Job-Runner Documentation

Release 1.4.3

Spil Games

March 19, 2013

CONTENTS

CHAPTER

ONE

SETUP JOB-RUNNER

Note: By default the settings in `job_runner.settings.env.development` are used. These settings should work out-of-the box for local development (using Sqlite as a database back-end). Use the `--settings` argument of `manage.py` to use different settings.

1. Make sure you have all requirements installed (the exact package names can vary per distribution, these are for Ubuntu).:
   - `python-dev`
   - `virtualenvwrapper`
   - `build-essential`
   - `libmysqlclient-dev`
2. Clone the project:

   ```
   $ git clone git@github.com:spilgames/job-runner.git
   ```

3. Create a Virtualenv (http://virtualenvwrapper.readthedocs.org/en/latest/), to make sure all requirements are installed in an isolated environment. This is not required, but it will keep your system clean :)

   ```
   $ mkvirtualenv job-runner
   ```

4. Install the Job-Runner (which will fetch all Python requirements as well):

   ```
   # OPTION 1: Install package in development mode.  Any changes you make
   # will be reflected immediately without having to do an install again.
   $ python setup.py develop
   $ pip install -r test-requirements.txt

   # OPTION 2: Just install (you will have to do a new install if you have
   # changed the code).
   $ python setup.py install
   ```

5. Initialize the database and run the migations:

   ```
   $ manage.py syncdb
   $ manage.py migrate
   $ manage.py collectstatic
   ```

6. Run `manage.py runserver`. This will start a development server with the default development settings.
7. Run `manage.py broadcast_queue`. This will start the queue broadcaster.

CHAPTER

TWO

GETTING STARTED

This page describes how to setup a working **Job-Runner** environment, including the **Job-Runner WebSocket Server** and the **Job-Runner Worker**. After a successful installation, you'll have four processes running:

1. `./manage.py runserver`, webserver for admin, REST api and dashboard (part of **Job-Runner**)
2. `./manage.py broadcast_queue`, queue broadcaster (part of **Job-Runner**)
3. `./scripts/job_runner_ws_server`, WebSocket server (part of **Job-Runner WebSocket Server**)
4. `./scripts/job_runner_worker --config-path job_runner_worker.ini`, the worker executing our jobs

2.1 Install all components

1. Create three virtualenv environments called `job-runner`, `job-runner-ws-server` and `job-runner-worker`. If you are on Ubuntu, you might want to install the `virtualenvwrapper` package first:

   ```
   # if you need to install virtualenvwrapper first
   $ sudo apt-get install virtualenvwrapper

   $ mkvirtualenv job-runner && deactivate
   $ mkvirtualenv job-runner-ws-server && deactivate
   $ mkvirtualenv job-runner-worker && deactivate
   ```

   See Also:

   http://virtualenvwrapper.readthedocs.org/en/latest/ if you are not familiar with virtualenvwrapper and virtualenv

2. Install the **Job-Runner** in the `job-runner` virtualenv. To activate this virtualenv, execute `workon job-runner`. See *Setup Job-Runner* for installation details.
3. Install the **Job-Runner WebSocket Server** in the `job-runner-ws-server` virtualenv. See the documentation in the `job-runner-ws-server` repository for installation instructions.
4. Install the **Job-Runner Worker** in the `job-runner-worker` virtualenv. See the documentation in the `job-runner-worker` repository for installation instructions.

2.2 Get all components up and running

2.2.1 Job-Runner

1. Open two console tabs and execute in the first one:

   ```
   $ workon job-runner
   $ manage.py runserver
   ```

   Execute in the second tab:

   ```
   $ workon job-runner
   $ manage.py broadcast_queue
   ```

2. Open a browser and point it to http://localhost:8000/admin/. This will open the admin interface of the **Job-Runner**. While executing `manage.py syncdb` you were asked to create superuser credentials, use these to login. If you did not create any, open a new console and execute:

   ```
   $ workon job-runner
   $ manage.py createsuperuser
   ```

3. In the admin interface, assign yourself to a group:
   (a) Go to *Auth - Users*
   (b) Click your username
   (c) Under *Permissions - Groups* click the **+** icon to add yourself to a new group (call it *Test Group* for now)
   (d) Save the user
4. Create a project:
   (a) Go to *Job-Runner - Projects*
   (b) Click the **Add project** button
      - Title: Test Project
      - Groups: select *Test Group*

   (c) Click the save button
5. Create a worker for this project:
   (a) Go to *Job Runner - Workers*
   (b) Click on the **Add Worker** button
      - Title: Test Worker
      - Api key: testworker
      - Secret: verysecret
      - Project: Select *Test Project*

   (c) Click the save button
6. You now have created a group, project and worker :) Leave both processes you started in the first step running!

2.2.2 Job-Runner WebSocket Server

1. Open a new console tab and execute:

```
$ workon job-runner-ws-server
$ job_runner_ws_server
```

2. That's it! Leave this process running :)

2.2.3 Job-Runner Worker

1. Open a new console tab and execute:

```
$ workon job-runner-worker
```

2. Create a file named `job-runner-worker.ini` with the following content:

```
[job_runner_worker]
api_base_url=http://localhost:8000/
api_key=testworker
secret=verysecret
concurrent_jobs=4
log_level=debug
script_temp_path=/tmp
ws_server_hostname=localhost
ws_server_port=5555
broadcaster_server_hostname=localhost
broadcaster_server_port=5556
```

   Please refer to the documentation in the `job-runner-worker` repository for the meaning of these variables.

3. Now start the worker by executing:

```
$ job_runner_worker --config-path job-runner-worker.ini
```

Congratulations! You now have all components up and running. If you point your browser to http://localhost:8000/, you will see an empty dashboard, with top-right a label **Dashboard is live**, meaning that the dashboard is connected to the WebSocket server. If this is red with a warning, please make sure the `job_runner_ws_server` process is still running!

2.3 Your first job

In this part, you'll setup and schedule your first job! This will be a simple Python script, printing "Hello world!" and then sleeping between 3 - 15 sec. This script will be re-scheduled every 1 minute after the schedule dts of the previous run.

1. Point your browser to http://localhost:8000/admin/
2. First create a Python template which will form the base for all future Python jobs:
   (a) Go to *Job-Runner - Job templates*
   (b) Click the **Add job template** button
   (c) Enter the following:
      - Title: Python

- Body:

```
#!/usr/bin/env python

{{ content|safe }}
```

- Worker: Select *Test Worker*
- Auth groups: Select *Test Group*

(d) Click the save button

3. Now create the actual job:
   (a) Go to *Job-Runner - Jobs*
   (b) Click the **Add job** button
   (c) Enter the following:
   - Title: Hello world!
   - Job template: Python
   - Script content:

```
import random
import time

print "Hello world!"
time.sleep(random.randint(3, 15))
```

   - Reschedule interval: 1
   - Reschedule interval type: select *Every x minutes*
   - Reschedule type: select *Increment schedule dts by interval*

   (d) Under *Runs*, select the current date and type by clicking on the date-picker and time-picker icons.
   (e) Save the job.
4. Now go to http://localhost:8000/. If all components are set-up correctly, you should see the job you just created moving from scheduled > in queue > started > completed!

CHAPTER

THREE

PROJECT SETTINGS

Apart from the settings available in Django (a complete list is available at https://docs.djangoproject.com/en/1.4/ref/settings/), the following list of settings is available:

`job_runner.settings.base.`**`HOSTNAME = ''`**
The hostname of the server.

This value is used for generating URL's in the notification e-mails.

`job_runner.settings.base.`**`JOB_RUNNER_ADMIN_EMAILS = []`**
A list of e-mail addresses of the Job-Runner admin(s).

This list will currently be used when a job failed to reschedule.

`job_runner.settings.base.`**`JOB_RUNNER_BROADCASTER_PORT = 5556`**
The port to which the queue broadcaster is binding to.

Unless there is a specific need, you can keep the default.

`job_runner.settings.base.`**`JOB_RUNNER_WORKER_PING_INTERVAL = 300`**
The interval in seconds for sending ping-requests to the workers.

`job_runner.settings.base.`**`JOB_RUNNER_WORKER_PING_MARGIN = 15`**
The time to add to the interval before considering a worker is not responding.

`job_runner.settings.base.`**`JOB_RUNNER_WS_SERVER = 'ws://localhost:5000/'`**
The URL to the Job-Runner WebSocket server.

This should be in the following format:

```
ws://hostname:port/
```

CHAPTER

FOUR

PERMISSION MANAGEMENT

Likely, after setting up a few jobs, you would like to give people within your team access to see the current job overview etc... For this, there are three levels of permissions:

- View status of jobs including log-output
- Permission to schedule a job *now* or to suspend a job
- Access to admin interface to add / edit / remove jobs

4.1 View status of jobs including log-output

To make jobs visible to a user, make sure the user is within at least one group that is linked to the project the job belongs to.

4.2 Permission to schedule a job *now* or to suspend a job

To grant the user permission to schedule a job *now* or to suspend a job, make sure the user is within at least one group that is linked to the job-template the job belongs to.

4.3 Access to admin interface to add / edit / remove jobs

When the above is already true, you can grant a user admin permission by ticking the *Staff status* box in the admin interface for this user. Make sure the user (or one of the groups the user belogs to) has at least the following permissions:

- All *admin* | *log entry* | ...
- All *job_runner* | *job* | ...
- All *job_runner* | *reschedule exclude* | ...
- All *job_runner* | *run* | ...
- All *sessions* | ...

See Also:

Admin interface for more technical details

CHAPTER

FIVE

OVERVIEW

Job-Runner is a crontab like tool, with a nice web-frontend for administration and (live) monitoring the current status. It is possible to schedule recurring jobs, chaining jobs and the option to run or kill jobs ad-hoc (from the dashboard). As well it provides permission management (eg: so that some people are able to only see jobs on the dashboard, where other people are also able to start them ad-hoc or kill them).

The whole project consists of three separate components (and repositories):

- **Job-Runner**: provides the REST interface, admin interface and (live) dashboard. As well this component provides a long-running process (`manage.py broadcast_queue`) to broadcast messages (over ZeroMQ) to the workers. See: https://github.com/spilgames/job-runner
- **Job-Runner Worker**: the process that is responsible for executing the job. It subscribes to (ZeroMQ) messages coming from `broadcast_queue`, send data back over the REST interface and publishes events to the *Job-Runner WebSocket Server*. You can run as many workers as you like, as long as every worker has it's own API key (eg: when you want to run jobs on multiple servers or under different usernames on the same server). API keys can be created in the *Job-Runner* admin interface. See: https://github.com/spilgames/job-runner-worker
- **Job-Runner WebSocket Server**: will subscribe to *Job-Runner Worker* events and re-broadcast them to WebSocket connections coming from the *Job-Runner* dashboard. This makes it possible to add realtime monitoring to the dashboard. See: https://github.com/spilgames/job-runner-ws-server

CHAPTER

SIX

LINKS

- documentation
- job-runner source
- job-runner-worker source
- job-runner-ws-server source

CHAPTER

SEVEN

INTERNALS

7.1 Applications

7.1.1 Job-Runner

Admin interface

The admin interface behaves like any normal Django admin interface. There is only one important modification which applies to the administration of jobs.

Normally all records would be visible when an user has access to a certain part of the admin. The job administration is customized (by including the `PermissionAdminMixin`) so that it only shows the jobs the user has access to. The same applies to the job-template and parent dropdown boxes when editing a job.

The user has access to a job when one of the groups he is assigned to, is in the job-template *auth groups* of the job.

> **Warning:** This only applies to the admin of jobs! Super-user status will overrule this logic!

See Also:

Permission management

PermissionAdminMixin

class job_runner.apps.job_runner.admin.**PermissionAdminMixin**
Mixin class to limit the number of visible items.

fk_groups_path = {}
A `dict` containing the FK field name and as a value the corresponding model and path to the groups.

Example:

```
{
    'job_template': {
        'path': 'auth_groups',
        'model': JobTemplate,
    }
}
```

groups_path = None
A `str` containing the path to the groups.

Example:

```
'job_template__auth_groups'
```

queryset (*request*)
Get results based on groups the user is in.

If the user is a super-user, the user has access to everything. Else, the results are limited based on matching groups as set in the model.

REST interface

The Job runner has a REST interface so that the workers can fetch and modify data. There are two types of authentication:

- Django session (for status dashboard)
- HMAC authentication (for worker daemons)

Django session

When the request contains a logged in user (the user has a session and is logged in) it will automatically get access, limited to `GET` requests.

HMAC authentication

When request contains a valid `Authentication` header, containing a public-key and api key, it will get access to the full set of available methods.

The used HMAC is HMAC-SHA1, with a message in the following format:

```
Uppercased request method + full request path (path + query string, if
applicable) + path + query string, if applicable.
```

The format of the header is:

```
Authentication: ApiKey api_key:hmac_sha1
```

Available end-points

Note: You can append `schema/` to the end of the URL to get information about the schema!

Groups

GET /api/v1/group/ Returns a list of available (and thus assigned) groups.

GET /api/v1/group/{GROUP_ID}/ Returns the details of a specific job-id.

Projects

GET /api/v1/project/ Returns a list of available projects.

GET /api/v1/project/{PROJECT_ID}/ Returns the details of a specific project-id.

Workers

GET /api/v1/worker/ Returns a list of available workers.

GET /api/v1/worker/{WORKER_ID}/ Returns the details of a specific worker-id.

Job-templates

GET /api/v1/job_template/ Returns a list of available job-templates.

GET /api/v1/job_template/{JOB_TEMPLATE_ID}/ Returns the details of a specific job-template id.

Jobs

GET /api/v1/job/ Returns a list of available jobs.

GET /api/v1/job/{JOB_ID}/ Returns the details of a specific job-id.

Runs

GET /api/v1/run/ Returs a list of runs. You can filter the state by adding `state` as a keyword argument. Possible values are:

- `scheduled` (scheduled by not picked up yet by a worker)
- `in_queue` (picked up by a worker, but not yet started)
- `started` (started, but not completed yet)
- `completed` (completed, either with or without error)
- `completed_successful` (completed without error)
- `completed_with_errors` (completed with error)

GET /api/v1/run/{RUN_ID}/ Returns the details of a specific job run.

PATCH /api/v1/run/{RUN_ID}/ When the `return_dts` is patched, the job will be automatically rescheduled (if needed).

PYTHON MODULE INDEX

j