



Django Q

Django Q Documentation

Release 0.6.4

Ilan Steemers

September 10, 2015

1	Features	3
1.1	Installation	3
1.2	Configuration	4
1.3	Brokers	9
1.4	Tasks	12
1.5	Schedules	17
1.6	Cluster	20
1.7	Monitor	26
1.8	Admin pages	28
1.9	Examples	30
	Python Module Index	35

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

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, Disque, IronMQ or SQS
- Python 2 and 3

Django Q is tested with: Python 2.7 & 3.4. Django 1.7.10 & 1.8.4

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
```

- Choose a message [broker](#) , configure it and install the appropriate client library.

1.1.1 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 *1.7.10* and *1.8.4*.
- [Django-picklefield](#)
Used to store args, kwargs and result objects in the database.
- [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.

Optional

- [Redis-py](#) client by Andy McCurdy is used to interface with both the Redis and Disque brokers:

```
$ pip install redis
```

- [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
```

- [Boto3](#) is used for the Amazon SQS broker in favor of the now deprecating boto library:

```
$ pip install boto3
```

- [Iron-mq](#) is the official python binding for the IronMQ broker:

```
$ pip install iron-mq
```

- [Redis](#) server is the default broker for Django Q. It provides the best performance and does not require Django's cache framework for monitoring.
- [Disque](#) server is based on Redis by the same author, but focuses on reliable queues. Currently in Alpha, but highly recommended. You can either build it from source or use it on Heroku through the [Tynd](#) beta.

1.2 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, }
}

```

All configuration settings are optional:

1.2.1 name

Used to differentiate between projects using the same broker. On most broker types this will be used as the queue name. Defaults to `'default'`.

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

1.2.2 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

1.2.3 recycle

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

1.2.4 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.

1.2.5 retry

The number of seconds a broker will wait for a cluster to finish a task, before it's presented again. Only works with brokers that support delivery receipts. Defaults to 60 seconds.

1.2.6 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.

1.2.7 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.

1.2.8 sync

When set to `True` this configuration option forces all `async()` calls to be run with `sync=True`. Effectively making everything synchronous. Useful for testing. Defaults to `False`.

1.2.9 queue_limit

This does not limit the amount of tasks that can be queued on the broker, 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. Defaults to `workers**2`.

1.2.10 label

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

1.2.11 catch_up

The default behavior for schedules that didn't run while a cluster was down, is to play catch up and execute all the missed time slots until things are back on schedule. You can override this behavior by setting `catch_up` to `False`. This will make those schedules run only once when the cluster starts and normal scheduling resumes. Defaults to `True`.

1.2.12 redis

Connection settings for Redis. Defaults:

```
# redis defaults
Q_CLUSTER = {
    '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

1.2.13 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 instead of a direct Redis connection:

```
# 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.

1.2.14 disque_nodes

If you want to use Disque as your broker, set this to a list of available Disque nodes and each cluster will randomly try to connect to them:

```
# example disque connection
Q_CLUSTER = {
    'name': 'DisqueBroker',
    'workers': 4,
    'timeout': 60,
    'retry': 60,
    'disque_nodes': ['127.0.0.1:7711', '127.0.0.1:7712']
}
```

Django Q is also compatible with the `Tynd Disque` addon on [Heroku](#):

```
# example Tynd Disque connection
import os

Q_CLUSTER = {
    'name': 'TyndBroker',
    'workers': 8,
    'timeout': 30,
    'retry': 60,
    'bulk': 10,
    'disque_nodes': os.environ['TYND_DISQUE_NODES'].split(','),
    'disque_auth': os.environ['TYND_DISQUE_AUTH']
}
```

1.2.15 disque_auth

Optional Disque password for servers that require authentication.

1.2.16 iron_mq

Connection settings for IronMQ:

```
# example IronMQ connection

Q_CLUSTER = {
    'name': 'IronBroker',
    'workers': 8,
    'timeout': 30,
    'retry': 60,
    'queue_limit': 50,
    'bulk': 10,
    'iron_mq': {
        'host': 'mq-aws-us-east-1.iron.io',
        'token': 'Et1En7.....0LuW39Q',
        'project_id': '500f7b....b0f302e9'
    }
}
```

All connection keywords are supported. See the [iron-mq](#) library for more info

1.2.17 sqs

To use Amazon SQS as a broker you need to provide the AWS region and credentials:

```
# example SQS broker connection

Q_CLUSTER = {
    'name': 'SQSExample',
    'workers': 4,
    'timeout': 60,
    'retry': 90,
    'queue_limit': 100,
    'bulk': 5,
    'sqs': {
        'aws_region': 'us-east-1',
        'aws_access_key_id': 'ac-Idr.....YwflZBaaxI',
        'aws_secret_access_key': '500f7b....b0f302e9'
    }
}
```

Please make sure these credentials have proper SQS access.

Amazon SQS only supports a bulk setting between 1 and 10, with the total payload not exceeding 256kb.

1.2.18 bulk

Sets the number of messages each cluster tries to get from the broker per call. Setting this on supported brokers can improve performance. Especially HTTP based or very high latency servers can benefit from bulk dequeue. Keep in mind however that settings this too high can degrade performance with multiple clusters or very large task packages.

Not supported by the default Redis broker. Defaults to 1.

1.2.19 cache

For some brokers, you will need to set up the Django [cache framework](#) to gather statistics for the monitor. You can indicate which cache to use by setting this value. Defaults to `default`.

1.2.20 `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.

Psutil does not support cpu affinity on OS X at this time.

1.3 Brokers

The broker sits between your Django instances and your Django Q cluster instances; accepting, saving and delivering task packages. Currently we support a variety of brokers from the default Redis, bleeding edge Disque to the convenient Amazon SQS.

The default Redis broker does not support message receipts. This means that in case of a catastrophic failure of the cluster server or worker timeouts, tasks that were being executed get lost. Keep in mind this is not the same as a failing task. If a tasks code crashes, this should only lead to a failed task status.

Even though this might be acceptable in some use cases, you might prefer brokers with message receipts support. These guarantee delivery by waiting for the cluster to send a receipt after the task has been processed. In case a receipt has not been received after a set time, the task package is put back in the queue. Django Q supports this behavior by setting the *retry* timer on brokers that support message receipts.

Some pointers:

- Don't set the *retry* timer to a lower or equal number than the task timeout.
- Retry time includes time the task spends waiting in the clusters internal queue.
- Don't set the *queue_limit* so high that tasks time out while waiting to be processed.
- In case a task is worked on twice, you will see a duplicate key error in the cluster logs.
- Duplicate tasks do generate additional receipt messages, but the result is discarded in favor of the first result.

Support for more brokers is being worked on.

1.3.1 Redis

The default broker for Django Q clusters.

- Atomic
- Requires *Redis-py* client library: `pip install redis`
- Does not need cache framework for monitoring
- Does not support receipts
- Can use existing *django_redis* connections.
- Configure with *redis-py* compatible configuration

1.3.2 Disque

Unlike Redis, Disque supports message receipts which make delivery to the cluster workers guaranteed. In our tests it is as fast or faster than the Redis broker. You can control the amount of time Disque should wait for completion of a task by configuring the *retry* setting. Bulk task retrieval is supported via the *bulk* option.

- Delivery receipts
- Atomic
- Needs Django's *Cache framework* configured for monitoring
- Compatible with *Tynd* Disque addon on *Heroku*
- Still considered Alpha software
- Supports bulk dequeue
- Requires *Redis-py* client library: `pip install redis`
- See the *disque_nodes* configuration section for more info.

1.3.3 IronMQ

This HTTP based queue service is both available directly via *Iron.io* and as an add-on on Heroku.

- Delivery receipts
- Supports bulk dequeue
- Needs Django's *Cache framework* configured for monitoring
- Requires the *iron-mq* client library: `pip install iron-mq`

- See the *iron_mq* configuration section for options.

1.3.4 Amazon SQS

Amazon's Simple Queue Service is another HTTP based message queue. Although **SQS** is not the fastest, it is stable, cheap and convenient if you already use AWS.

- Delivery receipts
- Maximum message size is 256Kb
- Supports bulk dequeue up to 10 messages with a maximum total size of 256Kb
- Needs Django's **Cache framework** configured for monitoring
- Requires the **boto3** client library: `pip install boto3`
- See the *sqs* configuration section for options.

1.3.5 Reference

The *Broker* class is used internally to communicate with the different types of brokers. You can override this class if you want to contribute and support your own broker.

class Broker

enqueue (*task*)

Sends a task package to the broker queue and returns a tracking id.

dequeue ()

Gets a task package from the broker and returns a tuple with a tracking id and the package.

acknowledge (*id*)

Notifies the broker that the task has been processed. Only works with brokers that support delivery receipts.

fail (*id*)

Tells the broker that the message failed to be processed by the cluster. Only available on brokers that support this. Currently only occurs when a cluster fails to unpack a task package.

delete (*id*)

Instructs the broker to delete this message from the queue.

purge_queue ()

Empties the current queue of all messages.

delete_queue ()

Deletes the current queue from the broker.

queue_size ()

Returns the amount of messages in the brokers queue.

ping ()

Returns True if the broker can be reached.

info ()

Shows the name and version of the currently configured broker.

1.4 Tasks

1.4.1 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. Can also be forced globally via the *sync* configuration option.

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 note that this will override any other option keywords.

1.4.2 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.

1.4.3 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. Instead of setting `sync` on each individual `async` you can also configure `sync` as a global override.

1.4.4 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.

1.4.5 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

queue_size ()

Returns the size of the broker queue. Note that this does not count tasks currently being processed.

Returns The amount of task packages in the broker

Return type *int*

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`.

1.5 Schedules

1.5.1 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)

# Run a schedule every 5 minutes, starting at 6 today
# for 2 hours
import arrow

schedule('math.hypot',
        3, 4,
        schedule_type=Schedule.MINUTES,
        minutes=5,
        repeats=24,
        next_run=arrow.utcnow().replace(hour=18, minute=0))
```

1.5.2 Missed schedules

If your cluster has not run for a while, the default behavior for the scheduler is to play catch up with the schedules and keep executing them until they are up to date. For example, if your cluster was down for a day and you start it again. Any hourly schedules you had will run 24 times, once every schedule loop, until they are back in sync. This behavior is intended to facilitate schedules that poll or gather statistics, but might not be suitable to your particular situation. You can change this by setting the *catch_up* configuration setting to `False`. The scheduler will then skip execution of scheduled events in the past. Instead those tasks will run only once and normal scheduling resumes.

1.5.3 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')
```

1.5.4 Reference

schedule (*func*, **args*, *name=None*, *hook=None*, *schedule_type='O'*, *minutes=None*, *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, M(I)nutes, (H)ourly, (D)aily, (W)eekly, (M)onthly, (Q)uarterly, (Y)early or *Schedule.TYPE*
- **minutes** (*int*) – Number of minutes for the Minutes 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, MINUTES, HOURLY, DAILY, WEEKLY, MONTHLY, QUARTERLY, YEARLY

minutes

The number of minutes the *MINUTES* schedule should use. Is ignored for other schedule types.

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.

MINUTES

'T' will run every *minutes* after its first run.

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.

1.6 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.

1.6.1 Multiple Clusters

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

- They connect to the same [broker](#).
- They use the same cluster name. See [Configuration](#)
- They share the same `SECRET_KEY` for Django.

1.6.2 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
```

1.6.3 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.

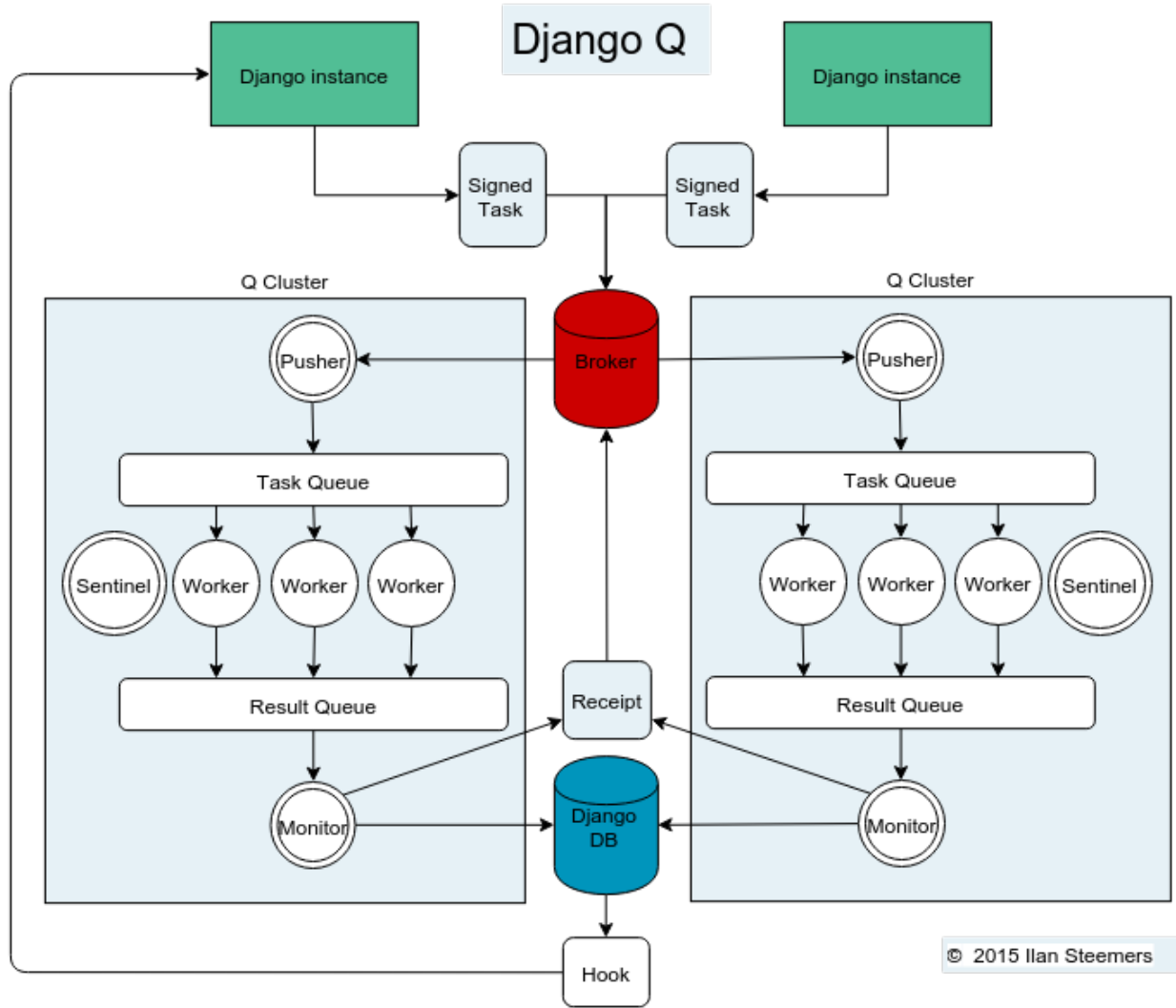
An example `circus.ini`

```
[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](#)

1.6.4 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 message broker. This ensures that task packages on the broker can only be executed and read by clusters and django servers who share the same secret key and cluster name. If a package fails to unpack, it will be marked failed with the broker and discarded. Optionally the packages can be compressed before transport.

Pusher

The pusher process continuously checks the broker 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. If the broker supports it, a delivery receipt is sent.

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*.

1.6.5 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

1.7 Monitor

The cluster monitor shows live 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

1.7.1 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.

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

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.

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.

1.7.2 Info

If you just want to see a one-off summary of your cluster stats you can use the *qinfo* management command:

```
$ python manage.py qinfo
```

-- djq summary --					
Clusters	1	Workers	4	Restarts	0
Queued	0	Successes	1000	Failures	2
Schedules	4	Tasks/hour	7.38	Avg time	0.1612

All stats are summed over all available clusters.

Task rate is calculated over the last 24 hours and will show the number of tasks per second, minute, hour or day depending on the amount. Average execution time (*Avg time*) is calculated in seconds over the last 24 hours.

Since some of these numbers are based on what is available in your tasks database, limiting or disabling the result backend will skew them.

1.7.3 Status

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

```
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)
```

1.7.4 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.

1.8 Admin pages

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

1.8.1 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 filter to filter your results by schedule name or group id. The table is searchable by *name*, *func* and *group*

Q [Search]							Filter
Action: [-----] Go 0 of 100 selected							By group
							All
							parzen
<input type="checkbox"/>	Name	Func	Started	Stopped	Time taken	Group	
<input type="checkbox"/>	mockingbird-west-speaker-carpet	<function parzen_estimation at 0x7ff1caac8378>	July 30, 2015, 7:37 p.m.	July 30, 2015, 7:37 p.m.	12.402957	parzen	
<input type="checkbox"/>	pizza-berlin-earth-romeo	<function parzen_estimation at 0x7ff1caac8378>	July 30, 2015, 7:37 p.m.	July 30, 2015, 7:37 p.m.	12.379033	parzen	
<input type="checkbox"/>	winner-nitrogen-vermont-sad	<function parzen_estimation at 0x7ff1caac8378>	July 30, 2015, 7:37 p.m.	July 30, 2015, 7:37 p.m.	12.371897	parzen	
<input type="checkbox"/>	neptune-early-orange-sodium	<function parzen_estimation at 0x7ff1caac8378>	July 30, 2015, 7:37 p.m.	July 30, 2015, 7:37 p.m.	12.371839	parzen	
<input type="checkbox"/>	stream-chicken-dakota-uranus	<function parzen_estimation at 0x7ff1caac8378>	July 30, 2015, 7:37 p.m.	July 30, 2015, 7:37 p.m.	12.283605	parzen	
<input type="checkbox"/>	avocado-blossom-bulldog-fifteen	<function parzen_estimation at 0x7ff1caac8378>	July 30, 2015, 7:37 p.m.	July 30, 2015, 7:37 p.m.	12.258351	parzen	
<input type="checkbox"/>	sierra-autumn-romeo-eight	<function parzen_estimation at 0x7ff1caac8378>	July 30, 2015, 7:37 p.m.	July 30, 2015, 7:37 p.m.	12.256816	parzen	
<input type="checkbox"/>	paris-uniform-fillet-georgia	<function parzen_estimation at 0x7ff1caac8378>	July 30, 2015, 7:37 p.m.	July 30, 2015, 7:37 p.m.	12.22891	parzen	

Uses the *Success* proxy model.

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

1.8.2 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

1.8.3 Scheduled tasks

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

Select Scheduled task to change

Action:

Go

0 of 3 selected

<div></div>	ID	Name	Func	Schedule Type	Repeats	Next Run	Last run	Success
<div></div>	2	Time test	djq.loadtest.time_test	Hourly	-71	Aug. 5, 2015, 5:02 a.m.	[beer-rugby-mars-delaware]	<div></div>
<div></div>	1	Math	math.copysign	Hourly	-82	Aug. 5, 2015, 5:12 a.m.	[pizza-item-lemon-asparagus]	<div></div>
<div></div>	3	Parzen	djq.tasks.parzen_async	Daily	-5	Aug. 5, 2015, 10:03 p.m.	[georgia-eleven-vegan-ceiling]	<div></div>

3 Scheduled tasks

Filter

By Next Run

Any date

Today

Past 7 days

This month

This year

By Schedule Type

All

Once

Hourly

Daily

Weekly

Monthly

Quarterly

Yearly

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.

You can pause a schedule by setting its repeats value to zero.

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

1.9 Examples

1.9.1 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.

1.9.2 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: {}'.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: {}'.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.

1.9.3 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, because it allows us to detach the sending task from the report generation function and to report on possible failures.

1.9.4 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.

1.9.5 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

`django_q`, 9

A

acknowledge() (Broker method), 11
args (Schedule attribute), 19
args (Task attribute), 16
async() (in module django_q), 14

B

Broker (built-in class), 11

C

Cluster (class in django_q), 25
cluster_id (Stat attribute), 28
count_group() (in module django_q), 16

D

DAILY (Schedule attribute), 20
delete() (Broker method), 11
delete_group() (django_q.Task class method), 17
delete_group() (in module django_q), 16
delete_queue() (Broker method), 11
dequeue() (Broker method), 11
django_q (module), 9
done_q_size (Stat attribute), 28

E

empty_queues() (Stat method), 28
enqueue() (Broker method), 11

F

fail() (Broker method), 11
Failure (class in django_q), 17
fetch() (in module django_q), 15
fetch_group() (in module django_q), 15
func (Schedule attribute), 19
func (Task attribute), 16

G

get() (Stat class method), 28
get_all() (Stat class method), 28
get_group_count() (django_q.Task class method), 17

get_result() (django_q.Task class method), 17
get_result_group() (django_q.Task class method), 17
get_task() (django_q.Task class method), 17
get_task_group() (django_q.Task class method), 17

H

has_stopped (Cluster attribute), 25
hook (Schedule attribute), 19
hook (Task attribute), 16
host (Cluster attribute), 25
HOURLY (Schedule attribute), 20

I

id (Schedule attribute), 19
id (Task attribute), 16
info() (Broker method), 11
is_running (Cluster attribute), 25
is_starting (Cluster attribute), 25
is_stopping (Cluster attribute), 25

K

kwargs (Schedule attribute), 19
kwargs (Task attribute), 16

L

last_run() (Schedule method), 20

M

MINUTES (Schedule attribute), 20
minutes (Schedule attribute), 19
monitor (Stat attribute), 28
MONTHLY (Schedule attribute), 20

N

name (Schedule attribute), 19
name (Task attribute), 16
next_run (Schedule attribute), 20

O

ONCE (Schedule attribute), 20

P

pid (Cluster attribute), 25
ping() (Broker method), 11
purge_queue() (Broker method), 11
pusher (Stat attribute), 28

Q

QUARTERLY (Schedule attribute), 20
queue_size() (Broker method), 11
queue_size() (in module django_q), 15

R

reincarnations (Stat attribute), 28
repeats (Schedule attribute), 19
result (Task attribute), 16
result() (in module django_q), 15
result_group() (in module django_q), 15

S

Schedule (class in django_q), 19
schedule() (in module django_q), 19
schedule_type (Schedule attribute), 19
sentinel (Cluster attribute), 25
sentinel (Stat attribute), 28
start() (Cluster method), 25
start_event (Cluster attribute), 25
started (Task attribute), 16
Stat (built-in class), 28
stat() (Cluster method), 25
status (Stat attribute), 28
stop() (Cluster method), 25
stop_event (Cluster attribute), 25
stopped (Task attribute), 17
Success (class in django_q), 17
success (Task attribute), 17
success() (Schedule method), 20

T

Task (class in django_q), 16
task (Schedule attribute), 20
task_q_size (Stat attribute), 28
time_taken() (Task method), 17
timeout (Cluster attribute), 25
tob (Stat attribute), 28
TYPE (Schedule attribute), 19

U

uptime() (Stat method), 28

W

WEEKLY (Schedule attribute), 20
workers (Stat attribute), 28

Y

YEARLY (Schedule attribute), 20