Chapter

15 Creating a Debian Package

Contents

It is quite common, for an administrator who has been handling Debian packages in a regular fashion, to eventually feel the need to create their own packages, or to modify an existing package. This chapter aims to answer the most common questions in this field, and provide the required elements to take advantage of the Debian infrastructure in the best way. With any luck, after trying your hand for local packages, you may even feel the need to go further than that and join the Debian project itself!

15.1. Rebuilding a Package from its Sources

Rebuilding a binary package is required under several sets of circumstances. In some cases, the administrator needs a software feature that requires the software to be compiled from sources, with a particular compilation option; in others, the software as packaged in the installed version of Debian is not recent enough. In the latter case, the administrator will usually build a more recent package taken from a newer version of Debian — such as *Testing* or even *Unstable* — so that this new package works in their *Stable* distribution; this operation is called "backporting". As usual, care should be taken, before undertaking such a task, to check whether it has been done already — a quick look on the Debian Package Tracker for that package will reveal that information.

➠ `https://tracker.debian.org/`

15.1.1. Getting the Sources

Rebuilding a Debian package starts with getting its source code. The easiest way is to use the `apt-get source` *`source-package-name`* command. This command requires a deb-src line in the `/etc/apt/sources.list` file, and up-to-date index files (i.e. `apt-get update`). These conditions should already be met if you followed the instructions from the chapter dealing with APT configuration (see section 6.1, "Filling in the `sources.list` File" page 100). Note however, that you will be downloading the source packages from the Debian version mentioned in the deb-src line. If you need another version, you may need to download it manually from a Debian mirror or from the web site. This involves fetching two or three files (with extensions `*.dsc` — for *Debian Source Control* — `*.tar.`*`comp`*, and sometimes `*.diff.gz` or `*.debian.tar.`*`comp`* — *comp* taking one value among gz, bz2 or xz depending on the compression tool in use), then run the `dpkg-source -x` *`file.dsc`* command. If the `*.dsc` file is directly accessible at a given URL, there is an even simpler way to fetch it all, with the `dget` *`URL`* command. This command (which can be found in the *devscripts* package) fetches the `*.dsc` file at the given address, then analyzes its contents, and automatically fetches the file or files referenced within. Once everything has been downloaded, it extracts the source package (unless the -d or --download-only option is used).

15.1.2. Making Changes

The source of the package is now available in a directory named after the source package and its version (for instance, *samba-4.1.17+dfsg*); this is where we'll work on our local changes.

The first thing to do is to change the package version number, so that the rebuilt packages can be distinguished from the original packages provided by Debian. Assuming the current version is 2:4.1.17+dfsg-2, we can create version 2:4.1.17+dfsg-2falcot1, which clearly indicates the origin of the package. This makes the package version number higher than the one provided by Debian, so that the package will easily install as an update to the original package. Such a change is best effected with the `dch` command (*Debian CHangelog*) from the *devscripts* package, with an

command such as `dch --local falcot`. This invokes a text editor (`sensible-editor` — this should be your favorite editor if it is mentioned in the `VISUAL` or `EDITOR` environment variables, and the default editor otherwise) to allow documenting the differences brought by this rebuild. This editor shows us that `dch` really did change the `debian/changelog` file.

When a change in build options is required, the changes need to be made in `debian/rules`, which drives the steps in the package build process. In the simplest cases, the lines concerning the initial configuration (./configure …) or the actual build ($(MAKE) … or make …) are easy to spot. If these commands are not explicitly called, they are probably a side effect of another explicit command, in which case please refer to their documentation to learn more about how to change the default behavior. With packages using `dh`, you might need to add an override for the `dh_auto_configure` or `dh_auto_build` commands (see their respective manual pages for explanations on how to achieve this).

Depending on the local changes to the packages, an update may also be required in the `debian/control` file, which contains a description of the generated packages. In particular, this file contains Build-Depends lines controlling the list of dependencies that must be fulfilled at package build time. These often refer to versions of packages contained in the distribution the source package comes from, but which may not be available in the distribution used for the rebuild. There is no automated way to determine if a dependency is real or only specified to guarantee that the build should only be attempted with the latest version of a library — this is the only available way to force an *autobuilder* to use a given package version during build, which is why Debian maintainers frequently use strictly versioned build-dependencies.

If you know for sure that these build-dependencies are too strict, you should feel free to relax them locally. Reading the files which document the standard way of building the software — these files are often called `INSTALL` — will help you figure out the appropriate dependencies. Ideally, all dependencies should be satisfiable from the distribution used for the rebuild; if they are not, a recursive process starts, whereby the packages mentioned in the Build-Depends field must be backported before the target package can be. Some packages may not need backporting, and can be installed as-is during the build process (a notable example is *debhelper*). Note that the backporting process can quickly become complex if you are not careful. Therefore, backports should be kept to a strict minimum when possible.

TIP

Installing `Build-Depends`

`apt-get` allows installing all packages mentioned in the `Build-Depends` fields of a source package available in a distribution mentioned in a `deb-src` line of the `/etc/apt/sources.list` file. This is a simple matter of running the `apt-get build-dep` *`source-package`* command.

15.1.3. Starting the Rebuild

When all the needed changes have been applied to the sources, we can start generating the actual binary package (`.deb` file). The whole process is managed by the `dpkg-buildpackage` command.

Example 15.1 *Rebuilding a package*

```
$ dpkg-buildpackage -us -uc
[...]
```

TOOL

fakeroot

In essence, the package creation process is a simple matter of gathering in an archive a set of existing (or built) files; most of the files will end up being owned by *root* in the archive. However, building the whole package under this user would imply increased risks; fortunately, this can be avoided with the `fakeroot` command. This tool can be used to run a program and give it the impression that it runs as *root* and creates files with arbitrary ownership and permissions. When the program creates the archive that will become the Debian package, it is tricked into creating an archive containing files marked as belonging to arbitrary owners, including *root*. This setup is so convenient that `dpkg-buildpackage` uses `fakeroot` by default when building packages.

Note that the program is only tricked into "believing" that it operates as a privileged account, and the process actually runs as the user running `fakeroot` *`program`* (and the files are actually created with that user's permissions). At no time does it actually get root privileges that it could abuse.

The previous command can fail if the Build-Depends fields have not been updated, or if the related packages are not installed. In such a case, it is possible to overrule this check by passing the -d option to `dpkg-buildpackage`. However, explicitly ignoring these dependencies runs the risk of the build process failing at a later stage. Worse, the package may seem to build correctly but fail to run properly: some programs automatically disable some of their features when a required library is not available at build time.

More often than not, Debian developers use a higher-level program such as `debuild`; this runs `dpkg-buildpackage` as usual, but it also adds an invocation of a program that runs many checks to validate the generated package against the Debian policy. This script also cleans up the environment so that local environment variables do not "pollute" the package build. The `debuild` command is one of the tools in the *devscripts* suite, which share some consistency and configuration to make the maintainers' task easier.

QUICK LOOK

pbuilder

The `pbuilder` program (in the similarly named package) allows building a Debian package in a *chrooted* environment. It first creates a temporary directory containing the minimal system required for building the package (including the packages mentioned in the *Build-Depends* field). This directory is then used as the root directory (`/`), using the `chroot` command, during the build process.

This tool allows the build process to happen in an environment that is not altered by users' manipulations. This also allows for quick detection of the missing build-dependencies (since the build will fail unless the appropriate dependencies are documented). Finally, it allows building a package for a Debian version that is not the one used by the system as a whole: the machine can be using *Stable* for its normal workload, and a `pbuilder` running on the same machine can be using *Unstable* for package builds.

15.2. Building your First Package

15.2.1. Meta-Packages or Fake Packages

Fake packages and meta-packages are similar, in that they are empty shells that only exist for the effects their meta-data have on the package handling stack.

The purpose of a fake package is to trick `dpkg` and `apt` into believing that some package is installed even though it's only an empty shell. This allows satisfying dependencies on a package when the corresponding software was installed outside the scope of the packaging system. Such a method works, but it should still be avoided whenever possible, since there is no guarantee that the manually installed software behaves exactly like the corresponding package would and other packages depending on it would not work properly.

On the other hand, a meta-package exists mostly as a collection of dependencies, so that installing the meta-package will actually bring in a set of other packages in a single step.

Both these kinds of packages can be created by the `equivs-control` and `equivs-build` commands (in the *equivs* package). The `equivs-control` *`file`* command creates a Debian package header file that should be edited to contain the name of the expected package, its version number, the name of the maintainer, its dependencies, and its description. Other fields without a default value are optional and can be deleted. The Copyright, Changelog, Readme and Extra-Files fields are not standard fields in Debian packages; they only make sense within the scope of `equivs-build`, and they will not be kept in the headers of the generated package.

Example 15.2 *Header file of the* libxml-libxml-perl *fake package*

```
Section: perl
Priority: optional
Standards-Version: 3.9.6

Package: libxml-libxml-perl
Version: 2.0116-1
Maintainer: Raphael Hertzog <hertzog@debian.org>
Depends: libxml2 (>= 2.7.4)
Architecture: all
Description: Fake package - module manually installed in site_perl
 This is a fake package to let the packaging system
 believe that this Debian package is installed.
 .
 In fact, the package is not installed since a newer version
 of the module has been manually compiled & installed in the
 site_perl directory.
```

The next step is to generate the Debian package with the `equivs-build` *`file`* command. Voilà: the package is created in the current directory and it can be handled like any other Debian package would.

15.2.2. Simple File Archive

The Falcot Corp administrators need to create a Debian package in order to ease deployment of a set of documents on a large number of machines. The administrator in charge of this task first reads the "New Maintainer's Guide", then starts working on their first package.

➠ https://www.debian.org/doc/manuals/maint-guide/

The first step is creating a `falcot-data-1.0` directory to contain the target source package. The package will logically, be named falcot-data and bear the 1.0 version number. The administrator then places the document files in a `data` subdirectory. Then they invoke the `dh_make` command (from the *dh-make* package) to add files required by the package generation process, which will all be stored in a `debian` subdirectory:

```
$ cd falcot-data-1.0
$ dh_make --native

Type of package: single binary, indep binary, multiple binary, library, kernel module
       , kernel patch?
 [s/i/m/l/k/n] i

Maintainer name : Raphael Hertzog
Email-Address   : hertzog@debian.org
Date            : Fri, 04 Sep 2015 12:09:39 -0400
Package Name    : falcot-data
Version         : 1.0
License         : gpl3
Type of Package : Independent
Hit <enter> to confirm:
Currently there is no top level Makefile. This may require additional tuning.
Done. Please edit the files in the debian/ subdirectory now. You should also
check that the falcot-data Makefiles install into $DESTDIR and not in / .
$
```

The selected type of package (*indep binary*) indicates that this source package will generate a single binary package that can be shared across all architectures (Architecture:all). *single binary* acts as a counterpart, and leads to a single binary package that is dependent on the target architecture (Architecture:any). In this case, the former choice is more relevant since the package only contains documents and no binary programs, so it can be used similarly on computers of all architectures.

The *multiple binary* type corresponds to a source package leading to several binary packages. A particular case, *library*, is useful for shared libraries, since they need to follow strict packaging rules. In a similar fashion, *kernel module* or *kernel patch* should be restricted to packages containing kernel modules.

TIP

Maintainer's name and email address

Most of the programs involved in package maintenance will look for your name and email address in the `DEBFULLNAME` and `DEBEMAIL` or `EMAIL` environment variables. Defining them once and for all will avoid you having to type them multiple times. If your usual shell is `bash`, it is a simple matter of adding the following two lines in your `~/.bashrc` file (you will obviously replace the values with more relevant ones):

```
export EMAIL="hertzog@debian.org"
export DEBFULLNAME="Raphael Hertzog"
```

The `dh_make` command created a `debian` subdirectory with many files. Some are required, in particular `rules`, `control`, `changelog` and `copyright`. Files with the `.ex` extension are example files that can be used by modifying them (and removing the extension) when appropriate. When they are not needed, removing them is recommended. The `compat` file should be kept, since it is required for the correct functioning of the *debhelper* suite of programs (all beginning with the `dh_` prefix) used at various stages of the package build process.

The `copyright` file must contain information about the authors of the documents included in the package, and the related license. In our case, these are internal documents and their use is restricted to within the Falcot Corp company. The default `changelog` file is generally appropriate; replacing the "Initial release" with a more verbose explanation and changing the distribution from unstable to internal is enough. The `control` file was also updated: the Section field has been changed to *misc* and the Homepage, Vcs-Git and Vcs-Browser fields were removed. The Depends fields was completed with iceweasel | www-browser so as to ensure the availability of a web browser able to display the documents in the package.

Example 15.3 *The `control` file*

```
Source: falcot-data
Section: misc
Priority: optional
Maintainer: Raphael Hertzog <hertzog@debian.org>
Build-Depends: debhelper (>= 9)
Standards-Version: 3.9.5

Package: falcot-data
Architecture: all
Depends: iceweasel | www-browser, ${misc:Depends}
Description: Internal Falcot Corp Documentation
 This package provides several documents describing the internal
 structure at Falcot Corp.  This includes:
  - organization diagram
  - contacts for each department.
 .
 These documents MUST NOT leave the company.
 Their use is INTERNAL ONLY.
```

Example 15.4 *The `changelog` file*

```
falcot-data (1.0) internal; urgency=low

  * Initial Release.
  * Let's start with few documents:
    - internal company structure;
    - contacts for each department.

 -- Raphael Hertzog <hertzog@debian.org>  Fri, 04 Sep 2015 12:09:39 -0400
```

Example 15.5 *The `copyright` file*

```
Format: http://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: falcot-data

Files: *
Copyright: 2004-2015 Falcot Corp
License:
 All rights reserved.
```

BACK TO BASICS

`Makefile` file

A `Makefile` file is a script used by the `make` program; it describes rules for how to build a set of files from each other in a tree of dependencies (for instance, a program can be built from a set of source files). The `Makefile` file describes these rules in the following format:

```
target: source1 source2 ...
        command1
        command2
```

The interpretation of such a rule is as follows: if one of the `source*` files is more recent than the `target` file, then the target needs to be generated, using `command1` and `command2`.

Note that the command lines must start with a tab character; also note that when a command line starts with a dash character (`-`), failure of the command does not interrupt the whole process.

The `rules` file usually contains a set of rules used to configure, build and install the software in a dedicated subdirectory (named after the generated binary package). The contents of this subdirectory is then archived within the Debian package as if it were the root of the filesystem. In our case, files will be installed in the `debian/falcot-data/usr/share/falcot-data/` subdirectory, so that installing the generated package will deploy the files under `/usr/share/falcot-data/`. The `rules` file is used as a `Makefile`, with a few standard targets (including

clean and binary, used respectively to clean the source directory and generate the binary package).

Although this file is the heart of the process, it increasingly contains only the bare minimum for running a standard set of commands provided by the `debhelper` tool. Such is the case for files generated by `dh_make`. To install our files, we simply configure the behavior of the `dh_install` command by creating the following `debian/falcot-data.install` file:

```
data/* usr/share/falcot-data/
```

At this point, the package can be created. We will however add a lick of paint. Since the administrators want the documents to be easily accessed from the menus of graphical desktop environments, we add a `falcot-data.desktop` file and get it installed in `/usr/share/applications` by adding a second line to `debian/falcot-data.install`.

Example 15.6 *The `falcot-data.desktop` file*

```
[Desktop Entry]
Name=Internal Falcot Corp Documentation
Comment=Starts a browser to read the documentation
Exec=x-www-browser /usr/share/falcot-data/index.html
Terminal=false
Type=Application
Categories=Documentation;
```

The updated `debian/falcot-data.install` looks like this:

```
data/* usr/share/falcot-data/
falcot-data.desktop usr/share/applications/
```

Our source package is now ready. All that's left to do is to generate the binary package, with the same method we used previously for rebuilding packages: we run the `dpkg-buildpackage -us -uc` command from within the `falcot-data-1.0` directory.

15.3. Creating a Package Repository for APT

Falcot Corp gradually started maintaining a number of Debian packages either locally modified from existing packages or created from scratch to distribute internal data and programs.

To make deployment easier, they want to integrate these packages in a package archive that can be directly used by APT. For obvious maintenance reasons, they wish to separate internal packages from locally-rebuilt packages. The goal is for the matching entries in a `/etc/apt/sources.list.d/falcot.list` file to be as follows:

```
deb http://packages.falcot.com/ updates/
deb http://packages.falcot.com/ internal/
```

The administrators therefore configure a virtual host on their internal HTTP server, with `/srv/vhosts/packages/` as the root of the associated web space. The management of the archive itself is delegated to the `mini-dinstall` command (in the similarly-named package). This tool keeps an eye on an `incoming/` directory (in our case, `/srv/vhosts/packages/mini-dinstall/incoming/`) and waits for new packages there; when a package is uploaded, it is installed into a Debian archive at `/srv/vhosts/packages/`. The `mini-dinstall` command reads the `*.changes` file created when the Debian package is generated. These files contain a list of all other files associated with the version of the package (`*.deb`, `*.dsc`, `*.diff.gz`/`*.debian.tar.gz`, `*.orig.tar.gz`, or their equivalents with other compression tools), and these allow `mini-dinstall` to know which files to install. `*.changes` files also contain the name of the target distribution (often `unstable`) mentioned in the latest `debian/changelog` entry, and `mini-dinstall` uses this information to decide where the package should be installed. This is why administrators must always change this field before building a package, and set it to `internal` or `updates`, depending on the target location. `mini-dinstall` then generates the files required by APT, such as `Packages.gz`.

ALTERNATIVE

apt-ftparchive

If `mini-dinstall` seems too complex for your Debian archive needs, you can also use the `apt-ftparchive` command. This tool scans the contents of a directory and displays (on its standard output) a matching `Packages` file. In the Falcot Corp case, administrators could upload the packages directly into `/srv/vhosts/packages/updates/` or `/srv/vhosts/packages/internal/`, then run the following commands to create the `Packages.gz` files:

```
$ cd /srv/vhosts/packages
$ apt-ftparchive packages updates >updates/Packages
$ gzip updates/Packages
$ apt-ftparchive packages internal >internal/Packages
$ gzip internal/Packages
```

The `apt-ftparchive sources` command allows creating `Sources.gz` files in a similar fashion.

Configuring `mini-dinstall` requires setting up a `~/.mini-dinstall.conf` file; in the Falcot Corp case, the contents are as follows:

```
[DEFAULT]
archive_style = flat
archivedir = /srv/vhosts/packages

verify_sigs = 0
mail_to = admin@falcot.com

generate_release = 1
release_origin = Falcot Corp
release_codename = stable

[updates]
release_label = Recompiled Debian Packages
```

```
[internal]
release_label = Internal Packages
```

One decision worth noting is the generation of `Release` files for each archive. This can help manage package installation priorities using the `/etc/apt/preferences` configuration file (see section 6.2.5, "Managing Package Priorities" page 112 for details).

SECURITY

`mini-dinstall` and permissions

Since `mini-dinstall` has been designed to run as a regular user, there's no need to run it as root. The easiest way is to configure everything within the user account belonging to the administrator in charge of creating the Debian packages. Since only this administrator has the required permissions to put files in the `incoming/` directory, we can deduce that the administrator authenticated the origin of each package prior to deployment and `mini-dinstall` does not need to do it again. This explains the `verify_sigs =0` parameter (which means that signatures need not be verified). However, if the contents of packages are sensitive, we can reverse the setting and elect to authenticate with a keyring containing the public keys of persons allowed to create packages (configured with the `extra_keyrings` parameter); `mini-dinstall` will then check the origin of each incoming package by analyzing the signature integrated to the `*.changes` file.

Invoking `mini-dinstall` actually starts a daemon in the background. As long as this daemon runs, it will check for new packages in the `incoming/` directory every half-hour; when a new package arrives, it will be moved to the archive and the appropriate `Packages.gz` and `Sources.gz` files will be regenerated. If running a daemon is a problem, `mini-dinstall` can also be manually invoked in batch mode (with the -b option) every time a package is uploaded into the `incoming/` directory. Other possibilities provided by `mini-dinstall` are documented in its `mini-dinstall(1)` manual page.

EXTRA

Generating a signed archive

The APT suite checks a chain of cryptographic signatures on the packages it handles before installing them, in order to ensure their authenticity (see section 6.5, "Checking Package Authenticity" page 121). Private APT archives can then be a problem, since the machines using them will keep displaying warnings about unsigned packages. A diligent administrator will therefore integrate private archives with the secure APT mechanism.

To help with this process, `mini-dinstall` includes a `release_signscript` configuration option that allows specifying a script to use for generating the signature. A good starting point is the `sign-release.sh` script provided by the *mini-dinstall* package in `/usr/share/doc/mini-dinstall/examples/`; local changes may be relevant.

15.4. Becoming a Package Maintainer

15.4.1. Learning to Make Packages

Creating a quality Debian package is not always a simple task, and becoming a package maintainer takes some learning, both with theory and practice. It's not a simple matter of building and installing software; rather, the bulk of the complexity comes from understanding the problems and conflicts, and more generally the interactions, with the myriad of other packages available.

Rules

A Debian package must comply with the precise rules compiled in the Debian policy, and each package maintainer must know them. There is no requirement to know them by heart, but rather to know they exist and to refer to them whenever a choice presents a non-trivial alternative. Every Debian maintainer has made mistakes by not knowing about a rule, but this is not a huge problem as long as the error gets fixed when a user reports it as a bug report (which tends to happen fairly soon thanks to advanced users).

➨ `https://www.debian.org/doc/debian-policy/`

Procedures

Debian is not a simple collection of individual packages. Everyone's packaging work is part of a collective project; being a Debian developer involves knowing how the Debian project operates as a whole. Every developer will, sooner or later, interact with others. The Debian Developer's Reference (in the *developers-reference* package) summarizes what every developer must know in order to interact as smoothly as possible with the various teams within the project, and to take the best possible advantages of the available resources. This document also enumerates a number of duties a developer is expected to fulfill.

➨ `https://www.debian.org/doc/manuals/developers-reference/`

Tools

Many tools help package maintainers in their work. This section describes them quickly, but does not give the full details, since they all have comprehensive documentation of their own.

The `lintian` Program This tool is one of the most important: it's the Debian package checker. It is based on a large array of tests created from the Debian policy, and detects quickly and automatically many errors that can then be fixed before packages are released.

This tool is only a helper, and it sometimes gets it wrong (for instance, since the Debian policy changes over time, `lintian` is sometimes outdated). It is also not exhaustive: not getting any

Lintian error should not be interpreted as a proof that the package is perfect; at most, it avoids the most common errors.

The `piuparts` Program This is another important tool: it automates the installation, upgrade, removal and purge of a package (in an isolated environment), and checks that none of these operations leads to an error. It can help in detecting missing dependencies, and it also detects when files are incorrectly left over after the package got purged.

devscripts The *devscripts* package contains many programs helping with a wide array of a Debian developer's job:

- `debuild` allows generating a package (with `dpkg-buildpackage`) and running `lintian` to check its compliance with the Debian policy afterwards.
- `debclean` cleans a source package after a binary package has been generated.
- `dch` allows quick and easy editing of a `debian/changelog` file in a source package.
- `uscan` checks whether a new version of a software has been released by the upstream author; this requires a `debian/watch` file with a description of the location of such releases.
- `debi` allows installing (with `dpkg -i`) the Debian package that was just generated without the need to type its full name and path.
- In a similar fashion, `debc` allows scanning the contents of the recently-generated package (with `dpkg -c`), without needing to type its full name and path.
- `bts` controls the bug tracking system from the command line; this program automatically generates the appropriate emails.
- `debrelease` uploads a recently-generated package to a remote server, without needing to type the full name and path of the related `.changes` file.
- `debsign` signs the `*.dsc` and `*.changes` files.
- `uupdate` automates the creation of a new revision of a package when a new upstream version has been released.

debhelper* and *dh-make Debhelper is a set of scripts easing the creation of policy-compliant packages; these scripts are invoked from `debian/rules`. Debhelper has been widely adopted within Debian, as evidenced by the fact that it is used by the majority of official Debian packages. All the commands it contains have a `dh_` prefix.

The `dh_make` script (in the *dh-make* package) creates files required for generating a Debian package in a directory initially containing the sources for a piece of software. As can be guessed from the name of the program, the generated files use debhelper by default.

`dupload` and `dput` The `dupload` and `dput` commands allow uploading a Debian package to a (possibly remote) server. This allows developers to publish their package on the main Debian server (ftp-master.debian.org) so that it can be integrated to the archive and distributed by mirrors. These commands take a `*.changes` file as a parameter, and deduce the other relevant files from its contents.

15.4.2. Acceptance Process

Becoming a "Debian developer" is not a simple administrative matter. The process comprises several steps, and is as much an initiation as it is a selection process. In any case, it is formalized and well-documented, so anyone can track their progression on the website dedicated to the new member process.

➡ `https://nm.debian.org/`

EXTRA

Lightweight process for "Debian Maintainers"

"Debian Maintainer" is another status that gives less privileges than "Debian developer" but whose associated process is quicker. With this status, the contributors can maintain their own packages only. A Debian developer only needs to perform a check on an initial upload, and issue a statement to the effect that they trust the prospective maintainer with the ability to maintain the package on their own.

Prerequisites

All candidates are expected to have at least a working knowledge of the English language. This is required at all levels: for the initial communications with the examiner, of course, but also later, since English is the preferred language for most of the documentation; also, package users will be communicating in English when reporting bugs, and they will expect replies in English.

The other prerequisite deals with motivation. Becoming a Debian developer is a process that only makes sense if the candidate knows that their interest in Debian will last for many months. The acceptance process itself may last for several months, and Debian needs developers for the long haul; each package needs permanent maintenance, and not just an initial upload.

Registration

The first (real) step consists in finding a sponsor or advocate; this means an official developer willing to state that they believe that accepting *X* would be a good thing for Debian. This usually implies that the candidate has already been active within the community, and that their work has been appreciated. If the candidate is shy and their work is not publicly touted, they can try to convince a Debian developer to advocate them by showing their work in a private way.

At the same time, the candidate must generate a public/private RSA key pair with GnuPG, which should be signed by at least two official Debian developers. The signature authenticates the name on the key. Effectively, during a key signing party, each participant must show an official

identification (usually an ID card or passport) together with their key identifiers. This step confirms the link between the human and the keys. This signature thus requires meeting in real life. If you have not yet met any Debian developers in a public free software conference, you can explicitly seek developers living nearby using the list on the following webpage as a starting point.

➨ `https://wiki.debian.org/Keysigning`

Once the registration on nm.debian.org has been validated by the advocate, an *Application Manager* is assigned to the candidate. The application manager will then drive the process through multiple pre-defined steps and checks.

The first verification is an identity check. If you already have a key signed by two Debian developers, this step is easy; otherwise, the application manager will try and guide you in your search for Debian developers close by to organize a meet-up and a key signing.

Accepting the Principles

These administrative formalities are followed by philosophical considerations. The point is to make sure that the candidate understands and accepts the social contract and the principles behind Free Software. Joining Debian is only possible if one shares the values that unite the current developers, as expressed in the founding texts (and summarized in chapter 1, "The Debian Project" page 2).

In addition, each candidate wishing to join the Debian ranks is expected to know the workings of the project, and how to interact appropriately to solve the problems they will doubtless encounter as time passes. All of this information is generally documented in manuals targeting the new maintainers, and in the Debian developer's reference. An attentive reading of this document should be enough to answer the examiner's questions. If the answers are not satisfactory, the candidate will be informed. They will then have to read (again) the relevant documentation before trying again. In the cases where the existing documentation does not contain the appropriate answer for the question, the candidate can usually reach an answer with some practical experience within Debian, or potentially by discussing with other Debian developers. This mechanism ensures that candidates get involved somewhat in Debian before becoming a full part of it. It is a deliberate policy, by which candidates who eventually join the project are integrated as another piece of an infinitely extensible jigsaw puzzle.

This step is usually known as the *Philosophy & Procedures* (P&P for short) in the lingo of the developers involved in the new member process.

Checking Skills

Each application to become an official Debian developer must be justified. Becoming a project member requires showing that this status is legitimate, and that it facilitates the candidate's job in helping Debian. The most common justification is that being granted Debian developer status eases maintenance of a Debian package, but it is not the only one. Some developers join

the project to contribute to porting to a specific architecture, others want to improve documentation, and so on.

This step represents the opportunity for the candidate to state what they intend to do within the Debian project and to show what they have already done towards that end. Debian is a pragmatic project and saying something is not enough, if the actions do not match what is announced. Generally, when the intended role within the project is related to package maintenance, a first version of the prospective package will have to be validated technically and uploaded to the Debian servers by a sponsor among the existing Debian developers.

COMMUNITY

Sponsoring

Debian developers can "sponsor" packages prepared by someone else, meaning that they publish them in the official Debian repositories after having performed a careful review. This mechanism enables external persons, who have not yet gone through the new member process, to contribute occasionally to the project. At the same time, it ensures that all packages included in Debian have always been checked by an official member.

Finally, the examiner checks the candidate's technical (packaging) skills with a detailed questionnaire. Bad answers are not permitted, but the answer time is not limited. All the documentation is available and several tries are allowed if the first answers are not satisfactory. This step does not intend to discriminate, but to ensure at least a modicum of knowledge common to new contributors.

This step is known as the *Tasks & Skills* step (T&S for short) in the examiners' jargon.

Final Approval

At the very last step, the whole process is reviewed by a DAM (*Debian Account Manager*). The DAM will review all the information about the candidate that the examiner collected, and makes the decision on whether or not to create an account on the Debian servers. In cases where extra information is required, the account creation may be delayed. Refusals are rather rare if the examiner does a good job of following the process, but they sometimes happen. They are never permanent, and the candidate is free to try again at a later time.

The DAM's decision is authoritative and (almost) without appeal, which explains why the people in that seat have often been criticized in the past.

Keywords

Future
Improvements
Opinions