Chronograph Documentation

Release 0.3.2.dev

Weston Nielson

Apr 18, 2017

Contents

Contents:

CHAPTER 1

Installation

Installing `chronograph` is pretty simple. First add it into `INSTALLED_APPS` in your `settings.py` file.

Note: As of version 0.3.1, `chronograph` is only compatible with Django versions >= 1.1.

After this run `syncdb`. The only thing left to do is set up a periodic call to run the jobs.

If you're using *cron*, the following example can be added to your *crontab*:

```
* * * * * /path/to/your/project/manage.py cron
```

You're done! Every minute `cron` will check to see if you have any pending jobs and if you do they'll be run. No more mucking about with your `crontab`.

If you have a more complicated setup where `manage.py` might not work by default see the section below on installing `chronograph` in a virtual environment.

New in version 0.2.0: Added `chronograph` script

If your project does not reside within your global python path then the above crontab snippet won't work. Instead, `django-chronograph` now comes with a script called `chronograph` (located at `bin/chronograph`) that provides an easy way to run all jobs that are due, like so:

```
/path/to/bin/chronograph -p /path/to/your/project
```

For use in a virtual environment, see below.

Within a Virtual Environment

New in version 0.2.0: Added `chronograph` script

When running `setup.py` a script named `chronograph` should get installed into your `bin` directory. This script is meant to make it really easy to call your `django-chronograph` jobs. For extended usage information, see the output from the script. Here is an example of how to use this script from within the system's crontab:

```
* * * * * /path/to/bin/chronograph -e /path/to/ve/bin/activate_this.py -p /path/to/
↪your/project
```

Note: This script is new and should be considered experimental. Please report any bugs to the issue tracker.

Deprecated since version 0.2.0: `chronograph.sh` has been removed. Use `chronograph` instead (see above).

If you're using a virtual environment, setting up `chronograph `` involves a bit more work, but not by much. Included is a script called ``chronograph.sh`. Copy this file to your project directory.

You should open up this script and modify the path to your virtual environment's `activate` script:

```
$PROJECT_PATH"/../../../ve/bin/activate"
```

Make sure that this file is executable and then update your `crontab` to execute the script. Running *crontab -e*:

```
* * * * * /path/to/your/project/chronograph.sh /path/to/your/project
```

Make sure that you pass `/path/to/your/project` to the script as the first argument. This will ensure that `cron` will not have any problems finding your project directory.

CHAPTER 2

Using Chronograph

If you've completed the above steps, you're all done. Now you can add some jobs to the system. Remember, `chronograph` is designed to run any installed `django-admin` management command and it accommodates command-line arguments as well.

Cleaning Out Old Job Logs

If you'd like an easy way to delete old job logs, there is a management command that will do it for you: `cron_clean`. You can use it like so:

```
python manage.py cron_clean [weeks|days|hours|minutes] [integer]
```

So, if you want to remove all jobs that are older than a week, you can do the following:

```
python manage.py cron_clean weeks 1
```

Since this is just a simple management command, you can also easily add it to `chronograph`, via the admin, so that it will clear out old logs automatically.

Other Useful Bits

There is anther included management command, `cronserver` which can be used to test the periodic running of jobs. It'll print out information to the screen about which job are due and also run them. Here is how you would use it:

```
python manage.py cronserver
```

This will start up a process that will check for and run any jobs that are due every 60 seconds. The interval can be changed by simply passing the number of seconds in between runs. For example, to make the process check for due jobs every 2 minutes, you would run:

```
python manage.py cronserver 120
```

CHAPTER 3

Architecture

The trickiest thing to get right in `Chronograph` is the ability to properly manage the state of a `Job`, i.e. reliably determining whether or not a job is or isn't running, if it has been killed or terminated prematurely. In the first version of `Chronograph` this issue was "solved" by keeping track of the PID of each running job and using the `ps` command to have the operating system tell us if the job was still running. However, this route was less than ideal, for a few reasons, but most importantly because isn't wasn't cross-platform. Additionally, using a series of `subprocess.Popen` calls was leading to path-related issues for some users, even on "supported" platforms.

Newer version of `Chronograph` have attempted to solve this problem in the following way:

1. Get a list of `Job`s that are "due"
2. For each `Job`, launch a `multiprocessing.Process` instance, which internally calls `django.core.management.call_command`
3. When the `Job` is run, we spawn a `threading.Thread` instance whose sole purpose is to keep track of a lock file. This thread exists only while the Job is running and updates the file every second. We store the path to this temporary file (an instance of `tempfile.NamedTemporaryFile`) on the `Job` model (which is then stored in the database). When we want to check if a `Job` is running we do the following:
   (a) If `is_running` equals `True`, and `lock_file` point to a file, then:
      i. If the lock file actually exists and has been updated more recently than `CHRONOGRAPH_LOCK_TIMEOUT` seconds, then we can assume that the `Job` is still running
   (b) Else we assume the `Job` is not running and update the database accordingly

This new method should would much more reliably across all platforms that support the threading and multiprocess libraries.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search