
# Django Q Documentation

***Release 0.2.0***

**Ilan Steemers**

July 07, 2015

Contents

Django Q is a native Django task queue and worker application using Python multiprocessing.

CHAPTER 1

# Features

- Multiprocessing worker pool
- Asynchronous tasks
- Scheduled and repeated tasks
- Encrypted and compressed packages
- Failure and success database
- Result hooks
- Django Admin integration
- PaaS compatible with multiple instances
- Multi cluster monitor
- Redis
- Python 2 and 3

Django Q is tested with: Python 2.7 & 3.4. Django 1.7.8 & 1.8.2

Contents:

## 1.1 Installation

- Install the latest version with pip:

```
$ pip install django-q
```

- Add `django_q` to `INSTALLED_APPS` in your projects `settings.py`:

```
INSTALLED_APPS = (
    # other apps
    'django_q',
)
```

- Run Django migrations to create the database tables:

```
$ python manage.py migrate
```

- Make sure you have a Redis server running somewhere

### 1.1.1 Configuration

Configuration is handled via the `Q_ClUSTER` dictionary in your `settings.py`

```
# settings.py example
Q_CLUSTER = {
    'name': 'myproject',
    'workers': 8,
    'recycle': 500,
    'timeout': 60,
    'compress': True,
    'save_limit': 250,
    'label': 'Django Q',
    'redis': {
        'host': '127.0.0.1',
        'port': 6379,
        'db': 0, }
}
```

#### name

Used to differentiate between projects using the same Redis server. Defaults to `'default'`. This can be useful if you have several projects using the same Redis server.

**Note:** Tasks are encrypted. When a worker encounters a task it can not decrypt, it will be discarded.

#### workers

The number of workers to use in the cluster. Defaults to CPU count of the current host, but can be set to a custom number.

#### recycle

The number of tasks a worker will process before recycling . Useful to release memory resources on a regular basis. Defaults to `500`.

#### timeout

The number of seconds a worker is allowed to spend on a task before it's terminated. Defaults to `None`, meaning it will never time out. Set this to something that makes sense for your project.

#### compress

Compresses task packages to Redis. Useful for large payloads, but can add overhead when used with many small packages. Defaults to `False`

#### save_limit

Limits the amount of successful tasks saved to Django. - Set to `0` for unlimited. - Set to `-1` for no success storage at all. - Defaults to `250` - Failures are always saved.

#### label

The label used for the Django Admin page. Defaults to `'Django Q'`

#### redis

Connection settings for Redis. Defaults:

```
redis: {
    'host': 'localhost',
    'port': 6379,
    'db': 0,
    'password': None,
    'socket_timeout': None,
    'charset': 'utf-8',
    'errors': 'strict',
    'unix_socket_path': None
}
```

For more information on these settings please refer to the Redis-py documentation

#### django_redis

If you are already using django-redis for your caching, you can take advantage of its excellent connection backend by supplying the name of the cache connection you want to use:

```
# example django-redis connection
Q_CLUSTER = {
    'name': 'DJRedis',
    'workers': 4,
    'timeout': 90,
    'django_redis: 'default'
}
```

---

**Tip:** Django Q uses your `SECRET_KEY` to encrypt task packages and prevent task crossover. So make sure you have it set up in your Django settings.

---

### 1.1.2 Requirements

Django Q is tested for Python 2.7 and 3.4

- Django

  Django Q aims to use as much of Django's standard offerings as possible The code is tested against Django version *1.7.8* and *1.8.2*.

- Django-picklefield

  Used to store args, kwargs and result objects in the database.

- Redis-py

  Andy McCurdy's excellent Redis python client.

- Arrow

  The scheduler uses Chris Smith's wonderful project to determine correct dates in the future.

- Blessed

  This feature-filled fork of Erik Rose's blessings project provides the terminal layout of the monitor.

**Tip:** Install the Hiredis parser:

```
$ pip install hiredis
```

This C library maintained by the core Redis team is faster than the standard PythonParser during high loads.

## 1.2 Tasks

Use *async()* from your code to quickly offload tasks to the `cluster`:

```
from django_q import async, result

# create the task
async('math.copysign', 2, -2)

# or with import and storing the id
import math.copysign

task_id = async(copysign, 2, -2)

# get the result
task_result = result(task_id)

# result returns None if the task has not been executed yet
# so in most cases you will want to use a hook:

async('math.modf', 2.5, hook='hooks.print_result')

# hooks.py
def print_result(task):
    print(task.result)
```

### 1.2.1 Connection pooling

Django Q tries to pass redis connections around its parts as much as possible to save you from running out of connections. When you are making individual calls to *async()* a lot though, it can help to set up a redis connection to pass to *async()*:

```
# redis connection economy example
from django_q import async
from django_q.conf import redis_client

for i in range(50):
    async('math.modf', 2.5, redis=redis_client)
```

**Tip:** If you are using django-redis , you can *configure* Django Q to use its connection pool.

### 1.2.2 Reference

**async** (*func*, **args*, *hook=None*, *redis=None*, ***kwargs*)

Puts a task in the cluster queue

**Parameters**

- **func** (*str or object*) – The task function to execute
- **args** – The arguments for the task function
- **hook** (*str or object*) – Optional function to call after execution
- **redis** – Optional redis connection
- **kwargs** – Keyword arguments for the task function

**Returns** The name of the task

**Return type** str

**result** (*name*)

Gets the result of a previously executed task

**Parameters** **name** (*str*) – the name of the task

**Returns** The result of the executed task

**fetch** (*name*)

Returns a previously executed task

**Parameters** **name** (*str*) – the name of the task

**Returns** The task

**Return type** *Task*

Changed in version 0.2.0.

Renamed from get_task

**class Task**

Database model describing an executed task

**name**

The name of the task

**func**

The function or reference that was executed

**hook**

The function to call after execution.

**args**

Positional arguments for the function.

**kwargs**

Keyword arguments for the function.

**result**

The result object. Contains the error if any occur.

**started**

The moment the task was picked up by a worker

**stopped**

The moment a worker finished this task

**success**

Was the task executed without problems?

**time_taken**()

Calculates the difference in seconds between started and stopped

**classmethod get_result**(*task_name*)
Get a result directly by task name

**class Success**
A proxy model of *Task* with the queryset filtered on *Task.success* is True.

**class Failure**
A proxy model of *Task* with the queryset filtered on *Task.success* is False.

## 1.3 Schedules

Schedules are regular Django models. You can manage them through the *Admin pages* or directly from your code with the *schedule()* function or the *Schedule* model:

```
from django_q import Schedule, schedule

# Use the schedule wrapper

schedule('math.copysign',
         2, -2,
         hook='hooks.print_result',
         schedule_type=Schedule.DAILY)

# Or create the object directly

Schedule.objects.create(func='math.copysign',
                        hook='hooks.print_result',
                        args='2,-2',
                        schedule_type=Schedule.DAILY
                        )
```

### 1.3.1 Reference

**schedule**(*func*, **args*, *hook=None*, *schedule_type='O'*, *repeats=-1*, *next_run=now()*, ***kwargs*)
Creates a schedule

**Parameters**

- **func** (*str*) – the function to schedule. Dotted strings only.
- **args** – arguments for the scheduled function.
- **hook** (*str*) – optional result hook function. Dotted strings only.

- **schedule_type** (*str*) – (O)nce, (H)ourly, (D)aily, (W)eekly, (M)onthly, (Q)uarterly, (Y)early or *Schedule.TYPE*
- **repeats** (*int*) – Number of times to repeat schedule. -1=Always, 0=Never, n =n.
- **next_run** (*datetime*) – Next or first scheduled execution datetime.
- **kwargs** – optional keyword arguments for the scheduled function.

**class Schedule**

A database model for task schedules.

**func**

The function to be scheduled

**hook**

Optional hook function to be called after execution.

**args**

Positional arguments for the function.

**kwargs**

Keyword arguments for the function

**schedule_type**

The type of schedule. Follows *Schedule.TYPE*

**TYPE**

*ONCE*, *HOURLY*, *DAILY*, *WEEKLY*, *MONTHLY*. *QUARTERLY*, *YEARLY*

**repeats**

Number of times to repeat the schedule. -1=Always, 0=Never, n =n. When set to -1, this will keep counting down.

**next_run**

Datetime of the next scheduled execution.

**task**

Name of the last task generated by this schedule.

**last_run**()

Admin link to the last executed task.

**success**()

Returns the success status of the last executed task.

## 1.4 Cluster

Django Q uses Python's multiprocessing module to manage a pool of workers that will handle your tasks. Start your cluster using Django's *manage.py* command:

```
$ python manage.py qcluster
```

You should see the cluster starting

```
10:57:40 [Q] INFO Q Cluster-31781 starting.
10:57:40 [Q] INFO Process-1:1 ready for work at 31784
10:57:40 [Q] INFO Process-1:2 ready for work at 31785
10:57:40 [Q] INFO Process-1:3 ready for work at 31786
10:57:40 [Q] INFO Process-1:4 ready for work at 31787
10:57:40 [Q] INFO Process-1:5 ready for work at 31788
10:57:40 [Q] INFO Process-1:6 ready for work at 31789
10:57:40 [Q] INFO Process-1:7 ready for work at 31790
10:57:40 [Q] INFO Process-1:8 ready for work at 31791
10:57:40 [Q] INFO Process-1:9 monitoring at 31792
10:57:40 [Q] INFO Process-1 guarding cluster at 31783
10:57:40 [Q] INFO Process-1:10 pushing tasks at 31793
10:57:40 [Q] INFO Q Cluster-31781 running.
```

Stopping the cluster with ctrl-c or either the *SIGTERM* and *SIGKILL* signals, will initiate the *Stop procedure*:

```
16:44:12 [Q] INFO Q Cluster-31781 stopping.
16:44:12 [Q] INFO Process-1 stopping cluster processes
16:44:13 [Q] INFO Process-1:10 stopped pushing tasks
16:44:13 [Q] INFO Process-1:6 stopped doing work
16:44:13 [Q] INFO Process-1:4 stopped doing work
16:44:13 [Q] INFO Process-1:1 stopped doing work
16:44:13 [Q] INFO Process-1:5 stopped doing work
16:44:13 [Q] INFO Process-1:7 stopped doing work
16:44:13 [Q] INFO Process-1:3 stopped doing work
16:44:13 [Q] INFO Process-1:8 stopped doing work
16:44:13 [Q] INFO Process-1:2 stopped doing work
16:44:14 [Q] INFO Process-1:9 stopped monitoring results
16:44:15 [Q] INFO Q Cluster-31781 has stopped.
```

### 1.4.1 Multiple Clusters

You can have multiple clusters on multiple machines, working on the same queue as long as:

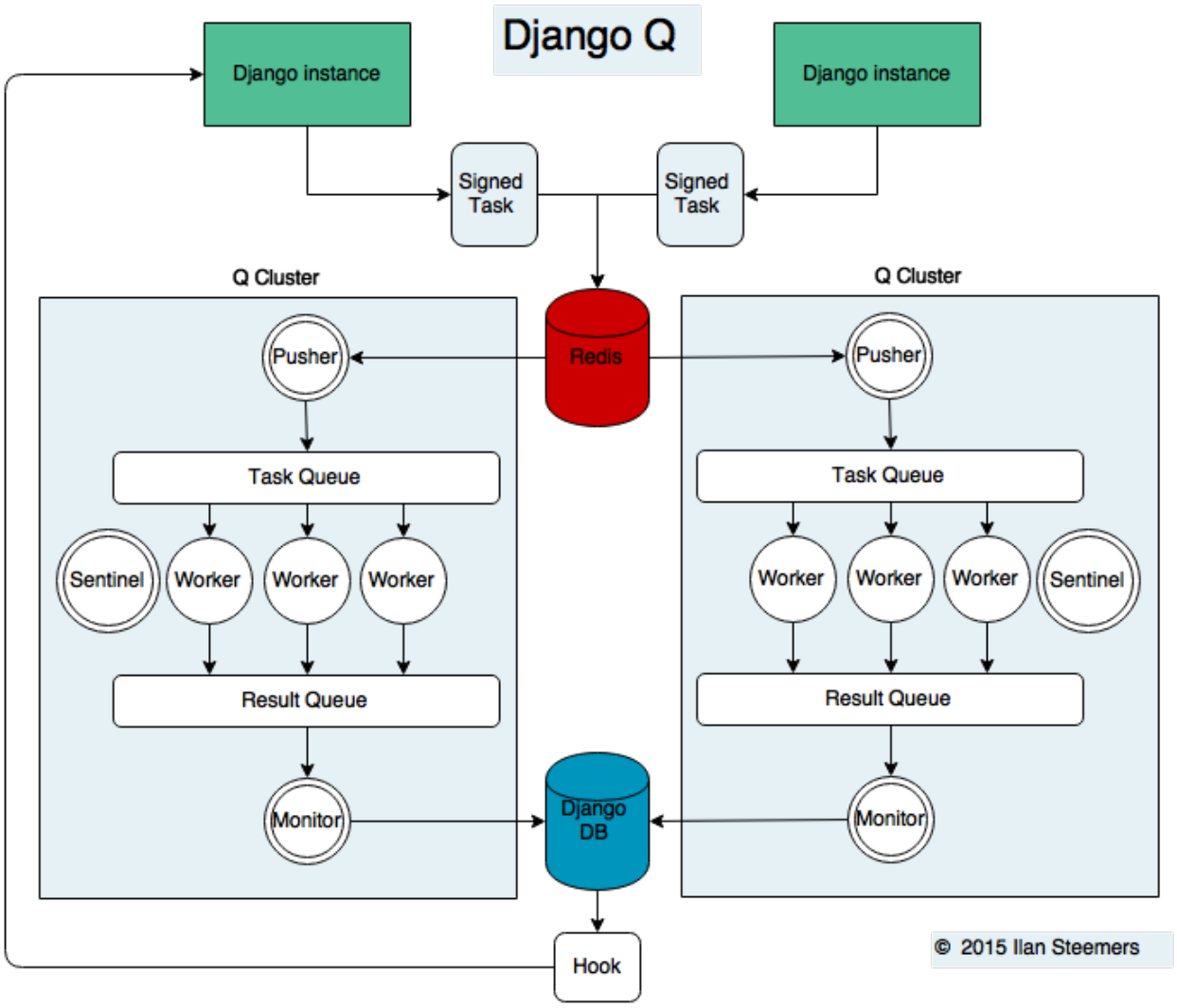
- They connect to the same Redis server.
- They use the same cluster name. See *Configuration*
- They share the same `SECRET_KEY`

### 1.4.2 Using a Procfile

If you host on Heroku or you are using Honcho you can start the cluster from a `Procfile` with an entry like this:

```
worker: python manage.py qcluster
```

### 1.4.3 Architecture




#### Signed Tasks

Tasks are first pickled and then signed using Django's own signing module before being sent to a Redis list. This ensures that task packages on the Redis server can only be executed and read by clusters and django servers who share the same secret key. Optionally the packages can be compressed before transport

#### Pusher

The pusher process continuously checks the Redis list for new task packages and pushes them on the Task Queue.

### Worker

A worker process pulls a package of the Task Queue and checks the signing and unpacks the task. Before executing the task it set a timer on the *Sentinel* indicating its about to start work. Afterwards it the timer is reset and any results (including errors) are saved to the pacjage. Irrespective of the failure or success of any of these steps, the package is then pushed onto the Result Queue.

### Monitor

The result monitor checks the Result Queue for processed packages and saves both failed and successful packages to the Django database.

### Sentinel

The sentinel spawns all process and then checks the health of all workers, including the pusher and the monitor. This includes checking timers on each worker for timeouts. In case of a sudden death or timeout, it will reincarnate the failing processes. When a stop signal, the sentinel will halt the pusher and instruct the workers and monitor to finish the remaining items. See *Stop procedure*

### Timeouts

Before each task execution the worker resets a timer on the sentinel and resets it again after execution. Meanwhile the the sentinel checks if the timers don't exceed the timeout amount, in which case it will terminate the worker and reincarnate a new one.

### Scheduler

Once a minute the scheduler checks for any scheduled task that should be starting.

- Creates a task from the schedule
- Subtracts 1 from `Schedule.repeats`
- Sets the next run time if there are repeats left or if its negative.

### Stop procedure

When a stop signal is given, the sentinel exits the guard loop and instructs the pusher to stop pushing. Once this is confirmed, the sentinel pushes poison pills onto the task queue and will wait for all the workers to die. This ensures that the queue is emptied before the workers exit. Afterwards the sentinel waits for the monitor to empty the result and then the stop procedure is complete.

- Send stop event to pusher
- Wait for pusher to exit
- Put poison pills in the Task Queue
- Wait for all the workers to clear the queue and stop
- Put a poison pill on the Result Queue
- Wait for monitor to process remaining results
- Signal that we have stopped

**Warning:** If you force the cluster to terminate before the stop procedure has completed, you can lose tasks and their results.

### 1.4.4 Reference

**class `Cluster`**

**`start`()**

Spawns a cluster and then returns

**`stop`()**

Initiates *Stop procedure* and waits for it to finish.

**`stat`()**

returns a *`Stat`* object with the current cluster status.

**`pid`**

The cluster process id.

**`host`**

The current hostname

**`sentinel`**

returns the `multiprocessing.Process` containing the `Sentinel`.

**`timeout`**

The clusters timeout setting in seconds

**`start_event`**

A `multiprocessing.Event` indicating if the `Sentinel` has finished starting the cluster

**`stop_event`**

A `multiprocessing.Event` used to instruct the `Sentinel` to initiate the *Stop procedure*

**`is_starting`**

Bool indicating if the cluster is busy starting up

**`is_running`**

Bool. Tells you if the cluster is up and running.

**`is_stopping`**

Bool. Shows that the stop procedure has been started.

**`has_stopped`**

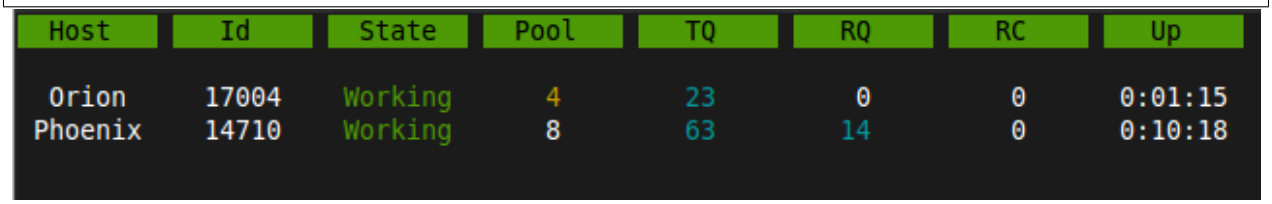
Bool. Tells you if the cluster finished the stop procedure

## 1.5 Monitor

The cluster monitor shows information about all the Q clusters connected to your project.

Start the monitor with Django's *manage.py* command:

```
$ python manage.py qmonitor
```




### 1.5.1 Legend

#### Host

Shows the hostname of the server this cluster is running on.

#### Id

The cluster Id. Same as the cluster process ID or pid.

#### State

Current state of the cluster:

- **Starting** The cluster is spawning workers and getting ready.
- **Idle** Everything is ok, but there are no tasks to process.
- **Working** Processing tasks like a good cluster should.
- **Stopping** The cluster does not take on any new tasks and is finishing.
- **Stopped** All tasks have been processed and the cluster is shutting down.

#### Pool

The current number of workers in the cluster pool.

#### TQ

**Task Queue** counts the number of tasks in the queue

If this keeps rising it means you are taking on more tasks than your cluster can handle.

#### RQ

**Result Queue** shows the number of results in the queue.

Since results are only saved by a single process which has to access the database. It's normal for the result queue to take slightly longer to clear than the task queue.

#### RC

**Reincarnations** shows the amount of processes that have been reincarnated after a sudden death or timeout. If this number is unusually high, you are either suffering from repeated task errors or severe timeouts and you should check your logs for details.

#### Up

**Uptime** the amount of time that has passed since the cluster was started.

Press *q* to quit the monitor and return to your terminal.

### 1.5.2 Status

You can check the status of your clusters straight from your code with *Stat*:

```
from django_q.monitor import Stat

for stat in Stat.get_all():
    print(stat.cluster_id, stat.status)

# or if you know the cluster id
cluster_id = 1234
stat = Stat.get(cluster_id)
print(stat.status, stat.workers)
```

### 1.5.3 Reference

**class Stat**

Cluster status object.

**cluster_id**

Id of this cluster. Corresponds with the process id.

**tob**

Time Of Birth

**uptime**()

Shows the number of seconds passed since the time of birth

**reincarnations**

The number of times the sentinel had to start a new worker process.

**status**

String representing the current cluster status.

**task_q_size**

The number of tasks currently in the task queue.

**done_q_size**

The number of tasks currently in the result queue.

**pusher**

The pid of the pushes process

**monitor**

The pid of the monitor process

**sentinel**

The pid of the sentinel process

**workers**

A list of process ids of the workers currently in the cluster pool.

**empty_queues**()

Returns true or false depending on any tasks still present in the task or result queue.

**classmethod get**(*cluster_id*, *r=redis_client*)

Gets the current *Stat* for the cluster id. Takes an optional redis connection.

**classmethod get_all**(*r=redis_client*)

Returns a list of *Stat* objects for all active clusters. Takes an optional redis connection.

## 1.6 Admin pages

Django Q does not use custom HTML pages, but instead uses what is offered by Django's model admin by default. When you open Django Q's admin pages you will see three models:

### 1.6.1 Successful tasks

Shows all successfully executed tasks. Meaning they did not encounter any errors during execution. From here you can look at details of each task or delete them.

Uses the *Success* proxy model.

**Tip:** The maximum number of successful tasks can be set using the *save_limit* *Configuration* option.

### 1.6.2 Failed tasks

Failed tasks have encountered an error, preventing them from finishing execution. The worker will try to put the error in the *result* field of the task so you can review what happened.

You can resubmit a failed task back to the queue using the admins action menu.

Uses the *Failure* proxy model

### 1.6.3 Scheduled tasks

Uses the *Schedule* model

- genindex
- search

## T



## U



## W