
dockyard Documentation

Release 0.1.0

Juergen Hermann

2020-03-09

1	Contributing	3
2	Documentation Contents	5
2.1	Fundamental Docker	5
2.2	Python Base Images	5
2.3	Building Debian Packages in Docker	7
2.4	Contribution Guidelines	14
2.5	Software License	16
3	References	17
3.1	Tools	17
3.2	Packages	17
4	Indices and Tables	19



Basic Dockerfile templates and other Docker build helpers.

CHAPTER 1

Contributing

To create a working directory for this project, call these commands:

```
git clone "https://github.com/jhermann/dockyard.git"  
cd "dockyard"  
. .env --yes  
invoke docs -b
```

Contributing to this project is easy, and reporting an issue or adding to the documentation also improves things for every user. You don't need to be a developer to contribute. See [Contribution Guidelines](#) for more.

2.1 Fundamental Docker

Since there are plenty of documents on the web with details about basic Docker use and writing state-of-the-art Dockerfiles, they're just listed here instead of repeating the same information.

2.1.1 Reference Documents

- [Official Docker Documentation](#)

2.1.2 Articles & How-Tos

- [Why Your Dockerized Application Isn't Receiving Signals](#) by Hynek Schlawack explains how to write your Dockerfile so that your main application actually receives shutdown signals sent by the Docker daemon. Otherwise it cannot do an orderly cleanup, and instead is terminated forcibly after the shutdown timeout.

See also [Gracefully Shutdown Docker Container](#).

2.2 Python Base Images

2.2.1 Official Docker Community Images

The Docker “Official Images” for Python 3 based on `debian:stretch-slim` (55.3 MiB) are around 140 MiB in size. Namely, `python:3.6-slim-stretch` comes in at 137.9 MiB, while Python 3.7 is a little bigger at 142.7 MiB. The ones based on Alpine have 74.2 MiB and 78.1 MiB. Google's *distroless* gcr.io/distroless/python3 beats that with 51 MiB.

The Dockerfiles and related sources for the images can be found at [docker-library/python](#).

2.2.2 Ubuntu Bionic + Python 3

Python 3.6 (Ubuntu default)

This shows how to create a Python 3 image based on *Ubuntu Bionic*. It is roughly 55 MiB larger than the Alpine-based equivalent, but also comes with runtime essentials like enabled locale, CA certificates, and `glibc` instead of the often problematic `musl libc`.

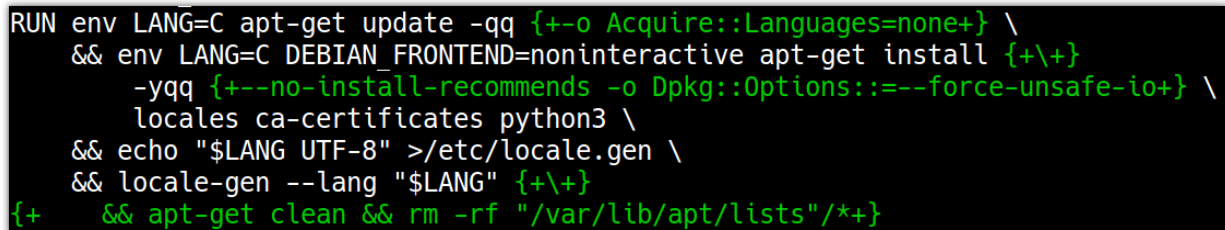
The new ‘minimized’ Ubuntu images are a good base when you want to stay in known waters, and your payload is not just a trivial script or something equally simple. Alpine is a good choice for these small payloads, which also often just have pure-Python dependencies, avoiding most trouble with the different `libc`.

There is a simple and an optimized version of the Dockerfile, with a few magic incantations added to the latter. To build both versions with clean caches and timings for each build, use this:

```
docker system prune --all --force
( cd biopy3 \
  && docker pull ubuntu:bionic \
  && time docker build -f Dockerfile.simple -t biopy3-simple . \
  && time docker build -f Dockerfile.optimized -t biopy3 . \
)
```

Both images are built in about 15 seconds (on an Intel Core i7-6700 with SSD storage). The ‘optimized’ one has a sub-second advantage regarding build time. If you install more packages, the difference should get more pronounced.

The image sizes show a much clearer picture: **129.6 MiB** for the ‘optimized’ version, and **176.5 MiB** for the ‘simple’ one.



```
RUN env LANG=C apt-get update -qq {+-o Acquire::Languages=none+} \
  && env LANG=C DEBIAN_FRONTEND=noninteractive apt-get install {+{+}+} \
  -yqq {+--no-install-recommends -o Dpkg::Options::=--force-unsafe-io+} \
  locales ca-certificates python3 \
  && echo "$LANG UTF-8" >/etc/locale.gen \
  && locale-gen --lang "$LANG" {+{+}+} \
  {+ && apt-get clean && rm -rf "/var/lib/apt/lists"/*+}
```

Fig. 1: Differences between simple and optimized Dockerfile versions

Here are the objectives for each of the changes as shown above:

- `-o Acquire::Languages=none` speeds up package list downloads by ignoring unneeded translation files.
- `--no-install-recommends` limits the installed package set to what you listed explicitly, and hard dependencies of that list – e.g. `nodejs` will otherwise install a *full* Python 2.7 for no good reason, instead of just `python-minimal`. That improves both build times and image size.
- `-o Dpkg::Options::=--force-unsafe-io` switches off `sync` system calls during package expansion, speeding up package installation – since data is saved to a container layer shortly afterwards anyway, this is safe despite the option’s name.
- `apt-get clean && rm -rf "/var/lib/apt/lists"/*` removes any cached packages and meta-data *before* the layer is stored. Both are things that we simply do not need in an immutable container. While the APT `clean` is already baked into newer Debian base images, having an explicit cleanup call doesn’t hurt either.

And the `env LANG=C` before the `apt-get` commands suppresses locale initialization warnings since locales are not generated yet.

Python 3.7 (Deadsnakes PPA)

In `biopy3/Dockerfile.deadsnakes` the newest Python version is added, as available from the Deadsnakes PPA.

Due to packaging mechanics, this gets installed in addition to Ubuntu's default Python 3.6 – the resulting image size is 168.9 MiB (i.e. ~40 MiB more). That means this is **not** a sensible option compared to images like `python:3.7-slim-stretch`. Also, timely security updates are not guaranteed for the PPA release channel.

Note that for the `pip` installation via `get-pip.py`, the command `set -o pipefail` is used to ensure the build fails if `wget` fails. That in turn requires using the `SHELL` instruction to switch the default shell from `dash`, which does not implement `set -o shell` options, to `bash` which supports that option.

```
SHELL ["/bin/bash", "-c"]
...
RUN ...
    && set -o pipefail \
    && wget -qO- https://bootstrap.pypa.io/get-pip.py | python3.7 \
    ...
```

Python 3.7 (pyenv)

To build images for Python 3.7 and up, compiled and installed using `pyenv`, run these commands:

```
command cd $(git rev-parse --show-toplevel)/biopy3
pyversion=$(grep ARG.pyversion= Dockerfile.pyenv | cut -f2 -d= | cut -f1-2 -d.)
pyenv_version=$(grep ARG.pyenv_version= Dockerfile.pyenv | cut -f2 -d=)
wget https://github.com/pyenv/pyenv/archive/v${pyenv_version}.tar.gz
./make-pyenv-tk.sh >/dev/null

declare -A base_images=( [biopyenv3]="ubuntu:bionic" [debpyenv3]="debian:stretch-slim
→" )
for tag in "${!base_images[@]}; do
    docker build --build-argument distro="${base_images[$tag]}" \
        -t ${tag} -t ${tag}:${pyversion} -f Dockerfile.pyenv .
    docker build --build-argument distro="${base_images[$tag]}" \
        -t ${tag}-tk -t ${tag}-tk:${pyversion} -f Dockerfile.pyenv-tk .
done
```

See the comments at the start of `Dockerfile.pyenv` for some more details and resulting image sizes.

TODO

Look at other options like Conda or PyRun.

2.3 Building Debian Packages in Docker

2.3.1 Deploying Python Code with `dh-virtualenv`

The code shown here is taken from the [debianized-jupyterhub](#) project, and explains how to use a Docker container to build a Debian package containing Python code.

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
- *only need* ‘docker-ce’ installed on your workstation / the build host

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 as build arguments. It also takes care of copying the resulting files out of the build container.

Besides the Dockerfile itself we also need a `.dockerignore` file, to avoid having a full development virtualenv or all of `.git` as part of the container build context.

So keep an eye on the Sending build context to Docker daemon message that docker build emits very early in a build, it should be only one to a few hundred KiB at most. If it is more, check your `.dockerignore` file for omissions that you might need to add.

The build script

Let’s get to the code – at the start of the build script, the given platform and existing project metadata is stored into a bunch of variables.

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

set -e

# If you change this, you MUST also change "debian/control" and "debiana/rules"
PYTHON_MINOR="3.8" # Deadsnakes version on Ubuntu

# Check build environment
if ! command -v dh_listpackages >/dev/null 2>&1; then
    echo >&2 "ERROR: You must 'apt install debhelper' on the build host."
    exit 1
fi

# 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)"
pypi_version="$(./setup.py --version)"
pkgname="$(dh_listpackages)"
tag=$pypi_name-$dist_id-$codename
staging="$pypi_name-$pypi_version"

build_opts=(
    -f Dockerfile.build
    --tag $tag
    --build-arg "DIST_ID=$dist_id"
    --build-arg "CODENAME=$codename"
    --build-arg "PKGNAME=$pkgname"
)

if test "$dist_id" = "ubuntu"; then
```

(continues on next page)

(continued from previous page)

```

    build_opts+=( --build-arg "PYVERSION=$PYTHON_MINOR" )
fi

```

On Ubuntu, the Deadsnakes packages are used, thus the default of the `PYVERSION` Docker argument is replaced. Since the `control` file contains that more detailed version, the reverse is done for Debian builds – `3.x` is replaced by just `3`.

To keep this and other possible changes out of `git`'s reach, a staging copy of the `workdir` is created via `sdist` – adding `-k` keeps the expanded distribution tree intact, so we can change and finally move it to `build/docker.staging`. Creating that copy based on a source distribution also tests that such a distribution is complete and buildable, as a welcome side-effect.

```

# Create Docker staging directory and apply environment changes
echo "*** Creating staging directory (sdist)"
echo "*** Ignore warnings regarding non-matching filters..."
python3 ./setup.py -q sdist -k
if test "$dist_id" = "debian"; then
    sed -i -e "s/python$PYTHON_MINOR/python3/g" "$staging/debian/control"
fi

mkdir -p build
rm -rf build/docker.staging 2>/dev/null
mv "$staging" build/docker.staging

```

After that prep work, we get to build our package, passing along the needed build arguments. The results are copied using `docker run` out of the `/dpkg` directory, where the Docker build process put them (see below). Also the package metadata is shown for a quick visual check if everything looks OK.

```

# Build in Docker container, save results, and show package info
echo
echo "*** Building DEB package (takes a while)"
rm -f dist/${pkgname}?*${pypi_version}/*.*
docker build "${build_opts[@]}" "$@" build/docker.staging
mkdir -p dist
docker run --rm $tag tar -C /dpkg -c . | tar -C dist -x
ls -lh dist/${pkgname}?*${pypi_version}/*.*

```

The Dockerfile

This is the start of the Dockerfile, setting up some parameters used in the `RUN` instructions, and pulling the base image.

```

# Build Debian package using dh-virtualenv
#
# To create a package for Stretch in `dist/`, call:
#
# ./build.sh debian:stretch
#
# Build arguments, as provided by 'build.sh'
ARG DIST_ID="debian"
ARG CODENAME="stretch"
ARG PYVERSION="3"
ARG PKGNAME

# Other build arguments (adapt as needed)
ARG NODEREPO="node_10.x"

```

(continues on next page)

(continued from previous page)

```

ARG DEB_POOL="http://ftp.nl.debian.org/debian/pool/main"

## Start package builder image for the chosen platform
FROM ${DIST_ID}:${CODENAME} AS dpkg-build

# Pass build args into image scope
ARG CODENAME
ARG PYVERSION
ARG PKGNAME
ARG NODEREPO
ARG DEB_POOL

```

The first RUN installs all the build dependencies on top of the base image. This also includes installing NodeJS, as explained in more detail by [Adding Node.js to your virtualenv](#) further below.

```

# Install build tools and package build deps including nodejs
RUN ( test "${CODENAME}" = "xenial" \
    && echo "deb [trusted=yes] http://ppa.launchpad.net/deadsnakes/ppa/ubuntu $
    ↪ ${CODENAME} main" \
    >/etc/apt/sources.list.d/deadsnakes.list || : ) \
    && env LANG=C apt-get update -qq -o Acquire::Languages=none \
    && env LANG=C DEBIAN_FRONTEND=noninteractive apt-get install \
    -yqq --no-install-recommends -o Dpkg::Options::=--force-unsafe-io \
    "" \
    apt-transport-https \
    apt-utils \
    build-essential \
    curl \
    debhelper \
    devscripts \
    equivs \
    gnupg2 \
    gzip \
    libjs-sphinxdoc \
    libparse-debianchangelog-perl \
    lsb-release \
    python${PYVERSION} \
    python${PYVERSION}-venv \
    python${PYVERSION}-tk \
    python${PYVERSION}-dev \
    python3-pkg-resources \
    python3-setuptools \
    python3-venv \
    sphinx-rtd-theme-common \
    tar \
    "" \
    libcurl4-openssl-dev \
    libffi-dev \
    libfontconfig1 \
    libfreetype6-dev \
    libjpeg-dev \
    libncurses5-dev \
    libncursesw5-dev \
    libssl-dev \
    libxml2-dev \
    libxslt1-dev \

```

(continues on next page)

(continued from previous page)

```

libyaml-dev \
libz-dev \
libzmq3-dev \
&& ( curl -sL https://deb.nodesource.com/gpgkey/nodesource.gpg.key \
    | env APT_KEY_DONT_WARN_ON_DANGEROUS_USAGE=true apt-key add - ) \
&& echo "deb https://deb.nodesource.com/${NODEREPO} ${CODENAME} main" \
    >/etc/apt/sources.list.d/nodesource.list \
&& apt-get update -qq -o Acquire::Languages=none \
&& env LANG=C DEBIAN_FRONTEND=noninteractive apt-get install \
    -yqq --no-install-recommends -o Dpkg::Options::=--force-unsafe-io nodejs \
&& apt-get clean && rm -rf "/var/lib/apt/lists"/*

```

The second one installs the latest `dh-virtualenv` version, and also updates the Python packaging toolset to the latest versions. The `ADD` instruction above it downloads the pre-built `dh-virtualenv` DEB from Debian ‘sid’ – this way we get the same version across all platforms, and can also rely on the features of the latest release.

```

# Uncomment and adapt these ENV instructions to use a local PyPI mirror
# (examples for devpi and JFrog Artifactory)
#ENV PIP_TRUSTED_HOST="devpi.local"
#ENV PIP_INDEX_URL="http://${PIP_TRUSTED_HOST}:3141/root/pypi/+simple/"
#ENV PIP_TRUSTED_HOST="artifactory.local"
#ENV PIP_INDEX_URL="https://${PIP_TRUSTED_HOST}/artifactory/api/pypi/pypi.python.org/
→simple"

# Install updated Python tooling and a current 'dh-virtualenv'
WORKDIR /dpkg-build
ADD "${DEB_POOL}/d/dh-virtualenv/dh-virtualenv_1.1-1_all.deb" ./
RUN dpkg -i --force-unsafe-io \
    --ignore-depends=python:any \
    --ignore-depends=virtualenv \
    --ignore-depends=sphinx-rtd-theme-common \
    *_all.deb \
    && python${PYVERSION} -m easy_install pip \
    && python${PYVERSION} -m pip install -U setuptools wheel

```

Finally, the third `RUN` builds the package for your project and makes a copy of the resulting files, for the build script to pick them up.

```

# Build project and show metadata of built package
COPY ./ ./
RUN hr() { printf '\n  %-74s\n' "${1}" | tr ' ' = ; } \
    && hr Versions && python${PYVERSION} -m pip --version && dh-virtualenv --version \
    && sed -i -r \
    -e "1s/(UNRELEASED|unstable|jessie|stretch|xenial|bionic)/$(lsb_release -
→cs)/g" \
    debian/changelog \
    && dpkg-buildpackage -us -uc -b && mkdir -p /dpkg && cp -pl /${PKGNAME}[_-]* /
→dpkg \
    && hr DPKG-Info && dpkg-deb -I /dpkg/${PKGNAME}_*.deb

```

See the comments in the Dockerfile for more details, and *Ubuntu Bionic + Python 3* for an explanation of ‘special’ `apt` arguments, used to speed up the build process and keep image sizes small.

To adapt this to your own project, you have to change these things:

- Remove the instructions and commands for installing NodeJS, if you don’t need that (ARG `NODEREPO`, and several commands near the end of the first `RUN` instruction).

- Check the second part of the package list in the first `apt` call – remove and add libraries depending on your project’s build dependencies.
- As mentioned in the comments, you can activate a local Python repository by setting `PIP_*` environment variables accordingly.

The `.dockerignore` file

As previously mentioned, we want to keep artifacts generated in the `workdir` out of Docker builds, for performance reasons and to avoid polluting the build context.

This is an example, you need to at least change the project name (`jupyterhub`) in your own copy:

```
# Keep in sync with ".gitignore" and "debian/source/options"!
.git/
.venv/
*~
*.egg-info/
*.py[co]
__pycache__/
.pip-cache/
dist/
build/
#.npmrc
debian/debhelper-build-stamp
debian/files
debian/*.debhelper
debian/*.substvars
debian/sdist/
debian/jupyterhub/
debian/jupyterhub-*.*/
```

Putting it all together

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

```
$ ./build.sh ubuntu:bionic
Sending build context to Docker daemon 127.5kB
Step 1/16 : ARG DIST_ID="debian"
Step 2/16 : ARG CODENAME="stretch"
Step 3/16 : ARG PKGNAME
Step 4/16 : ARG NODEREPO="node_8.x"
Step 5/16 : ARG DEB_POOL="http://ftp.nl.debian.org/debian/pool/main"
Step 6/16 : FROM ${DIST_ID}:${CODENAME} AS dpkg-build
--> cd6d8154f1e1
...
dpkg-buildpackage: info: binary-only upload (no source included)
new Debian package, version 2.0.
size 21160196 bytes: control archive=192388 bytes.
  110 bytes, 4 lines control
  1250 bytes, 25 lines control
  1023901 bytes, 8079 lines md5sums
  4853 bytes, 156 lines * postinst #!/bin/sh
  1475 bytes, 48 lines * postrm #!/bin/sh
  696 bytes, 35 lines * preinst #!/bin/sh
  1047 bytes, 41 lines * prerm #!/bin/sh
```

(continues on next page)

(continued from previous page)

```

    70 bytes,      2 lines      shlibs
    419 bytes,    10 lines      triggers
Package: jupyterhub
Version: 0.9.4-0.1~bionic
Architecture: amd64
Maintainer: l&l Group <jh@web.de>
Installed-Size: 112648
Pre-Depends: dpkg (>= 1.16.1), python3 (>= 3.5)
Depends: libc6 (>= 2.25), libcurl4 (>= 7.18.0), libexpat1 (>= 2.1~beta3), libgcc1 (>
↳= 1:3.0),
    libssl1.1 (>= 1.1.0), libstdc++6 (>= 4.1.1), zlib1g (>= 1:1.2.0), python3-tk (>=
↳3.5),
    nodejs (>= 8), nodejs (< 9)
Suggests: oracle-java8-jre | openjdk-8-jre | zulu-8
Section: contrib/python
Priority: extra
Homepage: https://github.com/landl/debianized-jupyterhub
Description: Debian packaging of JupyterHub, a multi-user server for Jupyter
↳notebooks.
...
Removing intermediate container 4fd85ab1f1cc
---> 4197e1d56385
Successfully built 4197e1d56385
Successfully tagged debianized-jupyterhub-ubuntu-bionic:latest
-rw-r----- 1 jhe jhe 9,8K Oct  1 15:33 dist/jupyterhub_0.9.4-0.1~bionic_amd64.
↳buildinfo
-rw-r----- 1 jhe jhe 1,4K Oct  1 15:33 dist/jupyterhub_0.9.4-0.1~bionic_amd64.changes
-rw-r----- 1 jhe jhe 21M Oct  1 15:33 dist/jupyterhub_0.9.4-0.1~bionic_amd64.deb
-rw-r----- 1 jhe jhe 330K Oct  1 15:32 dist/jupyterhub-dbgsym_0.9.4-0.1~bionic_amd64.
↳ddeb

```

The package files are now in `dist/`, and you can `dput` them into your local repository, or install them using `dpkg`

```
-i ....
```

2.3.2 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
endif

%:
    dh $@ --with python-virtualenv

.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 – when building in a container it obviously has to be the project-local one so it becomes part of the build context.

2.4 Contribution Guidelines

2.4.1 Overview

Contributing to this project is easy, and reporting an issue or adding to the documentation also improves things for every user. You don't need to be a developer to contribute.

Reporting issues

Please use the *GitHub issue tracker*, and describe your problem so that it can be easily reproduced. Providing relevant version information on the project itself and your environment helps with that.

Improving documentation

The easiest way to provide examples or related documentation that helps other users is the *GitHub wiki*.

If you are comfortable with the Sphinx documentation tool, you can also prepare a pull request with changes to the core documentation. GitHub's built-in text editor makes this especially easy, when you choose the “*Create a new branch for this commit and start a pull request*” option on saving. Small fixes for typos and the like are a matter of minutes when using that tool.

Code contributions

Here's a quick guide to improve the code:

1. Fork the repo, and clone the fork to your machine.
2. Add your improvements, the technical details are further below.
3. Run the tests and make sure they're passing (`invoke test`).
4. Check for violations of code conventions (`invoke check`).
5. Make sure the documentation builds without errors (`invoke build --docs`).
6. Push to your fork and submit a [pull request](#).

Please be patient while waiting for a review. Life & work tend to interfere.

2.4.2 Details on contributing code

This project is written in [Python](#), and the documentation is generated using [Sphinx](#). [setuptools](#) and [Invoke](#) are used to build and manage the project. Tests are written and executed using [pytest](#) and [tox](#).

Set up a working development environment

To set up a working directory from your own fork, follow [these steps](#), but replace the repository `https` URLs with `SSH` ones that point to your fork.

For that to work on Debian type systems, you need the `git`, `python`, and `python-virtualenv` packages installed. Other distributions are similar.

Add your changes to a feature branch

For any cohesive set of changes, create a *new* branch based on the current upstream `master`, with a name reflecting the essence of your improvement.

```
git branch "name-for-my-fixes" origin/master
git checkout "name-for-my-fixes"
... make changes...
invoke ci # check output for broken tests, or PEP8 violations and the like
... commit changes...
git push origin "name-for-my-fixes"
```

Please don't create large lumps of unrelated changes in a single pull request. Also take extra care to avoid spurious changes, like mass whitespace diffs. All Python sources use spaces to indent, not TABs.

Make sure your changes work

Some things that will increase the chance that your pull request is accepted:

- Follow style conventions you see used in the source already (and read [PEP8](#)).
- Include tests that fail *without* your code, and pass *with* it. Only minor refactoring and documentation changes require no new tests. If you are adding functionality or fixing a bug, please also add a test for it!
- Update any documentation or examples impacted by your change.
- Styling conventions and code quality are checked with `invoke check`, tests are run using `invoke test`, and the docs can be built locally using `invoke build --docs`.

Following these hints also expedites the whole procedure, since it avoids unnecessary feedback cycles.

2.5 Software License

Copyright (c) 2018 Juergenm Hermann

The contained files can be used either directly as base images or as templates to create new Dockerfiles with common optimizations already baked in.

2.5.1 Full License Text

MIT License

Copyright (c) 2018 Jürgen Hermann

Permission **is** hereby granted, free of charge, to **any** person obtaining a copy of this software **and** associated documentation files (the "**Software**"), to deal **in** the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, **and/or** sell copies of the Software, **and** to permit persons to whom the Software **is** furnished to do so, subject to the following conditions:

The above copyright notice **and** this permission notice shall be included **in all** copies **or** substantial portions of the Software.

THE SOFTWARE IS PROVIDED "**AS IS**", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

3.1 Tools

- [Cookiecutter](#)
- [PyInvoke](#)

3.2 Packages

- [Rituals](#)

CHAPTER 4

Indices and Tables

- `genindex`
- `modindex`
- `search`