THREE

FURTHER READING

You can read this guide offline in different formats, if you install one of the binary packages.

If you want to learn more about building Debian packages, here are some Debian resources you may find useful:

- How to package for Debian;
- Debian Policy Manual;
- Debian New Maintainers' Guide — available in many languages;
- Packaging tutorial (also available as a package);
- Guide for Packaging Python Modules.

We are always looking to improve this guide. If you find any problems or have some suggestions, please report a bug on Launchpad. If you'd like to help work on the guide, grab the source there as well.