Piwheels 0.20 Documentation

Release 0.20

Ben Nuttall

Sep 09, 2023

CONTENTS

CHAPTER ONE

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 11)	Builds packages 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-sense (page 17)	Provides a friendly Sense HAT-based UI for interacting with the piwheels master.
piw-initdb (page 19)	A simple maintenance script for initializing or upgrading the database to the current version.
piw-import (page 21)	A tool for importing wheels manually into the piwheels database and file-system.
piw-add (page 25)	A tool for manually adding packages or versions to the database (and therefore, the build queue).
piw-remove (page 27)	A tool for manually removing builds from the database and file-system.
piw-rebuild (page 23)	A tool for regenerating certain elements of the piwheels web-site.
piw-logger (page 29)	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

TWO

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* (page 11) 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* (page 19) script. It is *strongly 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] [--debug TASK] [--pypi-xmlrpc URL]
                [--pypi-simple URL] [--pypi-json 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 instead of the defaults at:

- `/etc/piwheels.conf`
- `/usr/local/etc/piwheels.conf`
- `~/.config/piwheels/piwheels.conf`

-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 connection string for the database to use; this database must be initialized with *piw-initdb* (page 19) 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; this reduces some timeouts and tweaks some defaults

--debug TASK
: Set logging to debug level for the named task; can be specified multiple times to debug many tasks

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

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

--pypi-json URL
: The URL of the PyPI JSON API (default: `https://pypi.org/pypi`)

--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 21), *piw-add* (page 25), *piw-remove* (page 27), and *piw-rebuild* (page 23) (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 29) (default: `ipc:///tmp/piw-logger`); this should always be an ipc address

--slave-queue ADDR
: The address of the queue used to talk to *piw-slave* (page 11) (default: `tcp://*:5555`); this is usually a tcp address

--file-queue ADDR
: The address of the queue used to transfer files to *piw-slave* (page 11) (default: `tcp://*:5556`); this is usually a tcp address

--web-queue ADDR
: The address of the queue used to request web page updates (default: `inproc://web`)

--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`)

[3] https://postgresql.org/

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 apache2 python3-configargparse python3-zmq \
              python3-voluptuous python3-cbor2 python3-requests \
              python3-sqlalchemy python3-psycopg2 python3-chameleon \
              python3-simplejson python3-urwid python3-geoip python3-pip
# pip3 install "piwheels[monitor,master,logger]"
```

   If you wish to install directly from the git repository:

```
# apt install git
# pip3 install git+https://github.com/piwheels/piwheels#egg=piwheels[monitor,
↪master,logger]
```

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 configuration file:

   Listing 1: /etc/piwheels.conf

```
[master]
dsn=postgresql:///piwheels
output-path=/var/www/piwheels
```

4. Set up the database:

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

5. 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). This part isn't optional; you won't get `pip` installing things from an unencrypted source without a lot of pain.
   - See below for an example Apache configuration

6. 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
```

7. Deploy some build slaves; see *piw-slave* (page 11) for deployment instructions.

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

2.4 Example httpd configuration

The following is an example Apache configuration similar to that used on the production piwheels master. The port 80 (http) server configuration should look something like this:

Listing 2: /etc/apache2/sites-available/000-default.conf

```
<VirtualHost *:80>
    ServerName www.example.org
    ServerAlias example.org
    RedirectMatch 302 ^(.*) https://www.example.org$1
</VirtualHost>
```

Note: Obviously, you will want to replace all instances of "example.org" with your own server's domain.

On the port 443 (https) side of things, you want the "full" configuration which should look something like this, assuming your output path is `/var/www/piwheels`:

Listing 3: /etc/apache2/sites-available/default-ssl.conf

```
<IfModule mod_ssl.c>
    <VirtualHost _default_:443>
        ServerName www.example.org
        ServerAlias example.org
        ServerAdmin webmaster@example.org
        DocumentRoot /var/www/piwheels

        ErrorLog ${APACHE_LOG_DIR}/ssl_error.log
        CustomLog ${APACHE_LOG_DIR}/ssl_access.log combined
        # Send Apache log records to piw-logger for transfer to piw-master
        CustomLog "|/usr/local/bin/piw-logger --drop" combined

        SSLEngine On
        SSLCertificateFile /var/lib/dehydrated/certs/example.org/fullchain.pem
        SSLCertificateKeyFile /var/lib/dehydrated/certs/example.org/privkey.pem

        <Directory /var/www/piwheels>
            Options -Indexes +FollowSymlinks
            AllowOverride None
            Require all granted
            <IfModule mod_rewrite.c>
                RewriteEngine On
                RewriteRule ^project/?$ /packages.html [L,R=301]
                RewriteRule ^p/(.*)/?$ /project/$1 [L,R=301]
            </IfModule>
            <IfModule mod_headers.c>
                Header set Access-Control-Allow-Origin "*"
            </IfModule>
            ErrorDocument 404 /404.html
            DirectoryIndex index.html
        </Directory>

        <Directory /var/www/piwheels/logs/>
            Options +MultiViews
            MultiviewsMatch Any
            RemoveType .gz
            AddEncoding gzip .gz
            <IfModule mod_filter.c>
                FilterDeclare gzip CONTENT_SET
                FilterProvider gzip INFLATE "! req('Accept-Encoding') =~ /gzip/"
```

(continues on next page)

(continued from previous page)

```
                FilterChain gzip
            </IfModule>
        </Directory>
    </VirtualHost>
</IfModule>
```

Several important things to note:

- A CustomLog[6] line pipes log entries to the *piw-logger* (page 29) script which buffers entries and passes them to piw-master for insertion into the database (which in turn is used to generate statistics for the homepage and the project pages)
- Only `index.html` is allowed as a directory index, no directory listings are generated (they can be enormous, and remember the master is expected to be deployable on a Raspberry Pi)
- There's a couple of mod_rewrite[7] redirections to deal with legacy path redirections, and providing a more friendly root for the `/project/` path
- The build logs are stored in pre-compressed gzip archives, and the server is configured to serve them verbatim to clients which provide an Accept-Encoding: gzip[8] header. For clients which do not (e.g. *curl(1)*), the server unpacks the log transparently
- An example configuration for the SSL certificate locations is given which assumes dehydrated[9] is being used to maintain them

2.5 Example database configuration

The following sections detail various setups for the database server. The simplest is the first, the combined configuration in which the machine hosting the master service also hosts the database.

The later sections detail separating the master and database hosts, and assume your master server is accessible at the IPv6 address `1234:abcd::1` and your database server is at the IPv6 address `1234:abcd::2`. Replace addresses accordingly.

2.5.1 Combined configuration

This is effectively covered in the prior deployment section. The default DSN of `dsn=postgresql:///piwheels` can either be implied by default, or explicitly specified in `/etc/piwheels.conf`.

The only thing to be aware of, particularly if you are deploying on a Pi, is that the calculation of the build queue is quite a big query. Assuming you are targeting all packages on PyPI (as the production piwheels instance does), you should never consider running the combined database+master on a machine (or VM) with less than 4 cores and 4GB of RAM, preferably more. If deploying a combined master+database on a Pi, use a Pi 4 with 8GB of RAM.

[6] https://httpd.apache.org/docs/2.4/mod/mod_log_config.html#customlog
[7] https://httpd.apache.org/docs/2.4/mod/mod_rewrite.html
[8] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Encoding
[9] https://github.com/lukas2511/dehydrated

2.5.2 Separate configuration

If you wish to deploy your PostgreSQL database on a separate server, you will first need to ensure that server can accept remote connections from the master server. A simple (but less secure) means of configuring this is to simply "trust" that connections from the master's IP address to the piwheels database by the piwheels user. This can be accomplished by adding the last line below to `pg_hba.conf`:

Listing 4: /etc/postgresql/**ver**/main/pg_hba.conf

```
# Database administrative login by Unix domain socket
local   all             postgres                                peer

# TYPE  DATABASE        USER            ADDRESS                 METHOD

# "local" is for Unix domain socket connections only
local   all             all                                     peer
# IPv4 local connections:
host    all             all             127.0.0.1/32            md5
# IPv6 local connections:
host    all             all             ::1/128                 md5
# Allow replication connections from localhost, by a user with the
# replication privilege.
local   replication     all                                     peer
host    replication     all             127.0.0.1/32            md5
host    replication     all             ::1/128                 md5
host    piwheels        piwheels        1234:abcd::1/128        trust
```

Then restarting the PostgreSQL server:

```
# systemctl restart postgresql
```

Then, on the master, use the following DSN in `/etc/piwheels.conf`:

Listing 5: /etc/piwheels.conf

```
[master]
dsn=postgresql://piwheels@[1234:abcd::2]/piwheels
```

Warning: *Never* provide remote access to the PostgreSQL superuser, `postgres`. Install the piwheels package directly on the database server and run the *piw-initdb* (page 19) script locally. This will also require creating a `/etc/piwheels.conf` on the database server, that uses a typical "local" DSN like `dsn=postgresql:///piwheels`.

2.5.3 SSH tunnelling

A more secure (but rather more complex) option is to create a persistent SSH tunnel from the master to the database server which forwards the UNIX socket for the database back to the master as the unprivileged `piwheels` user.

Firstly, on the master, generate an SSH key-pair for the `piwheels` user and copy the public key to the database server.

```
# su - piwheels
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/piwheels/.ssh/id_rsa):
Created directory '/home/piwheels/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

(continues on next page)

(continued from previous page)

```
Your identification has been saved in /home/piwheels/.ssh/id_rsa
Your public key has been saved in /home/piwheels/.ssh/id_rsa.pub
...
$ ssh-copy-id piwheels@1234:abcd::2
```

Note: This assumes that you *temporarily* permit password-based login for the piwheels user on the database server.

Secondly, set up a `systemd(1)` service to maintain the tunnel:

Listing 6: /etc/systemd/system/piwheelsdb-tunnel.service

```
[Unit]
Description=A secure tunnel for the piwheelsdb connection
After=local-fs.target network.target

[Service]
User=piwheels
Group=piwheels
RuntimeDirectory=postgresql
RuntimeDirectoryPreserve=restart
ExecStart=/usr/bin/ssh -NT \
  -o BatchMode=yes \
  -o ExitOnForwardFailure=yes \
  -o StreamLocalBindUnlink=yes \
  -L /run/postgresql/.s.PGSQL.5432:/run/postgresql/.s.PGSQL.5432 \
  piwheels@1234:abcd::2
RestartSec=5
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

```
# systemctl daemon-reload
# systemctl enable piwheelsdb-tunnel.service
# systemctl start piwheelsdb-tunnel.service
```

At this point, you should be able to switch back to the piwheels user and connect to the piwheels database (however, note that as the tunnel is owned by the unprivileged piwheels user, only it can access the database remotely):

```
# su - piwheels
$ psql piwheels
psql (13.5 (Debian 13.5-0+deb11u1))
Type "help" for help.

piwheels=>
```

Note: This method requires *no* alteration of `pg_hba.conf` on the database server; the default should be sufficient. As far as the database server is concerned the local piwheels user is simply accessing the database via the local UNIX socket.

2.6 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:

Listing 7: /etc/systemd/system/piwheels-master.service

```
[Unit]
Description=The piwheels master service
After=local-fs.target network.target

[Service]
Type=notify
Restart=on-failure
User=piwheels
NoNewPrivileges=true
TimeoutStartSec=3m
TimeoutStopSec=5m
ExecStart=/usr/local/bin/piw-master -v
ExecStartPost=-chmod g+w /tmp/piw-status /tmp/piw-control

[Install]
WantedBy=multi-user.target
```

```
# systemctl daemon-reload
# systemctl enable piwheels-master
# systemctl start piwheels-master
```

2.7 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 19) script as the PostgreSQL super-user to upgrade the database between versions (downgrades are not supported, so take a backup first!).

CHAPTER

THREE

PIW-SLAVE

The piw-slave script is intended to be run on a standalone machine to build packages on behalf of the *piw-master* (page 3) service. 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 (see *`signal(7)`*), or by the remote *piw-master* (page 3) script.

3.1 Synopsis

```
piw-slave [-h] [--version] [-c FILE] [-q] [-v] [-l FILE] [--debug]
          [-m HOST] [-t DURATION] [-d DIR] [-L STR]
```

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

`--debug`
Set logging to debug level

`-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)

`-d` `DIR`, **`--dir`** `DIR`
The temporary directory to use when building wheels (default: /tmp)

`-L` `STR`, **`--label`** `STR`
The label to transmit to the master identifying this build slave (default: the local hostname)

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:

Listing 1: deploy_slave.sh

```
#!/bin/bash

set -eu

if [ $# -ne 2 ]; then
    echo "Usage: deploy_slave.sh HOSTNAME MASTER_IP"
    exit 1
fi

echo $1 > /etc/hostname
echo "[slave]" > /etc/piwheels.conf
echo "master=$2" >> /etc/piwheels.conf

DEBIAN_FRONTEND=noninteractive

sed -i 's/#PasswordAuthentication.*/PasswordAuthentication no/' /etc/ssh/sshd_
↪config

source /etc/os-release

LIBXLST=libxslt1-dev
LIBGLES=libgles2-mesa-dev
SOUNDFONT=timgm6mb-soundfont
POSTGRES_SERVER_DEV=postgresql-server-dev-15
QMAKE=qt5-qmake
FPRINT=libfprint-2-dev

if [ $VERSION_ID -eq 10 ]; then
    QMAKE=qt4-qmake
    POSTGRES_SERVER_DEV=postgresql-server-dev-11
    FPRINT=libfprint-dev
elif [ $VERSION_ID -eq 11 ]; then
    POSTGRES_SERVER_DEV=postgresql-server-dev-13
fi

apt update
apt -y upgrade
apt -y install vim wget curl ssh-import-id tree byobu htop pkg-config cmake time␣
↪pandoc \
    gfortran ipython3 git qt5-qmake python3-dev python3-pip python3-apt \
    zlib1g-dev libpq-dev libffi-dev libxml2-dev libhdf5-dev libldap2-dev \
    libjpeg-dev libbluetooth-dev libusb-dev libhidapi-dev libfreetype6-dev \
    liblcms2-dev libzbar-dev libbz2-dev libblas-dev liblapack-dev \
    liblapacke-dev libcurl4-openssl-dev libgmp-dev libgstreamer1.0-dev \
    libsdl2-dev libsdl2-image-dev libsdl2-mixer-dev libsdl2-ttf-dev libssl-dev \
    libsasl2-dev libldap2-dev libavcodec-dev libavformat-dev libswscale-dev \
    libv4l-dev libxvidcore-dev libx264-dev libgtk2.0-dev libgtk-3-dev \
    libatlas-base-dev python3-numpy python3-cairocffi libsdl-image1.2-dev \
    libsdl-mixer1.2-dev libsdl-ttf2.0-dev libsdl1.2-dev libportmidi-dev \
    libtiff5-dev libx11-6 libx11-dev xfonts-base xfonts-100dpi xfonts-75dpi \
    xfonts-cyrillic fluid-soundfont-gm libsystemd-dev libusb-1.0-0-dev \
    libudev-dev libopus-dev libvpx-dev libc-bin libavdevice-dev libadios-dev \
    libavfilter-dev libavutil-dev libcec-dev lsb-release pybind11-dev \
```

(continues on next page)

(continued from previous page)

```
    libsnappy-dev libpcap0.8-dev swig libzmq5 portaudio19-dev libqpdf-dev \
    coinor-libipopt-dev libsrtp2-dev default-libmysqlclient-dev golang \
    libgeos-dev $LIBGLES $LIBXLST $SOUNDFONT $POSTGRES_SERVER_DEV \
    $QMAKE $FPRINT libgphoto2-dev libsqlite3-dev libsqlcipher-dev \
    ninja-build libgirepository1.0-dev libfmt-dev libopenblas-dev

apt purge python3-cryptography python3-yaml -y

pip3 install setuptools --upgrade --break-system-packages
pip3 install pip --upgrade --break-system-packages
hash -r

pip3 install pypandoc versioneer kervi scikit-build cython numpy scipy \
    setuptools_rust conan cbor2 \
    --upgrade --extra-index-url https://www.piwheels.org/simple --prefer-binary --
↪break-system-packages

getent passwd piwheels && userdel -fr piwheels
getent group piwheels || groupadd piwheels
getent passwd piwheels || useradd -g piwheels -m -s /bin/bash piwheels
passwd -d piwheels

curl -sSf 'https://sh.rustup.rs' | runuser -- - piwheels -s -- -y --profile␣
↪minimal --default-host arm-unknown-linux-gnueabihf

if [ -d piwheels ]; then
    cd piwheels
    git pull
    pip3 uninstall -y piwheels
else
    git clone https://github.com/piwheels/piwheels
    cd piwheels
fi

cp piwheels-slave.service /etc/systemd/system/
systemctl enable piwheels-slave.service
pip3 install .[slave] --break-system-packages

fallocate -x -l 1G /swapfile
chmod 0600 /swapfile
mkswap /swapfile
echo "/swapfile none swap x-systemd.makefs,nofail 0 0" >> /etc/fstab
systemctl daemon-reload

rm -f /etc/pip.conf

byobu-enable

reboot
```

```
# chmod +x deploy_slave.sh
# ./deploy_slave.sh myname 1234:abcd::1
```

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, the machine should reboot, automatically connect to the master node and start building packages as and when the build queue becomes populated. If you need to manually execute the application (assuming the master's IP address is `1234:abcd::1`:

```
# su - piwheels
$ piw-slave -m 1234:abcd::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:

Listing 2: /etc/systemd/system/piwheels-slave.service

```
[Unit]
Description=The piwheels slave service
After=local-fs.target network.target

[Service]
Type=notify
WatchdogSec=2min
StartLimitInterval=5min
StartLimitBurst=4
StartLimitAction=reboot-force
Restart=on-failure
User=piwheels
PrivateTmp=true
NoNewPrivileges=true
ExecStart=/usr/local/bin/piw-slave -v

[Install]
WantedBy=multi-user.target
```

```
# 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

FOUR

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 32) to stop querying PyPI, *Slave Driver* (page 33) 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 33)), 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

FIVE

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 15)).

- 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

SIX

PIW-INITDB

The piw-initdb script is used to initialize or upgrade the piwheels master database. The target PostgreSQL[10] 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

-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

[10] https://postgresql.org/

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

SEVEN

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 22) 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 22) 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

EIGHT

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* (page 3) service.

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)
in-dex	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 23) 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 34) 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

NINE

PIW-ADD

The piw-add script is used to manually add new packages (and versions of packages) to the system. This script must be run on the same node as the piw-master script.

9.1 Synopsis

```
piw-add [-h] [--version] [-c FILE] [-q] [-v] [-l FILE] [-y] [-s REASON]
             [--unskip] [-d TEXT] [-a NAME] [-r TIMESTAMP] [--yank]
             [--unyank] [--import-queue ADDR]
             package [version]
```

9.2 Description

package
    The name of the package to add

version
    The version of the package to add; if omitted, adds the package only

-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 package or version with a skip reason to prevent build attempts

`--unskip`
Remove a skip reason for the package or version to enable build attempts

`-d` `TEXT`, **`--description`** `TEXT`
The package description; defaults to retrieving the description from PyPI

`-a` `ALIAS`, **`--alias`** `ALIAS`
Any package aliases to use; may be specified multiple times

`-r` `TIMESTAMP`, **`--released`** `TIMESTAMP`
The version's release date (can only be provided for a new version, cannot be updated); defaults to now

`--yank`
Mark the version as yanked (can only be applied to a new version - use *piw-remove* (page 27) to yank a known version

`--unyank`
Mark a known version as not yanked

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

9.3 Usage

This utility is intended to permit administrators to tweak the content of the database to correct issues that arise from either incorrect scraping of the PyPI history, inadvertent mistakes made with *piw-remove* (page 27), or other inconsistencies found in the database.

The utility can be run in a batch mode with `--yes` (page 25) but still requires invoking once per addition required (you cannot define multiple packages or versions in a single invocation).

The return code will be 0 if the package (or version) was successfully added to the database. If anything fails, the return code will be non-zero and the database should remain unchanged.

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

TEN

PIW-REMOVE

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

By default, the package (or version) removed will be deleted entirely (*not* marked to skip). Optionally, you can provide a skip reason; in this case the version will not be deleted (though its builds and files will), but will be left marked to skip to prevent future rebuilds.

10.1 Synopsis

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

10.2 Description

package
: The name of the package to remove

version
: The version of the package to remove. If omitted, removes the entire package

-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
Leave the version in place, but marked 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

10.3 Usage

This utility is typically used in response to a request from a package maintainer to remove a specific build from the system. Usually because the presence of a piwheels build is causing issues in and of itself.

Note: Older versions of piwheels didn't heed PyPI deletion messages. This is no longer the case and this utility is no longer required to manually remove deleted packages.

The utility can be run in a batch mode with *--yes* (page 27) 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 package (or 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

ELEVEN

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.

11.1 Synopsis

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

11.2 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[11] script

11.3 Usage

This utility is typically used to pipe logs from a web-server, such as Apache[12] 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[13] directive as follows, within the Apache virtual host responsible 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
```

[11] http://httpd.apache.org/docs/current/logs.html#piped
[12] https://httpd.apache.org/
[13] http://httpd.apache.org/docs/current/mod/mod_log_config.html#customlog

CHAPTER

TWELVE

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.

12.1 Testing

Executing the test suite requires that you have a local PostgreSQL[14] 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`.

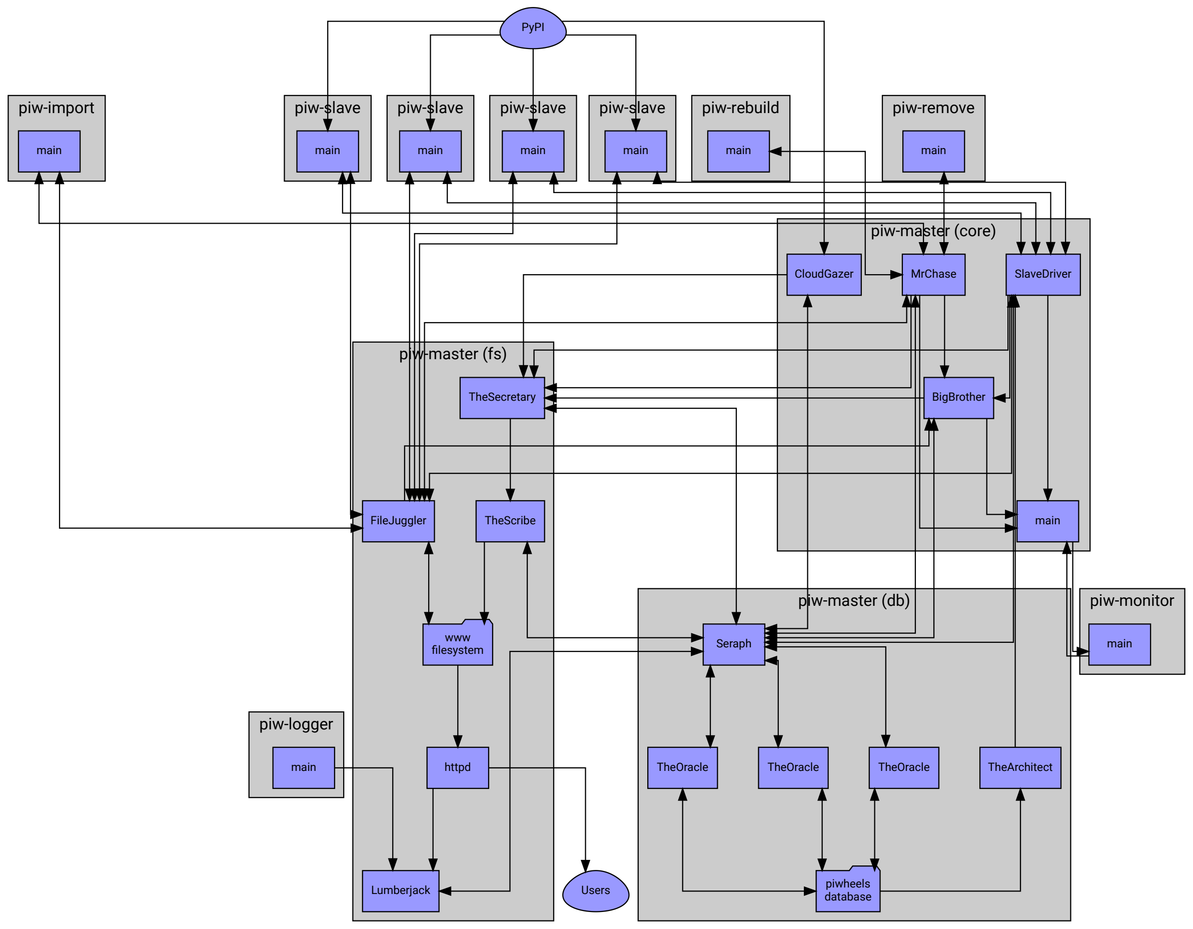
Note: If you are not using your local PostgreSQL installation for anything else you may wish to set `fsync=off` and `synchronous_commit=off` in your local `postgresql.conf` to speed up execution of the test suite. Do *NOT* do this on any production PostgreSQL server!

12.2 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[15] sockets. This is also how communication occurs between the master and the *piw-slave* (page 11), 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:

[14] https://postgresql.org/
[15] 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).

12.3 Tasks

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

12.3.1 Cloud Gazer

Implemented in: `piwheels.master.cloud_gazer.CloudGazer` (page 49).

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 33).

12.3.2 The Oracle

Implemented in: *piwheels.master.the_oracle.TheOracle* (page 49).

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 33).

12.3.3 Seraph

Implemented in: *piwheels.master.seraph.Seraph* (page 53).

Seraph is a simple load-balancer for the various instances of *The Oracle* (page 33). 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.

12.3.4 The Architect

Implemented in: *piwheels.master.the_architect.TheArchitect* (page 53).

This task is the final database related task in the master script. Unlike *The Oracle* (page 33) it periodically queries the database for the packages that need building and passes this information along to the *Slave Driver* (page 33).

12.3.5 Slave Driver

Implemented in: *piwheels.master.slave_driver.SlaveDriver* (page 54).

This task is the main coordinator of the build slaves' 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. If there is a pending package matching the build slave's ABI, it will be told to build that package.

Periodically, *The Architect* (page 33) refreshes this task's list of packages that require building.

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 33). If the build was successful, it informs the *File Juggler* (page 34) 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 34) that the package's index and project page will need (re)writing. It also periodically informs *Big Brother* (page 34) of the size of the build queue.

12.3.6 Mr. Chase

Implemented in: *piwheels.master.mr_chase.MrChase* (page 56).

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 33) with a correspondingly simpler protocol. It is also the end-point for **piw-rebuild** and **piw-remove**.

This task writes information to the database via *The Oracle* (page 33). If the imported build was successful, it informs the *File Juggler* (page 34) 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 34) that the package's index and project pages will need (re)writing.

12.3.7 File Juggler

Implemented in: *piwheels.master.file_juggler.FileJuggler* (page 57).

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 33)).

12.3.8 Big Brother

Implemented in: *piwheels.master.big_brother.BigBrother* (page 59).

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 34).

12.3.9 The Scribe

Implemented in: *piwheels.master.the_scribe.TheScribe* (page 60).

This task generates the web output for piwheels. It generates the home-page with statistics from *Big Brother* (page 34), the overall package index, individual package file lists, and project pages with messages from *Slave Driver* (page 33).

12.3.10 The Secretary

Implemented in *piwheels.master.the_secretary.TheSecretary* (page 59).

This task sits in front of *The Scribe* (page 34) 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 (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 34) for processing.

12.4 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 33) and *File Juggler* (page 34) tasks (documented in the *piw-slave* (page 11) 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[16].

[16] https://cbor.io/

12.5 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).

For example the message telling a build slave what package and version to build looks like this in Python syntax:

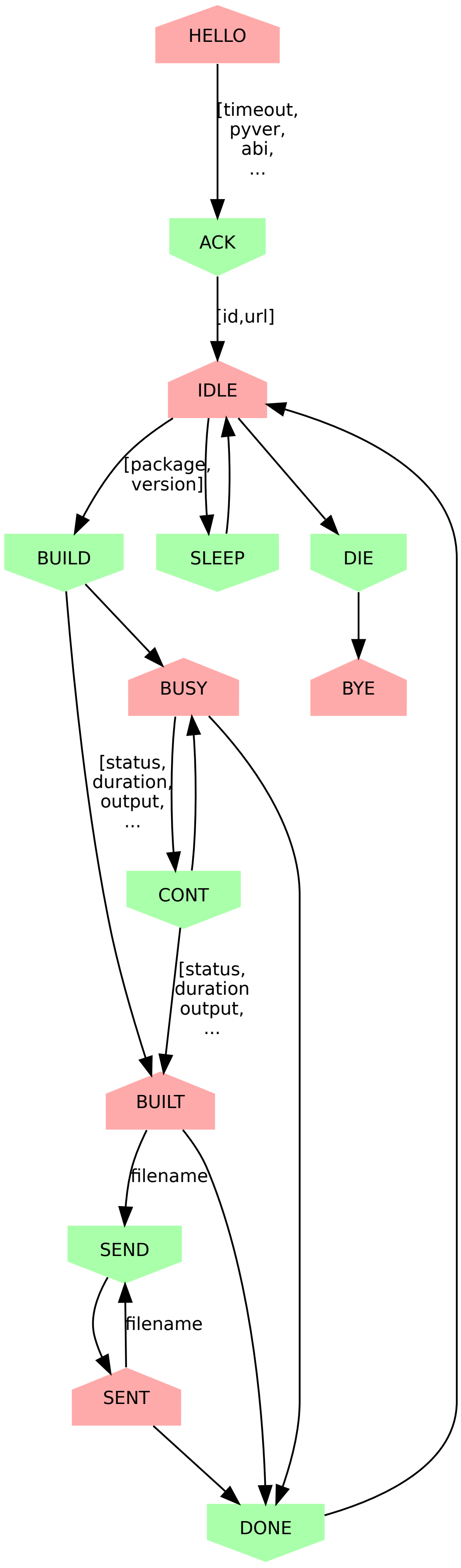
```
['BUILD', 'numpy', '1.14.0']
```

If a message is not associated with any data whatsoever, it is transmitted as a simple unicode string (without the list encapsulation). The serialization format for all messages in the system is currently CBOR[17].

12.5.1 Slave Driver

The queue that talks to *Slave Driver* (page 33) is a ZeroMQ REQ socket, hence the protocol follows a strict request-reply sequence which is illustrated below:

[17] https://cbor.io/


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

1. The new build slave sends “HELLO” with data `[build_timeout, master_timeout, py_version_tag, abi_tag, platform_tag, label, os_name, os_version, board_revision, board_serial]` where:

   - *build_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`[18].
   - *master_timeout* is the maximum length of time the slave will wait for communication from the master. After this timeout it will assume the connection has failed, terminate and clean-up any on-going build, then attempt to restart the connection to the master.
   - *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
   - *os_name* is a string identifying the operating system, e.g. “Raspbian GNU/Linux”.
   - *os_version* is a string identifying the release of the operating system, e.g. “10 (buster)”.
   - *board_revision* is a code indicating the revision of the board that the slave is running upon, e.g. “c03111” for a Raspberry Pi 4B.
   - *board_serial* is the serial number of the board that the slave is running upon.

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. The “IDLE” message is accompanied with the data `[now, disk_total, disk_free, mem_total, mem_free, load_avg, cpu_temp]` where:

   - *now* is a `datetime`[19] indicating the current time on the build slave.
   - *disk_total* is the total size (in bytes) of the file-system used to build wheels.
   - *disk_free* is the number of bytes free in the file-system used to build wheels.
   - *mem_total* is the total size (in bytes) of the RAM on the build slave.
   - *mem_free* is the number of bytes of RAM currently available (not necessarily unused, but potentially usable by builds).
   - *load_avg* is the one minute load average.
   - *cpu_temp* is the temperature, in degrees celsius of the CPU.

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 with “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).

[18] https://docs.python.org/3.9/library/datetime.html#datetime.timedelta
[19] https://docs.python.org/3.9/library/datetime.html#datetime.datetime

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. While the build is underway, the slave must periodically ping the master with the "BUSY" message, which is accompanied by the exact same stats as in the "IDLE" message.

8. If the master wishes the build slave to continue with the build it will reply with "CONT". If the master wants to build slave to terminate the build early it will reply with "DONE" (goto step 13).

9. Assuming the master doesn't request termination of the build, eventually it will finish. In response to the next "CONT" message, the slave sends "BUILT" with data `[status, duration, output, files]`:

   - *status* is `True` if the build succeeded and `False` otherwise.
   - *duration* is a `timedelta`[20] 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`[21] 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.
     - *dependencies* is a `set`[22] 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

10. 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.

11. At this point the slave should use the *File Juggler* (page 42) 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.

12. 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".

13. The build slave is now free to destroy all resources associated with the build, and returns to step 3 ("IDLE").

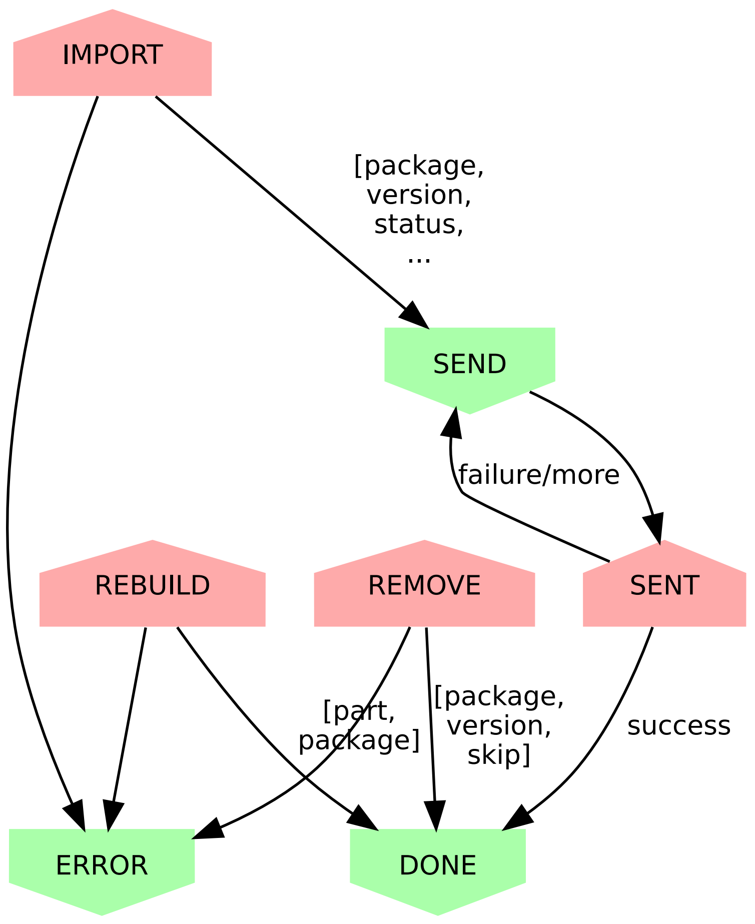
If at any point, the master takes longer than *master_timeout* (default: 5 minutes) to respond to a slave's request, the slave will assume the master has disappeared. If a build is still active, it will be cleaned up and terminated, the connection to the master will be closed, the slave's ID will be 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 misinterpreted 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 *master_timeout* has elapsed.

[20] https://docs.python.org/3.9/library/datetime.html#datetime.timedelta
[21] https://docs.python.org/3.9/library/stdtypes.html#list
[22] https://docs.python.org/3.9/library/stdtypes.html#set

12.5.2 Mr Chase (importing)

The queue that talks to *Mr. Chase* (page 33) 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):



1. The importer sends "IMPORT" with data `[slave_id, package, version, abi_tag, status, duration, output, files]`:
   - *slave_id* is the integer id of the build slave that created the wheel. This is usually 0 and is ignored by the master anyway.
   - *package* is the name of the package that the build is for.
   - *version* is the version of the package that the build is for.
   - *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.
   - *status* is `True` if the build succeeded and `False` otherwise.
   - *duration* is a `float`[23] 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.

[23] https://docs.python.org/3.9/library/functions.html#float

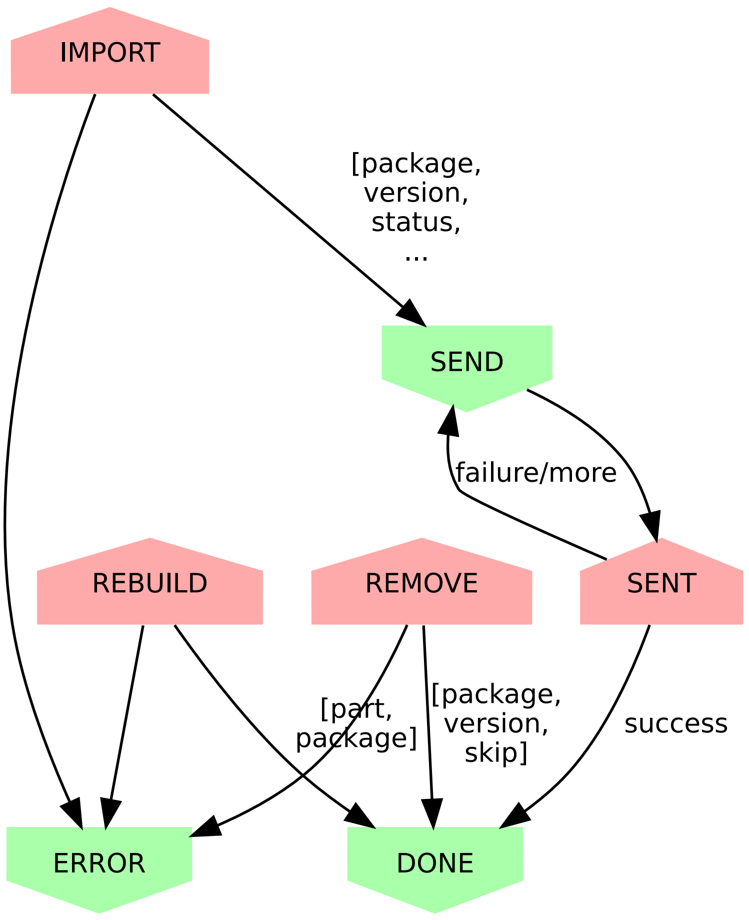
    - *package_version_tag* is the version tag extracted from the filename.
    - *py_version_tag* is the python version tag extracted from the filename.
    - *abi_tag* is the ABI tag extracted from the filename (sanitized).
    - *platform_tag* is the platform tag extracted from the filename.
    - *dependencies* is a `set`[24] 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 42) 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.

12.5.3 Mr Chase (removing)

The queue that talks to *Mr. Chase* (page 33) 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):

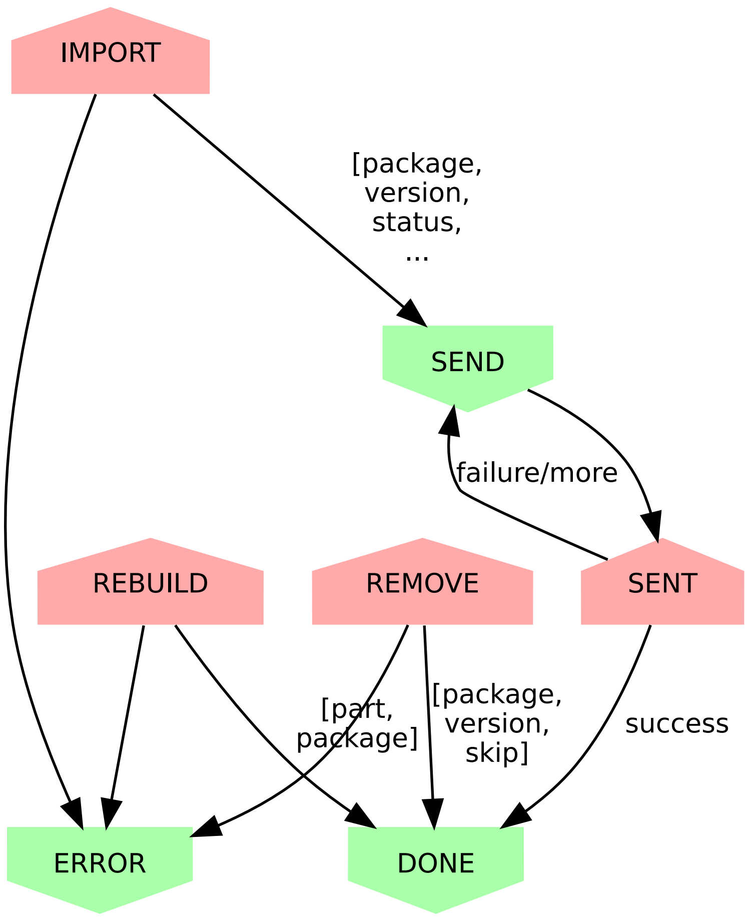
[24] https://docs.python.org/3.9/library/stdtypes.html#set



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.
   - *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".

12.5.4 Mr Chase (rebuilding)

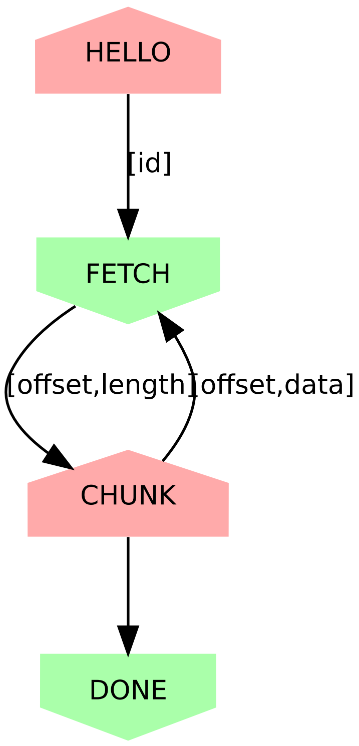
The queue that talks to *Mr. Chase* (page 33) is a ZeroMQ REQ socket, hence the protocol follows a strict request-reply sequence which is illustrated below (see above for documentation of the `IMPORT` path):



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", "PROJECT" or "BOTH".
   - *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".

12.5.5 File Juggler

The queue that talks to *File Juggler* (page 34) 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.

12.6 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

THIRTEEN

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.

13.1 piwheels.master

Defines the *PiWheelsMaster* (page 46) 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**).

broadcast_status()

Publish messages from the internal status queue to the external status queue, in case any monitors are attached.

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 the master's HELLO message and all the build slave's HELLO messages replayed to it.

do_kill(*slave_id*)

Handler for the KILL message; this tells the specified build slave (or all slaves and the master if *slave_id* is `None`) to terminate.

do_quit()

Handler for the QUIT message; this terminates the master.

do_skip(*slave_id*)

Handler for the SKIP message; this tells the specified build slave to skip its current build next time it contacts the master.

do_sleep(*slave_id*)

Handler for the SLEEP message; this tells the specified build slave (or all slaves and the master if *slave_id* is `None`) to pause their operations.

do_wake(*slave_id*)

Handler for the WAKE message; this tells the specified build slave (or all slaves and the master if *slave_id* is `None`) to resume their operations.

main_loop(*systemd*)

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`[25] which will cause the *PiWheelsMaster.main_loop()* (page 46) method to terminate.

[25] https://docs.python.org/3.9/library/exceptions.html#SystemExit

13.2 piwheels.master.db

This module defines the low level database API, `Database` (page 47). 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*, *skip=''*, *description=''*)

Insert a new package record into the database. Returns True if the row was inserted successfully, or False if a key violation occurred.

`add_new_package_version`(*package*, *version*, *released=None*, *skip=''*)

Insert a new package version record into the database. Returns True if the row was inserted successfully, or False if a key violation occurred.

`add_package_name`(*package*, *name*, *seen*)

Add a new package alias or update the last seen timestamp.

`delete_build`(*package*, *version*)

Remove all builds for the specified package and version, along with all files records.

`delete_package`(*package*)

Remove the specified package, along with all builds and files.

`delete_version`(*package*, *version*)

Remove the specified version of the specified package, along with all builds and files.

`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_abis`(*exclude_skipped=False*)

Return a set of ABIs. If **exclude_skipped** is `False`, return all ABIs from the build_abis table, otherwise return only active ABIs (not skipped).

`get_build_queue`(*limit=1000*)

Returns a mapping of ABI tags to an ordered list of up to *limit* package version tuples which currently need building for that ABI.

`get_package_aliases`(*package*)

Retrieve all aliases for *package* (not including the canonical name itself).

`get_package_files`(*package*)

Returns a mapping of filenames to file hashes; this is all the data required to build the simple index.html for the specified package.

`get_project_data`(*package*)

Returns all details required to build the project page of the specified *package*.

`get_project_display_name`(*package*)

Retrieve the last seen name for *package*.

`get_pypi_serial`()

Return the serial number of the last PyPI event.

`get_search_index`()

Return a mapping of all packages to their download count for the last month. This is used to construct the searchable package index.

`get_statistics`()
: Return various build related statistics from the database.

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

`get_version_skip`(*package*, *version*)
: Returns the reason for skipping *version* of *package*.

`get_versions_deleted`(*package*)
: Return any versions of *package* which have been marked for deletion.

`load_rewrites_pending`()
: Loads the rewrites-pending queue (the internal state of `TheSecretary`) from the database.

`log_build`(*build*)
: Log a build attempt in the database, including wheel info if successful.

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

`log_json`(*json*)
: Log a project's JSON page hit in the database.

`log_page`(*page*)
: Log a web page hit in the database.

`log_project`(*project*)
: Log a project page hit in the database.

`log_search`(*search*)
: Log a search in the database, including data derived from JSON in pip's user-agent.

`package_marked_deleted`(*package*)
: Check whether *package* has been marked for deletion.

`save_rewrites_pending`(*queue*)
: Save the rewrites-pending queue (the internal state of `TheSecretary`) in the database. The *queue* parameter is expected to be a list of `RewritePendingRow` tuples.

`set_package_description`(*package*, *description*)
: Update the description for *package* in the packages table.

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

`skip_package`(*package*, *reason*)
: Mark a package with a reason to prevent future builds of all versions (and all future versions).

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

`test_package`(*package*)
: Check whether *package* already exists in the database. Returns a boolean.

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

`unyank_version`(*package*, *version*)
: Mark the specified version of the specified package version as "unyanked".

`yank_version`(*package*, *version*)
: Mark the specified version of the specified package version as "yanked".

13.3 piwheels.master.cloud_gazer

Defines the `CloudGazer` (page 49) 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 53) to use.

`once`()

This method is called once before the task loop starts. It the task needs to do some initialization or setup within the task thread, this is the place to do it.

13.4 piwheels.master.the_oracle

Defines `TheOracle` (page 49) task and the `DbClient` (page 51) 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 54), `TheScribe` (page 60), and `CloudGazer` (page 49).

Note that because database requests are notoriously variable in length the client RPC class (`DbClient` (page 51)) doesn't *directly* talk to `TheOracle` (page 49). Rather, multiple instances of `TheOracle` (page 49) are spawned and `Seraph` (page 53) sits in front of these acting as a simple load-sharing router for the RPC clients.

`close`()

Close all registered queues. This should be overridden to close any additional queues the task holds which aren't registered.

`do_allpkgs`()

Handler for "ALLPKGS" message, sent by `DbClient` (page 51) to request the set of all packages define known to the database.

`do_allvers`()

Handler for "ALLVERS" message, sent by `DbClient` (page 51) 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 51) to remove all builds (and files and downloads by cascade) for *version* of *package*.

`do_delpkg`(*package*)

Handler for "DELPKG" message, sent by `DbClient` (page 51) to delete a package.

`do_delver`(*package*, *version*)

Handler for "DELVER" message, sent by `DbClient` (page 51) to delete a specific version of a package.

`do_getabis`(*exclude_skipped*)

Handler for "GETABIS" message, sent by `DbClient` (page 51) to request the list of all ABIs to build for.

`do_getpkgnames`(*package*)

Handler for "GETPKGNAMES" message, sent by `DbClient` (page 51) to retrieve all aliases for *package* (not including the canonical name itself).

`do_getpypi`()

Handler for "GETPYPI" message, sent by `DbClient` (page 51) to request the record of the last serial number from the PyPI changelog.

do_getsearch()

Handler for "GETSEARCH" message, sent by `DbClient` (page 51) to request the recent download statistics, returned as a mapping of package to (downloads_recent, downloads_all) tuples.

do_getskip(*package*, *version*)

Handler for "GETSKIP" message, send by `DbClient` (page 51) to request the reason for skipping builds of *version* of *package*.

do_getstats()

Handler for "GETSTATS" message, sent by `DbClient` (page 51) to request the latest database statistics, returned as a list of (field, value) tuples.

do_loadrwp()

Handler for "LOADRWP" message, sent by `DbClient` (page 51) to request the content of the `rewrites_pending` table.

do_logbuild(*build*)

Handler for "LOGBUILD" message, sent by `DbClient` (page 51) to register a new build result.

do_logdownload(*download*)

Handler for "LOGDOWNLOAD" message, sent by `DbClient` (page 51) to register a new download.

do_logjson(*json*)

Handler for "LOGJSON" message, sent by `DbClient` (page 51) to register a new project JSON download.

do_logpage(*page*)

Handler for "LOGPAGE" message, sent by `DbClient` (page 51) to register a new web page hit.

do_logproject(*project*)

Handler for "LOGPROJECT" message, sent by `DbClient` (page 51) to register a new project page hit.

do_logsearch(*search*)

Handler for "LOGSEARCH" message, sent by `DbClient` (page 51) to register a new search.

do_newpkg(*package*, *skip*, *description*)

Handler for "NEWPKG" message, sent by `DbClient` (page 51) to register a new package.

do_newpkgname(*package*, *name*, *seen*)

Handler for "NEWPKGNAME" message, sent by `DbClient` (page 51) to register a new package alias or update the last seen timestamp.

do_newver(*package*, *version*, *released*, *skip*)

Handler for "NEWVER" message, sent by `DbClient` (page 51) to register a new (package, version) tuple.

do_pkgdeleted(*package*)

Handler for "PKGDELETED" message, sent by `DbClient` (page 51) to request whether or not the specified *package* has been marked for deletion.

do_pkgexists(*package*)

Handler for "PKGEXISTS" message, sent by `DbClient` (page 51) to request whether or not the specified *package* exists.

do_pkgfiles(*package*)

Handler for "PKGFILES" message, sent by `DbClient` (page 51) to request details of all wheels associated with *package*.

do_projdata(*package*)

Handler for "PROJDATA" message, sent by `DbClient` (page 51) to request details of *package* and all its releases.

do_saverwp (*queue*)

Handler for "SAVERWP" message, sent by `DbClient` (page 51) to request that *queue* is saved to the `rewrites_pending` table.

do_setdesc (*package*, *description*)

Handler for "SETDESC" message, sent by `DbClient` (page 51) to update a package's project description.

do_setpypi (*serial*)

Handler for "SETPYPI" message, sent by `DbClient` (page 51) to update the last seen serial number from the PyPI changelog.

do_skippkg (*package*, *reason*)

Handler for "SKIPPKG" message, sent by `DbClient` (page 51) to skip building all versions of a package.

do_skipver (*package*, *version*, *reason*)

Handler for "SKIPVER" message, sent by `DbClient` (page 51) to skip building a specific version of a package.

do_unyankver (*package*, *version*)

Handler for "UNYANKVER" message, sent by `DbClient` (page 51) to mark a specific version of a package as not "yanked".

do_verexists (*package*, *version*)

Handler for "VEREXISTS" message, sent by `DbClient` (page 51) to request whether or not the specified *version* of *package* exists.

do_verfiles (*package*, *version*)

Handler for "VERFILES" message, sent by `DbClient` (page 51) to request the filenames of all wheels associated with *version* of *package*.

do_versdeleted (*package*)

Handler for "VERSDELETED" message, sent by `DbClient` (page 51) to request any versions for *package* which have been marked for deletion.

do_yankver (*package*, *version*)

Handler for "YANKVER" message, sent by `DbClient` (page 51) to mark a specific version of a package as "yanked".

handle_db_request (*queue*)

Handle incoming requests from `DbClient` (page 51) instances.

class piwheels.master.the_oracle.**DbClient** (*config*, *logger=None*)

RPC client class for talking to `TheOracle` (page 49).

add_new_package (*package*, *skip=''*, *description=''*)

See `db.Database.add_new_package()` (page 47).

add_new_package_version (*package*, *version*, *released=None*, *skip=''*)

See `db.Database.add_new_package_version()` (page 47).

add_package_name (*package*, *name*, *seen=datetime.datetime(1970, 1, 1, 0, 0, tzinfo=datetime.timezone.utc)*)

See `db.Database.add_package_name()` (page 47).

delete_build (*package*, *version*)

See `db.Database.delete_build()` (page 47).

delete_package (*package*)

See `db.Database.delete_package()` (page 47).

delete_version(*package*, *version*)
 See `db.Database.delete_version()` (page 47).

get_all_package_versions()
 See `db.Database.get_all_package_versions()` (page 47).

get_all_packages()
 See `db.Database.get_all_packages()` (page 47).

get_build_abis(*, *exclude_skipped=False*)
 See `db.Database.get_build_abis()` (page 47).

get_package_aliases(*package*)
 See `db.Database.get_package_aliases()` (page 47).

get_package_files(*package*)
 See `db.Database.get_package_files()` (page 47).

get_project_data(*package*)
 See `db.Database.get_project_data()` (page 47).

get_pypi_serial()
 See `db.Database.get_pypi_serial()` (page 47).

get_search_index()
 See `db.Database.get_search_index()` (page 47).

get_statistics()
 See `db.Database.get_statistics()` (page 47).

get_version_files(*package*, *version*)
 See `db.Database.get_version_files()` (page 48).

get_version_skip(*package*, *version*)
 See `db.Database.get_version_skip()` (page 48).

get_versions_deleted(*package*)
 See `db.Database.get_versions_deleted()` (page 48).

load_rewrites_pending()
 See `db.Database.load_rewrites_pending()` (page 48).

log_build(*build*)
 See `db.Database.log_build()` (page 48).

log_download(*download*)
 See `db.Database.log_download()` (page 48).

log_json(*json*)
 See `db.Database.log_json()` (page 48).

log_page(*page*)
 See `db.Database.log_page()` (page 48).

log_project(*project*)
 See `db.Database.log_project()` (page 48).

log_search(*search*)
 See `db.Database.log_search()` (page 48).

package_marked_deleted(*package*)
 See `db.Database.package_marked_deleted()` (page 48).

save_rewrites_pending (*queue*)
See `db.Database.save_rewrites_pending()` (page 48).

set_package_description (*package, description*)
See `db.Database.update_project_description()`.

set_pypi_serial (*serial*)
See `db.Database.set_pypi_serial()` (page 48).

skip_package (*package, reason*)
See `db.Database.skip_package()` (page 48).

skip_package_version (*package, version, reason*)
See `db.Database.skip_package_version()` (page 48).

test_package (*package*)
See `db.Database.test_package()` (page 48).

test_package_version (*package, version*)
See `db.Database.test_package_version()` (page 48).

unyank_version (*package, version*)
See `db.Database.unyank_version()` (page 48).

yank_version (*package, version*)
See `db.Database.yank_version()` (page 48).

13.5 piwheels.master.seraph

Defines the `Seraph` (page 53) task; see class for more details.

class piwheels.master.seraph.**Seraph** (*config*)
This task is a simple load-sharing router for `TheOracle` (page 49) tasks.

handle_back (*queue*)
Receive a response from an instance of `TheOracle` (page 49) 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 51) requests from the front queue and send it on to the worker including the client's address frame.

13.6 piwheels.master.the_architect

Defines `TheArchitect` (page 53) 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 pushes the results to `SlaveDriver` (page 54) to sort out.

close ()
Close all registered queues. This should be overridden to close any additional queues the task holds which aren't registered.

quit ()
Overridden to cancel any existing long-running query.

update_build_queue()

The architect simply runs the build queue query repeatedly, with a break of a minute between each execution.

All entries found within this limit are sorted into per-ABI queues and pushed to `SlaveDriver` (page 54) which queues and dispatches jobs to build ABI-matched slaves as they become available.

13.7 piwheels.master.slave_driver

Defines the `SlaveDriver` (page 54) 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 46) method passes this information on to any listening monitors). Also, the internal "indexes" queue is informed of any packages that need web page indexes rebuilding (as a result of a successful build).

close()

Close all registered queues. This should be overridden to close any additional queues the task holds which aren't registered.

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 57) 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_busy(*slave*)

Handler for the build slave's "BUSY" message, which is sent periodically during package builds. If the slave fails to respond with a BUSY ping for a duration longer than `SlaveState.busy_timeout` then the master will assume the slave has died and remove it from the internal state mapping (if the slave happens to resurrect itself later the master will simply treat it as a new build slave).

In response to "BUSY" the master can respond "CONT" to indicate the build should continue processing, or "DONE" to indicate that the build slave should immediately terminate and discard the build and return to "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 74)) – 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 74)) – 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 74)) – 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 `TheScribe` 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_build`(*queue*)

Refresh the ABI-specific queues of package versions waiting to be built. The queues are limited to 1000 packages per ABI, and are kept as lists ordered by release date. When a message arrives from `TheArchitect` it refreshes (replaces) all current queues. There is, however, still a duplication possibility as `TheArchitect` doesn't know what packages are actively being built; this method handles filtering out such packages.

Even if the active builds fail (because build slaves crash, or the network dies) this doesn't matter as a future re-run of the build queue query will return these packages again, and if no build slaves are actively working on them at that time they will then be retried.

`handle_control`(*queue*)

Handle incoming requests to the internal control queue.

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

`handle_delete`(*queue*)

Handle package or version deletion requests.

When the PyPI upstream deletes a version or package, the `CloudGazer` task requests that other tasks perform the deletion on its behalf. In the case of this task, this involves cancelling any pending builds of that package (version), and ignoring any builds involving that package (version) in the next queue update from `TheArchitect`.

`handle_slave`(*queue*)

Handle requests from build slaves.

See the *piw-slave* (page 11) chapter for an overview of the protocol for messages between build slaves and *`SlaveDriver`* (page 54). This method retrieves the message from the build slave, finds the associated *`SlaveState`* (page 74) 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 `handle_control()` (page 55) for more information.

list_slaves()

Additional task control method to trigger a "HELLO" message to the internal control queue. See `quit()` (page 71) for more information.

remove_expired()

Remove slaves which have exceeded their timeout.

skip_slave(*slave_id*)

Additional task control method to trigger a "SKIP" message to the internal control queue. See `handle_control()` (page 55) for more information.

sleep_slave(*slave_id*)

Additional task control method to trigger a "SLEEP" message to the internal control queue. See `handle_control()` (page 55) for more information.

wake_slave(*slave_id*)

Additional task control method to trigger a "WAKE" message to the internal control queue. See `handle_control()` (page 55) for more information.

13.8 piwheels.master.mr_chase

Defines the `MrChase` (page 56) 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.

close()

Close all registered queues. This should be overridden to close any additional queues the task holds which aren't registered.

do_add_package(*state*)

Handler for the remover's "ADDPKG" message, indicating a request to add a package to the system, or update it.

do_add_package_aliases(*package*, *aliases*)

Add aliases for a package name

do_add_version(*state*)

Handler for the remover's "ADDVER" message, indicating a request to add a specific version of a package to the system, or update it.

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_rebuild(*state*)

Handler for the rebuilder's "REBUILD" message, indicating a request to rebuild part of the website.

do_remove_package(*state*)

Handler for the remover's "REMPKG" message, indicating a request to remove or alter a whole package.

do_remove_version (*state*)

Handler for the remover's "REMVER" message, indicating a request to remove or alter a specific package version.

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 mesasge is sent to `TheScribe` 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 21) and *piw-remove* (page 27) chapters for an overview of the protocol for messages between the importer and *MrChase* (page 56).

13.9 piwheels.master.file_juggler

Defines the *FileJuggler* (page 57) task and the *FsClient* (page 58) 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 58)).

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 72) 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 54) 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 54) 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 74)) – The object representing the state of the transfer.
- **msg** (*str*[26]) – 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 58) to inform file juggler that a build slave is about to start a file transfer. The message includes the full `FileState` (page 72). The state is stored in the `pending` map.

Parameters

- **slave_id** (`int`[27]) – The identity of the build slave about to begin the transfer.
- **file_state** (`list`[28]) – The details of the file to be transferred including the expected hash.

do_verify (*slave_id*, *package*)

Message sent by `FsClient` (page 58) 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`[29]) – The identity of the build slave that sent the file.
- **package** (`str`[30]) – 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.

The file transfer protocol is in some ways very simple (see the chart in the *piw-slave* (page 11) 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 58) 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 57)).

Parameters

- **msg** (`str`[31]) – The message sent to start the transfer (must be "HELLO")
- ***args** – All additional arguments (expected to be an integer slave id).

once ()

This method is called once before the task loop starts. It the task needs to do some initialization or setup within the task thread, this is the place to do it.

class `piwheels.master.file_juggler.`**FsClient** (*config*, *logger=None*)

RPC client class for talking to `FileJuggler` (page 57).

expect (*slave_id*, *file_state*)

See `FileJuggler.do_expect()` (page 57).

verify (*slave_id*, *package*)

See `FileJuggler.do_verify()` (page 58).

[26] https://docs.python.org/3.9/library/stdtypes.html#str
[27] https://docs.python.org/3.9/library/functions.html#int
[28] https://docs.python.org/3.9/library/stdtypes.html#list
[29] https://docs.python.org/3.9/library/functions.html#int
[30] https://docs.python.org/3.9/library/stdtypes.html#str
[31] https://docs.python.org/3.9/library/stdtypes.html#str

13.10 piwheels.master.big_brother

Defines the *BigBrother* (page 59) 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 46) uses to pass statistics to any listening monitors.

close()

Close all registered queues. This should be overridden to close any additional queues the task holds which aren't registered.

handle_control(*queue*)

Handle incoming requests to the internal control queue.

This just adds handling for the custom STATS verb to replay the master stats history.

13.11 piwheels.master.the_secretary

Defines the *TheSecretary* (page 59) task; see class for more details.

class piwheels.master.the_secretary.**TheSecretary**(*config*)

This task buffers requests for the scribe, for the purpose of consolidating multiple consecutive (duplicate) requests.

Requests to write the project page for a package (which is a relatively expensive operation in terms of database accesses) can come in thick and fast, particularly when a new version is being registered with lots of files. There's little point in writing the project page 5 times in as many seconds, or writing the project page, then the index and project page immediately afterward. This class is used to buffer requests for up to a minute, allowing us to eliminate many of the duplicate requests.

close()

Close all registered queues. This should be overridden to close any additional queues the task holds which aren't registered.

handle_input(*queue*)

Handle incoming write requests with buffering and de-dupe.

Some incoming requests (currently "HOME", "SEARCH", "LOG", "DELPKG", and "DELVER") are passed directly through to `TheScribe` as these are either sufficiently rare ("HOME", "SEARCH") that no benefit is gained by buffering them or sufficiently urgent ("DELPKG", "DELVER", "LOG") that they must be acted on immediately.

For other requests ("PROJECT" and "BOTH"), requests can come thick and fast in the case of multiple file registrations picked up by `CloudGazer`. In this case, requests are buffered for a minute and de-duplicated; e.g. if several requests are made to re-write the project page for package "foo" within that period, they will be combined into a single request. After the minute of buffering, the request is passed down to `TheScribe`.

handle_output()

Passes buffered requests downstream.

This sub-task runs periodically to pluck things from the internal buffer that have reached the minute delay, and passes them downstream to `TheScribe`. The process stops when we run out of things that have expired.

once()

This method is called once before the task loop starts. It the task needs to do some initialization or setup within the task thread, this is the place to do it.

13.12 piwheels.master.the_scribe

Defines the `TheScribe` (page 60) task; see class for more details.

class `piwheels.master.the_scribe.`**`TheScribe`**(*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).

`close`()

Close all registered queues. This should be overridden to close any additional queues the task holds which aren't registered.

`delete_package`(*package*)

Attempts to remove the index and project page directories (including all known wheel files) of the specified *package*.

Parameters

package (`str`[32]) – The name of the package to delete.

`delete_version`(*package*, *version*)

Attempts to remove any known wheel files corresponding with deleted *versions* of the specified *package*.

Parameters

- **package** (`str`[33]) – The name of the package to delete files for.
- **version** (`str`[34]) – The version of *package* to delete files for.

`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
- "BOTH", a request to write the index and project page for the specified package
- "PROJECT", a request to write just the project page for the specified package
- "LOG", a request to write a build log

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.

`once`()

This method is called once before the task loop starts. It the task needs to do some initialization or setup within the task thread, this is the place to do it.

`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`(*statistics*)

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

Parameters

statistics (`dict`[35]) – A dict containing statistics obtained by `BigBrother`.

write_log (*build_id*, *log*)

Attempts to write the *log* of build *build_id* to the log output directories, splitting the numeric build id into three parts to flatten the output hierarchy. Log data is also gzip compressed.

write_package_index (*package*, *data*)

(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.

The *data* parameter is expected to be the dictionary of package data returned by `db.Database.get_project_data()` (page 47). This is expected to have at least the following content in the example case of a package named "foo" with version "1.0" containing a validly built wheel:

```
{
    'releases': {
        '1.0': {
            'files': {
                'foo-1.0-py3-none-any.whl': {
                    'hash': 'abcdef1234567890...',
                    'requires_python': '>= 3.6',
                },
            },
            'yanked': False,
        },
    },
}
```

Parameters

- **package** (`str`[36]) – The name of the package to write the index page for.
- **data** (`dict`[37]) – The dictionary of data returned by `db.Database.get_project_data()` (page 47) which is expected to have at least the structure documented above.

write_pages (*package*, *, *both=False*, *exclude=None*)

(Re)writes the project page and project JSON file (and simple index if *both* is True) for the specified *package*.

Parameters

- **package** (`str`[38]) – The name of the package to write the pages for
- **both** (`bool`[39]) – Write both the project page and the simple page if True, otherwise only write the project page. Note project page also includes project JSON.
- **exclude** (`set`[40] `or None`) – The set of (deleted) versions to exclude from pages. Defaults to `None`.

write_project_json (*package*, *data*)

(Re)writes the project JSON data of the specified package.

The *data* parameter is expected to be the dictionary of package data returned by `db.Database.get_project_data()` (page 47). This is expected to have at least the following content in the example case of a package named "foo" with version "1.0" containing a validly built wheel:

```
{
    'name': 'foo',
    'description': 'A foomatic package',
    'releases': {
        '1.0': {
            'abis': {
                'cp35m': {
                    'build_id': 1,
```

(continues on next page)

(continued from previous page)

```
                    'status': 'success',
                    'skip': '',
                }
            }
            'files': {
                'foo-1.0-py3-none-any.whl': {
                    'hash': 'abcdef1234567890...',
                    'size': 123456,
                    'apt_dependencies': {'libc6'},
                },
            },
            'released': datetime(2000, 1, 1, 12, 34, 56),
            'yanked': False,
            'skip': '',
        },
    },
}
```

Parameters

- **package** (*str*[41]) – The name of the package to write the project data for.
- **data** (*dict*[42]) – The dictionary of data returned by *db.Database.get_project_data()* (page 47) which is expected to have at least the structure documented above.

write_project_page (*package*, *data*)

(Re)writes the project page of the specified package.

The *data* parameter is expected to be the dictionary of package data returned by *db.Database.get_project_data()* (page 47). This is expected to have at least the following content in the example case of a package named "foo" with version "1.0" containing a validly built wheel:

```
{
    'name': 'foo',
    'description': 'A foomatic package',
    'releases': {
        '1.0': {
            'abis': {
                'cp35m': {
                    'build_id': 1,
                    'status': 'success',
                    'skip': '',
                }
            }
            'files': {
                'foo-1.0-py3-none-any.whl': {
                    'hash': 'abcdef1234567890...',
                    'size': 123456,
                    'apt_dependencies': {'libc6'},
                },
            },
            'released': datetime(2000, 1, 1, 12, 34, 56),
            'yanked': False,
            'skip': '',
        },
    },
}
```

Parameters

- **package** (*str*[43]) – The name of the package to write the project page for.
- **data** (*dict*[44]) – The dictionary of data returned by *db.Database.get_project_data()* (page 47) which is expected to have at least the structure documented above.

write_search_index(*search_index*)

Re-writes the JSON search index using the provided statistics.

Parameters

search_index (*dict*[45]) – A dict mapping package names to their download count obtained by `BigBrother`.

write_simple_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.

write_sitemap()

(Re)writes the XML sitemap pages and index.

13.13 piwheels.slave

Defines the *PiWheelsSlave* (page 63) 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 0MQ sockets) to a master (see **piw-master**) then loops around the slave protocol (see the *piw-slave* (page 11) chapter). It retrieves source packages directly from PyPI[46], attempts to build a wheel in a sandbox directory and, if successful, transmits the results to the master.

clean_up_build(*timeout=datetime.timedelta(seconds=60)*)

Terminate any existing build and clean up its temporary storage. Raises an exception if termination does not occur in a reasonable time.

do_ack(*new_id*, *pypi_url*)

In response to our initial "HELLO" (detailing our various **PEP 425**[47] tags), the master is expected to send "ACK" 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_build(*package*, *version*)

Alternatively, in response to "IDLE", the master may send "BUILD" *package version*. We should then attempt to build the specified wheel and send back a "BUSY" message with more status info.

[32] https://docs.python.org/3.9/library/stdtypes.html#str
[33] https://docs.python.org/3.9/library/stdtypes.html#str
[34] https://docs.python.org/3.9/library/stdtypes.html#str
[35] https://docs.python.org/3.9/library/stdtypes.html#dict
[36] https://docs.python.org/3.9/library/stdtypes.html#str
[37] https://docs.python.org/3.9/library/stdtypes.html#dict
[38] https://docs.python.org/3.9/library/stdtypes.html#str
[39] https://docs.python.org/3.9/library/functions.html#bool
[40] https://docs.python.org/3.9/library/stdtypes.html#set
[41] https://docs.python.org/3.9/library/stdtypes.html#str
[42] https://docs.python.org/3.9/library/stdtypes.html#dict
[43] https://docs.python.org/3.9/library/stdtypes.html#str
[44] https://docs.python.org/3.9/library/stdtypes.html#dict
[45] https://docs.python.org/3.9/library/stdtypes.html#dict

do_cont()

Once we're busy building something the master will periodically ping us with "CONT" if it wishes us to continue building. We wait up to 10 seconds for the build to finish and reply with "BUILT" (and a full status report on the build) if it finishes in that time, or "BUSY" and more status info otherwise.

If the master wishes to terminate a build prior to completion, it'll send "DONE" instead of "CONT" and we move straight to clean-up.

do_die()

The master may respond with "DIE" at any time indicating we should immediately terminate. We raise `SystemExit`[48] to cause *main_loop()* (page 64) to exit. Clean-up of any extant build is handled by our caller.

do_done()

The master can send "DONE" at any point during a built to terminate it prematurely. Alternately, this is also the standard response after a successful build has finished and all files have been sent (and successfully verified).

In response we must clean-up all resources associated with the build (including terminating an on-going build) and return "IDLE" with the usual stats.

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.Wheel.transfer()* (page 65) for more details). Once the transfers concludes, reply to the master with "SENT".

do_sleep(*paused*)

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.

get_status()

Returns the set of statistics periodically required by the master when reporting our status.

handle_reply(*msg*, *data*)

Dispatch a message from the master to an appropriate handler method.

main_loop(*queue*, *master_timeout=datetime.timedelta(seconds=300)*)

The main messaging loop. Sends the initial request, and dispatches replies via *handle_reply()* (page 64). Implements a *timeout* for responses from the master and raises `MasterTimeout` if *timeout* seconds are exceeded.

piwheels.slave.**duration**(*s*)

Convert *s*, a string representing a duration, into a `datetime.timedelta`[49].

13.14 piwheels.slave.builder

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

class piwheels.slave.builder.**Wheel**(*path*, *dependencies=None*)

Records the state of a build artifact, i.e. a wheel package. The filename is deconstructed into the fields specified by **PEP 425**[50].

Parameters

- **path** (*pathlib.Path*[51]) – The path to the wheel on the local filesystem.

[46] https://pypi.python.org/
[47] https://peps.python.org/pep-0425/
[48] https://docs.python.org/3.9/library/exceptions.html#SystemExit
[49] https://docs.python.org/3.9/library/datetime.html#datetime.timedelta

- **dependencies** (*dict*[52]) – A dict mapping tool to dependencies that are required to use these particular wheel files. Defaults to a `None` (no dependencies).

as_message()

Return the state as a list suitable for use in the `BUILT` message of **piw-slave**.

open()

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 57) protocol.

property abi_tag

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

property build_tag

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

property dependencies

Return the dependencies required by the wheel as a mapping of dependency system (e.g. "apt", "pip", etc.) to set of package names for that system.

property filehash

Return an SHA256 digest of the wheel's contents.

property filename

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

property filesize

Return the size of the wheel in bytes.

property metadata

Return the contents of the `METADATA` file inside the wheel.

property package_canon

Return the package part of the wheel's filename, canonicalized according to PyPI's rules.

property package_tag

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

property package_version_tag

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

property platform_tag

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

property py_version_tag

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

property requires_python

Return the contents of the `Requires-Python` specification from the wheel metadata.

class piwheels.slave.builder.**Builder**(*package*, *version*, *, *timeout=datetime.timedelta(seconds=300)*, *index_url='https://pypi.python.org/simple'*, *extra_index_urls={'https://www.piwheels.org/simple'}*, *dir=None*)

[50] https://peps.python.org/pep-0425/
[51] https://docs.python.org/3.9/library/pathlib.html#pathlib.Path
[52] https://docs.python.org/3.9/library/stdtypes.html#dict

Class responsible for building wheels for a given *version* of a *package*. Note that this class derives from `Thread`[53] and hence is expected to run in the background after calling `start()`[54].

Parameters

- **`package`** (*`str`*[55]) – The name of the package to attempt to build wheels for.
- **`version`** (*`str`*[56]) – The version of the package to attempt to build.
- **`timeout`** (*`datetime.timedelta`*[57]) – The number of seconds to wait for `pip` to finish before raising `subprocess.TimeoutExpired`[58].
- **`index_url`** (*`str`*[59]) – The URL of the **PEP 503**[60] compliant repository from which to fetch packages for building.
- **`extra_index_urls`** (*`set`*[61]) – The URLs of any additional **PEP 503**[62] compliant repositories from which to fetch packages.
- **`dir`** (*`str`*[63]) – The directory in which to store wheel and log output.

`as_message`()

Return the state as a list suitable for use in the `BUILT` message of **`piw-slave`**.

`build_command`(*log_file*)

Generate the pip command line used to run the build.

`build_dependencies`(*wheel*)

Calculate the apt dependencies of *wheel* (which is a *`Wheel`* (page 64) instance representing a built wheel).

`build_environment`()

Configure the environment for the build.

`build_wheel`(*log_file*)

Call pip and attempt to build the wheel; handle killing the subprocess if termination is requested, and watch the clock for a build timeout.

`close`()

Remove the temporary build directory and all its contents.

`run`()

Attempt to build the package within the configured timeout.

`stop`()

Tell the build to stop prematurely.

`property duration`

The `timedelta`[64] indicating how long the actual build took (without any extraneous tasks like dependency calculation). This is an indication of how long a user would spend installing the package without piwheels.

`property extra_index_urls`

The URLs of any additional indexes from which the builder will also check when retrieving packages. This is intended to be used for fetching compiled platform wheels for specified *build dependencies*.

`property index_url`

The URL of primary index from which the builder will attempt to obtain the source to build.

`property output`

The log output from the build.

`property package`

The package that the builder will attempt to build.

property status

A `bool`[65] indicating if the build succeeded or failed. If the build is still on-going, returns `None`[66].

property timeout

The `timedelta`[67] after which the builder will assume the build has failed.

property version

The version of *`package`* (page 66) that the builder will attempt to build.

property wheels

A list of *`Wheel`* (page 64) instances generated by the build.

13.15 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`[68] 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`[69] 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 67)) 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`,

[53] https://docs.python.org/3.9/library/threading.html#threading.Thread
[54] https://docs.python.org/3.9/library/threading.html#threading.Thread.start
[55] https://docs.python.org/3.9/library/stdtypes.html#str
[56] https://docs.python.org/3.9/library/stdtypes.html#str
[57] https://docs.python.org/3.9/library/datetime.html#datetime.timedelta
[58] https://docs.python.org/3.9/library/subprocess.html#subprocess.TimeoutExpired
[59] https://docs.python.org/3.9/library/stdtypes.html#str
[60] https://peps.python.org/pep-0503/
[61] https://docs.python.org/3.9/library/stdtypes.html#set
[62] https://peps.python.org/pep-0503/
[63] https://docs.python.org/3.9/library/stdtypes.html#str
[64] https://docs.python.org/3.9/library/datetime.html#datetime.timedelta
[65] https://docs.python.org/3.9/library/functions.html#bool
[66] https://docs.python.org/3.9/library/constants.html#None
[67] https://docs.python.org/3.9/library/datetime.html#datetime.timedelta
[68] https://docs.python.org/3.9/library/exceptions.html#RuntimeError
[69] https://docs.python.org/3.9/library/exceptions.html#RuntimeError

`"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.

13.16 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 required information that *`MrChase`* (page 56) needs.

`piwheels.importer.`**`print_state`**`(`*state*`)`

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

Parameters

`state` (`BuildState` (page 73)) – The build state to print the description of.

`piwheels.importer.`**`do_import`**`(`*config*, *packages*, *state*`)`

Handles constructing and sending the initial "IMPORT" message to `master.mr_chase.MrChase`. If "SEND" is then received, uses *`do_send()`* (page 68) to handle transmitting files.

Parameters

- **`config`** – The configuration obtained from parsing the command line.
- **`packages`** (*`list`*[70]) – A sequence of `Wheel` objects corresponding to files in the *state*.
- **`state`** (`BuildState` (page 73)) – The object representing the state of the build.

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

Handles sending files when requested by *`do_import()`* (page 68).

13.17 piwheels.add

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

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

This is the main function for the **`piw-add`** script. It uses *`MrChase`* (page 56) to add mew packages and versions to the system.

`piwheels.add.`**`do_add`**`(`*config*`)`

Handles constructing and sending the ADDPKG/ADDVER message to *`MrChase`* (page 56).

Parameters

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

[70] https://docs.python.org/3.9/library/stdtypes.html#list

13.18 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 56) to remove built packages from the system.

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

Handles constructing and sending the REMPKG/REMVER message to *`MrChase`* (page 56).

Parameters

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

13.19 piwheels.transport

This module augments the classes provided by pyzmq (the 0MQ Python bindings) to use CBOR encoding, and voluptuous for message validation. It also tweaks a few minor things like using seconds for timeouts.

`class` `piwheels.transport.`**`Context`**

Wrapper for 0MQ `zmq.Context`. This extends the `socket()` method to include parameters for the socket's protocol and logger.

`class` `piwheels.transport.`**`Socket`**`(`*socket*, *protocol=None*, *logger=None*`)`

Wrapper for `zmq.Socket`. This extends 0MQ's sockets to include a protocol which will be used to validate messages that are sent and received (via a voluptuous schema), and a logger which can be used to debug socket behaviour.

`bind``(`*address*`)`

Binds the socket to listen on the specified *address*.

`close``(`*linger=None*`)`

Closes the socket. If *linger* is specified, it is the number of seconds to wait for pending messages to be flushed.

`connect``(`*address*`)`

Connects the socket to the listening socket at *address*.

`drain``()`

Receives all pending messages in the queue and discards them. This is typically useful during shutdown routines or for testing.

`poll``(`*timeout=None*, *flags=zmq.POLLIN*`)`

Polls the socket for pending data (by default, when *flags* is POLLIN). If no data is available after *timeout* seconds, returns False. Otherwise returns True.

If *flags* is POLLOUT instead, tests whether the socket has available slots for queueing new messages.

`recv``(`*flags=0*`)`

Receives the next message as a `bytes`[71] string.

`recv_addr_msg``(`*flags=0*`)`

Receive a CBOR-encoded message (and associated data) along with the address it came from (represented as a `bytes`[72] string).

`recv_msg``(`*flags=0*`)`

Receive a CBOR-encoded message, returning a tuple of the unicode message string and its associated data. This is the primary method used in piwheels for receiving information into a task.

The message, and its associated data, will be validated against the `protocol` associated with the socket on construction.

recv_multipart (*flags=0*)

Receives a multi-part message, returning its content as a list of `bytes`[73] strings.

send (*buf*, *flags=0*)

Send *buf* (a `bytes`[74] string).

send_addr_msg (*addr*, *msg*, *data=NoData*, *flags=0*)

Send a CBOR-encoded message (and associated data) to *addr*, a `bytes`[75] string.

send_msg (*msg*, *data=NoData*, *flags=0*)

Send the unicode string *msg* with its associated *data* as a CBOR-encoded message. This is the primary method used in piwheels for sending information between tasks.

The message, and its associated data, must validate against the `protocol` associated with the socket on construction.

send_multipart (*msg_parts*, *flags=0*)

Send *msg_parts*, a list of `bytes`[76] strings as a multi-part message which can be received intact with `recv_multipart()` (page 69).

subscribe (*topic*)

Subscribes SUB type sockets to the specified *topic* (a string prefix).

unsubscribe (*topic*)

Unsubscribes SUB type sockets from the specified *topic* (a string prefix).

property hwm

The high-water mark of the socket, i.e. the number of messages that can be queued before the socket blocks (or drops, depending on the socket type) messages.

class piwheels.transport.**Poller**

Wrapper for 0MQ `zmq.Poller`. This simply tweaks 0MQ's poller to use seconds for timeouts, and to return a `dict`[77] by default from `poll()` (page 70).

poll (*timeout=None*)

Poll all registered sockets for the events they were registered with, for *timeout* seconds. Returns a dictionary mapping sockets to events or an empty dictionary if the *timeout* elapsed with no events occurring.

register (*sock*, *flags=None*)

Register *sock* with the poller, watching for events as specified by *flags* (which defaults to POLLIN and POLLOUT events).

unregister (*sock*)

Unregister *sock* from the poller. After this, calls to `poll()` (page 70) will never return references to *sock*.

[71] https://docs.python.org/3.9/library/stdtypes.html#bytes
[72] https://docs.python.org/3.9/library/stdtypes.html#bytes
[73] https://docs.python.org/3.9/library/stdtypes.html#bytes
[74] https://docs.python.org/3.9/library/stdtypes.html#bytes
[75] https://docs.python.org/3.9/library/stdtypes.html#bytes
[76] https://docs.python.org/3.9/library/stdtypes.html#bytes
[77] https://docs.python.org/3.9/library/stdtypes.html#dict

13.20 piwheels.tasks

Implements the base classes (`Task` (page 71) and its derivative `PauseableTask` (page 72)) which form the basis of all the tasks in the piwheels master.

exception `piwheels.tasks.`**`TaskQuit`**

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

class `piwheels.tasks.`**`Task`**(*config*, *control_protocol=Protocol(recv={'PAUSE': NoData, 'RESUME': NoData, 'QUIT': NoData}, send={})*)

The `Task` (page 71) class is a `Thread`[78] derivative which is the base for all tasks in the piwheels master. The `run()` (page 72) method is overridden to perform a simple task loop which calls `poll()` (page 71) once a cycle to react to any messages arriving into queues, and to dispatch any periodically executed methods.

Queues are associated with handlers via the `register()` (page 71) method. Periodic methods are associated with an interval via the `every()` (page 71) method. These should be called during initialization (don’t attempt to register handlers from within the thread itself).

Generally this shouldn’t be used as a base-class. Use one of the descendents that implements a pausing mechanism, `NonStopTask`, `PauseableTask` (page 72), or `PausingTask`.

`close`()

Close all registered queues. This should be overridden to close any additional queues the task holds which aren’t registered.

`every`(*interval*, *handler*)

Register *handler* to be called every *interval* periodically.

Parameters

- **`interval`** (`timedelta`) – The time interval between each run of *handler*.
- **`handler`** – The function or method to call periodically.

`force`(*handler*)

Force *handler* to run next time its interval is polled.

`handle_control`(*queue*)

Default handler for the internal control queue. In this base class it simply handles the “QUIT” message by raising `TaskQuit` (page 71) (which the `run()` (page 72) method will catch and use as a signal to end).

Messages other than QUIT, PAUSE and RESUME raise `TaskControl` which can be caught in descendents to implement custom control messages.

`once`()

This method is called once before the task loop starts. It the task needs to do some initialization or setup within the task thread, 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=1*)

This method is called once per loop of the task’s `run()` (page 72) method. It runs all periodic handlers, then 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=zmq.POLLIN*)

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** (`transport.Socket` (page 69)) – The queue to poll.
- **handler** – The function or method to call when a message with matching *flags* arrives in *queue*.
- **flags** (*int*[79]) – 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.

socket (*sock_type*, *protocol=None*)

Construct a socket and link it to the logger for this task. This is primarily useful for debugging purposes, but also ensures that the task will implicitly close and clean up the socket when it closes.

class piwheels.tasks.**PauseableTask** (*config*, *control_protocol=Protocol(recv={'PAUSE': NoData, 'RESUME': NoData, 'QUIT': NoData}, send={})*)

Derivative of *Task* (page 71) 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()` and *Task.poll()* (page 71) will not be called) until the task is resumed (or terminated).

If you need a more complex pausing implementation which can still do some work while paused (to drain incoming queues for instance), use `PausingTask` instead.

handle_control (*queue*)

Default handler for the internal control queue. In this base class it simply handles the "QUIT" message by raising *TaskQuit* (page 71) (which the `run()` method will catch and use as a signal to end).

Messages other than QUIT, PAUSE and RESUME raise `TaskControl` which can be caught in descendents to implement custom control messages.

13.21 piwheels.states

This module defines several classes which permit interested tasks to track the state of build slaves (*SlaveState* (page 74)), file transfers (*TransferState* (page 74)), build attempts (*BuildState* (page 73)), build artifacts (*FileState* (page 72)) and various loggers.

class piwheels.states.**FileState** (*filename*, *filesize*, *filehash*, *package_tag*, *package_version_tag*, *py_version_tag*, *abi_tag*, *platform_tag*, *requires_python*, *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*[80]) – The original filename of the build artifact.

[78] https://docs.python.org/3.9/library/threading.html#threading.Thread
[79] https://docs.python.org/3.9/library/functions.html#int

- **filesize** (*int*[81]) – The size of the file in bytes.
- **filehash** (*str*[82]) – The SHA256 hash of the file contents.
- **package_tag** (*str*[83]) – The package tag extracted from the filename (first "-" separated component).
- **package_version_tag** (*str*[84]) – The package version tag extracted from the filename (second "-" separated component).
- **py_version_tag** (*str*[85]) – The python version tag extracted from the filename (third from last "-" separated component).
- **abi_tag** (*str*[86]) – The python ABI tag extracted from the filename (second from last "-" separated component).
- **platform_tag** (*str*[87]) – The platform tag extracted from the filename (last "-" separated component).
- **requires_python** (*str*[88]) – The `Requires-Python` specification for the file.
- **dependencies** (*set*[89]) – The set of dependencies that are required to use this particular wheel.
- **transferred** (*bool*[90]) – `True` if the file has been transferred from the build slave that generated it to the file server.

as_message()

Convert the `FileState` (page 72) object into a simpler list for serialization and transport.

classmethod from_message(*value*)

Convert the output from `as_message()` (page 73) back into a `BuildState` (page 73).

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 74) attribute is a mapping containing details of each successfully built package file.

Parameters

- **slave_id** (*int*[91]) – The master's identifier for the build slave.
- **package** (*str*[92]) – The name of the package to build.
- **version** (*str*[93]) – The version number of the package to build.
- **abi_tag** (*str*[94]) – The ABI for which the build was attempted (must not be `'none'`).
- **status** (*bool*[95]) – `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*[96]) – The log output of the build.

[80] https://docs.python.org/3.9/library/stdtypes.html#str
[81] https://docs.python.org/3.9/library/functions.html#int
[82] https://docs.python.org/3.9/library/stdtypes.html#str
[83] https://docs.python.org/3.9/library/stdtypes.html#str
[84] https://docs.python.org/3.9/library/stdtypes.html#str
[85] https://docs.python.org/3.9/library/stdtypes.html#str
[86] https://docs.python.org/3.9/library/stdtypes.html#str
[87] https://docs.python.org/3.9/library/stdtypes.html#str
[88] https://docs.python.org/3.9/library/stdtypes.html#str
[89] https://docs.python.org/3.9/library/stdtypes.html#set
[90] https://docs.python.org/3.9/library/functions.html#bool

- **files** (*dict*[97]) – A mapping of filenames to *FileState* (page 72) objects for each artifact produced by the build.
- **build_id** (*int*[98]) – The integer identifier generated for the build by the database (`None` until the build has been inserted into the database).

as_message()

Convert the *BuildState* (page 73), and its nested *FileState* (page 72) objects into simpler lists for serialization and transport.

classmethod from_message(*value*)

Convert the output from *as_message()* (page 74) back into a *BuildState* (page 73).

logged(*build_id*)

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

property files

A mapping of filename to *FileState* (page 72) instances.

property next_file

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

property transfers_done

Returns `True` if all files have been transferred.

class piwheels.states.**SlaveState**(*address*, *build_timeout*, *busy_timeout*, *native_py_version*, *native_abi*, *native_platform*, *label*, *os_name*, *os_version*, *board_revision*, *board_serial*)

Tracks the state of a build slave. The master updates this state with each request and reply sent to and received from the slave, and this class in turn manages the associated *BuildState* (page 73) (accessible from `build`) and *TransferState* (page 74) (accessible from `transfer`). The class also tracks the time a request was last seen from the build slave, and includes a `kill()` method.

Parameters

- **address** (*bytes*[99]) – The slave's ephemeral 0MQ address.

Note: This is *not* the slave's IP address; it's a unique identifier generated on connection to the master's ROUTER socket. It will be different each time the slave re-connects (due to timeout, reboot, etc).

- **timeout** (*int*[100]) – The number of seconds after which any build will be considered to have timed out (and the slave will be assumed crashed).
- **native_py_version** (*str*[101]) – The slave's native Python version.
- **native_abi** (*str*[102]) – The slave's native Python ABI.
- **native_platform** (*str*[103]) – The slave's native platform.
- **label** (*str*[104]) – A label representing the slave.

[91] https://docs.python.org/3.9/library/functions.html#int
[92] https://docs.python.org/3.9/library/stdtypes.html#str
[93] https://docs.python.org/3.9/library/stdtypes.html#str
[94] https://docs.python.org/3.9/library/stdtypes.html#str
[95] https://docs.python.org/3.9/library/functions.html#bool
[96] https://docs.python.org/3.9/library/stdtypes.html#str
[97] https://docs.python.org/3.9/library/stdtypes.html#dict
[98] https://docs.python.org/3.9/library/functions.html#int
[99] https://docs.python.org/3.9/library/stdtypes.html#bytes
[100] https://docs.python.org/3.9/library/functions.html#int
[101] https://docs.python.org/3.9/library/stdtypes.html#str
[102] https://docs.python.org/3.9/library/stdtypes.html#str
[103] https://docs.python.org/3.9/library/stdtypes.html#str
[104] https://docs.python.org/3.9/library/stdtypes.html#str

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 received `chunk()`, and to determine the next chunk to `fetch()`, as well as a property to determine when the transfer is `done`.

Parameters

- **slave_id** (*str*[105]) – The ID number of the slave which built the file.
- **file_state** (`FileState` (page 72)) – The details of the file to be transferred (filename, size, hash, etc.)

class piwheels.states.**DownloadState**(*filename*, *host*, *timestamp*, *arch*, *distro_name*, *distro_version*, *os_name*, *os_version*, *py_name*, *py_version*, *installer_name*, *installer_version*, *setuptools_version*)

Represents the state of the log entry for a download of a package wheel file, including its `filename`, the user's `host` IP, access `timestamp` and information about the operating system and installer.

Parameters

- **filename** (*str*[106]) – The filename of the downloaded wheel file.
- **host** – The hostname or IP address of the user.
- **timestamp** (*datetime.datetime*[107]) – The timestamp at which the file was downloaded.
- **arch** (*str*[108] *or None*) – The architecture of the user's computer system (usually armv6 or armv7).
- **distro_name** (*str*[109] *or None*) – The user's operating system distribution name (e.g. Raspbian).
- **distro_version** (*str*[110] *or None*) – The version of the user's operating system distribution.
- **os_name** (*str*[111] *or None*) – The name of the user's operating system (e.g. Linux).
- **os_version** (*str*[112] *or None*) – The version of the user's operating system (e.g. Linux kernel version).
- **py_name** (*str*[113] *or None*) – The Python implementation used (e.g. CPython).
- **py_version** (*str*[114] *or None*) – The Python version used (e.g. 3.7.3).
- **installer_name** (*str*[115] *or None*) – The name of the tool used to install the file (e.g. pip).
- **installer_version** (*str*[116] *or None*) – The version of the tool (e.g. pip) used to install the file.
- **setuptools_version** (*str*[117] *or None*) – The version of setuptools used.

[105] https://docs.python.org/3.9/library/stdtypes.html#str
[106] https://docs.python.org/3.9/library/stdtypes.html#str
[107] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[108] https://docs.python.org/3.9/library/stdtypes.html#str
[109] https://docs.python.org/3.9/library/stdtypes.html#str
[110] https://docs.python.org/3.9/library/stdtypes.html#str
[111] https://docs.python.org/3.9/library/stdtypes.html#str
[112] https://docs.python.org/3.9/library/stdtypes.html#str
[113] https://docs.python.org/3.9/library/stdtypes.html#str
[114] https://docs.python.org/3.9/library/stdtypes.html#str
[115] https://docs.python.org/3.9/library/stdtypes.html#str
[116] https://docs.python.org/3.9/library/stdtypes.html#str
[117] https://docs.python.org/3.9/library/stdtypes.html#str

class `piwheels.states.`**`SearchState`**(*package*, *host*, *timestamp*, *arch*, *distro_name*, *distro_version*, *os_name*, *os_version*, *py_name*, *py_version*, *installer_name*, *installer_version*, *setuptools_version*)

Represents the state of the log entry for an instance of a package search, including the `package` name, user's `host` IP, access `timestamp` and information about the operating system and installer.

Parameters

- **`package`** (`str`[118]) – The name of the package searched for.
- **`host`** (`str`[119]) – The hostname or IP address of the user.
- **`timestamp`** (`datetime.datetime`[120]) – The timestamp at which the search occurred.
- **`arch`** (`str`[121] *or None*) – The architecture of the user's computer system (usually armv6 or armv7).
- **`distro_name`** (`str`[122] *or None*) – The user's operating system distribution name (e.g. Raspbian).
- **`distro_version`** (`str`[123] *or None*) – The version of the user's operating system distribution.
- **`os_name`** (`str`[124] *or None*) – The name of the user's operating system (e.g. Linux).
- **`os_version`** (`str`[125] *or None*) – The version of the user's operating system (e.g. Linux kernel version).
- **`py_name`** (`str`[126] *or None*) – The Python implementation used (e.g. CPython).
- **`py_version`** (`str`[127] *or None*) – The Python version used (e.g. 3.7.3).
- **`installer_name`** (`str`[128] *or None*) – The name of the tool used (e.g. pip).
- **`installer_version`** (`str`[129] *or None*) – The version of the tool (e.g. pip) used.
- **`setuptools_version`** (`str`[130] *or None*) – The version of setuptools used.

class `piwheels.states.`**`ProjectState`**(*package*, *host*, *timestamp*, *user_agent*)

Represents the state of the log entry for an instance of project page hit, including the `page` name, the user's `host` IP, access `timestamp` and the user's `user_agent`.

Parameters

- **`package`** (`str`[131]) – The name of the package searched for.
- **`host`** (`str`[132]) – The hostname or IP address of the user.
- **`timestamp`** (`datetime.datetime`[133]) – The timestamp at which the page was accessed.
- **`user_agent`** (`str`[134]) – The user agent of the page request.

[118] https://docs.python.org/3.9/library/stdtypes.html#str
[119] https://docs.python.org/3.9/library/stdtypes.html#str
[120] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[121] https://docs.python.org/3.9/library/stdtypes.html#str
[122] https://docs.python.org/3.9/library/stdtypes.html#str
[123] https://docs.python.org/3.9/library/stdtypes.html#str
[124] https://docs.python.org/3.9/library/stdtypes.html#str
[125] https://docs.python.org/3.9/library/stdtypes.html#str
[126] https://docs.python.org/3.9/library/stdtypes.html#str
[127] https://docs.python.org/3.9/library/stdtypes.html#str
[128] https://docs.python.org/3.9/library/stdtypes.html#str
[129] https://docs.python.org/3.9/library/stdtypes.html#str
[130] https://docs.python.org/3.9/library/stdtypes.html#str
[131] https://docs.python.org/3.9/library/stdtypes.html#str
[132] https://docs.python.org/3.9/library/stdtypes.html#str
[133] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[134] https://docs.python.org/3.9/library/stdtypes.html#str

class piwheels.states.**JSONState**(*package*, *host*, *timestamp*, *user_agent*)

Represents the state of the log entry for an instance of project JSON download, including the `page` name, the user's `host` IP, access `timestamp` and the user's `user_agent`.

Parameters

- **package** (*str*[135]) – The name of the package whose JSON file was accessed.
- **host** (*str*[136]) – The hostname or IP address of the user.
- **timestamp** (*datetime.datetime*[137]) – The timestamp at which the page was accessed.
- **user_agent** (*str*[138]) – The user agent of the request.

class piwheels.states.**PageState**(*page*, *host*, *timestamp*, *user_agent*)

Represents the state of the log entry for an instance of web page hit, including the `page` name, the user's `host` IP, access `timestamp` and the user's `user_agent`.

Parameters

- **page** (*str*[139]) – The name of the page accessed.
- **host** (*str*[140]) – The IP address of the user.
- **timestamp** (*datetime.datetime*[141]) – The timestamp at which the page was accessed.
- **user_agent** (*str*[142]) – The user agent of the page request.

class piwheels.states.**SlaveStats**(*timestamp*, *disk_size*, *disk_free*, *mem_size*, *mem_free*, *swap_size*, *swap_free*, *load_average*, *cpu_temp*)

class piwheels.states.**MasterStats**(*timestamp*, *packages_built*, *builds_last_hour*, *builds_time*, *builds_size*, *builds_pending*, *new_last_hour*, *files_count*, *downloads_last_hour*, *downloads_last_month*, *downloads_all*, *disk_size*, *disk_free*, *mem_size*, *mem_free*, *swap_size*, *swap_free*, *load_average*, *cpu_temp*)

piwheels.states.**mkdir_override_symlink**(*pkg_dir*)

Make *pkg_dir*, replacing any existing symlink in its place. See the notes in `TheScribe.write_package_index()` for more information.

13.22 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* (page 57) 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:

[135] https://docs.python.org/3.9/library/stdtypes.html#str
[136] https://docs.python.org/3.9/library/stdtypes.html#str
[137] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[138] https://docs.python.org/3.9/library/stdtypes.html#str
[139] https://docs.python.org/3.9/library/stdtypes.html#str
[140] https://docs.python.org/3.9/library/stdtypes.html#str
[141] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[142] https://docs.python.org/3.9/library/stdtypes.html#str

```
>>> 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 FOURTEEN

LICENSE

[143] https://github.com/bennuttall
[144] dave@waveform.org.uk

PYTHON MODULE INDEX

INDEX