dh-virtualenv Documentation

Release 1.2

Spotify AB

2019-03-08

Contents

CHAPTER 1

Overview

`dh-virtualenv` is a tool that aims to combine Debian packaging with self-contained Python software deployment in a pre-built virtualenv. To do this, the project extends debhelper's build sequence by providing the new `dh_virtualenv` command.

This new command effectively replaces the following commands in the default sequence:

- `dh_auto_install`
- `dh_python2`
- `dh_pycentral`
- `dh_pysupport`

In the debhelper build sequence, `dh_virtualenv` is inserted right after `dh_perl`.

Reading Guide

1. *Getting Started* helps you to set up your build machine and then package your first simple project.
2. *Packaging Guide* explains all available features in more detail.
3. The *Packaging Cookbook* demonstrates specific features and tricks needed for packaging more challenging projects.
4. The *Trouble-Shooting Guide* explains some typical errors you might enounter, and their solution.
5. To take a look into complete projects, see *Real-World Projects Show-Case*.
6. *API / Code Reference* has a short overview of the implementation and links to the source code.
7. Finally, the *Changelog* provides a history of releases with their new features and fixes.

CHAPTER 2

Contents of this Manual

2.1 Getting Started

This tutorial will guide you through setting up your first project using *dh-virtualenv*. Having some knowledge on how Debian packages work might help, but it is not a mandatory requirement when working on simple projects.

You also need some basic build tools, so you should install these packages:

```
sudo apt-get install build-essential debhelper devscripts equivs
```

These are only required on the *build host*, not the *target hosts* you later install the built packages on.

If you perform your *package builds in a Docker container*, you can also skip installing these tools, because then only `docker-ce` is needed.

2.1.1 Step 1: Install dh-virtualenv

Overview

In order to use it, you need to install *dh-virtualenv* as a debhelper add-on on the build host. For Debian and Ubuntu, there are pre-built packages for the 1.0 version available – note that some of this info might get outdated over time, so take extra care to check the version numbers you're actually getting vs. what features you need.

The following paragraphs describe the various installation options, including building from source when your specific environment provides no packages or only older versions.

Using pre-1.1 versions is possible, but you don't get all features described in this document, and not all projects using dh-virtualenv might work with older versions (check their documentation).

Package installation from OS repositories

On Debian *Stretch* (stable) it is a simple `apt install dh-virtualenv` to get v1.0 installed. To install on *Jessie* (oldstable) from their package repositories, use these commands:

```
echo "deb http://ftp.debian.org/debian jessie-backports main" \
    | sudo tee /etc/apt/sources.list.d/jessie-backports.list >/dev/null
sudo apt-get update -qq
sudo apt-get install -t jessie-backports dh-virtualenv
```

Note that `jessie-backports` also only offers the 1.0 version.

Another option to check out for *Ubuntu* is this PPA, maintained by the author.

It is also possible to get newer versions from Debian testing (sid) or recent releases in the official Ubuntu repositories. Since dh-virtualenv has the `all` architecture (contains no native code), that is generally possible, but you might need to take extra care of dependencies. The recommendation is to only follow that route in *Docker container builds*, where manipulating dependencies has no lasting effect – don't do that on your workstation.

Build your own package

For all other platforms you have to build and install the tool yourself. The easiest way (since v1.1) is to build the package using Docker with the `invoke bdist_deb` command in a boot-strapped working directory, see the README for details on that. Using Docker also allows cross-distribution builds.

Otherwise, after you have cloned the repository, you must install build tools and dependencies on your worksatation, and then start the build:

```
# Install needed packages
sudo apt-get install devscripts python-virtualenv python-sphinx \
                     python-sphinx-rtd-theme git equivs
# Clone git repository
git clone https://github.com/spotify/dh-virtualenv.git
# Change into working directory
cd dh-virtualenv
# This will install build dependencies
sudo mk-build-deps -ri
# Build the *dh-virtualenv* package
dpkg-buildpackage -us -uc -b

# And finally, install it (you might have to solve some
# dependencies when doing this)
sudo dpkg -i ../dh-virtualenv_<version>.deb
```

2.1.2 Step 2: Set up packaging for your project

Grab your favourite Python project you want to use *dh-virtualenv* with and set it up. Only requirement is that your project has a somewhat sane `setup.py` and requirements listed in a `requirements.txt` file. Note however that defining any requirements is not mandatory, if you have none.

Instead of following all the steps outlined below, you can use cookiecutters (project templates) to quickly create the needed information in the `debian/` directory for any existing project.

- dh-virtualenv-mold is a cookiecutter template to add easy Debianization to any existing Python project.

- debianized-pypi-mold does the same for 3rd party software released to PyPI which you want to package for deployment.

See the related READMEs for details.

For the manual way, start with defining the Debian packaging metadata for your software. To do this, create a directory called `debian` in the project root.

To be able to build a debian package, a few files are needed. First, we need to define the compatibility level of the project. For this, do:

```
echo "9" > debian/compat
```

The 9 is a magic number for latest compatibility level, but we don't need to worry about that. Next we need a file that tells what our project is about, a file called `control`. Create a `debian/control` file similar to the following:

```
Source: my-awesome-python-software
Section: python
Priority: extra
Maintainer: Matt Maintainer <matt@example.com>
Build-Depends: debhelper (>= 9), python, dh-virtualenv (>= 0.8)
Standards-Version: 3.9.5

Package: my-awesome-python-software
Architecture: any
Pre-Depends: dpkg (>= 1.16.1), python2.7 | python2.6, ${misc:Pre-Depends}
Depends: ${misc:Depends}
Description: A short summary of what this is.
    Further indented lines can contain extra information.
    .
    A single dot separates paragraphs.
```

The `control` file is used to define the build dependencies, so if you are building a package that requires for example `lxml`, make sure you define `libxml2-dev` in *Build-Depends*.

Depends in the 2nd section is used to define run-time dependencies. The *debhelper* magic will usually take care of that via the `${misc:Depends}` you see above.

To help keeping your installed virtualenv in sync with the host's Python interpreter in case of updates, create a file named `debian/«pkgname».triggers`, where «pkgname» is what you named your package in the `control` file. It triggers a special script whenever the Python binary changes; don't worry, that script is provided by `dh-virtualenv` automatically.

«pkgname».triggers

```
# Register interest in Python interpreter changes (Python 2 for now); and
# don't make the Python package dependent on the virtualenv package
# processing (noawait)
interest-noawait /usr/bin/python2.6
interest-noawait /usr/bin/python2.7

# Also provide a symbolic trigger for all dh-virtualenv packages
interest dh-virtualenv-interpreter-update
```

That file *must* end with a new-line – if your editor is misconfigured to eat the end of the last line in a file, you better fix that.

Note that if you provide a custom `postinst` script with your package, then don't forget to put the `#DEBHELPER#` marker into it, else the trigger script will be missing. The same applies to other maintainer scripts.

Next, we need a changelog file. It is basically a documentation of changes in your package plus the source for version number for Debian package builder. Here's a short sample changelog to be entered in `debian/changelog`:

```
my-awesome-python-software (0.1-1) unstable; urgency=low

  * Initial public release

 -- Matt Maintainer <matt@example.com>  Fri, 01 Nov 2013 17:00:00 +0200
```

You don't need to create this file by hand, a handy tool called `dch` exists for entering new changelog entries.

Now, the last bit left is adding the `debian/rules` file. This file is usually an executable *Makefile* that Debian uses to build the package. The content for that is fairly simple:

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

%:
	dh $@ --with python-virtualenv
```

And there we go, debianization of your new package is ready!

Tip: Do not forget to `git add` the `debian/` directory *before* you build for the first time, because generated files will be added there that you don't want in your source code repository.

2.1.3 Step 3: Build your project

Now you can just build your project by running `( deactivate ; dpkg-buildpackage -us -uc -b )`. Enjoy your newly baked *dh-virtualenv* backed project!

2.2 Packaging Guide

Building packages with *dh-virtualenv* is relatively easy to start with, but it also supports lot of customization to match your specific needs.

By default, *dh-virtualenv* installs your packages under `/opt/venvs/«packagename»`. The package name is provided by the `debian/control` file.

To use an alternative install prefix, add a line like the following to the top of your `debian/rules` file.

```
export DH_VIRTUALENV_INSTALL_ROOT=«/your/custom/install/dir»
```

dh_virtualenv will now use the value of *DH_VIRTUALENV_INSTALL_ROOT* instead of `/opt/venvs` when it constructs the install path.

To use an install suffix other than the package name, call `dh_virtualenv` using the *--install-suffix* command line option. See *Advanced usage* for further information on passing options.

2.2.1 Simple usecase

To signal debhelper to use *dh-virtualenv* for building your package, you need to pass `--with python-virtualenv` to the debhelper sequencer.

In a nutshell, the simplest `debian/rules` file to build using *dh-virtualenv* looks like this:

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

%:
        dh $@ --with python-virtualenv
```

However, the tool makes a few assumptions of your project's structure:

- For installing requirements, you need to have a file called `requirements.txt` in the root directory of your project. The requirements file is not mandatory.
- The project must have a `setup.py` file in the root of the project. `dh_virtualenv` will run `setup.py install` to add your project to the virtualenv.

After these preparations, you can just build the package with your favorite tool!

2.2.2 Environment variables

Certain environment variables can be used to customise the behaviour of the debhelper sequencer in addition to the standard debhelper variables.

DH_VIRTUALENV_INSTALL_ROOT
: Define a custom root location to install your package(s). The resulting location for a specific package will be `$DH_VIRTUALENV_INSTALL_ROOT/«<packagename»`, unless *`--install-suffix`* is also used to change «<packagename».

2.2.3 Command line options

To change its default behavior, the `dh_virtualenv` command accepts a few command line options:

-p <package>**, --package** <package>
: Act on the package named *<package>*.

-N <package>**, --no-package** <package>
: Do not act on the specified package.

-v, --verbose
: Turn on verbose mode. This has a few effects: it sets the root logger level to `DEBUG`, and passes the verbose flag to `pip` when installing packages. This can also be provided using the standard `DH_VERBOSE` environment variable.

--install-suffix <suffix>
: Override virtualenv installation suffix. The suffix is appended to `/opt/venvs`, or the *`DH_VIRTUALENV_INSTALL_ROOT`* environment variable if specified, to construct the installation path.

--extra-index-url <url>
: Use extra index url *<url>* when running `pip` to install packages. This can be provided multiple times to pass multiple URLs to `pip`. A common use-case is enabling a private Python package repository.

--preinstall <package>
Package to install before processing the requirements. This flag can be used to provide a package that is installed by `pip` before processing the requirements file. It is handy if you need to install a custom setup script or other packages needed to parse `setup.py`, and can be provided multiple times to pass multiple packages for pre-install.

--extras <name>
New in version 1.1.

Name of extras defined in the main package (specifically its `setup.py`, in `extras_require`). You can pass this multiple times to add different extra requirements.

--pip-tool <exename>
Executable that will be used to install requirements after the preinstall stage. Usually you'll install this program by using the `--preinstall` argument. The replacement is expected to be found in the virtualenv's `bin/` directory.

--upgrade-pip
New in version 1.0.

Force upgrading to the latest available release of `pip`. This is the first thing done in the pre-install stage, and uses a separate `pip` call.

Options provided via *--extra-pip-arg* are ignored here, because the default `pip` of your system might not support them (since version 1.1).

Note: This can produce non-repeatable builds. See also *--upgrade-pip-to*.

--upgrade-pip-to <VERSION>
New in version 1.2.

Same as *--upgrade-pip*, but install an explicitly provided version. You can specify `latest` to get the exact same behaviour as with the simple option.

Note: This can be used for more repeatable builds that do not have the risk of breaking on a new `pip` release.

--index-url <URL>
Base URL of the PyPI server. This flag can be used to pass in a custom URL to a PyPI mirror. It's useful if you have an internal PyPI mirror, or you run a special instance that only exposes selected packages of PyPI. If this is not provided, the default will be whatever `pip` uses as default (usually the API of `https://pypi.org/`).

--extra-pip-arg <PIP ARG>
Extra arguments to pass to the pip executable. This is useful if you need to change the behaviour of pip during the packaging process. You can use this flag multiple times to pass in different pip flags.

As an example, adding `--extra-pip-arg --no-compile` in the call of a `override_dh_virtualenv` rule in the `debian/rules` file will disable the generation of `*.pyc` files.

--extra-virtualenv-arg <VIRTUALENV ARG>
Extra parameters to pass to the virtualenv executable. This is useful if you need to change the behaviour of virtualenv during the packaging process. You can use this flag multiple times to pass in different virtualenv flags.

--requirements <REQUIREMENTS FILE>
Use a different requirements file when installing. Some packages such as pbr expect the `requirements.txt` file to be a simple list of requirements that can be copied verbatim into the `install_requires` list. This command option allows specifying a different `requirements.txt` file that may include pip specific flags such as `-i`, `-r-` and `-e`.

--setuptools
Use setuptools instead of distribute in the virtualenv.

--setuptools-test
New in version 1.0.

Run `python setup.py test` when building the package. This was the old default behaviour before version 1.0. This option is incompatible with the deprecated *--no-test*.

--python <path>
Use a specific Python interpreter found in `path` as the interpreter for the virtualenv. Default is to use the system default, usually `/usr/bin/python`.

--builtin-venv
Enable the use of the build-in `venv` module, i.e. use `python -m venv` to create the virtualenv. It will only work with Python 3.4 or later, e.g. by using the option *--python* `/usr/bin/python3.4`.

-S, --use-system-packages
Enable the use of system site-packages in the created virtualenv by passing the `--system-site-packages` flag to `virtualenv`.

--skip-install
Skip running `pip install .` after dependencies have been installed. This will result in anything specified in `setup.py` being ignored. If this package is intended to install a virtualenv and a program that uses the supplied virtualenv, it is up to the user to ensure that if `setup.py` exists, any installation logic or dependencies contained therein are handled.

This option is useful for web application deployments, where the package's virtual environment merely supports an application installed via other means. Typically, the `debian/«packagename».install` file is used to place the application at a location outside of the virtual environment.

--pypi-url <URL>
Deprecated since version 1.0: Use *--index-url* instead.

--no-test
Deprecated since version 1.0: This option has no effect. See *--setuptools-test*.

2.2.4 Advanced usage

To provide command line options to the `dh_virtualenv` step, use debhelper's override mechanism.

The following `debian/rules` will provide *http://example.com* as an additional source of Python packages:

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

%:
        dh $@ --with python-virtualenv

override_dh_virtualenv:
        dh_virtualenv --extra-index-url http://example.com
```

2.2.5 pbuilder and dh-virtualenv

Building your Debian package in a pbuilder environment can help to ensure proper dependencies and repeatable builds. However, precisely because pbuilder creates its own build environment, build failues can be much more difficult to understand and troubleshoot. This is especially true when there is a pip error inside the pbuilder environment. For that reason, make sure that you can build your Debian package successfully outside of a pbuilder environment before trying to build it inside.

With those caveats, here are some tips for making pip and dh_virtual work inside pbuilder.

If you want pip to retrieve packages from the network, you need to add `USENETWORK=yes` to your /etc/pbuilderrc or ~/.pbuilderrc file.

pip has several options that can be used to make it more compatible with pbuilder.

Use `--no-cache-dir` to stop creating wheels in your home directory, which will fail when running in a pbuilder environment, because pbuilder sets the HOME environment variable to "/nonexistent".

Use `--no-deps` to make pip builds more repeatable.

Use `--ignore-installed` to ensure that pip installs every package in `requirements.txt` in the virtualenv. This option is especially important if you are using the –system-site-packages option in your virtualenv.

Here's an example of how to use these arguments in your `rules` file.

```
override_dh_virtualenv:
        dh_virtualenv \
        --extra-pip-arg "--ignore-installed" \
        --extra-pip-arg "--no-deps" \
        --extra-pip-arg "--no-cache-dir"
```

2.2.6 Experimental buildsystem support

Important: This section describes a completely experimental functionality of dh-virtualenv.

Starting with version 0.9 of dh-virtualenv, there is a buildsystem alternative. The main difference in use is that instead of the `--with python-virtualenv` option, `--buildsystem=dh_virtualenv` is passed to debhelper. The `debian rules` file should look like this:

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

%:
        dh $@ --buildsystem=dh_virtualenv
```

Using the buildsystem instead of the part of the sequence (in other words, instead of the `--with python-virtualenv`) one can get more flexibility into the build process.

Flexibility comes from the fact that buildsystem will have individual steps for configure, build, test and install and those can be overridden by adding `override_dh_auto_<STEP>` target into the `debian/rules` file. For example:

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

%:
        dh $@ --buildsystem=dh_virtualenv

override_dh_auto_test:
        py.test test/
```

In addition the separation of build and install steps makes it possible to use `debian/install` files to include built files into the Debian package. This is not possible with the sequencer addition.

The build system honors the *DH_VIRTUALENV_INSTALL_ROOT* environment variable. Following other environment variables can be used to customise the functionality:

DH_VIRTUALENV_ARGUMENTS
: Pass given extra arguments to the `virtualenv` command

  For example:

```
export DH_VIRTUALENV_ARGUMENTS="--no-site-packages --always-copy"
```

The default is to create the virtual environment with `--no-site-packages`.

DH_VIRTUALENV_INSTALL_SUFFIX
Override the default virtualenv name, instead of source package name.

For example:

DH_REQUIREMENTS_FILE
New in version 1.0.

Override the location of requirements file. See *--requirements*.

DH_UPGRADE_PIP
New in version 1.0.

Force upgrade of the `pip` tool by setting *DH_UPGRADE_PIP* to empty (latest version) or specific version. For example:

DH_UPGRADE_SETUPTOOLS
New in version 1.0.

Force upgrade of setuptools by setting *DH_UPGRADE_SETUPTOOLS* to empty (latest version) or specific version.

DH_UPGRADE_WHEEL
New in version 1.0.

Force upgrade of wheel by setting `DH_UPGRADE_WHEEL` to empty (latest version) or specific version.

DH_PIP_EXTRA_ARGS
New in version 1.0.

Pass additional parameters to the `pip` command. For example:

```
export DH_PIP_EXTRA_ARGS="--no-index --find-links=./requirements/wheels"
```

2.3 Packaging Cookbook

This chapter has recipes and tips for specific dh-virtualenv use-cases, like proper handling of binary `manylinux1` wheels. It also demonstrates some Debian packaging and debhelper features that are useful in the context of Python software packaging.

List of Recipes

2.3.1 Building Packages for Python3

The Python2 EOL in 2020 is not so far away, so you better start to use Python3 for new projects, and port old ones that you expect to survive until then. The following is for *Ubuntu Xenial* or *Debian Stretch* with Python 3.5, and on *Ubuntu Bionic* you get Python 3.6.

In `debian/control`, the `Build-Depends` and `Pre-Depends` lists have to refer to Python3 packages.

```
Source: py3software
Section: contrib/python
...
Build-Depends: debhelper (>= 9), python3, dh-virtualenv (>= 1.0),
    python3-setuptools, python3-pip, python3-dev, libffi-dev
...

Package: py3software
...
Pre-Depends: dpkg (>= 1.16.1), python3 (>= 3.5), ${misc:Pre-Depends}
```

And the Python update triggers in `debian/«pkgname».triggers` need to be adapted, too.

```
...
interest-noawait /usr/bin/python3.5
...
```

You may also need to add the *`--python`* option in `debian/rules`.

```
%:
    dh $@ --with python-virtualenv

override_dh_virtualenv:
    dh_virtualenv --python python3
```

If you're using the buildsystem alternative, it is instead specified through the *`DH_VIRTUALENV_ARGUMENTS`* variable.

```
export DH_VIRTUALENV_ARGUMENTS := --no-site-packages --python python3

%:
    dh $@ --buildsystem dh_virtualenv
```

2.3.2 Making executables available

To make executables in your virtualenv's `bin` directory callable from any shell prompt, do **not** add that directory to the global `PATH` by a `profile.d` hook or similar. This would add all the other stuff in there too, and you simply do not want that.

So use the `debian/«pkgname».links` file to add a symbolic link to *those* exectuables you want to be visible, typically the one created by your main application package.

```
opt/venvs/«venvname»/bin/«cmdname» usr/bin/«cmdname»
```

Replace the contained «`placeholders`» with the correct names. Add more links if there are additional tools, one line per extra executable. For `root`-only commands, use `usr/sbin/...`.

2.3.3 Handling binary wheels

The introduction of manylinux wheels via PEP 513 is a gift, sent by the PyPA community to us lowly developers wanting to use packages like Numpy while *not* installing a Fortran compiler just for that.

However, two steps during package building often clash with the contained shared libraries, namely *stripping* (reducing the size of symbol tables) and scraping package dependencies out of shared libraries (*shlibdeps*).

So if you get errors thrown at you by either `dh_strip` or `dh_shlibdeps`, extend your `debian/rules` file as outlined below.

```
.PHONY: override_dh_strip override_dh_shlibdeps

override_dh_strip:
        dh_strip --exclude=cffi

override_dh_shlibdeps:
        dh_shlibdeps -X/x86/ -X/numpy/.libs -X/scipy/.libs -X/matplotlib/.libs
```

This example works for the Python data science stack – you have to list the packages that cause *you* trouble.

2.3.4 Adding Node.js to your virtualenv

There are polyglot projects with a mix of Python and Javascript code, and some of the JS code might be executed server-side in a Node.js runtime. A typical example is server-side rendering for Angular apps with Angular Universal.

If you have this requirement, there is a useful helper named `nodeenv`, which extends a Python virtualenv to also support installation of NPM packages.

The following changes in `debian/control` require *Node.js* to be available on both the build and the target hosts. As written, the current LTS version is selected (i.e. *8.x* in mid 2018). The NodeSource packages are recommended to provide that dependency.

```
...
Build-Depends: debhelper (>= 9), python3, dh-virtualenv (>= 1.0),
    python3-setuptools, python3-pip, python3-dev, libffi-dev,
    nodejs (>= 8), nodejs (<< 9)
...
Depends: ${shlibs:Depends}, ${misc:Depends}, nodejs (>= 8), nodejs (<< 9)
...
```

You also need to extend `debian/rules` as follows, change the variables in the first section to define different versions and filesystem locations.

```
export DH_VIRTUALENV_INSTALL_ROOT=/opt/venvs
SNAKE=/usr/bin/python3
EXTRA_REQUIREMENTS=--upgrade-pip --preinstall "setuptools>=17.1" --preinstall "wheel"
NODEENV_VERSION=1.3.1

PACKAGE=$(shell dh_listpackages)
DH_VENV_ARGS=--setuptools --python $(SNAKE) $(EXTRA_REQUIREMENTS)
DH_VENV_DIR=debian/$(PACKAGE)$(DH_VIRTUALENV_INSTALL_ROOT)/$(PACKAGE)

ifeq (,$(wildcard $(CURDIR)/.npmrc))
    NPM_CONFIG=~/.npmrc
else
    NPM_CONFIG=$(CURDIR)/.npmrc
```

(continues on next page)

(continued from previous page)

```
endif


%:
        dh $@ --with python-virtualenv $(DH_VENV_ARGS)

.PHONY: override_dh_virtualenv

override_dh_virtualenv:
        dh_virtualenv $(DH_VENV_ARGS)
        $(DH_VENV_DIR)/bin/python $(DH_VENV_DIR)/bin/pip install nodeenv==$(NODEENV_
↪VERSION)
        $(DH_VENV_DIR)/bin/nodeenv -C '' -p -n system
        . $(DH_VENV_DIR)/bin/activate \
            && node /usr/bin/npm install --userconfig=$(NPM_CONFIG) \
                    -g configurable-http-proxy
```

You want to always copy all but the last line literally. The lines above it install and embed `nodeenv` into the virtualenv freshly created by the `dh_virtualenv` call. Also remember to use TABs in makefiles (`debian/rules` is one).

The last (logical) line globally installs the `configurable-http-proxy` NPM package – one important result of using `-g` is that Javascript commands appear in the `bin` directory just like Python ones. That in turn means that in the activated virtualenv Python can easily call those JS commands, because they're on the `PATH`.

Change the NPM package name to what you want to install. `npm` uses either a local `.npmrc` file in the project root, or else the `~/.npmrc` one. Add local repository URLs and credentials to one of these files.

2.3.5 Multi-platform builds in Docker

The code shown here is taken from the *debianized-jupyterhub* project, and explains how to build a package in a Docker container.

Why build a package in a container? This is why:

- *repeatable* builds in a *clean* environment
- explicitly *documented installation* of build requirements *(as code)*
- easy *multi-distro multi-release builds*

The build is driven by a small shell script named `build.sh`, which we use to get the target platform and some project metadata we already have, and feed that into the Dockerfile via simple `sed` templating.

So we work on a copy of the Dockerfile, and that is one reason for anything in the project workdir that is controlled by git being copied to a staging area (a separate build directory). The other reason is performance – we present Docker with a pristine copy of our workdir, and so there are no accidents like `COPY`ing a full development virtualenv or all of `.git` into the container build.

The build script

Let's get to the code – since we apply the *Adding Node.js to your virtualenv* recipe, we first set the repository where to get Node.js from.

```
#! /usr/bin/env bash
#
# Build Debian package in a Docker container
```

(continues on next page)

(continued from previous page)

```
#

set -e

NODEREPO="node_8.x"
```

Next, the given platform and existing project metadata is stored into a bunch of variables.

```
# Get build platform as 1st argument, and collect project metadata
image="${1:?You MUST provide a docker image name}"; shift
dist_id=${image%%:*}
codename=${image#*:}
pypi_name="$(./setup.py --name)"
pkgname="$(dh_listpackages)"
tag=$pypi_name-$dist_id-$codename
staging_dir="build/staging"
```

Based on the collected input parameters, the staging area is set up in the `build/staging` directory. `tar` does the selective copy work, and `sed` is used to inject dynamic values into the copied files.

```
# Prepare staging area
rm -rf $staging_dir 2>/dev/null || true
mkdir -p $staging_dir
git ls-files >build/git-files
test ! -f .npmrc || echo .npmrc >>build/git-files
tar -c --files-from build/git-files | tar -C $staging_dir -x
sed -i -r -e 1s/stretch/$codename/g $staging_dir/debian/changelog
sed -r -e s/#UUID#/$(< /proc/sys/kernel/random/uuid)/g \
    -e s/#DIST_ID#/$dist_id/g -e s/#CODENAME#/$codename/g \
    -e s/#NODEREPO#/$NODEREPO/g -e s/#PYPI#/$pypi_name/g -e s/#PKGNAME#/$pkgname/g \
    <Dockerfile.build >$staging_dir/Dockerfile
```

After all that prep work, we finally get to build our package. The results are copied from `/dpkg` where the Dockerfile put them (see below), and then the package metadata is shown for a quick visual check if everything looks OK.

```
# Build in Docker container, save results, and show package info
docker build --tag $tag "$@" $staging_dir
docker run --rm $tag tar -C /dpkg -c . | tar -C build -xv
dpkg-deb -I build/${pkgname}_*~${codename}*.deb
```

The Dockerfile

This is the complete Dockerfile, the important things are the two `RUN` directives.

```
# Build Debian package using dh-virtualenv

FROM #DIST_ID#:#CODENAME# AS dpkg-build
ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get update -qq && apt-get install -yqq \
        build-essential debhelper devscripts equivs dh-virtualenv \
        curl tar gzip lsb-release apt-utils apt-transport-https libparse-
↪debianchangelog-perl \
        python3 python3-setuptools python3-pip python3-dev libffi-dev \
        libxml2-dev libxslt1-dev libyaml-dev libjpeg-dev \
```

(continues on next page)

(continued from previous page)

```
        libssl-dev libncurses5-dev libncursesw5-dev libzmq3-dev \
    && ( curl -s https://deb.nodesource.com/gpgkey/nodesource.gpg.key | apt-key add -␣
↪) \
    && echo 'deb https://deb.nodesource.com/#NODEREPO# #CODENAME# main' \
            >/etc/apt/sources.list.d/nodesource.list \
    && apt-get update -qq && apt-get install -y nodejs \
    && rm -rf "/var/lib/apt/lists"/*
WORKDIR /dpkg-build
COPY ./ ./
RUN dpkg-buildpackage -us -uc -b && mkdir -p /dpkg && cp -pl /#PKGNAME#[-_]* /dpkg
# RUN pwd && dh_virtualenv --version && ls -la && du -sch . ##UUID#
```

The first `RUN` installs all the build dependencies on top of the base image. The second one then builds the package and makes a copy of the resulting files, for the build script to pick them up.

Putting it all together

Here's a sample run of building for *Ubuntu Bionic*.

```
$ ./build.sh ubuntu:bionic
Sending build context to Docker daemon    106kB
Step 1/6 : FROM ubuntu:bionic AS dpkg-build
...
Successfully tagged debianized-jupyterhub-ubuntu-bionic:latest
./
./jupyterhub_0.9.1-1~bionic_amd64.deb
./jupyterhub_0.9.1-1~bionic_amd64.buildinfo
./jupyterhub-dbgsym_0.9.1-1~bionic_amd64.ddeb
./jupyterhub_0.9.1-1~bionic_amd64.changes
 new debian package, version 2.0.
 size 265372284 bytes: control archive=390780 bytes.
      84 bytes,     3 lines      conffiles
    1214 bytes,    25 lines      control
 2350661 bytes, 17055 lines      md5sums
    4369 bytes,   141 lines   *  postinst             #!/bin/sh
    1412 bytes,    47 lines   *  postrm               #!/bin/sh
     696 bytes,    35 lines   *  preinst              #!/bin/sh
    1047 bytes,    41 lines   *  prerm                #!/bin/sh
     217 bytes,     6 lines      shlibs
     419 bytes,    10 lines      triggers
 Package: jupyterhub
 Version: 0.9.1-1~bionic
 Architecture: amd64
 Maintainer: 1&1 Group <jh@web.de>
 Installed-Size: 563574
 Pre-Depends: dpkg (>= 1.16.1), python3 (>= 3.5)
 Depends: perl:any, libc6 (>= 2.25), libexpat1 (>= 2.1~beta3), libgcc1 (>= 1:4.0), ...
 Suggests: oracle-java8-jre | openjdk-8-jre | zulu-8
 Section: contrib/python
 Priority: extra
 Homepage: https://github.com/1and1/debianized-jupyterhub
 Description: Debian packaging of JupyterHub, a multi-user server for Jupyter␣
↪notebooks.
...
```

The package files are now in `build/`, and you can `dput` them into your local repository.

2.3.6 Cross-packaging for ARM targets

If you need to create packages that can be installed on ARM architectures, but want to use any build host (e.g. a CI worker), first install the `qemu-user-static` and `binfmt-support` packages.

Then build the package by starting a container in QEMU using this `Dockerfile`.

```
FROM arm32v7/debian:latest

RUN apt-get update && apt-get -y upgrade && apt-get update \
    && apt-get -y install sudo dpkg-dev debhelper dh-virtualenv python3 python3-venv
...
```

The build might fail from time to time, due to unknown causes (maybe instabilities in QEMU). If you get a package out of it, that works 100% fine, however.

See *configsite* for the full project that uses this.

— with input from @Nadav-Ruskin

2.4 Trouble-Shooting Guide

2.4.1 Installing on older Debian releases

TODO

2.4.2 Fixing package building problems

'pkg-resources not found' or similar

If you get errors regarding `pkg-resources` during the virtualenv creation, update your build machine's `pip` and `virtualenv`. The versions on previous releases of many distros are just too old to handle current infrastructure (especially PyPI) – even Debian Jessie comes with the ancient pip 1.5.6.

This is the one exception to "never sudo pip", so go ahead and do this:

```
sudo pip install -U pip virtualenv
```

Then try building the package again.

2.4.3 Fixing package installation problems

dpkg: too-long line or missing newline in '.../triggers'

TODO https://github.com/spotify/dh-virtualenv/pull/84

2.5 Real-World Projects Show-Case

These complete projects show how to combine the features of dh-virtualenv and Debian packaging in general to deliver actual software in the wild. You'll also see some of the recipes of the *Packaging Cookbook* applied in a wider context.

List of Projects

- *debianized-sentry*
- *debianized-jupyterhub*
- *configsite*

2.5.1 debianized-sentry

Author Jürgen Hermann

URL https://github.com/1and1/debianized-sentry

The project packages *Sentry.io*, adding systemd integration and default configuration for the Sentry Django/uWSGI app and related helper services. It also shows how to package 3rd party software as relased on PyPI, keeping the packaging code separate from the packaged project.

It is based on the debianized-pypi-mold cookiecutter, which allows you to set up such projects *from scratch* to the first build in typically under an hour.

2.5.2 debianized-jupyterhub

Author Jürgen Hermann

URL https://github.com/1and1/debianized-jupyterhub

JupyterHub has a Node.js service that implements its *configurable HTTP proxy* component, so this project applies the *Adding Node.js to your virtualenv* recipe to install CHP. It also uses Python 3.5 instead of Python 2.

Otherwise, it is very similar to the *debianized-sentry* project, which is no surprise since they're based on the same cookiecutter template.

2.5.3 configsite

Author Nadav-Ruskin

URL https://github.com/Nadav-Ruskin/configsite

This project shows how to cross-package a web service for the ARM platform, using QEMU and Docker.

2.6 API / Code Reference

2.6.1 dh_virtualenv package

Submodules

dh_virtualenv.cmdline module

Helpers to handle debhelper command line options.

class dh_virtualenv.cmdline.**DebhelperOptionParser**(*usage=None*, *option_list=None*, *option_class=<class optparse.Option>*, *version=None*, *conflict_handler='error'*, *description=None*, *formatter=None*, *add_help_option=True*, *prog=None*, *epilog=None*)

Bases: optparse.OptionParser

Special OptionParser for handling Debhelper options.

Basically this means converting -O–option to –option before parsing.

parse_args(*args=None*, *values=None*)

dh_virtualenv.cmdline.**get_default_parser**()

dh_virtualenv.deployment module

class dh_virtualenv.deployment.**Deployment**(*package*, *extra_urls=[]*, *preinstall=[]*, *extras=[]*, *pip_tool='pip'*, *upgrade_pip=False*, *index_url=None*, *setuptools=False*, *python=None*, *builtin_venv=False*, *sourcedirectory=None*, *verbose=False*, *extra_pip_arg=[]*, *extra_virtualenv_arg=[]*, *use_system_packages=False*, *skip_install=False*, *install_suffix=None*, *requirements_filename='requirements.txt'*, *upgrade_pip_to=''*)

Bases: object

clean()

create_virtualenv()

find_script_files()
Find list of files containing python shebangs in the bin directory

fix_activate_path()
Replace the *VIRTUAL_ENV* path in bin/activate to reflect the post-install path of the virtualenv.

fix_local_symlinks()

fix_shebangs()
Translate /usr/bin/python and /usr/bin/env python shebang lines to point to our virtualenv python.

classmethod from_options(*package*, *options*)

install_dependencies()

install_package()

pip(**args*)

pip_preinstall(**args*)

run_tests()

venv_bin(*binary_name*)

2.7 Changelog

Following list contains the most notable changes by version. For a full list, consult the git history of the project.

2.7.1 1.2 (UNRELEASED)

- New option `--upgrade-pip-to` for increased build stability (#266) [@jhermann]

2.7.2 1.1

- Support new style shebangs generated by recent pip (#226) [@nailor]
- Add `--extras` option (#243) [@jhermann]
- Python 3.4 and 3.5 added to test environments (#238) [@jhermann]
- New build dependendcies (dh-exec + python-sphinx-rtd-theme) (#231) [@labeneator]
- Disallow building a package whilst within an activated virtualenv (#224) [@lamby]
- Use `python -m pip` instead of direct pip calls (#219) [@moritz]
- Ignore `--extra-pip-arg` in call for `--upgrade-pip` (#197) [@jhermann]
- buildsystem: Allow to specify a virtualenv name (#180) [@dzen]
- docs: Improved structure, new chapters [@jhermann]
- docs: Fix reference to pbuilder's USENETWORK option (#246) [@mkohler]
- Fix setuptools and pip setup when using built-in virtualenv with *–system-site-packages* (#247) [@lucasrangit]

2.7.3 1.0

- **Backwards incompatible** Change the default install root to `/opt/venvs`. This is due to the old installation root (`/usr/share/python`) clashing with Debian provided Python utilities. To maintain the old install location, use `DH_VIRTUALENV_INSTALL_ROOT` and point it to `/usr/share/python`.
- **Backwards incompatible** By default, do not run *setup.py test* upon building. The `--no-test` flag has no longer has any effect. To get the old behaviour, use the `--setuptools-test` flag instead.
- **Backwards incompatible** Buildsystem: Move files into build folder in install step instead of build step. Thanks to Ludwig Hähne for the patch!
- Deprecate `--pypi-url` in favour of `--index-url`
- Support upgrading pip to the latest release with `--upgrade-pip` flag.
- Buildsystem: Add support for `DH_UPGRADE_PIP`, `DH_UPGRADE_SETUPTOOLS` and `DH_UPGRADE_WHEEL`. Thanks to Kris Kvilekval for the implementation!
- Buildsystem: Add support for custom requirements file location using `DH_REQUIREMENTS_FILE` and for custom `pip` command line arguments using `DH_PIP_EXTRA_ARGS`. Thanks to Einar Forselv for implementing!
- Fixing shebangs now supports multiple interpreters. Thanks Javier Santacruz!
- Allow a custom pip executable via `--pip-tool` flag. Thanks Anthony Sottile for the implementation!

- Fix handling of shebang lines for cases where interpreter was wrapped in quotes. Thanks to Kamil Niechajewicz for fixing!
- Support extra arguments to be passed at virtualenv using `--extra-virtualenv-arg`. Thanks to Julien Duponchelle for the fix.

2.7.4 0.11

- Allow passing explicit filename for *requirements.txt* using `--requirements` option. Thanks to Eric Larson for implementing!
- Ensure that venv is configured before starting any daemons. Thanks to Chris Lamb for fixing this!
- Make sure *fix_activate_path* updates all activate scripts. Thanks to walrusVision for fixing this!

2.7.5 0.10

- **Backwards incompatible** Fix installation using the built-in virtual environment on 3.4. This might break installation on Python versions prior to 3.4 when using `--builtin-venv` flag. Thanks to Elonen for fixing!
- Honor `DH_VIRTUALENV_INSTALL_ROOT` in build system. Thanks to Ludwig Hähne for implementing!
- Allow overriding virtualenv arguments by using the `DH_VIRTUALENV_ARGUMENTS` environment variable when using the build system. Thanks to Ludwig Hähne for implementing!
- Add option to skip installation of the actual project. In other words using `--skip-install` installs only the dependencies of the project found in requirements.txt. Thanks to Phillip O'Donnell for implementing!
- Support custom installation suffix instead of the package name via `--install-suffix`. Thanks to Phillip O'Donnell for implementing!

2.7.6 0.9

- Support using system packages via a command line flag `--use-system-packages`. Thanks to Wes Mason for implementing this feature!
- Introduce a new, experimental, more modular build system. See the *Packaging Guide* for documentation.
- Respect the `DEB_NO_CHECK` environment variable.

2.7.7 0.8

- Support for running triggers upon host interpreter update. This new feature makes it possible to upgrade the host Python interpreter and avoid breakage of all the virtualenvs installed with dh-virtualenv. For usage, see the the *Getting Started*. Huge thanks to Jürgen Hermann for implementing this long wanted feature!
- Add support for the built-in `venv` module. Thanks to Petri Lehtinen!
- Allow custom `pip` flags to be passed via the `--extra-pip-arg` flag. Thanks to @labeneator for the feature.

2.7.8 0.7

- **Backwards incompatible** Support running tests. This change breaks builds that use distutils. For those cases a flag `--no-test` needs to be passed.
- Add tutorial to documentation

- Don't crash on debbuild parameters `-i` and `-a`
- Support custom source directory (debhelper's flag `-D`)

2.7.9 0.6

First public release of *dh-virtualenv*

CHAPTER 3

Indices and Tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

C

D

E

F

G

I

P

R

V