crontabber Documentation

Release 0.7

Peter Bengtsson, Lars K Lohn

Sep 27, 2017

Contents

`crontabber` is a cron job manager. Written in Python. Uses PostgreSQL for storage.

Killer features include:

- Retries jobs on failure automatically
- Dependency-aware, and won't execute child jobs that depend on parents that have failed
- Nagios integration making it easy to monitor health of jobs

You start crontabber with `crontab` and internally it figures out which jobs to run when and in what ideal order.

`crontabber` requires Python 2.6 or Python 2.6 and PostgreSQL 9.2 or greater.

CHAPTER 1

User Guide

Introduction

Quickstart

To get started, install with *pip*:

```
pip install crontabber
```

It installs all the dependencies you need.

Note if you're trying to install Python dependencies that rely on C bindings on OSX Maverick not only do you need to have XCode installed, you might also need to set:

```
export CPPFLAGS=-Qunused-arguments
export CFLAGS=-Qunused-arguments
```

Once it's installed it creates an executable called `crontabber` so you should now be able to run:

```
crontabber --help
```

The first thing you need to do is to create a config file. You do that with:

```
crontabber --admin.dump_conf=crontabber.ini
```

That creates a file called `crontabber.ini` which you can open and get familiar with. The file is big and possibly confusing as there are many things you can change. The most important thing is to note that it shows all default settings left commented out.

Before we can start writing our first app we need to set credentials to connect to Postgres. Open your newly created `crontabber.ini` file and look for the settings: `dbname`, `user` and `password`. Perhaps you want to create a new database first to test against:

```
createdb crontabber
```

Depending on how you have set up Postgres server you might need to supply a username and password. Proceed to edit your `crontabber.ini` and set:

```
dbname=crontabber
user=myusername
password=mypostgrespassword
```

You can see where the default settings are set and you can change those lines. Let's try to see if it works:

```
crontabber --admin.conf=crontabber.ini --list
```

You'll possibly see lots of logging on stdout but you shouldn't see any errors.

Great progress so far!

Creating your first app

The most important setting is the `jobs` setting. Let's create our first job. First change the line `#jobs=''` to this:

```
jobs='''
jobs.myapp.MyFirstApp|5m
'''
```

Now `crontabber` is going to need to do the equivalent of:

```
from jobs.myapp import MyFirstApp
```

So, let's create a very simple sample app:

```
mkdir jobs
touch jobs/__init__.py
emacs jobs/myapp.py
```

So now we're creating our app (`myapp.py`). Let's start with this:

```
import datetime
from crontabber.base import BaseCronApp

class MyFirstApp(BaseCronApp):
    app_name = 'my-first-app'

    def run(self):
        with open(self.app_name + '.log', 'a') as f:
            f.write('Now is %s\n' % datetime.datetime.now())
```

And that's it! Let's try that it can be imported by opening a python interpreter:

```
$ python
>>> from jobs.myapp import MyFirstApp
```

Because you created this job in current directory you're in and when you run `crontabber` it won't know which Python path that is referring to, so you're going to need to add `PYTHONPATH=.` to the command line or you can, for now, just run:

```
export PYTHONPATH=.
```

Finally we're ready to run:

```
crontabber --admin.conf=crontabber.ini --list
```

Since you've never run the job before you should see something like this:

```
=== JOB ========================================================================
Class:           jobs.myapp.MyFirstApp
App name:        my-first-app
Frequency:       5m
*NO PREVIOUS RUN INFO*
```

OK. Brace yourself, we're about to run `crontabber` for the first time:

```
crontabber --admin.conf=crontabber.ini
```

Remember what our little app does. It creates a file called `my-first-app.log`. Open that file and you should see something like:

```
$ cat my-first-app.log
Now is 2014-05-08 14:28:14.593252
```

Try running `crontabber` again, noticing that it's not been 5 minutes since we last run it:

```
crontabber --admin.conf=crontabber.ini
```

Did it write another line to `my-first-app.log`? Try waiting more than 5 minutes and run again. You can run the above mentioned command as many times as you like.

If you're curious how this state is remembered, you can open your database and look at the two tables it created automatically:

```
$ psql crontabber

crontabber=# select * from crontabber;
...
crontabber=# select * from crontabber_log;
...
```

Let's move on to write *More Advanced Apps*.

More Advanced Apps

This documentation carries on the *Quickstart*.

Apps with dependencies

When you wrote your first app (`jobs.myapp.MyFirstApp`) you had to set an `app_name` on the class. That's how you reference other apps when setting up dependency management. This is important to note. The name of the Python file or the name of class does not matter.

Diving in, let's now create two more apps. For simplicity we can continue in the file `myapp.py` you created. Add this:

```
class MySecondApp(BaseCronApp):
    app_name = 'my-second-app'
    depends_on = ('my-first-app',)   # NOTE!

    def run(self):
        with open(self.app_name + '.log', 'a') as f:
            f.write('Second app run at %s\n' % datetime.datetime.now())
```

Exactly where you add this app doesn't really matter. Before or after or even in a different file. All the matters is the `app_name` attribute and the `depends_on`. It event doesn't matter which order you place it in your `crontabber.ini`'s `jobs` setting. You can change your `crontabber.ini` to be like this:

```
jobs='''
jobs.myapp.MySecondApp|1h
jobs.myapp.MyFirstApp|5m
'''
```

`crontabber` reads the `jobs` setting but when there's dependency linking apps, even though it reads the `jobs` setting from top to bottom, it knows that `jobs.myapp.MySecondApp` needs to be run **after** `jobs.myapp.MyFirstApp`.

Go ahead and try it:

```
crontabber --admin.conf=crontabber.ini
```

If you now look at the timestamps in `my-first-app.log` and `my-second-app.log` are in the correct order according to the dependency linkage rather than the order they are written in the `jobs` setting.

Another important thing to appreciate is that if a job fails for some reason, i.e. a python exception is raised, it will stop any of the **dependening jobs from running**. Basically, if job `B` depends on job `A`, job `B` will not run until job `A` ran without a failure. Basically, `crontabber` not only makes sure the order is correct, it also guards from running dependents if their "parent" fails.

About the job frequency

In the above example note the notation used for the `jobs` setting. It's `python.module.and.classname|5m` or `python.module.and.classname|1h`.

The frequency is pretty self explanatory. `5m` means **every 5 minutes** and `1h` means **every hour**. The other thing you could use is, for example, `3d` meaning **every 3 days**.

Running at specific times

Suppose you have a job that is really intensive and causing a lot of stress in your server. Then you might want to run that "at night" (in quotes because it means different things in different parts of the world) or whenever you have the least load in your server.

The way to specify time is to write it in `HH:MM` notation on a 24-hour clock. E.g. `22:30`.

The way you specify the time is to add it to the `jobs` second like this example shows:

```
jobs='''
jobs.myapp.MyBigFatWeddingApp|2d|21:00
```
```
```

But here's a very important thing to remember. The timezone is that of your **PostgreSQL server**. Not the timezone of your server. However, when you install PostgreSQL it will take the timezone from your server's timezone. So if you have a server on the US west coast, the default timezone will be `US/Pacific`.

However, you can, and it's a good idea to do, change the timezone of your PostgreSQL server. So if you have set your PostgreSQL server to `UTC` and the `crontabber` will adjust these times in `UTC` time.

Postgres specific apps

`crontabber` provides several class decorators to make use of postgres easier within a crontabber app. These decorators can imbue your app class with the correct configuration to automatically connect with Postgres and handle transactions automatically. The three decorators provide differing levels of automation so that you can choose how much control you want.

@using_postgres()

This decorator tells crontabber that you want to use postgres by adding to your class two class attributes: `self.database_connection_factory` and `self.database_transaction_executor`. When execution reaches your run method, you may use these two attributes to talk to postgres. If you want a connection to Postgres you can grab one from the `database_connection_factory` and use it as a context manager:

```
# ...
with self.database_connection_factory() as pg_connection:
    cursor = pg_connection.cursor()
```

The connection that you get from the factory is a psycopg2 connection, so you have all the resources of that module available for use with your connection. You don't have to worry about opening or closing the connection, the context manager will do that for you. The connection is open and ready to use when it is handed to you, and is closed when the context ends. You are responsible for transactions within the lifetime of the context.

If you want help with transactions, there is also a the `database_transaction_executor` at your service. Give it a function that accepts a database connection as its first argument, and it will execute the function within a postgres transaction. If your function ends normally (with or without a return value), the transaction will be automatically committed. If an exception is raised and that exception escapes outside of your function, then the transaction will be automatically rolled back.

```
@using_postgres()
class MyPGApp(BaseCronApp):
    def execute_lots_of_sql(connection, sql_in_a_list):
        '''run multiple sql statements in a single transaction'''
        cursor = connection.cursor()
        for an_sql_statement in sql_in_a_list:
           cursor.execute(an_sql_statement)

    def run(self):
        sql = [
            'insert into A (a, b, c) values (2, 3, 4)",
            'update A set a=26 where b > 11',
            'drop table B'
        ]
        self.database_transaction_executor(
            execute_lots_of_sql,
            sql_in_a_list
        )
```

@with_postgres_connection_as_argument()

This decorator is to be used in conjunction with the previous decorator. When using this decorator, your run method must be declared with a database connection as its first argument:

```
@using_postgres()
@with_postgres_connection_as_argument()
class MyCrontabberApp(BaseCronApp):
    app_name = 'postgres-enabled-app'
    def run(self, connection):
        # the connection is live and ready to use
        cursor = connection.cursor()
        # ...
```

With this decorator, the database connection is handed to you. You don't have to get it yourself. You don't have to worry about closing the connection, it will be closed for you when your 'run' function ends. However, you are still responsible for your own transactions: you must explicitly use 'commit' or 'rollback'. If you do not 'commit' your changes, they will be lost when the connection gets closed at the end of your function.

You still have the transaction manager available if you want to use it. Note, however, that it will acquire its own database connection and not use the one that was passed into your run function. Don't deadlock yourself.

@as_single_postgres_transaction()

This decorator gives you the most automation. It considers your entire run function to be a single postgres transaction. You're handed a connection through the parameters to your run function. You use that connection to accomplish database stuff. If your run function exits normally, the 'commit' will happen automatically. If your run function exits with a Exception being raised, the connection will be rolled back automatically.

```
@using_postgres()
@as_single_postgres_transaction()
class MyCrontabberApp(BaseCronApp):
    app_name = 'postgres-enabled-app'

    def run(self, connection):
        # the connection is live and ready to use
        cursor = connection.cursor()
        cusor.execute('insert into A (a, b, c) values (11, 22, 33)')
        if bad_situation_detected():
            raise GetMeOutOfHereError()
```

In this example, connections are as automatic as we can make them. If the exception is raised, the insert will be rolled back. If the exception is not raised and the 'run' function exits normally, the insert will be committed.

@with_subprocess

`crontabber` is all Python but some of the tasks might be something other than Python. For example, you might want to run `rm /var/logs/oldjunk.log` or something more advanced.

What you do then is use the `with_subprocess` helper. When you use this helper on your application class, you can use `self.run_process()` and it will return a tuple of exit code, stdout, stderr. This example shows how to use it:

```
from crontabber.base import BaseCronApp
from crontabber.mixins import with_subprocess
```

```
@with_subprocess
class MyFirstCommandlineApp(BaseCronApp):
    app_name = 'my-first-commandline-app'

    def run(self):
        command = 'rm -f /var/logs/oldjunk.log'
        exit_code, stdout, stderr = self.run_process(command)
        if exit_code != 0:
            self.config.logger.error(
                'Failed to execute %r' % command,
            )
            raise Exception(stderr)
```

Nagios reporting

What is Nagios reporting?

Nagios is a common system administration tool for doing system health checks. It works by a central node continually asking questions about "various parts". These parts can be scripts. The scripts have a simple protocol that they have to adhere to; it's the exit code these scripts exit on.

- 0 - everything is fine
- 1 - warning (don't get out of bed)
- 2 - critical (things are on fire!)

The script also has an opportunity to emit a message. It does this by emitting a single line on `stdout` followed by a newline. The convention is to prefix the message according to the exit code. For example:

```
$ ./is-everything-ok.sh
OK - Everything is fine!
$ echo $?
0

$ ./is-everything-ok.sh
WARNING - This could get very bad!
$ echo $?
1

$ ./is-everything-ok.sh
CRITICAL - Call the fire department!
$ echo $?
2
```

How `crontabber` can be a Nagios script

This is very simple. You simply use the `--nagios` parameter. Like this:

```
crontabber --admin.conf=crontabber.ini  --nagios
```

The rules for which exit code to exit on are fairly simple. However, you need to understand a bit more about *Backfillable Jobs*.

If no application in your configuration has errored in the last run the exit code is simply `0` ("OK").

If any of your applications that is **not** a backfillable job has errored the exit code is `2` ("CRITICAL").

Suppose you have a backfillable job and it has only errored **once**, then the exit code is `1` ("WARNING").

Suppose you get a `1` or a `2` then the message that is printed on `stdout` will look like this for example:

```
CRITICAL - my-first-app (MyFirstApp) | <type 'exceptions.OSError'> | [Errno 13]␣
↪Permission denied: '/etc'
```

If you have multiple apps that have failed, the messages (like the example above) will be concatenated with a `;` character so it's all one long line.

Backfillable Jobs

What is backfilling?

Backfilling is basically a `crontabber` app that receives a date to its `run()` function. For example:

```
import datetime
from crontabber.base import BaseCronApp
from crontabber.mixins import as_backfill_cron_app

@as_backfill_cron_app
class MyBackfillApp(BaseCronApp):
    app_name = 'my-backfill-app'

    def run(self, date):
        with open(self.app_name + '.log', 'a') as f:
            f.write('Date supplied: %s\n' % date)
```

The `date` parameter is a Python `<datetime.datetime>` instance variable with timezone information.

What `crontabber` guarantees is that that method will never be called with the same `date` value twice.

The point of all this is if the app was to fail, it will be retried automatically by `crontabber` and when it does that needs to know exactly what dates have been tried before.

An example explains it

Suppose that you have a stored procedure in a PostgreSQL database. It needs to be called exactly once every day. Internally the stored procedure is programmed to raise an exception if the same day is supplied twice. For

example it might do something like this:

```
CREATE OR REPLACE FUNCTION cleanup(report_date DATE)
    RETURNS boolean
    LANGUAGE plpgsql
AS $$
BEGIN

SELECT 1 FROM reports_clean
WHERE report_date = report_date;
IF FOUND THEN
    RAISE ERROR 'Already run for %.',report_date;
    RETURN FALSE;
END IF;
```

```
INSERT INTO reports_clean (
    name, sex, dob, report_date
)
SELECT
    name, sex, dob, report_date
FROM ( SELECT
            TRIM(both ' ' from full_name)
            gender,
            date_of_birth::DATE
       FROM data_collection
       WHERE
            collection_date = report_date
            AND
            gender = 'male' OR gender = 'female'
);

RETURN TRUE;
END;
$$;
```

The example is not a real-world example but it demonstrates the importance of really making sure the same date isn't passed into the function twice. If it was, you'd have duplicates for a particular date and that would be bad.

When does the magic kick in?

When things go wrong. If for example, you have some network outtage or a bug in your code or something then the triggering will cause an error. That's OK because `crontabber` will catch that and take note of exactly what date it tried to pass in.

Then, the next time `crontabber` runs it will re-attempt to execute the job app with the same date, even if the wall clock says it's the next day. It will also know which other days it has not been able to execute and re-attempt those too.

Suppose you have a daily app that is configured to be backfillable. The app depends on presence of some external third party service which unfortunately goes offline for three days. It's not a problem, `crontabber` will try and try till it works and will accordinly pass in the correct dates.

A caveat about backfillable jobs

Because the integrity of which apps have been passed with which dates is important, it means you can't use `crontabber` to run an individual job as a "one off". That means that if you try:

```
crontabber --admin.conf=crontabber.ini --job=my-backfill-app
```

It will deliberately ignore that since there's a risk it then "disrupts" its predictable rythem. Otherwise it could potentially be calling the same app with the same date twice.

Running from bash

Locking

Note: At the time of writing, `crontabber` **does not handle locking**.

This might change in the future.

Generally, locking is a standard bash task that is best described elsewhere. However, this chapter should hopefully get you going in the right direction.

One example implementation of a lockfile is this:

```
#!/bin/bash
lockdir=/tmp/crontabber.lock
if mkdir "$lockdir"
then
    echo >&2 "successfully acquired lock"
    PYTHONPATH=. crontabber --admin.conf=crontabber.ini

    # Remove lockdir when the script finishes, or when it receives a signal
    trap 'rm -rf "$lockdir"' 0    # remove directory when script finishes

    # Optionally create temporary files in this directory, because
    # they will be removed automatically:
    tmpfile=$lockdir/filelist

else
    echo >&2 "cannot acquire lock, giving up on $lockdir"
    exit 0
fi
```

This means that if you have a job that sometimes takes longer than how frequently your `crontab` runs, you won't run the risk of starting the same job more than once.

crontab

This is the heart of it all. Installing and setting up `crontabber` doesn't run anything until you actually start running it yourself and the best way to do that is with `crontab`.

Before you set up your `crontab` it's recommended that you wrap this in a shell script that takes care of paths and options and stuff. That means you can keep your `crontab` clean and simple. Something like this should good enough:

```
*/5 * * * * myuser /path/to/crontabber_wrapper.sh
```

And then you can put the actual execution in that one script. For example, suppose you need a Python `virtualenv`. Like this for example:

```
#!/bin/bash
source /home/users/django/venv/bin/active
HOMEDIR=/home/users/django

PYTHONPATH="$HOMEDIR/jobs" crontabber --admin.conf="$HOMEDIR/crontabber.ini"
```

There are many more things you can do and set up. The point is that you basically do what you were able to do on the command line and freeze that into one script that can be executed from anywhere.

You will probably also want to combine this with the section on Locking above.

Parallel crontabbers

Suppose you have some jobs that take a reeeeeaaallly long time. Equally, you might have some jobs that are quick and needs to run often too. Because `crontabber` is single threaded running your jobs will block other jobs. This is a good thing because it asserts that dependent jobs don't start until their "parents" have finished successfully.

To prevent completely independent jobs from waiting for each other, you can run multiple parallel instances of `crontabber`. This means that you will need to have two lines (or more) in `crontab`. Here's an example:

```
*/5 * * * * myuser /path/to/crontabber_wrapper.sh A
*/5 * * * * myuser /path/to/crontabber_wrapper.sh B
```

And in your wrapper script you take that first parameter like this for example:

```
PYTHONPATH="$HOMEDIR/jobs" crontabber \
  --admin.conf="$HOMEDIR/crontabber.$1.ini"
```

That means you need two config files:

- `crontabber.A.ini`
- `crontabber.B.ini`

You might think that means you have to duplicate things across two different files. Thankfully that's not the case. See *Advanced Configuration*.

Advanced Configuration

configman

Work in progress...

Advanced settings

All configuration in `crontabber` is handled by the fact that it's built on top of configman. `configman` is agnostic to configuration file format (e.g. `.ini` or `.json`) and that means you can reference much more than just strings and integers. For example, you can reference Python classes by their name and they get imported automatically when need be.

We strongly recommend that when you write a `crontabber` app that you set a sensibile default and only use a configuration file when you need to override it. Sometimes you can't put in a sensible default as the value can't be written in code. Like a password for example.

Setting up settings

The trick to adding configuration is to set a class attribute on your `crontabber` app called `required_config`. Let's dive straight into an example:

```
import datetime
from crontabber.base import BaseCronApp
from configman import Namespace

class MyFirstConfigApp(BaseCronApp):
```

```
    app_name = 'my-first-config-app'

    required_config = Namespace()
    required_config.add_option(
        'date_format',
        default='%m/%d %Y - %H:%M',
        doc="Format for how the date is reported in the log file."
    )

    def run(self):
        with open(self.app_name + '.log', 'a') as f:
            dt = datetime.datetime.now()
            f.write('Now is %s\n' % dt.strftime(self.config.date_format))
```

The magic to notice is how you import `Namespace` from `configman`, create a class attribute called `required_config` and then inside the `run()` method you can reference to by `self.config.date_format`.

Overriding settings

So, there are now two ways of overriding this other than letting the default value play. You can either do it in your existing configuration file (`crontabber.ini` if you've played along from the *Introduction*) or you can do it right on the command line as local environment variables.

If you intend to use non-trivial notation for environment variables in bash you have to prefix the command with a program called `env` that is built in on almost all version of bash. So, here's an example of doing just that:

```
env crontabber.class-MyFirstConfigApp.date_format="%A" crontabber --admin.
↪conf=crontabber.ini
```

Run that and you'll notice it picked up the override setting.

Another way of specifying this is in your `crontabber.ini` file. Note! Setting this requires that you do it under the `[crontabber]` section heading. It looks like this:

```
...

[crontabber]

    ...

    [[class-MyFirstConfigApp]]

        # Format for how the date is reported in the log file.
        date_format=%W %y %h:%M
```

If you ever forget this notation, after you have added some setting options you can run:

```
crontabber --admin.conf=crontabber.ini --admin.print_conf=ini
```

and look at the commented out examples.

Now, run it again and it should pick this up. Now you don't need to specify anything extra on the command line, so you can use:

```
crontabber --admin.conf=crontabber.ini
```

Let's now make a setting that is something the app needs to import (as a Python module, class or function) on the fly. Let's say we want override what function our simple app uses to generate the datetime. So we add another config called `date_function` and tell the config that this is something it needs to import:

```
import datetime
from crontabber.base import BaseCronApp
from configman import Namespace

class MyFirstConfigApp(BaseCronApp):
    app_name = 'my-first-config-app'

    required_config = Namespace()
    required_config.add_option(
        'date_format',
        default='%m/%d %Y - %H:%M',
        doc="Format for how the date is reported in the log file."
    )
    required_config.add_option(
        'date_function',
        default=datetime.datetime.now,
        doc="Function that generates datetime instance"
    )

    def run(self):
        with open(self.app_name + '.log', 'a') as f:
            dt = self.config.date_function()
            f.write('Now is %s\n' % dt.strftime(self.config.date_format))
```

Configman automatically notices that the default isn't a string but something pythonic that it can use. But if you want to change that, in a `crontabber.ini` file you have to reference it as a string. How do you do that? This trick isn't for the faint of heart but it's very powerful one. What you do is you write a `from_string_converter` function.

Mind you, this is a rather odd and complicated example but it shows the power of being able to change anything from a config file:

```
import datetime
from crontabber.base import BaseCronApp
from configman import Namespace

def function_converter(function_reference):
    module, callable, function = function_reference.rsplit('.', 2)
    module = __import__(module, globals())
    callable = getattr(module, callable)
    return getattr(callable, function)

class MyFirstConfigApp(BaseCronApp):
    app_name = 'my-first-config-app'

    required_config = Namespace()
    required_config.add_option(
        'date_format',
        default='%m/%d %Y - %H:%M',
        doc="Format for how the date is reported in the log file."
    )
    required_config.add_option(
        'date_function',
        default=datetime.datetime.now,
        doc="Function that generates datetime instance",
        from_string_converter=function_converter
```

```
    )

    def run(self):
        with open(self.app_name + '.log', 'a') as f:
            dt = self.config.date_function()
            f.write('Now is %s\n' % dt.strftime(self.config.date_format))
```

Now, let's try this out on the command line:

```
env crontabber.class-MyFirstConfigApp.date_function="datetime.datetime.utcnow"\
crontabber --admin.conf=crontabber.ini
```

The documentation on configman has more examples of using the `from_string_converter`.

Raven

raven is a Python program for sending in Python exceptions as a message to a Sentry server. Both as free and Open Source but Sentry exists as a paid service if you don't want to self-host.

What *raven* does is that it makes it possible to package up a Python exception (type, value, traceback) and send it in to a server. Once the server receives it, it makes a hash of the error and adds an entry to its database. If the same error is sent again, instead of logging another entry to its database it increments the previous one.

Configure your API key

To configure your *crontabber* to send all exceptions into a Sentry server you need an API key. When you have that you add that to your `crontabber.ini` file. So it looks something like this:

```
[sentry]

# DSN for Sentry via raven
dsn=https://d3683ad...27f9fbd:0ce...4aeb810311dc@errormill.mozilla.org/14
```

Note, this is not mandatory. You can always reach the full error details in the logs of the database. Either by interrogating the database table yourself or by using the command like this:

```
crontabber --admin.conf=crontabber.ini --list-jobs
```

Different protocols

It's important to note that the protocol used by Sentry has changed in recent years. That means that you need to be careful with what version of `raven` you install. If you have an older version of Sentry you can not install the latest version of `raven` because the messages it transmits won't be understood.

Command line options

This chapter aims to digest some of the command line options available in `crontabber`.

One of the important command line options is that on `--nagios` which we explored in its *own chapter*.

--configtest

If you change the config file it's always a highly recommended and good idea to first run:

```
crontabber --admin.conf=crontabber.ini --configtest
```

first. It checks that you haven't made any trivial errors in the config file that will not get caught until it's too late.

It's important to remember that this basically only checks the `jobs` setting. But if you have some other typo or corruption anywhere in your files it might catch it simply because it's unable to load up the `jobs` setting at all.

When it validates the `jobs` setting it checks:

1. That the job can be found and imported
2. That the frequency is a valid frequency (e.g. `2d` is valid `2r` is not)
3. That the time (clock time the job should fire) is a valid time.
4. If a job has a less than daily frequency that a time is not set.

Running this should exit the application with an exit code `0` if all is well. If not the exit code will be a count of how many apps are misconfigured.

--reset-job=

This keyword parameter option is pretty self explanatory. It resets the job and basically pretends the job has never run. Just like the state database didn't know about it before before it was ever run the first time.

It's important to note that this does not clean out the mentioned job from the logs.

You can either specify the `app_name` or the notation that specifies the location of the app class. For example:

```
crontabber --admin.conf=crontabber.ini --reset-job=my-first-app
```

Or:

```
crontabber --admin.conf=crontabber.ini --reset-job=jobs.myapp.MyFirstApp
```

If you reset a job that has already been reset nothing happens.

--job= and --force

Sometimes you just know a particular job needs to be run here and now. You can obviously do this outside of `crontabber` but suppose the app you have written has a fair amount of business logic in it and not just a wrapper around something written elsewhere.

The notation is pretty straight forward as you can guess:

```
crontabber --admin.conf=crontabber.ini --job=my-first-app
```

or:

```
crontabber --admin.conf=crontabber.ini --job=my-first-app
```

However, this will still check if the job is ready to run next. Suppose a job is not due to run for another hour, then typing in this command the job **will not be run** straight away. There's also another chance that the job you're trying to run has a blocking dependency (ie. a job it depends on failed last time).

If you really want to run it now and can't wait, add `--force` like this:

```
crontabber --admin.conf=crontabber.ini --job=my-first-app --force
```

There is an important limitation of this command line option. It **does not work with backfill apps**. Because backfill apps are very sensitive about exactly when they run they simply ignore both the `--job=` and even the `--force` parameter.

`--audit-ghosts`

This command finds jobs that are in the state database but no longer anywhere in your configuration. For example, suppose you decide to change your list of configured jobs after they have been run. At that point there will be so called "ghosts". I.e. jobs that are in the database but not in the config.

But be careful though! You might have 1 database but two different configuration files. For example, you might have one configuration called `virtualenv-X-crontabber.ini` and one called `virtualenv-Y-crontabber.ini`. So the term "ghosts" is only applicable to one configuration file basically.

If you know for certain that a job is no longer needed and stuck as a ghost in the state database, then use `--reset-job=` (see above) to clear it out.

`--version`

Spits out the version of `crontabber` on standard out.

Contributing

GitHub

Please use the GitHub project page to report bugs, issues and feature requests.

And don't be afraid to express yourself. If something isn't clear or you don't understand something perhaps the documentation or the configuration or some parameter is not easy enough to understand.

But please try to avoid using GitHub issues to ask questions specific to your installation. If you need help, the best place to go is to use IRC and the `#breakpad` room on `irc.mozilla.org`.

When reporting bugs, please try to include as much information as you possibly can. Don't forget to include exactly what versions you have of `crontabber`, `configman`, Python and PostgreSQL. But before you share your config files, please please please make sure they don't contain any passwords or any other secrets that could put your system at risk.

Coding style

We try to stick to a strict PEP8 guideline with lines no longer than 79 characters. But functionality is more important than form. If you have some code to contribute don't feel intimidated that your code isn't perfect. We can always change it later.

But please try to continue the existing patterns. If the code around uses `'` instead of `"` then continue to use `'`. Consistency makes the code easier to read and debug.

Running tests

nosetests

All the dependencies you need to be able to run tests are encapsulated in the test-requirements.txt file. First install that into your virtualenv:

```
pip install -r test-requirementst.txt
```

You also need to create a dedicated PostgreSQL database that you can run the tests against. And you also you need to be able to connect to this database. So you need the username and password.

Next, in the root directory of the project create a file called `test-crontabber.ini` and it should look something like this:

```
[resource]
[[postgresql]]
user=myusername
password=mypassword
dbname=test_crontabber
```

To start all the tests run:

```
PYTHONPATH=. nosetests
```

If you want to run a specific test in a specific file in a specific class you can define it per the `nosetests` standard like this for example:

```
PYTHONPATH=. nosetests tests crontabber/tests/test_crontabber.py:TestCrontabber.test_
↪basic_run_job
```

If you want the tests to stop as soon as the first test fails add `-x` to that same command above.

Also, if you want `nosetests` to *not* capture `stdout` add `-s` to that same command as above.

Example project

The `exampleapp` project helps you set up a playground to play around with and test `crontabber` to gain a better understanding of how it works.

The best place to start with is to read the exampleapp/README.md file and go through its steps. Once you get the basics to work you can start experimenting with adding your job classes.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search