Canonical

Ubuntu Packaging Guide

© 2024 Canonical Ltd.
All rights reserved.

Canonical

Contents

1 Tutorial **3**
- 1.1 Core tutorial . . . 3

2 How-to guides **12**
- 2.1 How do I...? . . . 12

3 Explanation **49**
- 3.1 Upstream and downstream . . . 49
- 3.2 Package model . . . 51
- 3.3 Patches . . . 56
- 3.4 Ubuntu development process . . . 58
- 3.5 Ubuntu releases . . . 64
- 3.6 Ubuntu package archive . . . 68
- 3.7 Launchpad . . . 74
- 3.8 Sponsorship . . . 78
- 3.9 Proposed migrations . . . 79
- 3.10 Importing changes from Debian (merges & syncs) . . . 79
- 3.11 Transitions . . . 81
- 3.12 Backports . . . 81
- 3.13 Main Inclusion Review (MIR) . . . 81

4 Reference **84**
- 4.1 Basic overview of the `debian/` directory . . . 84
- 4.2 Debian policy . . . 92
- 4.3 Supported architectures . . . 95
- 4.4 Filesystem hierarchy standard . . . 96
- 4.5 Package version format . . . 99
- 4.6 DEP 3 – Patch file headers . . . 99
- 4.7 Launchpad text markup . . . 102
- 4.8 Glossary . . . 116

5 Contribute to the Ubuntu Packaging Guide **145**
- 5.1 How to contribute . . . 145
- 5.2 Contribution format for the project . . . 145

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

1. Tutorial

This section contains step-by-step tutorials to help you get started with Ubuntu packaging and development. We hope the tutorials make as few assumptions as possible and are accessible to anyone with an interest in Ubuntu packaging.

This should be a great place to start learning about packaging and development.

1.1. Core tutorial

This tutorial will introduce you to the basics of Ubuntu packaging, while helping to set up your computer so that you can start working with packages.

> **Caution**
>
> The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.
>
> The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.
>
> If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

1.1.1. Getting set up

> **Caution**
>
> The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.
>
> The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.
>
> If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

1.1.2. Making changes to a package

This tutorial goes through the process of adding a patch to a package in Ubuntu. Specifically, we add a command-line option to the `hello` command, which greets a user using their username. This covers topics like `git-ubuntu`, `quilt`, and changelogs.

Getting the tools

We are using a `git-ubuntu` workflow. Install it with `snap`:

```
$ sudo snap install --classic git-ubuntu
```

Once we have `git-ubuntu` installed, use it to fetch the source code for the `hello` package in Ubuntu:

```
$ git-ubuntu clone hello
$ cd hello/
```

We are using some tools from the `ubuntu-dev-tools` package. Install it with:

```
$ sudo apt install -y ubuntu-dev-tools
```

Understanding the package

Initially, we have the `ubuntu/devel` branch checked out. At the time of writing this tutorial, the development series is `plucky`, so the `ubuntu/devel` branch is in line with the `plucky` version.

Let's explore the packaging we are dealing with. In the `debian` directory, there are files like `changelog`, `rules`, `control`, and more. Everything outside of the `debian` directory is from the original upstream source. In `debian/source/format`, we see:

```
3.0 (quilt)
```

This means that, like most packages, this package uses the `quilt` tool to manage patches to the upstream source code. So, even though we are using Git to track our changes to the *packaging*, we need to use a `quilt` patch to maintain the changes required for our new command-line option.

In particular, instead of ending up with a Git commit that modifies the upstream source code directly, our commit adds a new file, `debian/patches/add-username-command-line-option.patch`, which contains the patch to apply to the upstream source code.

Creating a patch with `quilt`

First, create the new patch file using `quilt`:

```
$ QUILT_PATCHES=debian/patches quilt new add-username-command-line-option.patch
Patch add-username-command-line-option.patch is now on top
```

This should create a new, empty file: `debian/patches/add-username-command-line-option.patch`. And it adds a corresponding entry to the `debian/patches/series` file. Once that is done, start writing the patch. For each source file that is modified by our patch, we need to tell `quilt` about it. In this case:

```
$ quilt add src/hello.c
```

After that, edit the source normally using our favorite text editor. To see our progress, use the usual Git tools to see the diff. So, after adding the new command line flag, the diff might look like:

```
$ git diff -- src/hello.c
diff --git a/src/hello.c b/src/hello.c
index 453962f..f1ccf0a 100644
--- a/src/hello.c
+++ b/src/hello.c
@@ -23,6 +23,10 @@
 #include "error.h"
 #include "progname.h"
 #include "xalloc.h"
+#include "unistd.h"
```

(continues on next page)

(continued from previous page)

```
+#include "sys/types.h"
+#include "pwd.h"
+#include "limits.h"

 static const struct option longopts[] = {
   {"greeting", required_argument, NULL, 'g'},
@@ -44,6 +48,8 @@ main (int argc, char *argv[])
   const char *greeting_msg;
   wchar_t *mb_greeting;
   size_t len;
+  struct passwd *pwd = NULL;
+  char user_greeting[sizeof("hello, !") + LOGIN_NAME_MAX] = {};

   set_program_name (argv[0]);

@@ -65,7 +71,7 @@ main (int argc, char *argv[])
      This is implemented in the Gnulib module "closeout".  */
   atexit (close_stdout);

-  while ((optc = getopt_long (argc, argv, "g:htv", longopts, NULL)) != -1)
+  while ((optc = getopt_long (argc, argv, "g:htvu", longopts, NULL)) != -1)
     switch (optc)
       {
       /* --help and --version exit immediately, per GNU coding standards.  */
@@ -83,6 +89,15 @@ main (int argc, char *argv[])
       case 't':
        greeting_msg = _("hello, world");
        break;
+      case 'u':
+        errno = 0;
+        pwd = getpwuid(geteuid());
+        if (!pwd)
+          error (EXIT_FAILURE, errno, _("failed to get user name"));
+
+        snprintf(user_greeting, sizeof(user_greeting), "hello, %s!", pwd->pw_
name);
+        greeting_msg = _(user_greeting);
+        break;
       default:
        lose = 1;
        break;
```

To save these changes in our quilt patch, we need to *refresh* the patch:

```
$ quilt refresh -p ab --no-timestamps --no-index
```

It is good practice to add DEP-3 headers[2] to patches to add additional context, such as the origin, author, and related bugs. The `quilt` tool has a helper for this:

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

[2] https://dep-team.pages.debian.net/deps/dep3/

This opens a text editor with pre-populated text:

```
Description: <short description, required>
 <long description that can span multiple lines, optional>
Author: <name and email of author, optional>
Origin: <upstream|backport|vendor|other>, <URL, required except if Author is
present>
Bug: <URL to the upstream bug report if any, implies patch has been forwarded,
optional>
Bug-<Vendor>: <URL to the vendor bug report if any, optional>
Forwarded: <URL|no|not-needed, useless if you have a Bug field, optional>
Applied-Upstream: <version|URL|commit, identifies patches merged upstream,
optional>
Reviewed-by: <name and email of a reviewer, optional>
Last-Update: 2025-04-23 <YYYY-MM-DD, last update of the meta-information,
optional>
---
This patch header follows DEP-3: http://dep.debian.net/deps/dep3/
```

Not everything here needs to be filled in. In this case, our headers might look like:

```
Description: Add -u command line option to hello
 This command line option adds a username-specific greeting. E.g.,
 $ hello -u
 hello, user123!
Author: Nick Rosbrook <enr0n@ubuntu.com>
Forwarded: no, Ubuntu only
Last-Update: 2025-04-23
---
This patch header follows DEP-3: http://dep.debian.net/deps/dep3/
```

Our final patch should look something like:

```
Description: Add -u command line option to hello
 This command line option adds a username-specific greeting. E.g.,
 $ hello -u
 hello, user123!
Author: Nick Rosbrook <enr0n@ubuntu.com>
Forwarded: no, Ubuntu only
Last-Update: 2025-04-23
---
This patch header follows DEP-3: http://dep.debian.net/deps/dep3/
--- a/src/hello.c
+++ b/src/hello.c
@@ -23,6 +23,10 @@
 #include "error.h"
 #include "progname.h"
 #include "xalloc.h"
+#include "unistd.h"
+#include "sys/types.h"
+#include "pwd.h"
```

(continues on next page)

(continued from previous page)

```
+#include "limits.h"

 static const struct option longopts[] = {
   {"greeting", required_argument, NULL, 'g'},
@@ -44,6 +48,8 @@
   const char *greeting_msg;
   wchar_t *mb_greeting;
   size_t len;
+  struct passwd *pwd = NULL;
+  char user_greeting[sizeof("hello, !") + LOGIN_NAME_MAX] = {};

   set_program_name (argv[0]);

@@ -65,7 +71,7 @@
      This is implemented in the Gnulib module "closeout".  */
   atexit (close_stdout);

-  while ((optc = getopt_long (argc, argv, "g:htv", longopts, NULL)) != -1)
+  while ((optc = getopt_long (argc, argv, "g:htvu", longopts, NULL)) != -1)
     switch (optc)
       {
  /* --help and --version exit immediately, per GNU coding standards.  */
@@ -83,6 +89,15 @@
       case 't':
  greeting_msg = _("hello, world");
  break;
+      case 'u':
+       errno = 0;
+       pwd = getpwuid(geteuid());
+       if (!pwd)
+         error (EXIT_FAILURE, errno, _("failed to get user name"));
+
+       snprintf(user_greeting, sizeof(user_greeting), "hello, %s!", pwd->pw_
name);
+       greeting_msg = _(user_greeting);
+       break;
       default:
  lose = 1;
  break;
```

The patch is currently applied in the working directory.

- To un-apply: `quilt pop -a`
- To apply again: `quilt push -a`

Committing the changes

Now that we have created our patch file, track the changes in Git. Add the new patch file (and in this case, the newly created `debian/patches/series` file) to the Git index and commit the change:

```
$ git add debian/patches/
$ git commit -m "debian/patches: add a new -u command line option to hello"
```

Next, some housekeeping changes:

1. Make sure that the `Maintainer:` field in `debian/control` is set correctly.
2. Add a new entry to `debian/changelog` explaining our changes and incrementing the package version number.

To update the maintainer field, use the `update-maintainer` tool from the `ubuntu-dev-tools` package. In this case, the field is already set correctly, so we should see:

```
$ update-maintainer
The Maintainer email is set to an ubuntu.com address. Doing nothing.
```

If a change was made, commit that change with:

```
$ git commit -m "update maintainer" -- debian/control
```

Once you have either updated the maintainer, or confirmed that it is already correct, update the changelog. The `dch` tool helps with this. If you run `dch -i`, you see something like this in your text editor:

```
hello (2.10-3ubuntu1) UNRELEASED; urgency=medium

  *

 -- Nick Rosbrook <enr0n@ubuntu.com>  Tue, 22 Apr 2025 17:03:03 -0400

hello (2.10-3build2) oracular; urgency=medium

  * No-change rebuild to bump version in oracular.

 -- Marc Deslauriers <marc.deslauriers@ubuntu.com>  Mon, 27 May 2024 07:18:24 -
0400

hello (2.10-3build1) noble; urgency=high

  * No change rebuild for 64-bit time_t and frame pointers.

 -- Julian Andres Klode <juliank@ubuntu.com>  Mon, 08 Apr 2024 17:58:52 +0200

 [...SNIP...]
```

The `dch` tool has done a few things:

1. Created a new empty changelog entry.
2. Set the author line using your name, email, and the current date and time.
3. Updated the package version number to `2.10-3ubuntu1`.

4. Set the release name to `UNRELEASED`.

Our job now is to fill in the entry and explain our changes. In this case, write something like:

```
hello (2.10-3ubuntu1) plucky; urgency=medium

  * debian/patches: add a new -u command line option to hello

 -- Nick Rosbrook <enr0n@ubuntu.com>  Tue, 22 Apr 2025 17:03:03 -0400
```

Once you are happy with the changelog entry, commit the change:

```
$ git commit -m "update changelog" -- debian/changelog
```

At this point, we should have two (or three if `update-maintainer` was needed) commits: one adding our new patch, and another updating the changelog:

```
$ git log
commit a62e1590cc6a12925c8fe9bce49d9b7f5834468e (HEAD -> ubuntu/devel)
Author: Nick Rosbrook <enr0n@ubuntu.com>
Date:   Wed Apr 23 10:04:32 2025 -0400

    update changelog

commit d6ef1607ce6163e6a611c591e94f478c2c06a35a
Author: Nick Rosbrook <enr0n@ubuntu.com>
Date:   Tue Apr 22 16:24:39 2025 -0400

    debian/patches: add a new -u command line option to hello

commit fd73db6d7406ee1fb8512a5b54c259f3b3368eab (tag: pkg/import/2.10-3build2, pkg/ubuntu/plucky
Author: Marc Deslauriers <marc.deslauriers@ubuntu.com>
Date:   Mon May 27 07:18:24 2024 -0400

    2.10-3build2 (patches unapplied)

    Imported using git-ubuntu import.

Notes (changelog):
      * No-change rebuild to bump version in oracular.
```

And that's it! We have successfully:

- Added a new patch to this package.
- Documented our change.
- Prepared the package for its next upload to the Ubuntu archive.

Next steps

From here, there are many options for testing our patch before proposing the change in a merge proposal:

- Build and test the package locally using `sbuild` and `autopkgtest`.
- Upload to a PPA and test from there.

Once you feel confident that the patch is working correctly, open a merge proposal and request *Sponsorship* (page 78) for your change.

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

1.1.3. Create a new package

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

1.1.4. Fix a bug

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

1.1.5. Merge a package from Debian

This article is still work in progress. You can use the Ubuntu Maintainer Handbook[3] in the meantime.

Note

Be aware that the Ubuntu Maintainer Handbook was written for server team and not a general audience.

[3] https://github.com/canonical/ubuntu-maintainers-handbook/blob/main/PackageMerging.md

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

2. How-to guides

If you have a specific goal in mind and are already familiar with the basics of Ubuntu packaging, our how-to guides cover some of the more common operations and tasks that you may need to complete.

They will help you to achieve a particular end result, but may require you to understand and adapt the steps to fit your specific requirements.

2.1. How do I...?

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

2.1.1. Get the source of a package

Before you can work on a *source package* you need to get the *source code* of that package. This article presents four ways to achieve this: `git-ubuntu`, `pull-pkg`, and `apt-get source`, and `dget`.

git-ubuntu

Note

`git-ubuntu` is the modern way of working with *Ubuntu* source packages.

Warning

`git-ubuntu` is still in active development and these instructions will likely change over time. While `git-ubuntu` will become the default packaging method, for now you may encounter rough edges or unsupported edge cases. You can ask for help in the `#ubuntu-devel` channel or open a bug report[4] on *Launchpad*. Bug reports are very welcome!

Install

The following command will install `git-ubuntu`:

[4] https://bugs.launchpad.net/git-ubuntu

```
sudo snap install --classic --edge git-ubuntu
```

Basic usage

To clone a source package git repository to a directory:

```
git-ubuntu clone PACKAGE [DIRECTORY]
```

To generate the *orig tarballs* for a given source package:

```
git-ubuntu export-orig
```

Example

```
git-ubuntu clone hello
cd hello
git-ubuntu export-orig
```

You can find further information in these two blog articles (note that they are from 2017):

- git-ubuntu clone[5]
- Git Ubuntu: More on the imported repositories[6]

pull-pkg

The `pull-pkg` command is part of the `ubuntu-dev-tools` package and downloads a specific version of a source package, or the latest version from a specified release.

Install

The following command will install `ubtuntu-dev-tools`, which includes `pull-pkg`:

```
sudo apt update && sudo apt install ubuntu-dev-tools
```

Basic usage

```
pull-pkg [OPTIONS] PACKAGE-NAME [SERIES|VERSION]
```

You can find further information on the manual page *`pull-pkg(1)`*[7].

Examples

There are convenience scripts that follow a similar syntax and set the `OPTIONS` for pull type and *Distribution* appropriately. Here are three examples (although there are others):

[5] https://ubuntu.com/blog/git-ubuntu-clone
[6] https://ubuntu.com/blog/git-ubuntu-more-on-the-imported-repositories
[7] https://manpages.ubuntu.com/manpages/noble/en/man1/pull-pkg.1.html

pull-lp-source

- To download the latest version of the `hello` source package for the *Current Release in Development* from Launchpad:

```
pull-lp-source hello
```

- To download the latest version of the `hello` source package for the Ubuntu `mantic` release from Launchpad:

```
pull-lp-source hello mantic
```

- To download version `2.10-3` of the `hello` source package from Launchpad:

```
pull-lp-source hello 2.10-3
```

pull-ppa-source

- To download the latest version of the `hello` source package from the Launchpad *Personal Package Archive* (PPA), also called `hello`, of the user `dviererbe`:

```
pull-ppa-source --ppa 'dviererbe/hello' 'hello'
```

- To download the latest version of the `hello` source package for the `mantic` release from the same Launchpad PPA:

```
pull-ppa-source --ppa 'dviererbe/hello' 'hello' 'mantic'
```

- To download version `2.10-3` of the `hello` source package for the `mantic` release from the same Launchpad PPA:

```
pull-ppa-source --ppa 'dviererbe/hello' 'hello' '2.10-3'
```

pull-debian-source

- To download the latest version of the `hello` source package from *Debian*:

```
pull-debian-source 'hello'
```

- To download the latest version of the `hello` source package for the `sid` release from Debian:

```
pull-debian-source 'hello' 'sid'
```

- To download the version `2.10-3` of the `hello` source package from Debian:

```
pull-debian-source 'hello' '2.10-3'
```

apt-get source

The *APT* package manager can also fetch source packages.

Important

Source packages are tracked separately from *binary packages* via `deb-src` lines in the *sources.list(5)*[8] files. This means that you will need to add such a line for each *repository* you want to get source packages from; otherwise you will probably get either the wrong (too old/too new) source package versions – or none at all.

Basic usage

apt

apt-get

```
apt source PACKAGE-NAME
```

You can find further information on the manual page *apt(8)*[9].

```
apt-get source PACKAGE-NAME
```

You can find further information on the manual page *apt-get(8)*[10].

Example

apt

apt-get

```
apt source 'hello'
```

```
apt-get source 'hello'
```

dget

The `dget` command is part of the `devscripts` package. If you call it with the URL of a `.dsc` or `.changes` file it acts as a source package aware *wget(1)*[11] and downloads all associated files that are listed in the `.dsc` or `.changes` file (debian tarball, *orig tarballs*, *upstream signatures*).

Install

```
sudo apt update && sudo apt install devscripts
```

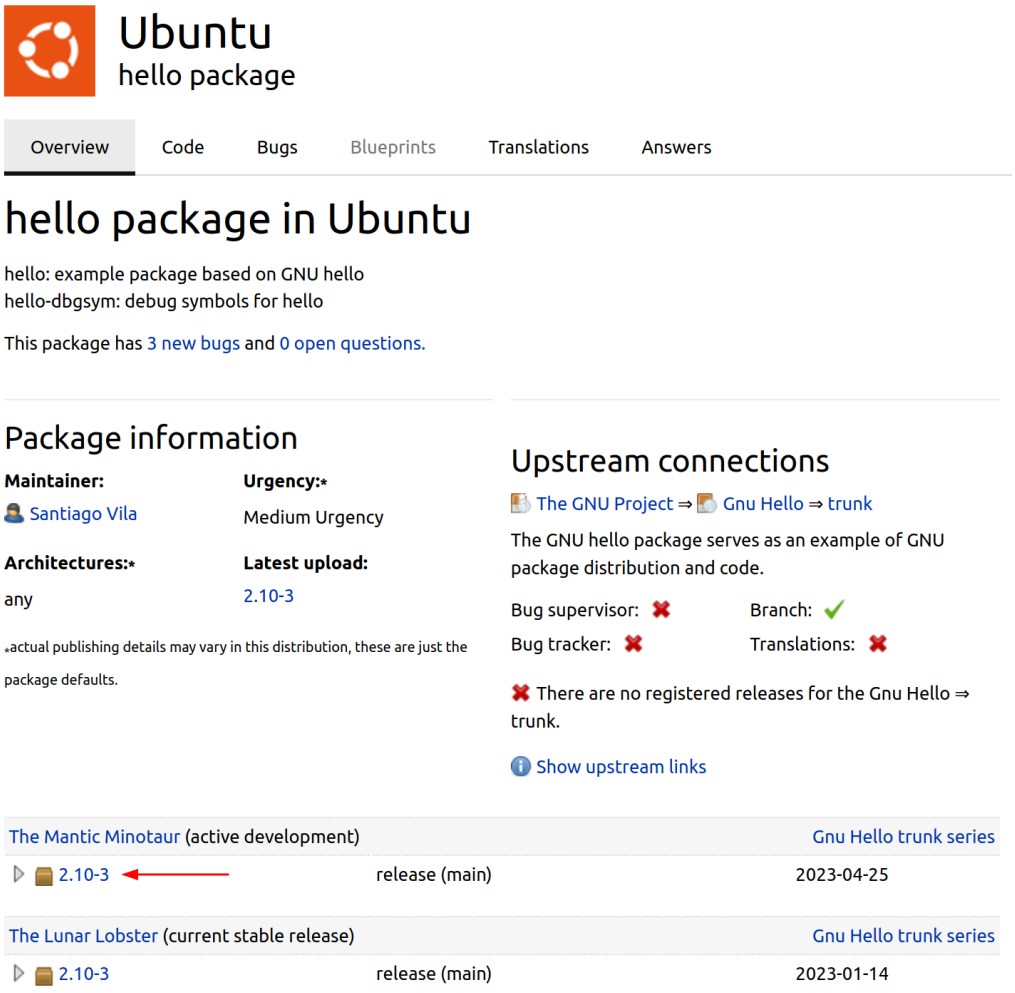
Basic usage

```
dget URL
```

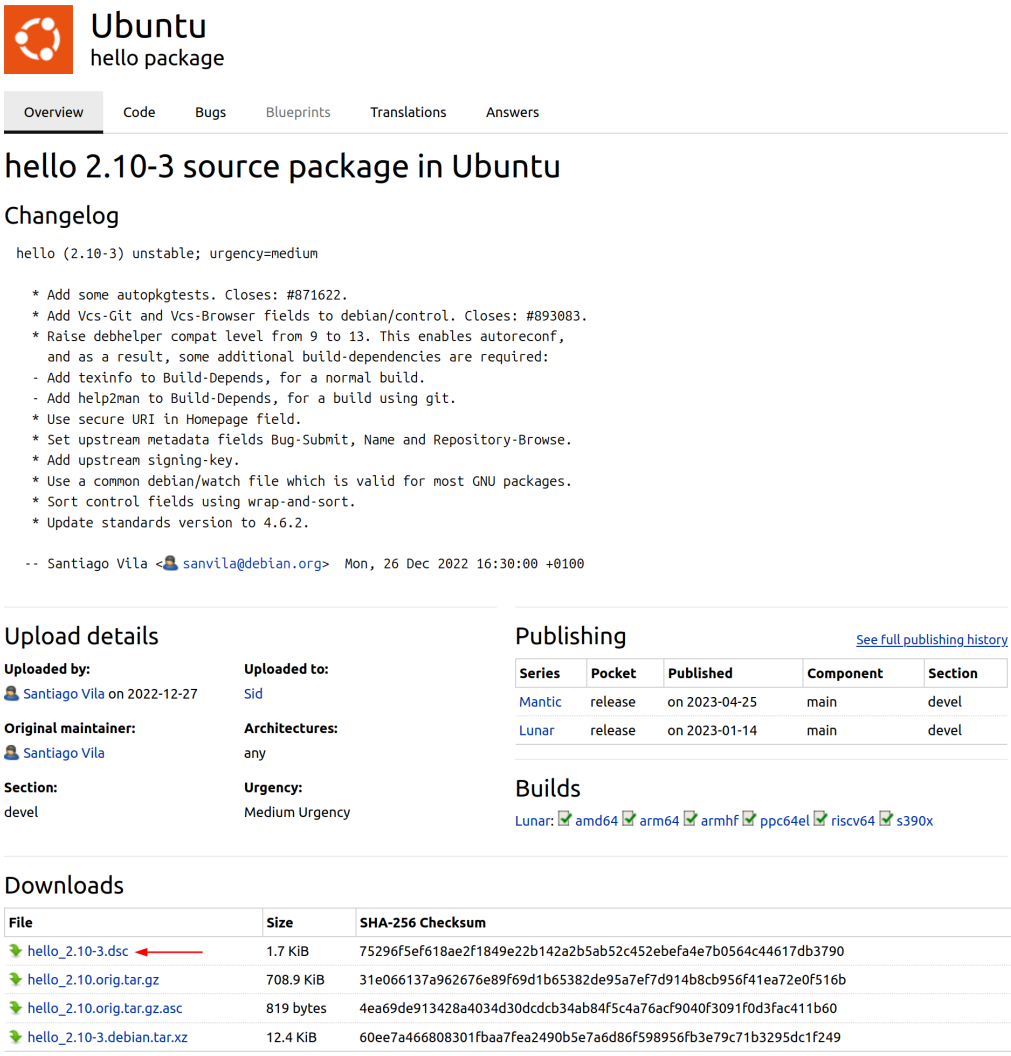
[8] https://manpages.ubuntu.com/manpages/noble/en/man5/sources.list.5.html
[9] https://manpages.ubuntu.com/manpages/noble/en/man8/apt.8.html
[10] https://manpages.ubuntu.com/manpages/noble/en/man8/apt-get.8.html
[11] https://manpages.ubuntu.com/manpages/noble/en/man1/wget.1.html

Example

Go to Launchpad and select the package you want to download (in this example; the latest version of the `hello` source package):



Next, copy the download link of the `.dsc` file:



Ubuntu
hello package

Overview Code Bugs Blueprints Translations Answers

hello 2.10-3 source package in Ubuntu

Changelog

```
hello (2.10-3) unstable; urgency=medium

  * Add some autopkgtests. Closes: #871622.
  * Add Vcs-Git and Vcs-Browser fields to debian/control. Closes: #893083.
  * Raise debhelper compat level from 9 to 13. This enables autoreconf,
    and as a result, some additional build-dependencies are required:
  - Add texinfo to Build-Depends, for a normal build.
  - Add help2man to Build-Depends, for a build using git.
  * Use secure URI in Homepage field.
  * Set upstream metadata fields Bug-Submit, Name and Repository-Browse.
  * Add upstream signing-key.
  * Use a common debian/watch file which is valid for most GNU packages.
  * Sort control fields using wrap-and-sort.
  * Update standards version to 4.6.2.

 -- Santiago Vila <sanvila@debian.org>  Mon, 26 Dec 2022 16:30:00 +0100
```

Upload details

Uploaded by: Santiago Vila on 2022-12-27
Uploaded to: Sid
Original maintainer: Santiago Vila
Architectures: any
Section: devel
Urgency: Medium Urgency

Publishing — See full publishing history

Series	Pocket	Published	Component	Section
Mantic	release	on 2023-04-25	main	devel
Lunar	release	on 2023-01-14	main	devel

Builds

Lunar: amd64 arm64 armhf ppc64el riscv64 s390x

Downloads

File	Size	SHA-256 Checksum
hello_2.10-3.dsc	1.7 KiB	75296f5ef618ae2f1849e22b142a2b5ab52c452ebefa4e7b0564c44617db3790
hello_2.10.orig.tar.gz	708.9 KiB	31e066137a962676e89f69d1b65382de95a7ef7d914b8cb956f41ea72e0f516b
hello_2.10.orig.tar.gz.asc	819 bytes	4ea69de913428a4034d30dcdcb34ab84f5c4a76acf9040f3091f0d3fac411b60
hello_2.10-3.debian.tar.xz	12.4 KiB	60ee7a466808301fbaa7fea2490b5e7a6d86f598956fb3e79c71b3295dc1f249

Finally, call `dget` with the copied URL:

```
dget https://launchpad.net/ubuntu/+archive/primary/+sourcefiles/hello/2.10-3/
hello_2.10-3.dsc
```

Note that this works for links from Debian and Launchpad Personal Package Archives too.

You can find further information on the manual page *dget(1)*[12].

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

[12] https://manpages.ubuntu.com/manpages/noble/en/man1/dget.1.html

2.1.2. Download a new upstream version

Once in a while you may need to download a new *upstream* release or check if a newer upstream release exists; for example:

- When fixing a bug, to rule out that a more recent version may have already fixed the bug.
- As a *source package maintainer*, to check for, download, and package a newer upstream release.

Most of the source packages contain a `watch` file in the `debian` folder. This is a configuration file for the *uscan(1)*[13] utility which allows you to automatically search HTTP or FTP sites or *git(1)*[14] repositories for newly available updates of the upstream project.

> **Note**
>
> If the source package does not contain a `debian/watch` file, there may be an explanation and instructions in the `debain/README.source` or `debian/README.debian` file (if available) that tell you how to proceed.

Best practices

You should download upstream file(s) manually only if there is no automatic download mechanism and you can't find any download instructions.

Remember that a package may get distributed to hundreds of thousands of users. Humans are the weakest link in this distribution chain, because we may accidentally miss or skip a verification step, misspell a *URL*, copy the wrong URL or copy a URL only partially, etc.

If you still have to download upstream file(s) manually make sure to verify *Cryptographic Signatures* (if available). The *Signing Key* of the upstream project should be stored in the source package as `debian/upstream/signing-key.asc` (if the upstream project has a signing key).

uscan(1)[15] verifies downloads against this signing key automatically (if available).

Download new upstream version (if available)

Running *uscan(1)*[16] from the *Root* of the *Source Tree* will check if a newer upstream version exists and downloads it:

```
uscan
```

If *uscan(1)*[17] could not find a newer upstream version it will return with the exit code *1* and print nothing to the *Standard Output*.

uscan(1)[18] reads the first entry in `debian/changelog` to determine the name and version of the source package.

[13] https://manpages.ubuntu.com/manpages/noble/en/man1/uscan.1.html
[14] https://manpages.ubuntu.com/manpages/noble/en/man1/git.1.html
[15] https://manpages.ubuntu.com/manpages/noble/en/man1/uscan.1.html
[16] https://manpages.ubuntu.com/manpages/noble/en/man1/uscan.1.html
[17] https://manpages.ubuntu.com/manpages/noble/en/man1/uscan.1.html
[18] https://manpages.ubuntu.com/manpages/noble/en/man1/uscan.1.html

You can always add the `--verbose` flag to see more information (e.g., which version *uscan(1)*[19] found):

```
uscan --verbose
```

Check for new upstream version (no download)

If you just want to check if a new update is available, but you don't want to download anything, you can run the *uscan(1)*[20] command with the `--safe` flag from the *Root* of the source tree:

```
uscan --safe
```

Force the download

You can use the `--force-download` flag to download an upstream release from the upstream project, even if the upstream Release is up-to-date with the source package:

```
uscan --force-download
```

Download the source of older versions from the upstream project

If you want to download the source of a specific version from the upstream project you can use the `--download-version` flag.

Basic syntax:

```
uscan --download-version VERSION
```

For example:

```
uscan --download-version '1.0'
```

In the special case that you want to download the source for the current version of the source package from the upstream project you can use the `--download-current-version` flag instead, which parses the version to download from the first entry in `debian/changelog` file:

```
uscan --download-current-version
```

Note

The `--download-version` and `--download-current-version` flags are both a *best-effort* features of *uscan(1)*[21].

There are special cases where they do not work for technical reasons.

[19] https://manpages.ubuntu.com/manpages/noble/en/man1/uscan.1.html
[20] https://manpages.ubuntu.com/manpages/noble/en/man1/uscan.1.html
[21] https://manpages.ubuntu.com/manpages/noble/en/man1/uscan.1.html

Note

In most cases you actually want to download the source from the *Ubuntu Archive* and not re-download the source from the upstream project.

How to get the Source from the Archive? (page 12)

Further Information

- Manual page – *uscan(1)*[22]
- Debian wiki – debian/watch[23]
- Debian policy `4.6.2.0` – Upstream source location: debian/watch[24]

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

2.1.3. Build packages

In Ubuntu, packages can be built in several ways, depending on the intended artifacts. We cover the following types of builds:

- Source and binary (using `sbuild` for a clean environment)
- Binary-only (using `sbuild` for a clean environment)
- Source-only (using `debuild`)
- Binary-only (using `debuild` and installed build dependencies)

(Many other backends are available, including an `schroot`-based backend.)

Only source uploads are permitted to PPAs or the archive. That being said, it is best practice to perform a local build and iron out any potential issues prior to uploading it to any archive.

Prerequisites

```
$ sudo apt install sbuild debhelper ubuntu-dev-tools piuparts
```

All of the following sections assume you have already fetched the packaging and are in the same directory as `debian/`.

For *sbuild(1)*[25], follow the instructions on the Debian and Ubuntu sbuild pages as linked in the Resources section.

[22] https://manpages.ubuntu.com/manpages/noble/en/man1/uscan.1.html
[23] https://wiki.debian.org/debian/watch
[24] https://www.debian.org/doc/debian-policy/ch-source.html#upstream-source-location-debian-watch
[25] https://manpages.ubuntu.com/manpages/noble/en/man1/sbuild.1.html

`sbuild`-based builds

This is the standard way of building a package for Ubuntu. All of the Debian and Ubuntu infrastructure use *`sbuild(1)`*[26], so it is beneficial to learn how to use it. For more information on setting up *`sbuild(1)`*[27], refer to the links in the Resources section.

To do a binary-only build of a package using `sbuild`, run:

```
$ sbuild -c <RELEASE>-<ARCH>[-shm]
```

> **Note**
>
> It is possible to use `-d` instead of `-c`, but that causes the produced files to contain the entire chroot name (`<RELEASE>-<ARCH>[-shm]`) instead of just `<RELEASE>`. An example chroot name is `noble-amd64-shm`.

To explicitly run Lintian following the build:

```
$ sbuild -c <RELEASE>-<ARCH>[-shm] --run-lintian [--lintian-opts="-EvIiL +pedantic
"]
```

To build a package without running *`dh_clean(1)`*[28], run:

```
$ sbuild -c <RELEASE>-<ARCH>[-shm] --no-clean-source
```

To build both a binary *and* a source package with one `sbuild` run:

```
$ sbuild -c <RELEASE>-<ARCH>[-shm] -s
```

> **Note**
>
> Launchpad rejects uploads that contains both binaries and sources. However, this is required for uploads to the Debian NEW queue. That being said, uploads to Debian with binaries do not migrate to Testing[29].

Here is a complete, working example of running the `autopkgtest` following the build:

```
$ sbuild -c noble-amd64-shm --run-autopkgtest \
  --autopkgtest-virt-server=qemu \
  --autopkgtest-virt-server-opt="/path/to/autopkgtest-noble-amd64.img" \
  --autopkgtest-opt="--apt-pocket=proposed=src:qt6-base" \
  --autopkgtest-opt="-U" --autopkgtest-opt="--ram-size=12000" \
  --autopkgtest-opt="--setup-commands='apt-get -y install aptitude \
    && aptitude -t noble-proposed -y install qt6-base-dev=6.8.1+dfsg-0ubuntu1'"
```

[26] https://manpages.ubuntu.com/manpages/noble/en/man1/sbuild.1.html
[27] https://manpages.ubuntu.com/manpages/noble/en/man1/sbuild.1.html
[28] https://manpages.ubuntu.com/manpages/noble/en/man1/dh_clean.1.html
[29] https://lists.debian.org/debian-devel-announce/2019/07/msg00002.html

Building with debuild

debuild(1)[30] (short for *dpkg-buildpackage(1)*[31]) is another tool used to build Debian packages. It is part of the *debhelper(7)*[32] package and written in Perl.

Ubuntu maintain its own version the debhelper package. Therefore, packages built on Debian may be slightly different than packages built on Ubuntu.

Source-only builds

To build a source package *without* including the upstream tarball, run:

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

To build a source package *with* the upstream tarball, run:

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

To build a source package without running Lintian, run:

```
$ debuild --no-lintian -S -d
```

> **Note**
>
> The --no-lintian flag only works in this case if it is first.

To build a source package without running *dh_clean(1)*[33], run:

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

> **Note**
>
> This tends to fix failures regarding missing build dependencies.

To build a source package without a cryptographic signature (not recommended), run:

```
$ debuild -S -d -us -uc
```

Local binary-only builds

This is really only useful for packages you need to test locally or packages with minimal build dependencies. Otherwise use *sbuild(1)*[34].

To do a binary-only build of a package, run:

[30] https://manpages.ubuntu.com/manpages/noble/en/man1/debuild.1.html
[31] https://manpages.ubuntu.com/manpages/noble/en/man1/dpkg-buildpackage.1.html
[32] https://manpages.ubuntu.com/manpages/noble/en/man7/debhelper.7.html
[33] https://manpages.ubuntu.com/manpages/noble/en/man1/dh_clean.1.html
[34] https://manpages.ubuntu.com/manpages/noble/en/man1/sbuild.1.html

```
$ debuild -b
```

Resources

- Chapter 6. Building the package (Debian New Maintainers' Guide)[35]
- SimpleSbuild (Ubuntu Wiki)[36]
- sbuild (Debian Wiki)[37]

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

2.1.4. Install built packages

You have a built *binary packages* of a *source package* and want to install it (e.g. to test the packages). This article demonstrates multiple ways how you can achieve that.

Using your package manager

You can use the *apt(8)*[38], *apt-get(8)*[39] or *dpkg(1)*[40] *package manager* to install or uninstall packages on an Ubuntu installation.

Note

apt(8)[41] is intended to be used interactively by humans and does not guarantee a stable *command line interface* (suitable for machine-readability) while *apt-get(8)*[42] is intended for unattended usage, for example, in scripts.

dpkg(1)[43] is a package manager for *Debian*-based systems. It can install, remove, and build packages, but unlike the *APT* package management systems, it cannot automatically download and install packages or their dependencies.

See also the package management[44] guide from the *Ubuntu Server* documentation for more details.

[35] https://www.debian.org/doc/manuals/maint-guide/build.html
[36] https://wiki.ubuntu.com/SimpleSbuild
[37] https://wiki.debian.org/sbuild
[38] https://manpages.ubuntu.com/manpages/noble/en/man8/apt.8.html
[39] https://manpages.ubuntu.com/manpages/noble/en/man8/apt-get.8.html
[40] https://manpages.ubuntu.com/manpages/noble/en/man1/dpkg.1.html
[41] https://manpages.ubuntu.com/manpages/noble/en/man8/apt.8.html
[42] https://manpages.ubuntu.com/manpages/noble/en/man8/apt-get.8.html
[43] https://manpages.ubuntu.com/manpages/noble/en/man1/dpkg.1.html
[44] https://ubuntu.com/server/docs/package-management

Install `.deb` files

apt

apt-get

dpkg

You can install one or multiple `.deb` files by using `apt install` command:

```
sudo apt install PACKAGE.deb...
```

For example, to install the `hello_2.10-3_amd64.deb` binary package file (version `2.10-3` of the `hello` package for the `amd64` architecture) you need to run:

```
sudo apt install 'hello_2.10-3_amd64.deb'
```

You can install one or multiple `.deb` files by using `apt-get install` command:

```
sudo apt-get install PACKAGE.deb...
```

For example, to install the `hello_2.10-3_amd64.deb` binary package file (version `2.10-3` of the `hello` package for the `amd64` architecture) you need to run:

```
sudo apt-get install hello_2.10-3_amd64.deb
```

You can install one or multiple `.deb` files by using `dpkg --install` command:

```
sudo dpkg --install PACKAGE.deb...
```

For example, to install the `hello_2.10-3_amd64.deb` binary package file (version `2.10-3` of the `hello` package for the `amd64` architecture) you need to run:

```
sudo dpkg --install hello_2.10-3_amd64.deb
```

Uninstall packages

Installed packages often setup configuration files and create other data files. When you want to uninstall a package you have to decide if you want to keep these files or want to delete them too.

Keeping configuration files can be useful to avoid having to reconfigure a package if it is reinstalled later, but this may have side-effects when testing to install multiple packages.

Keep the configuration files

apt

apt-get

dpkg

You can uninstall one or multiple packages and **keep** their configuration files by using the `apt remove` command:

```
sudo apt remove PACKAGE-NAME...
```

For example, to uninstall the currently installed `hello` package and keep its configuration files you need to run:

```
sudo apt remove hello
```

You can uninstall one or multiple packages and **keep** their configuration files by using the `apt-get remove` command:

```
sudo apt-get remove PACKAGE-NAME...
```

For example, to uninstall the currently installed `hello` package and keep its configuration files you need to run:

```
sudo apt-get remove hello
```

You can uninstall one or multiple packages and **keep** their configuration files by using the `dpkg --remove` command:

```
sudo dpkg --remove PACKAGE-NAME...
```

For example, to uninstall the currently installed `hello` package and keep its configuration files you need to run:

```
sudo dpkg --remove hello
```

Delete the configuration files

apt

apt-get

dpkg

You can uninstall one or multiple packages and **delete** their configuration files by using the `apt purge` command:

```
sudo apt purge PACKAGE-NAME...
```

For example, to uninstall the currently installed `hello` package and delete its configuration files you need to run:

```
sudo apt purge hello
```

You can uninstall one or multiple packages and **delete** their configuration files by using the `apt-get purge` command:

```
sudo apt-get purge PACKAGE-NAME...
```

For example, to uninstall the currently installed `hello` package and delete its configuration files you need to run:

```
sudo apt-get purge hello
```

You can uninstall one or multiple packages and **delete** their configuration files by using the `dpkg --purge` command:

```
sudo dpkg --purge PACKAGE-NAME...
```

For example, to uninstall the currently installed `hello` package and delete its configuration files you need to run:

```
sudo dpkg --purge hello
```

Install packages from a PPA

Using `add-apt-repository`

The `add-apt-repository` command adds a *Repository* (e.g. a *Personal Package Archive* (PPA) from *Launchpad*) to the `/etc/apt/sources.list.d` directory (see the *`sources.list(5)`*[45] manual page for more details), so you can install the packages provided by the repository like any other package from the *Ubuntu Archive*.

```
sudo add-apt-repository ppa:LP-USERNAME/PPA-NAME
```

`LP-USERNAME`
: The username of the Launchpad user who owns the PPA.

`PPA-NAME`
: The name of the PPA.

For example, to add the Launchpad PPA with the name `hello` of the Launchpad user `dviererbe` you need to run:

```
sudo add-apt-repository ppa:dviererbe/hello
```

Then, you can install, just as normal, the `hello` package contained in the PPA:

apt

apt-get

```
sudo apt install hello
```

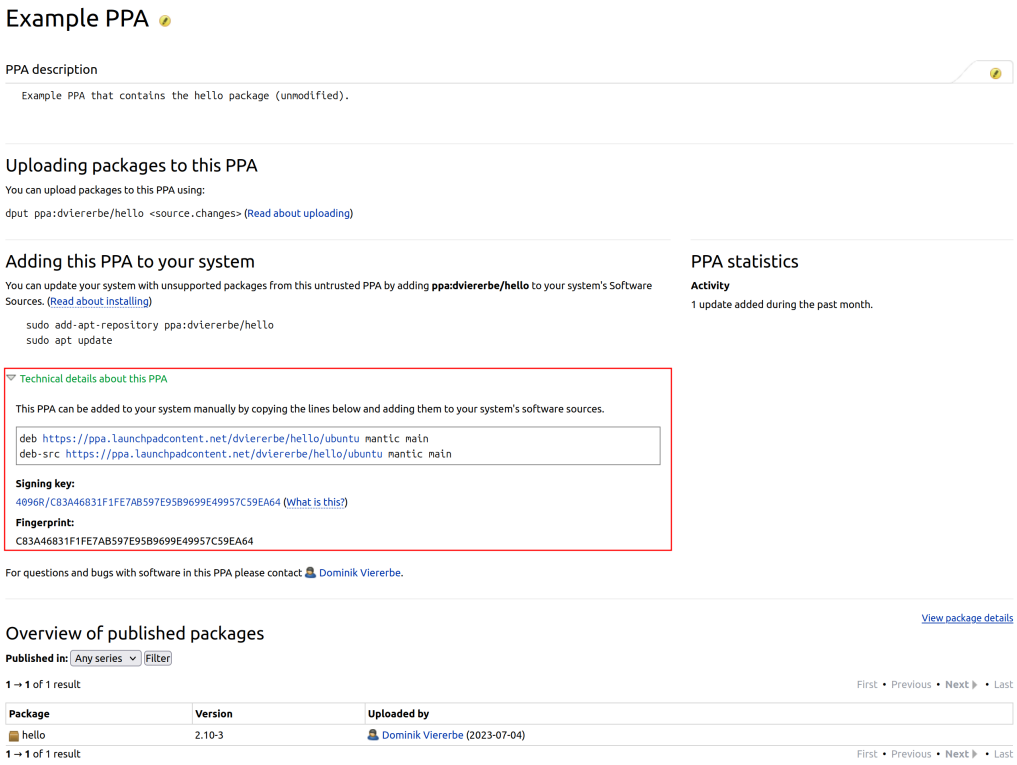
```
sudo apt-get install hello
```

See the *`add-apt-repository(1)`*[46] manual page for more details.

Add PPA manually

When you visit the web interface of the Launchpad PPA you want to add, you can see a text reading something like “Technical details about this PPA”. When you click on the text, it will unfold and show the details you need to add the PPA.

[45] https://manpages.ubuntu.com/manpages/noble/en/man5/sources.list.5.html
[46] https://manpages.ubuntu.com/manpages/noble/en/man1/add-apt-repository.1.html



The steps to add the PPA are as follows:

1. Add the PPA entry to `/etc/apt/sources.list.d` directory

   ```
   sudo editor /etc/apt/sources.list.d/launchpad_ppa.sources
   ```

   Add the following lines (substituting `LAUNCHPAD-USERNAME` AND `PPA-NAME` for your own case) and save the file:

   ```
   deb https://ppa.launchpadcontent.net/LAUNCHPAD-USERNAME/PPA-NAME/ubuntu
   SERIES main
   deb-src https://ppa.launchpadcontent.net/LAUNCHPAD-USERNAME/PPA-NAME/ubuntu
   SERIES main
   ```

2. Add the of the PPA *Signing Key* to `/etc/apt/trusted.gpg.d` directory.

   The following command will download the PPA signing key from the *Ubuntu Keyserver* and store it in the correct format in the `/etc/apt/trusted.gpg.d` directory. Substitute `SIGNING_KEY` with the Fingerprint (see picture above) of the PPA signing key.

   ```
   wget --quiet --output-document - \
   "https://keyserver.ubuntu.com/pks/lookup?op=get&search=0x${SIGNING_KEY,,}" \
   | sudo gpg --output /etc/apt/trusted.gpg.d/launchpad-ppa.gpg --dearmor -
   ```

3. Update the package information:

   apt

   apt-get

   ```
   sudo apt update
   ```

```
sudo apt-get update
```

4. Install the package from the PPA:

   apt

   apt-get

```
sudo apt install PACKAGE-NAME
```

```
sudo apt-get PACKAGE-NAME
```

For example, here is the full script to add the Launchpad PPA named `hello` of the user `dviererbe` and install the `hello` package from it.

```
sudo sh -c 'cat <<EOF > /etc/apt/sources.list.d/launchpad_ppa2.sources
deb https://ppa.launchpadcontent.net/dviererbe/hello/ubuntu mantic main
deb-src https://ppa.launchpadcontent.net/dviererbe/hello/ubuntu mantic main
EOF'

SIGNING_KEY=C83A46831F1FE7AB597E95B9699E49957C59EA64
wget --quiet --output-document - \
"https://keyserver.ubuntu.com/pks/lookup?op=get&search=0x${SIGNING_KEY,,}" \
| sudo gpg --output /etc/apt/trusted.gpg.d/launchpad-ppa.gpg --dearmor -

sudo apt update
sudo apt install hello
```

Download the `.deb` files

You can also download binary packages (`.deb` files) from a Launchpad PPA and install them with a package manager (like demonstrated in the section *Install .deb files* (page 24)).

Using `pull-ppa-debs`

The `pull-ppa-debs` command downloads the `.deb` files of one specific binary package or all binary packages, which are built by a source package in a Launchpad PPA.

```
pull-ppa-debs --ppa LP-USERNAME/PPA-NAME [--arch ARCH] PKG-NAME [SERIES|VERSION]
```

--ppa LP-USERNAME/PPA-NAME
: The PPA to download the binary package(s) from.

  LP-USERNAME
  : The username of the Launchpad user who owns the PPA.

  PPA-NAME
  : The name of the PPA.

--arch ARCH
: The architecture of the binary package(s) to download. Defaults to the system architecture of your host machine.

PKG-NAME
: The name of the package to download. This can be the name of the source package to download all binary packages build by the source package or just the name of one specific binary package.

SERIES
: Downloads the package with the latest version that targets the Ubuntu *Series* with the specified name. Defaults to the *Current Release in Development*.

VERSION
: The version of the package to download.

The `pull-ppa-debs` command is part of the `ubuntu-dev-tools` package. You need to install it, before you can use it:

```
sudo apt install ubuntu-dev-tools
```

Tip

The `ubuntu-dev-tools` package also provides the commands:

- `pull-lp-debs` (to download binary packages from Launchpad) and
- `pull-debian-debs` (to download binary packages from the Debian archive).

For example, on an *amd64* machine, the following command will download the binary package named `hello` and targeting `amd64` from the Launchpad PPA named `hello` of the Launchpad user `dviererbe`:

```
pull-ppa-deb --ppa dviererbe/hello hello
```

The downloaded file will be `hello_2.10-3_amd64.deb`.

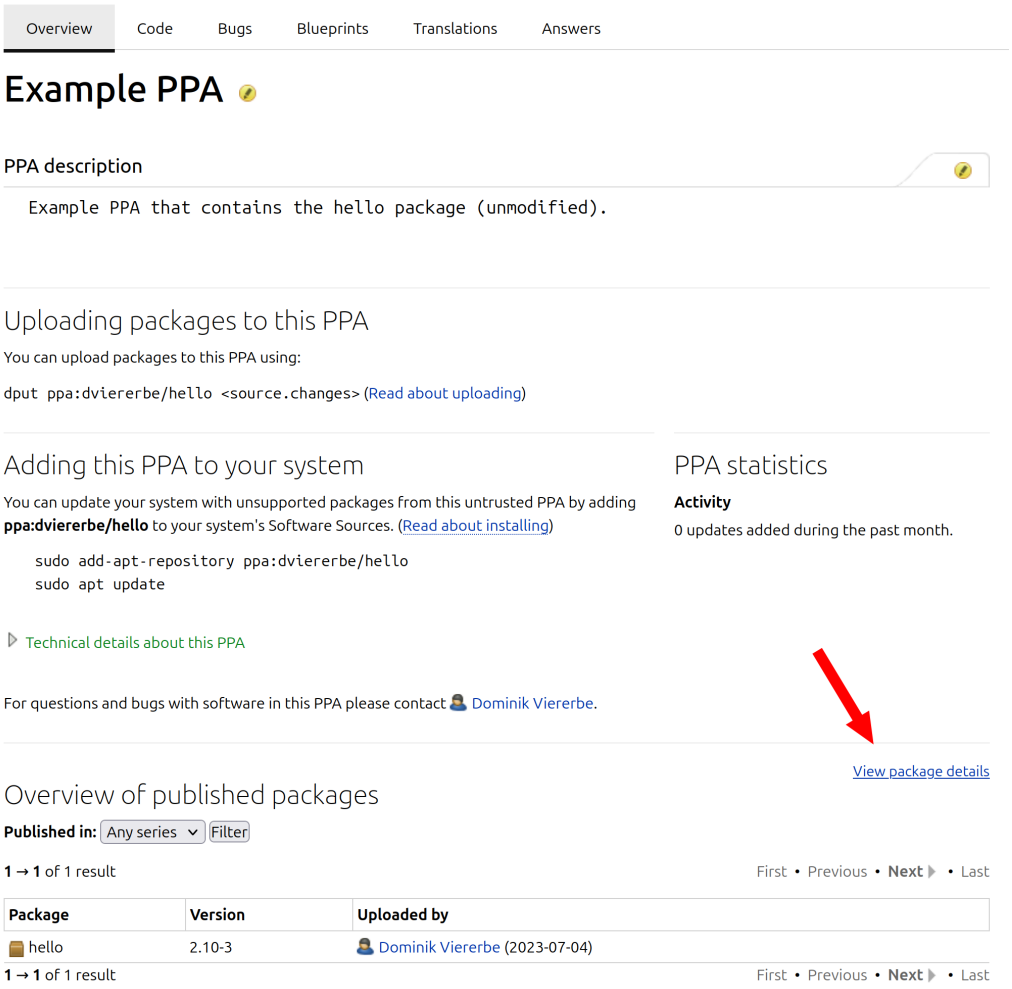
See the *`pull-pkg(1)`*[47] manual page for more details.

Using the Launchpad web interface

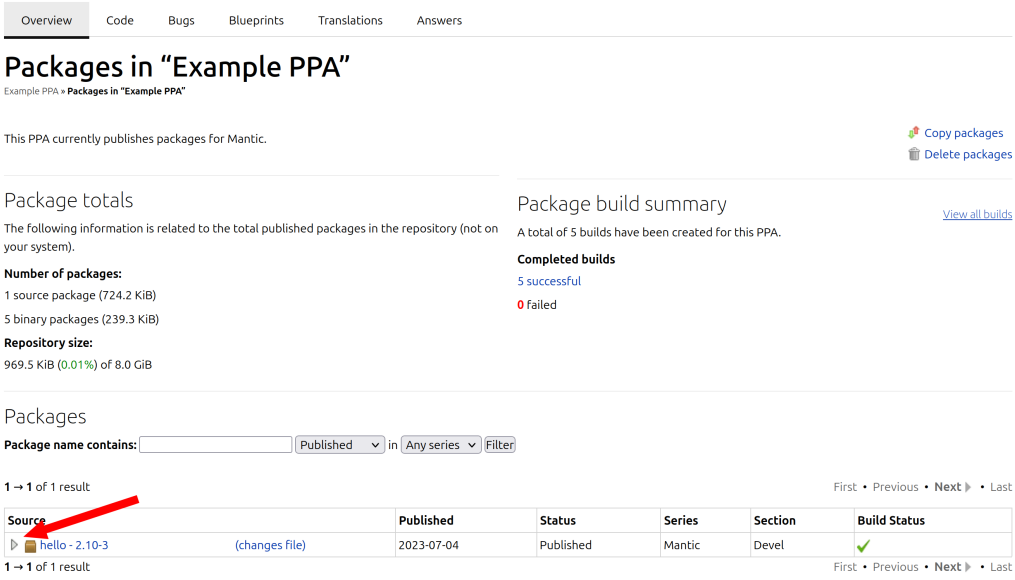
You can download `.deb` files from a Launchpad PPA via the web interface like this:

1. Go to the Launchpad PPA web interface and click on the link called “View package details”:

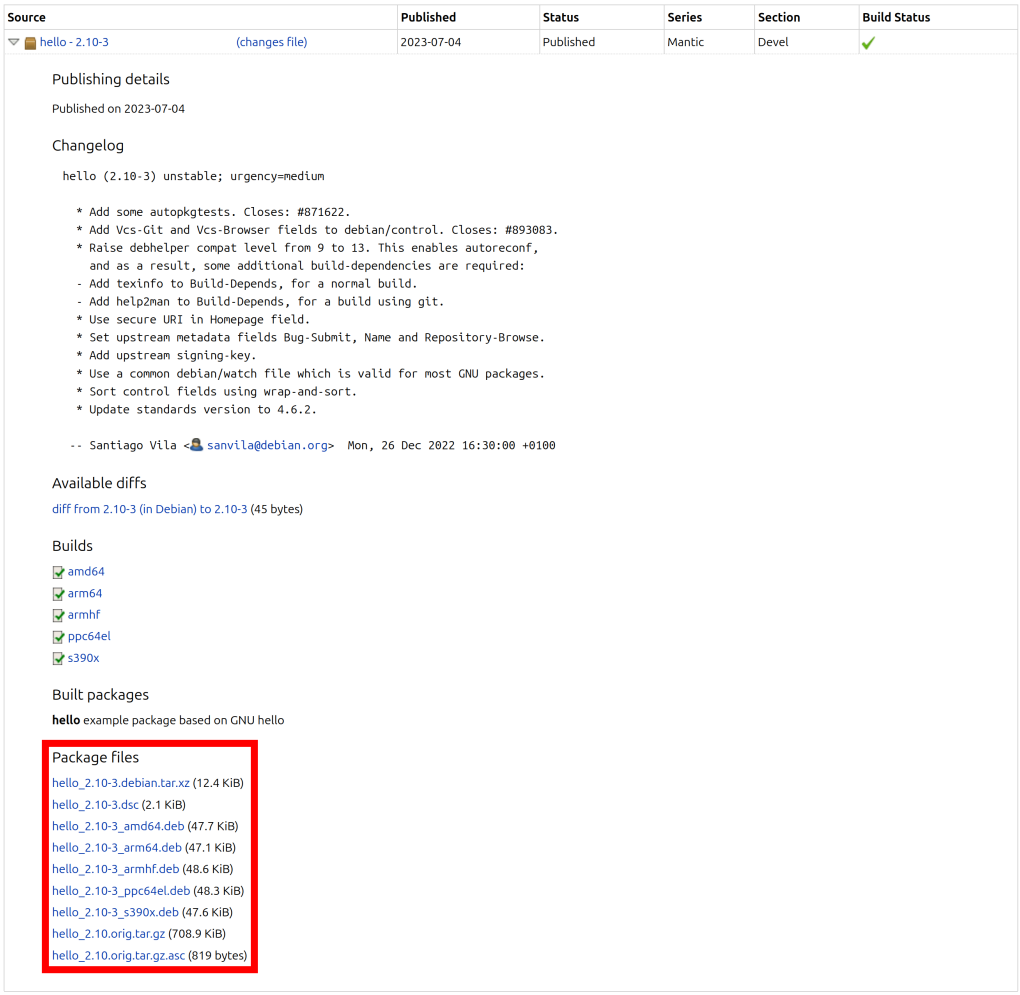
[47] https://manpages.ubuntu.com/manpages/noble/en/man1/pull-pkg.1.html



2. Expand the details of the package you want to download by clicking on the little triangle next to the name of the package:



3. Download the file(s) you need from the "Package files" section by clicking on the respective links:



Source	Published	Status	Series	Section	Build Status
hello - 2.10-3 (changes file)	2023-07-04	Published	Mantic	Devel	✓

Publishing details

Published on 2023-07-04

Changelog

```
hello (2.10-3) unstable; urgency=medium

  * Add some autopkgtests. Closes: #871622.
  * Add Vcs-Git and Vcs-Browser fields to debian/control. Closes: #893083.
  * Raise debhelper compat level from 9 to 13. This enables autoreconf,
    and as a result, some additional build-dependencies are required:
  - Add texinfo to Build-Depends, for a normal build.
  - Add help2man to Build-Depends, for a build using git.
  * Use secure URI in Homepage field.
  * Set upstream metadata fields Bug-Submit, Name and Repository-Browse.
  * Add upstream signing-key.
  * Use a common debian/watch file which is valid for most GNU packages.
  * Sort control fields using wrap-and-sort.
  * Update standards version to 4.6.2.

 -- Santiago Vila <sanvila@debian.org>  Mon, 26 Dec 2022 16:30:00 +0100
```

Available diffs

diff from 2.10-3 (in Debian) to 2.10-3 (45 bytes)

Builds

amd64
arm64
armhf
ppc64el
s390x

Built packages

hello example package based on GNU hello

Package files

hello_2.10-3.debian.tar.xz (12.4 KiB)
hello_2.10-3.dsc (2.1 KiB)
hello_2.10-3_amd64.deb (47.7 KiB)
hello_2.10-3_arm64.deb (47.1 KiB)
hello_2.10-3_armhf.deb (48.6 KiB)
hello_2.10-3_ppc64el.deb (48.3 KiB)
hello_2.10-3_s390x.deb (47.6 KiB)
hello_2.10.orig.tar.gz (708.9 KiB)
hello_2.10.orig.tar.gz.asc (819 bytes)

Resources

- Ubuntu Server documentation – Package management[48]
- Ubuntu wiki – Installing Software[49]
- manual page *add-apt-repository(1)*[50]
- manual page *pull-pkg(1)*[51]

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

[48] https://ubuntu.com/server/docs/package-management
[49] https://help.ubuntu.com/community/InstallingSoftware
[50] https://manpages.ubuntu.com/manpages/noble/en/man1/add-apt-repository.1.html
[51] https://manpages.ubuntu.com/manpages/noble/en/man1/pull-pkg.1.html

2.1.5. Run tests

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

2.1.6. Upload packages to a PPA

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

2.1.7. Patch management

This article demonstrates how to manage the *patches* of a *source package* in the `3.0 (quilt)` format.

See the *patches explanation article* (page 56) for more background information about patches in the context of *Ubuntu*.

Note

If the format is `3.0 (native)`, this article is not of interest for you and simply write your changes to the files. There is no need to explicitly create and track a patch for native packages. See the *package model* (page 51) article for more information about package formats.

As the source package format implies, we will use the *`quilt(1)`*[52] tool to manage the patches of a source package. Quilt manages patches like a *Stack*. It maintains a list of patches (also called "series") that get applied one after another from top to bottom in the order they are listed in `debian/patches/series`, excluding lines starting with #.

Important

Quilt will create a `.pc/` directory at the source package root directory. This is the location where Quilt will store control files similar to the `.git/` folder of a *Git* repository.

[52] https://manpages.ubuntu.com/manpages/noble/en/man1/quilt.1.html

Before you commit any changes (e.g., with *git-ubuntu*) or attempt to build the source package, **do not forget to unapply all patches and delete the directory:**

```
quilt pop -a && rm -r .pc
```

You can avoid removing `.pc` if it exists in the `.gitignore` file, or if you can otherwise avoid it when running `git add`.

Prerequisites

If you haven't already, install *quilt(1)*[53]:

```
sudo apt update && sudo apt install quilt
```

By running the following script once in a terminal you will configure *quilt(1)*[54] to look for patches in the `debian/patches/` directory if the `quilt` command is invoked within a *source package* directory:

```
cat <<'EOF' > ~/.quiltrc
#!/usr/bin/env bash
set -euo pipefail

# find the root of a debian source tree:
SourcePackageRoot="${PWD}"
while [ ! -s "$SourcePackageRoot/debian/source/format" ]
do
    if [ "${SourcePackageRoot}" = '/' ]
    then
        echo -e '\033[1;33mWARNING\033[0m: You are not in a debian source tree!'
        exit 0
    fi

    SourcePackageRoot="$(readlink --canonicalize-existing "${SourcePackageRoot}/..
")"
done

if ! grep --silent --fixed-strings '3.0 (quilt)' \
    "${SourcePackageRoot}/debian/source/format"
then
    echo -e '\033[1;33mWARNING\033[0m: This source package does \033[1mNOT\033[0m
use the 3.0 (quilt) format. The corresponding defaults defined in ~/.quiltrc do
not get applied.'
    exit 0
fi

# tell quilt where to find patches for a 3.0 (quilt) source package
: "${QUILT_PATCHES:="${SourcePackageRoot}/debian/patches"}"

# create the quilt control files directory at the root of the source package
```

(continues on next page)

[53] https://manpages.ubuntu.com/manpages/noble/en/man1/quilt.1.html
[54] https://manpages.ubuntu.com/manpages/noble/en/man1/quilt.1.html

(continued from previous page)

```
: "${QUILT_PC:="${SourcePackageRoot}/.pc"}"

# default options for the patch(1) tool
: "${QUILT_PATCH_OPTS:="--reject-format=unified"}"

# how quilt output should be colored
: "${QUILT_COLORS:="diff_hdr=1;32:diff_add=1;34:diff_rem=1;31:diff_hunk=1;33:diff_
ctx=35:diff_cctx=33"}"

# set default arguments for quilt commands:
: "${QUILT_DIFF_ARGS:="-p ab --no-timestamps --no-index --color=auto"}"
: "${QUILT_PATCHES_ARGS:="--color=auto"}"
: "${QUILT_PUSH_ARGS:="--color=auto"}"
: "${QUILT_REFRESH_ARGS:="-p ab --no-timestamps --no-index"}"
: "${QUILT_SERIES_ARGS:="--color=auto"}"
EOF
```

Note

If you later want to undo this configuration – simply delete `~/.quiltrc`:

```
rm ~/.quiltrc
```

List patches

List all available patches:

```
quilt series
```

This will also color code which patches are applied (green), which patch is the latest applied patch (yellow) and which patches are unapplied (white).

List applied patches:

```
quilt applied
```

Display the topmost applied patch:

```
quilt top
```

List unapplied patches:

```
quilt unapplied
```

Note

Quilt patches are applied from top to bottom in the order they are listed.

Apply patches

Apply the next patch:

```
quilt push
```

Apply all patches:

```
quilt push -a
```

Apply the next `N` patches

```
quilt push N
```

Apply all patches until (including) a specific patch:

```
quilt push PATCH-NAME
```

This can also be the path of the patch (allowing for auto-completion):

```
quilt push debian/series/PATCH-NAME
```

Unapply patches

This works similar to applying patches.

Unapply the patch on top:

```
quilt pop
```

Unapply all patches:

```
quilt pop -a
```

Unapply the `N` topmost applied patches

```
quilt pop N
```

Unapply all patches until (excluding) a specific patch:

```
quilt pop PATCH-NAME
```

This can also be the path of the patch (allowing for auto-completion):

```
quilt pop debian/series/PATCH-NAME
```

Verify patches

Now that we know how to apply and unapply patches we can verify if all patches apply and unapply cleanly. This can be useful when you merge changes from Debian into an Ubuntu package and want to check if everything is still in order.

1. We verify that all patches apply cleanly:

```
quilt push -a
```

2. We verify that all patches unapply cleanly:

```
quilt pop -a
```

3. (optional) We remove the Quilt control file folder:

```
rm -r .pc
```

Show details about a patch file

Print the header of the topmost applied or specified patch:

```
quilt header [PATCH-NAME]
```

Print the list of files that the topmost applied or specified patch changes:

```
quilt files [PATCH-NAME]
```

Print the changes by the topmost applied or specified patch to the specified file(s) in a diff format. If no files are specified, all files that are changes are included.

```
quilt diff [-P PATCH-NAME] [FILE-PATH ...]
```

Rename a patch file

Rename the topmost applied or specified patch:

```
quilt rename [-P PATCH-NAME] NEW-PATCH-NAME
```

Remove a patch file

Remove the topmost applied or specified patch from the `debian/patches/series` file. Use `-r` to also delete the patch file from the `debian/patches` directory:

```
quilt delete [-r] [PATCH-NAME]
```

Generate a patch file

1. Create a new patch after the topmost applied patch:

```
quilt new PATCH-NAME
```

> **Note**
>
> It is best practice to read the existing patch-file-names in `debian/patches` and ensure your new patch name is consistent with the existing ones.

2. Edit files outside the `debian/` directory by following the same steps as outlined by the following section *Edit a patch file* (page 37).

Edit a patch file

1. Apply all patches until the patch we want to edit:

```
quilt push PATCH-NAME
```

2. There are multiple approaches how to edit the patch file:

   - Edit the patch header:

```
quilt header -e
```

> **Tip**
>
> If the patch does not already have a header and you want to add one; add the `--dep3` flag to insert a *DEP 3* patch header template:
>
> ```
> quilt header --dep3 -e
> ```

> **Tip**
>
> See the *DEP 3 – Patch file headers reference* (page 99) which lists and briefly explains standard DEP 3 compliant fields and shows sample DEP 3 compliant headers.

   - Edit specific file(s) with a text editor after adding the changes to the patch file:

```
quilt edit FILE-PATH ...
```

> **Note**
>
> Opens the files in `$EDITOR` – this is usually your default terminal editor.

   - Edit specific file(s) manually (without immediately opening an editor):

     a. Check which files are already changed by the patch file:

```
quilt files
```

     b. If you want to edit file(s) that the patch currently does NOT change – add these files to the patch before editing them:

```
quilt add FILE-PATH ...
```

> **Note**
>
> You can directly edit files which are already changed by the patch.

> **Tip**
>
> To see the changes of the patch file to a specific file:
>
> ```
> quilt diff FILE-PATH
> ```
>
> To see the changes you made of the patch file:
>
> ```
> quilt diff -z
> ```

c. Save the changes to the patch file:

```
quilt refresh
```

- Delete the changes of a patch to specific file(s):

```
quilt remove FILE-PATH ...
```

3. (recommended) If there are patches after the patch you have edited – *verify that all patches still apply cleanly* (page 35).

Import a patch file

Insert patch files following the current topmost applied patch:

```
quilt import PATCH-FILE-PATH ...
```

> **Important**
>
> The patch files have to be outside the `debian/patches/` directory.

> **Note**
>
> The imported patches do not get applied automatically. You must *apply the patches* (page 35) after importing them.

Resources

- *DEP 3 – Patch file headers (reference)* (page 99)
- *Patches (explanation)* (page 56)
- manual page *quilt(1)*[55]
- Debian wiki – Using quilt in Debian source packages[56]

> **Caution**

[55] https://manpages.ubuntu.com/manpages/noble/en/man1/quilt.1.html
[56] https://wiki.debian.org/UsingQuilt

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

2.1.8. Propose changes

This guide walks you through the process for proposing changes to Ubuntu. The process is straightforward. When you find a problem, you obtain the code, work on a solution, test the fix, push your changes to *Launchpad*, and then request a review and merge.

Attention

There are information placed within angle brackets in this guide. Ensure you replace them with the appropriate values. For example, replace `<package-name>` with the name of the package you are working on.

Find a bug to fix

Start by identifying an issue to work on. This can be a bug you encountered while using an application, a problem described in a bug report, or a known issue in the Ubuntu community.

You can also explore known bugs using these resources:

- Bitesize bugs on Launchpad[57]: These are small, well-scoped bugs that are great for new contributors.
- One Hundred Papercuts[58]: This resource focuses on fixing minor bugs that negatively affect the user experience.
- Launchpad's bug tracker[59] and Debian's bug trackers[60]: These contain issue reports for Ubuntu and Debian packages.

Evaluate the bug report

Once you find a bug report on Launchpad that you want to work on, the next step is to evaluate the bug.

Read the bug description carefully and look for:

- steps to reproduce the problem
- crash logs or terminal output
- details about the affected version and package
- attached `.crash` files or Apport crash files[61]

[57] https://bugs.launchpad.net/ubuntu/+bugs?field.tag=bitesize
[58] https://launchpad.net/hundredpapercuts
[59] https://bugs.launchpad.net/ubuntu
[60] https://www.debian.org/Bugs/
[61] https://github.com/canonical/ubuntu-maintainers-handbook/blob/main/PackageFixing.md#evaluate-the-bug

For example, this bug report[62] shows a segmentation fault in `postconf` on Ubuntu 18.04. It includes logs from `/var/log/kern.log`, shell commands that reproduce the issue, and metadata about the system environment. This information helps confirm whether the bug still affects current versions and if the report is complete.

If the bug includes a `.crash` file, extract and inspect the stack trace. Use the information to better understand where the failure occurred in the code.

For more details, see Evaluate the bug[63].

Identify the source package

After selecting a bug to fix, the first step is to identify the source package that contains the code related to the issue.

Start by identifying the name of the binary package. If you know the path or part of the filename of the affected program, run the following command:

```
apt-file find <filename-or-path>
```

Note

Replace `<filename-or-path>` with either the full path or part of the filename.

For example, both of the following work:

```
apt-file find /usr/games/bumprace
apt-file find bumprace
```

Running the command returns an output similar to:

```
bumprace: /usr/games/bumprace
```

In the preceding output, the part before the colon is the name of the binary package.

After identifying the name of the binary package, the next step is to find the source package. Use the following command:

```
apt show <binary-package-name>
```

Note

Replace `<binary-package-name>` with the actual name of the binary package.

For example:

```
apt show bumprace
```

Now check the output for the `Source` field. This field indicates the name of the source package.

[62] https://bugs.launchpad.net/ubuntu/+source/postfix/+bug/1753470
[63] https://github.com/canonical/ubuntu-maintainers-handbook/blob/main/PackageFixing.md#evaluate-the-bug

It's possible for the name of a binary package to be the same as its source package. If this is the case, then the `apt show <binary-package-name>` command won't display the `Source` field in its output. In such cases, you can assume the source package name is the same as the binary package name.

Check if the bug has been fixed

Once you identify the source package, make sure the issue still exists. A fix may already exist in a newer Ubuntu release, in Debian, or upstream. Checking first will save time and avoid duplicate work.

Follow the steps in the following subsections to check whether the problem has already been addressed.

Check if the bug is fixed in a newer Ubuntu

Use `rmadison` to review the versions of the package available across Ubuntu releases.

```
rmadison <package-name>
```

This shows you which versions are available in different Ubuntu series. Look for a newer version than the one you are using. If a fix was introduced in a later version, check the changelog or commit history to verify.

To review changes, clone the package with *git ubuntu*:

```
git ubuntu clone postfix postfix
cd postfix
git log -b pkg/ubuntu/<ubuntu-series>
```

Look through the commit messages and patch files to identify if the issue has been resolved.

Check if the bug is fixed in Debian

Debian is a key source for Ubuntu packages. Search for bug reports or patches applied there.

First, check Debian's bug tracker using the URL `https://bugs.debian.org/src:<package-name>`.

To inspect changes in more detail, find the source repository used by Debian. You can do this in a few ways:

- use `debcheckout`:

```
debcheckout <package-name>
cd <package-name>
git log
```

- look for the `Vcs-Git` and `Vcs-Browser` fields from the `apt showsrc` command output. These point to the package's source code repository and its web interface:

```
apt showsrc --only-source <package-name>
```

Now, look for commit messages that describe fixes relevant to your issue. If a bug number is referenced, open the link and review the context.

Check if the bug is fixed upstream

If the problem originates from the software itself and not the package, investigate upstream. Each project has its own bug tracker and code repository.

You can find the upstream project by doing the following:

- search the package homepage listed by running the command `apt show <package>`
- look up the project through web search if no homepage is set
- check the metadata in the package description or Debian tracker

Once you find the upstream repository, do the following:

1. look through open and closed issues
2. search the commit history for relevant fixes
3. clone the upstream Git repository if available and inspect the logs

If upstream has resolved the problem, consider if that version has reached Debian or Ubuntu. If not, you may propose packaging the new version or backporting the patch.

Offer to help

Once you confirm the issue still exists, a bug report is open, and no one is working on it, you can offer to help. This step signals your interest in resolving the issue and helps prevent duplicated efforts.

Start by commenting on the bug report in Launchpad. Let others know that you intend to work on the issue. Include any relevant details you have, such as:

- when and how the bug occurred
- how you plan to fix the issue, or what you've tried so far
- any testing you've done or plan to do

If the bug doesn't yet exist in Launchpad, create a new bug report. Provide a clear title and description. Explain how the issue can be reproduced, and add logs or screenshots if helpful.

Get the source code

Once you're assigned to the bug, get the source code for the affected package. You can get the source code using any of these four methods:

- `git-ubuntu`
- `pull-pkg`
- `apt-get source`
- `dget`

For detailed instructions on using these methods to get the source code, see *Get the source of a package* (page 12).

Create a patch to fix the issue

You may need to create a patch to make changes to a package. Start by checking where your changes are located. If your changes are only within the `debian/` directory, for example, in `debian/control`, you don't need to create a patch. However, if you changed upstream source code, that is anything outside `debian/`, then you must create a patch and include it in `debian/patches`.

There are two main methods for creating patches for Ubuntu packages. The method you choose will depend on the workflow that the package source uses:

- If the package uses **quilt**, use the `quilt` tool to create and manage patches. To learn how to create a patch using `quilt`, see Making a patchfile[64].
- If the package is maintained using `git-ubuntu`, commit your changes directly in Git.

Document the fix

It's important to document your changes so future developers can understand your reasoning and assumptions without having to guess.

Explain your changes in the `debian/changelog` file. This file tracks every change uploaded to Ubuntu or Debian, and future developers rely on it to understand what changed, where it happened, and why.

Run the following command to create a new changelog entry:

```
dch -i
```

This command generates a new entry and opens your text editor. The top and bottom lines will be filled out for you automatically. The top line includes the package name, version, Ubuntu release, and urgency. The bottom line shows your name, email, and a timestamp.

You should also write a short and informative message between the top and bottom lines. This message should include:

- where you made the change, such as file or component
- what the change does
- why you made the change
- link to the Launchpad bug or mailing list discussion, if available

An example of the changelog entry is as follows:

```
my-package <version> UNRELEASED; urgency=low

* fix crash in system monitor when reading temperature sensors
    - updated <script.py> to handle missing sensor values
    - added error handling to prevent crashes

-- name <name@example.com>  Tue, 13 May 2025 15:42:10 +0000
```

Reference Launchpad bugs like this:

[64] https://github.com/canonical/ubuntu-maintainers-handbook/blob/main/DebianPatch.md

```
LP: #<bug-number>
```

This ensures the bug will close automatically when the fix is uploaded.

Test the fix

Run package tests to check that your change doesn't introduce regressions. Ubuntu uses *autopkgtest* to automate this process. You can run tests in several ways:

- in a local virtual machine (VM)
- through a *Personal Package Archive* (PPA) on Launchpad
- in a container

For local testing, use a VM or container. The *autopkgtest* tool builds test images and runs the tests in an isolated environment. Use this method when you want to debug failures or verify changes before uploading to a PPA. If your testbed needs to reboot or be isolated, use a VM or container as defined in the package's `debian/tests/control` file.

You can also use PPA-based method whenever possible. It produces results closest to what Launchpad runs for archive packages. After uploading your package to a PPA and building it, you can trigger tests using the `PPA` tool from `ppa-dev-tools`. You will need special permissions to launch these tests. Ask for help in the `#ubuntu-devel` IRC channel if needed.

To learn how to set up and run these test methods, see Running package tests[65].

Submit the fix

Once you've documented and saved your changes in a new changelog entry, run `debuild`:

```
debuild -S -d
```

The command signs the changes in the file. After that, you can submit your fix by opening a merge proposal. For details on how to do this, see the section on Merge proposal[66] in the Ubuntu Maintainer's Handbook.

In many cases, Debian would benefit from the fix as well. Submitting to Debian is considered best practice because it ensures that a wider audience receives the fix. You can submit the fix to Debian by running:

```
submittodebian
```

Running the preceding command walks you through a series of steps to ensure the bug report ends up in the correct place. Be sure to review the diff again to confirm it doesn't include unrelated changes you made earlier.

Also, ensure you add a clear description of the fix to the inclusion request.

If everything goes well, you will get an email from Debian's bug tracking system with more information. This may take a few minutes.

Sometimes it's best to get your fix included in Debian first. It will then flow down to Ubuntu automatically. In that case, skip the following steps.

[65] https://github.com/canonical/ubuntu-maintainers-handbook/blob/main/PackageTests.md
[66] https://github.com/canonical/ubuntu-maintainers-handbook/blob/main/MergeProposal.md

For security updates or updates to stable releases, the fix might already be in Debian or intentionally ignored. In these cases, follow the process described here.

If you're doing a security or stable release update, read the article on Security and stable release updates.

You can also follow this process when dealing with Ubuntu-only packages that don't build correctly, or with issues that affect Ubuntu specifically.

If you're submitting your fix to Ubuntu, generate a `debdiff`. A `debdiff` shows the difference between two Debian source packages. The command is also called `debdiff`, and it comes from the `devscripts` package. For full details, see the manpages for debdiff[67].

To compare two source packages, use the `.dsc` files as arguments:

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

Compare the original `.dsc` file with the one you generated after making your changes. 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 and attach it to the bug report:

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

The format of the filename shown in `1-1.0-1ubuntu1.debdiff` has some meaning:

1. `1-` tells the sponsor that this is the first revision of your patch.
2. `1.0-1ubuntu1` shows the version you are working on.
3. `.debdiff` makes it clear that it's a `debdiff` file.

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

Next, go to the bug report on Launchpad. Log in, then click **Add attachment or patch** near the comment box. Attach the `debdiff` and leave a comment. Explain how the patch can be applied and what testing you've done.

Here's an example:

```
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.
```

Mark the attachment as a patch. This will notify the Ubuntu Sponsors team. Also, make sure you're subscribed to the bug report so you get updates.

You will usually get a review within a few hours to a few weeks. If it takes too long, join `#ubuntu-motu` on Libera Chat and ask for help. Stay in the channel until someone responds. They will guide you through your next steps.

After review, the sponsor might upload your fix, request changes, or reject it. If changes are needed, follow the same steps and submit a new `debdiff` to the bug. If the fix is rejected because it's not a fit for Ubuntu, you might need to send it to Debian instead.

[67] https://manpages.debian.org/testing/devscripts/debdiff.1.en.html

If you have questions, email `ubuntu-motu@lists.ubuntu.com` or join `#ubuntu-motu` on Libera Chat. You will find people who share your passion for improving open source and making the world better.

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

2.1.9. Use schroots

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

2.1.10. Request a freeze exception

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

2.1.11. Merge a package from Debian

This article is still work in progress. You can use the Ubuntu Maintainer Handbook[68] in the meantime.

Note

Be aware that the Ubuntu Maintainer Handbook was written for server team and not a general audience.

[68] https://github.com/canonical/ubuntu-maintainers-handbook/blob/main/PackageMerging.md#build-source-package

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

2.1.12. Extract packages

This article demonstrates how to extract the contents of Debian packages.

See also the article *Package model* (page 51) for a deeper understanding of package formats.

Extract a source package

This section demonstrates how to extract the content of a source package.

Note

A source package archive has the file extension *.dsc*. See also the manual page *dsc(5)*[69] for further information.

Important

Make sure that you have the *dpkg-dev* package installed. To install it, run the following commands in a terminal:

```
sudo apt update && sudo apt install dpkg-dev
```

Run the following command in a terminal:

```
dpkg-source --extract SOURCE-PACKAGE.dsc [OUTPUT-DIRECTORY]
```

`SOURCE-PACKAGE.dsc`
: The path to the source package control file.

`OUTPUT-DIRECTORY` (optional)
: The path to the directory where to extract the content of the source package to. This directory **must not** exist. If no output directory is specified, the content is extracted into a directory named `NAME-VERSION` (where `NAME` is the name of the source package and `VERSION` its version) under the current working directory.

See the manual page *dpkg-source(1)*[70] for further information.

[69] https://manpages.ubuntu.com/manpages/noble/en/man5/dsc.5.html
[70] https://manpages.ubuntu.com/manpages/noble/en/man1/dpkg-source.1.html

Extract a binary package

This section demonstrates how to extract the content a binary package.

> **Note**
>
> A binary package archive has the file extension *.deb*. See also the manual page *deb(5)*[71] for further information.

Run the following command in a terminal:

```
dpkg-deb --extract BINARY-PACKAGE.deb OUTPUT-DIRECTORY
```

BINARY-PACKAGE.deb
: The path to the binary package control file.

OUTPUT-DIRECTORY
: The path to the directory where to extract the content of the binary package to. In comparison to *Extract a source package* (page 47), this directory can already exist and even contain files.

See the manual page *dpkg-deb(1)*[72] for further information.

> **Tip**
>
> Using `--vextract` instead of `--extract` also outputs a list of the extracted files to *standard output*.
>
> To just list the files that the package contains, use the `--contents` option:
>
> ```
> dpkg-deb --contents BINARY-PACKAGE.deb
> ```

> **Tip**
>
> You can also replace `dpkg-deb` with `dpkg` for the examples demonstrated here. `dpkg` forwards the options to `dpkg-deb`. See the manual page *dpkg(1)*[73] for further information.

> **Caution**
>
> The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.
>
> The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.
>
> If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

[71] https://manpages.ubuntu.com/manpages/noble/en/man5/deb.5.html
[72] https://manpages.ubuntu.com/manpages/noble/en/man1/dpkg-deb.1.html
[73] https://manpages.ubuntu.com/manpages/noble/en/man1/dpkg.1.html

3. Explanation

Our explanatory and conceptual guides are written to provide a better understanding of how packaging works in Ubuntu. They enable you to expand your knowledge and become better at packaging and development.

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

3.1. Upstream and downstream

An *Ubuntu* installation consists of *packages* - copied and unpacked onto the target machine. The Ubuntu project packages, distributes and maintains software of thousands of *open source* projects for users, ready to install. The collection of Ubuntu packages is derived from the collection of packages maintained by the community-driven *Debian* project.

An important duty of an Ubuntu package *Maintainer* is to collaborate with the open source projects the Ubuntu packages are derived from – especially with Debian. We do this by keeping the Ubuntu copies of packages up-to-date and by sharing improvements made in Ubuntu back up to Debian.

3.1.1. Terminology

In the context of open source software development, the analogy of a stream that carries modifications, improvements, and code is used. It describes the relationship and direction of changes made between projects. This stream originates (upwards) from the original project (and related entities like *Source Code*, authors, and maintainers) and flows downwards to projects (and associated entities) that depend on it.

Ubuntu delta

Ubuntu delta (noun):
A modification to an Ubuntu package that is derived from a Debian package.

Upstream

Upstream (noun):
A software project (and associated entities) that another software project depends on either directly or indirectly.

Examples:

- Debian is the upstream of Ubuntu.

- Upstream is not interested in the patch.

Usage note:

- There can be many layers. For example, **Kubuntu** is a *flavour* of Ubuntu, therefore Ubuntu and Debian are both upstreams of Kubuntu.
- The adjective/adverb form is much more commonly used.

Upstream (adjective, adverb):
Something (usually a code modification like a *patch*) that flows in the direction or is relative to a software project closer to the original software project.

Examples:

- Debian is the upstream project of Ubuntu.
- There is a new upstream release.
- A pull request was created upstream.
- A bug was patched upstream.

upstream (verb):
Sending something (usually a patch) upstream that originated from a *Fork* or project that depended on the upstream project.

Examples:

- We upstreamed the patch.
- Can you upstream the bugfix?

Downstream

Downstream (noun):
Similar to *Upstream (noun):* (page 49) A software project(s) (and associated entities) that depend on another software project either directly or indirectly.

Example:

- Ubuntu is a downstream of Debian and there are many downstreams of Ubuntu.

Usage note:

- The *adjective/adverb form* (page 50) is much more commonly used.
- There can be many layers. For example, **Kubuntu** is a flavour of Ubuntu, therefore Kubuntu and Ubuntu are both downstreams of Debian.

Downstream (adjective, adverb):
Similar to *Upstream (adjective, adverb):* (page 50) Something (usually a code modification like a patch) that flows in the direction or is relative to a software project farther away from the original software project.

Examples:

- Ubuntu is a downstream project of Debian.
- The bug is already patched downstream.
- The bug was reported by a downstream user.

- Downstream maintainers have submitted a bugfix.
- The change may affect downstream users.

Downstream (verb):
Similar to *upstream (verb):* (page 50) Sending something (usually a patch) downstream that originated from an upstream project.

Example:

- We downstreamed the patch.

3.1.2. Why do we upstream changes?

Note

The following list does not aim for completeness. There are plenty of other good arguments for why changes should be upstreamed.

- **Decreased maintenance complexity**: Think of any Ubuntu package derived from a Debian package that carries a *delta*. Every time the Debian package gets updated, the Ubuntu package may be subject to a *merge conflict* when the changes to the Debian package get applied to the Ubuntu package. By upstreaming changes we reduce the maintenance cost to resolve merge conflicts when they occur.
- **Quality assurance and security**: Any changes that get upstreamed will also be subject to the quality assurance of the upstream project and the testing coverage that the user base of the upstream project provides. This increases the likelihood of discovering regressions/bugs/unwanted behaviour (especially security-related bugs). Also, be aware that an unpatched *security vulnerability* in any system could lead to the indirect exposure of other systems.
- **Mutual benefit**: By syncing the Debian packages into the Ubuntu package collection, Ubuntu benefits from the upstream maintenance work. In exchange, Ubuntu Maintainers upstream changes to Debian. This results in a win-win situation where both parties benefit from working together.

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

3.2. Package model

Because *Ubuntu* is based on the community-driven *Debian* project, Ubuntu uses the Debian packaging model/format.

This consists of *source packages* (page 52) and *binary packages* (page 56).

3.2.1. Source packages

A source package contains the *source* material used to build one or more binary packages.

A source package is composed of:

- a Debian Source Control (`.dsc`) file,
- one or more compressed tar files, and
- optionally additional files depending on the type and format of the source package.

The **Source Control** file contains metadata about the source package, for instance, a list of additional files, name and version, list of the binary packages it produces, dependencies, a *digital signature* and many more fields.

> **Note**
>
> The *basic overview of the debian/ directory* (page 84) article showcases the layout of an unpacked source package.

Source package formats

There are multiple formats for how the source is packaged. The format of a source package is declared in the `debian/source/format` file. This file should always exist. If this file can not be found, the *format 1.0* (page 55) is assumed for backwards compatibility, but *`lintian(1)`*[74] will warn you about it when you try to build a source package.

> **Tip**
>
> We strongly recommend to use the *3.0 (quilt)* (page 53) format for new packages.
>
> You should only pick a different format if you **really** know what you are doing.

Native source packages

In most cases, a software project is packaged by external contributors called the *maintainers* of the package. Because the packaging is often done by a 3rd-party (from the perspective of the software project), the software to be packaged is often not designed to be packaged. In these cases the source package has to do modifications to solve specific problems for its target *distribution*. The source package can, in these cases, be considered as its own software project, like a *fork*. Consequently, the *Upstream* releases and source package releases do not always align.

Native packages almost always originate from software projects designed with Debian packaging in mind and have no independent existence outside its target distribution. Consequently native packages do not differentiate between Upstream releases and source package releases. Therefore, the version identifier of a native package does not have an Debian-specific component.

For example:

[74] https://manpages.ubuntu.com/manpages/noble/en/man1/lintian.1.html

- The debhelper package[75] (provides tools for building Debian packages) is a native package from Debian. Because it is designed with packaging in mind, the packaging specific files are part of the original *source code*. The debhelper developers are also maintainers of the Debian package. The Debian debhelper package gets merged into the Ubuntu debhelper package and has therefore a `ubuntu` suffix in the version identifier.
- In contrast, the Ubuntu bash package[76] (the default *shell* on Ubuntu) is **NOT** a native package. The bash Software[77] originates from the *GNU project*. The bash releases of the GNU project project will get packaged by Debian maintainers and the Debian bash package[78] is merged into the Ubuntu bash package by Ubuntu maintainers. The Debian and Ubuntu packages both are effectively their own separate software projects maintained by other people than the developers of the software that gets packaged. This is the process how most software is packaged on Ubuntu.

Warning

Although native packages sound like the solution to use for your software project if you want to distribute your software to Ubuntu/Debian, we **strongly** recommend against using native package formats for new packages. Native packages are known to cause long-term maintenance problems.

Format: `3.0 (quilt)`

A new-generation source package format that records modifications in a *`quilt(1)`*[79] *Patch* series within the `debian/patches` folder. The patches are organised as a *stack*, and you can apply, unapply, and update them easily by traversing the stack (push/pop). These changes are automatically applied during the extraction of the source package.

A source package in this format contains at least an original tarball (`.orig.tar.ext` where `ext` can be `gz`, `bz2`, `lzma` or `xz`) and a debian tarball (`.debian.tar.ext`). It can also contain additional original tarballs (`.orig-component.tar.ext`), where `component` can only contain alphanumeric (`a-z`, `A-Z`, `0-9`) characters and hyphens (`-`). Optionally, each original tarball can be accompanied by a *detached signature* from the upstream project (`.orig.tar.ext.asc` and `.orig-component.tar.ext.asc`).

For example, take a look at the `hello` package:

```
pull-lp-source --download-only 'hello' '2.10-3'
```

Note

You need to install `ubuntu-dev-tools` to run the `pull-lp-source`:

```
sudo apt install ubuntu-dev-tools
```

[75] https://launchpad.net/ubuntu/+source/debhelper
[76] https://launchpad.net/ubuntu/+source/bash
[77] https://www.gnu.org/software/bash/
[78] https://tracker.debian.org/pkg/bash
[79] https://manpages.ubuntu.com/manpages/noble/en/man1/quilt.1.html

When you now run *ls(1)*[80]:

```
ls -1 hello_*
```

you should see the following files:

- `hello_2.10-3.dsc`: The **Debian Source Control** file of the source package.
- `hello_2.10.orig.tar.gz`: The tarball containing the original source code of the upstream project.
- `hello_2.10.orig.tar.gz.asc`: The detached upstream signature of `hello_2.10.orig.tar.gz`.
- `hello_2.10-3.debian.tar.xz`: The tarball containing the content of the Debian directory.

Format: `3.0 (native)`

A new-generation source package format extends the native package format defined in the *format 1.0* (page 55).

A source package in this format is a tarball (`.tar.ext` where `ext` can be `gz`, `bz2`, `lzma` or `xz`).

For example, let's take a look at the `debhelper` package:

```
pull-lp-source --download-only 'debhelper' '13.11.6ubuntu1'
```

When you now run *ls(1)*[81]:

```
ls -1 debhelper_*
```

you should see the following files:

- `debhelper_13.11.6ubuntu1.dsc`: The **Debian Source Control** file of the source package.
- `debhelper_13.11.6ubuntu1.tar.xz`: The tarball containing the source code of the project.

Other examples of native source packages are:

- ubuntu-dev-tools[82]
- ubuntu-release-upgrader[83]
- dh-cargo[84]
- ubiquity[85]
- subiquity[86]

[80] https://manpages.ubuntu.com/manpages/noble/en/man1/ls.1.html
[81] https://manpages.ubuntu.com/manpages/noble/en/man1/ls.1.html
[82] https://launchpad.net/ubuntu/+source/ubuntu-dev-tools
[83] https://launchpad.net/ubuntu/+source/ubuntu-release-upgrader
[84] https://launchpad.net/ubuntu/+source/dh-cargo
[85] https://launchpad.net/ubuntu/+source/ubiquity
[86] https://launchpad.net/ubuntu/+source/subiquity

Format: `1.0`

The original source package format. Nowadays, this format is rarely used.

A native source package in this format consists of a single `.tar.gz` file containing the source.

A non-native source package in this format consists of a `.orig.tar.gz` file (containing the Upstream source) associated with a `.diff.gz` file (the patch containing Debian packaging modifications). Optionally, the original tarball can be accompanied by a detached Upstream signature `.orig.tar.gz.asc`.

> **Note**
>
> This format does not specify a patch system, which makes it harder for *maintainers* to track modifications. There were multiple approaches to how packages tracked patches. Therefore, the source packages of this format often contained a `debian/README.source` file explaining how to use the patch system.

`3.0` formats improvements

Some of the improvements that apply to most `3.0` formats are:

- Support for additional compression formats: `bzip2`, `lzma` and `xz`.
- Support for multiple Upstream tarballs.
- Supports inclusion of binary files.
- Debian-specific changes are no longer stored in a single `.diff.gz`.
- The Upstream tarball does not need to be repacked to strip the Debian directory.

Other formats

The following formats are rarely used, experimental and/or historical. You should only choose these if you know what you are doing.

- `3.0 (custom)`: Doesn't represent an actual source package format but can be used to create source packages with arbitrary files.
- `3.0 (git)`: An experimental format to package from a *git* repository.
- `3.0 (bzr)`: An experimental format to package from a *Bazaar* repository.
- `2.0`: The first specification of a new-generation source package format. It was never widely adopted and eventually replaced by *3.0 (quilt)* (page 53).

`.changes` file

Although technically not part of a source package – every time a source package is built, a `.changes` file will be created alongside it. The `.changes` file contains metadata from the Source Control file and other information (e.g. the latest changelog entry) about the source package. *Archive* tools and *Archive Administrators* use this data to process changes to source packages and determine the appropriate action to upload the source package to the *Ubuntu Archive*.

3.2.2. Binary packages

A **binary package** is a standardised format that the *Package Manager* (*dpkg(1)*[87] or *apt(8)*[88]) can understand to install and uninstall software on a target machine. This simplifies distributing software to a target machine and managing the software on that machine.

A Debian binary package uses the `.deb` file extension and contains a set of files that will be installed on the host system and a set of files that control how the files will be installed or uninstalled.

3.2.3. Resources

- Debian policy manual v4.6.2.0 – Chapter 3. Binary packages[89]
- Debian policy manual v4.6.2.0 – Chapter 4. Source packages[90]
- The manual page *dpkg-source(1)*[91]
- Debian wiki – 3.0 source package format[92]

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

3.3. Patches

Patches record modifications to *source code*. Patches can come in many forms, including but not limited to:

- *Upstream* features or bug-fixes not present in the current release.
- *Ubuntu* specific changes, such as custom defaults and theming.
- *CVE* fixes and other security-related updates.

All changes to a *source package* in the `3.0 (quilt)` format (see *explanation* (page 53)) to the source files (all files inside the *orig tarball* / outside the `debian/` directory) must be applied in the form of a patch.

Changes to a source package in the `3.0 (native)` format (see *explanation* (page 54)) get applied directly and will not be further discussed in this article.

It is important to treat patches with care, and ensure the format and headers follow best practices. This makes it easier to maintain a package long-term. In Ubuntu, we try to follow the *DEP 3* specification, which details a standard format for patch headers.

[87] https://manpages.ubuntu.com/manpages/noble/en/man1/dpkg.1.html
[88] https://manpages.ubuntu.com/manpages/noble/en/man8/apt.8.html
[89] https://www.debian.org/doc/debian-policy/ch-binary.html
[90] https://www.debian.org/doc/debian-policy/ch-source.html
[91] https://manpages.ubuntu.com/manpages/noble/en/man1/dpkg-source.1.html
[92] https://wiki.debian.org/Projects/DebSrc3.0

The source package stores the patches in the `debian/patches/` directory. These patches get applied from top to bottom in the order they are listed in `debian/patches/series`, excluding empty lines and lines starting with #.

3.3.1. Sending patches upstream

Changes to the upstream source code which are not Ubuntu specific should be sent to the upstream authors in whatever form they prefer. This allows the upstream authors to include the patch in the upstream version of the package.

3.3.2. When you should (not) rewrite a patch header to follow DEP 3

You **should** rewrite a patch header to follow DEP 3 if:

- You are introducing a new patch altogether.
- You are making substantive modifications to an existing patch.
- More information is known about the patch, and a DEP 3 header would contain updated information.

You **should not** rewrite a patch header to follow DEP 3 if:

- You are preparing a non-*micro-release exception SRU* and changing the patch header is not directly related to the bug being fixed.
- You intend on keeping only the modifications to the header as part of the *Ubuntu delta* without making substantive changes to the diff contents, and have no plans to forward it to *Debian*.
- The team claiming responsibility for this package in Ubuntu explicitly disagrees with the usage of DEP 3 headers. (This should be brought up on the ubuntu-devel mailing list.)

3.3.3. Resources

- *Patch management with quilt (how-to)* (page 32)
- *DEP 3 – Patch file headers (reference)* (page 99)
- Debian Policy Section 4.3. – Changes to the upstream sources[93]
- Debian Policy Section 4.13. – Embedded code copies[94]
- Debian Policy Section 4.17. – Vendor-specific patch series[95]
- Debian Policy Appendix 7. – Diversions - overriding a package's version of a file (from old Packaging Manual)[96]

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

[93] https://www.debian.org/doc/debian-policy/ch-source.html#changes-to-the-upstream-sources
[94] https://www.debian.org/doc/debian-policy/ch-source.html#embedded-code-copies
[95] https://www.debian.org/doc/debian-policy/ch-source.html#vendor-specific-patch-series
[96] https://www.debian.org/doc/debian-policy/ap-pkg-diversions.html

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

3.4. Ubuntu development process

Each release cycle follows the same general pattern, with the following major phases. Ubuntu contributors are expected to follow this process closely to ensure that their work is aligned with that of others. Because of the time-based release cycle, Ubuntu contributors must coordinate well to produce an on-time release.

See also the article *Ubuntu releases* (page 64) for more details about the release cadence.

3.4.1. Beginning a new release

The Ubuntu infrastructure is prepared for a new development branch at the beginning of each cycle. The package build system is set up, the toolchain is organised, *seeds* are branched, and many other pieces are made ready before development can properly begin. Once these preparations are made, the new branch is officially announced on the ubuntu-devel-announce mailing list[97] and opened for uploads to the *Ubuntu package archive* (page 68).

Note

See the Ubuntu 24.04 LTS (Noble Numbat) archive opening announcement email[98] as an example.

3.4.2. Planning

Ubuntu contributors discuss the targeted features for each release cycle via the various channels (e.g., IRC, Matrix, Discourse, Launchpad). Some of these come from strategic priorities for the distribution as a whole, and some are proposed by individual developers.

The broader open-source community gets together at the *Ubuntu Summit* (similar but different to the past *Ubuntu Developer Summits*) to share experiences and ideas and to inspire future projects covering development as well as design, writing, and community leadership with a wide range of technical skill levels.

3.4.3. Merging with upstream and feature development

The first phase of the release cycle is characterised by bringing new releases of *upstream* components into Ubuntu, either directly or via *Merges and syncs from Debian* (page 79). The development of planned projects for the release often begins while merging is still underway, and the development accelerates once the package archive is reasonably consistent and usable.

The automatic import of new package versions from Debian ends at the *Debian Import Freeze* (page 59).

3.4.4. Stabilisation and milestones (freezes)

Developers should increasingly exercise caution in making changes to Ubuntu to ensure a stable state is reached in time for the final release date. Archive admins incrementally restrict modifications to the Ubuntu package archive, effectively freezing the state of the Ubuntu package archive. The milestones when these restrictions get enabled are called “freezes”. During freezes, developers must request exceptions to approve changes. See *how to request a freeze exception* (page 46). The release team usually posts the current Release Schedule as

[97] https://lists.ubuntu.com/mailman/listinfo/ubuntu-devel-announce
[98] https://lists.ubuntu.com/archives/ubuntu-devel-announce/2023-October/001341.html

a Discourse article under the "Release" topic[99]. It shows the typical order and length of the various freezes.

> **Note**
>
> In the past, the Release Schedule was published in the Ubuntu Wiki. See, for example, the release schedule of Ubuntu 20.04 LTS (Focal Fossa)[100].

Testing weeks

During a release's development phase, the release team organise testing weeks to focus the Ubuntu community's efforts on testing Ubuntu's latest daily *ISO images* and its *flavours*. These weeks are crucial for discovering bugs and getting early feedback about new features.

> **Note**
>
> The testing weeks replaced the older practice of alpha and beta milestones. For example, Ubuntu 14.04 LTS (Trusty Tahr) had Alpha 1, Alpha 2, Beta 1, and Beta 2 milestones.
>
> See the email[101] that announced the process change.

Debian Import Freeze

Archive admins disable the automatic import of new packages and versions of existing packages from Debian. The import of a new package or version of an existing package from Debian has to be requested.

> **Note**
>
> The general development activity is still unrestricted until the Feature Freeze; however, the Feature Freeze is often scheduled for the same day.

Feature Freeze (FF)

At this point, Ubuntu developers should stop introducing new features, packages, and *API*/*ABI* changes, and instead concentrate on fixing bugs in the current release in development.

User Interface Freeze (UIF)

The user interface should be finalised to allow documentation writers and translators to work on a consistent target that doesn't render screenshots or documentation obsolete.

After the user interface freeze, the following things are not allowed to change without a freeze exception:

- User interface of individual applications that are installed by default
- Appearance of the desktop

[99] https://discourse.ubuntu.com/c/project/release
[100] https://wiki.ubuntu.com/FocalFossa/ReleaseSchedule
[101] https://lists.ubuntu.com/archives/ubuntu-release/2018-April/004434.html

- Distribution-specific artwork
- All user-visible strings in the desktop and applications that are installed by default

Documentation String Freeze

Documentation strings should no longer be created or modified. This freeze ensures that the documentation can be accurately translated.

Exceptions to this rule may be considered before the release for significant and glaring errors or exceptional circumstances.

Kernel Feature Freeze

The *kernel* feature development should end at this point, and the kernels can be considered feature-complete for the release. From now on, only bugfix changes are expected.

Note

The Kernel Feature Freeze occurs after the *Feature Freeze (FF)* (page 59) because the Linux Kernel is typically released upstream after the Feature Freeze. Additionally, the Kernel Feature Freeze is deliberately scheduled so that the Beta images have a fully featured kernel suitable for testing.

Hardware Enablement Freeze

All new hardware enablement tasks for devices targeting the given release should be finished, and all the respective packages should be in the Ubuntu package archive. The release team no longer accepts changes in the Ubuntu package archive related to supporting new image types or platforms. This freeze ensures that any new platforms are already available for testing of the beta images and in the weeks leading to the *Final Freeze* (page 61).

Note

The Hardware Enablement Freeze is usually scheduled for the same day as the Beta Freeze.

Beta Freeze

In preparation for the beta release, all uploads are queued and subject to manual approval by the release team. Changes to packages that affect beta release images (flavours included) require the release team's approval before uploading. Uploads for packages that do not affect images are generally accepted as time permits.

Tip

You can use the *`seeded-in-ubuntu(1)`*[102] tool, provided by the `ubuntu-dev-tools` package, to list all the current daily images containing a specified package or to determine whether the specified package is part of the supported seed.

If the list output is empty, uploading it during a freeze should be safe.

The freeze allows Archive Admins to fix package inconsistencies or critical bugs quickly and in an isolated manner. Once the beta release is shipped, the Beta Freeze restrictions no longer apply.

Kernel Freeze

The Kernel Freeze is the final date for kernel updates because they require several lockstep actions that must be folded into the image-building process.

Exceptional circumstances may justify exemptions to the freeze at the discretion of the release managers.

Non-language-pack translation deadline

Some translation data cannot currently be updated via the language pack mechanism. Because these items require more disruptive integration work, they are subject to an earlier deadline to give time to developers to manually export translations from Launchpad and integrate them into the package.

This marks the date after which translations for such packages are not guaranteed to be included in the final release. Depending on the package and its maintainers workflow, they may be exported later.

Other packages can still be translated until the *Language pack translation deadline* (page 62).

Final Freeze

This freeze marks an **extremely** high-caution period until the *Final Release* (page 62). Only bug fixes for release-critical, security-critical or otherwise exceptional circumstantial bugs are included in the Final Release, which the release team and relevant section teams must confirm.

Unseeded packages

Packages in *universe* (page 71) that aren't seeded in any of the Ubuntu flavours remain in *Feature Freeze (FF)* (page 59) because they do not affect the release; however, when the Ubuntu package archive is frozen, fixes must be manually reviewed and accepted by the release team members.

When the Final Release is close (~1.5 days out), developers should consider uploading to the *proposed pocket* (page 69), from which the release team cherry-picks into the *release pocket* (page 69) if circumstances allow. All packages uploaded to the proposed pocket that do not make it into the release pocket until the Final Release become candidates for *Stable Release Updates* (page 62). Therefore, uploads to the proposed pocket during Final Freeze should meet the requirements of Stable Release Updates if the upload is not accepted into the release pocket. In particular, the upload must reference at least one bug, which is used to track the stable update.

Note

[102] https://manpages.ubuntu.com/manpages/noble/en/man1/seeded-in-ubuntu.1.html

If you are sure that your upload will be accepted during Final Freeze, you can upload directly to the release pocket, but be aware that you have to re-upload after Final Release if the upload gets rejected.

Release Candidate

The images produced during the week before the *Final Release* (page 62) are considered "release candidates". In an ideal world, the first release candidate would end up being the Final Release; however, we don't live in a perfect world, and this week is used to get rid of the last release-critical bugs and do as much testing as possible. Until the Final Release, changes are only permitted at the release team's discretion and will only be allowed for high-priority bugs that might justify delaying the release.

Language pack translation deadline

Translations done up until this date will be included in the final release's language packs.

3.4.5. Finalisation

As the final release approaches, the focus narrows to fixing "showstopper" bugs and thoroughly validating the installation images. Every image is tested to ensure that the installation methods work as advertised. Low-impact bugs and other issues are deprioritised to focus developers on this effort.

This phase is vital, as severe bugs that affect the experience of booting or installing the images must be fixed before the final release. In contrast, ordinary bugs affecting the installed system can be fixed with Stable Release Updates.

3.4.6. Final Release

Once the release team declares the *Release Candidate* (page 62) ISO stable and names it the "Final Release", a representative of the team announces it on the ubuntu-announce mailing list[103].

Note

See, for example, the Ubuntu 24.04 LTS (Noble Numbat) release announcement[104].

3.4.7. Stable Release Updates

After publication of an *Ubuntu Stable Release*, there may be a need to update it or fix bugs. You can fix these newly-discovered bugs and make updates through a special process known as **Stable Release Update (SRU)**.

The SRU process ensures that any changes made to a stable release are thoroughly vetted and tested before being made available to users. This is because many users rely on the stability of the stable release for their day-to-day operations, and any problem they experience with it can be disruptive.

The following paragraphs intend to give you a brief introduction to the SRU process. See the dedicated Ubuntu SRU Documentation[105] for more details about this process.

[103] https://lists.ubuntu.com/archives/ubuntu-announce/
[104] https://lists.ubuntu.com/archives/ubuntu-announce/2024-April/000301.html
[105] https://documentation.ubuntu.com/sru/en/latest/#home

When are SRUs necessary?

SRUs require great caution because they're automatically recommended to a large number of users. So, when you propose an update, there should be a strong rationale for it. Also, the update should present a low risk of *regressions* (page 64).

You can propose an SRU in the following cases:

- To fix high-impact bugs, including those that may directly cause security vulnerabilities, severe regressions from the previous release, or bugs that may directly cause loss of user data.
- To adjust to changes in the environment, server protocols, or web services. This ensures that Ubuntu remains compatible with evolving technologies.
- For safe cases with low regression potential but high user experience improvement.
- To introduce new features in *LTS releases*, usually under strict conditions.
- To update commercial software in the *Canonical partner archive*.
- To fix *Failed to build from Source* issues.
- To fix *autopkgtest* failures, usually in conjunction with other high-priority fixes.

See also: SRU requirements[106]

Overview

A typical SRU is performed like this:

1. Ensure the bug is fixed in the *current development release* and all subsequent supported releases to ensure consistency across different Ubuntu versions, especially preventing regressions when users upgrade to newer releases.
2. Update the **existing** bug report detailing the Impact of the Bug, the Test Plan to verify that the bug was fixed and highlight where problems could occur.
3. Get the package with the SRU patch into the upload queue.
4. The SRU team then reviews from the unapproved queue. When the upload is ready, the SRU team accepts the upload into the proposed pocket.
5. Once the builds are ready, autopkgtest are triggered. Test the binaries in the *Ubuntu Archive* and follow up in the bug report with your verification results.
6. The Ubuntu SRU Team evaluates the testing feedback and moves the package into *updates* (page 69) after it passes a minimum ageing period of 7 days without regressions.

See how to perform an SRU[107].

Verification

Once the SRU team accepts the SRU into the proposed pocket, the SRU has to be verified by the reporter or affected users of the SRU bug in a software environment that closely resembles the state after the SRU team copies the package to the updates pocket. Generally, this is with a system that's up to date with the release, security, and updates pockets. It shouldn't

[106] https://documentation.ubuntu.com/sru/en/latest/explanation/requirements/
[107] https://documentation.ubuntu.com/sru/en/latest/howto/standard/#howto-perform-standard-sru

include other packages from the proposed or backports pocket, except commonly-installed packages built from the affected source package.

Read more about this process[108].

SRU phasing

Once a package is released to the updates pocket, the update is then phased, so it is gradually made available to expanding subsets of Ubuntu users.

Read more about phasing[109].

Regressions

Regressions are unintended negative consequences that updates introduce. They appear as new bugs or failures in previously well-functioning aspects of an Ubuntu release.

Read more about regressions[110] and how to handle regressions[111].

Updates removal

If a bug fixed by an update doesn't get any testing or verification feedback for 90 days, an automated "call for testing" comment is made on the bug report. If no testing occurs within an additional 15 days, totalling 105 days without any testing, the *Stable Release Managers* removes the package from proposed and close the bug task as `Won't Fix`.

Also, updates are removed from proposed if they introduce a non-trivial regression.

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

3.5. Ubuntu releases

3.5.1. Release cadence

Ubuntu follows a strict time-based release cycle. Every six months since 2004, *Canonical* publishes a new Ubuntu version and its set of *packages* are declared stable (production-quality). Simultaneously, a new version begins development; it is given its own *Code name*, but also referred to as the "*Current Release in Development*" or "*Devel*".

[108] https://documentation.ubuntu.com/sru/en/latest/howto/release/
[109] https://documentation.ubuntu.com/sru/en/latest/explanation/standard-processes/#explanation-phasing
[110] https://documentation.ubuntu.com/sru/en/latest/explanation/standard-processes/#explanation-regressions
[111] https://documentation.ubuntu.com/sru/en/latest/howto/regression/#howto-report-regression

LTS releases

Since 2006, every fourth release, made every two years in April, receives *Long Term Support (LTS)* (page 66) for large-scale deployments. This is the origin of the term "LTS" for stable, maintained releases.

An estimated 95% of all Ubuntu installations are LTS releases.

> **Note**
>
> Because of the strict time-based six months release cycle, you will only see LTS releases in even-numbered years (e.g. `18`, `20`, `22`) in April (`04`). The only exception to this rule was Ubuntu 6.06 LTS (Dapper Drake).

Point releases

To ensure that a fresh install of an *LTS release* (page 65) will work on newer hardware and not require a big download of additional updates, Canonical publishes **point releases** that include all the updates made so far.

The first point release of an LTS is published three months after the initial release and repeated every six months at least until the next LTS is published. In practice, Canonical may publish even more point releases for an LTS series, depending on the popularity of that LTS series.

For example, the Ubuntu 16.04.7 LTS (Xenial Xerus) point release was published more than four years after the initial release of Ubuntu 16.04 LTS.

Interim releases

In the years between LTS releases, Canonical also produces **interim releases**, sometimes also called "regular releases".

Many developers use interim releases because they provide newer compilers or access to newer *Kernels* and newer libraries, and they are often used inside rapid DevOps processes like *CI*/*CD* pipelines where the lifespan of an artefact is likely to be shorter than the support period of the interim release.

Why does Ubuntu use time-based releases?

Ubuntu releases represent an aggregation of the work of thousands of independent software projects. The time-based release process provides users with the best balance of the latest software, tight integration, and excellent overall quality.

3.5.2. Ubuntu version format

```
YY.MM[.POINT-RELEASE] [LTS]
```

Ubuntu version identifier as used for Ubuntu releases consist of four components, which are:

`YY`
: The 2-digit year number of the initial release.

`MM`
: The 2-digit month number of the initial release.

> **Note**
>
> Because of the strict time-based six months release cycle, you will usually only see releases in April (`04`) and October (`10`).

`POINT-RELEASE`
: The *point release* (page 65) number starts at `1` and increments with every additional point release.

  This component is omitted for the initial release, in which case zero is assumed.

`LTS`
: Any Ubuntu release that receives long term support will be marked with `LTS` (see the *release lifespan* (page 66) section for more information).

  Any Ubuntu release that does not receive long term support omits this component.

Examples

Version Identifier	Release Date	Support	End of Standard Support	End of Life
`22.04 LTS`	21 April 2022	Long term	April 2027	April 2032
`22.04.1 LTS`	11 August 2022	Long term	April 2027	April 2032
`22.10`	22 October 2022	Regular	July 2023	July 2023
`22.04.2 LTS`	13 February 2023	Long term	April 2027	April 2032
`23.04`	20 April 2022	Regular	January 2024	January 2024

3.5.3. Release lifespan

Every Ubuntu *Series* receives the same production-grade support quality, but the length of time for which an Ubuntu series receives support varies.

Regular support

Interim releases (page 65) are production-quality releases and are supported for nine months, with sufficient time provided for users to update, but these releases do not receive the long-term commitment of LTS releases.

Long Term Support (LTS)

LTS releases receive five years of standard security maintenance for all packages in the *Main Component*. With an *Ubuntu Pro* subscription, you get access to *Expanded Security Maintenance* (*ESM*), covering security fixes for packages in the *Universe Component*. ESM also extends the lifetime of an LTS series from five years to ten years.

3.5.4. Editions

Every Ubuntu release is provided as both a *Server* and *Desktop* edition.

Ubuntu Desktop provides a graphical *User Interface* (*GUI*) for everyday computing tasks, making it suitable for personal computers and laptops. *Ubuntu Server*, on the other hand, provides a text-based *User Interface* (*TUI*) instead of a *GUI*, optimised for server environments, that allows machines on the server to be run headless, focusing on server-related services and applications, making it ideal for hosting web services, databases, and other server functions.

Additionally, each release of Ubuntu is available in minimal configurations, which have the fewest possible packages installed: available in the installer for Ubuntu Server, Ubuntu Desktop, and as separate cloud images.

Canonical publishes Ubuntu on all major public clouds, and the latest *image* for each LTS version will always include any security update provided since the LTS release date, until two weeks prior to the image creation date.

3.5.5. Ubuntu flavours

Ubuntu flavours are *Distributions* of the default Ubuntu releases, which choose their own default applications and settings. Ubuntu flavours are owned and developed by members of the Ubuntu community and backed by the full *Ubuntu Archive* for packages and updates.

Officially recognised flavours are:

- Edubuntu[112]
- Kubuntu[113]
- Lubuntu[114]
- Ubuntu Budgie[115]
- Ubuntu Cinnamon[116]
- Ubuntu Kylin[117]
- Ubuntu MATE[118]
- Ubuntu Studio[119]
- Ubuntu Unity[120]
- Xubuntu[121]

In addition to the officially recognised flavours, dozens of other *Linux* distributions take Ubuntu as a base for their own distinctive ideas and approaches.

[112] https://edubuntu.org/
[113] https://kubuntu.org/
[114] https://lubuntu.me/
[115] https://ubuntubudgie.org/
[116] https://ubuntucinnamon.org/
[117] https://www.ubuntukylin.com/index-en.html
[118] https://ubuntu-mate.org/
[119] https://ubuntustudio.org/
[120] https://ubuntuunity.org/
[121] https://xubuntu.org/

3.5.6. Resources

- The Ubuntu life cycle and release cadence[122]
- Ubuntu wiki – List of releases[123]
- Ubuntu flavours[124]
- Ubuntu wiki – Ubuntu flavours[125]
- Ubuntu wiki – time-based releases[126]
- Ubuntu wiki – point release process[127]
- Ubuntu wiki – end of life process[128]
- Ubuntu releases[129]
- *Ask a bug supervisor <https://answers.launchpad.net/launchpad/+question/140509>*
- *contact the Ubuntu SRU Team <https://wiki.ubuntu.com/StableReleaseUpdates#Contacting_the_SRU_te*

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

3.6. Ubuntu package archive

Linux distributions like *Ubuntu* use *repositories* (page 69) to hold *packages* you can install on target machines. Ubuntu has several repositories that anyone can access. The **Ubuntu package archive** hosts *Debian binary packages* (`.deb` files) and *source packages* (`.dsc` files). On Ubuntu installations, the Ubuntu package archive is configured as the default source for the *APT* package manager to download and install packages from. This Archive splits into many layers, each with its own terminology. The different terminology is quite confusing at first, but if we take a look, layer-by-layer, we'll see not just what all the terms mean, but how they all fit together.

Let's have a quick overview with this diagram. The general flow is that the Archive splits into *Ubuntu series* (page 69). Each series is split up into *pockets* (page 69), and then each pocket contains four *components* (page 70). If we tried to show all of this on one diagram, it would be quite extensive, so let's take a look through a single path:

[122] https://ubuntu.com/about/release-cycle
[123] https://wiki.ubuntu.com/Releases
[124] https://ubuntu.com/desktop/flavours
[125] https://wiki.ubuntu.com/UbuntuFlavors
[126] https://wiki.ubuntu.com/TimeBasedReleases
[127] https://wiki.ubuntu.com/PointReleaseProcess
[128] https://wiki.ubuntu.com/EndOfLifeProcess
[129] https://releases.ubuntu.com/

> **Note**
>
> Some of the following terminologies have only loose or informal definitions. Also, be aware that the terminology surrounding the Ubuntu package archive gets mixed up in day-to-day communications. This can be confusing, but the meaning is usually evident from the surrounding context once you are familiar with the following terminologies.

3.6.1. Repositories

In the context of package management, **repositories** are servers containing sets of packages that a *package manager* can download and install.

This term can refer to the Ubuntu package archive as a whole or just *suites* (page 70), *pockets* (page 69), or *components* (page 70).

3.6.2. Series

A **series** refers to the packages that target a specific Ubuntu version. A series is usually referred to by its *code name*.

Examples of series are: `mantic`, `lunar`, `jammy`, `focal`, `bionic`, `xenial`, `trusty`.

> **Note**
>
> In practice, the terms "Ubuntu series" and "Ubuntu release" are often used synonymously or are mistaken for each other. There is technically a difference; for example, an LTS version usually has an initial release (e.g. 22.04 LTS) and multiple point releases (e.g. 22.04.1 LTS, 22.04.2 LTS), which are all part of the same *series* (e.g. `jammy`).

3.6.3. Pockets

Pockets are package sub-repositories within the Ubuntu package archive. Every Ubuntu series has the following pockets:

release

This pocket contains the packages that an Ubuntu series was initially released with. After the initial release of an Ubuntu series, the packages in this pocket are not updated (not even for security-related fixes).

security

This pocket contains security-related updates to packages in the *release* (page 69) pocket.

updates

This pocket contains non-security-related updates to packages in the *release* (page 69) pocket.

proposed

This pocket is a *staging environment* the Ubuntu community can opt into, to verify the stability of any updates before they get deployed to a broader range of consumers.

- Before the initial release of an Ubuntu series, this pocket contains non-security-related updates to packages in the *release* (page 69) pocket before they get uploaded to the

release (page 69) pocket.

- After the initial release of an Ubuntu series, this pocket contains non-security-related updates to packages in the *release* (page 69) pocket before they get uploaded to the *updates* (page 69) pocket.

backports

This pocket contains packages the Ubuntu series was initially **NOT** released with.

The *backports article* (page 81) provides more information on backporting software.

Important

The **backports pocket** does not come with any security support guarantee. The Ubuntu Security Team does not update packages in the backports pocket. The Ubuntu community is responsible for maintaining packages in backports with later patches for bug fixes and security updates.

3.6.4. Suite

A combination of a series and a pocket. For example:

Suite	Series	Pocket
`jammy`	`jammy`	*release* (page 69)
`jammy-security`	`jammy`	*security* (page 69)
`jammy-updates`	`jammy`	*updates* (page 69)
`jammy-proposed`	`jammy`	*proposed* (page 69)
`jammy-backports`	`jammy`	*backports* (page 70)

You can see all active suites[130] in the archive.

Note

The `devel` series always mirrors the series with the code name of the *current release in development*.

3.6.5. Components

Components are logical subdivisions or *namespaces* of the packages in a suite. The APT package manager can subscribe to the individual components of a suite.

The packages of an Ubuntu series are categorised according to whether they are *Open Source Software* or *Closed Source Software*, and whether or not they are part of the *base packages* for a given series. On this basis they are sorted into the components “main”, “restricted”, “universe”, or “multiverse”, as shown in the following table:

	Open source software	Closed source software
Ubuntu base packages	*main* (page 71)	*restricted* (page 71)
Community packages	*universe* (page 71)	*multiverse* (page 71)

[130] http://archive.ubuntu.com/ubuntu/dists/

Canonical maintains the base packages and provides security updates. See *release lifespan* (page 66) for more information about the official support provided by Canonical.

For example, if you look into any of the *Pockets* (page 69) of the `devel` series (`devel-release`[131], `devel-updates`[132], `devel-security`[133], `devel-proposed`[134], `devel-backports`[135]) you will see the four components (main, restricted, universe, multiverse) as directories.

main

This component contains open source software packages for a given series that are supported and maintained by Canonical.

restricted

This component contains closed source software packages for a given series that are supported and maintained by Canonical. Packages in this component are mostly proprietary drivers for devices and similar.

universe

This component contains open source software packages for a given series that are supported and maintained by the Ubuntu community.

multiverse

This component contains packages (for a given series) of closed source software, or open source software restricted by copyright or legal issues. These packages are maintained and supported by the Ubuntu community, but because of the restrictions, patching bugs or updates may not be possible.

3.6.6. Mirrors

Every day, hundreds of thousands of people want to download and install packages from the Ubuntu package archive. To provide a good *user experience*, the content of `http://archive.ubuntu.com/ubuntu` gets mirrored (replicated and kept in sync) by other servers to distribute network traffic, reduce latency, and provide redundancy, which ensures high availability and fault tolerance.

Here is a complete list of officially recognised Ubuntu package archive mirrors[136].

Note

There are also mirrors for the Ubuntu *ISO* images (also called “CD images”, because ISO images can be downloaded and burned to a CD to make installation disks.)

You can find a complete list of officially recognised Ubuntu CD mirrors[137].

[131] http://archive.ubuntu.com/ubuntu/dists/devel/
[132] http://archive.ubuntu.com/ubuntu/dists/devel-updates/
[133] http://archive.ubuntu.com/ubuntu/dists/devel-security/
[134] http://archive.ubuntu.com/ubuntu/dists/devel-proposed/
[135] http://archive.ubuntu.com/ubuntu/dists/devel-backports/
[136] https://launchpad.net/ubuntu/+archivemirrors
[137] https://launchpad.net/ubuntu/+cdmirrors

Country mirrors

Ubuntu package archive mirrors that provide a very reliable service in a country can request to be the official **country mirror** for that country. Ubuntu installations are configured by default to use the country mirror for their selected country.

Country mirrors are accessible via the domain name format:

```
<country-code>.archive.ubuntu.com
```

You can see which mirror is the country mirror by doing a simple *DNS* lookup. For example:

Finland (FI)

Tunisia (TN)

```
dig fi.archive.ubuntu.com +noall +answer
```

```
fi.archive.ubuntu.com.      332     IN      CNAME   mirrors.nic.funet.fi.
mirrors.nic.funet.fi.       332     IN      A       193.166.3.5
```

Therefore, `mirrors.nic.funet.fi` is Finland's country mirror.

Tunisia does not have any third-party mirrors in its country. Therefore the Tunisia country mirror is just the primary Ubuntu package archive server (`archive.ubuntu.com`).

```
dig tn.archive.ubuntu.com +noall +answer
```

```
tn.archive.ubuntu.com.      60      IN      A       185.125.190.36
tn.archive.ubuntu.com.      60      IN      A       91.189.91.83
tn.archive.ubuntu.com.      60      IN      A       91.189.91.82
tn.archive.ubuntu.com.      60      IN      A       185.125.190.39
tn.archive.ubuntu.com.      60      IN      A       91.189.91.81
```

which are just the `archive.ubuntu.com` IP addresses:

```
dig archive.ubuntu.com +noall +answer
```

```
archive.ubuntu.com. 1       IN      A       185.125.190.39
archive.ubuntu.com. 1       IN      A       185.125.190.36
archive.ubuntu.com. 1       IN      A       91.189.91.83
archive.ubuntu.com. 1       IN      A       91.189.91.81
archive.ubuntu.com. 1       IN      A       91.189.91.82
```

3.6.7. Package uploads

Ubuntu encourages contributions from any person in the wider community. However, direct uploading to the Ubuntu package archive is restricted. These general contributions need to be reviewed and uploaded by a *sponsor*.

See our *article on sponsorship* (page 78) that explains this process in more detail.

3.6.8. Security update propagation

This section is a niche technical explanation. You can skip it if you don't feel that this is currently relevant for you.

Because security updates contain fixes for *Common Vulnerabilities and Exposures* (CVE), it is mission critical to distribute them as fast as possible to end users. Mirrors are a technical burden in this case, because there is a delay between the synchronisation of a mirror and the primary Ubuntu package archive server.

In the worst case a bad actor gets informed about a CVE and can use it, before the update reaches a target machine.

Therefore the APT package manager is configured by default (on Ubuntu) to also check for updates from `security.ubuntu.com`. Security updates will get uploaded here first. If a mirror does not provide the update yet a client will download it from `security.ubuntu.com` instead from the mirror.

You can see this yourself if you look what the *`sources.list(5)`*[138] file contains on your Ubuntu machine:

```
cat /etc/apt/sources.list
```

At the end of the file you will find something similar to this:

```
deb http://security.ubuntu.com/ubuntu SERIES-security main restricted
# deb-src http://security.ubuntu.com/ubuntu SERIES-security main restricted
deb http://security.ubuntu.com/ubuntu SERIES-security universe
# deb-src http://security.ubuntu.com/ubuntu SERIES-security universe
deb http://security.ubuntu.com/ubuntu SERIES-security multiverse
# deb-src http://security.ubuntu.com/ubuntu SERIES-security multiverse
```

Because the *`sources.list(5)`*[139] file is read from top to bottom, the APT package manager will download updates from the mirror first and only download it from `security.ubuntu.com` if the mirror has an older version, because the mirror has not synchronised with the primary Ubuntu package archive server yet.

`security.ubuntu.com` points to the same servers as `archive.ubuntu.com` if you do a DNS lookup. It is used in the *`sources.list(5)`*[140] file for the security pocket to prevent a user/script from accidentally changing it to a mirror.

3.6.9. Resources

- Ubuntu release cycle[141]
- Ubuntu blog – Ubuntu updates, releases and repositories explained[142]
- Ubuntu Server docs – package management[143]
- Ubuntu wiki – mirrors[144]
- Ubuntu help – repositories[145]
- Ubuntu help – repositories/Ubuntu[146]

[138] https://manpages.ubuntu.com/manpages/noble/en/man5/sources.list.5.html
[139] https://manpages.ubuntu.com/manpages/noble/en/man5/sources.list.5.html
[140] https://manpages.ubuntu.com/manpages/noble/en/man5/sources.list.5.html
[141] https://ubuntu.com/about/release-cycle
[142] https://ubuntu.com/blog/ubuntu-updates-releases-and-repositories-explained
[143] https://ubuntu.com/server/docs/package-management
[144] https://wiki.ubuntu.com/Mirrors
[145] https://help.ubuntu.com/community/Repositories
[146] https://help.ubuntu.com/community/Repositories/Ubuntu

Landscape repositories

Landscape[147] is a management and administration tool for Ubuntu. Landscape allows you to mirror *APT* repositories like the Ubuntu package archive. Although it is not directly related to the Ubuntu package archive it can be educational to understand how APT repositories work in general.

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

3.7. Launchpad

Launchpad is a software collaboration and hosting platform similar to platforms like GitHub[148]. Launchpad is also the platform where the *Ubuntu* project lives. This is one of the major differences between the Ubuntu and *Debian* infrastructure.

Note

Although the Ubuntu project is probably the largest user base of Launchpad, Launchpad can be used by anyone.

Launchpad features, among others, are:

- **Bugs**: *Bug Tracking System*
- **Code**: *source code* hosting with *Git* or *Bazaar*, *version control* and *code review* features
- **Answers**: community support site and knowledge base
- **Translations**: collaboration platform for localising software
- **Blueprints**: feature planning and specification tracking
- Ubuntu *package* building and hosting
- Team/Group management

While platforms like GitHub put users and groups at the top level, Launchpad puts projects at the top level. If you take Ubuntu as an example, you can see that you can access it at the top level: *https://launchpad.net/ubuntu*. Users and groups begin with a ~, for instance *https://launchpad.net/~ubuntu-foundations-team*.

[147] https://ubuntu.com/landscape
[148] https://github.com/

3.7.1. Why not use platforms like GitHub?

Although Launchpad's *UI* and *UX* are a bit dated, Launchpad offers an unparalleled Ubuntu package building and hosting infrastructure that no other platform offers. Even simple requirements like building for architectures like *PowerPC*, *s390x*, or *RISC-V* can not be fulfilled by GitHub or similar platforms.

3.7.2. Personal Package Archive (PPA)

Launchpad PPA repositories allow you to build installable Ubuntu packages for multiple *architectures* and to host them in your own software *repository*.

Using a PPA is straightforward; you don't need the approval of anyone, therefore users have to enable it manually. See how to *Install packages from a PPA* (page 26).

This is useful when you want to test a change, or to show others that a change builds successfully or is installable. Some people have special permission to trigger the *autopkgtests* for packages in a PPA.

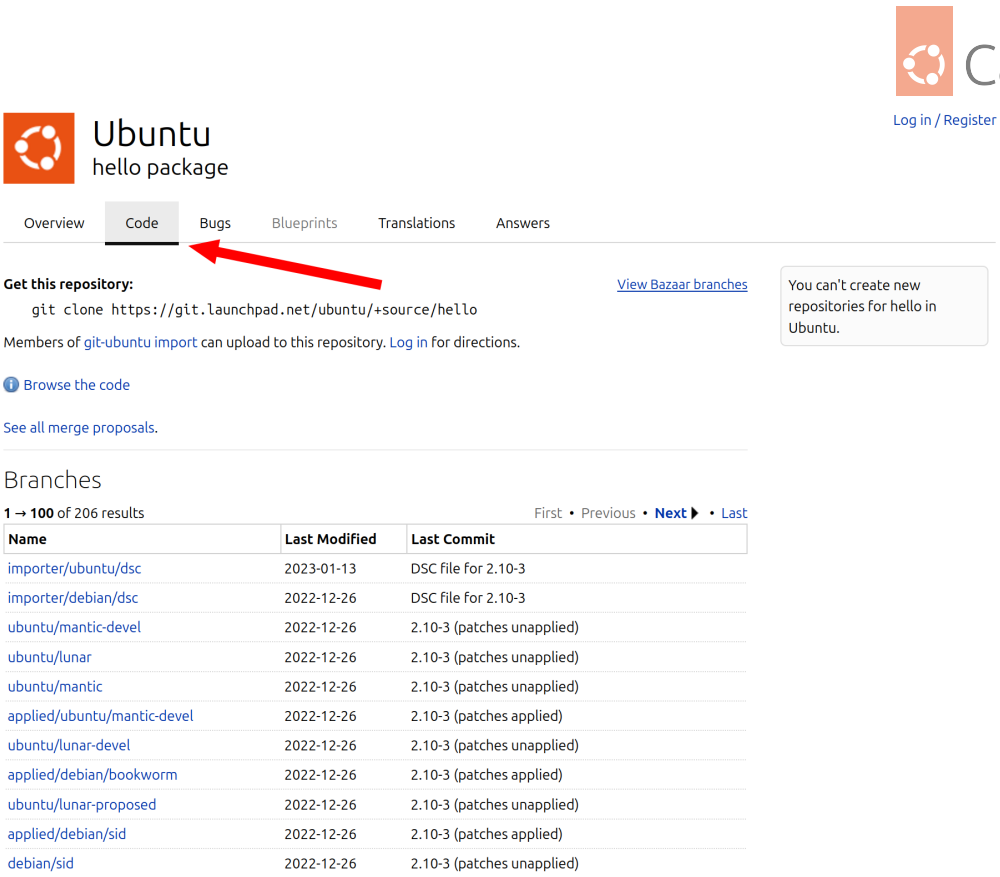
> **Tip**
>
> You can ask in the *IRC* channel `#ubuntu-devel` if someone can trigger autopkgtests in your PPA if you don't have the permission.

3.7.3. git-based workflow for the development of Ubuntu source packages

Launchpad hosts a *git-ubuntu* importer service that maintains a view of the entire packaging version history of Ubuntu *source packages* using git repositories with a common branching and tagging scheme. The git-ubuntu *CLI* provides tooling and automation that understands these repositories to make the development of Ubuntu itself easier.

You can see the web-view of these repositories when you click on the "Code" tab of any source package on Launchpad, for example, in the "hello" source package[149] as shown in the following screenshot:

[149] https://code.launchpad.net/ubuntu/+source/hello



3.7.4. Text markup

Launchpad has some markup features that you can use when you e.g. report bugs, write comments, create merge proposals.

See the *Launchpad text markup* (page 102) reference for more details.

3.7.5. Getting help

If you need help with Launchpad you can choose any of the following methods:

IRC chat rooms

On the `irc.libera.chat` *IRC* server you will find the `#launchpad` channel, where you can ask the Launchpad team and the Ubuntu community for help.

Mailing lists

If you prefer to ask for help via email, you can write to the launchpad-users[150] mailing list (`launchpad-users@lists.launchpad.net`).

Ask a question

As mentioned above, Launchpad has a community FAQ feature[151] (called “Answers”) where you can see other people’s questions or ask one yourself. Use can use the *Answers* feature of the Launchpad project on Launchpad itself.

[150] https://launchpad.net/~launchpad-users
[151] https://answers.launchpad.net/launchpad

Report a bug

If you encounter any bug related to Launchpad, you can submit a bug report to the *Bug Tracking System* of the Launchpad project on Launchpad itself[152].

3.7.6. Staging environment

Before new features are deployed to the production environment they get deployed to a staging environment[153] where the changes can get tested.

You can use the staging environment, to try out Launchpad features.

3.7.7. API

Launchpad has a web *API* that you can use to interact with its services. This makes it easy for developer communities like Ubuntu's to automate specific workflows.

You can find the reference documentation for the web API[154] on Launchpad.

The Launchpad team even created an *open source* Python library, launchpadlib[155].

3.7.8. Resources

- Launchpad home page[156]
- **The Launchpad software project on Launchpad itself**[157]
  - Launchpad bug tracker[158]
  - Launchpad questions and answers[159]
- Launchpad wiki[160]
- Launchpad development wiki[161]
- Launchpad blog[162]

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

[152] https://bugs.launchpad.net/launchpad
[153] https://qastaging.launchpad.net/
[154] https://launchpad.net/+apidoc/
[155] https://help.launchpad.net/API/launchpadlib
[156] https://launchpad.net
[157] https://launchpad.net/launchpad
[158] https://bugs.launchpad.net/launchpad
[159] https://answers.launchpad.net/launchpad
[160] https://help.launchpad.net/
[161] https://dev.launchpad.net/
[162] https://blog.launchpad.net/

3.8. Sponsorship

Sponsorship is a process that allows developers without upload rights to submit their patches or new packages for review. If approved, an authorised developer will upload the changes on their behalf.

3.8.1. When can you request sponsorship?

Since upload rights are carefully managed to ensure system stability and security, new contributors don't have them. So if you don't have upload rights, you can request sponsorship in the following situations:

- Making changes to existing packages or incremental updates.
- Submitting security updates or bug fixes.
- Introducing new packages to Ubuntu.

3.8.2. Requesting sponsorship

To request sponsorship, follow these steps:

1. File an Ubuntu bug in Launchpad[163] or follow up on an existing one.
2. Add the necessary files, such as patches or `.diff.gz` files, according to the package's requirements. If the change is a patch, follow the patch tagging guidelines. For security updates, follow the security update packaging guidelines described in Packaging[164].
3. Link your changes to the bug. See Seeking Sponsorship[165].
4. Subscribe `ubuntu-sponsors` or `ubuntu-security-sponsors` to the bug.

3.8.3. Sponsoring a patch

Members of the *Ubuntu Sponsors* and *Ubuntu Security Sponsors* teams have the right to sponsor patches or new packages. If you are interested in sponsoring, you can apply to join these teams.

3.8.4. Responding to feedback from sponsors

If a sponsor reviews your changes and requests further modifications, make the modifications to the branch you were working on, then commit them by running:

```
$ bzr commit
```

Now, push your modifications to *Launchpad*. Since `bzr` remembers the previous push location, you can run:

```
$ bzr push
```

After pushing your modifications, reply to the sponsor's request explaining the modifications you made and request a re-review. You can also respond directly on the merge proposal page in Launchpad.

3.8.5. Resources

- Sponsorship Process[166]
- Seeking Sponsorship[167]

[163] https://bugs.launchpad.net/ubuntu/+filebug
[164] https://wiki.ubuntu.com/SecurityTeam/UpdatePreparation#Packaging
[165] https://wiki.ubuntu.com/DistributedDevelopment/Documentation/SeekingSponsorship
[166] https://wiki.ubuntu.com/SponsorshipProcess
[167] https://wiki.ubuntu.com/DistributedDevelopment/Documentation/SeekingSponsorship

- Update Preparation[168]
- Seeking Review and Sponsorship[169]

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

3.9. Proposed migrations

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

3.10. Importing changes from Debian (merges & syncs)

This article explains how and why changes from *Debian* are imported into Ubuntu.

3.10.1. How does Ubuntu import changes from Debian?

Because Ubuntu is derived from Debian and uses the same package management system (*APT*), most changes made to Debian can also be applied to Ubuntu.

Syncs and **merges** are the two processes through which Ubuntu developers integrate updates and improvements from Debian into the *Ubuntu package archive* (page 68).

Sync

Beginning with the archive opening for a new Ubuntu release until the *Debian Import Freeze* (page 59), new *packages* and packages with higher version identifiers than the corresponding Ubuntu packages are automatically copied from Debian unstable (also known as *Code name* "Sid") into the Ubuntu package archive if the corresponding Ubuntu packages do not carry *Ubuntu Delta*. This process is called "synchronisation with Debian", or "sync" for short.

On request (via a *Launchpad* bug-ticket), *Archive Admins* can sync a package from Debian even if the Ubuntu package carries Ubuntu Delta. In this case, the Ubuntu Delta will be dropped. A

[168] https://wiki.ubuntu.com/SecurityTeam/UpdatePreparation#Packaging
[169] https://ubuntu-packaging-guide.readthedocs.io/en/latest/ubuntu-packaging-guide/udd-sponsorship.html

good example is when Ubuntu-specific changes have been merged into the Debian package or *Upstream* project and are no longer needed.

> **Note**
>
> The *Feature Freeze (FF)* (page 59) is often scheduled for the same day as the *Debian Import Freeze* (page 59).
>
> After the Debian Import Freeze and before the *Final Release* (page 62), you must also request the respective freeze exception.
>
> After the Final Release, you must follow the *Stable Release Updates* (page 62) process. For additional details about the freezes, see the *Ubuntu development process* (page 58) article.

Merges

When importing a newer Debian package into Ubuntu, a merge must be performed if the corresponding Ubuntu package carries Ubuntu Delta that needs to be partially or fully applied to the Debian package.

The Ubuntu Merge-o-Matic (MoM) automatically performs merges and publishes the reports on this page[170]. See the lists of outstanding merges for:

- main[171]
- universe[172]
- restricted[173]
- multiverse[174]

To complete a merge, interaction and supervision by Ubuntu maintainers are required. See the *tutorial* (page 10) and *how-to* (page 46) for details on performing a merge.

See the section *Components* (page 70) in the article that explains the Ubuntu package archive for an explanation of `main`, `universe`, `restricted` and `multiverse`.

3.10.2. Why does Ubuntu import changes from Debian?

Ubuntu incorporates changes from Debian through merging and syncing to leverage the extensive work and improvements made by the Debian community. Debian provides a stable foundation and a vast repository of packages. By integrating changes from Debian, Ubuntu can focus on refining the *user experience*. At the same time, the consistency between Ubuntu and Debian allows for sharing resources (e.g., testing and bug fixing) and contributing back to the open-source ecosystem, ultimately benefiting both *distributions* and their users.

> **Caution**
>
> The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

[170] https://merges.ubuntu.com/
[171] https://merges.ubuntu.com/main.html
[172] https://merges.ubuntu.com/universe.html
[173] https://merges.ubuntu.com/restricted.html
[174] https://merges.ubuntu.com/multiverse.html

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

3.11. Transitions

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

3.12. Backports

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

3.13. Main Inclusion Review (MIR)

Important

Do not confuse the abbreviation *MIR* with the display server[175] Mir.

Packages in *Main* and *Restricted* are officially maintained, supported and recommended by the *Ubuntu* project. *Canonical's* support services applies to these packages, which include security updates and certain *SLA* guarantees when bugs are reported and technical support is requested.

Therefore, special consideration is necessary before adding new packages to Main or Restricted. The Ubuntu *MIR Team* reviews packages for promotion:

- from *Universe* to *Main*, or
- from *Multiverse* to *Restricted*.

[175] https://mir-server.io/

This review process is called **Main Inclusion Review (MIR)**.

3.13.1. Submit a package for Main Inclusion Review

The Main Inclusion Review documentation[176] by the MIR team provides instructions on how to apply for *Main Inclusion Review* for a package. The documentation even contains details of how the application gets reviewed by the MIR team.

> **Note**
>
> The guidelines and review process is constantly evolving. Therefore you should re-read the MIR documentation even if you have submitted a package for Main Inclusion Review in the past.
>
> The MIR documentation is also a living document. External contributions, suggestions, discussions or questions about the process are always welcome.

3.13.2. MIR team weekly meeting

The MIR team holds weekly meetings every Tuesday at 16:30 CET on the *IRC* server `irc.libera.chat` in the `#ubuntu-meeting` channel. You can follow these instructions[177] on how to connect to `irc.libera.chat`.

The purpose of the meeting is:

- to distribute the workload fairly between the members of the MIR team
- to provide a timely response to reporters of MIR applications
- detection and discussion of any current or complex cases

You should attend these meetings if you submit an MIR request until it is approved or rejected.

Usually, the amount of MIR requests increases during the six-month development period of a new Ubuntu release. Especially right before the various feature freezes (see *Ubuntu development process* (page 58)), Ubuntu developers submit MIR requests they have been working on before they have to submit an exception request. As a result, the meetings tend to be quieter, and response times to MIR requests are, on average, faster after the release of a new Ubuntu version.

3.13.3. Resources

- **Main Inclusion Review documentation[178] by the MIR team**
  - MIR process overview[179]
  - MIR application template[180]
  - Helper tools[181]
  - Bug lists[182]

[176] https://github.com/canonical/ubuntu-mir
[177] https://libera.chat/guides/connect
[178] https://github.com/canonical/ubuntu-mir
[179] https://github.com/canonical/ubuntu-mir#process-states
[180] https://github.com/canonical/ubuntu-mir#main-inclusion-requirements
[181] https://github.com/canonical/ubuntu-mir#tools
[182] https://github.com/canonical/ubuntu-mir#bug-lists

  - Pull requests[183]
  - Issues[184]
- MIR team on *Launchpad*: `~ubuntu-mir`[185]

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

[183] https://github.com/canonical/ubuntu-mir/pulls
[184] https://github.com/canonical/ubuntu-mir/issues
[185] https://launchpad.net/~ubuntu-mir

4. Reference

Our reference section contains support information related to packaging in Ubuntu. This includes details on the network requirements, API definitions, support matrices, and so on.

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

4.1. 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 `debian/changelog`, `debian/control`, `debian/copyright`, and `debian/rules`. These are required for all packages. A number of additional files in the `debian/` directory may be used in order to customise and configure the behaviour of the package. Some of these files are discussed in this article, but this is not meant to be a complete list.

4.1.1. The `changelog` file

This file is 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. The following is a template `debian/changelog`:

```
package (version) distribution; urgency=urgency
[optional blank line(s), stripped]
  * change details
    - more change details
  * even more change details
[optional blank line(s), stripped]
 -- maintainer name <email address>[two spaces]  date
```

`package` and `version` are the source package name and version number, respectively.

The `distribution` field lists the distribution(s) in which this release should be installed.

`urgency` describes how important an upgrade is. Its value can be one of the following: `low`, `medium`, `high`, `emergency`, or `critical`.

The change details consist of lines indented by at least two spaces, but these conventionally are a list. Major bullet points use an asterisk "*", while minor bullet points are indicated by a dash "-".

The changelog entry ends with a line indented by one space that contains the name, email of the maintainer, and date of change. The maintainer here is the one responsible for the

release, but it need not be the package maintainer.

> **Note**
>
> If you have a *signing key* (see *Getting set up* (page 3)), then make sure to use the same name and email address in `debian/changelog` entry as you have in your key.

> **Important**
>
> The date should be in **RFC 5322**[186] format, which can be obtained by using the command `date -R`. For convenience, the command `dch` may be used to edit the changelog. It will update the date automatically. For further information, see *dch(1)*[187].

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 `debian/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 distribution revision, used so that the package 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)[188] of the Debian Policy Manual.

4.1.2. The `control` file

The `debian/control` file contains the information that the *package manager* (such as *APT*) uses, build-time dependencies, maintainer information, and much more. The file consists of one or more stanzas of fields, with each stanza separated by empty lines. The fields consist of key-value pairs separated by a colon ":"; conventionally, a single space follows the colon.

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

```
Source: hello
Section: devel
```

(continues on next page)

[186] https://datatracker.ietf.org/doc/html/rfc5322.html
[187] https://manpages.ubuntu.com/manpages/noble/en/man1/dch.1.html
[188] https://www.debian.org/doc/debian-policy/ch-source.html#s-dpkgchangelog

(continued from previous page)

```
Priority: optional
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
XSBC-Original-Maintainer: Jane Doe <packager@example.com>
Standards-Version: 4.6.2
Build-Depends: debhelper-compat (= 13), help2man, texinfo
Homepage: https://www.gnu.org/software/hello/

Package: hello
Architecture: any
Depends: ${misc: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 stanza describes the source package. It contains the following fields:

- `Source` (required): The name of the source package.
- `Maintainer` (required): The name and email of the package maintainer.

Note

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[189] on the Ubuntu wiki.

- `Uploaders`: The list of names and email addresses of co-maintainers.
- `Section` (recommended): The application area into which the package has been classified.
- `Priority` (recommended): How important the package is.
- `Build-Depends` fields: Lists the packages required to build the package from source. For a full list of the
- `Standards-Version` (required): The version of Debian Policy that the package complies with.
- `Homepage`: The *upstream* home page.
- *Version Control System* fields:
  - `VCS-Browser`: Web interface to browse the repository.

[189] https://wiki.ubuntu.com/DebianMaintainerField

  - `VCS-<type>`: The repository location. See Version Control System fields (Section 5.6.26)[190] of the Debian Policy Manual for more details.

- `Testsuite`: A comma-separated list of values allowing test execution environments to discover packages which provide tests.
- `Rules-Requires-Root`: Defines whether the source package requires root access during selected targets.

Each additional stanza describes a *binary package* to be built. These stanzas contain the following fields:

- `Package` (required): The name of the binary package.
- `Architecture` (required): The *architectures* supported.
- `Section` (recommended): The application area into which the package has been classified.
- `Priority` (recommended): How important the package is.
- `Essential`: Optional boolean field to prevent the package manager from removing the package when set to `yes`. When this field is absent, the default behaviour is `no`.
- `Depends` fields:
- `Description` (required): Contains a description of the binary package. This field consists of a synopsis and a long description.
- `Homepage`: The upstream home page.
- `Built-Using`: This field is used in cases where the package incorporates parts of other packages and relies on specific versions.
- `Package-Type`: Indicates the type of the package, for example: `deb` or `udeb`.

For further information, see the control file section (Chapter 5)[191] of the Debian Policy Manual.

4.1.3. The `copyright` file

This file gives the *copyright* information for both the upstream source and the packaging. Ubuntu and Debian Policy (Section 12.5)[192] 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+
```

(continues on next page)

[190] https://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-vcs-fields
[191] https://www.debian.org/doc/debian-policy/ch-controlfields.html
[192] https://www.debian.org/doc/debian-policy/ch-docs.html#s-copyrightfile

(continued from previous page)

```
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[193] format. You are encouraged to use this format as well.

4.1.4. The `rules` file

The `debian/rules` file 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.

More specifically, the `debian/rules` file has the following targets:

- `build` (required)

  This target configures and compiles the package.

- `build-arch` (required), `build-indep` (required)

  The `build-arch` target configures and compiles architecture-dependent binary packages (distinguished by not having the `all` value in the `Architecture` field).

  The `build-indep` target configures and compiles architecture-independent binary packages (distinguished by the `all` value for the `Architecture` field).

- `binary` (required), `binary-arch` (required), `binary-indep` (required)

  The `binary` target is all that the user needs to build the binary package(s) from the source package. It is typically an empty target that depends on its two parts,

[193] https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/

`binary-arch` and `binary-indep`.

The `binary-arch` target builds the binary packages which are architecture-dependent.

The `binary-indep` target builds the binary packages which are architecture-independent.

- `clean` (required)

  This target undoes the effects of the `build` and `binary` targets, but it does not affect output files that a `binary` target creates in the parent directory.

- `patch` (optional)

  This target prepares the source for editing. For example, it may unpack additional upstream archives, apply patches, etc.

Here is a simplified version of the `debian/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 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 behaviour.

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**
>
> To use the override feature, you should Build-Depend on `debhelper` version 7.0.50 or above.

Have a look at `/usr/share/doc/debhelper/examples/` and *dh(1)*[194] for more examples. Also see the rules section (Section 4.9)[195] of the Debian Policy Manual.

4.1.5. 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 the `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 stanza in `debian/control`.

See *dh_install(1)*[196] and the install file section (Section 5.11)[197] of the Debian New Maintainers' Guide for additional details.

[194] https://manpages.ubuntu.com/manpages/noble/en/man1/dh.1.html
[195] https://www.debian.org/doc/debian-policy/ch-source.html#s-debianrules
[196] https://manpages.ubuntu.com/manpages/noble/en/man1/dh_install.1.html
[197] https://www.debian.org/doc/manuals/maint-guide/dother.en.html#install

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 (4, at the time of this writing), while the following lines contain any URLs to parse. For example:

```
version=4
http://ftp.gnu.org/gnu/hello/hello-(.*).tar.gz
```

Note

If your tarballs live on *Launchpad*, the `debian/watch` file is a little more complicated (see Question 21146[198] and Bug 231797[199] for why this is). In that case, use something like:

```
version=4
https://launchpad.net/flufl.enum/+download http://launchpad.net/flufl.enum/.*/
flufl.enum-(.+).tar.gz
```

Running `uscan` in the root source directory will now compare the upstream version number in the `debian/changelog` with the latest 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
```

For further information, see *uscan(1)*[200] and the watch file section (Section 4.11)[201] of the Debian Policy Manual.

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

Note

The `debian/source/format` file should always exist. If the file can not be found, the format `1.0` is assumed for backwards compatibility, but *lintian(1)*[202] will warn you about it when

[198] https://answers.launchpad.net/launchpad/+question/21146
[199] https://launchpad.net/launchpad/+bug/231797
[200] https://manpages.ubuntu.com/manpages/noble/en/man1/uscan.1.html
[201] https://www.debian.org/doc/debian-policy/ch-source.html#s-debianwatch

you try to build a source package.

You are strongly recommended 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/`. The patches to be applied automatically are listed in the `debian/patches/series` file.

The Debian DebSrc3.0[203] page summarises additional information concerning the switch to the `3.0` source package formats.

See *dpkg-source(1)*[204] and the source/format section (Section 5.21)[205] of the Debian New Maintainers' Guide for additional details.

4.1.6. 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"[206] further discusses the control, changelog, copyright and rules files. Chapter 5, "Other files under the debian directory"[207] discusses additional files that may be used.

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

4.2. Debian policy

The Debian policy defines the requirements and guidelines for packages in the Debian distribution. It governs how packages should behave, how they interact with each other, and how they fit into the system as a whole.

The policy specifies:

- the structure and contents of the Debian archive
- mandatory technical requirements for inclusion in the distribution

[202] https://manpages.ubuntu.com/manpages/noble/en/man1/lintian.1.html
[203] https://wiki.debian.org/Projects/DebSrc3.0
[204] https://manpages.ubuntu.com/manpages/noble/en/man1/dpkg-source.1.html
[205] https://www.debian.org/doc/manuals/maint-guide/dother.en.html#sourcef
[206] https://www.debian.org/doc/manuals/maint-guide/dreq.en.html
[207] https://www.debian.org/doc/manuals/maint-guide/dother.en.html

- package formatting and control files
- filesystem layout
- *operating system* design principles
- maintainer scripts
- inter-package relationships
- shared library handling

See the Debian Policy Manual[208] for the latest version of the Debian policy.

4.2.1. Policy conformance

It's recommended but not mandatory that every *source package* should conform to the latest version of the Debian policy available at the time of the package's last update.

The `Standards-Version` field in the `debian/control` file of the source package must be filled out to indicate the version of the Debian policy that the package complies with. Also, package maintainers must review policy changes before updating this field.

`debian/control` file

The `debian/control` file defines key metadata for source packages and *binary packages*. It resides in the root of the source package directory and is required for all source packages.

This file consists of stanzas, which are sections of fields separated by empty lines. The first stanza defines the source package. Each following stanza describes a binary package built from that source package.

Here are the required fields in the first stanza of the `debian/control` file of the source package:

`Source`
: The name of the source package, which must be unique within the Debian archive.

`Maintainer`
: Name and email of the primary maintainer. This field is crucial for contacting the maintainer regarding issues, updates, or questions related to the package.

`Standards-Version`
: The version of the Standard the package complies with.

Recommended fields in the first stanza of the `debian/control` file include `Section` and `Priority`.

For more information on `debian/control` files and their fields, see Control files and their fields[209].

`Standards-Version` field

The `Standards-Version` field in the `debian/control` file indicates which version of the Debian policy the package has been reviewed against. The field must appear in the first stanza of the `debian/control` file.

The value of the field, which is the Debian policy version number, has four components:

[208] https://www.debian.org/doc/debian-policy/index.html
[209] https://www.debian.org/doc/debian-policy/ch-controlfields.html

```
Standards-Version: <major>.<minor>.<patch>.<subpatch>
```

Here is a breakdown of the components:

- **Major version** (`<major>`): Incremented for significant policy changes requiring widespread updates.
- **Minor version** (`<minor>`): Changed for substantial but less disruptive updates.
- **Major patch level** (`<patch>`): Updated for any normative binding changes.
- **Minor patch level** (`<subpatch>`): Used for non-functional fixes like typos and clarifications.

Only the first three components are significant. You may include or omit the fourth.

When updating an existing package, only update the `Standards-Version` field after reviewing the differences between the old and new policy versions and updating the package if necessary.

Upgrading checklist

Before updating the `Standards-Version` field, follow these steps to ensure compliance:

1. Check the `Standards-Version` value in `debian/control`.
2. Review the changes introduced in newer versions. Refer to the Upgrading checklist[210] section of the Debian Policy Manual for a summary of the changes made in each version.
3. Review relevant sections of the policy based on listed changes and apply updates only when necessary.
4. Test the package to confirm that it builds and behaves correctly with the new standard.
5. Update the `Standards-Version` field in `debian/control` file to the new version.

4.2.2. Resources

- Debian Policy Manual[211]

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

[210] https://www.debian.org/doc/debian-policy/upgrading-checklist.html
[211] https://www.debian.org/doc/debian-policy/index.html

4.3. Supported architectures

Identifier	Alternative Architecture Names	Endianness	Architecture Type
`amd64`	x86-64, x86_64, x64, AMD64, Intel 64	*Little-Endian*	*CISC*
`i386`[1]	Intel x86, 80x86	*Little-Endian*	*CISC*
`arm64`	ARM64, ARMv8, AArch64	*Little-Endian*	*RISC*
`armhf`	ARM32, ARMv7, AArch32, ARM Hard Float	*Little-Endian*	*RISC*
`ppc64el`	PowerPC64 Little-Endian	*Little-Endian*	*RISC*
`powerpc`	PowerPC (32-bit)	*Big-Endian*	*RISC*
`s390x`	IBM System z, S/390, S390X	*Big-Endian*	*CISC*
`riscv64`	RISC-V (64-bit)	*Little-Endian*	*RISC*

4.3.1. Other architectures

Ubuntu doesn't currently support any other *architectures*. This doesn't mean that Ubuntu won't run on other architectures – in fact it is entirely possible for it to install without a problem, because Ubuntu is based on the *Debian* distribution, which has support for eight additional architectures (see Debian Supported Architectures[212]).

However, if you run into problems, the Ubuntu community may not be able to help you.

4.3.2. Resources

- Ubuntu Wiki – Supported Architectures[213]
- Ubuntu Wiki – i386[214]
- Statement on 32-bit i386 packages for Ubuntu 19.10 and 20.04 LTS[215]
- Ubuntu Wiki – S390X[216]
- Ubuntu Downloads[217]
- Endianness[218]

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

[1] `i386` is a partial-port of Ubuntu, which is supported as a multi-arch supplementary architecture. There is no kernel, no installers, and no bootloaders for `i386`, therefore it cannot be booted as a pure `i386` installation. You have to crossbuild `i386` or build in a `i386` chroot on a `amd64` host.

[212] https://wiki.debian.org/SupportedArchitectures

[213] https://help.ubuntu.com/community/SupportedArchitectures

[214] https://wiki.ubuntu.com/i386

[215] https://canonical.com/blog/statement-on-32-bit-i386-packages-for-ubuntu-19-10-and-20-04-lts

[216] https://wiki.ubuntu.com/S390X

[217] https://ubuntu.com/download

[218] https://en.wikipedia.org/wiki/Endianness

4.4. Filesystem hierarchy standard

Ubuntu adheres to the Filesystem Hierarchy Standard (FHS)[219]. FHS prescribes the structure and organization of directories and files in UNIX-like operating systems. It also promotes uniformity in documentation across systems.

The FHS prescribes required directories, their roles, and their minimum expected contents. It provides a framework for separating shareable from unshareable files and static from variable files.

So, compliance with the FHS ensures that software developers and system administrators can predict where files reside.

4.4.1. File classification

FHS classifies files based on two main distinctions. This classification determines their placement in the directory structure:

- **Shareable vs. unshareable:** Shareable files can be stored on one host and used by others. Examples include libraries in `/usr/lib` and documentation in `/usr/share/doc`. Unshareable files are specific to a single system and cannot be shared. Examples include system configuration files in `/etc` and user-specific files in `/home`.
- **Static vs. Variable:** Static files include binaries, libraries, documentation, and other files that don't change without the system administrator's intervention. Examples include files in `/usr/bin`, `/usr/lib`, or `/etc`. Variable files change during normal system operation. Examples include mail files in `/var/mail` and PID files in `/run`.

These distinctions are interrelated as a file can fall into both classifications. For example:

- Files in `/var/mail` are shareable and variable.
- Files in `/etc` are unshareable and static.

However, static and variable files should be separated since static files can reside on read-only media and don't require frequent backups. To support this, the /var hierarchy was introduced to isolate variable files from static directories like `/usr`, making `/usr` safely mountable as read-only.

4.4.2. Core filesystem hierarchies

The FHS prescribes three main filesystem hierarchies for UNIX-like operating systems:

- the root filesystem (/) hierarchy
- the `/usr` hierarchy
- the `/var` hierarchy

The standard requires these hierarchies to maintain compatibility across UNIX-like systems and support features like read-only mounting of `/usr`. Each hierarchy contains specific subdirectories with defined purposes.

Root filesystem (/) hierarchy

The root filesystem is the top-level directory of the filesystem hierarchy. It contains all the essential components needed to boot, restore, recover, and repair the system. It must remain minimal to ensure reliability, portability, and ease of recovery.

A minimal root filesystem has the following benefits:

[219] https://refspecs.linuxfoundation.org/fhs.shtml

- supports mounting from minimal media, such as recovery disks
- avoids storing unshareable, system-specific files on networked systems
- reduces the risk and impact of data corruption
- supports systems with limited storage or a separate partition

The root filesystem must not contain application-specific directories. All additional components should be in `/usr` or `/var`.

The following are the required directories in the root filesystem:

Directory	Purpose
`/bin`	for essential user command binaries used by all users, such as `cp`, `ls`, `sh`, `mount`
`/boot`	for static files for the bootloader, including the kernel
`/dev`	for device files representing system hardware
`/etc`	for host-specific system configuration files and must not contain binary executables
`/lib`	for shared libraries and kernel modules required by binaries in `/bin` and `/sbin`
`/media`	holds mount points for removable media such as USB drives or CDs
`/mnt`	holds temporary mount point for filesystems intended for manual use by system administrators
`/opt`	holds add-on application packages, with each package in its own subdirectory
`/run`	stores runtime variable data cleared on boot, including process IDs and UNIX-domain sockets
`/sbin`	for essential system binaries for booting and system recovery, such as `fsck` and `shutdown`
`/srv`	for site-specific data served by the system, such as web or FTP data
`/tmp`	holds temporary files, which are not preserved across reboots
`/usr`	secondary hierarchy for read-only user utilities and applications
`/var`	for variable data like logs, mail, and spool files

The following directories may be present in the root filesystem if the corresponding subsystems are installed:

Directory	Purpose
`/home`	for user home directories
`/lib<qualifier>`	for alternate format libraries
`/root`	serves as home directory of the root user

`/usr` hierarchy

The `/usr` hierarchy contains shareable, read-only data. It must not contain any host-specific or variable files to:

- support safe mounting across multiple systems
- support read-only operation
- separate variable data from static program files

- maintain consistency across UNIX-like systems

Large software packages must not use a direct subdirectory under `/usr`. Instead, they should reside in structured paths like `/usr/share`, `/usr/lib`, or `/opt`.

The following are the required directories in the `/usr` hierarchy:

Directory	Purpose
`/usr/bin`	serves as primary directory for user-executable programs
`/usr/lib`	contains object files, libraries, and internal binaries for programs, with subdirectories used per application for architecture-dependent files
`/usr/local`	reserved for system administrator use when installing local software
`/usr/sbin`	holds non-essential system binaries for administration and used by root, but not required for boot or recovery
`/usr/share`	stores read-only, architecture-independent data such as documentation, icons, and manuals

`/var` hierarchy

The `/var` hierarchy stores variable data files. These include system logs, mail, print spool files, cache data, and files generated at runtime. Files in `/var` are modified frequently during system operation. This separation from the static filesystem in `/usr` ensures that the `/usr` filesystem remains read-only.

`/var` should be minimal to reduce the risk of system corruption and to simplify management. Also, applications must not add top-level directories in `/var` without a system-wide implication.

The following are the required directories in the `/var` hierarchy:

Directory	Purpose
`/var/cache`	stores application-generated cache data. The data must be safely disposable and reproducible
`/var/lib`	holds variable state information specific to applications
`/var/local`	holds variable data for software stored in `/usr/local`
`/var/lock`	contains lock files to coordinate access to resources
`/var/log`	stores system log files and directories
`/var/opt`	holds variable data for add-on software packages in `/opt`
`/var/run`	holds transient runtime data, such as `PID` files
`/var/spool`	contains spool directories for tasks like mail and printing
`/var/tmp`	holds temporary files that are preserved between reboots

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

4.5. Package version format

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

4.6. DEP 3 – Patch file headers

This article lists and briefly explains standard fields of the Debian Enhancement Proposal 3 Specification (DEP-3) – Patch Tagging Guidelines[220] for `.patch` file headers and also shows *Sample DEP-3 compliant headers* (page 101).

4.6.1. Standard fields

`Description` or `Subject` required

This obligatory field contains at least a short description on the first line. When `Subject` is used, it is expected that the long description is outside of the structured fields. With `Description` it is possible to embed them in the field using continuation lines.

In both cases, the long description allows for a more verbose explanation of the patch and its history.

This field should explain why the patch is vendor-specific (e.g., branding patch) when that is the case. If the patch has been submitted *upstream* but has been rejected, the description should also document why it's kept and what were the reasons for the reject.

It's recommended to keep each line shorter than 80 characters.

`Origin` required

Note

If the `Author` field is present, the `Origin` field can be omitted and it's assumed that the patch comes from its author.

[220] https://dep-team.pages.debian.net/deps/dep3/

This field should document the origin of the patch. In most cases, it should be a simple URL. For patches backported/taken from upstream, it should point into the upstream *VCS* web interface when possible, otherwise it can simply list the relevant commit identifier (it should be prefixed with "commit:" in that case). For other cases, one should simply indicate the URL where the patch was taken from (mailing list archives, distribution *bug tracking system*, etc.) when possible.

The field can be optionally prefixed with a single keyword followed by a comma and a space to categorize the origin. The allowed keywords are:

- `upstream` in the case of a patch cherry-picked from the upstream VCS,
- `backport` in the case of an upstream patch that had to be modified to apply on the current version,
- `vendor` for a patch created by Debian or another distribution vendor, or
- `other` for all other kind of patches.

In general, a user-created patch grabbed from a *Bug Tracking System* should be categorized as *other*. When copying a patch from another vendor, the meta-information (and hence this field) should be kept if present, or created if necessary with a *vendor* origin.

`Bug-<Vendor>` or `Bug` optional

It contains one URL pointing to the related bug (possibly fixed by the patch). The `Bug` field is reserved for the bug URL of the upstream bug tracker. Those fields can be used multiple times if several bugs are concerned.

The `<Vendor>` name is explicitly encoded in the field name so that vendors can share patches among them without having to update the meta-information in most cases. The upstream bug URL is special cased because it's the central point of cooperation and it must be easily distinguishable among all the bug URLs.

`Forwarded` optional

Any value other than `no` or `not-needed` means that the patch has been forwarded upstream. Ideally the value is an URL proving that it has been forwarded and where one can find more information about its inclusion status.

If the field is missing, its implicit value is `yes` if the `Bug` field is present, otherwise it's `no`. The field is really required only if the patch is vendor specific, in that case its value should be `not-needed` to indicate that the patch must not be forwarded upstream (whereas `no` simply means that it has not yet been done).

`Author` or `From` optional

This field can be used to record the name and email of the patch author (e.g., `John Bear <foo@example.com>`). Its usage is recommended when the patch author did not add copyright notices for his work in the patch itself. It's also a good idea to add this contact information when the patch needs to be maintained over time because it has very little chance of being integrated upstream.

This field can be used multiple times if several people authored the patch.

`Reviewed-by` or `Acked-by` optional

This field can be used to document the fact that the patch has been reviewed and approved by someone. It should list their name and email in the standard format (e.g., `John Bear <foo@example.com>`).

This field can be used multiple times if several people reviewed the patch.

`Last-Update` optional
: This field can be used to record the date when the meta-information was last updated. It should use the ISO date format `YYYY-MM-DD`.

`Applied-Upstream` optional
: This field can be used to document the fact that the patch has been applied upstream.

  It may contain the upstream version expected to contain this patch, or the URL or commit identifier of the upstream commit (with commit identifiers prefixed with `commit:`, as in the `Origin` field), or both separated by a comma and a space.

4.6.2. Sample DEP-3 compliant headers

A patch cherry-picked from upstream:

```
From: Ulrich Drepper <drepper@redhat.com>
Subject: Fix regex problems with some multi-bytes characters

* posix/bug-regex17.c: Add testcases.
* posix/regcomp.c (re_compile_fastmap_iter): Rewrite COMPLEX_BRACKET
handling.

Origin: upstream, http://sourceware.org/git/?p=glibc.git;a=commitdiff;h=bdb56bac
Bug: http://sourceware.org/bugzilla/show_bug.cgi?id=9697
Bug-Debian: http://bugs.debian.org/510219
```

A patch created by the Debian maintainer John Doe, which got forwarded and rejected:

```
Description: Use FHS compliant paths by default
Upstream is not interested in switching to those paths.
.
But we will continue using them in Debian nevertheless to comply with
our policy.
Forwarded: http://lists.example.com/oct-2006/1234.html
Author: John Doe <johndoe-guest@users.alioth.debian.org>
Last-Update: 2006-12-21
```

A vendor specific patch not meant for upstream submitted on the BTS by a Debian developer:

```
Description: Workaround for broken symbol resolving on mips/mipsel
The correct fix will be done in etch and it will require toolchain
fixes.
Forwarded: not-needed
Origin: vendor, http://bugs.debian.org/cgi-bin/bugreport.cgi?msg=80;bug=265678
Bug-Debian: http://bugs.debian.org/265678
Author: Thiemo Seufer <ths@debian.org>
```

A patch submitted and applied upstream:

```
Description: Fix widget frobnication speeds
Frobnicating widgets too quickly tended to cause explosions.
Forwarded: http://lists.example.com/2010/03/1234.html
Author: John Doe <johndoe-guest@users.alioth.debian.org>
Applied-Upstream: 1.2, http://bzr.example.com/frobnicator/trunk/revision/123
Last-Update: 2010-03-29
```

4.6.3. Resources

- DEP-3 Specification – Patch Tagging Guidelines[221]

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

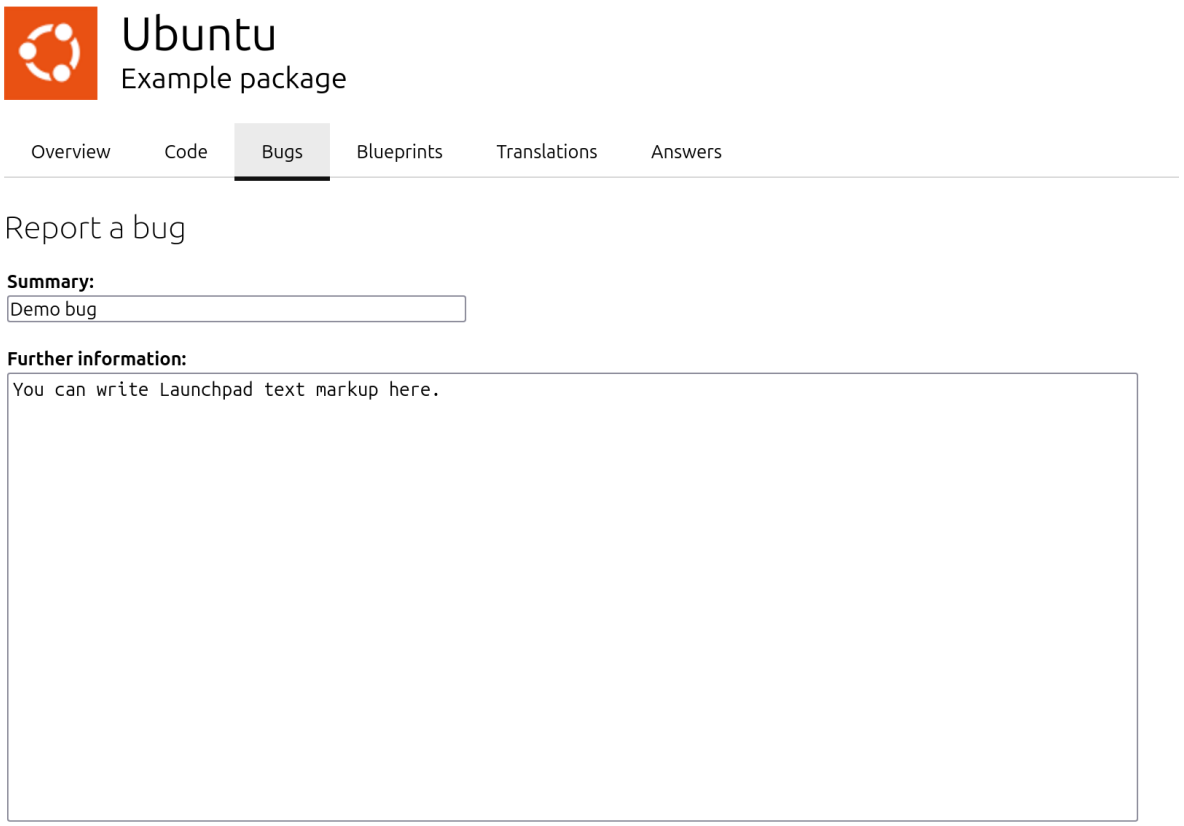
The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

4.7. Launchpad text markup

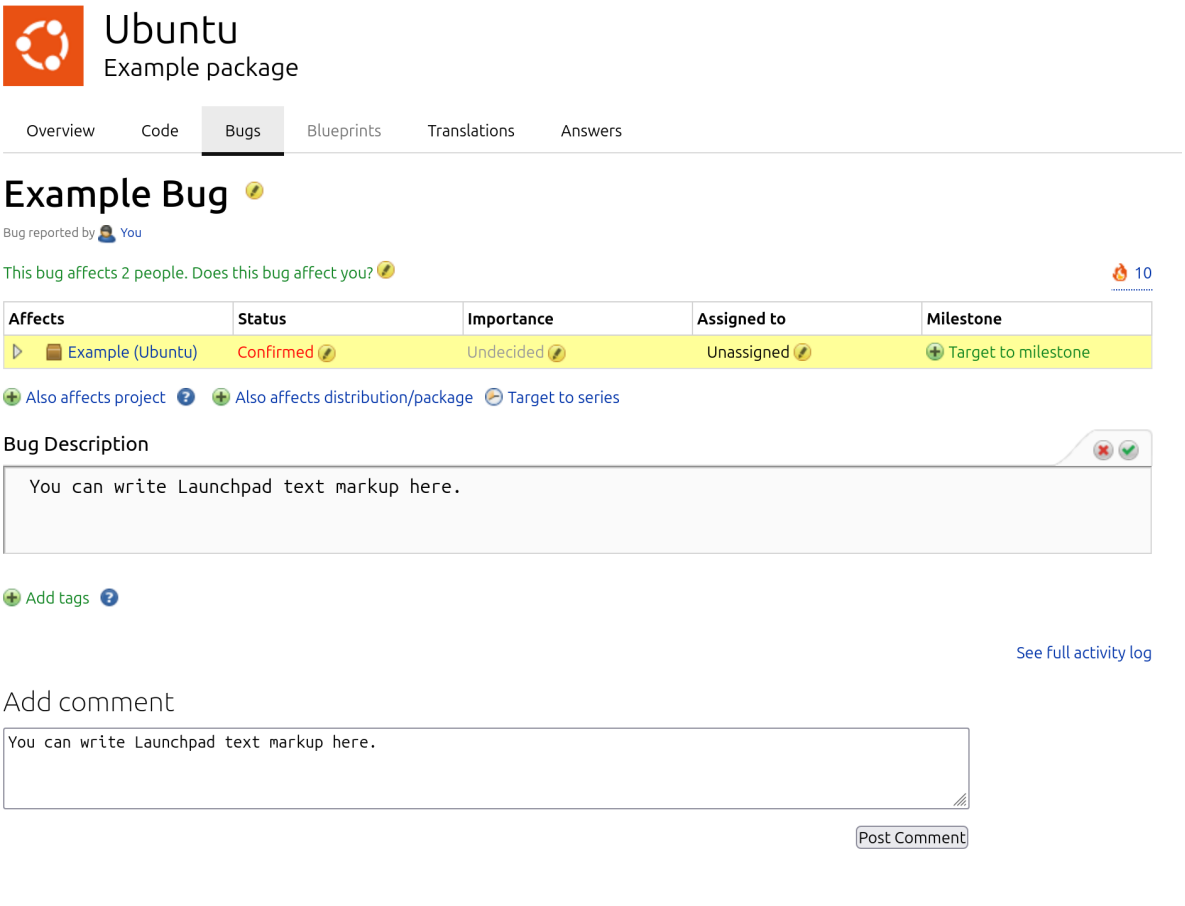
Any textarea[222] input field on Launchpad will process the entered text to recognise certain patterns to enhance the resulting displayed output.

Examples of textareas where the Launchpad text markup is accepted are:

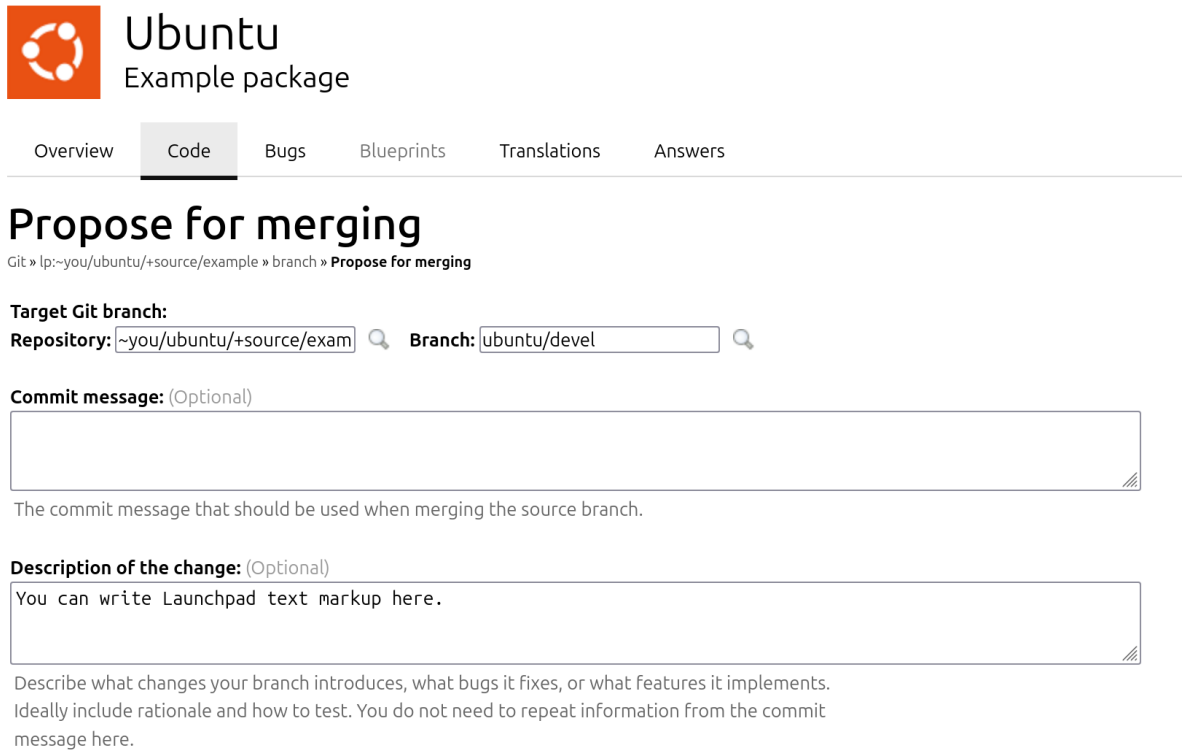


Bug reporting

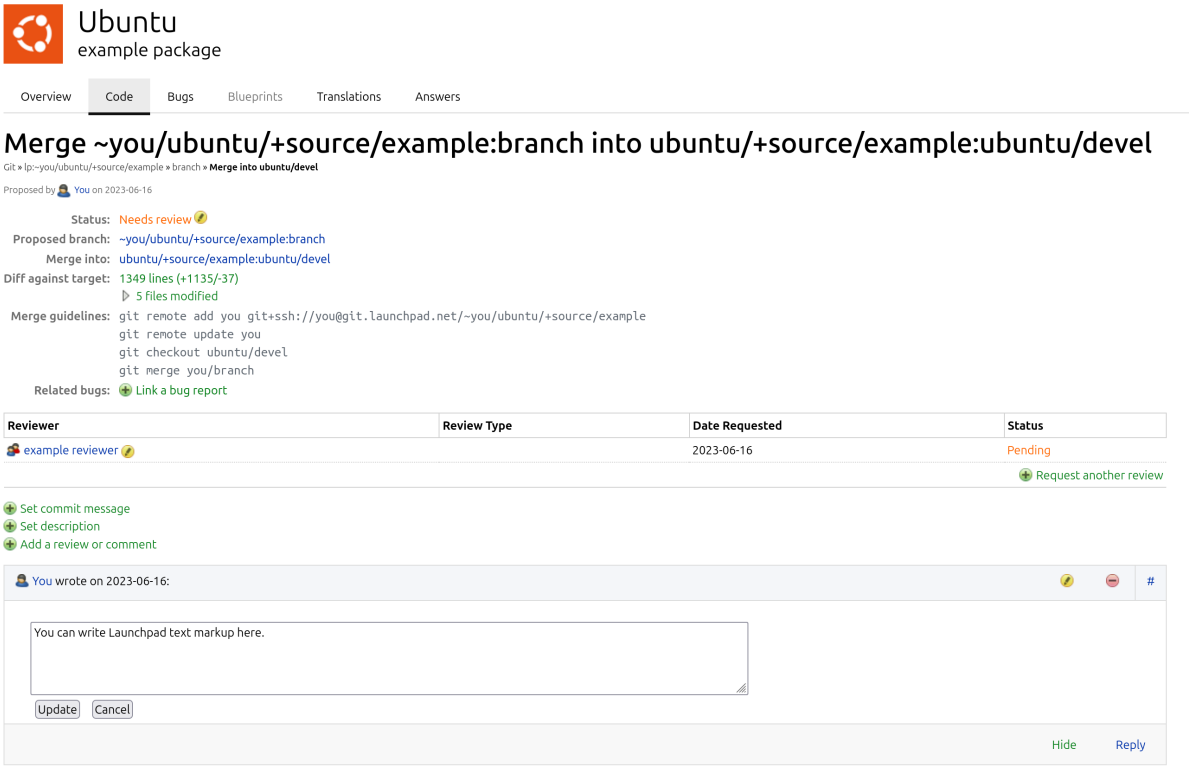
[221] https://dep-team.pages.debian.net/deps/dep3/
[222] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea



Bug report descriptions and comments



Merge proposal creation



Comment for a Merge proposal

You

Overview | Code | Bugs | Blueprints | Translations | Answers

```
You can write Launchpad text markup here.
```

Related packages | Related projects | Authorized applications | OCI registry credentials

User information

Launchpad Id:
you

Email:
you@example.com

Change email settings
Manage mailing list subscriptions

Jabber:
No Jabber IDs registered.

OpenID login:
https://launchpad.net/~you

Member since:
2023-04-11

Signed Ubuntu Code of Conduct:
Yes

IRC:
No IRC nicknames registered.

SSH keys:
No SSH keys registered.

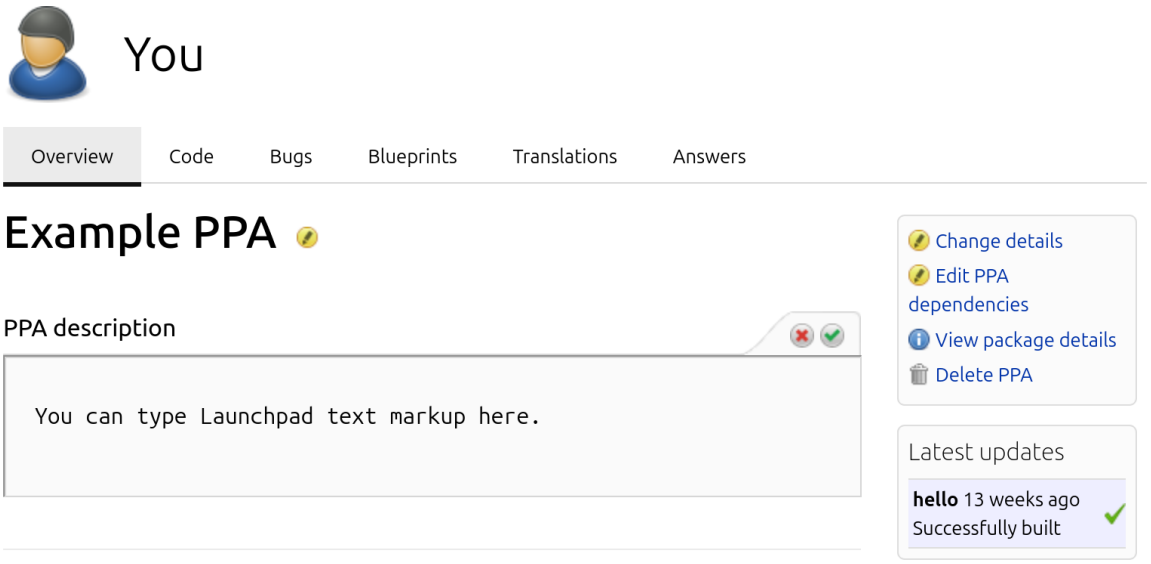
Time zone:
UTC (UTC+0000)

OpenPGP keys:
No OpenPGP keys registered.

Languages:
English

Karma:
0

Profile description



PPA description

Unlike platforms like GitHub, Launchpad unfortunately only recognises a very limited set of markup patterns when you write comments. The most useful pattern are documented in this article.

Note

Support for a wider range of markup patterns is a very common and old request/wish; take for example LP: #391780[223].

You can "upvote" (mark yourself as affected) or leave a comment on this bug report to show your support for the feature request.

Reminder: Please stay civil! The Launchpad team has only limited resources.

4.7.1. Referencing Launchpad bugs

It is very common to refer to a specific Launchpad bug e.g. to point other people to a bug during a discussion.

Pattern

The following pattern is used by Launchpad to detect bug references:

```
LP: #<LP-Bug-Number>[, #<LP-Bug-Number>]...
```

This pattern is case invariant, and the amount of blank space can be variable, but if you place blank space anywhere else, the regular expression used by Launchpad might not parse the bug reference correctly.

Note

[223] https://bugs.launchpad.net/launchpad/+bug/391780

This pattern is also commonly used outside of Launchpad e.g. on *IRC*, in *source package changelogs* or on *Discourse*.

Examples

The following table shows examples how text entered into a text input field will be displayed on Launchpad:

Input	Result	Comment
`LP: #1`	LP: #1[224]	references Launchpad bug with the number 1
`(LP: #1)`	(LP: #1[225])	a bug reference can be surrounded by brackets
`LP: #1, #2.`	LP: #1[226], #2[227].	there can be multiple bug references separated by a ,

`LP:`
`#1,`
`#2,`
`#3,`
`#4`
	LP:
	#1[228],
	#2[229],
	#3[230],
	#4[231]
		the amount of *blank space* can be variable and
		a new-line will not disrupt this pattern

`lp: #1`	lp: #1[232]	the pattern is case invariant
`(lp: #1)`	(lp: #1[233])	the pattern is case invariant
`lp: #1, #2.`	lp: #1[234], #2[235].	the pattern is case invariant
`LP #1`	LP #1	the : is strictly needed
`LP: #1 , #2`	LP: #1[236] , #2	if you place blank space anywhere else the *regular expression* might not parse the input correctly

`LP: #1, #2,`
 	
`#3`
	LP: #1[237], #2[238],
	
	#3
		an empty new-line will interrupt the pattern,
		but a trailing , will not

4.7.2. Blank spaces

Launchpad will:

- cut off any blank space to the right,
- keep any blank space to the left, and
- reduce any blank space between non-blank-space characters to just one (this includes new-line characters as well).

Note

Technically Launchpad passes blank space through and the browser just ignores the blank space.

Warning

Because of the behaviour described above you will have a hard time trying to write a table or long chunks of blank space between two sections.

The following table shows examples how text entered into a text input field will be displayed on Launchpad:

[224] https://bugs.launchpad.net/ubuntu/+bug/1
[225] https://bugs.launchpad.net/ubuntu/+bug/1
[226] https://bugs.launchpad.net/ubuntu/+bug/1
[227] https://bugs.launchpad.net/ubuntu/+bug/2
[228] https://bugs.launchpad.net/ubuntu/+bug/1
[229] https://bugs.launchpad.net/ubuntu/+bug/2
[230] https://bugs.launchpad.net/ubuntu/+bug/3
[231] https://bugs.launchpad.net/ubuntu/+bug/4
[232] https://bugs.launchpad.net/ubuntu/+bug/1
[233] https://bugs.launchpad.net/ubuntu/+bug/1
[234] https://bugs.launchpad.net/ubuntu/+bug/1
[235] https://bugs.launchpad.net/ubuntu/+bug/2
[236] https://bugs.launchpad.net/ubuntu/+bug/1
[237] https://bugs.launchpad.net/ubuntu/+bug/1
[238] https://bugs.launchpad.net/ubuntu/+bug/2

Input	Result

`| Column 1 | Column 2 | Column 3 |`
`|-----------+----------+-----------|`
`| Example | table | text |`
`| Example | table | text |`
`| Example | table | text |`
	| Column 1 | Column 2 | Column 3 |
	|-----------+----------+-----------|
	| Example | table | text |
	| Example | table | text |
	| Example | table | text |

`Here are two paragraphs with lots of blank space between them.`
 	
`But they're still just two paragraphs`
	Here are two paragraphs with lots of blank space between them.
	
	But they're still just two paragraphs

4.7.3. URI addresses

Launchpad can recognise `http`, `https`, `ftp`, `sftp`, `mailto`, `news`, `irc` and `jabber` *URIs*.

Note

`tel`, `urn`, `telnet`, `ldap` *URI*, relative *URLs* like `example.com` and email addresses like `test@example.com` are **NOT** recognised.

Examples

The following examples show how text entered into a text input field will be displayed on Launchpad:

Input	`http://localhost:8086/example/sample.html`
Result	http://localhost:8086/example/sample.html

Input	`http://localhost:8086/example/sample.html`
Result	http://localhost:8086/example/sample.html

Input	`ftp://localhost:8086/example/sample.html`
Result	ftp://localhost:8086/example/sample.html

Input	`sftp://localhost:8086/example/sample.html.`
Result	sftp://localhost:8086/example/sample.html.

Input	`http://localhost:8086/example/sample.html;`
Result	http://localhost:8086/example/sample.html;

Input	`news://localhost:8086/example/sample.html:`
Result	news://localhost:8086/example/sample.html:

Input	`http://localhost:8086/example/sample.html?`
Result	http://localhost:8086/example/sample.html?

Input	`http://localhost:8086/example/sample.html,`
Result	http://localhost:8086/example/sample.html,

Input	`<http://localhost:8086/example/sample.html>`
Result	<http://localhost:8086/example/sample.html>

Input	`<http://localhost:8086/example/sample.html>,`
Result	<http://localhost:8086/example/sample.html>,

Input	`<http://localhost:8086/example/sample.html>.`
Result	<http://localhost:8086/example/sample.html>.

Input	`<http://localhost:8086/example/sample.html>;`
Result	<http://localhost:8086/example/sample.html>;

Input	`<http://localhost:8086/example/sample.html>:`
Result	<http://localhost:8086/example/sample.html>:

Input	`<http://localhost:8086/example/sample.html>?`
Result	<http://localhost:8086/example/sample.html>?

Input	`(http://localhost:8086/example/sample.html)`
Result	(http://localhost:8086/example/sample.html)

Input	`(http://localhost:8086/example/sample.html),`
Result	(http://localhost:8086/example/sample.html),

Input	`(http://localhost:8086/example/sample.html).`
Result	(http://localhost:8086/example/sample.html).

Input	`(http://localhost:8086/example/sample.html);`
Result	(http://localhost:8086/example/sample.html);

Input	`(http://localhost:8086/example/sample.html):`
Result	(http://localhost:8086/example/sample.html):

Input	`http://localhost/example/sample.html?a=b&b=a`
Result	http://localhost/example/sample.html?a=b&b=a

Input	`http://localhost/example/sample.html?a=b&b=a.`
Result	http://localhost/example/sample.html?a=b&b=a.

Input	`http://localhost/example/sample.html?a=b&b=a,`
Result	http://localhost/example/sample.html?a=b&b=a,

Input	`http://localhost/example/sample.html?a=b&b=a;`
Result	http://localhost/example/sample.html?a=b&b=a;

Input	`http://localhost/example/sample.html?a=b&b=a:`
Result	http://localhost/example/sample.html?a=b&b=a:

Input	`http://localhost/example/sample.html?a=b&b=a:b;c@d_e%f~g#h,j!k-l+m$n*o'p`
Result	http://localhost/example/sample.html?a=b&b=a:b;c@d_e%l+m$n*o'p[239]

[239] http://localhost/example/sample.html?a=b&b=a:b;c@d_e%f~g#h,j!k-l+m\protect\TU\textdollarn*o'p

Input	`http://www.example.com/test/example(parentheses).html`
Result	http://www.example.com/test/example(parentheses).html

Input	`http://www.example.com/test/example-dash.html`
Result	http://www.example.com/test/example-dash.html

Input	`http://www.example.com/test/example_underscore.html`
Result	http://www.example.com/test/example_underscore.html

Input	`http://www.example.com/test/example.period.x.html`
Result	http://www.example.com/test/example.period.x.html

Input	`http://www.example.com/test/example!exclamation.html`
Result	http://www.example.com/test/example!exclamation.html

Input	`http://www.example.com/test/example~tilde.html`
Result	http://www.example.com/test/example~tilde.html

Input	`http://www.example.com/test/example*asterisk.html`
Result	http://www.example.com/test/example*asterisk.html

Input	`irc://chat.freenode.net/launchpad`
Result	irc://chat.freenode.net/launchpad

Input	`irc://chat.freenode.net/%23launchpad, isserver`
Result	irc://chat.freenode.net/%23launchpad, isserver

Input	`mailto:noreply@launchpad.net`
Result	mailto:noreply@launchpad.net[240]

Input	`jabber:noreply@launchpad.net`
Result	jabber:noreply@launchpad.net

Input	`http://localhost/foo?xxx&`
Result	http://localhost/foo?xxx&

Input	`http://localhost?testing=[square-brackets-in-query]`
Result	http://localhost?testing={}square-brackets-in-query{}]

[240] noreply@launchpad.net

4.7.4. Removal of “

If the entire comment is encapsulated in “ like this Launchpad will remove the “.

The following table shows an example how text entered into a text input field will be displayed on Launchpad:

Input	Result
`"Content"`	Content

4.7.5. Resources

- Comments (help.launchpad.net)[241]

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

4.8. Glossary

80x86
: See *i386*

AA
: Abbreviation for *Archive Admin*

AArch32
: See *armhf*

AArch64
: See *arm64*

ABI
: Abbreviation for *Application Binary Interface*

Warning

Do not confuse with *Application Programming Interface* (*API*)!

amd64
: *CPU Architecture* identifier for the `AMD64` (also known as *x64*, *x86-64*, *x86_64*, and *Intel 64*) architecture; a 64-bit version of the *i386* instruction set.

[241] https://help.launchpad.net/Comments

See also: X86-64 (Wikipedia)[242]

ANAIS
Abbreviation for *Architecture Not Allowed In Source*

API
Abbreviation for *Application Programming Interface*

> **Warning**
>
> Do not confuse with *Application Binary Interface* (*ABI*)!

Application Binary Interface
Defines how two binary applications interface eachother like calling conventions, data type sizes, and system call interfaces, ensuring compatibility and proper communication between different parts of a software system, such as libraries, executables, and the *Operating System*. *Application Binary Interfaces* are crucial for enabling software components compiled on different systems to work together seamlessly.

See also: Kernel ABI (Ubuntu Wiki)[243], Application binary interface (Wikipedia)[244]

> **Warning**
>
> Do not confuse with *Application Programming Interface* (*API*)!

Application Programming Interface
An *Application Programming Interface* (API), is a set of rules that allows different software applications to communicate with each other. It defines the methods and data formats that applications can use to request and exchange information, perform specific tasks, or access the functionality of another software component, such as an *Operating System*, library, or online service. *APIs* enable developers to build upon existing software and create new applications by providing a standardized way to interact with external systems, services, or libraries without needing to understand their internal workings.

> **Warning**
>
> Do not confuse with *Application Binary Interface* (*ABI*)!

APT
Abbreviation for *Advanced Package Manager*.

See: Advanced Packaging Tool (Ubuntu Server documentation)[245]

Architecture
Within the context of *Ubuntu*, this refers to the system architecture (more specifically, the CPU architecture and its instruction set) an application is designed for.

[242] https://en.wikipedia.org/wiki/X86-64
[243] https://wiki.ubuntu.com/KernelTeam/BuildSystem/ABI
[244] https://en.wikipedia.org/wiki/Application_binary_interface
[245] https://ubuntu.com/server/docs/package-management#advanced-packaging-tool

See also: *Supported architectures* (page 95), Computer Architecture (Wikipedia)[246]

Architecture Not Allowed In Source
Work in Progress

Archive
See *Ubuntu Archive*

Archive Admin
An administrator that is responsible for maintenance tasks of the *Ubuntu Package Archive*, including processing of new *Packages*, migration of *Packages* between *Components*, and other administrative matters.

See also: "Ubuntu Package Archive Administrators" team on Launchpad[247]

Archive Mirror
A *Mirror* of the *Ubuntu Archive*.

See the section *Mirrors* (page 71) for more details.

ARM
ARM (formerly an acronym for *Advanced RISC Machines* and originally *Acorn RISC Machine*) is a widely used family of *RISC CPU Architectures* known for their efficiency, low power consumption, and versatility, which are widely used in *Embedded Systems* and mobile devices.

Notable examples are *arm64* and *armhf*.

See also: ARM architecture family (Wikipedia)[248]

ARM Hard Float
See *armhf*

arm64
CPU Architecture identifier (also known as ARM64, *ARMv8*, and *AArch64*) for a 64-bit *ARM Architecture* variant.

See also: AArch64 (Wikipedia)[249]

armhf
CPU Architecture identifier (also known as ARM32, *ARMv7*, *AArch32*, and *ARM Hard Float*) for a 32-bit *ARM Architecture* variant.

See also: AArch64 (Wikipedia)[250]

ARMv7
See *armhf*

ARMv8
See *arm64*

autopkgtest
`autopkgtest(1)`[251] is a software that interprets and executes tests found in *source packages* that follow the *DEP 8* specification.

[246] https://en.wikipedia.org/wiki/Computer_architecture
[247] https://launchpad.net/~ubuntu-archive
[248] https://en.wikipedia.org/wiki/ARM_architecture_family
[249] https://en.wikipedia.org/wiki/AArch64
[250] https://en.wikipedia.org/wiki/AArch64
[251] https://manpages.ubuntu.com/manpages/noble/en/man1/autopkgtest.1.html

See also: autopkgtest.ubuntu.com[252]

autopkgtest Cloud
: The *Ubuntu project* operates a testing infrastructure used to execute automated tests for Ubuntu *source packages*. It is an implementation of the *DEP 8* specification, enabling large-scale testing across a variety of architectures and environments.

  See:

Backports
: *Work in Progress*

Bazaar
: A distributed *Version Control System* to collaborate on software development, that was developed by *Canonical* and is part of the *GNU* system.

  Bazaar as a *Canonical* project is discontinued. Development has been carried forward in the community as *Breezy*.

  See also: *Bazaar (Launchpad) <https://launchpad.net/bzr>*

> **Note**
>
> *Bazaar* is replaced in favor of a *git*-based workflow as the main *Version Control System* within *Ubuntu*. There are some projects that still use it, but be aware that documents that reference *Bazaar* as an actively used *Version Control System* within *Ubuntu* are most likely outdated.
>
> See also: *git-ubuntu*

best-effort
: *Work in Progress*

Big-Endian
: *Work in Progress*

  See also: *Endianness*

Binaries
: *Work in Progress*

Binary Package
: A *Debian binary package* is a standardized format with the file extension `.deb` that the *Package Manager* (*dpkg(1)*[253] or *apt(8)*[254]) can understand to install and uninstall software on a target machine to simplify distributing software to a target machine and managing software on a target machine.

  See: *Binary Packages (explanation)* (page 56)

Blank space
: *Blank space* characters refer to characters in a text (especially *Source Code*) that are used for formatting and spacing but do not produce visible marks or symbols when rendered. Common *blank space* characters include spaces, tabs and newline characters.

[252] https://autopkgtest.ubuntu.com/
[253] https://manpages.ubuntu.com/manpages/noble/en/man1/dpkg.1.html
[254] https://manpages.ubuntu.com/manpages/noble/en/man8/apt.8.html

Branch
Work in Progress

Breezy
A *Fork* of the *Bazaar Version Control System*.

See also: Breezy (Launchpad)[255]

BTS
Abbreviation for *Bug Tracking System*

Bug
In software development a *"bug"* refers to unintended or unexpected behaviour of a computer program or system that produce incorrect results, or crashes. *Bugs* can occur due to programming mistakes, design issues, or unexpected interactions between different parts of the software.

Identifying and fixing *Bugs* is a fundamental part of the software development process to ensure that the software functions as intended and is free of errors.

See also: Software bug (Wikipedia)[256]

Bug supervisor
Work in Progress

Bug Tracking System
A platform used by software development teams to manage and monitor the progress of reported issues or *Bugs* within a software project. It provides a centralized platform for users to report problems, assign tasks to developers, track the status of issues, prioritize fixes, and maintain a comprehensive record of software defects and their resolutions. This system helps streamline the debugging process and enhances communication among team members, ultimately leading to improved software quality.

Launchpad is the *Bug Tracking System* for *Ubuntu Packages*.

See also: Bug tracking system (Wikipedia)[257]

BZR
Abbreviation for *Bazaar*

Canonical
Canonical Ltd. is a UK-based private company that is devoted to the *Free and Open Source Software* philosophy and created several notable software projects, including *Ubuntu*. *Canonical* offers commercial support for *Ubuntu* and related services and is responsible for delivering six-monthly milestone releases and regular *LTS* releases for enterprise production use, as well as security updates, support and the entire online infrastructure for community interaction.

Find out more on the Canonical website: canonical.com[258]

Canonical Discourse
A *Discourse* instance for internal/company-wide discussions. The discussions here will only be accessible to the *Canonical* employes.

[255] https://launchpad.net/brz
[256] https://en.wikipedia.org/wiki/Software_bug
[257] https://en.wikipedia.org/wiki/Bug_tracking_system
[258] https://canonical.com/

See: discourse.canonical.com[259]

Canonical partner archive
Work in Progress

CD
Abbreviation for *Continuous Delivery*

CD Mirror
A *Mirror* of the *Ubuntu Image* archive (cdimage.ubuntu.com[260]).

See the complete list of officially recognized Ubuntu image archive mirrors[261].

Central Processing Unit
The main component of a computer, that is responsible for executing the instructions of a computer program, such as arithmetic, logic, and input/output (I/O) operations.

Certified Ubuntu Engineer
Develop and certify your skills on the world's most popular *Linux OS*. https://ubuntu.com/credentials

Changelog
The `debian/changelog` file in a *Source Package*.

See: *Basic overview of the debian/ directory* (page 84)

See also: Section 4.4 Debian changelog (Debian Policy Manual v4.6.2.0)[262]

Checkout
Work in Progress

CI
Abbreviation for *Continuous Integration*

Circle of Friends
The *Ubuntu* logo is called *Circle of Friends*, because it is derived from a picture that shows three friends extending their arms, overlapping in the shape of a circle. It should represent the core values of Ubuntu[263]: *Freedom*, *Reliable*, *Precise* and *Collaborative*.

[259] https://discourse.canonical.com
[260] https://cdimage.ubuntu.com/
[261] https://launchpad.net/ubuntu/+cdmirrors
[262] https://www.debian.org/doc/debian-policy/ch-source.html#debian-changelog-debian-changelog
[263] https://design.ubuntu.com/brand

CISC
Abbreviation for *Complex Instruction Set* Computer

CLA
Abbreviation for *Contributor Licence Agreement*

CLI
Abbreviation for *Command Line Interface*

Closed Source Software
Work in Progress

CoC
Abbreviation for *Code of Conduct*

Code name
Work in Progress

Code of Conduct
Work in Progress

See also: *Ubuntu Code of Conduct*

Code Review
Work in Progress

CoF
Abbreviation for *Circle of Friends*

Command Line Interface
Work in Progress

Commit
Work in Progress

Common Vulnerabilities and Exposures
Work in Progress

Complex Instruction Set
A *CPU Architecture* featuring a rich and diverse set of instructions, often capable of performing complex operations in a single instruction. *CISC* processors aim to minimize the number of instructions needed to complete a task, potentially sacrificing execution speed for instruction richness.

See also: Complex instruction set computer (Wikipedia)[264]

[264] https://en.wikipedia.org/wiki/Complex_instruction_set_computer

Component

Components are logical subdivisions or namespaces of the *Packages* in a *Suite* (page 70). The *APT Package Manager* can individually subscribe to the *components* of a *Suite* (page 70).

The *Packages* of an *Ubuntu Series* (page 69) are categorized if they are *Open Source Software* and part of the Base *Packages* for a given *Series* (page 69) and sorted into the *components* *main* (page 71), *restricted* (page 71), *universe* (page 71), or *multiverse* (page 71), as shown in the following table:

	Open Source Software	*Closed Source Software*
Ubuntu Base Packages	*main* (page 71)	*restricted* (page 71)
Community Packages	*universe* (page 71)	*multiverse* (page 71)

See: *Components (explanation)* (page 70)

Continuous Delivery

Work in Progress

See also: Continuous delivery (Wikipedia)[265]

Continuous Integration

Work in Progress

See also: Continuous integration (Wikipedia)[266]

Contributor Licence Agreement

Work in Progress

Control File

The `debian/control` file in a *Source Package*.

See: *Basic overview of the debian/ directory* (page 84)

This can also refer to a *Debian* source control file (`.dsc` file) or the control file in a *Binary Package* (`.deb` file).

See: Chapter 5. Control files and their fields (Debian Policy Manual v4.6.2.0)[267]

Coordinated Release Date

The date at which the details of a *CVE* are to be publicly disclosed.

Copyleft

Licenses which implement copyleft grant certain freedoms to their works, under the condition that these freedoms are preserved in all derivative works.

One famous example of copyleft is the *GNU General Public License*, which gives its users *Free Software* rights as long as equivalent rights are maintained in modified distributions of said software.

Copyright

Work in Progress

Copyright File

The `debian/copyright` file in a *Source Package*.

[265] https://en.wikipedia.org/wiki/Continuous_delivery
[266] https://en.wikipedia.org/wiki/Continuous_integration
[267] https://www.debian.org/doc/debian-policy/ch-controlfields.html

See: *Basic overview of the debian/ directory* (page 84)

See also: Section 4.5. Copyright (Debian Policy Manual v4.6.2.0)[268]

CPU
Abbreviation for *Central Processing Unit*

CRD
Abbreviation for *Coordinated Release Date*

Cryptographic Signature
Work in Progress

CUE
Abbreviation for *Certified Ubuntu Engineer*

Current Release in Development
Ubuntu follows a strict time-based release cycle. Every six months a new *Ubuntu* version is released.

The *"Current Release in Development"* is the *Ubuntu* version that is in development for the next release at any given time. It is also often referred to as "devel".

See: *Ubuntu Releases (explanation)* (page 64)

CVE
Abbreviation for *Common Vulnerabilities and Exposures*

Debian
Debian is a widely used community-driven *Free and Open Source Operating System* known for its stability and extensive software *Repository*. It follows a strict commitment to *Free and Open Source Software* principles and serves as the basis for various *Linux Distributions* (including *Ubuntu*). *Debian' Package Manager*, *APT*, simplifies software installation and updates, making it a popular choice for servers and desktops.

See also: www.debian.org[269]

Debian Enhancement Proposal
A Debian Enhancement Proposal (*DEP*) is a formal document that outlines proposed changes, enhancements, or new processes within the *Debian project*. DEPs provide a structured way for contributors to suggest, discuss, and document improvements to Debian' software, policies, or workflows.

See: dep-team.pages.debian.net[270]

Debian System Administration
Work in Progress

deb
debs
.deb is the file extension of a *Debian Binary Package*.

DEP
Abbreviation for *Debian Enhancement Proposal*

DEP 3
DEP 3 is a specification from the *Debian project* that defines

[268] https://www.debian.org/doc/debian-policy/ch-source.html#copyright-debian-copyright
[269] https://www.debian.org/
[270] https://dep-team.pages.debian.net/

See: *DEP 3 – Patch file headers (reference article)* (page 99)

See also: *Patches (explanation article)* (page 56)

DEP 8

DEP 8 is a specification from the *Debian project* that defines a standardized framework for automated testing of *source* and *binary packages*.

See: Current DEP-8 Specification[271]

Detached Signature

A detached signature is a *Digital Signature* that is separated from the data it signs. In contrast to an embedded signature, which is included within the data it signs, a detached signature is kept as a separate file or entity.

Devel

Shorthand term for the *Current Release in Development*.

Developer Membership Board

Work in Progress

See also: Developer Membership Board (Ubuntu Wiki)[272]

diff

A text format that shows the difference between files that are compared. A file that contains text in this format usually has the file extension *.diff*. This file format does not work well for comparing files in a non-text encoded fromat (e.g. `.bin`, `.png`, `.jpg`).

See also *`diff(1)`*[273], *`git-diff(1)`*[274]

Discourse

An *open-source* forum software that is used by *Ubuntu* and *Canonical*.

See also: *Ubuntu Discourse*, *Canonical Discourse*, Discourse Project Homepage[275]

Distribution

In general, a software *distribution* (also called *"distro"*) is a set of software components that is distributed as a whole to users.

Usually people think specifically of *Linux distributions*. A *Linux distribution* (or distro), is a complete *Operating System* based on the *Linux Kernel*. It includes essential system components, software applications, and *Package Management Tools*, tailored to a specific purpose or user preferences. *Linux* distributions vary in features, desktop environments, and software *Repositories*, allowing users to choose the one that best suits their needs.

See also: Linux distribution (Wikipedia)[276]

DMB

Abbreviation for *Developer Membership Board*

[271] https://dep-team.pages.debian.net/deps/dep8/
[272] https://wiki.ubuntu.com/DeveloperMembershipBoard
[273] https://manpages.ubuntu.com/manpages/noble/en/man1/diff.1.html
[274] https://manpages.ubuntu.com/manpages/noble/en/man1/git-diff.1.html
[275] https://www.discourse.org/
[276] https://en.wikipedia.org/wiki/Linux_distribution

DNS
Abbreviation for *Domain Name System*

Domain Name System
Work in Progress

Downstream
A software project(s) (and associated entities) that depend on another software project directly or indirectly.

See *Downstream (explanation)* (page 50)

DSA
Abbreviation for *Debian System Administration*

dsc
.dsc is the file extension of a *Debian* source control file.

See: Chapter 5. Control files and their fields (Debian Policy Manual v4.6.2.0)[277]

End of Life
Refers to the *End of Support* (Life) for a product/software.

End of Line
The end of a line of *encoded text* is indicated by a control character or sequence of control characters.

This is relevant for text parser which often parse text line by line.

The most common examples for control character(s) that indicate a *end of line* are:

Operating System	Abbreviation*	hex value(s)*	dec value(s)*	Escape sequence*
Unix and *Unix*-like systems	`LF`	`0A`	`10`	`\n`
Windows systems	`CR LF`	`0D 0A`	`13 10`	`\r \n`

* for the character encoding `ASCII`

End of Support
Work in Progress

End-user license agreement
Work in Progress

Embedded Systems
Work in Progress

Endianness
Work in Progress

See also: *Little-Endian*, *Big-Endian*, Endianness (Wikipedia)[278]

EoL
Abbreviation for either *End of Life* or *End of Line*

[277] https://www.debian.org/doc/debian-policy/ch-controlfields.html
[278] https://en.wikipedia.org/wiki/Endianness

EoS
Abbreviation for *End of Support*

ESM
Abbreviation for *Expanded Security Maintenance*

EULA
Abbreviation for *End-user license agreement*

Expanded Security Maintenance
Work in Progress

See also: Expanded Security Maintenance (homepage)[279]

Failed to build from Source
Work in Progress

Failed to install
Work in Progress

Feature Freeze Exception
Work in Progress (see https://wiki.ubuntu.com/FreezeExceptionProcess)

Feature Request
Work in Progress

Federal Information Processing Standards
A set of standards and guidelines of the United States federal government developed by *National Institute of Standards and Technology* (*NIST*) to ensure the security and interoperability of computer systems and software used by non-military federal agencies and its contractors.

See also: Federal Information Processing Standards (Wikipedia)[280]

FFE
Abbreviation for *Feature Freeze Exception*

FIPS
Abbreviation for *Federal Information Processing Standards*

Fork
In the context of *Open Source Software* development, a *"fork"* refers to the process of creating a new, independent version of a software project by copying its *Source Code* to evolve separately, potentially with different goals, features, or contributors.

FOSS
Abbreviation for *Free and Open Source Software*

FR
Abbreviation for *Feature Request*

Free and Open Source Software
The term "Free and Open Source Software" encompasses both *Free Software* and *Open Source Software*. In short, free and open-source software not only makes its *Source Code* publicly available, but also allows users to use, distribute, modify, and distribute modified copies of the software free of charge.

[279] https://ubuntu.com/esm
[280] https://en.wikipedia.org/wiki/Federal_Information_Processing_Standards

See also: Free and open-source software (Wikipedia)[281]

Free Software
A common definition of Free Software is any software which guarantees the Free Software Foundation's four essential freedoms:

- The freedom to run the program as you wish, for any purpose (freedom 0).
- The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). This requires access to the source code.
- The freedom to redistribute copies of the original software program (freedom 2).
- The freedom to distribute copies of your modified versions to others (freedom 3). This also requires access to the source code.

The definition of free software has significant overlap with *Open Source Software*, so the two categories are often collectively referred to as *Free and Open Source Software*.

See also: Free software (Wikipedia)[282], FSF Four Freedoms[283]

FTBFS
Abbreviation for *Failed to build from Source*

FTI
Abbreviation for *Failed to install*

GA
Abbreviation for *General Availability*

General Availability
Work in Progress

General Public License
The *GNU* General Public Licenses (GPL) are a set of *Free Software* licenses. They grant users the ability to use, study, modify, and distribute the software and source code. Additionally, the GPLs are *Copyleft*, so any derivative works must be distributed with the same or requivalent freedoms.

Prominent projects which use a version of the GPL include *git* and *Linux*.

git
Work in Progress

git-ubuntu
Work in Progress

GNU
GNU is a recursive acronym for *"GNU's Not Unix!"*. It is a collection of *Free and Open Source Software* that can be used as an *Operating System* and aims to respect its users' freedom. The collection of *Free and Open Source Software* is often used with *Unix*-like kernels like *Linux* (these *Distributions* are commonly referred to as *"GNU/Linux"*).

For example, *Debian* and *Ubuntu* are *GNU/Linux Distributions*.

Most of the *GNU* software is licensed under the *GNU General Public License* (*GPL*).

[281] https://en.wikipedia.org/wiki/Free_and_open-source_software
[282] https://en.wikipedia.org/wiki/Free_software
[283] https://static.fsf.org/nosvn/posters/handout-four-freedoms.pdf

See also: GNU (Wikipedia)[284], www.gnu.org[285]

GPL
Abbreviation for *GNU General Public License*

GUI
Abbreviation for Graphical *User Interface*

i386
CPU Architecture identifier (also known as *Intel x86*, *80x86*, and *x86*), that was originally released as 80386; a 32-Bit Microprocessor by Intel.

See also: i386 (Wikipedia)[286]

IBM
Work in Progress Abbreviation for *International Business Machines*

Find more information on the IBM website[287].

IBM zSystems
Work in Progress

IC
Abbreviation for *Individual Contributor*

ICE
Abbreviation for *Internal Compiler Error*

IEEE
Abbreviation for *Institute of Electrical and Electronics Engineers*

Intel 64
See *arm64*

Intel x86
See *i386*

IRC
Abbreviation for *Internet Relay Chat*

IRCC
Abbreviation for *Ubuntu IRC Council*

Image
Within the context of *Ubuntu* development, an *"Image"* refers to an `.iso` file that contains a bootable *Ubuntu* installer that can be burned to a CD to make installation disks.

See also: www.releases.ubuntu.com[288], Optical disc image (Wikipedia)[289]

Individual Contributor
Work in Progress

Institute of Electrical and Electronics Engineers
Work in Progress (see https://www.ieee.org/)

[284] https://en.wikipedia.org/wiki/GNU
[285] https://www.gnu.org
[286] https://en.wikipedia.org/wiki/I386
[287] https://www.ibm.com/
[288] https://www.releases.ubuntu.com/
[289] https://en.wikipedia.org/wiki/Optical_disc_image

Intent to Package
Work in Progress (see https://wiki.debian.org/ITP)

Internal Compiler Error
Work in Progress

Internet Relay Chat
Internet Relay Chat (*IRC*)

ISO
Work in Progress

ITP
Abbreviation for *Intent to Package*

Kernel
Work in Progress

Keyring
Work in Progress

Launchpad
The general development platform where *Ubuntu* itself and most of *Ubuntu* related software projects live.

See: *Launchpad (explanation article)* (page 74)

Linux
Linux is an *Open Source Operating System Kernel* originally created by *Linus Torvalds* in 1991. It forms the core of various *Linux Distributions*, such as *Debian* and *Ubuntu*. *Linux* is known for its stability, security, and flexibility, making it a popular choice for servers, desktops, and embedded systems.

See also: Linux (Wikipedia)[290]

LinuxONE
Work in Progress

Linux Containers
See *LXC*

Little-Endian
Work in Progress

See also: *Endianness*

Long Term Support
Work in Progress

LP
Abbreviation for *Launchpad*

LTS
Abbreviation for *Long Term Support*

LXC
Linux Containers (see https://linuxcontainers.org/lxc/introduction/)

[290] https://en.wikipedia.org/wiki/Linux

LXD
LXD is system container manager (see https://documentation.ubuntu.com/lxd/en/latest/)

Main
A *Component* of every *Ubuntu Series* (page 69) in the *Ubuntu Archive* that contains *Open Source Packages* which are supported and maintained by *Canonical*.

See: *Components* (page 70)

Main Inclusion Review
The review process when a *Package* in *Universe* or *Multiverse* gets requested to be promoted to *Main* or *Restricted*.

See: *Main Inclusion Review (explanation article)* (page 81)

Mailing List
Work in Progress

Maintainer
Work in Progress

Masters of the Universe
Work in Progress

Merge
Work in Progress

Merge Conflict
Work in Progress

Merge Proposal
Work in Progress

Micro-Release Exception
In some cases, when *upstream* fixes *bugs*, they do a new "micro-release" instead of just sending *patches*. If all of the changes are appropriate for an *SRU*, then it is acceptable (and usually easier) to just upload the complete new upstream micro-release instead of backporting the individual patches.

See: New upstream microreleases (Ubuntu SRU Documentation)[291]

MIR
Abbreviation for *Main Inclusion Review*

MIR Team
The *Ubuntu* team that reviews requests to promote *Packages* in *Universe* or *Multiverse* to *Main* or *Restricted*.

See: *Main Inclusion Review (explanation article)* (page 81)

Mirror
A server that *"mirrors"* (replicates and keeps in sync) the content of another server to distribute network traffic, reduce latency, and provide redundancy, ensuring high availability and fault tolerance.

See also: *Archive Mirror*, *CD Mirror*

[291] https://documentation.ubuntu.com/sru/en/latest/reference/requirements/#reference-criteria-microreleases

MOTU
Abbreviation for *Masters of the Universe*

MP
Abbreviation for *Merge Proposal*

MRE
Abbreviation for *Micro-Release Exception*

Multiverse
A *Component* of every *Ubuntu Series* (page 69) in the *Ubuntu Archive* that contains *Packages* of *Closed Source Software* or *Open Source Software* restricted by copyright or legal issues. These *Packages* are maintained and supported by the *Ubuntu* community.

See: *Components* (page 70)

Namespace
A concept in computer science and software development that defines a scope or context in which identifiers (such as variable names, functions, or classes) are unique and distinct. It helps prevent naming conflicts and organizes code elements into separate compartments. Namespaces are commonly used in programming languages to group and categorize code, making it more manageable and maintainable. They play a crucial role in encapsulation and modularity, allowing developers to create reusable and organized code structures. Namespaces are particularly important in larger software projects where numerous components and libraries need to coexist without clashing with each other's names.

National Institute of Standards and Technology
Work in Progress

Native Package
Native source packages are *Source Packages* that are their own *Upstream*, therefore they do not have an *orig tarball*.

See: *Native Source Packages (explanation)* (page 52)

Not built from Source
Work in Progress

NBS
Abbreviation for *Not built from Source*

Never Part Of A Stable Release
Work in Progress

NIST
Abbreviation for *National Institute of Standards and Technology*

NPOASR
Abbreviation for *Never Part Of A Stable Release*

NVIU
Abbreviation for *Newer Version in Unstable*

Newer Version in Unstable
Work in Progress

Open Source Software
Open source software is any software with a license that guarantees a certain set of

rights to users of the software: the rights to use, study, modify, and distribute both the software and its source code for any purpose.

The definition of open source software has significant overlap with *Free Software*, so the two categories are often collectively referred to as *Free and Open Source Software*.

See also: The Open Source Initiative's standard definition of Open Source[292]

Operating System
An *operating system* (OS) is essential system software that manages computer hardware and software resources. It provides crucial services for computer programs, including hardware control, task scheduling, memory management, file operations, and user interfaces, simplifying program development and execution.

See also: Operating system (Wikipedia)[293]

orig tarball
original tarball
The `.orig.tar.ext` and `.orig-component.tar.ext` (where `ext` can be `gz`, `bz2`, `lzma` and `xz` and `component` can contain alphanumeric characters (`a-zA-Z0-9`) and hyphens `-`) *`tar(5)`*[294] archive files of a *Debian Source Package* that contains the original *Source* of the *Upstream* project.

See also: *`dpkg-source(1)`*[295], *tarball*

OS
Abbreviation for *Operating System*

OSS
Abbreviation for *Open Source Software*

Package
Work in Progress

Package Manager
Work in Progress

Patch
A *patch* is a (often small) piece of code or a software update designed to fix or improve a computer program or system. It is typically applied to address *Security Vulnerabilities*, *Bugs*, or enhance functionality, ensuring the software remains up-to-date and reliable. *Patches* are essential for maintaining software integrity and security.

See: *Patches (explanation)* (page 56)
See also: Patch (Wikipedia)[296]

PCRE
Abbreviation for *Perl Compatible Regular Expressions*

[292] https://opensource.org/osd
[293] https://en.wikipedia.org/wiki/Operating_system
[294] https://manpages.ubuntu.com/manpages/noble/en/man5/tar.5.html
[295] https://manpages.ubuntu.com/manpages/noble/en/man1/dpkg-source.1.html
[296] https://en.wikipedia.org/wiki/Patch_(computing)

Perl Compatible Regular Expressions
Work in Progress

See also: PCRE (Reference Implementation)[297]

Personal Package Archive
Work in Progress

PKCS
Abbreviation for *Public Key Cryptography Standards*

Pocket
A *pocket* is a *Package* sub-*repository* within the *Ubuntu Archive*. Every *Ubuntu Series* has the *pockets release* (page 69), *security* (page 69), *updates* (page 69), *proposed* (page 69), and *backports* (page 70).

See: *Pockets (explanation)* (page 69)

POSIX
Abbreviation for *Portable Operating System Interface*: A family of standards specified by the *IEEE* Computer Society for maintaining compatibility between *Operating Systems*. POSIX defines the *API*, along with command line shells and utility interfaces, for software compatibility with variants of Unix and other *Operating Systems*.

PowerPC
Work in Progress

PPA
Abbreviation for *Personal Package Archive*

ppc64el
Work in Progress (PowerPC64 Little-Endian)

PR
Abbreviation for *Pull Request*

Public Key Cryptography Standards
Work in Progress

See also: PKCS (Wikipedia)[298]

Pull
Work in Progress

Pull Request
Work in Progress

Push
Work in Progress

Real Time Operating System
Work in Progress

Rebase
Work in Progress

Reduced Instruction Set
a *CPU* characterized by a simplified and streamlined set of instructions, optimized for

[297] https://www.pcre.org/
[298] https://en.wikipedia.org/wiki/PKCS

efficient and fast execution of basic operations. *RISC* processors typically prioritize speed over complexity.

Examples of *RISC Architectures* are *arm64*, *armhf*, *RISC-V*, *ppc64el*, and *PowerPC*.

See also: Reduced instruction set computer (Wikipedia)[299]

RegEx
Abbreviation for *Regular Expression*

Regular Expression
A sequence of characters that specifies a text-matching pattern. String-search algorithms usually use these patterns for input validation or find (and replace) operations on strings.

While this general term stems from theoretical computer science and formal language theory, people usually think of *Perl Compatible Regular Expressions* (*PCRE*).

Repository
Work in Progress

> **Note**
> ambiguity between git or apt repository

Request for Comments
Work in Progress

See also: Request for Comments (Wikipedia)[300]

Request of Maintainer
Work in Progress

Request of Porter
Work in Progress

Requested by the QA team
Work in Progress

Request of Security Team
Work in Progress

Request of Stable Release Manager
Work in Progress

Restricted
A *Component* of every *Ubuntu Series* (page 69) in the *Ubuntu Archive* that contains *Closed Source Packages* which are supported and maintained by *Canonical*.

See: *Components* (page 70)

RFC
Abbreviation for *Request for Comments*

RISC
Abbreviation for *Reduced Instruction Set* Computer

[299] https://en.wikipedia.org/wiki/Reduced_instruction_set_computer
[300] https://en.wikipedia.org/wiki/Request_for_Comments

RISC-V
Work in Progress

riscv64
Work in Progress

RoM
Abbreviation for *Request of Maintainer*

Root
Work in Progress

RoP
Abbreviation for *Request of Porter*

RoQA
Abbreviation for *Requested by the QA team*

RoSRM
Abbreviation for *Request of Stable Release Manager*

RoST
Abbreviation for *Request of Security Team*

RTOS
Abbreviation for *Real Time Operating System*

Rules File
The `debian/rules` file in a *Source Package*.

See: *Basic overview of the debian/ directory* (page 84)

See also: Section 4.9. Main building script (Debian Policy Manual v4.6.2.0)[301]

s390x
Work in Progress

Seeds
Seeds are lists of packages, that define which packages goes into the *Main* component of the *Ubuntu Archive* and which packages goes into the distribution *images*.

Series
A *series* refers to the *Packages* in the *Ubuntu Archive* that target a specific *Ubuntu* version. A *series* is usually referred to by its *Code name*.

See: *Series (explanation)* (page 69)

Service-level Agreement
Work in Progress

Shell
Work in Progress

Signature
A digital signature is a cryptographic record that verifies the authenticity and integrity of data.

Every *Package* in the *Ubuntu Archive* is digitally signed, enabling users to detect data corruption during the download or unwanted/malicious modifications. Furthermore,

[301] https://www.debian.org/doc/debian-policy/ch-source.html#main-building-script-debian-rules

some *Upstream* projects sign their releases, which lets Ubuntu *Maintainers* and users of the corresponding packages verify that the *Source Code* is from the developers of the upstream project.

The tool *gpg(1)*[302] is commonly used to create and modify digital signatures. Further information can be found in the GNU Privacy Handbook[303].

Signing Key
Work in Progress

SLA
Abbreviation for *Service-level Agreement*

Source
Work in Progress

Source Code
The source code of a program is a set of human-readable instructions written in a programming language. Those instructions are later converted to machine code to be directly executed by a computer. Generally, programmers study and modify software by reading and editing the source code.

Source Package
A *Debian source package* contains the *Source* material used to build one or more *Binary Packages*.

See: *Source Packages (explanation)* (page 52)

Source Tree
Work in Progress

Sponsor
Work in Progress

SRU
Abbreviation for *Stable Release Update*

SRU Verification Team
Work in Progress

Stable Release Managers
Work in Progress

Stable Release Update
Work in Progress

Stack
In computer science, a **Stack** is a data-structure that can store a collection of elements linearly with two primary operations:

- "Push": adds an element to the collection
- "Pop": removes the most recently added element in the collection

Stack implementatuons also often have a "Peak" operation to see the most recently added element in the collection without removing it.

[302] https://manpages.ubuntu.com/manpages/noble/en/man1/gpg.1.html
[303] https://www.gnupg.org/gph/en/manual.html#AEN136

The name **Stack** stems from the analogy of items "stacked" ontop of eachother like a stack of plates where you have to remove the plates above to access the plates below.

See also: Stack (abstract data type)[304]

Staging Environment
Work in Progress

Standard Output
Work in Progress

tarball
A file in the *tar(5)*[305] archive format, which collects any number of files, directories, and other file system objects (symbolic links, device nodes, etc.) into a single stream of bytes. The format was originally designed to be used with tape drives, but nowadays it is widely used as a general packaging mechanism.

See also: *orig tarball*

Text Encoding
Text encoding refers to the method or schema used to represent and store text characters in a digital format. It involves assigning numerical codes (typically binary) to each character in a character set, which allows computers to process and display text.

For example, `ASCII` and `UTF-8` are commonly used text encoding formats.

The choice of a text encoding format is essential for ensuring proper character representation, especially when dealing with different languages and special characters.

TLS
Abbreviation for *Transport Layer Security*

TPM
Abbreviation for *Trusted Platform Module*

Transport Layer Security
Work in Progress

Trusted Platform Module
Work in Progress

TUI
Abbreviation for text-based *User Interface*

Ubuntu
The word *"ubuntu"* is derived from the pronunciation of an an ancient African word *"oŏ'boŏntoō"* meaning *'humanity to others'*. It is often described as reminding us that *'I am what I am because of who we all are'*.

The *Ubuntu Operating System* tries to bring that spirit to the world of computers and software. The *Ubuntu Distribution* is a *Debian*-based *Linux Distribution* and aims to represents the best of what the world's software community has shared with the world.

See: The story of Ubuntu[306], Ubuntu ethos[307], Ubuntu Project Governance[308]

[304] https://en.wikipedia.org/wiki/Stack_(abstract_data_type)
[305] https://manpages.ubuntu.com/manpages/noble/en/man5/tar.5.html
[306] https://ubuntu.com/about
[307] https://ubuntu.com/community/ethos
[308] https://ubuntu.com/community/governance

Ubuntu Archive
The *Ubuntu Package Archive* is and *APT Repository* that is preconfigured by default on *Ubuntu* installations. It hosts *Debian Binary Packages* (`.deb` files) and *Source Packages* (`.dsc` files).

See: *Ubuntu Package Archive (explanation)* (page 68)

Ubuntu autopkgtest Cloud
Work in Progress

See: autopkgtest.ubuntu.com[309]

Ubuntu Base Packages
Packages that are in the *Main* or *Restricted Component*. These are packages that are maintained by *Canonical*, because they are fundamental for *Ubuntu*.

See also: *Main Inclusion Review*

Ubuntu Cloud Archive
Work in Progress

See: Cloud Archive (Ubuntu Wiki)[310]

Ubuntu Code of Conduct
Work in Progress

See: https://ubuntu.com/community/ethos/code-of-conduct

Ubuntu CVE Tracker
Work in Progress (see https://launchpad.net/ubuntu-cve-tracker and https://ubuntu.com/security/cves)

Ubuntu Delta
A modification to an *Ubuntu Package* that is derived from a *Debian Package*.

See also: *Upstream & Downstream (explanation)* (page 49)

Ubuntu Desktop
Work in Progress

Ubuntu Developer Summit
Between 2004 and 2012, *Ubuntu* releases were planned during regularly scheduled summits, where the greater *Ubuntu* community would come together for planning and hacking sessions. This event occurred two times a year, each one running for a week. The discussions were highly technical and heavily influenced the direction of the subsequent *Ubuntu* release.

These events were called *"Ubuntu Developer Summit"* (UDS).

These events are continued since November 2022 as "*Ubuntu Summit*" (US) to include the broader *Ubuntu* community and not only developers.

See also: Ubuntu Developer Summit is now Ubuntu Summit (Ubuntu Blog)[311], Developer Summit (Ubuntu Wiki)[312]

[309] https://autopkgtest.ubuntu.com/
[310] https://wiki.ubuntu.com/OpenStack/CloudArchive
[311] https://ubuntu.com/blog/uds-is-now-ubuntu-summit
[312] https://wiki.ubuntu.com/DeveloperSummit

Ubuntu Discourse

A *Discourse* instance about general *Ubuntu* development that is accessible to the general public, where you can find discussions, announcements, team updates, documentation and much more.

Feel free to introduce yourself[313].

See: discourse.ubuntu.com[314]

Ubuntu ESM Team

Work in Progress

Ubuntu flavours

Ubuntu flavours are *Distributions* of the default *Ubuntu* releases, which choose their own default applications and settings. *Ubuntu flavours* are owned and developed by members of the *Ubuntu* community and backed by the full *Ubuntu Archive* for *Packages* and updates.

Officially recognised flavours are:

- Edubuntu[315]
- Kubuntu[316]
- Lubuntu[317]
- Ubuntu Budgie[318]
- Ubuntu Cinnamon[319]
- Ubuntu Kylin[320]
- Ubuntu MATE[321]
- Ubuntu Studio[322]
- Ubuntu Unity[323]
- Xubuntu[324]

Ubuntu IRC Council

Work in Progress

See also: IRC Council (Ubuntu Wiki)[325]

Ubuntu Keyserver

Work in Progress

[313] https://discourse.ubuntu.com/c/intro/101
[314] https://discourse.ubuntu.com
[315] https://edubuntu.org/
[316] https://kubuntu.org/
[317] https://lubuntu.me/
[318] https://ubuntubudgie.org/
[319] https://ubuntucinnamon.org/
[320] https://www.ubuntukylin.com/index-en.html
[321] https://ubuntu-mate.org/
[322] https://ubuntustudio.org/
[323] https://ubuntuunity.org/
[324] https://xubuntu.org/
[325] https://wiki.ubuntu.com/IRC/IrcCouncil

Ubuntu Pro
Work in Progress

See: Ubuntu Pro (homepage)[326]

Ubuntu Server
Work in Progress

Ubuntu SRU Team
Work in Progress

See also: Ubuntu SRU Team[327]

Ubuntu Sponsors
Work in Progress

See also: Ubuntu Sponsors[328]

Ubuntu Security Sponsors
Work in Progress

See also: Ubuntu Security Sponsors Team[329]

Ubuntu Stable Release
Ubuntu stable releases are officially-published versions of Ubuntu and their *packages*.

Ubuntu Summit
The *Ubuntu Summit* (US) is a continuation of *Ubuntu Developer Summit* since November 2022. The change in name aims to broadening the scope, which opens the event up to additional audiences.

While the *Ubuntu Developer Summit* was focused on technical development, the talks and workshops of the *Ubuntu Summit* will cover development as well as design, writing, and community leadership with a wide range of technical skill levels.

The name also results in a nifty new acronym, *"US"*, or more appropriately, simply *"Us"*. This fits very nicely with the meaning of *Ubuntu*, *"I am what I am because of who we all are"*.

If you have any question feel free to send an email at summit@ubuntu.com.

Also, check out the Ubuntu Summit mailing list[330].

You can find more information at summit.ubuntu.com[331].

UCA
Abbreviation for *Ubuntu Cloud Archive*

UCT
Abbreviation for *Ubuntu CVE Tracker*

UDS
Abbreviation for *Ubuntu Developer Summit*

[326] https://ubuntu.com/pro
[327] https://wiki.ubuntu.com/StableReleaseUpdates#Contacting_the_SRU_team
[328] https://launchpad.net/~ubuntu-sponsors
[329] https://launchpad.net/~ubuntu-security-sponsors
[330] https://lists.ubuntu.com/mailman/listinfo/summit-news
[331] https://summit.ubuntu.com/

UI
Abbreviation for *User Interface*

UIFe
Abbreviation for *User Interface Freeze Exception*

Uniform Resource Identifier
Work in Progress

See also: Uniform Resource Identifier (Wikipedia)[332]

Uniform Resource Locator
Work in Progress

See also: URL (Wikipedia)[333]

Universe
A *Component* of every *Ubuntu Series* (page 69) in the *Ubuntu Archive* that contains *Open Source Packages* which are supported and maintained by the *Ubuntu* community.

See: *Components* (page 70)

Unix
Unix is an *Operating System* whose development started in the late 1960s at AT&T Bell Labs. It is characterized by its multi-user and multi-tasking capabilities, hierarchical file system, and a suite of *Command Line* utilities. *Unix* has been influential in shaping modern *Operating Systems* and remains the basis for various *Unix*-like systems, including *Linux* and *macOS*.

See also: Unix (Wikipedia)[334]

Upstream
A software project (and associated entities), another software project depends on directly or indirectly.

See *Upstream (explanation)* (page 49)

URI
Abbreviation for *Uniform Resource Identifier*

URL
Abbreviation for *Uniform Resource Locator*

US
Abbreviation for *Ubuntu Summit*

User Experience
The overall experience and satisfaction a user has while interacting with a product or system. It considers usability, accessibility, user flow, and the emotional response of users to ensure a positive and efficient interaction with the *User Interface* and the product as a whole.

User Interface
Refers to the visual elements and design of a digital product or application that users interact with. It includes components like buttons, menus, icons, and layout, focusing on how information is presented and how users navigate through the interface.

[332] https://en.wikipedia.org/wiki/Uniform_Resource_Identifier
[333] https://en.wikipedia.org/wiki/URL
[334] https://en.wikipedia.org/wiki/Unix

User Interface Freeze Exception
Work in Progress

See: *Ubuntu development process* (page 58)

UX
Abbreviation for *User Experience*

VCS
Abbreviation for *Version Control System*

Version Control System
A software tool or system that enables developers to track and manage changes to their *Source Code* and collaborate with others effectively. It maintains a history of *Source Code* revisions, allowing users to revert to previous versions, track modifications, and work on different *Branches* of *Source Code* simultaneously. *Version Control Systems* are crucial for *Source Code* management and collaboration in *Open Source Software* development projects.

Waiting on Upstream
Work in Progress

See also: *Upstream*

Watch File
The `debian/watch` file in a *Source Package*.

See: *Basic overview of the debian/ directory* (page 84)

See also: *uscan(1)*[335], Section 4.11. Upstream source location (Debian Policy Manual v4.6.2.0)[336]

WoU
Abbreviation for *Waiting on Upstream*

x64
See *amd64*

x86
See *i386*

x86-64
See *amd64*

x86_64
See *amd64*

Caution

The Packaging and Development guide is currently undergoing a major overhaul to bring it up to date. The current state you are seeing now is a preview of this effort.

The current version is unstable (changing URLs can occur at any time) and most content is not in properly reviewed yet. Proceed with caution and be aware of technical inaccuracies.

[335] https://manpages.ubuntu.com/manpages/noble/en/man1/uscan.1.html
[336] https://www.debian.org/doc/debian-policy/ch-source.html#upstream-source-location-debian-watch

If you are an experienced packager and would like to contribute, we would love for you to be involved! See *our contribution page* (page 145) for details of how to join in.

5. Contribute to the Ubuntu Packaging Guide

The Ubuntu Packaging Guide[337] is an open source project that warmly welcomes community contributions and suggestions.

This document describes how to contribute changes to the Ubuntu Packaging Guide. If you don't already have a GitHub account, you can sign up on their website[338].

5.1. How to contribute

5.1.1. I want to raise an issue

We use GitHub issues to track things that need to be fixed. If you find a problem and want to report it to us, you can click on the "Give feedback" button at the top of any page in the Guide, and it will open an issue for you.

Alternatively, you can open an issue directly[339] and describe the problem you're having, or the suggestion you want to add.

5.1.2. I have a question about packaging

If you're stuck and have a question, you can use our GitHub discussion board to ask, or start a discussion[340].

Note that we may not be able to respond immediately, so please be patient!

5.1.3. I want to submit a fix

If you found an issue and want to submit a fix for it, or have written a guide you would like to add to the documentation, feel free to open a pull request to submit your fix[341] against our `2.0-preview` branch. If you need help, please use the discussion board or contact one of the repository administrators.

5.2. Contribution format for the project

5.2.1. Sphinx & reStructuredText

The Guide is built using Sphinx[342]. Articles should be written in reStructuredText. The following links might be helpful:

- A ReStructuredText Primer[343]
- Quick reStructuredText[344]

[337] https://github.com/canonical/ubuntu-packaging-guide
[338] https://github.com
[339] https://github.com/canonical/ubuntu-packaging-guide/issues
[340] https://github.com/canonical/ubuntu-packaging-guide/discussions
[341] https://github.com/canonical/ubuntu-packaging-guide/pulls
[342] https://www.sphinx-doc.org/
[343] https://docutils.sourceforge.io/docs/user/rst/quickstart.html
[344] https://docutils.sourceforge.io/docs/user/rst/quickref.html

5.2.2. How to add a new Sphinx extension

In general, there are two places you will need to update to add new extensions.

- `docs/conf.py` - add the name of the extension to the extensions configuration parameter
- `docs/.sphinx/requirements.txt` - add the name of the extension to the bottom of the list

The documentation for most Sphinx extensions will tell you what text to add to the `conf.py` file, as in this example:

```
extensions = [
    'sphinx_copybutton',
    'sphinx_design',
]
```

5.2.3. Translations

We use the localisation (l10n) module for Sphinx and gettext for translating the Ubuntu Packaging Guide.

Some notes about translating the guide:

- Some formatting is part of reStructuredText and should not be changed, including emphasis (which uses asterisks or underscores), paragraph ending before a code block (`::`) and double backtick quotes (``` `` ```).
- The Guide uses email-style reStructuredText links. If you see a link in the text like:

  ```
  `Translatable link text <Link_Reference_>`_
  ```

  Then replace the "Translatable link text" with your translations, but keep the `Link_Reference` unchanged (even if it is in English). The same applies if a URL is used instead of `Link_Reference`.

To test your translation, use `make BUILDER-LANGUAGE` command (for example, `make html-it` will build HTML docs in Italian language).