Piwheels 0.16 Documentation

Release 0.16

Ben Nuttall

Jul 03, 2020

Contents

CHAPTER 1

Overview

The piwheels project is designed to automate building of wheels from packages on PyPI for a set of pre-configured ABIs. As the name suggests, it was originally built for Raspberry Pis but there's nothing particular in the codebase that should limit it to that platform. The system relies on the following components:

Component	Description
piw-master (page 3)	Coordinates the various build slaves, using the database to store all relevant information, and keeps the web site up to date.
piw-slave (page 7)	Builds package on behalf of the piwheels master. Is intended to run on separate machines to the master, partly for performance and partly for security.
piw-monitor (page 9)	Provides a friendly curses-based UI for interacting with the piwheels master.
piw-sense (page 11)	Provides a friendly Sense HAT-based UI for interacting with the piwheels master.
piw-initdb (page 13)	A simple maintenance script for initializing or upgrading the database to the current version.
piw-import (page 15)	A tool for importing wheels manually into the piwheels database and file-system.
piw-remove (page 19)	A tool for manually removing builds from the database and file-system.
piw-logger (page 21)	A tool for transferring download statistics into the piwheels database.
database server	Currently only PostgreSQL[1] is supported (and frankly that's all we're ever likely to support). This provides the master's data store.
web server	Anything that can serve from a static directory is fine here. We use Apache[2] in production.

Note: At present the master is a monolithic application, but the internal architecture is such that it could, in future, be split into three parts: one that deals exclusively with the database server, one that deals exclusively with the file-system served by the web server, and one that talks to the piwheels slave and monitor processes.

[1] https://postgresql.org/
[2] https://httpd.apache.org/

CHAPTER 2

piw-master

The piw-master script is intended to be run on the database and file-server machine. It is recommended you do not run piw-slave on the same machine as the piw-master script. The database specified in the configuration must exist and have been configured with the piw-initdb script. It is recommended you run piw-master as an ordinary unprivileged user, although obviously it will need write access to the output directory.

2.1 Synopsis

```
piw-master [-h] [--version] [-c FILE] [-q] [-v] [-l FILE] [-d DSN]
                [-o PATH] [--dev-mode] [--pypi-xmlrpc URL]
                [--pypi-simple URL] [--status-queue ADDR]
                [--control-queue ADDR] [--import-queue ADDR]
                [--log-queue ADDR] [--slave-queue ADDR] [--file-queue ADDR]
                [--web-queue ADDR] [--builds-queue ADDR] [--db-queue ADDR]
                [--fs-queue ADDR] [--stats-queue ADDR]
```

2.2 Description

-h, --help
Show this help message and exit

--version
Show program's version number and exit

-c FILE, **--configuration** FILE
Specify a configuration file to load

-q, --quiet
Produce less console output

-v, --verbose
Produce more console output

-l FILE, **--log-file** FILE
Log messages to the specified file

-d DSN, **--dsn** DSN
: The database to use; this database must be configured with piw-initdb and the user must *not* be a PostgreSQL[3] superuser (default: postgres:///piwheels)

-o PATH, **--output-path** PATH
: The path under which the website should be written; must be writable by the current user

--dev-mode
: Run the master in development mode, which reduces some timeouts and tweaks some defaults

--pypi-xmlrpc URL
: The URL of the PyPI XML-RPC service (default: https://pypi.python.org/pypi)

--pypi-simple URL
: The URL of the PyPI simple API (default: https://pypi.python.org/simple)

--status-queue ADDR
: The address of the queue used to report status to monitors (default: ipc:///tmp/piw-status); this is usually an ipc address

--control-queue ADDR
: The address of the queue a monitor can use to control the master (default: ipc:///tmp/piw-control); this is usually an ipc address

--import-queue ADDR
: The address of the queue used by *piw-import* (page 15) (default: ipc:///tmp/piw-import); this should always be an ipc address

--log-queue ADDR
: The address of the queue used by *piw-logger* (page 21) (default: ipc:///tmp/piw-logger); this should always be an ipc address

--slave-queue ADDR
: The address of the queue used to talk to the build slaves (default: tcp://*:5555); this is usually a tcp address

--file-queue ADDR
: The address of the queue used to transfer files from slaves (default: tcp://*:5556); this is usually a tcp address

--builds-queue ADDR
: The address of the queue used to store pending builds (default: inproc://builds)

--db-queue ADDR
: The address of the queue used to talk to the database server (default: inproc://db)

--fs-queue ADDR
: The address of the queue used to talk to the file- system server (default: inproc://fs)

--stats-queue ADDR
: The address of the queue used to send statistics to the collator task (default: inproc://stats)

2.3 Deployment

A typical deployment of the master service on a Raspbian server goes something like this (each step assumes you start as root):

1. Install the pre-requisite software:

```
# apt install postgresql-9.6 apache2 python3-psycopg2 python3-geoip
# apt install python3-sqlalchemy python3-urwid python3-zmq python3-voluptuous␣
↪python3-chameleon
# pip install piwheels[monitor,master,logger]
```

[3] https://postgresql.org/

2. Set up the (unprivileged) piwheels user and the output directory:

```
# groupadd piwheels
# useradd -g piwheels -m piwheels
# mkdir /var/www/piwheels
# chown piwheels:piwheels /var/www/piwheels
```

3. Set up the database:

```
# su - postgres
$ createuser piwheels
$ createdb -O postgres piwheels
$ piw-initdb
```

4. Set up the web server:
   - Point the document root to the output path (`/var/www/piwheels` above, but it can be anywhere your piwheels user has write access to; naturally you want to make sure your web-server's user only has *read* access to the location).
   - Set up SSL for the web server (e.g. with Let's Encrypt[4]; the dehydrated[5] utility is handy for getting and maintaining the SSL certificates).
5. Start the master running (it'll take quite a while to populate the list of packages and versions from PyPI on the initial run so get this going before you start bringing up build slaves):

```
# su - piwheels
$ piw-master -v
```

6. Deploy some build slaves; see *piw-slave* (page 7) for deployment instructions.

2.4 Automatic start

If you wish to ensure that the master starts on every boot-up, you may wish to define a systemd unit for it. Example units can be also be found in the root of the piwheels repository:

```
# wget https://raw.githubusercontent.com/bennuttall/piwheels/master/piwheels-
↪master.service
# cp piwheels-master.service /etc/systemd/system/
# systemctl daemon-reload
# systemctl enable piwheels-master
# systemctl start piwheels-master
```

2.5 Upgrades

The master will check that build slaves have the same version number and will reject them if they do not. Furthermore, it will check the version number in the database's *configuration* table matches its own and fail if it does not. Re-run the *piw-initdb* (page 13) script as the PostgreSQL super-user to upgrade the database between versions (downgrades are not supported, so take a backup first!).

[4] https://letsencrypt.org/
[5] https://github.com/lukas2511/dehydrated

CHAPTER 3

piw-slave

The piw-slave script is intended to be run on a standalone machine to build packages on behalf of the piw-master script. It is intended to be run as an unprivileged user with a clean home-directory. Any build dependencies you wish to use must already be installed. The script will run until it is explicitly terminated, either by Ctrl+C, SIGTERM, or by the remote piw-master script.

3.1 Synopsis

```
piw-slave [-h] [--version] [-c FILE] [-q] [-v] [-l FILE] [-m HOST]
          [-t DURATION]
```

3.2 Description

-h, --help
Show this help message and exit

--version
Show program's version number and exit

-c FILE, **--configuration** FILE
Specify a configuration file to load

-q, --quiet
Produce less console output

-v, --verbose
Produce more console output

-l FILE, **--log-file** FILE
Log messages to the specified file

-m HOST, **--master** HOST
The IP address or hostname of the master server (default: localhost)

-t DURATION, **--timeout** DURATION
The time to wait before assuming a build has failed (default: 3h)

3.3 Deployment

Our typical method of deployment is to spin up a new Pi as a build slave (through Mythic Beasts' control panel) then execute a script to install the piwheels code, and all the build dependencies that we feel are reasonable to support under various Raspbian versions. The deployment script can be found in the root of the piwheels repository:

```
# wget https://raw.githubusercontent.com/bennuttall/piwheels/master/deploy_slave.sh
# chmod +x deploy_slave.sh
# ./deploy_slave.sh
```

However, you will very likely wish to customize this script for your own purposes, e.g. to support a different set of dependencies, or to customize the typical build environment.

Once the script is complete, simply switch to the unprivileged user used to run the build slave, and execute *piw-slave* (page 7). For example, assuming the master's IP address is 10.0.0.1:

```
# su - piwheels
$ piw-slave -m 10.0.0.1
```

3.4 Automatic start

If you wish to ensure that the build slave starts on every boot-up, you may wish to define a systemd unit for it. Example units can be also be found in the root of the piwheels repository:

```
# wget https://raw.githubusercontent.com/bennuttall/piwheels/master/piwheels-slave.
↪service
# cp piwheels-slave.service /etc/systemd/system/
# systemctl daemon-reload
# systemctl enable piwheels-slave
# systemctl start piwheels-slave
```

Warning: Be aware that this example unit forces a reboot in the case that the build slave fails (as occasionally happens with excessively complex packages).

Because of this you *must* ensure that the slave executes successfully prior to installing the unit, otherwise you're liable to leave your build slave in permanent reboot cycle. This isn't a huge issue for a build slave that's physically in front of you (from which you can detach and tweak the storage), but it may be an issue if you're dealing with a cloud builder.

CHAPTER 4

piw-monitor

The piw-monitor application is used to monitor (and optionally control) the piw-master script. Upon startup it will request the status of all build slaves currently known to the master, and will then continually update its display as the slaves progress through builds. The controls at the bottom of the display allow the administrator to pause or resume the master script, kill build slaves that are having issues (e.g. excessive resource consumption from a huge build) or terminate the master itself.

4.1 Synopsis

```
piw-monitor [-h] [--version] [-c FILE] [--status-queue ADDR]
            [--control-queue ADDR]
```

4.2 Description

-h, --help
Show this help message and exit

--version
Show program's version number and exit

-c FILE, **--configuration** FILE
Specify a configuration file to load

--status-queue ADDR
The address of the queue used to report status to monitors (default: ipc:///tmp/piw-status)

--control-queue ADDR
The address of the queue a monitor can use to control the master (default: ipc:///tmp/piw-control)

4.3 Usage

The monitor application should be started on the same machine as the master after the *piw-master* (page 3) script has been started. After initialization it will request the current status of all build slaves from the master, displaying this in a list in the middle of the screen.

The `Tab` key can be used to navigate between the list of build slaves and the controls at the bottom of the screen. Mouse control is also supported, provided the terminal emulator supports it. Finally, hot-keys for all actions are available. The actions are as follows:

4.3.1 Pause

Hotkey: `p`

Pauses operations on the master. This causes *Cloud Gazer* (page 25) to stop querying PyPI, *Slave Driver* (page 25) to return "SLEEP" in response to any build slave requesting new packages, and so on. This is primarily a debugging tool to permit the developer to peek at the system in a more or less frozen state before resuming things.

4.3.2 Resume

Hotkey: `r`

Resumes operations on the master when paused.

4.3.3 Kill Slave

Hotkey: `k`

The next time the selected build slave requests a new package (with "IDLE") the master will return "BYE" indicating the slave should terminate. Note that this cannot kill a slave in the middle of a build (that would require a more complex asynchronous protocol in *Slave Driver* (page 25)), but is useful for shutting things down in an orderly fashion.

4.3.4 Terminate Master

Hotkey: `t`

Tells the master to shut itself down. In a future version, the master *should* request all build slaves to terminate as well, but currently this is unimplemented.

4.3.5 Quit

Hotkey: `q`

Terminate the monitor. Note that this won't affect the master.

CHAPTER 5

piw-sense

The piw-sense application is an alternative monitor for the piw-master script that uses the Raspberry Pi Sense HAT as its user interface. Upon startup it will request the status of all build slaves currently known to the master, and will then continually update its display as the slaves progress through builds. The Sense HAT's joystick can be used to navigate information about current builds, and kill builds slaves that are having issues, or terminate the master itself.

5.1 Synopsis

```
piw-sense [-h] [--version] [-c FILE] [--status-queue ADDR]
               [--control-queue ADDR] [-r DEGREES]
```

5.2 Description

-h, --help
Show this help message and exit

--version
Show program's version number and exit

-c FILE, **--configuration** FILE
Specify a configuration file to load

--status-queue ADDR
The address of the queue used to report status to monitors (default: ipc:///tmp/piw-status)

--control-queue ADDR
The address of the queue a monitor can use to control the master (default: ipc:///tmp/piw-control)

-r DEGREES, **--rotate** DEGREES
The rotation of the HAT in degrees; must be 0 (the default), 90, 180, or 270

5.3 Usage

The Sense monitor can be started on the same machine as the master after the *piw-master* (page 3) script has been started. After initialization it will request the current status of all build slaves from the master.

5.3.1 Layout

The top three (normally blue) rows of the display are used for some important statistics:

- The top row represents the ping time from the master, or more specifically the time since the last message was received. This will continually increase (changing white), and reset with each message received. If 30 seconds elapse without any messages being received, this row will pulse red until another message is received, resetting the count.
- The second row represents available disk space for the output directory on the master. White pixels represent remaining space, and the scale is simply percentage (all blue = 0%, all white = 100%).
- The third row represents the number of pending builds on the master. The scale is one white pixel = 8 builds in the queue (with partial shades representing <8 builds).

The remaining rows represent all build slaves. Each pixel represents a single build slave, working vertically then horizontally. Build slaves are sorted first by ABI, then by label (as in *piw-monitor* (page 9)).

- A gray pixel indicates an idle build slave.
- A green pixel indicates an active build.
- A blue pixel indicates an active file transfer after a successful build.
- A purple pixel indicates a build slave cleaning up after a build.
- A yellow pixel indicates an active build that's been running for more than 15 minutes; not necessarily a problem but longer than average.
- A red pixel indicates a build slave that's either timed out or been terminated; it should disappear from the display within a few seconds.

5.3.2 Navigation

The pixel that pulses white indicates your current position, which can be moved with the Sense HAT joystick. Pressing the joystick in when a build-slave is selected (indicated by it pulsing white) will bring up detailed information on that build slave.

Scroll left and right to navigate through the build-slave information (label, ABI, current task, and kill option). Press the joystick in to return to the main display (optionally killing the build slave if the kill screen is selected).

Scroll the cursor off the top of the display to go to detailed statistics information. Scroll left and right to navigate through the available statistics (ping time, disk free, queue size, build rate, total build time, and total build size). Most statistics are displayed as scrolling text, and a background fill representing the information graphically. Scroll down to return to the main screen.

Scroll the cursor off the bottom of the display to go to the quit and terminate options (scroll left and right to navigate between them). Press the joystick in to activate either option, or scroll up to return to the main screen.

CHAPTER 6

piw-initdb

The piw-initdb script is used to initialize or upgrade the piwheels master database. The target PostgreSQL[6] database must already exist, and the DSN should connect as a cluster superuser (e.g. the postgres user), in contrast to the piw-master script which should *not* use the cluster superuser. The script will prompt before making any permanent alterations, and all actions will be executed within a single transaction so that in the event of failure the database will be left unchanged. Nonetheless, it is strongly recommended you take a backup of your database before using this script for upgrades.

6.1 Synopsis

```
piw-initdb [-h] [--version] [-c FILE] [-q] [-v] [-l FILE] [-d DSN]
           [-u NAME] [-y]
```

6.2 Description

-h, --help
show this help message and exit

--version
show program's version number and exit

-c FILE, **--configuration** FILE
Specify a configuration file to load

-q, --quiet
produce less console output

-v, --verbose
produce more console output

-l FILE, **--log-file** FILE
log messages to the specified file

[6] https://postgresql.org/

-d DSN, **--dsn** DSN
: The database to create or upgrade; this DSN must connect as the cluster superuser (default: postgres:///piwheels)

-u NAME, **--user** NAME
: The name of the ordinary piwheels database user (default: piwheels); this must *not* be a cluster superuser

-y, **--yes**
: Proceed without prompting before init/upgrades

6.3 Usage

This script is intended to be used after installation to initialize the piwheels master database. Note that it does *not* create the database or the users for the database. It merely creates the tables, views, and other structures within an already existing database. See the *Overview* (page 1) chapter for typical usage.

The script can also be used to upgrade an existing piwheels database to the latest version. The update scripts used attempt to preserve all data, and all upgrades are performed in a single transaction so that, theoretically, if anything goes wrong the database should be rolled back to its original state. However, it is still strongly recommended that you back up your master database before proceeding with any upgrade.

CHAPTER 7

piw-import

The piw-import script is used to inject the specified file(s) manually into the piwheels database and file-system. This script must be run on the same node as the piw-master script. If multiple files are specified, they are registered as produced by a *single* build.

7.1 Synopsis

```
piw-import [-h] [--version] [-c FILE] [-q] [-v] [-l FILE]
           [--package PACKAGE] [--package-version VERSION] [--abi ABI]
           [--duration DURATION] [--output FILE] [-y] [-d]
           [--import-queue ADDR]
           files [files ...]
```

7.2 Description

-h, --help
Show this help message and exit

--version
Show program's version number and exit

-c FILE, **--configuration** FILE
Specify a configuration file to load

-q, --quiet
Produce less console output

-v, --verbose
Produce more console output

-l FILE, **--log-file** FILE
Log messages to the specified file

--package PACKAGE
The name of the package to import; if omitted this will be derived from the file(s) specified

--package-version VERSION
The version of the package to import; if omitted this will be derived from the file(s) specified

--abi ABI
The ABI of the package to import; if omitted this will be derived from the file(s) specified

--duration DURATION
The time taken to build the package (default: 0s)

--output FILE
The filename containing the build output to insert into the database; if this is omitted an appropriate message will be inserted instead

-y, --yes
Run non-interactively; never prompt during operation

-d, --delete
Remove the specified file(s) after a successful import; if the import fails, no files will be removed

--import-queue ADDR
The address of the queue used by **piw-import** (default: (ipc:///tmp/piw-import); this should always be an ipc address

7.3 Usage

This utility is used to import wheels manually into the system. This is useful with packages which have no source available on PyPI, or binary-only packages from third parties. If invoked with multiple files, all files will be associated with a single "build" and the build will be for the package and version of the first file specified. No checks are made for equality of package name or version (as several packages on PyPI would violate such a rule!).

The utility can be run in a batch mode with *--yes* (page 16) but still requires invoking once per build required (you cannot register multiple builds in a single invocation).

The return code will be 0 if the build was registered and all files were uploaded successfully. Additionally the *--delete* (page 16) option can be specified to remove the source files once all uploads are completed successfully. If anything fails, the return code will be non-zero and no files will be deleted.

The utility should only ever be run directly on the master node (opening the import queue to other machines is a potential security risk).

CHAPTER 8

piw-rebuild

The piw-rebuild script is used to inject rebuild requests for various web pages into the piwheels system. This script must be run on the same node as the piw-master script.

8.1 Synopsis

```
piw-rebuild [-h] [--version] [-c FILE] [-q] [-v] [-l FILE] [-y]
            [--import-queue ADDR]
            part [package]
```

8.2 Description

-h, --help
Show this help message and exit

--version
Show program's version number and exit

-c FILE, **--configuration** FILE
Specify a configuration file to load

-q, --quiet
Produce less console output

-v, --verbose
Produce more console output

-l FILE, **--log-file** FILE
Log messages to the specified file

-y, --yes
Run non-interactively; never prompt during operation

--import-queue ADDR
The address of the queue used by **piw-rebuild** (default: (ipc:///tmp/piw-import); this should always be an ipc address

8.3 Usage

This utility is used to request rebuilds of parts of the piwheels website. This is primarily useful after manual fixes to the database, manipulation of the file-system, or after large-scale upgrades which require rebuilding many pages.

The mandatory *part* parameter can be one of the following values, which specify which part of the website to rebuild:

Part	Description
home	Rebuild the home-page (/index.html)
search	Rebuild the JSON search-index (/packages.json)
project	Rebuild the project-page for the specified package (/project/*package*/index.html)
index	Rebuild the simple-index *and* the project-page for the specified package (/simple/*package*/index.html *and* /project/*package*/index.html)

If *part* is "project" or "index" you may optionally specify a *package* name for which to rebuild the specified part. If the *package* name is omitted, the utility will request a rebuild of the specified part for **all** known packages in the system.

Warning: In the case a rebuild of **all** packages is requested, you will be prompted to make sure you wish to continue (this option can take hours to process on a system with many builds). The `--yes` (page 17) option can be used to skip this prompt but should be used carefully!

Note that the utility only requests the rebuild of the specified part. This request will be queued, and acted upon as soon as *The Scribe* (page 26) reaches it but there is no guarantee this has occurred by the time the utility exits. The return code will be 0 if the rebuild request was queued successfully. If anything fails the return code will be non-zero and the request may or may not have been queued.

The utility should only ever be run directly on the master node (opening the import queue to other machines is a potential security risk).

CHAPTER 9

piw-remove

The piw-remove script is used to manually remove a version of a package from the system. All builds for the specified version will be forgotten, all files generated by such builds will be deleted, and all logged downloads will be deleted too.

By default, the version removed will *not* be marked to skip. Hence, after a short while the master is likely to attempt to re-build it. What happens at this point depends on several factors:

- If the version is still available on PyPI, and the build dependencies on the chosen slave are sufficient, it will (potentially) build successfully and re-appear on the system.
- If the version has been removed from PyPI (which is a reason to remove it from piwheels), the build will fail. The failed build will be logged in the system and will not be attempted again.

9.1 Synopsis

```
piw-remove [-h] [--version] [-c FILE] [-q] [-v] [-l FILE] [-y]
           [-s REASON] [--import-queue ADDR]
           package version
```

9.2 Description

package
: The name of the package to remove

version
: The version of the package to remove

-h, --help
: Show this help message and exit

--version
: Show program's version number and exit

-c FILE, **--configuration** FILE
: Specify a configuration file to load

-q, --quiet
Produce less console output

-v, --verbose
Produce more console output

-l FILE, **--log-file** FILE
Log messages to the specified file

-y, --yes
Run non-interactively; never prompt during operation

-s REASON, **--skip** REASON
Mark the version with a reason to prevent future build attempts

--import-queue ADDR
The address of the queue used by piw-remove (default: (ipc:///tmp/piw-import); this should always be an ipc address

9.3 Usage

This utility is typically used in response to a request from a package maintainer to remove a specific build from the system. Either because it has been withdrawn from PyPI itself, or because the presence of a piwheels build is causing issues in and of itself (both circumstances have occurred).

The utility can be run in a batch mode with *--yes* (page 20) but still requires invoking once per deletion required (you cannot remove multiple versions in a single invocation).

The return code will be 0 if the version was successfully removed. If anything fails, the return code will be non-zero and no files should be deleted (but this cannot be guaranteed in all circumstances).

The utility should only ever be run directly on the master node (opening the import queue to other machines is a potential security risk).

CHAPTER 10

piw-logger

The piw-logger script is intended for use as an Apache "piped log script" but can also be used to feed pre-existing Apache logs to the master by feeding logs to the script's stdin. This script must be run on the same node as the *piw-master* (page 3) script.

10.1 Synopsis

```
piw-logger [-h] [--version] [-c FILE] [-q] [-v] [-l FILE]
                [--format FORMAT] [--log-queue ADDR] [--drop]
                [files [files ...]]
```

10.2 Description

files
: The log file(s) to load into the master; if omitted or "-" then stdin will be read which is the default for piped log usage

-h, --help
: Show this help message and exit

--version
: Show program's version number and exit

-c FILE, **--configuration** FILE
: Specify a configuration file to load

-q, --quiet
: Produce less console output

-v, --verbose
: Produce more console output

-l FILE, **--log-file** FILE
: Log messages to the specified file

--format FORMAT
: The Apache log format that log lines will be expected to be in (default: combined); the short-cuts common, combined and common_vhost can be used in addition to Apache LogFormat strings

--log-queue ADDR
: The address of the queue used by piw-logger (default: (ipc:///tmp/piw-logger); this should always be an ipc address

--drop
: Drop log records if unable to send them to the master after a short timeout; this should generally be specified when **piw-logger** is used as a piped log[7] script

10.3 Usage

This utility is typically used to pipe logs from a web-server, such as Apache[8] into the piwheels database where they can be used for analysis, and to keep the stats on the homepage up to date. Apache provides a capability to pipe all logs to a given script which can be used directly with **piw-logger**.

A typical configuration under a Debian-like operating system might use the Apache CustomLog[9] directive as follows, within the Apache virtual host reponsible for serving files to `pip` clients:

```
ErrorLog ${APACHE_LOG_DIR}/ssl_error.log
CustomLog ${APACHE_LOG_DIR}/ssl_access.log combined
CustomLog "|/usr/local/bin/piw-logger --drop" combined
```

[7] http://httpd.apache.org/docs/current/logs.html#piped
[8] https://httpd.apache.org/
[9] http://httpd.apache.org/docs/current/mod/mod_log_config.html#customlog

CHAPTER 11

Development

The main GitHub repository for the project can be found at:

https://github.com/piwheels/piwheels

After cloning, we recommend you set up a virtualenv for development and then execute `make develop` within that virtualenv. This should install all requirements for executing all tools, building the documentation and executing the test suite.

11.1 Testing

Executing the test suite requires that you have a local PostgreSQL[10] installation configured with an unprivileged user, a privileged super user, and a test database.

The test suite uses environment variables to discover the name of the test database, and the aforementioned users. See the top of `tests/conftest.py` for more details. A typical execution of the test suite might look as follows:

```
$ export PIWHEELS_TESTDB=piwtest
$ export PIWHEELS_USER=piwheels
$ export PIWHEELS_PASS=piwheels
$ export PIWHEELS_SUPERUSER=piwsuper
$ export PIWHEELS_SUPERPASS=foobar
$ cd piwheels
$ make test
```

You may wish to construct a script for exporting the environment variables, or add these values to your `~/.bashrc`.

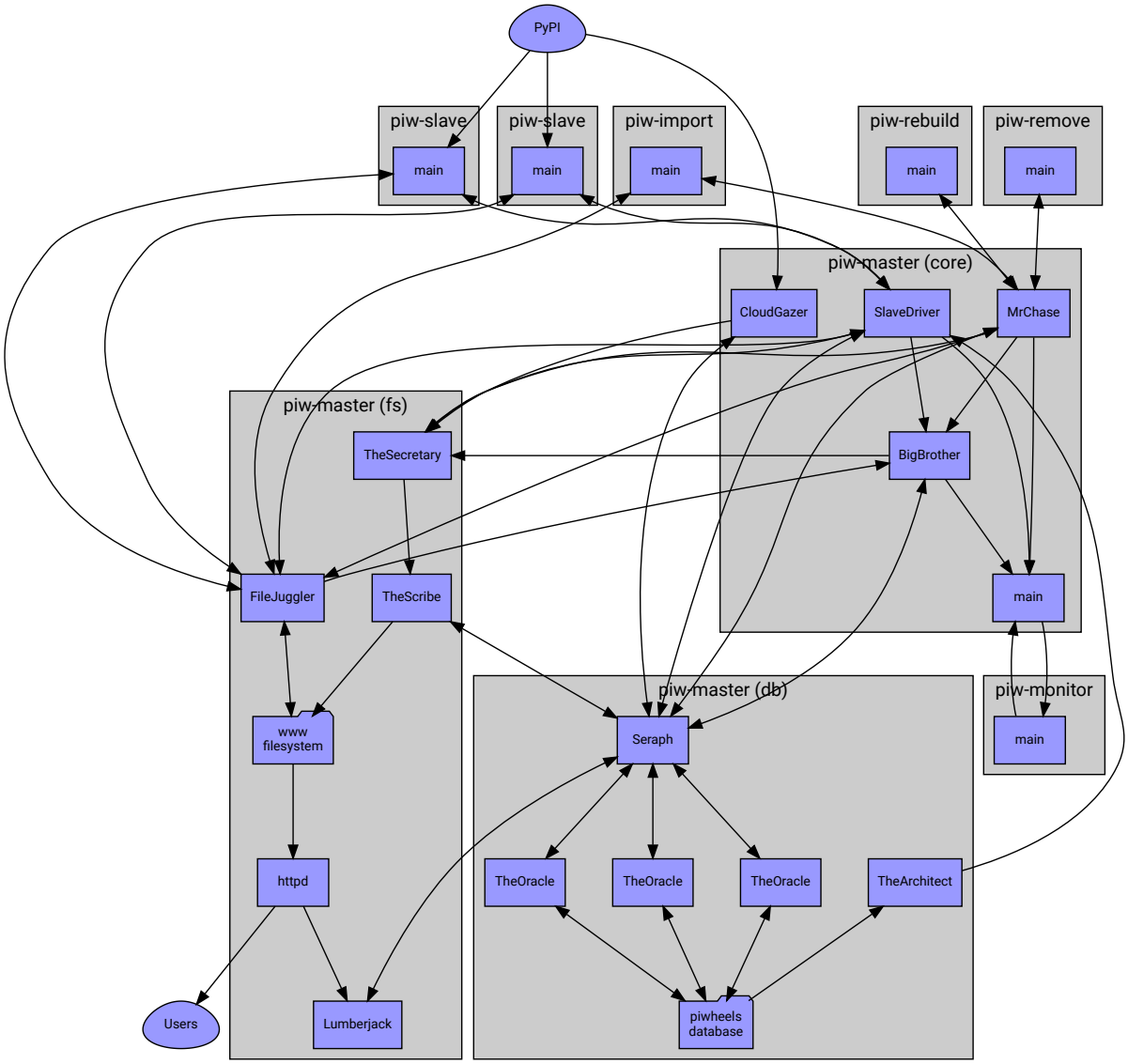
Note: If you are not using your local PostgreSQL installation for anything else you may wish to set `fsync=off` and `synchronous_commit=off` in your local `postgresql.conf` to speed up execution of the test suite. Do *NOT* do this on any production PostgreSQL server!

[10] https://postgresql.org/

11.2 Design

Although the piwheels master appears to be a monolithic script, it's actually composed of numerous (often extremely simple) tasks. Each task runs its own thread and all communication between tasks takes place over ZeroMQ[11] sockets. This is also how communication occurs between the master and the *piw-slave* (page 7), and the *piw-monitor* (page 9).

The following diagram roughly illustrates all the tasks in the system (including those of the build slaves and the monitor), along with details of the type of ZeroMQ socket used to communicate between them:



It may be confusing that the file server and database server appear to be separate to the master in the diagram. This is deliberate as the system's architecture is such that certain tasks can be easily broken off into entirely separate processes (potentially on separate machines), if required in future (either for performance or security reasons).

11.3 Tasks

The following sections document the tasks shown above (listed from the "front" at PyPI to the "back" at Users):

[11] https://zeromq.org/

11.3.1 Cloud Gazer

Implemented in: `piwheels.master.cloud_gazer.CloudGazer`.

This task is the "front" of the system. It follows PyPI's event log for new package and version registrations, and writes those entries to the database. It does this via *The Oracle* (page 25).

11.3.2 The Oracle

Implemented in: `piwheels.master.the_oracle.TheOracle`.

This task is the main interface to the database. It accepts requests from other tasks ("register this new package", "log this build", "what files were built with this package", etc.) and executes them against the database. Because database requests are extremely variable in their execution time, there are actually several instances of the oracle which sit behind *Seraph* (page 25).

11.3.3 Seraph

Implemented in: `piwheels.master.seraph.Seraph`.

Seraph is a simple load-balancer for the various instances of *The Oracle* (page 25). This is the task that *actually* accepts database requests. It finds a free oracle and passes the request along, passing back the reply when it's finished.

11.3.4 The Architect

Implemented in: `piwheels.master.the_architect.TheArchitect`.

This task is the final database related task in the master script. Unlike *The Oracle* (page 25) it simply queries the database for the packages that need building. Whenever *Slave Driver* (page 25) needs a task to hand to a build slave, it asks the Architect for one matching the build slave's ABI.

11.3.5 Slave Driver

Implemented in: `piwheels.master.slave_driver.SlaveDriver`.

This task is the main coordinator of the build slave's activities. When a build slave first comes online it introduces itself to this task (with information including the ABI it can build for), and asks for a package to build. As described above, this task asks *The Architect* (page 25) for the next package matching the build slave's ABI and passes this back.

Eventually the build slave will communicate whether or not the build succeeded, along with information about the build (log output, files generated, etc.). This task writes this information to the database via *The Oracle* (page 25). If the build was successful, it informs the *File Juggler* (page 26) that it should expect a file transfer from the relevant build slave.

Finally, when all files from the build have been transferred, the Slave Driver informs the *The Scribe* (page 26) that the package's index will need (re)writing.

11.3.6 Mr. Chase

Implemented in: `piwheels.master.mr_chase.MrChase`.

This task talks to **`piw-import`** and handles importing builds manually into the system. It is essentially a cut-down version of the *Slave Driver* (page 25) with a correspondingly simpler protocol.

This task writes information to the database via *The Oracle* (page 25). If the imported build was successful, it informs the *File Juggler* (page 26) that it should expect a file transfer from the importer.

Finally, when all files from the build have been transferred, it informs the *The Scribe* (page 26) that the package's index will need (re)writing.

11.3.7 File Juggler

Implemented in: `piwheels.master.file_juggler.FileJuggler`.

This task handles file transfers from the build slaves to the master. Files are transferred in multiple (relatively small) chunks and are verified with the hash reported by the build slave (retrieved from the database via *The Oracle* (page 25)).

11.3.8 Big Brother

Implemented in: `piwheels.master.big_brother.BigBrother`.

This task is a bit of a miscellaneous one. It sits around periodically generating statistics about the system as a whole (number of files, number of packages, number of successful builds, number of builds in the last hour, free disk space, etc.) and sends these off to the *The Scribe* (page 26).

11.3.9 The Scribe

Implemented in: `piwheels.master.the_scribe.TheScribe`.

This task generates the web output for piwheels. It generates the home-page with statistics from *Big Brother* (page 26), the overall package index, and individual package file lists with messages from *Slave Driver* (page 25).

11.3.10 The Secretary

Implemented in `piwheels.master.the_secretary.TheSecretary`.

This task sits in front of *The Scribe* (page 26) and attempts to mitigate many of the repeated requests that typically get sent to it. For example, project pages (which are relatively expensive to generate, in database terms), may need regenerating every time a file is registered against a package version.

This often happens in a burst when a new package version is released, resulting in several (frankly redundant) requests to re-write the same page with minimally changed information. The secretary buffers up such requests, eliminating duplicates before finally passing them to *The Scribe* (page 26) for processing.

11.4 Queues

It should be noted that the diagram omits several queues for the sake of brevity. For instance, there is a simple PUSH/PULL control queue between the master's "main" task and each sub-task which is used to relay control messages like `PAUSE`, `RESUME`, and `QUIT`.

Most of the protocols used by the queues are (currently) undocumented with the exception of those between the build slaves and the *Slave Driver* (page 25) and *File Juggler* (page 26) tasks (documented in the *piw-slave* (page 7) chapter).

However, all protocols share a common basis: messages are lists of Python objects. The first element is always string containing the action. Further elements are parameters specific to the action. Messages are encoded with CBOR[12].

[12] https://cbor.io/

11.5 Protocols

The following sections document the protocols used between the build slaves and the three sub-tasks that they talk to in the *piw-master* (page 3). Each protocol operates over a separate queue. All messages in the piwheels system follow a similar structure of being a tuple containing:

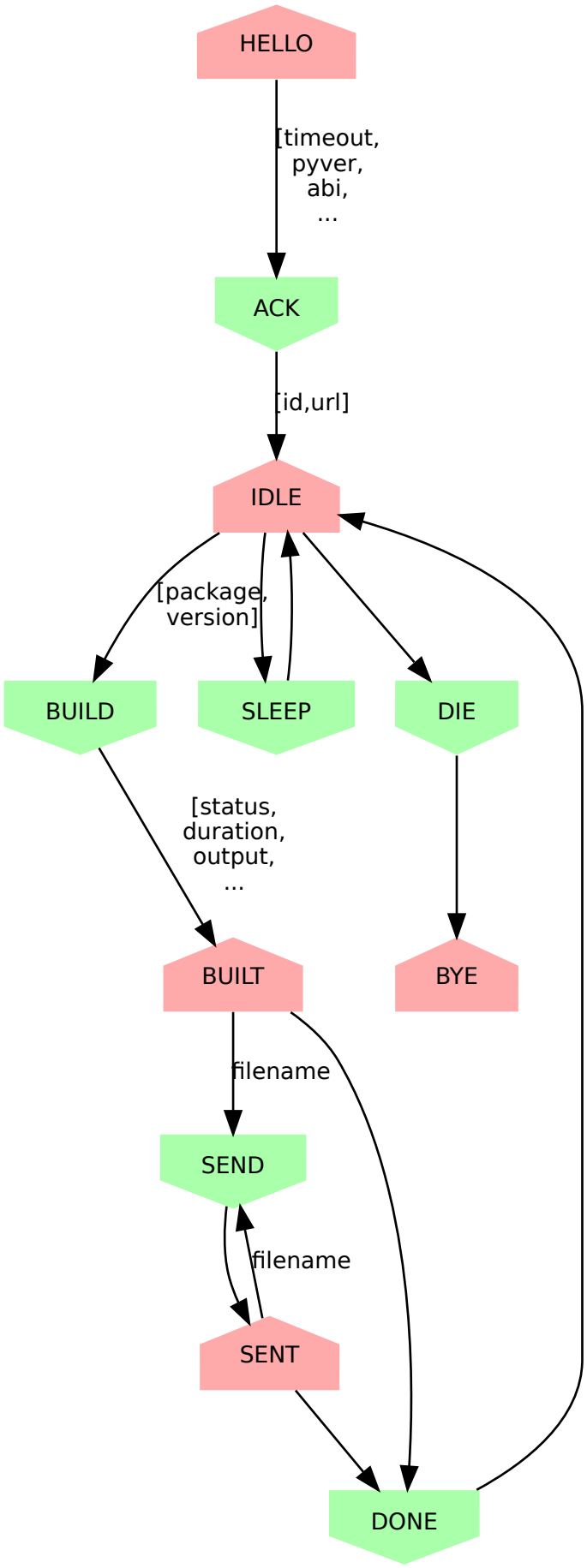
- A short unicode string indicating what sort of message it is.
- Data. The structure of the data is linked to the type of the message, and validated on both transmission and reception (see `piwheels.protocols` for more information).

If a message is not associated with any data whatsoever, it is transmitted as a simple unicode string (without the tuple encapsulation). The serialization format for all messages in the system is currently CBOR[13].

11.5.1 Slave Driver

The queue that talks to *Slave Driver* (page 25) is a ZeroMQ REQ socket, hence the protocol follows a strict request-reply sequence which is illustrated below:

[13] https://cbor.io/


HELLO
[timeout,
pyver,
abi,
...
ACK
[id,url]
IDLE
[package,
version]
BUILD
SLEEP
DIE
[status,
duration,
output,
...
BUILT
BYE
filename
SEND
filename
SENT
DONE

1. The new build slave sends "HELLO" with data `[timeout, py_version_tag, abi_tag, platform_tag, label]` where:
   - `timeout` is the slave's configured timeout (the length of time after which it will assume a build has failed and attempt to terminate it) as a `timedelta`[14].
   - `py_version_tag` is the python version the slave will build for (e.g. "27", "35", etc.)
   - `abi_tag` is the ABI the slave will build for (e.g. "cp35m")
   - `platform_tag` is the platform of the slave (e.g. "linux_armv7l")
   - `label` is an identifying label for the slave (e.g. "slave2"); note that this label doesn't have to be anything specific, it's purely a convenience for administrators displayed in the monitor. In the current implementation this is the unqualified hostname of the slave
2. The master replies sends "ACK" with data `[slave_id, pypi_url]` where *slave_id* is an integer identifier for the slave. Strictly speaking, the build slave doesn't need this identifier but it can be helpful for admins or developers to see the same identifier in logs on the master and the slave which is the only reason it is communicated.

   The *pypi_url* is the URL the slave should use to fetch packages from PyPI.
3. The build slave sends "IDLE" to indicate that it is ready to accept a build job.
4. The master can reply with "SLEEP" which indicates that no jobs are currently available for that slave (e.g. the master is paused, or the build queue is empty, or there are no builds for the slave's particular ABI at this time). In this case the build slave should pause a while (the current implementation waits 10 seconds) before retrying "IDLE".
5. The master can also reply wih "DIE" which indicates the build slave should shutdown. In this case, after cleaning up any resources the build slave should send back "BYE" and terminate (generally speaking, whenever the slave terminates it should send "BYE" no matter where in the protocol it occurs; the master will take this as a sign of termination).
6. The master can also reply "BUILD" with data `[package, version]` where *package* is the name of a package to build and *version* is the version to build. At this point, the build slave should attempt to locate the package on PyPI and build a wheel from it.
7. Whatever the outcome of the build, the slave sends "BUILT" with data `[status, duration, output, files]`:
   - *status* is `True` if the build succeeded and `False` otherwise.
   - *duration* is a `timedelta`[15] value indicating the length of time it took to build in seconds.
   - *output* is a string containing the complete build log.
   - *files* is a `list`[16] of file state tuples containing the following fields in the specified order:
     - *filename* is the filename of the wheel.
     - *filesize* is the size in bytes of the wheel.
     - *filehash* is the SHA256 hash of the wheel contents.
     - *package_tag* is the package tag extracted from the filename.
     - *package_version_tag* is the version tag extracted from the filename.
     - *py_version_tag* is the python version tag extracted from the filename.
     - *abi_tag* is the ABI tag extracted from the filename (sanitized).
     - *platform_tag* is the platform tag extracted from the filename.

[14] https://docs.python.org/3.5/library/datetime.html#datetime.timedelta
[15] https://docs.python.org/3.5/library/datetime.html#datetime.timedelta
[16] https://docs.python.org/3.5/library/stdtypes.html#list

  - *dependencies* is a `set`[17] of dependency tuples containing the following fields in the specified order:
    * *tool* is the name of the tool used to install the dependency
    * *package* is the name of the package to install with the tool

8. If the build succeeded, the master will send "SEND" with data `filename` where *filename* is one of the names transmitted in the prior "BUILT" message.
9. At this point the slave should use the *File Juggler* (page 33) protocol documented below to transmit the contents of the specified file to the master. When the file transfer is complete, the build slave sends "SENT" to the master.
10. If the file transfer fails to verify, or if there are more files to send the master will repeat the "SEND" message. Otherwise, if all transfers have completed and have been verified, the master replies with "DONE".
11. The build slave is now free to destroy all resources associated with the build, and returns to step 3 ("IDLE").

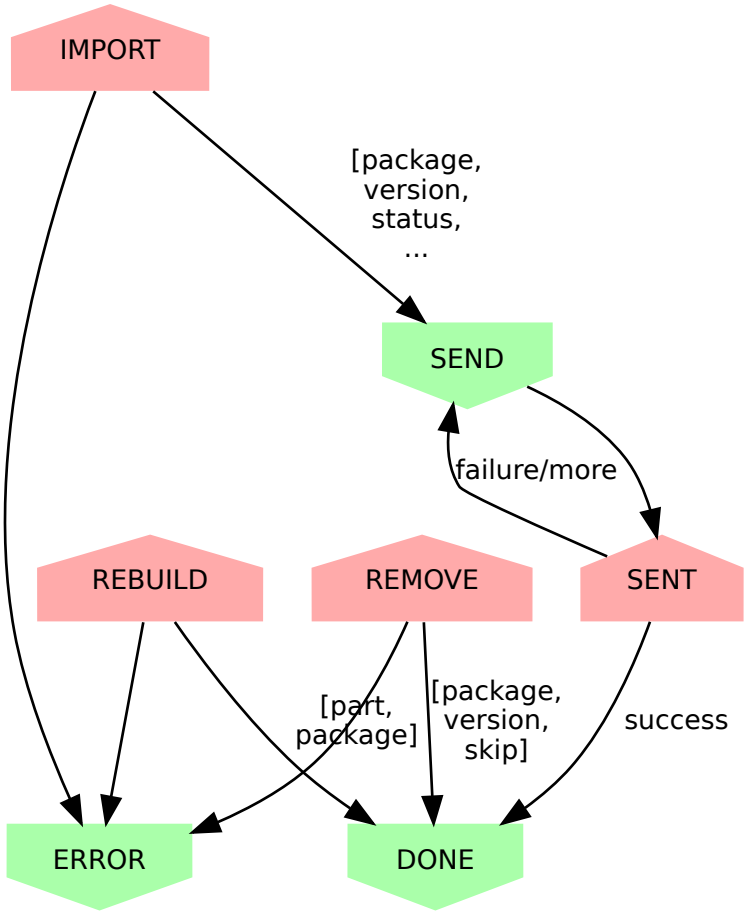
If at any point, the master takes more than 60 seconds to respond to a slave's request, the slave will assume the master has disappeared. If a build is still active, it will be cleaned up and terminated, the connection to the master will be closed, the slave's ID will be reset and the slave must restart the protocol from the top ("HELLO").

This permits the master to be upgraded or replaced without having to shutdown and restart the slaves manually. It is possible that the master is restarted too fast for the slave to notice. In this case the slave's next message will be mis-interpreted by the master as an invalid initial message, and it will be ignored. However, this is acceptable behaviour as the re-connection protocol described above will then effectively restart the slave after the 60 second timeout has elapsed.

11.5.2 Mr Chase (importing)

The queue that talks to *Mr. Chase* (page 25) is a ZeroMQ REQ socket, hence the protocol follows a strict request-reply sequence which is illustrated below (see below for documentation of the "REMOVE" path):

[17] https://docs.python.org/3.5/library/stdtypes.html#set



1. The importer sends "IMPORT" with data `[abi_tag, package, version, status, duration, output, files]`:
   - *abi_tag* is either `None`, indicating that the master should use the "default" (minimum) build ABI registered in the system, or is a string indicating the ABI that the build was attempted for.
   - *package* is the name of the package that the build is for.
   - *version* is the version of the package that the build is for.
   - *status* is `True` if the build succeeded and `False` otherwise.
   - *duration* is a `float`[18] value indicating the length of time it took to build in seconds.
   - *output* is a string containing the complete build log.
   - *files* is a list of file state tuples containing the following fields in the specified order:
     - *filename* is the filename of the wheel.
     - *filesize* is the size in bytes of the wheel.
     - *filehash* is the SHA256 hash of the wheel contents.
     - *package_tag* is the package tag extracted from the filename.
     - *package_version_tag* is the version tag extracted from the filename.
     - *py_version_tag* is the python version tag extracted from the filename.
     - *abi_tag* is the ABI tag extracted from the filename (sanitized).
     - *platform_tag* is the platform tag extracted from the filename.

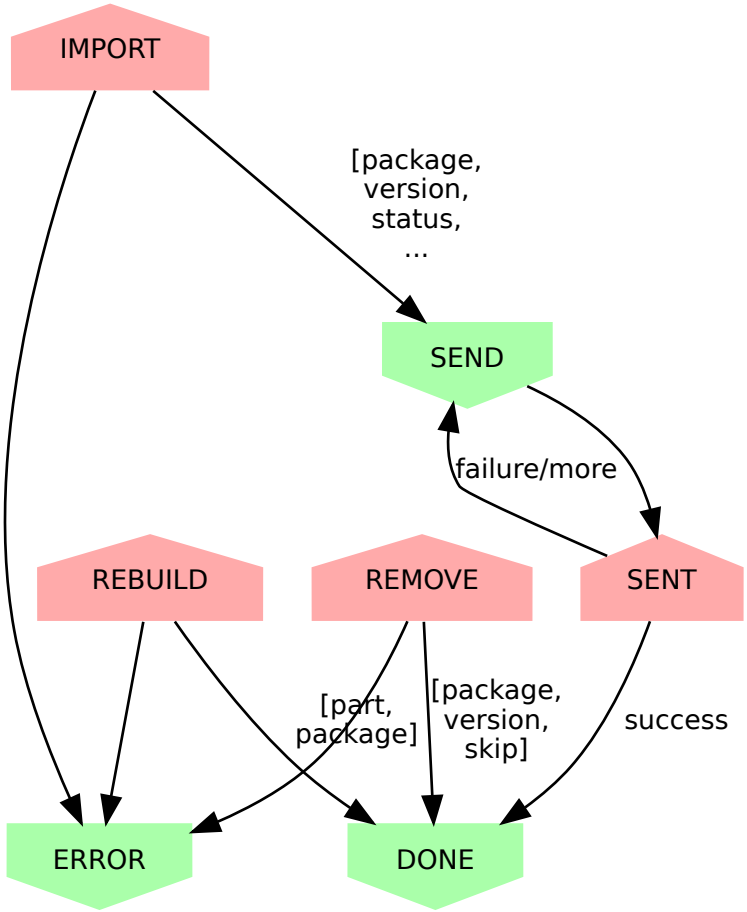
[18] https://docs.python.org/3.5/library/functions.html#float

  - *dependencies* is a `set`[19] of dependency tuples containing the following fields in the specified order:
    * *tool* is the name of the tool used to install the dependency
    * *package* is the name of the package to install with the tool

2. If the import information is insufficient or incorrect, the master will send "ERROR" with data `message` which is the description of the error that occurred.
3. If the import information is okay, the master will send "SEND" with data `filename` for each file mentioned in the build.
4. At this point the importer should use the *File Juggler* (page 33) protocol to transmit the contents of the specified file to the master. When the file transfer is complete, the importer sends "SENT" to the master.
5. If the file transfer fails to verify, or if there are more files to send the master will repeat the "SEND" message. Otherwise, if all transfers have completed and have been verified, the master replies with "DONE".
6. The importer is now free to remove all files associated with the build, if requested to.

11.5.3 Mr Chase (removing)

The queue that talks to *Mr. Chase* (page 25) is a ZeroMQ REQ socket, hence the protocol follows a strict request-reply sequence which is illustrated below (see above for documentation of the `IMPORT` path):



1. The utility sends "REMOVE" with data `[package, version, skip]`:
   - *package* is the name of the package to remove.
   - *version* is the version of the package to remove.

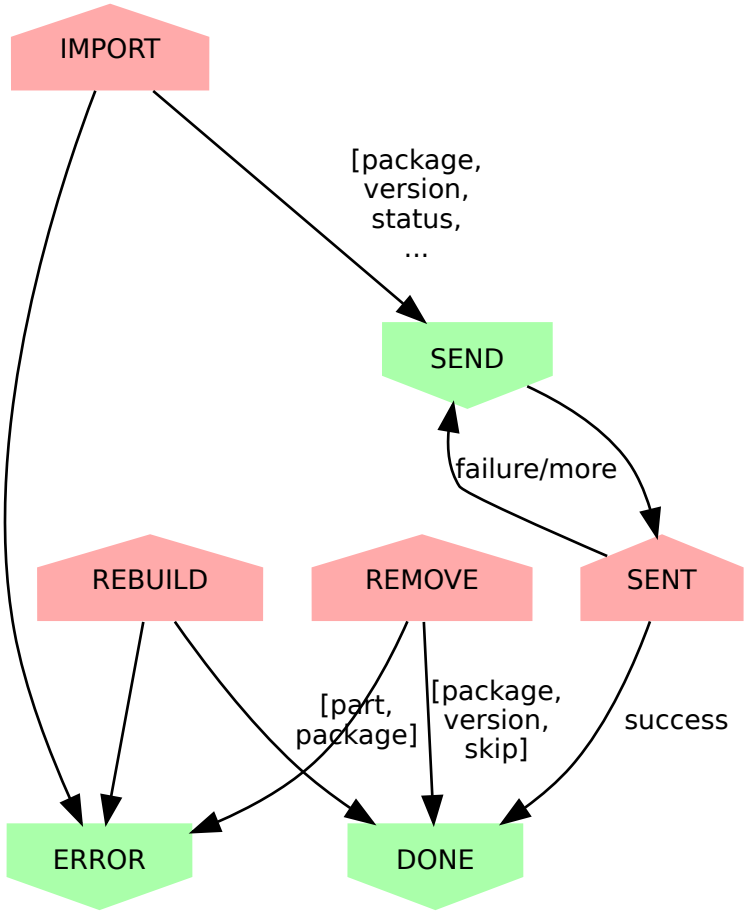
[19] https://docs.python.org/3.5/library/stdtypes.html#set

- *skip* is a string containing the reason the version should never be built again, or is a blank string indicating the version should be rebuilt.

2. If the removal fails (e.g. if the package or version does not exist), the master will send "ERROR" with data `message` (a string describing the error that occurred).
3. If the removal is successful, the master replies with "DONE".

11.5.4 Mr Chase (rebuilding)

The queue that talks to *Mr. Chase* (page 25) is a ZeroMQ REQ socket, hence the protocol follows a strict request-reply sequence which is illustrated below (see above for documentation of the `IMPORT` path):

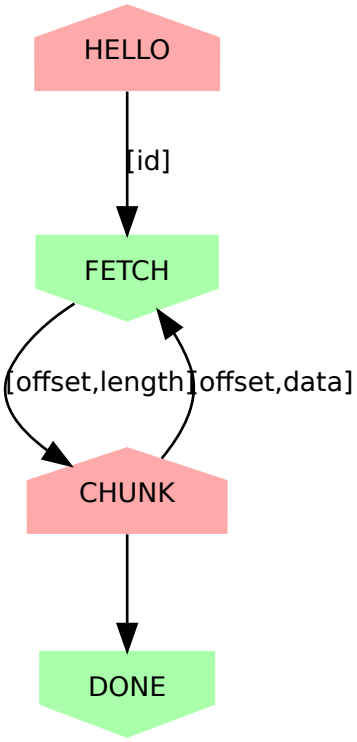


1. The utility sends "REBUILD" with data `[part, package]`:
   - *part* is the part of the website to rebuild. It must be one of "HOME", "SEARCH", "PKGPROJ" or "PKGBOTH".
   - *package* is the name of the package to rebuild indexes and/or project pages for or `None` if pages for all packages should be rebuilt. This parameter is omitted if *part* is "HOME" or "SEARCH".
2. If the rebuild request fails (e.g. if the package does not exist), the master will send "ERROR" with data `message` (a string describing the error that occurred).
3. If the rebuild request is successful, the master replies with "DONE".

11.5.5 File Juggler

The queue that talks to *File Juggler* (page 26) is a ZeroMQ DEALER socket. This is because the protocol is semi-asynchronous (for performance reasons). For the sake of illustration, a synchronous version of the protocol

is illustrated below:



1. The build slave initially sends "HELLO" with data `slave_id` where *slave_id* is the integer identifier of the slave. The master knows what file it requested from this slave (with "SEND" to the Slave Driver), and knows the file hash it is expecting from the "BUILT" message.
2. The master replies with "FETCH" with data `[offset, length]` where *offset* is a byte offset into the file, and *length* is the number of bytes to send.
3. The build slave replies with "CHUNK" with `data` where *data* is a byte-string containing the requested bytes from the file.
4. The master now either replies with another "FETCH" message or, when it has all chunks successfully received, replies with "DONE" indicating the build slave can now close the file (though it can't delete it yet; see the "DONE" message on the Slave Driver side for that).

"FETCH" messages may be repeated if the master drops packets (due to an overloaded queue). Furthermore, because the protocol is semi-asynchronous multiple "FETCH" messages will be sent before the master waits for any returning "CHUNK" messages.

11.6 Security

Care must be taken when running the build slave. Building all packages in PyPI effectively invites the denizens of the Internet to run arbitrary code on your machine. For this reason, the following steps are recommended:

1. Never run the build slave on the master; ensure they are entirely separate machines.
2. Run the build slave as an unprivileged user which has access to nothing it doesn't absolutely require (it shouldn't have any access to the master's file-system, the master's database, etc.)
3. Install the build slave's code in a location the build slave's unprivileged user does not have write access (i.e. *not* in a virtualenv under the user's home dir).
4. Consider whether to make the unprivileged user's home-directory read-only.

We have experimented with read-only home directories, but a significant portion of (usually scientifically oriented) packages attempt to be "friendly" and either write data to the user's home directory or modify the user's profile

(`~/.bashrc` and so forth).

The quandry is whether it is better to fail with such packages (a read-only home-directory will most likely crash such setup scripts, failing the build), or partially support them (leaving the home-directory writeable even though the modifications on the build-slave won't be recorded in the resulting wheel and thus won't be replicated on user's machines). There is probably no universally good answer.

Currently, while the build slave cleans up the temporary directory used by pip during wheel building, it doesn't attempt to clean its own home directory (which setup scripts are free to write to). This is something that ought to be addressed in future as it's a potentially exploitable hole.

CHAPTER 12

Module Reference

This chapter contains all the documentation auto-generated from the source code. It is probably not terribly useful for reading through, but may be useful as a searchable reference.

12.1 piwheels.master

12.2 piwheels.master.db

12.3 piwheels.master.cloud_gazer

12.4 piwheels.master.the_oracle

12.5 piwheels.master.seraph

12.6 piwheels.master.the_architect

12.7 piwheels.master.slave_driver

12.8 piwheels.master.mr_chase

12.9 piwheels.master.file_juggler

12.10 piwheels.master.big_brother

12.11 piwheels.master.the_secretary

12.12 piwheels.master.the_scribe

12.13 piwheels.slave

12.14 piwheels.slave.builder

12.15 piwheels.initdb

Contains the functions that make up the **piw-initdb** script.

piwheels.initdb.**main**(*args=None*)
: This is the main function for the **piw-initdb** script. It creates the piwheels database required by the master or, if it already exists, upgrades it to the current version of the application.

piwheels.initdb.**detect_users**(*conn*, *test_user*)
: Test that the user for *conn* is a cluster superuser (so we can drop and create anything we want in the database), and that *test_user* (which will be granted limited rights to various objects for the purposes of the **piw-master** script) exists and is *not* a cluster superuser.

piwheels.initdb.**detect_version**(*conn*)
: Detect the version of the database. This is typically done by reading the contents of the `configuration` table, but before that was added we can guess a couple of versions based on what tables exist (or don't). Returns `None` if the database appears uninitialized, and raises `RuntimeError`[20] is the version is so ancient we can't do anything with it.

[20] https://docs.python.org/3.5/library/exceptions.html#RuntimeError

piwheels.initdb.**get_connection**(*dsn*)
Return an SQLAlchemy connection to the specified *dsn* or raise RuntimeError[21] if the database doesn't exist (the administrator is expected to create the database before running this script).

piwheels.initdb.**get_script**(*version=None*)
Generate the script to get the database from *version* (the result of *detect_version()* (page 38)) to the current version of the software. If *version* is None, this is simply the contents of the sql/create_piwheels.sql script. Otherwise, it is a concatenation of various update scripts.

piwheels.initdb.**parse_statements**(*script*)
This is an extremely crude statement splitter for PostgreSQL's dialect of SQL. It understands --comments, "quoted identifiers", 'string literals' and $delim$ extended strings $delim$, but not E'\escaped strings' or /* C-style comments */. If you start using such things in the update scripts, you'll need to extend this function to accommodate them.

It returns a generator which yields individiual statements from *script*, delimited by semi-colon terminators.

12.16 piwheels.importer

12.17 piwheels.remove

12.18 piwheels.transport

12.19 piwheels.protocols

12.20 piwheels.tasks

12.21 piwheels.states

This module defines several classes which permit interested tasks to track the state of build slaves (*SlaveState* (page 41)), file transfers (*TransferState* (page 41)), build attempts (*BuildState* (page 40)), build artifacts (*FileState* (page 39)) and various loggers.

class piwheels.states.**FileState**(*filename*, *filesize*, *filehash*, *package_tag*, *package_version_tag*, *py_version_tag*, *abi_tag*, *platform_tag*, *dependencies*, *transferred=False*)
Represents the state of an individual build artifact (a package file, or wheel) including its filename, filesize, the SHA256 filehash, and various tags extracted from the build. Also tracks whether or not the file has been transferred.

Parameters

- **filename** (*str*[22]) – The original filename of the build artifact.
- **filesize** (*int*[23]) – The size of the file in bytes.
- **filehash** (*str*[24]) – The SHA256 hash of the file contents.
- **package_tag** (*str*[25]) – The package tag extracted from the filename (first "-" separated component).

[21] https://docs.python.org/3.5/library/exceptions.html#RuntimeError
[22] https://docs.python.org/3.5/library/stdtypes.html#str
[23] https://docs.python.org/3.5/library/functions.html#int
[24] https://docs.python.org/3.5/library/stdtypes.html#str
[25] https://docs.python.org/3.5/library/stdtypes.html#str

- **package_version_tag** (*str*[26]) – The package version tag extracted from the filename (second "-" separated component).
- **py_version_tag** (*str*[27]) – The python version tag extracted from the filename (third from last "-" separated component).
- **abi_tag** (*str*[28]) – The python ABI tag extracted from the filename (second from last "-" separated component).
- **platform_tag** (*str*[29]) – The platform tag extracted from the filename (last "-" separated component).
- **dependencies** (*set*[30]) – The set of dependencies that are required to use this particular wheel.
- **transferred** (*bool*[31]) – `True` if the file has been transferred from the build slave that generated it to the file server.

as_message()
Convert the *FileState* (page 39) object into a simpler list for serialization and transport.

classmethod from_message(*value*)
Convert the output from *as_message()* (page 40) back into a *BuildState* (page 40).

verified()
Called to set `transferred` to `True` after a file transfer has been successfully verified.

class piwheels.states.**BuildState**(*slave_id*, *package*, *version*, *abi_tag*, *status*, *duration*, *output*, *files*, *build_id=None*)
Represents the state of a package build including the `package`, `version`, `status`, build `duration`, and all the lines of `output`. The *files* (page 41) attribute is a mapping containing details of each successfully built package file.

Parameters

- **slave_id** (*int*[32]) – The master's identifier for the build slave.
- **package** (*str*[33]) – The name of the package to build.
- **version** (*str*[34]) – The version number of the package to build.
- **abi_tag** (*str*[35]) – The ABI for which the build was attempted (must not be `'none'`).
- **status** (*bool*[36]) – `True` if the build succeeded, `False` if it failed.
- **duration** (*timedelta*) – The amount of time (in seconds) it took to complete the build.
- **output** (*str*[37]) – The log output of the build.
- **files** (*dict*[38]) – A mapping of filenames to *FileState* (page 39) objects for each artifact produced by the build.
- **build_id** (*int*[39]) – The integer identifier generated for the build by the database (`None` until the build has been inserted into the database).

[26] https://docs.python.org/3.5/library/stdtypes.html#str
[27] https://docs.python.org/3.5/library/stdtypes.html#str
[28] https://docs.python.org/3.5/library/stdtypes.html#str
[29] https://docs.python.org/3.5/library/stdtypes.html#str
[30] https://docs.python.org/3.5/library/stdtypes.html#set
[31] https://docs.python.org/3.5/library/functions.html#bool
[32] https://docs.python.org/3.5/library/functions.html#int
[33] https://docs.python.org/3.5/library/stdtypes.html#str
[34] https://docs.python.org/3.5/library/stdtypes.html#str
[35] https://docs.python.org/3.5/library/stdtypes.html#str
[36] https://docs.python.org/3.5/library/functions.html#bool
[37] https://docs.python.org/3.5/library/stdtypes.html#str
[38] https://docs.python.org/3.5/library/stdtypes.html#dict
[39] https://docs.python.org/3.5/library/functions.html#int

as_message()
Convert the *BuildState* (page 40), and its nested *FileState* (page 39) objects into simpler lists for serialization and transport.

classmethod from_message(*value*)
Convert the output from *as_message()* (page 41) back into a *BuildState* (page 40).

logged(*build_id*)
Called to fill in the build's ID in the backend database.

files
A mapping of filename to *FileState* (page 39) instances.

next_file
Returns the filename of the next file that needs transferring or `None` if all files have been transferred.

transfers_done
Returns `True` if all files have been transferred.

class piwheels.states.**SlaveState**(*address*, *timeout*, *native_py_version*, *native_abi*, *native_platform*, *label*)
Tracks the state of a build slave. The master updates this state which each request and reply sent to and received from the slave, and this class in turn manages the associated *BuildState* (page 40) (accessible from `build`) and *TransferState* (page 41) (accessible from `transfer`). The class also tracks the time a request was last seen from the build slave, and includes a `kill()` method.

class piwheels.states.**TransferState**(*slave_id*, *file_state*)
Tracks the state of a file transfer. All file transfers are held in temporary locations until `verify()` indicates the transfer was successful, at which point they are atomically renamed into their final location.

The state is intimately tied to the file transfer protocol and includes methods to write a recevied `chunk()`, and to determine the next chunk to `fetch()`, as well as a property to determine when the transfer is `done`.

class piwheels.states.**DownloadState**

class piwheels.states.**SearchState**

class piwheels.states.**ProjectState**

class piwheels.states.**JSONState**

class piwheels.states.**PageState**

12.22 piwheels.ranges

A set of utility routines for efficiently tracking byte ranges within a stream. These are used to track which chunks of a file have been received during file transfers from build slaves.

See `FileJuggler` for the usage of these functions.

piwheels.ranges.**consolidate**(*ranges*)
Given a list of *ranges* in ascending order, this generator function returns the list with any overlapping ranges consolidated into individual ranges. For example:

```
>>> list(consolidate([range(0, 5), range(4, 10)]))
[range(0, 10)]
>>> list(consolidate([range(0, 5), range(5, 10)]))
[range(0, 10)]
>>> list(consolidate([range(0, 5), range(6, 10)]))
[range(0, 5), range(6, 10)]
```

piwheels.ranges.**exclude**(*ranges*, *ex*)
Given a list of non-overlapping *ranges* in ascending order, and a range *ex* to exclude, this generator function returns *ranges* with all values covered by *ex* removed from any contained ranges. For example:

```
>>> list(exclude([range(10)], range(2)))
[range(2, 10)]
>>> list(exclude([range(10)], range(2, 4)))
[range(0, 2), range(4, 10)]
```

piwheels.ranges.**intersect**(*range1*, *range2*)

Given two ranges *range1* and *range2* (which must both have a step of 1), returns the range formed by the intersection of the two ranges, or `None` if the ranges do not overlap. For example:

```
>>> intersect(range(10), range(5))
range(0, 5)
>>> intersect(range(10), range(10, 2))
>>> intersect(range(10), range(2, 5))
range(2, 5)
```

piwheels.ranges.**split**(*ranges*, *i*)

Given a list of non-overlapping *ranges* in ascending order, this generator function returns the list with the range containing *i* split into two ranges, one ending at *i* and the other starting at *i*. If *i* is not contained in any of the ranges, then *ranges* is returned unchanged. For example:

```
>>> list(split([range(10)], 5))
[range(0, 5), range(5, 10)]
>>> list(split([range(10)], 0))
[range(0, 10)]
>>> list(split([range(10)], 20))
[range(0, 10)]
```

CHAPTER 13

License

[40] https://github.com/bennuttall
[41] dave@waveform.org.uk

Python Module Index

p

Index

Symbols

S

T

V