

---

# **Job-Runner Documentation**

*Release 3.4.0*

**Spil Games**

June 25, 2013



# CONTENTS



# 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.

If you want to override the default settings, you could create a module (eg: `job_runner_config`) with your setting overrides:

```
from job_runner.settings.base import *  
  
HOSTNAME = 'my.host.name'
```

See *Project settings* and <https://docs.djangoproject.com/en/1.4/ref/settings/> for available 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`

2. 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
```

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

```
$ pip install job-runner
```

Alternatively, you could clone the `job-runner` repository and install the 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
```

4. Initialize the database and run the migrations:

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

5. Run `manage.py runserver`. This will start a development server with the default development settings. The development server will serve the admin interface, the dashboard and the RESTful API.

6. Run `manage.py broadcast_queue`. This will start the queue broadcaster. The queue broadcaster will broadcast the runs that are scheduled for execution to the subscribed workers.
7. Run `manage.py health_check`. This will monitor the health of the workers and alert (don't forget to setup e-mail addresses) when there are problems.

# 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

For an explanation of the role of each component, see *Overview*.

## 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
```

---

**Note:** The first process is responsible for serving all HTTP requests (dashboard, admin and RESTful API). The second process is responsible for broadcasting scheduled runs to the subscribed workers.

---

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

---

**Note:** You need to be assigned to at least one group, so in the following steps, you can give yourself access to a project.

---

4. Create a worker:

- (a) Go to *Job Runner - Workers*
- (b) Click on the **Add Worker** button
  - Title: Test Worker
  - API key: testworker
  - Secret: verysecret
- (c) Click the save button

---

**Note:** A worker is responsible for the actual execution of the job and will communicate back with the **Job-Runner** over the RESTful API with the credentials you've just setup.

---

5. Create a worker-pool:

- (a) Go to *Job Runner - Worker pools*
- (b) Click on the **Add worker pool** button

- Title: Test Worker Pool
- Workers: Select *Test Worker*

(c) Click the save button

---

**Note:** A worker-pool contains one or multiple workers. The advantage of having multiple workers is that jobs using this pool will be loadbalanced. For now however we keep it simple and assign only one worker.

---

6. Create a project:

(a) Go to *Job-Runner - Projects*

(b) Click the **Add project** button

- Title: Test Project
- Viewers (groups): select *Test Group*
- Project admins (groups): select *Test Group*
- Worker pools: select *Test Worker Pool*

(c) Click the save button

---

**Note:** You've just created a project which is visible for users in *Test Group*, as well users in *Test Group* will be able to start had-hoc job-runs and are able to enable / disable the enqueue of a job. *Test Worker Pool* will be available when you create a new job within this project.

---

7. You now have created a group, project and a worker-pool with one 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
```

---

**Note:** The WebSocket server is responsible for sending events (sent by the workers) to the dashboard. This makes it possible to show the realtime state of job-runs.

---

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! An other reason could be that your browser can't connect to the port the WebSocket server is listening on (eg: when you are running the components in a VM and port 5000 is not forwarded).

## 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.

---

**Note:** There are two ways of rescheduling:

1. re-scheduling by *schedule dts*
2. re-scheduling by *complete dts*

If you have a job-run that is scheduled to run every minute (starting 00:00:00) and would take 25 seconds to complete, it would run at the following times for each case:

1. 00:00:00 - 00:01:00 (00:00:00 + 1 min) - 00:02:00 (00:01:00 + 1 min)
2. 00:00:00 - 00:01:25 (00:00:25 + 1 min) - 00:02:50 (00:01:50 + 1 min)

- 
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 }}
```
      - Project: Select *Test Project*

- (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: Select *Python*
    - Worker pool: Select *Test Worker Pool*
    - 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*
  - (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*!



# 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_HEALTH_CHECK_INTERVAL = 300
```

The interval in seconds for running the health check.

```
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.

This is needed since the ping / pong are async (the PING is sent over ZMQ, the pong is done by making a request to the REST API).

```
job_runner.settings.base.JOB_RUNNER_WORKER_UNRESPONSIVE_AFTER_INTERVALS = 3
```

The number of missed ping responses after which to declare a worker unresponsive.

```
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/
```

```
job_runner.settings.base.JOB_RUNNER_WS_SERVER_HOSTNAME = 'localhost'
```

The hostname of the WebSocket Server.

```
job_runner.settings.base.JOB_RUNNER_WS_SERVER_PORT = 5555
```

The port of the WebSocket Server.



# 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 auth-group that is linked to the project 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 belongs to) has at least the following permissions:

- All *admin* | *log entry* | ...
- All *job\_runner* | *job* | ...
- All *job\_runner* | *job template* | ...
- All *job\_runner* | *reschedule exclude* | ...
- All *job\_runner* | *run* | ...
- All *sessions* | ...

Of course, you can finegrain this to your own needs (eg when you don't want your users to create or delete job-templates).

**See Also:**

*Admin interface* for more technical details

# OVERVIEW

**Job-Runner** is a crontab like tool, with a nice web-frontend for administration and (live) monitoring the current status.

Features:

- Schedule recurring jobs
- Chaining of jobs
- Load-balance workers by putting them in a pool
- Schedule jobs to run on all workers within a pool
- Live dashboard (with option to kill runs and ad-hoc scheduling)
- Multiple projects and per-project permission management

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 and a long-running process to alert when workers are unresponsive (`manage.py health_check`). 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>



## LINKS

- [documentation](#)
- [job-runner source](#)
- [job-runner-worker source](#)
- [job-runner-ws-server source](#)



# 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 job-templates and jobs the user has access to. The same applies to the project and job-template and parent dropdown boxes when editing a job(-template).

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

**Warning:** This only applies to the admin of job-templates and 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': 'project__auth_groups',
        'model': JobTemplate,
    }
}
```

### **groups\_path = None**

A `str` containing the path to the groups.

Example:

```
'job_template__project__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.

## Worker-pools

**GET** `/api/v1/worker_pool/` Returns a list of available worker-pools.

**GET** `/api/v1/worker-pools/{WORKER_POOL_ID}/` Return the details of a specific worker-pool 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.

**PATCH** `/api/v1/worker/{WORKER_ID}/` Update one or more worker fields (used to update the `ping_response_dts`).

## 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.

**PUT** `/api/v1/job/{JOB_ID}/` Update the job (used by AngularJS to enable / disable enqueue of a job).

## Runs

**GET** `/api/v1/run/` Returns 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)
- `last_completed` (last completed runs for each job)

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

**POST** `/api/v1/run/` Create a new run (used for ad-hoc scheduling a job in the dashboard).

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

### Kill-requests

**GET** `/api/v1/kill_request/` Returns a list of kill-requests.

**GET** `/api/v1/kill_request/{KILL_REQUEST_ID}/` Returns the details of a specific kill-request id.

**POST** `/api/v1/kill_request/` Create a new kill-request.

**PATCH** `/api/v1/kill_request/{KILL_REQUEST_ID}/` Update one or more fields of the given kill-request id.

### Run-logs

**GET** `/api/v1/run_log/` Returns a list of run-logs.

**GET** `/api/v1/run_log/{RUN_LOG_ID}/` Returns the details of a specific run-log id.

**POST** `/api/v1/run_log/` Create a new run-log.

**PATCH** `/api/v1/run_log/{RUN_LOG_ID}/` Update one or more fields of the given run-log id.

# PYTHON MODULE INDEX

j

`job_runner.settings.base`, ??