Package training

3. Deb package

deb

deb packages are used in Debian and derivative (Ubuntu, ...)

apt-get install mypackage
dpkg -i mypackage.deb

Where to start

A nice startup guide is available:
https://www.debian.org/doc/manuals/maint-guide/start.en.html

Dependencies

A few packages are needed for a good packaging:

$sudo apt-get install dpkg-dev lintian gnupg dh-make autotools-dev

Env setup

In your ~/.bashrc add the following:

```
DEBEMAIL="your.email.address@example.org"
DEBFULLNAME="Firstname Lastname"
export DEBEMAIL DEBFULLNAME
```

Create ~/.quiltrc: (quilt is for patches, we'll see that later on)

```
QUILT_PATCHES=debian/patches
QUILT_NO_DIFF_INDEX=1
QUILT_NO_DIFF_TIMESTAMPS=1
QUILT_REFRESH_ARGS="-p ab"
```

Get your upstream code

$git clone https://github.com/osallou/training-packaging-1.git training-1.0

Create the Debian files

A quick way to start is to use dh-make. It will create required and optional files to create a Debian package. This is not mandatory however, once you're are familiar with those files, you can create directly a *debian* directory and create the required files.

dh-make

$cd training-1.0
$dh_make --createorig -p training_1.0
=> Select “m” for the type of package, for the purpose of the training we will create a source package that generates multiple binary packages (training , libtraining, libtraining-dev).
vagrant@vagrant-ubuntu-trusty-64:~/training-1.0$ ls debian/
changelog docs manpage.xml.ex preinst.ex rules training.doc-base.EX
compat init.d.ex menu.ex prerm.ex source training-doc.docs
control manpage.1.ex postinst.ex README.Debian training.cron.d.ex training-doc.install
copyright manpage.sgml.ex postrm.ex README.source training.default.ex watch.ex
.ex/.EX files are example files, and will not be taken into account. They are optional files. If one is needed, simply remove the .ex extension.
vagrant@vagrant-ubuntu-trusty-64:~/training-1.0$ ls ..
training-1.0 **training_1.0.orig.tar.xz**
An archive with a specific naming schema has been created, it is necessary to be able to build the package and check that code is not modified by the build.

Debian files

Have a look at https://www.debian.org/doc/manuals/maint-guide/dreq.en.html for main files explanations and https://www.debian.org/doc/manuals/maint-guide/dother.en.html for optional files.

debhelper

debhelper is an helper tool in the build system that will do most of the work for you for common build systems (autotools, ant, python...). It runs a full workflow from configure to build/install/installdoc/....

http://manpages.ubuntu.com/manpages/utopic/en/man7/debhelper.7.html

Clean

The first step in package is the clean target. The clean should always work and must always revert files to the original state (when you downloaded the code).

```
$ debian/rules clean
```

First build attempt

There are different tools to build a .deb but we will focus on dpkg-buildpackage

```
$dpkg-buildpackage -rfakeroot
.....
  dh_install
  dh_installdocs
  dh_installchangelogs
  .....
  dh_builddeb
dpkg-deb: building package `training' in `../training_1.0-1_amd64.deb'.
dpkg-deb: building package `training-doc' in `../training-doc_1.0-1_all.deb'.
 ...
```

Wow, a package!

```
vagrant@vagrant-ubuntu-trusty-64:~/training-1.0$ ls -lstr ..
total 1648
1612 -rw-rw-r--  1 vagrant vagrant 1648556 Feb 17 08:47 training_1.0.orig.tar.xz
  12 -rw-rw-r--  1 vagrant vagrant    9769 Feb 17 09:00 training_1.0-1.debian.tar.gz
   4 -rw-rw-r--  1 vagrant vagrant     893 Feb 17 09:00 training_1.0-1.dsc
   4 drwxrwxr-x 10 vagrant vagrant    4096 Feb 17 09:00 training-1.0
   8 -rw-r--r--  1 vagrant vagrant    4236 Feb 17 09:00 training_1.0-1_amd64.deb
   4 -rw-r--r--  1 vagrant vagrant    2020 Feb 17 09:00 training-doc_1.0-1_all.deb
   4 -rw-rw-r--  1 vagrant vagrant    1800 Feb 17 09:00 training_1.0-1_amd64.changes
```

We see that several files have been created. the deb files are the binary package we have specified (dh-make specified in fact) in our *control* file.
The .dsc and .changes files are “summary” files with generated files list and md5 checksums. We will use them later on.

Let’s open the package

```
vagrant@vagrant-ubuntu-trusty-64:~/training-1.0$ dpkg --contents ../training_1.0-1_amd64.deb
drwxr-xr-x root/root         0 2015-02-17 09:00 ./
drwxr-xr-x root/root         0 2015-02-17 09:00 ./usr/
drwxr-xr-x root/root         0 2015-02-17 09:00 ./usr/share/
drwxr-xr-x root/root         0 2015-02-17 09:00 ./usr/share/doc/
drwxr-xr-x root/root         0 2015-02-17 09:00 ./usr/share/doc/training/
-rw-r--r-- root/root       168 2015-02-17 08:47 ./usr/share/doc/training/changelog.Debian.gz
-rw-r--r-- root/root       173 2015-02-17 08:47 ./usr/share/doc/training/README.Debian
-rw-r--r-- root/root      1666 2015-02-17 08:47 ./usr/share/doc/training/copyright
-rw-r--r-- root/root       335 2015-02-17 08:38 ./usr/share/doc/training/NEWS.gz
-rw-r--r-- root/root       440 2015-02-17 08:38 ./usr/share/doc/training/README
-rw-r--r-- root/root      2882 2015-02-17 08:38 ./usr/share/doc/training/TODO
```

It’s empty. We have not specified what should be put in which package.

Where are my bin, libs...?

```
vagrant@vagrant-ubuntu-trusty-64:~/training-1.0$ ls debian/tmp/usr/bin
squizz
```

The build installed all bins/libs/... in a temporary directory *debian/tmp*.

In the case of a single package (remembered? we selected multiple), this directory would have be named *debian/training* and the package (.deb file) would have contained everything under this directory.

For multiple packages, we need to tell to the system which files should go in X or Y package.

Let’s go for the full package

Important files

- copyright: Check your licenses !!! and embedded ones
- rules: how to build the package, manage its content
- control: package definition, dependencies….

Control file

```
# Source package name
Source: traning
# Where to place the package in repos
Section: science
# You don’t care
Priority: optional
# name/email of person in charge of the package
Maintainer: Myself <my@email.com>
Uploaders: Myself <my@email.com>
Vcs-Svn: svn://...
Vcs-Browser: http://...
Homepage: ftp://ftp.pasteur.fr/pub/gensoft/projects/squizz/
# dependencies to build the package
Build-Depends: debhelper (>= 9), dh-autoreconf
# this evolves in time, add checks/controls
Standards-Version: 3.9.5
```

Control: packages definition

```
# Package ... : binary package to produce (.deb files)
# Usually, base package contains binary and man page
Package: training
# which architectures ?
# any => architecture dependent binary package
# all => architecture independent binary package (java,
python...)
# Can specify a list of specific archs
Architecture: any
# Runtime dependencies
# ${shlibs:Depends}... are debhelper macros that
automatically adds a bunch of info/deps... in our case it will
detect dependency on libtraining and add it
Depends: ${shlibs:Depends}, ${misc:Depends}
Description: bla
 longer bla
```

```
# Contains .so.x.y lib file

Package: libtraining
Architecture: any
Section: libs
Depends: ${shlibs:Depends}, ${misc:Depends}
Description: bla
 longer bla
```

```
# Contains .h and .so symlink

Package: libtraining-dev
Section: libdevel
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends},
libtraining

Description: bla

 longer bla
```

Rules

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

# Rules file contains all build rules from configure/make/make test/make install etc...
# All commands are indented with a tabulation (no whitespace)
# All commands are executed relative to source code directory
# Warning: all commands should return exit code 0

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

%:
	# --with xx , debhelper function, here we force an autoreconf, other helpers are available:
	# --with javahelper (associated with debian/package.jlibs), --with python2
	dh $@ --with autoreconf
```

Rules: custom instructions

Sometimes, the build helpers are not able to do the job automatically (makefile in subdirectories for example), or we need to put specific file manipulations.
It is possible to override the default behavior of the macro with the override_xx function (cf. http://www.man-linux-magique.net/man7/debhelper.html)

```
%:
	# --with xx , debhelper function, here we force an autoreconf, other helpers are available:
	# --with javahelper (associated with debian/package.jlibs), --with python2
	dh $@ --with autoreconf


# Override dh_clean to remove some generated files, not cleaned by traditional make clean command
override_dh_clean:
	# optionally execute the default command
	dh_clean
	# Additional commands
	rm -f a_generated_file_that_should_not_be_kept
	cd some_dir && make clean
```

Copyright

Copyright is not “needed” for build, but required for a package.
It contains all copyright/license information for the software.
It is possible to specify different licenses per file/directory. You should check all embedded files license before distributing your package (or code)

Where should files go?

In multi package configuration, we need to tell the package system which file/dir goes in which package. For single package, default install should work if build system is correctly set (though may need adaptation to debian filesystem hierarchy)
=> debian/mybinarypackage.install

https://wiki.debian.org/FilesystemHierarchyStandard

Install files

```
# This file contains a list of files / dir specifying the “source” data, and their location in the binary package
# In debian/training.install we would add:
debian/tmp/usr/bin/squizz usr/bin/
debian/tmp/usr/share/man/man1/* usr/share/man/man1/
```

Origin path is relative to the source directory. We can refer to debian/tmp/… to refer to a file installed with (for example) a make install, or from any file in original source directory (doc/mydoc.pdf) for example.

Path MUST not start with a “/”

Install files

```
# libtraining.install
# install versioned shared library
debian/tmp/usr/lib/*/lib*.so.*  usr/lib/

# libtraining-dev.install
# install header, static lib and .so symlink
debian/tmp/usr/lib/*/*.a usr/lib/
debian/tmp/usr/include/* usr/include
debian/tmp/usr/lib/*/*.so  usr/lib/
```

Of course the content of each package will be package dependent. Debian policy gives a lot of recommendations. If you create a package within your own repository, you will have less contraints but it is highly recommended to follow them.

Other files

```
# training.manpages
# will automatically install some man pages to the man directories
doc/squizz.man

# docs
#any README, LICENSE, ... file to add to the packages, will be compress if too large

# training.examples
# path to example files to include in package

# training.links
# creates some symlinks, even if original file is not (yet) present
# Example:
# usr/share/java/somelib.jar usr/share/training/libs/somelib.jar
```

{post,pre}{inst,rm}

postinst, postrm, preinst, prerm are scripts executed at package installation/removal. They should be used with care to avoid errors. They can be used for specific upgrades or checks. Here is a preinst file example to check package version.

```
#!/bin/sh
set -e
case "$1" in
    install)
    ;;
    upgrade)
        if [ $2 = "1.1.0-1" ]; then
         echo "There is an issue in version 1.1.0-1, you need to first remove the package, then to install it. "
         exit 1
        fi
    ;;
    abort-upgrade)
    ;;
    *)
        echo "preinst called with unknown argument \`$1'" >&2
        exit 1
    ;;
esac
```

postrm example

```
#!/bin/sh
set -e
# Source debconf library.
. /usr/share/debconf/confmodule

case "$1" in
   purge)
   # Remove generated data for complete purge of data
   if [ -e /var/lib/training ]; then
     rm -rf /var/lib/training
   fi
   ;;
   remove|upgrade|failed-upgrade|abort-install|disappear)
   echo "Warning, database for MySQL is not deleted by process removal."    ;;
   abort-upgrade)
   ;;
   *)
      echo "postrm called with unknown argument \`$1'" >&2
      exit 1
   ;;
esac
#DEBHELPER#
```

Let’s build again

vagrant@vagrant-ubuntu-trusty-64:~/training-1.0$dpkg-buildpackage -rfakeroot
....
vagrant@vagrant-ubuntu-trusty-64:~/training-1.0$ dpkg --contents ../training_1.0-1_amd64.deb

Now your package should contain all binaries, libs, … and is ready for install

Lintian - check your package

lintian -I –pedantic *.deb

Warnings, errors etc…. will appear, they SHOULD be fixed (MUST to be put in official repo)

pbuilder

pbuilder helps to try to build of the package in a clean/fresh system to check there is no missing dependency, config issues.

```
## Create an archive for later builds
$sudo pbuilder create
$sudo pbuilder build mypackage.dsc

## To update the archive (get latest packages...)
$sudo pbuilder update
```

piuparts

Piuparts checks the install/reinstall/clean removal of the package

https://piuparts.debian.org/doc/README_1st.html

Test install/removal

Always test your package!

```
$ dpkg -i libtraining_1.0-1_amd64.deb
$ dpkg -i training_1.0-1_amd64.deb
$ apt-get remove libtraining
```

Patches

Patches are sometimes needed to adapt to the Debian directories layout, ….

Solution is *quilt*

Quilt

In your source directory:

```
$quilt new mypatchname
$quilt add thefileiwantotopatch
## edit the file
$quilt refresh
## Patches are created/updates in debian/patches
```

To unapply patches: quilt pop -a

To apply all patches: quilt push -a

Using quilt

$git clone https://github.com/osallou/training-packaging-2.git trainingwitherrors-1.0
$ tar cvfz trainingwitherrors_1.0.orig.tar.gz trainingwitherrors-1.0

Copy the debian directory of training-1.0 in trainingwitherrors-1.0
Update debian/control Source parameter and debian/changelog to the new package name (training => trainingwitherrors)

$dpkg-buildpackage -rfakeroot

Package build will fail, now let's patch it using previous slide.
Error:
squizz.c:29:1: error: unknown type name 'ERRORint'
 ERRORint main(int argc, char **argv) {

using quilt

Patch src/squizz.c, line 29:
=> Patch “ERRORint” to be “int”

After quilt refresh, have a look at debian/patches directory and files

Once patches are done, rebuild!

Solution

You can get the full debian dir files of the provided example at http://anonscm.debian.org/viewvc/debian-med/trunk/packages/squizz/trunk/debian/