Piwheels 0.10 Documentation

Release 0.10

Ben Nuttall

Jan 12, 2018

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 9)	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 15)	Provides a friendly curses-based UI for interacting with the piwheels master.
piw-initdb (page 17)	A simple maintenance script for initializing or upgrading the database to the current version.
piw-import (page 19)	A tool for importing wheels manually into the piwheels database and file-system.
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.

1.1 Deployment

A typical deployment of the master service on a Raspbian server goes something like this (all chunks assume you start as root):

1. Install the pre-requisite software:

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

```
# apt install postgresql-9.6 apache2 python3-psycopg2 python3-geoip
# apt install python3-sqlalchemy python3-urwid python3-zmq
# pip install piwheels[monitor,master,log]
```

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[3]; the dehydrated[4] 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 *on separate machines*:

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

7. Start the build slave running (assuming your master's IP address is 10.0.0.1):

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

1.2 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`** script as the postgres super-user to upgrade the database between versions (downgrades are not supported, so take a backup first!).

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

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]
           [--pypi-xmlrpc URL] [--pypi-simple URL] [-o PATH]
           [--index-queue ADDR] [--status-queue ADDR]
           [--control-queue ADDR] [--builds-queue ADDR]
           [--db-queue ADDR] [--fs-queue ADDR] [--slave-queue ADDR]
           [--file-queue ADDR] [--import-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 should *not* be a PostgreSQL superuser (default: postgres:///piwheels)

--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)

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

--index-queue ADDR
The address of the IndexScribe queue (default: inproc://indexes)

--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)

--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)

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

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

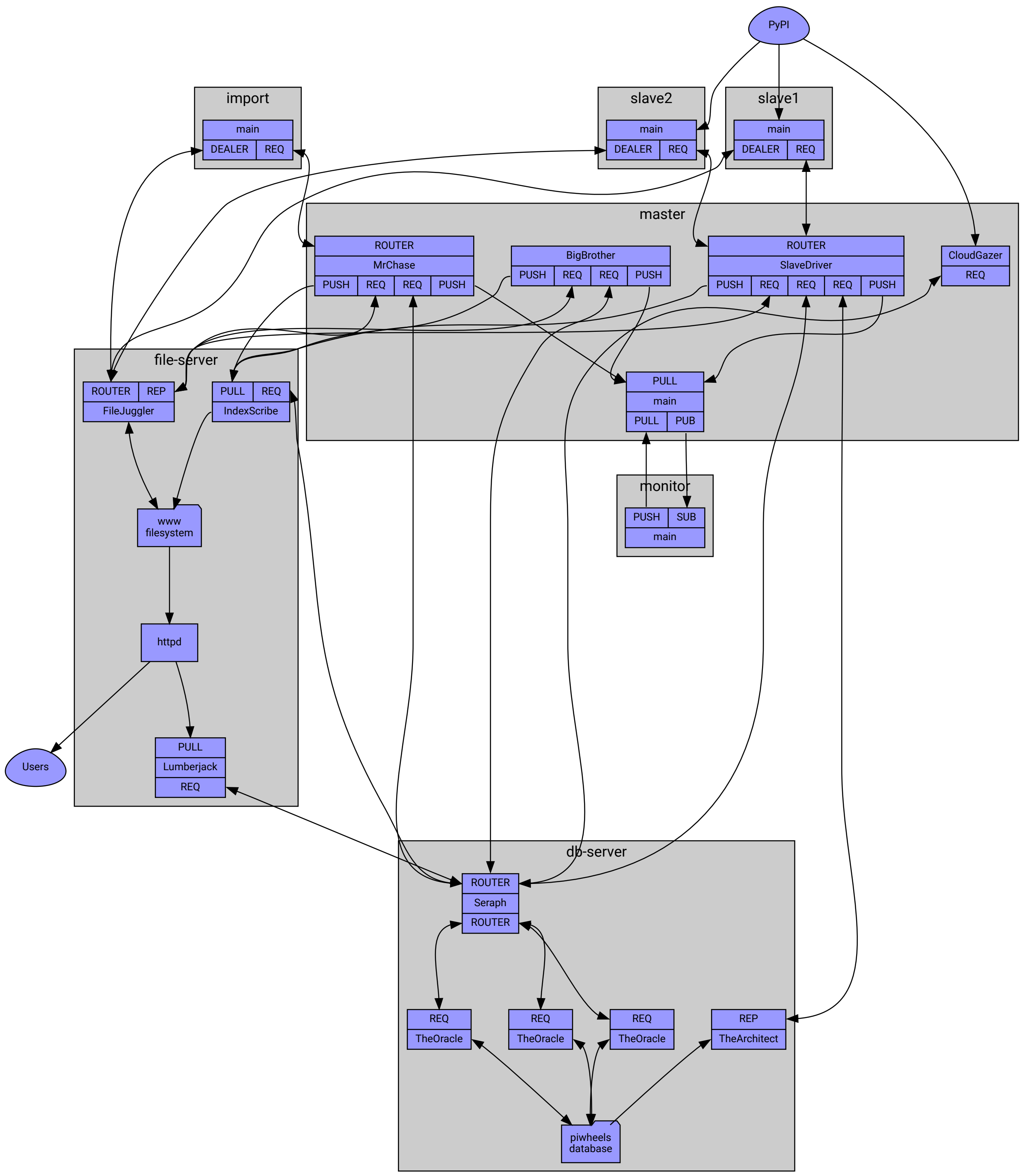
--import-queue ADDR
The address of the queue used by piw-import (default: (ipc:///tmp/piw-import); this should always be an ipc address

2.3 Development

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[5] sockets. This is also how communication occurs between the master and the *piw-slave* (page 9), and the *piw-monitor* (page 15).

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:

[5] https://zeromq.org/



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).

2.4 Tasks

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

2.4.1 Cloud Gazer

Implemented in: `piwheels.master.cloud_gazer.CloudGazer` (page 33).

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 6).

2.4.2 The Oracle

Implemented in: `piwheels.master.the_oracle.TheOracle` (page 33).

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 6).

2.4.3 Seraph

Implemented in: `piwheels.master.seraph.Seraph` (page 35).

Seraph is a simple load-balancer for the various instances of *The Oracle* (page 6). 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.

2.4.4 The Architect

Implemented in: `piwheels.master.the_architect.TheArchitect` (page 35).

This task is the final database related task in the master script. Unlike *The Oracle* (page 6) it simply queries the database for the packages that need building. Whenever *Slave Driver* (page 6) needs a task to hand to a build slave, it asks the Architect for one matching the build slave's ABI.

2.4.5 Slave Driver

Implemented in: `piwheels.master.slave_driver.SlaveDriver` (page 35).

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 6) 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 6). If the build was successful, it informs the *File Juggler* (page 7) 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 *Index Scribe* (page 7) that the package's index will need (re)writing.

2.4.6 Mr. Chase

Implemented in: `piwheels.master.mr_chase.MrChase` (page 37).

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 6) with a correspondingly simpler protocol.

This task writes information to the database via *The Oracle* (page 6). If the imported build was successful, it informs the *File Juggler* (page 7) that it should expect a file transfer from the importer.

Finally, when all files from the build have been transferred, it informs the *Index Scribe* (page 7) that the package's index will need (re)writing.

2.4.7 File Juggler

Implemented in: *`piwheels.master.file_juggler.FileJuggler`* (page 38).

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 6)).

2.4.8 Big Brother

Implemented in: *`piwheels.master.big_brother.BigBrother`* (page 39).

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 *Index Scribe* (page 7).

2.4.9 Index Scribe

Implemented in: *`piwheels.master.index_scribe.IndexScribe`* (page 39).

This task generates the web output for piwheels. It generates the home-page with statistics from *Big Brother* (page 7), the overall package index, and individual package file lists with messages from *Slave Driver* (page 6).

2.5 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 6) and *File Juggler* (page 7) tasks (documented in the *piw-slave* (page 9) 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 `pickle`[6]. This is an untrusted format but was the quickest to get started with (and the inter-process queues aren't exposed to the internet). A future version may switch to something slightly safer like JSON[7] or better still CBOR[8].

[6] https://docs.python.org/3.4/library/pickle.html#module-pickle
[7] https://www.json.org/
[8] https://cbor.io/

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

```
usage: 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)

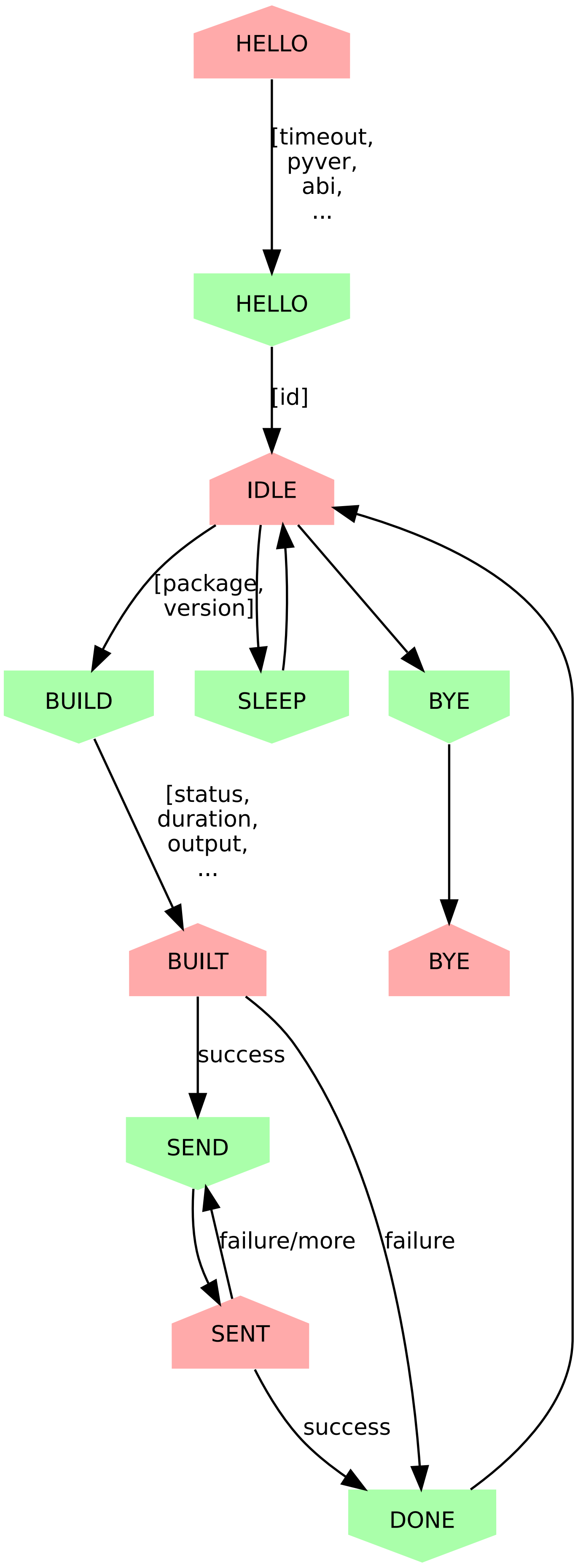
3.3 Protocols

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

1. Each message is a list of Python objects.
2. The first element in the list is a string indicating the type of message.
3. Additional elements depend on the type of the message.
4. A given message type always contains the same number of elements (there are no variable length messages).

3.3.1 Slave Driver

The queue that talks to *Slave Driver* (page 6) is a ZeroMQ REQ socket, hence the protocol follows a strict request-reply sequence which is illustrated below:

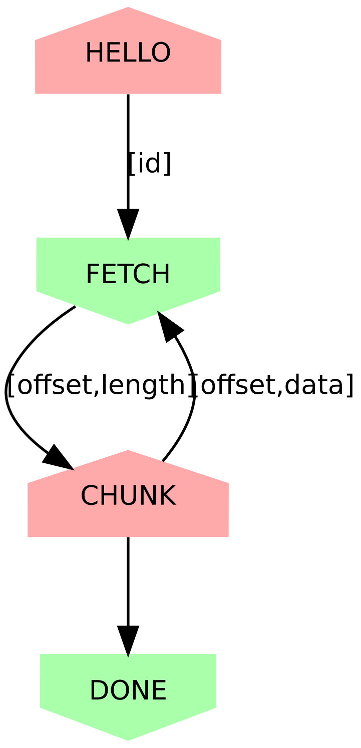

HELLO
[timeout, pyver, abi, ...
HELLO
[id]
IDLE
[package, version]
BUILD
SLEEP
BYE
[status, duration, output, ...
BUILT
BYE
success
SEND
failure/more
failure
SENT
success
DONE

1. The new build slave sends `["HELLO", timeout, py_version_tag, abi_tag, platform_tag]` 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)
   - `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")
2. The master replies with `["HELLO", slave_id]` 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.
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 `["BYE"]` 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 with `["BUILD", 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", status, duration, output, files]`:
   - *status* is `True` if the build succeeded and `False` otherwise.
   - *duration* is a `float`[9] 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.
8. If the build succeeded, the master will send `["SEND", 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 13) 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").

[9] https://docs.python.org/3.4/library/functions.html#float

3.3.2 File Juggler

The queue that talks to *File Juggler* (page 7) 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", 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", 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", 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.

3.4 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 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

```
usage: 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 can should be started on the same machine as the master after the **piw-master** 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 5) to stop querying PyPI, *Slave Driver* (page 6) 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 6)), 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-initdb

The piw-initdb script is used to initialize or upgrade the piwheels master database. The target PostgreSQL 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.

5.1 Synopsis

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

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

-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 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)

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

5.3 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 6

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.

6.1 Synopsis

```
usage: 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 ...]
```

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

--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

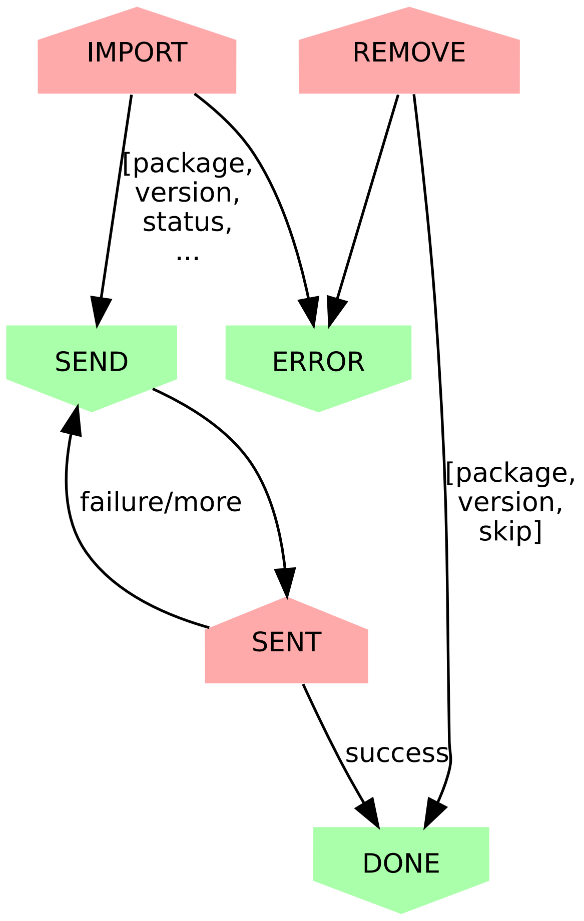
6.3 Protocols

The following section documents the protocol used between the importer and the tasks that it talks to in the *piw-master* (page 3). Each protocol operates over a separate queue. All protocols in the piwheels system follow a similar structure:

1. Each message is a list of Python objects.
2. The first element in the list is a string indicating the type of message.
3. Additional elements depend on the type of the message.
4. A given message type always contains the same number of elements (there are no variable length messages).

6.3.1 Mr Chase

The queue that talks to *Mr. Chase* (page 6) is a ZeroMQ REQ socket, hence the protocol follows a strict request-reply sequence which is illustrated below (see *piw-remove* (page 23) for documentation of the `REMOVE` path):



1. The importer sends `["IMPORT", 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`[10] 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.

[10] https://docs.python.org/3.4/library/functions.html#float

2. If the import information is insufficient or incorrect, the master will send `["ERROR", args, ...]` where args and any further fields are the arguments of the exception that was raised.
3. If the import information is okay, the master will send `["SEND", filename]` for each file mentioned in the build.
4. At this point the importer should use the *File Juggler* (page 13) 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.

6.4 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 20) 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 20) 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 7

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.

7.1 Synopsis

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

7.2 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, --skip
mark the version 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

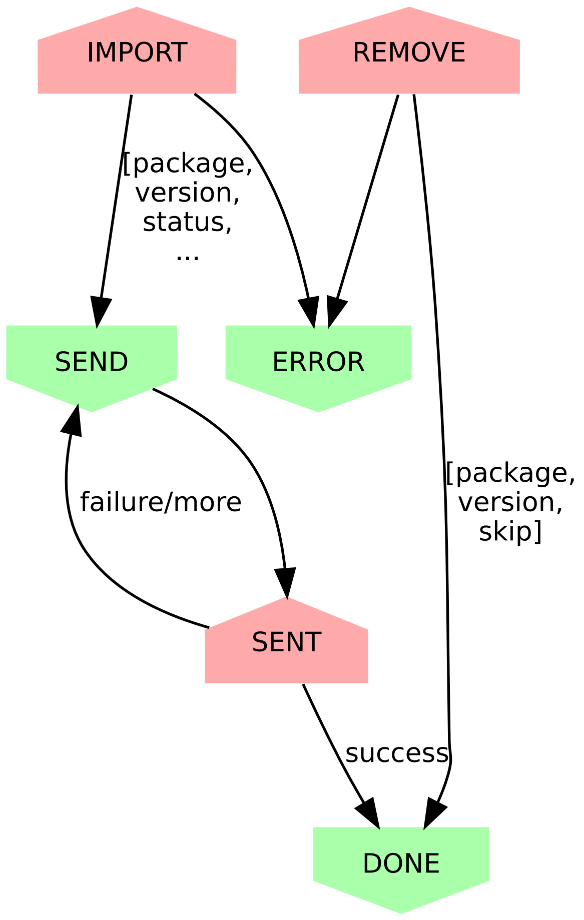
7.3 Protocols

The following section documents the protocol used between the importer and the tasks that it talks to in the *piw-master* (page 3). Each protocol operates over a separate queue. All protocols in the piwheels system follow a similar structure:

1. Each message is a list of Python objects.
2. The first element in the list is a string indicating the type of message.
3. Additional elements depend on the type of the message.
4. A given message type always contains the same number of elements (there are no variable length messages).

7.3.1 Mr Chase

The queue that talks to *Mr. Chase* (page 6) is a ZeroMQ REQ socket, hence the protocol follows a strict request-reply sequence which is illustrated below (see *piw-import* (page 19) for documentation of the `IMPORT` path):



1. The utility sends `["REMOVE", package, version, skip]`:
   - *package* is the name of the package to remove.
   - *version* is the version of the package to remove.
   - *skip* is `True` if the version should never be built again, and `False` otherwise.
2. If the removal fails (e.g. if the package or version does not exist), the master will send `["ERROR", args, ...]`.
3. If the removal is successful, the master replies with `["DONE"]`.

7.4 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 24) 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 8

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.

8.1 piwheels.master

Defines the *`PiWheelsMaster`* (page 27) class. An instance of this is the entry-point for the **`piw-master`** script.

`class` `piwheels.master.`**`PiWheelsMaster`**

This is the main class for the **`piw-master`** script. It spawns various worker threads, then spends its time communicating with any attached monitor applications (see **`piw-monitor`**) and build slaves (see **`piw-slave`**).

`static configure_parser``()`

Construct the command line parser for **`piw-master`** with its many options (this method only exists to simplify the main method).

`do_hello``()`

Handler for the HELLO message; this indicates a new monitor has been attached and would like all the build slave's HELLO messages replayed to it.

`do_kill``(`*slave_id*`)`

Handler for the KILL message; this terminates the specified build slave by its master id.

`do_pause``()`

Handler for the PAUSE message; this requests all tasks pause their operations.

`do_quit``()`

Handler for the QUIT message; this terminates the master.

`do_resume``()`

Handler for the RESUME message; this requests all tasks resume their operations.

`main_loop``()`

This is the main loop of the **`piw-master`** script. It receives messages from the internal status queue and forwards them onto the external status queue (for any **`piw-monitor`** scripts that are attached). It also retrieves any messages sent to the control queue and dispatches them to a handler.

piwheels.master.**sig_term**(*signo*, *stack_frame*)

Handler for the SIGTERM signal; raises SystemExit[11] which will cause the *PiWheelsMaster.main_loop()* (page 27) method to terminate.

8.2 piwheels.master.tasks

Implements the base classes (*Task* (page 28) and its derivative *PauseableTask* (page 28)) which form the basis of all the tasks in the piwheels master.

exception piwheels.master.tasks.**TaskQuit**

Exception raised when the "QUIT" message is received by the internal control queue.

class piwheels.master.tasks.**Task**(*config*)

The *Task* (page 28) class is a Thread[12] derivative which is the base for all tasks in the piwheels master. The *run()* (page 28) method is overridden to perform a simple task loop which calls *loop()* (page 28) once a cycle, and *poll()* (page 28) to react to any messages arriving into queues. Queues are associated with handlers via the *register()* (page 28) method.

handle_control(*queue*)

Default handler for the internal control queue. In this base implementation it simply handles the "QUIT" message by raising TaskQuit (which the *run()* (page 28) method will catch and use as a signal to end).

loop()

This method is called once per loop of the task's *run()* (page 28) method. If the task needs to do some work periodically, this is the place to do it.

pause()

Requests that the task pause itself. This is an idempotent method; it's always safe to call repeatedly and even if the task isn't pauseable it'll simply be ignored.

poll(*timeout=1000*)

This method is called once per loop of the task's *run()* (page 28) method. It polls all registered queues and calls their associated handlers if the poll is successful.

quit()

Requests that the task terminate at its earliest convenience. To wait until the task has actually closed, call `join()` afterwards.

register(*queue*, *handler*, *flags=<Mock object>*)

Register *queue* to be polled on each cycle of the task. Any messages with the relevant *flags* (defaults to `POLLIN`) will trigger the specified *handler* method which is expected to take a single argument which will be *queue*.

Parameters

- **queue** (*zmq.Socket*) – The queue to poll.
- **handler** – The function or method to call when a message with matching *flags* arrives in *queue*.
- **flags** (*int*[13]) – The flags to match in the queue poller (defaults to `POLLIN`).

resume()

Requests that the task resume itself. This is an idempotent method; it's safe to call repeatedly and even if the task isn't pauseable it'll simply be ignored.

run()

This method is the main task loop. Override this to perform one-off startup processing within the task's background thread, and to perform any finalization required.

[11] https://docs.python.org/3.4/library/exceptions.html#SystemExit
[12] https://docs.python.org/3.4/library/threading.html#threading.Thread
[13] https://docs.python.org/3.4/library/functions.html#int

class piwheels.master.tasks.**PauseableTask**(*config*)

Derivative of `Task` (page 28) that implements a rudimentary pausing mechanism. When the "PAUSE" message is received on the internal control queue, the task will enter a loop which simply polls the control queue waiting for "RESUME" or "QUIT". No other work will be done (`Task.loop()` (page 28) and `Task.poll()` (page 28) will not be called) until the task is resumed (or terminated).

8.3 piwheels.master.states

This module defines several classes which permit interested tasks to track the state of build slaves (`SlaveState` (page 30)), file transfers (`TransferState` (page 30)), build attempts (`BuildState` (page 29)) and build artifacts (`FileState` (page 29)).

class piwheels.master.states.**FileState**(*filename*, *filesize*, *filehash*, *package_tag*, *package_version_tag*, *py_version_tag*, *abi_tag*, *platform_tag*, *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`[14]) – The original filename of the build artifact.
- **filesize** (`int`[15]) – The size of the file in bytes.
- **filehash** (`str`[16]) – The SHA256 hash of the file contents.
- **package_tag** (`str`[17]) – The package tag extracted from the filename (first "-" separated component).
- **package_version_tag** (`str`[18]) – The package version tag extracted from the filename (second "-" separated component).
- **py_version_tag** (`str`[19]) – The python version tag extracted from the filename (third from last "-" separated component).
- **abi_tag** (`str`[20]) – The python ABI tag extracted from the filename (second from last "-" separated component).
- **platform_tag** (`str`[21]) – The platform tag extracted from the filename (last "-" separated component).
- **transferred** (`bool`[22]) – `True` if the file has been transferred from the build slave that generated it to the file server.

verified()

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

class piwheels.master.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 30) attribute is a mapping containing details of each successfully built package file.

Parameters

[14] https://docs.python.org/3.4/library/stdtypes.html#str
[15] https://docs.python.org/3.4/library/functions.html#int
[16] https://docs.python.org/3.4/library/stdtypes.html#str
[17] https://docs.python.org/3.4/library/stdtypes.html#str
[18] https://docs.python.org/3.4/library/stdtypes.html#str
[19] https://docs.python.org/3.4/library/stdtypes.html#str
[20] https://docs.python.org/3.4/library/stdtypes.html#str
[21] https://docs.python.org/3.4/library/stdtypes.html#str
[22] https://docs.python.org/3.4/library/functions.html#bool

- **slave_id** (*int*[23]) – The master's identifier for the build slave.
- **package** (*str*[24]) – The name of the package to build.
- **version** (*str*[25]) – The version number of the package to build.
- **abi_tag** (*str*[26]) – The ABI for which the build was attempted (must not be `'none'`).
- **status** (*bool*[27]) – `True` if the build succeeded, `False` if it failed.
- **duration** (*datetime.timedelta*[28]) – The amount of time it took to complete the build.
- **output** (*str*[29]) – The log output of the build.
- **files** (*dict*[30]) – A mapping of filenames to *FileState* (page 29) objects for each artifact produced by the build.
- **build_id** (*int*[31]) – The integer identifier generated for the build by the database (`None` until the build has been inserted into the database).

classmethod from_db(*db*, *build_id*)

Construct an instance by querying the database for the specified *build_id*.

Parameters

- **db** (`Database` (page 31)) – A *Database* (page 31) instance to query.
- **build_id** (*int*[32]) – The integer identifier of an attempted build.

logged(*build_id*)

Called to fill in the build's ID in the backend database.

files

A mapping of filename to *FileState* (page 29) 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.master.states.**SlaveState**(*address*, *timeout*, *native_py_version*, *native_abi*, *native_platform*)

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 29) (accessible from `build`) and *TransferState* (page 30) (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.master.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`.

[23] https://docs.python.org/3.4/library/functions.html#int
[24] https://docs.python.org/3.4/library/stdtypes.html#str
[25] https://docs.python.org/3.4/library/stdtypes.html#str
[26] https://docs.python.org/3.4/library/stdtypes.html#str
[27] https://docs.python.org/3.4/library/functions.html#bool
[28] https://docs.python.org/3.4/library/datetime.html#datetime.timedelta
[29] https://docs.python.org/3.4/library/stdtypes.html#str
[30] https://docs.python.org/3.4/library/stdtypes.html#dict
[31] https://docs.python.org/3.4/library/functions.html#int
[32] https://docs.python.org/3.4/library/functions.html#int

8.4 piwheels.master.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` (page 38) for the usage of these functions.

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

Returns 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.master.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)]
```

8.5 piwheels.master.db

This module defines the low level database API, `Database` (page 31). This is a simple core SQLAlchemy affair which runs trivial queries against the PostgreSQL database. All the serious logic is defined within views in the database itself.

`class` `piwheels.master.db.`**`Database`**`(`*dsn*`)`

PiWheels database connection class

add_new_package(*package*)
Insert a new package record into the database. Key violations are ignored as packages is effectively an append-only table.

add_new_package_version(*package*, *version*)
Insert a new package version record into the database. Key violations are ignored as versions is effectively an append-only table.

delete_build(*package*, *version*)
Remove all builds for the specified package and version, along with all files and download records

get_all_package_versions()
Returns the set of all known (package, version) tuples

get_all_packages()
Returns the set of all known package names

get_build(*build_id*)
Return all details about a given build.

get_build_abis()
Return the set of ABIs that the master should attempt to build

get_build_queue()
Returns a generator covering the entire builds_pending view; streaming results are activated for this query as it's more important to get the first result quickly than it is to retrieve the entire set.

get_files(*build_id*)
Return all details about the files generated by a given build.

get_package_files(*package*)
Returns all details required to build the index.html for the specified package.

get_pypi_serial()
Return the serial number of the last PyPI event

get_statistics()
Return various build related statistics from the database (see the definition of the `statistics` view in the database creation script for more information.

get_version_files(*package*, *version*)
Returns the names of all files for *version* of *package*

log_build(*build*)
Log a build attempt in the database, including build output and wheel info if successful

log_download(*download*)
Log a download in the database, including data derived from JSON in pip's user-agent.

log_file(*build*, *file*)
Log a pending file transfer in the database, including file-size, hash, and various tags

set_pypi_serial(*serial*)
Update the serial number of the last PyPI event

skip_package(*package*)
Mark a package to prevent future builds of all versions (and all future versions).

skip_package_version(*package*, *version*)
Mark a version of a package to prevent future build attempts.

test_package_version(*package*, *version*)
Check whether *version* of *package* already exists in the database. Returns a boolean.

8.6 piwheels.master.cloud_gazer

Defines the `CloudGazer` (page 33) task; see class for more details.

class `piwheels.master.cloud_gazer.`**`CloudGazer`**(*config*)

This task scrapes PyPI for the list of available packages, and the versions of those packages. This information is written into the backend database for `TheArchitect` (page 35) to use.

8.7 piwheels.master.the_oracle

Defines `TheOracle` (page 33) task and the `DbClient` (page 34) RPC class for talking to it.

class `piwheels.master.the_oracle.`**`TheOracle`**(*config*)

This task provides an RPC-like interface to the database; it handles requests such as registering a new package, version, or build, and answering queries about the hashes of files. The primary clients of this class are `SlaveDriver` (page 35), `IndexScribe` (page 39), and `CloudGazer` (page 33).

Note that because database requests are notoriously variable in length the client RPC class (`DbClient` (page 34)) doesn't *directly* talk to `TheOracle` (page 33). Rather, multiple instances of `TheOracle` (page 33) are spawned and `Seraph` (page 35) sits in front of these acting as a simple load-sharing router for the RPC clients.

`do_allpkgs`()

Handler for "ALLPKGS" message, sent by `DbClient` (page 34) to request the set of all packages define known to the database.

`do_allvers`()

Handler for "ALLVERS" message, sent by `DbClient` (page 34) to request the set of all (package, version) tuples known to the database.

`do_delbuild`(*package*, *version*)

Handler for "DELBUILD" message, sent by `DbClient` (page 34) to remove all builds (and files and downloads by cascade) for *version* of *package*.

`do_getabis`()

Handler for "GETABIS" message, sent by `DbClient` (page 34) to request the list of all ABIs to build for.

`do_getpypi`()

Handler for "GETPYPI" message, sent by `DbClient` (page 34) to request the record of the last serial number from the PyPI changelog.

`do_getstats`()

Handler for "GETSTATS" message, sent by `DbClient` (page 34) to request the latest database statistics, returned as a list of (field, value) tuples.

`do_logbuild`(*build*)

Handler for "LOGBUILD" message, sent by `DbClient` (page 34) to register a new build result.

`do_logdownload`(*download*)

Handler for "LOGDOWNLOAD" message, sent by `DbClient` (page 34) to register a new download.

`do_newpkg`(*package*)

Handler for "NEWPKG" message, sent by `DbClient` (page 34) to register a new package.

`do_newver`(*package*, *version*)

Handler for "NEWVER" message, sent by `DbClient` (page 34) to register a new (package, version) tuple.

`do_pkgexists`(*package*, *version*)

Handler for "PKGEXISTS" message, sent by `DbClient` (page 34) to request whether or not the specified *version* of *package* exists.

do_pkgfiles (*package*)
Handler for "PKGFILES" message, sent by `DbClient` (page 34) to request details of all wheels assocated with *package*.

do_setpypi (*serial*)
Handler for "SETPYPI" message, sent by `DbClient` (page 34) to update the last seen serial number from the PyPI changelog.

do_skippkg (*package*)
Handler for "SKIPPKG" message, sent by `DbClient` (page 34) to skip building all versions of a package.

do_skipver (*package*, *version*)
Handler for "SKIPVER" message, sent by `DbClient` (page 34) to skip building a specific version of a package.

do_verfiles (*package*, *version*)
Handler for "VERFILES" message, sent by `DbClient` (page 34) to request the filenames of all wheels associated with *version* of *package*.

handle_db_request (*queue*)
Handle incoming requests from `DbClient` (page 34) instances.

class piwheels.master.the_oracle.**DbClient** (*config*)
RPC client class for talking to `TheOracle` (page 33).

add_new_package (*package*)
See `db.Database.add_new_package()` (page 31).

add_new_package_version (*package*, *version*)
See `db.Database.add_new_package_version()` (page 32).

delete_build (*package*, *version*)
See `db.Database.delete_build()` (page 32).

get_all_package_versions ()
See `db.Database.get_all_package_versions()` (page 32).

get_all_packages ()
See `db.Database.get_all_packages()` (page 32).

get_build_abis ()
See `db.Database.get_build_abis()` (page 32).

get_package_files (*package*)
See `db.Database.get_package_files()` (page 32).

get_pypi_serial ()
See `db.Database.get_pypi_serial()` (page 32).

get_statistics ()
See `db.Database.get_statistics()` (page 32).

get_version_files (*package*, *version*)
See `db.Database.get_version_files()` (page 32).

log_build (*build*)
See `db.Database.log_build()` (page 32).

log_download (*download*)
See `db.Database.log_download()` (page 32).

set_pypi_serial (*serial*)
See `db.Database.set_pypi_serial()` (page 32).

skip_package (*package*)
See `db.Database.skip_package()` (page 32).

skip_package_version(*package*, *version*)
See `db.Database.skip_package_version()` (page 32).

test_package_version(*package*, *version*)
See `db.Database.test_package_version()` (page 32).

8.8 piwheels.master.seraph

Defines the `Seraph` (page 35) task; see class for more details.

class piwheels.master.seraph.**Seraph**(*config*)
This task is a simple load-sharing router for `TheOracle` (page 33) tasks.

handle_back(*queue*)
Receive a response from an instance of `TheOracle` (page 33) on the back queue. Strip off the worker's address frame and add it back to the available queue then send the response back to the client that made the original request.

handle_front(*queue*)
If any workers are currently available, receive `DbClient` (page 34) requests from the front queue and send it on to the worker including the client's address frame.

8.9 piwheels.master.the_architect

Defines `TheArchitect` (page 35) task; see class for more details.

class piwheels.master.the_architect.**TheArchitect**(*config*)
This task queries the backend database to determine which versions of packages have yet to be built (and aren't marked to be skipped). It places a tuple of (package, version) for each such build into the internal "builds" queue for `SlaveDriver` (page 35) to read.

handle_builds(*queue*)
Handler for the task's builds queue. Whenever a build slave asks `SlaveDriver` (page 35) for a new task, `SlaveDriver` (page 35) passes the slave's ABI to `TheArchitect` (page 35) via this queue. We simply pop the first entry (if any) off the relevant queue and send it back.

loop()
The architect simply runs the build queue query repeatedly. On each loop iteration, an entry from the result set is added to the relevant ABI queue. The queues are limited in length to prevent silly memory usage on the initial run (which will involve millions of entries). This does mean that a single loop over the query will potentially miss entries, but that's fine as it'll just be repeated again.

8.10 piwheels.master.slave_driver

Defines the `SlaveDriver` (page 35) task; see class for more details.

class piwheels.master.slave_driver.**SlaveDriver**(*config*)
This task handles interaction with the build slaves using the slave protocol. Interaction is driven by the slaves (i.e. the master doesn't *push* jobs, rather the slaves *request* a job and the master replies with the next (package, version) tuple from the internal "builds" queue).

The task also incidentally interacts with several other queues: the internal "status" queue is sent details of every reply sent to a build slave (the `main_loop()` (page 27) method passes this information on to any listening monitors). Also, the internal "indexes" queue is informed of any packages that need web page indexes re-building (as a result of a successful build).

active_builds()
Generator method which yields all (package, version) tuples currently being built by build slaves.

`do_built`(*slave*)
Handler for the build slave's "BUILT" message, which is sent after an attempted package build succeeds or fails. The handler logs the result in the database and, if files have been generated by the build, informs the *`FileJuggler`* (page 38) task to expect a file transfer before sending "SEND" back to the build slave with the required filename.

If no files were generated (e.g. in the case of a failed build, or a degenerate success), "DONE" is returned indicating that the build slave is free to discard all resources generated during the build and return to its idle state.

`do_bye`(*slave*)
Handler for the build slave's final "BYE" message upon shutdown. This removes the associated state from the internal `slaves` dict.

Parameters **`slave`** (`SlaveState` (page 30)) – The object representing the current status of the build slave.

`do_hello`(*slave*)
Handler for the build slave's initial "HELLO" message. This associates the specified *slave* state with the slave's address and returns "HELLO" with the master's id for the slave (the id communicated back simply for consistency of logging; administrators can correlate master log messages with slave log messages when both have the same id number; we can't use IP address for this as multiple slaves can run on one machine).

Parameters **`slave`** (`SlaveState` (page 30)) – The object representing the current status of the build slave.

`do_idle`(*slave*)
Handler for the build slave's "IDLE" message (which is effectively the slave requesting work). If the master wants to terminate the slave, it sends back "BYE". If the build queue (for the slave's ABI) is empty or the task is currently paused, "SLEEP" is returned indicating the slave should wait a while and then try again.

If a job can be retrieved from the (ABI specific) build queue, then a "BUILD" message is sent back with the required package and version.

Parameters **`slave`** (`SlaveState` (page 30)) – The object representing the current status of the build slave.

`do_sent`(*slave*)
Handler for the build slave's "SENT" message indicating that it's finished sending the requested file to `FileJuggler`. The `FsClient` RPC mechanism is used to ask `FileJuggler` to verify the transfer against the stored hash and, if this is successful, a message is sent to `IndexScribe` to regenerate the package's index.

If further files remain to be transferred, another "SEND" message is returned to the build slave. Otherwise, "DONE" is sent to free all build resources.

If a transfer fails to verify, another "SEND" message with the same filename is returned to the build slave.

`handle_control`(*queue*)
Handle incoming requests to the internal control queue.

Whilst the *`SlaveDriver`* (page 35) task is "pauseable", it can't simply stop responding to requests from build slaves. Instead, its pause is implemented as an internal flag. While paused it simply tells build slaves requesting a new job that none are currently available but otherwise continues servicing requests.

It also understands a couple of extra control messages unique to it, specifically "KILL" to tell a build slave to terminate, and "HELLO" to cause all "HELLO" messages from build slaves to be replayed (for the benefit of a newly attached monitor process).

`handle_slave`(*queue*)
Handle requests from build slaves.

See the *piw-slave* (page 9) chapter for an overview of the protocol for messages between build slaves and `SlaveDriver` (page 35). This method retrieves the message from the build slave, finds the associated `SlaveState` (page 30) and updates it with the message, then calls the appropriate message handler. The handler will be expected to return a reply (in the usual form of a list of strings) or `None` if no reply should be sent (e.g. for a final "BYE" message).

`kill_slave`(*slave_id*)
Additional task control method to trigger a "KILL" message to the internal control queue. See `quit()` (page 28) for more information.

`list_slaves`()
Additional task control method to trigger a "HELLO" message to the internal control queue. See `quit()` (page 28) for more information.

8.11 piwheels.master.mr_chase

Defines the `MrChase` (page 37) task; see class for more details.

class `piwheels.master.mr_chase.`**`MrChase`**(*config*)
This task handles smuggling packages into the database manually. It is the task that the **`piw-import`** script talks to in order to import packages.

Internally, the task is essentially an abbreviated `SlaveDriver` (in as much as it has to perform similar database and file-system interactions) but without having to handle talking to lots of build slaves.

`do_import`(*state*)
Handler for the importer's initial "IMPORT" message. This method checks the information in the state passes some simple tests, then ensures that the requested package and version exist in the database (creating them if necessary).

`do_remove`(*state*)
Handler for the importer's "REMOVE" message, indicating a request to remove a specific version of a package from the system.

`do_sent`(*state*)
Handler for the importer's "SENT" message indicating that it's finished sending the requested file to `FileJuggler`. The file is verified (as in `SlaveDriver`) and, if this is successful, a mesage is sent to `IndexScribe` to regenerate the package's index.

If further files remain to be transferred, another "SEND" message is returned to the build slave. Otherwise, "DONE" is sent to free all build resources.

If a transfer fails to verify, another "SEND" message with the same filename is returned to the build slave.

`handle_import`(*queue*)
Handle requests from **`piw-import`** instances.

See the *piw-import* (page 19) and *piw-remove* (page 23) chapters for an overview of the protocol for messages between the importer and `MrChase` (page 37).

8.12 piwheels.master.file_juggler

Defines the `FileJuggler` (page 38) task and the `FsClient` (page 39) RPC class for interacting with it.

exception `piwheels.master.file_juggler.`**`TransferError`**
Base class for errors raised during a file transfer.

exception `piwheels.master.file_juggler.`**`TransferIgnoreChunk`**
Exception raised when a build slave sends CHUNK instead of HELLO as the first message (see `FileJuggler.new_transfer()` (page 39)).

exception `piwheels.master.file_juggler.`**`TransferDone`**

Exception raised when a transfer is complete. It may seem a little odd to use an exception for this, but it is "exceptional" behaviour to terminate the file transfer.

class `piwheels.master.file_juggler.`**`FileJuggler`**(*config*)

This task handles file transfers from the build slaves. The specifics of the file transfer protocol are best understood from the implementation of the *`FileState`* (page 29) class.

However, to detail how a file transfer begins: when a build slave has successfully completed a build it informs the master via the *`SlaveDriver`* (page 35) task. That task replies with a "SEND" instruction to the slave (including a filename). The slave then initiates the transfer with a "HELLO" message to this task. Once transfers are complete the slave sends a "SENT" message to the *`SlaveDriver`* (page 35) task which verifies the transfer and either retries it (when verification fails) or sends back "DONE" indicating the slave can wipe the source file.

`current_transfer`(*transfer*, *msg*, **args*)

Called for messages associated with an existing file transfer.

Usually this is "CHUNK" indicating another chunk of data. Rarely, it can be "HELLO" if the master has fallen silent and dropped tons of packets.

Parameters

- **`transfer`** (`TransferState` (page 30)) – The object representing the state of the transfer.
- **`msg`** (*`str`*[33]) – The message sent during the transfer.
- **`*args`** – All additional arguments; for "CHUNK" the first must be the file offset and the second the data to write to that offset.

`do_expect`(*slave_id*, *file_state*)

Message sent by *`FsClient`* (page 39) to inform file juggler that a build slave is about to start a file transfer. The message includes the full *`FileState`* (page 29). The state is stored in the `pending` map.

Parameters

- **`slave_id`** (*`int`*[34]) – The identity of the build slave about to begin the transfer.
- **`file_state`** (`FileState` (page 29)) – The details of the file to be transferred including the expected hash.

`do_remove`(*package*, *filename*)

Message sent by *`FsClient`* (page 39) to request that *filename* in *package* should be removed.

`do_statvfs`()

Message sent by *`FsClient`* (page 39) to request that file juggler return stats on the output file-system.

`do_verify`(*slave_id*, *package*)

Message sent by *`FsClient`* (page 39) to request that juggler verify a file transfer against the expected hash and, if it matches, rename the file into its final location.

Parameters

- **`slave_id`** (*`int`*[35]) – The identity of the build slave that sent the file.
- **`package`** (*`str`*[36]) – The name of the package that the file is to be committed to, if valid.

`handle_file`(*queue*)

Handle incoming file-transfer messages from build slaves.

[33] https://docs.python.org/3.4/library/stdtypes.html#str
[34] https://docs.python.org/3.4/library/functions.html#int
[35] https://docs.python.org/3.4/library/functions.html#int
[36] https://docs.python.org/3.4/library/stdtypes.html#str

The file transfer protocol is in some ways very simple (see the chart in the *piw-slave* (page 9) chapter for an overview of the message sequence) and in some ways rather complex (read the ZeroMQ guide chapter on file transfers for more detail on why multiple messages must be allowed in flight simultaneously).

The "normal" state for a file transfer is to be requesting and receiving chunks. Anything else, including redundant re-sends, and transfer completion is handled as an exceptional case.

handle_fs_request (*queue*)
Handle incoming messages from `FsClient` (page 39) instances.

new_transfer (*msg*, **args*)
Called for messages initiating a new file transfer.

The first message must be HELLO along with the id of the slave starting the transfer. The metadata for the transfer will be looked up in the `pending` list (which is written to by `do_expect()` (page 38)).

Parameters

- **msg** (`str`[37]) – The message sent to start the transfer (must be "HELLO")
- ***args** – All additional arguments (expected to be an integer slave id).

class piwheels.master.file_juggler.**FsClient** (*config*)
RPC client class for talking to `FileJuggler` (page 38).

expect (*slave_id*, *file_state*)
See `FileJuggler.do_expect()` (page 38).

remove (*package*, *filename*)
See `FileJuggler.do_remove()` (page 38).

statvfs ()
See `FileJuggler.do_statvfs()` (page 38).

verify (*slave_id*, *package*)
See `FileJuggler.do_verify()` (page 38).

8.13 piwheels.master.big_brother

Defines the `BigBrother` (page 39) task; see class for more details.

class piwheels.master.big_brother.**BigBrother** (*config*)
This task periodically queries the database and output file-system for various statistics like the number of packages known to the system, the number built, the number of packages built in the last hour, the remaining file-system space, etc. These statistics are written to the internal "status" queue which `main_loop()` (page 27) uses to pass statistics to any listening monitors.

handle_index (*queue*)
Handler for the index_queue. Whenever a slot becomes available, and an updated status_info1 package is available, send a message to update the home page.

handle_status (*queue*)
Handler for the internal status queue. Whenever a slot becomes available, and an updated status_info2 package is available, send a message with the latest status (ultimately this winds up going to any attached monitors via the external status queue).

8.14 piwheels.master.index_scribe

Defines the `IndexScribe` (page 39) task; see class for more details.

[37] https://docs.python.org/3.4/library/stdtypes.html#str

class `piwheels.master.index_scribe.`**`IndexScribe`**(*config*)

This task is responsible for writing web-page `index.html` files. It reads the names of packages off the internal "indexes" queue and rebuilds the `index.html` for that package and, optionally, the overall `index.html` if the package is one that wasn't previously present.

Note: It is important to note that package names are never pushed into the internal "indexes" queue until all file-transfers associated with the build are complete. Furthermore, while the entire index for a package is re-built, hashes are *never* re-calculated from the disk files (they are always read from the database).

`handle_index`(*queue*)

Handle incoming requests to (re)build index files. These will be in the form of "HOME", a request to write the homepage with some associated statistics, or "PKG", a request to write the index for the specified package.

Note: In all handlers below, care is taken to ensure clients never see a partially written file and that temporary files are cleaned up in the event of any exceptions.

`setup_output_path`()

Called on task startup to copy all static resources into the output path (and to make sure the output path exists as a directory).

`write_homepage`(*status_info*)

Re-writes the site homepage using the provided statistics in the homepage template (which is effectively a simple Python format string).

Parameters **`status_info`** (`tuple`[38]) – A namedtuple containing statistics obtained by `BigBrother`.

`write_package_index`(*package*, *files*)

(Re)writes the index of the specified package. The file meta-data (including the hash) is retrieved from the database, *never* from the file-system.

Parameters

- **`package`** (`str`[39]) – The name of the package to write the index for
- **`files`** (`list`[40]) – A list of (filename, filehash) tuples.

`write_root_index`()

(Re)writes the index of all packages. This is implicitly called when a request to write a package index is received for a package not present in the task's cache.

8.15 piwheels.slave

Defines the `PiWheelsSlave` (page 40) class. An instance of this is the entry-point for the **`piw-slave`** script.

class `piwheels.slave.`**`PiWheelsSlave`**

This is the main class for the **`piw-slave`** script. It connects (over zmq sockets) to a master (see **`piw-master`**) then loops around the slave protocol (see the *piw-slave* (page 9) chapter). It retrieves source packages directly from PyPI[41], attempts to build a wheel in a sandbox directory and, if successful, transmits the results to the master.

`do_build`(*package*, *version*)

Alternatively, in response to "IDLE", the master may send "BUILD" *package version*. We should

[38] https://docs.python.org/3.4/library/stdtypes.html#tuple
[39] https://docs.python.org/3.4/library/stdtypes.html#str
[40] https://docs.python.org/3.4/library/stdtypes.html#list
[41] https://pypi.python.org/

then attempt to build the specified wheel and send back a "BUILT" message with a full report of the outcome.

do_bye()
The master may respond with "BYE" at any time indicating we should immediately terminate (first cleaning up any extant build). We return `None` to tell the main loop to quit.

do_done()
After all files have been sent (and successfully verified), the master will reply with "DONE" indicating we can remove all associated build artifacts. We respond with "IDLE".

do_hello(*new_id*, *pypi_url*)
In response to our initial "HELLO" (detailing our various **PEP 425**[42] tags), the master is expected to send "HELLO" back with an integer identifier and the URL of the PyPI repository to download from. We use the identifier in all future log messages for the ease of the administrator.

We reply with "IDLE" to indicate we're ready to accept a build job.

do_send(*filename*)
If a build succeeds and generates files (detailed in a "BUILT" message), the master will reply with "SEND" *filename* indicating we should transfer the specified file (this is done on a separate socket with a different protocol; see `builder.PiWheelsPackage.transfer()` (page 41) for more details). Once the transfers concludes, reply to the master with "SENT".

do_sleep()
If, in response to an "IDLE" message we receive "SLEEP" this indicates the master has nothing for us to do currently. Sleep for a little while then try "IDLE" again.

handle_reply(*reply*, **args*)
Dispatch a message from the master to an appropriate handler method.

piwheels.slave.**duration**(*s*)
Convert *s*, a string representing a duration, into a `datetime.timedelta`[43].

8.16 piwheels.slave.builder

Defines the classes which use `pip` to build wheels.

class piwheels.slave.builder.**PiWheelsPackage**(*path*)
Records the state of a build artifact, i.e. a wheel package. The filename is deconstructed into the fields specified by **PEP 425**[44].

Parameters path (`pathlib.Path`[45]) – The path to the wheel on the local filesystem.

open(*mode='rb'*)
Open the wheel in binary mode and return the open file object.

transfer(*queue*, *slave_id*)
Transfer the wheel via the specified *queue*. This is the client side implementation of the `file_juggler.FileJuggler` (page 38) protocol.

abi_tag
Return the ABI part of the wheel's filename (the penultimate "-" separated element).

build_tag
Return the optional build part of the wheel's filename (the third "-" separated element when 6 elements exist in total).

filehash
Return an SHA256 digest of the wheel's contents.

[42] https://www.python.org/dev/peps/pep-0425
[43] https://docs.python.org/3.4/library/datetime.html#datetime.timedelta
[44] https://www.python.org/dev/peps/pep-0425
[45] https://docs.python.org/3.4/library/pathlib.html#pathlib.Path

filename
Return the filename of the wheel as a simple string (with no path components).

filesize
Return the size of the wheel in bytes.

metadata
Return the contents of the `metadata.json` file inside the wheel.

package_tag
Return the package part of the wheel's filename (the first "-" separated element).

package_version_tag
Return the version part of the wheel's filename (the second "-" separated element).

platform_tag
Return the platform part of the wheel's filename (the last "-" separated element).

py_version_tag
Return the python version part of the wheel's filename (third from last "-" separated element).

class `piwheels.slave.builder.`**PiWheelsBuilder**(*package*, *version*)
Class responsible for building wheels for a given *version* of a *package*.

Parameters

- **package** (*str*[46]) – The name of the package to attempt to build wheels for.
- **version** (*str*[47]) – The version of the package to attempt to build.

build(*timeout=None*, *pypi_index='https://pypi.python.org/simple'*)
Attempt to build the package within the specified *timeout*.

Parameters

- **timeout** (*float*[48]) – The number of seconds to wait for `pip` to finish before raising `subprocess.TimeoutExpired`[49].
- **pypi_index** (*str*[50]) – The URL of the **PEP 503**[51] compliant repository from which to fetch packages for building.

clean()
Remove the temporary build directory and all its contents.

as_message
Return the state as a list suitable for use in several protocol messages (specifically those used by **piw-slave** and **piw-import**).

8.17 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.

[46] https://docs.python.org/3.4/library/stdtypes.html#str
[47] https://docs.python.org/3.4/library/stdtypes.html#str
[48] https://docs.python.org/3.4/library/functions.html#float
[49] https://docs.python.org/3.4/library/subprocess.html#subprocess.TimeoutExpired
[50] https://docs.python.org/3.4/library/stdtypes.html#str
[51] https://www.python.org/dev/peps/pep-0503

`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`[52] is the version is so ancient we can't do anything with it.

`piwheels.initdb.`**`get_connection`**`(`*dsn*`)`

Return an SQLAlchemy connection to the specified *dsn* or raise `RuntimeError`[53] if the database doesn't exist (the administrator is expected to create the database before running this script).

`piwheels.initdb.`**`get_script`**`(`*version*`)`

Generate the script to get the database from *version* (the result of *`detect_version()`* (page 42)) 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.

8.18 piwheels.importer

Contains the functions that implement the **`piw-import`** script.

`piwheels.importer.`**`main`**`(`*args=None*`)`

This is the main function for the **`piw-import`** script. It uses some bits of the **`piw-slave`** script to deconstruct the filenames passed to it in order to build all the requried information that *`MrChase`* (page 37) needs.

`piwheels.importer.`**`print_builder`**`(`*config*, *builder*`)`

Dumps a human-readable description of the *builder* to the log / console.

Parameters

- **`config`** – The configuration generated from the command line argument parser.
- **`builder`** (`PiWheelsBuilder` (page 42)) – The builder to print the description of.

`piwheels.importer.`**`abi`**`(`*config*, *builder*, *default=None*`)`

Calculate the ABI from the given *config* and the first file contained by the *builder* state. If the configuration contains no ABI override, and the ABI of the first file is 'none', return *default*.

`piwheels.importer.`**`do_import`**`(`*config*, *builder*`)`

Handles constructing and sending the initial "IMPORT" message to `master.mr_chase.MrChase`. If "SEND" is then received, uses *`do_send()`* (page 43) to handle transmitting files.

Parameters

- **`config`** – The configuration obtained from parsing the command line.
- **`builder`** (`PiWheelsBuilder` (page 42)) – The object representing the state of the build.

`piwheels.importer.`**`do_send`**`(`*builder*, *filename*`)`

Handles sending files when requested by *`do_import()`* (page 43).

[52] https://docs.python.org/3.4/library/exceptions.html#RuntimeError
[53] https://docs.python.org/3.4/library/exceptions.html#RuntimeError

8.19 piwheels.remove

Contains the functions that implement the **`piw-remove`** script.

`piwheels.remove.`**`main`**`(`*args=None*`)`

This is the main function for the **`piw-remove`** script. It uses *`MrChase`* (page 37) to remove built packages from the system.

`piwheels.remove.`**`do_remove`**`(`*config*`)`

Handles constructing and sending the "REMOVE" message to `master.mr_chase.MrChase`.

Parameters **`config`** – The configuration obtained from parsing the command line.

CHAPTER 9

License

[54] https://github.com/bennuttall
[55] dave@waveform.org.uk

Python Module Index

Index