Django Q Documentation

Release 0.4.6

Ilan Steemers

August 19, 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 and groups
- Django Admin integration
- PaaS compatible with multiple instances
- Multi cluster monitor
- Redis broker
- Python 2 and 3

Django Q is tested with: Python 2.7 & 3.4. Django 1.7.9 & 1.8.3

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 and know how to connect to it.

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,
    'queue_limit': 500,
    'cpu_affinity': 1,
    '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. [1]

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. Can be overridden for individual tasks.

compress

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

[1] Uses `multiprocessing.cpu_count()` which can fail on some platforms. If so , please set the worker count in the configuration manually or install *psutil* to provide an alternative cpu count method.

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.

queue_limit

This does not limit the amount of tasks that can be queued overall on Redis, but rather how many tasks are kept in memory by a single cluster. Setting this to a reasonable number, can help balance the workload and the memory overhead of each individual cluster. It can also be used to manage the loss of data in case of a cluster failure. Defaults to `None`, meaning no limit.

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.

cpu_affinity

Sets the number of processor each worker can use. This does not affect auxiliary processes like the sentinel or monitor and is only useful for tweaking the performance of very high traffic clusters. The affinity number has to be higher than zero and less than the total number of processors to have any effect. Defaults to using all processors:

```
# processor affinity example.

4 processors, 4 workers, cpu_affinity: 1

worker 1 cpu [0]
worker 2 cpu [1]
worker 3 cpu [2]
worker 4 cpu [3]

4 processors, 4 workers, cpu_affinity: 2

worker 1 cpu [0, 1]
worker 2 cpu [2, 3]
worker 3 cpu [0, 1]
worker 4 cpu [2, 3]

8 processors, 8 workers, cpu_affinity: 3

worker 1 cpu [0, 1, 2]
worker 2 cpu [3, 4, 5]
worker 3 cpu [6, 7, 0]
worker 4 cpu [1, 2, 3]
worker 5 cpu [4, 5, 6]
worker 6 cpu [7, 0, 1]
worker 7 cpu [2, 3, 4]
worker 8 cpu [5, 6, 7]
```

In some cases, setting the cpu affinity for your workers can lead to performance improvements, especially if the load is high and consists of many repeating small tasks. Start with an affinity of 1 and work your way up. You will have to experiment with what works best for you. As a rule of thumb; cpu_affinity 1 favors repetitive short running tasks, while no affinity benefits longer running tasks.

Note: The `cpu_affinity` setting requires the optional *psutil* module.

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 *1.7.9* and *1.8.3*.

- 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.

- Redis server

  Django Q uses Redis as a centralized hub between your Django instances and your Q clusters.

Optional

- Psutil python system and process utilities module by Giampaolo Rodola', is an optional requirement and adds cpu affinity settings to the cluster:

```
$ pip install psutil
```

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

```
$ pip install hiredis
```

1.2 Tasks

1.2.1 Async

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)
```

async() can take the following optional keyword arguments:

hook

The function to call after the task has been executed. This function gets passed the complete *Task* object as its argument.

group

A group label. Check *Groups* for group functions.

save

Overrides the result backend's save setting for this task.

timeout

Overrides the cluster's timeout setting for this task.

sync

Simulates a task execution synchronously. Useful for testing.

redis

A redis connection. In case you want to control your own connections.

q_options

None of the option keywords get passed on to the task function. As an alternative you can also put them in a single keyword dict named `q_options`. This enables you to use these keywords for your function call:

```
# Async options in a dict

opts = {'hook': 'hooks.print_result',
        'group': 'math',
        'timeout': 30}

async('math.modf', 2.5, q_options=opts)
```

Please not that this will override any other option keywords.

1.2.2 Groups

You can group together results by passing *`async()`* the optional `group` keyword:

```
# result group example
from django_q import async, result_group

for i in range(4):
    async('math.modf', i, group='modf')

# after the tasks have finished you can get the group results
result = result_group('modf')
print(result)
```

```
[(0.0, 0.0), (0.0, 1.0), (0.0, 2.0), (0.0, 3.0)]
```

Take care to not limit your results database too much and call *delete_group()* before each run, unless you want your results to keep adding up. Instead of *result_group()* you can also use *fetch_group()* to return a queryset of *Task* objects.:

```
# fetch group example
from django_q import fetch_group, count_group, result_group

# count the number of failures
failure_count = count_group('modf', failures=True)

# only use the successes
results = fetch_group('modf')
if failure_count:
    results = results.exclude(success=False)
results =  [task.result for task in successes]

# this is the same as
results = fetch_group('modf', failures=False)
results =  [task.result for task in successes]

# and the same as
results = result_group('modf') # filters failures by default
```

Getting results by using *result_group()* is of course much faster than using *fetch_group()*, but it doesn't offer the benefits of Django's queryset functions.

Note: Although *fetch_group()* returns a queryset, due to the nature of the PickleField , calling `Queryset.values` on it will return a list of encoded results. Use list comprehension or an iterator instead.

1.2.3 Synchronous testing

async() can be instructed to execute a task immediately by setting the optional keyword `sync=True`. The task will then be injected straight into a worker and the result saved by a monitor instance:

```
from django_q import async, fetch

# create a synchronous task
task_id = async('my.buggy.code', sync=True)

# the task will then be available immediately
task = fetch(task_id)

# and can be examined
if not task.success:
    print('An error occurred: {}'.format(task.result))
```

```
An error occurred: ImportError("No module named 'my'",)
```

Note that *async()* will block until the task is executed and saved. This feature bypasses the Redis server and is intended for debugging and development.

1.2.4 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 reuse

for *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.

1.2.5 Reference

async(*func*, **args*, *hook=None*, *group=None*, *timeout=None*, *save=None*, *sync=False*, *redis=None*, *q_options=None*, ***kwargs*)

Puts a task in the cluster queue

Parameters

- **func** (*object*) – The task function to execute
- **args** (*tuple*) – The arguments for the task function
- **hook** (*object*) – Optional function to call after execution
- **group** (*str*) – An optional group identifier
- **timeout** (*int*) – Overrides global cluster *timeout*.
- **save** (*bool*) – Overrides global save setting for this task.
- **sync** (*bool*) – If set to True, async will simulate a task execution
- **redis** – Optional redis connection
- **q_options** (*dict*) – Options dict, overrides option keywords
- **kwargs** (*dict*) – Keyword arguments for the task function

Returns The uuid of the task

Return type str

result(*task_id*)

Gets the result of a previously executed task

Parameters **task_id** (*str*) – the uuid or name of the task

Returns The result of the executed task

fetch(*task_id*)

Returns a previously executed task

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

Returns The task if any

Return type *Task*

Changed in version 0.2.0.

Renamed from get_task

result_group(*group_id*, *failures=False*)
Returns the results of a task group

Parameters

- **group_id** (*str*) – the group identifier
- **failures** (*bool*) – set this to `True` to include failed results

Returns a list of results

Return type list

fetch_group(*group_id*, *failures=True*)
Returns a list of tasks in a group

Parameters

- **group_id** (*str*) – the group identifier
- **failures** (*bool*) – set this to `False` to exclude failed tasks

Returns a list of Tasks

Return type list

count_group(*group_id*, *failures=False*)
Counts the number of task results in a group.

Parameters

- **group_id** (*str*) – the group identifier
- **failures** (*bool*) – counts the number of failures if `True`

Returns the number of tasks or failures in a group

Return type int

delete_group(*group_id*, *tasks=False*)
Deletes a group label from the database.

Parameters

- **group_id** (*str*) – the group identifier
- **tasks** (*bool*) – also deletes the associated tasks if `True`

Returns the numbers of tasks affected

Return type int

class **Task**
Database model describing an executed task

id

An `uuid.uuid4()` identifier

name

The name of the task as a humanized version of the *id*

Note: This is for convenience and can be used as a parameter for most functions that take a *task_id*. Keep in mind that it is not guaranteed to be unique if you store very large amounts of tasks in the database.

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 created by an async command

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.

> **Note:** Time taken represents the time a task spends in the cluster, this includes any time it may have waited in the queue.

classmethod get_result(*task_id*)

Gets a result directly by task uuid or name.

classmethod get_result_group(*group_id*, *failures=False*)

Returns a list of results from a task group. Set failures to `True` to include failed results.

classmethod get_task(*task_id*)

Fetches a single task object by uuid or name.

classmethod get_task_group(*group_id*, *failures=True*)

Gets a queryset of tasks with this group id. Set failures to `False` to exclude failed tasks.

classmethod get_group_count(*group_id*, *failures=False*)

Returns a count of the number of tasks results in a group. Returns the number of failures when `failures=True`

classmethod delete_group(*group_id*, *objects=False*)

Deletes a group label only, by default. If `objects=True` it will also delete the tasks in this group from the database.

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

1.3.1 Schedule

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
                        )

# In case you want to use async options
schedule('math.sqrt',
         9,
         hook='hooks.print_result',
         q_options={'timeout': 30},
         schedule_type=Schedule.HOURLY)
```

1.3.2 Management Commands

If you want to schedule regular Django management commands, you can use the `django.core.management` module to make a wrapper function which you can schedule in Django Q:

```
# tasks.py
from django.core import management

# wrapping `manage.py clearsessions`
def clear_sessions_command():
    return management.call_command('clearsessions')

# now you can schedule it to run every hour
from django_q import schedule

schedule('tasks.clear_sessions_command', schedule_type='H')
```

1.3.3 Reference

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

Creates a schedule

Parameters

- **func** (*str*) – the function to schedule. Dotted strings only.

- **args** – arguments for the scheduled function.
- **name** (*str*) – An optional name for your schedule.
- **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.
- **q_options** (*dict*) – async options to use for this schedule
- **kwargs** – optional keyword arguments for the scheduled function.

class Schedule

A database model for task schedules.

id

Primary key

name

A name for your schedule. Tasks created by this schedule will assume this or the primary key as their group id.

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

Id 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.

ONCE

'O' the schedule will only run once. If it has a negative `repeats` it will be deleted after it has run. If you want to keep the result, set `repeats` to a positive number.

HOURLY

'H' the scheduled task will run every hour after its first run.

DAILY

'D' the scheduled task will run every day at the time of its first run.

WEEKLY

'W' the task will run every week on they day and time of the first run.

MONTHLY

'M' the tasks runs every month on they day and time of the last run.

Note: Months are tricky. If you schedule something on the 31st of the month and the next month has only 30 days or less, the task will run on the last day of the next month. It will however continue to run on that day, e.g. the 28th, in subsequent months.

QUARTERLY

'Q' this task runs once every 3 months on the day and time of the last run.

YEARLY

'Y' only runs once a year. The same caution as with months apply; If you set this to february 29th, it will run on february 28th in the following years.

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.
```

The number of workers, optional timeouts, recycles and cpu_affinity can be controlled via the *Configuration* settings.

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 or Redis cluster.
- They use the same cluster name. See *Configuration*
- They share the same `SECRET_KEY` for Django.

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 Process managers

While you certainly can run a Django Q with a process manager like Supervisor or Circus it is not strictly necessary. The cluster has an internal sentinel that checks the health of all the processes and recycles or reincarnates according to your settings or in case of unexpected crashes. Because of the multiprocessing daemonic nature of the cluster, it is impossible for a process manager to determine the clusters health and resource usage.

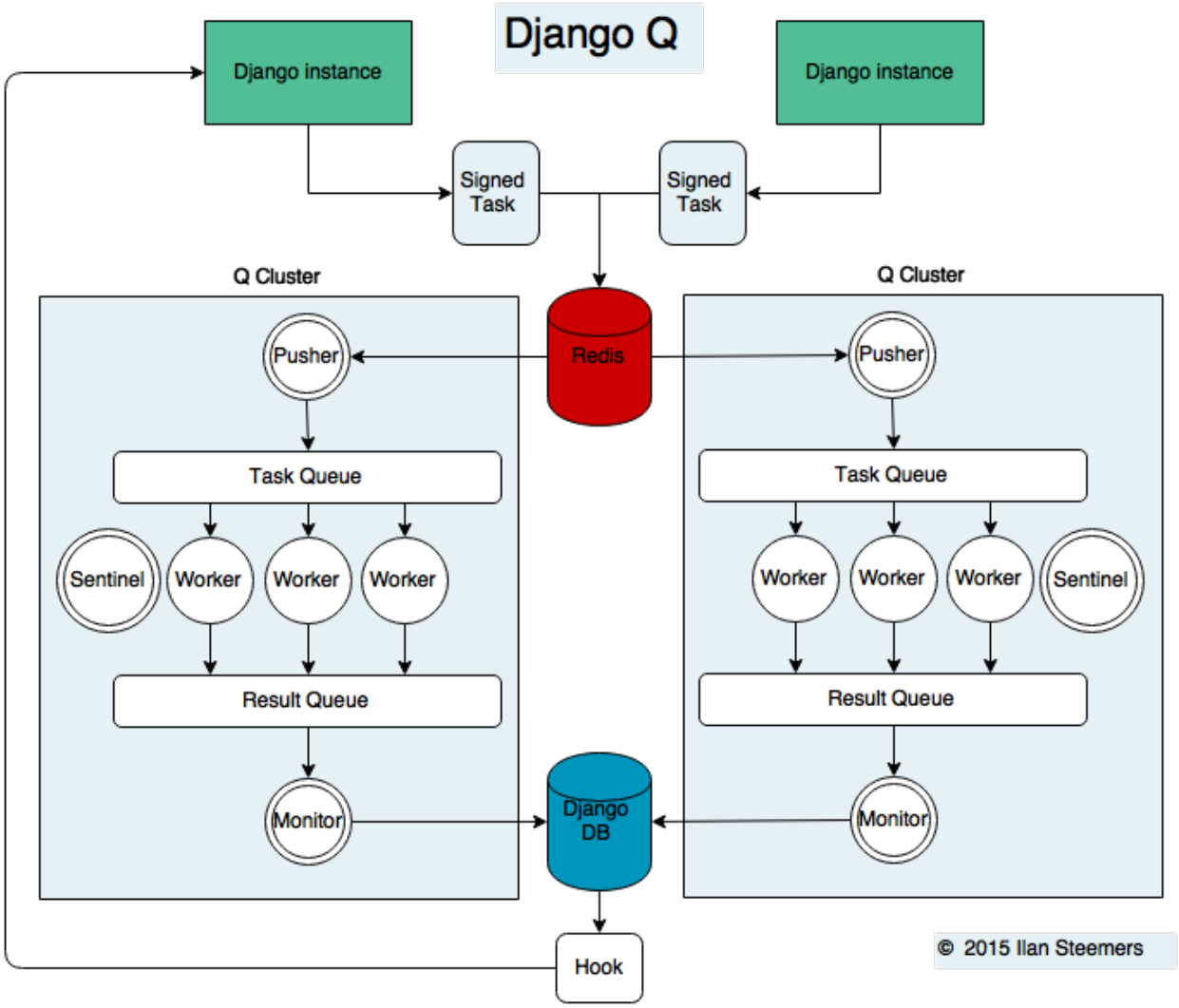
An example `circus.ini`

```
[circus]
check_delay = 5
endpoint = tcp://127.0.0.1:5555
pubsub_endpoint = tcp://127.0.0.1:5556
stats_endpoint = tcp://127.0.0.1:5557

[watcher:django_q]
cmd = python manage.py qcluster
numprocesses = 1
copy_env = True
```

Note that we only start one process. It is not a good idea to run multiple instances of the cluster in the same environment since this does nothing to increase performance and in all likelihood will diminish it. Control your cluster using the `workers`, `recycle` and `timeout` settings in your *Configuration*

1.4.4 Architecture



Signed Tasks

Tasks are first pickled and then signed using Django's own `django.core.signing` module using the `SECRET_KEY` and cluster name as salt, 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 and cluster name. Optionally the packages can be compressed before transport

Pusher

The pusher process continuously checks the Redis list for new task packages. It checks the signing and unpacks the task to the Task Queue.

Worker

A worker process pulls a task of the Task Queue and it sets a shared countdown timer with *Sentinel* indicating it is about to start work. The worker then tries to execute the task and afterwards the timer is reset and any results (including

errors) are saved to the package. 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 is received, 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 sets a countdown timer on the sentinel and resets it again after execution. Meanwhile the sentinel checks if the timers don't reach zero, in which case it will terminate the worker and reincarnate a new one.

Scheduler

Twice a minute the scheduler checks for any scheduled tasks that should be starting.

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

Stop procedure

When a stop signal is received, 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 exit. This ensures that the task queue is emptied before the workers exit. Afterwards the sentinel waits for the monitor to empty the result queue and 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 and exit
- Signal that we have stopped

Warning: If you force the cluster to terminate before the stop procedure has completed, you can lose tasks or results still being held in memory. You can manage the amount of tasks in a clusters memory by setting the *queue_limit*.

1.4.5 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 that 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`

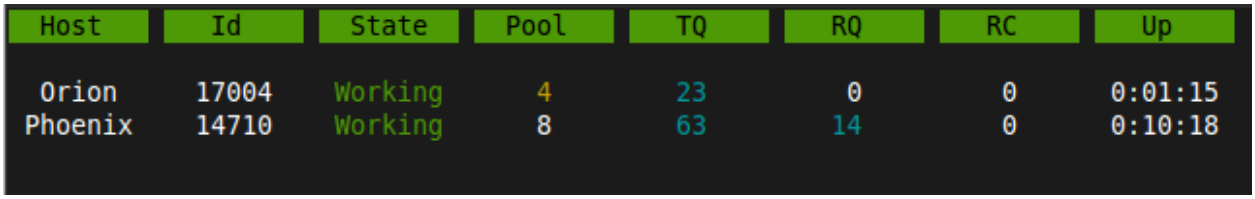
Bool. Tells you if the cluster has finished the stop procedure

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
```



Host	Id	State	Pool	TQ	RQ	RC	Up
Orion	17004	Working	4	23	0	0	0:01:15
Phoenix	14710	Working	8	63	14	0	0:10:18

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 [2]

If this keeps rising it means you are taking on more tasks than your cluster can handle. You can limit this by settings the *queue_limit* in your cluster configuration, after which it will turn green when that limit has been reached. If your task queue is always hitting its limit and your running out of resources, it may be time to add another cluster.

RQ

Result Queue shows the number of results in the queue. [1]

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.

[2] Uses `multiprocessing.Queue.qsize()` which is not implemented on OS X and always returns 0.

RC

Reincarnations shows the amount of processes that have been reincarnated after a recycle, 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. [1]

done_q_size

The number of tasks currently in the result queue. [1]

pusher

The pid of the pusher 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. Use the group column to sort your results by schedule name or group id. The table is searchable by *name*, *func* and *group*

Uses the *Success* proxy model.

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

Here you can check on the status of your scheduled tasks, create, edit or delete them.

Repeats

If you want a schedule to only run a finite amount of times, e.g. every hour for the next 24 hours, you can do that using the *Schedule.repeats* attribute. In this case you would set the schedule type to *Schedule.HOURLY* and the repeats to *24*. Every time the schedule runs the repeats count down until it hits zero and schedule is no longer run.

When you set repeats to `-1` the schedule will continue indefinitely and the repeats will still count down. This can be used as an indicator of how many times the schedule has been executed.

An exception to this are schedules of type *Schedule.ONCE*. Negative repeats for this schedule type will cause it to be deleted from the database. This behavior is useful if you have many delayed actions which you do not necessarily need a result for. A positive number will keep the ONCE schedule, but it will not run again.

Note: To run a `ONCE` schedule again, change the repeats to something other than *0*. Set a new run time before you do this or let it execute immediately.

Next run

Shows you when this task will be added to the queue next.

Last run

Links to the task result of the last scheduled run. Shows nothing if the schedule hasn't run yet or if task result has been deleted.

Success

Indicates the success status of the last scheduled task, if any.

Note: if you have set the *save_limit* configuration option to not save successful tasks to the database, you will only see the failed results of your schedules.

Uses the *Schedule* model

1.7 Examples

1.7.1 Emails

Sending an email can take a while so why not queue it:

```
# Welcome mail with follow up example
from datetime import timedelta
from django.utils import timezone
from django_q import async, schedule, Schedule


def welcome_mail(user):
    msg = 'Welcome to our website'
    # send this message right away
    async('django.core.mail.send_mail',
          'Welcome',
```

```
            msg,
            'from@example.com',
            [user.email])
    # and this follow up email in one hour
    msg = 'Here are some tips to get you started...'
    schedule('django.core.mail.send_mail',
             'Follow up',
             msg,
             'from@example.com',
             [user.email],
             schedule_type=Schedule.ONCE,
             next_run=timezone.now() + timedelta(hours=1))

    # since the `repeats` defaults to -1
    # this schedule will erase itself after having run
```

Since you're only telling Django Q to take care of the emails, you can quickly move on to serving web pages to your user.

1.7.2 Signals

A good place to use async tasks are Django's model signals. You don't want to delay the saving or creation of objects, but sometimes you want to trigger a lot of actions:

```
# Message on object change
from django.contrib.auth.models import User
from django.db.models.signals import pre_save
from django.dispatch import receiver
from django_q import async

# set up the pre_save signal for our user
@receiver(pre_save, sender=User)
def email_changed(sender, instance, **kwargs):
    try:
        user = sender.objects.get(pk=instance.pk)
    except sender.DoesNotExist:
        pass  # new user
    else:
        # has his email changed?
        if not user.email == instance.email:
            # tell everyone
            async('tasks.inform_everyone', instance)
```

The task will send a message to everyone else informing them that the users email address has changed. Note that this adds almost no overhead to the save action:

```
# tasks.py
def inform_everyone(user):
    mails = []
    for u in User.objects.exclude(pk=user.pk):
        msg = 'Dear {}, {} has a new email address: {}'
        msg = msg.format(u.username, user.username, user.email)
        mails.append(('New email', msg,
                      'from@example.com', [u.email]))
    return send_mass_mail(mails)
```

```
# or do it async again
def inform_everyone_async(user):
    for u in User.objects.exclude(pk=user.pk):
        msg = 'Dear {}, {} has a new email address: {}'
        msg = msg.format(u.username, user.username, user.email)
        async('django.core.mail.send_mail',
              'New email', msg, 'from@example.com', [u.email])
```

Of course you can do other things beside sending emails. These are just generic examples. You can use signals with async to update fields in other objects too. Let's say this users email address is not just on the User object, but you stored it in some other places too without a reference. By attaching an async action to the save signal, you can now update that email address in those other places without impacting the the time it takes to return your views.

1.7.3 Reports

In this example the user requests a report and we let the cluster do the generating, while handling the result with a hook.

```
# Report generation with hook example
from django_q import async

# views.py
# user requests a report.
def create_report(request):
    async('tasks.create_html_report',
          request.user,
          hook='tasks.email_report')
```

```
# tasks.py
from django_q import async

# report generator
def create_html_report(user):
    html_report = 'We had a great quarter!'
    return html_report

# report mailer
def email_report(task):
    if task.success:
        # Email the report
        async('django.core.mail.send_mail',
              'The report you requested',
              task.result,
              'from@example.com',
              task.args[0].email)
    else:
        # Tell the admins something went wrong
        async('django.core.mail.mail_admins',
              'Report generation failed',
              task.result)
```

The hook is practical here, cause it allows us to detach the sending task from the report generation function and to report on possible failures.

1.7.4 Haystack

If you use Haystack as your projects search engine, here's an example of how you can have Django Q take care of your indexes in real time using model signals:

```
# Real time Haystack indexing
from .models import Document
from django.db.models.signals import post_save
from django.dispatch import receiver
from django_q import async

# hook up the post save handler
@receiver(post_save, sender=Document)
def document_changed(sender, instance, **kwargs):
    async('tasks.index_object', sender, instance, save=False)
    # turn off result saving to not flood your database
```

```
# tasks.py
from haystack import connection_router, connections

def index_object(sender, instance):
    # get possible backends
    backends = connection_router.for_write(instance=instance)

    for backend in backends:
        # get the index for this model
        index = connections[backend].get_unified_index()\
            .get_index(sender)
        # update it
        index.update_object(instance, using=backend)
```

Now every time a Document is saved, your indexes will be updated without causing a delay in your save action. You could expand this to dealing with deletes, by adding a `post_delete` signal and calling `index.remove_object` in the async function.

1.7.5 Groups

A group example with Kernel density estimation for probability density functions using the Parzen-window technique. Adapted from Sebastian Raschka's blog

```
# Group example with Parzen-window estimation
import numpy

from django_q import async, result_group,\
    count_group, delete_group

# the estimation function
def parzen_estimation(x_samples, point_x, h):
    k_n = 0
    for row in x_samples:
        x_i = (point_x - row[:, numpy.newaxis]) / h
        for row in x_i:
            if numpy.abs(row) > (1 / 2):
                break
        else:
            k_n += 1
    return h, (k_n / len(x_samples)) / (h ** point_x.shape[1])
```

```
# create 100 calculations and send them to the cluster
def parzen_async():
    # clear the previous results
    delete_group('parzen', tasks=True)
    mu_vec = numpy.array([0, 0])
    cov_mat = numpy.array([[1, 0], [0, 1]])
    sample = numpy.random.\
        multivariate_normal(mu_vec, cov_mat, 10000)
    widths = numpy.linspace(1.0, 1.2, 100)
    x = numpy.array([[0], [0]])
    # async them with a group label and a hook
    for w in widths:
        async(parzen_estimation, sample, x, w,
         group='parzen', hook=parzen_hook)

# wait for 100 results to return and print it.
def parzen_hook(task):
    if count_group('parzen') == 100:
        print(result_group('parzen'))
```

Django Q is not optimized for distributed computing, but this example will give you an idea of what you can do with task *Groups*.

Note: If you have an example you want to share, please submit a pull request on github.

- genindex
- search

d

S

T

U

W

Y