Django Q Documentation

Release 0.8.0

Ilan Steemers

Apr 05, 2017

Contents

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

CHAPTER 1

Features

- Multiprocessing worker pools
- Asynchronous tasks
- Scheduled and repeated tasks
- Encrypted and compressed packages
- Failure and success database or cache
- Result hooks, groups and chains
- Django Admin integration
- PaaS compatible with multiple instances
- Multi cluster monitor
- Redis, Disque, IronMQ, SQS, MongoDB or ORM
- Rollbar support

Django Q is tested with: Python 2.7 & 3.6. Django 1.8.18 LTS, 1.10.7 and 1.11

Contents:

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.
- Run Django Q cluster in order to handle tasks async:

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

Requirements

Django Q is tested for Python 2.7 and 3.6

- Django

  Django Q aims to use as much of Django's standard offerings as possible The code is tested against Django versions *1.8.18 LTS*, *1.10.7* and *1.11*.

- 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
```

- Pymongo is needed if you want to use MongoDB as a message broker:

```
$ pip install pymongo
```

- 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.
- MongoDB is a highly scalable NoSQL database which makes for a very fast and reliably persistent at-least-once message broker. Usually available on most PaaS providers.
- Pyrollbar is an error notifier for Rollbar which lets you manage your worker errors in one place. Needs a Rollbar account and access key:

```
$ pip install rollbar
```

Compatibility

Django Q is still a young project. If you do find any incompatibilities please submit an issue on github.

OS X

Running Django Q on OS X should work fine, except for the following known issues:

- `multiprocessing.Queue.qsize()` is not supported. This leads to the monitor not reporting the internal queue size of clusters running under OS X.
- CPU count through `multiprocessing.cpu_count()` does not work. Installing *psutil* provides Django Q with an alternative way of determining the number of CPU's on your system
- CPU affinity is provided by *psutil* which at this time does not support this feature on OSX. The code however is aware of this and will fake the CPU affinity assignment in the logs without actually assigning it. This way you can still develop with this setting.

Windows

The cluster and worker multiprocessing code depend on the OS's ability to fork, unfortunately forking is not supported under windows. You should however be able to develop and test without the cluster by setting the `sync` option to `True` in the configuration. This will run all `async` calls inline through a single cluster worker without the need for forking. Other known issues are:

- `os.getppid()` is only supported under windows since Python 3.2. If you use an older version you need to install *psutil* as an alternative.
- CPU count through `multiprocessing.cpu_count()` occasionally fails on servers. Installing *psutil* provides Django Q with an alternative way of determining the number of CPU's on your system
- The monitor and info commands rely on the Curses package which is not officially supported on windows. There are however some ports available like this one by Christoph Gohlke.

Python

The code is always tested against the latest version of Python 2 and Python 3 and we try to stay compatible with the last two versions of each. Current tests are performed with Python 2.7.12 and 3.6.1 If you do encounter any regressions with earlier versions, please submit an issue on github

Note: Django 1.7.10 or earlier is not compatible with Python 3.5 Django releases before 1.11 are not officially supported on Python 3.6

Open-source packages

Django Q is always tested with the latest versions of the required and optional Python packages. We try to keep the dependencies as up to date as possible. You can reference the requirements file to determine which versions are currently being used for tests and development.

Django

We strive to be compatible with last two major version of Django. At the moment this means we support the 1.8.18 LTS, 1.10.7 and 1.11 releases.

You might find that Django Q still works fine with Django 1.7 and 1.9, but new releases are no longer tested for it.

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:

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.

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]

daemonize_workers

Set the daemon flag when spawning workers. You may need to disable this flag if your worker needs to spawn child process but be carefull with orphaned child processes in case of sudden termination of the main process. Defaults to `True`.

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.

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.

compress

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

save_limit

Limits the amount of successful tasks saved to Django.

- Set to `0` for unlimited.
- Set to `-1` for no success storage at all.
- Defaults to `250`
- Failures are always saved.

[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.

guard_cycle

Guard loop sleep in seconds, must be greater than 0 and less than 60.

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`.

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`.

label

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

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`.

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

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.

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']
}
```

disque_auth

Optional Disque password for servers that require authentication.

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

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.

orm

If you want to use Django's database backend as a message broker, set the `orm` keyword to the database connection you want it to use:

```
# example ORM broker connection

Q_CLUSTER = {
    'name': 'DjangORM',
    'workers': 4,
    'timeout': 90,
    'retry': 120,
    'queue_limit': 50,
    'bulk': 10,
    'orm': 'default'
}
```

Using the Django ORM backend will also enable the Queued Tasks table in the Admin.

If you need better performance , you should consider using a different database backend than the main project. Set `orm` to the name of that database connection and make sure you run migrations on it using the `--database` option.

mongo

To use MongoDB as a message broker you simply provide the connection information in a dictionary:

```
# example MongoDB broker connection

Q_CLUSTER = {
    'name': 'MongoDB',
    'workers': 8,
    'timeout': 60,
    'retry': 70,
    'queue_limit': 100,
    'mongo': {
            'host': '127.0.0.1',
            'port': 27017
    }
}
```

The `mongo` dictionary can contain any of the parameters exposed by pymongo's MongoClient If you want to use a mongodb uri, you can supply it as the `host` parameter.

mongo_db

When using the MongoDB broker you can optionally provide a database name to use for the queues. Defaults to default database if available, otherwise `django-q`

broker_class

You can use a custom broker class for your cluster workers:

```
# example Custom broker class connection

Q_CLUSTER = {
    'name': 'Custom',
    'workers': 8,
    'timeout': 60,
    'broker_class: 'myapp.broker.CustomBroker'
}
```

Make sure your `CustomBroker` class inherits from either the base *`Broker`* class or one of its children.

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`.

poll

Sets the queue polling interval for database brokers that don't have a blocking call. Currently only affects the ORM and MongoDB brokers. Defaults to `0.2` (seconds).

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`.

cached

Switches all task and result functions from using the database backend to the cache backend. This is the same as setting the keyword `cached=True` on all task functions. Instead of a bool this can also be set to the number of seconds you want the cache to retain results. e.g. `cached=60`

scheduler

You can disable the scheduler by setting this option to `False`. This will reduce a little overhead if you're not using schedules, but is most useful if you want to temporarily disable all schedules. Defaults to `True`

rollbar

You can redirect worker exceptions directly to your Rollbar dashboard by installing the python notifier with `pip install rollbar` and adding this configuration dictionary to your config:

```
# rollbar config
Q_CLUSTER = {
    'rollbar': {
        'access_token': '32we33a92a5224jiww8982',
        'environment': 'Django-Q'
    }
}
```

Please check the Pyrollbar configuration reference for more options. Note that you will need a Rollbar account and access token to use this feature.

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.

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 ORM and fast MongoDB.

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, the task result will be updated with the latest results.
- In some rare cases a non-atomic broker will re-queue a task after it has been acknowledged.
- If a task runs twice and a previous run has succeeded, the new result wil be discarded.

- Limiting the number of retries is handled globally in your actual broker's settings.

Support for more brokers is being worked on.

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

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.

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.

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.

MongoDB

This highly scalable NoSQL database makes for a very fast and reliably persistent at-least-once message broker. Usually available on most PaaS providers, as open-source or commercial enterprise edition.

- Delivery receipts
- Needs Django's Cache framework configured for monitoring
- Can be configured as the Django cache-backend through several open-source cache providers.
- Requires the pymongo driver: `pip install pymongo`
- See the *mongo* configuration section for options.

Django ORM

Select this to use Django's database backend as a message broker. Unless you have configured a dedicated database backend for it, this should probably not be your first choice for a high traffic setup. However for a medium message rate and scheduled tasks, this is the most convenient guaranteed delivery broker.

- Delivery receipts
- Supports bulk dequeue
- Needs Django's Cache framework configured for monitoring
- Can be configured as its own cache backend.
- Queue editable in Django Admin
- See the *orm* configuration on how to set it up.

Custom Broker

You can override the *Broker* or any of its existing derived broker types.

```
# example Custom broker.py
from django_q.brokers import Broker

class CustomBroker(Broker):
    def info(self):
        return 'My Custom Broker'
```

Using the *broker_class* configuration setting you can then instruct Django Q to use this instead of one of the existing brokers:

```
# example Custom broker class connection

Q_CLUSTER = {
    'name': 'Custom',
    'workers': 8,
    'timeout': 60,
    'broker_class: 'myapp.broker.CustomBroker'
}
```

If you do write a custom broker for one of the many message queueing servers out there we don't support yet, please consider contributing it to the project.

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 if available.

dequeue()
Gets packages from the broker and returns a list of tuples 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.

lock_size()
Optional method that returns the number of messages currently awaiting acknowledgement. Only implemented on brokers that support it.

ping()
Returns True if the broker can be reached.

info()
Shows the name and version of the currently configured broker.

brokers.**get_broker**()
Returns a *Broker* instance based on the current configuration.

Tasks

async()

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

```
from django_q.tasks 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
# you can wait for it
task_result = result(task_id, 200)

# but 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.

cached

Redirects the result to the cache backend instead of the database if set to `True` or to an integer indicating the cache timeout in seconds. e.g. `cached=60`. Especially useful with large and group operations where you don't need the all results in your database and want to take advantage of the speed of your cache backend.

broker

A broker instance, in case you want to control your own connections.

task_name

Optionally overwrites the auto-generated task name.

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.

Note: For tasks to be processed you will need to have a worker cluster running in the background using `python manage.py qcluster` or you need to configure Django Q to run in synchronous mode for testing using the *sync* option.

Async

Optionally you can use the *Async* class to instantiate a task and keep everything in a single object.:

```
# Async class instance example
from django_q.tasks import Async

# instantiate an async task
a = Async('math.floor', 1.5, group='math')

# you can set or change keywords afterwards
a.cached = True
```

```
# run it
a.run()

# wait indefinitely for the result and print it
print(a.result(wait=-1))

# change the args
a.args = (2.5,)

# run it again
a.run()

# wait max 10 seconds for the result and print it

print(a.result(wait=10))
```

```
1
2
```

Once you change any of the parameters of the task after it has run, the result is invalidated and you will have to *Async.run()* it again to retrieve a new result.

Cached operations

You can run your tasks results against the Django cache backend instead of the database backend by either using the global *cached* setting or by supplying the `cached` keyword to individual functions. This can be useful if you are not interested in persistent results or if you run large group tasks where you only want the final result. By using a cache backend like Redis or Memcached you can speed up access to your task results significantly compared to a relational database.

When you set `cached=True`, results will be saved permanently in the cache and you will have to rely on your backend's cleanup strategies (like LRU) to manage stale results. You can also opt to set a manual timeout on the results, by setting e.g. `cached=60`. Meaning the result will be evicted from the cache after 60 seconds. This works both globally or on individual async executions.:

```
# simple cached example
from django_q.tasks import async, result

# cache the result for 10 seconds
id = async('math.floor', 100, cached=10)

# wait max 50ms for the result to appear in the cache
result(id, wait=50, cached=True)

# or fetch the task object
task = fetch(id, cached=True)

# and then save it to the database
task.save()
```

As you can see you can easily turn a cached result into a permanent database result by calling `save()` on it.

This also works for group actions:

```
# cached group example
from django_q.tasks import async, result_group
from django_q.brokers import get_broker

# set up a broker instance for better performance
broker = get_broker()

# async a hundred functions under a group label
for i in range(100):
    async('math.frexp',
          i,
          group='frexp',
          cached=True,
          broker=broker)

# wait max 50ms for one hundred results to return
result_group('frexp', wait=50, count=100, cached=True)
```

If you don't need hooks, that exact same result can be achieved by using the more convenient *async_iter()*.

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.tasks 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 broker and is intended for debugging and development. Instead of setting `sync` on each individual `async` you can also configure *sync* as a global override.

Connection pooling

Django Q tries to pass broker instances 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 broker to reuse for *async()*:

```
# broker connection economy example
from django_q.tasks import async
from django_q.brokers import get_broker

broker = get_broker()
```

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

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

Reference

async(*func*, **args*, *hook=None*, *group=None*, *timeout=None*, *save=None*, *sync=False*, *cached=False*, *broker=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
- **cached** – Output the result to the cache backend. Bool or timeout in seconds
- **broker** – Optional broker connection from *brokers.get_broker()*
- **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*, *wait=0*, *cached=False*)

Gets the result of a previously executed task

Parameters

- **task_id** (*str*) – the uuid or name of the task
- **wait** (*int*) – optional milliseconds to wait for a result. -1 for indefinite
- **cached** (*bool*) – run this against the cache backend.

Returns The result of the executed task

fetch(*task_id*, *wait=0*, *cached=False*)

Returns a previously executed task

Parameters

- **task_id** (*str*) – the uuid or name of the task
- **wait** (*int*) – optional milliseconds to wait for a result. -1 for indefinite
- **cached** (*bool*) – run this against the cache backend.

Returns A task object

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

delete_cached(*task_id*, *broker=None*)
Deletes a task from the cache backend

Parameters

- **task_id** (*str*) – the uuid of the task
- **broker** – an optional broker instance

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.

group_result(*failures=False*)

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

group_count(*failures=False*)

Returns a count of the number of task results in this task's group. Returns the number of failures when `failures=True`

group_delete(*tasks=False*)

Resets the group label on all the tasks in this task's group. If `tasks=True` it will also delete the tasks in this group from the database, including itself.

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`.

class Async(*func*, **args*, ***kwargs*)

A class wrapper for the *async()* function.

Parameters

- **func** (*object*) – The task function to execute
- **args** (*tuple*) – The arguments for the task function

- **kwargs** (*dict*) – Keyword arguments for the task function, including async options

id

The task unique identifier. This will only be available after it has been *run()*

started

Bool indicating if the task has been run with the current parameters

func

The task function to execute

args

A tuple of arguments for the task function

kwargs

Keyword arguments for the function. Can include any of the optional async keyword attributes directly or in a *q_options* dictionary.

broker

Optional *Broker* instance to use

sync

Run this task inline instead of asynchronous.

save

Overrides the global save setting.

hook

Optional function to call after a result is available. Takes the result *Task* as the first argument.

group

Optional group identifier

cached

Run the task against the cache result backend.

run()

Send the task to a worker cluster for execution

result(*wait=0*)

The task result. Always returns None if the task hasn't been run with the current parameters.

param int wait the number of milliseconds to wait for a result. -1 for indefinite

fetch(*wait=0*)

Returns the full *Task* result instance.

param int wait the number of milliseconds to wait for a result. -1 for indefinite

result_group(*failures=False*, *wait=0*, *count=None*)

Returns a list of results from this task's group.

param bool failures set this to `True` to include failed results

param int wait optional milliseconds to wait for a result or count. -1 for indefinite

param int count block until there are this many results in the group

fetch_group (*failures=True*, *wait=0*, *count=None*)

Returns a list of task results from this task's group

param bool failures set this to `False` to exclude failed tasks

param int wait optional milliseconds to wait for a task or count. -1 for indefinite

param int count block until there are this many tasks in the group

Groups

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

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

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

# wait until the group has 4 results
result = result_group('modf', count=4)
print(result)
```

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

Note that this particular example can be achieved much faster with *Iterable*

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.tasks 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: Calling `Queryset.values` for the result on Django 1.7 or lower will return a list of encoded results. If you can't upgrade to Django 1.8, use list comprehension or an iterator to return decoded results.

You can also access group functions from a task result instance:

```
from django_q.tasks import fetch

task = fetch('winter-speaker-alpha-ceiling')
if  task.group_count() > 100:
    print(task.group_result())
    task.group_delete()
    print('Deleted group {}'.format(task.group))
```

or call them directly on *Async* object:

```
from django_q.tasks import Async

# add a task to the math group and run it cached
a = Async('math.floor', 2.5, group='math', cached=True)

# wait until this tasks group has 10 results
result = a.result_group(count=10)
```

Reference

result_group(*group_id*, *failures=False*, *wait=0*, *count=None*, *cached=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
- **wait** (*int*) – optional milliseconds to wait for a result or count. -1 for indefinite
- **count** (*int*) – block until there are this many results in the group
- **cached** (*bool*) – run this against the cache backend

Returns a list of results

Return type list

fetch_group(*group_id*, *failures=True*, *wait=0*, *count=None*, *cached=False*)
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
- **wait** (*int*) – optional milliseconds to wait for a task or count. -1 for indefinite
- **count** (*int*) – block until there are this many tasks in the group
- **cached** (*bool*) – run this against the cache backend.

Returns a list of *Task*

Return type list

count_group(*group_id*, *failures=False*, *cached=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`
- **cached** (*bool*) – run this against the cache backend.

Returns the number of tasks or failures in a group

Return type int

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

Parameters

- **group_id** (*str*) – the group identifier
- **tasks** (*bool*) – also deletes the associated tasks if `True`
- **cached** (*bool*) – run this against the cache backend.

Returns the numbers of tasks affected

Return type int

Iterable

If you have an iterable object with arguments for a function, you can use *async_iter()* to async them with a single command:

```
# Async Iterable example
from django_q.tasks import async_iter, result

# set up a list of arguments for math.floor
iter = [i for i in range(100)]

# async iter them
id=async_iter('math.floor',iter)

# wait for the collated result for 1 second
result_list = result(id, wait=1000)
```

This will individually queue 100 tasks to the worker cluster, which will save their results in the cache backend for speed. Once all the 100 results are in the cache, they are collated into a list and saved as a single result in the database. The cache results are then cleared.

You can also use an *Iter* instance which can sometimes be more convenient:

```
from django_q.tasks import Iter

i = Iter('math.copysign')

# add some arguments
i.append(1, -1)
i.append(2, -1)
i.append(3, -1)

# run it
i.run()
```

```
# get the results
print(i.result())
```

```
[-1.0, -2.0, -3.0]
```

Reference

async_iter (*func*, *args_iter*, ***kwargs*)

Runs iterable arguments against the cache backend and returns a single collated result. Accepts the same options as *async()* except hook. See also the *Iter* class.

Parameters

- **func** (*object*) – The task function to execute
- **args** – An iterable containing arguments for the task function
- **kwargs** (*dict*) – Keyword arguments for the task function. Ignores hook.

Returns The uuid of the task

Return type str

class Iter (*func=None*, *args=None*, *kwargs=None*, *cached=Conf.CACHED*, *sync=Conf.SYNC*, *broker=None*)

An async task with iterable arguments. Serves as a convenient wrapper for *async_iter()* You can pass the iterable arguments at construction or you can append individual argument tuples.

param func the function to execute

param args an iterable of arguments.

param kwargs the keyword arguments

param bool cached run this against the cache backend

param bool sync execute this inline instead of asynchronous

param broker optional broker instance

append (**args*)

Append arguments to the iter set. Returns the current set count.

param args the arguments for a single execution

return the current set count

rtype int

run ()

Start queueing the tasks to the worker cluster.

return the task result id

result (*wait=0*)

return the full list of results.

param int wait how many milliseconds to wait for a result

return an unsorted list of results

fetch (*wait=0*)

get the task result objects.

param int wait how many milliseconds to wait for a result

return an unsorted list of task objects

length()

get the length of the arguments list

return int length of the argument list

Chains

Sometimes you want to run tasks sequentially. For that you can use the *async_chain()* function:

```
# Async a chain of tasks
from django_q.tasks import async_chain, result_group

# the chain must be in the format
# [(func,(args),{kwargs}),(func,(args),{kwargs}),..]
group_id = async_chain([('math.copysign', (1, -1)),
                        ('math.floor', (1,))])

# get group result
result_group(group_id, count=2)
```

A slightly more convenient way is to use a *Chain* instance:

```
# Chain async
from django_q.tasks import Chain

# create a chain that uses the cache backend
chain = Chain(cached=True)

# add some tasks
chain.append('math.copysign', 1, -1)
chain.append('math.floor', 1)

# run it
chain.run()

print(chain.result())
```

```
[-1.0, 1]
```

Reference

async_chain(*chain*, *group=None*, *cached=Conf.CACHED*, *sync=Conf.SYNC*, *broker=None*)

Async a chain of tasks. See also the *Chain* class.

Parameters

- **chain** (*list*) – a list of tasks in the format [(func,(args),{kwargs}), (func,(args),{kwargs})]
- **group** (*str*) – an optional group name.

- **cached** (*bool*) – run this against the cache backend
- **sync** (*bool*) – execute this inline instead of asynchronous

class Chain (*chain=None*, *group=None*, *cached=Conf.CACHED*, *sync=Conf.SYNC*)

A sequential chain of tasks. Acts as a convenient wrapper for *async_chain()* You can pass the task chain at construction or you can append individual tasks before running them.

param list chain a list of task in the format [(func,(args),{kwargs}), (func,(args),{kwargs})]

param str group an optional group name.

param bool cached run this against the cache backend

param bool sync execute this inline instead of asynchronous

append (*func*, **args*, ***kwargs*)

Append a task to the chain. Takes the same arguments as *async()*

return the current number of tasks in the chain

rtype int

run ()

Start queueing the chain to the worker cluster.

return the chains group id

result (*wait=0*)

return the full list of results from the chain when it finishes. Blocks until timeout or result.

param int wait how many milliseconds to wait for a result

return an unsorted list of results

fetch (*failures=True*, *wait=0*)

get the task result objects from the chain when it finishes. Blocks until timeout or result.

param failures include failed tasks

param int wait how many milliseconds to wait for a result

return an unsorted list of task objects

current ()

get the index of the currently executing chain element

return int current chain index

length ()

get the length of the chain

return int length of the chain

Schedules

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:

```
# Use the schedule wrapper
from django_q.tasks import schedule

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

# Or create the object directly
from django_q.models import Schedule

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))
```

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. In practical terms this means the scheduler will execute tasks in the past, reschedule them in the past and immediately execute them again until the schedule is set in the future. This default 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 once when the cluster starts again and the scheduler will find the next available slot in the future according to original schedule parameters.

Management Commands

If you want to schedule regular Django management commands, you can use the `django.core.management` module to call them directly:

```
from django_q.tasks import schedule

# run `manage.py clearsession` every hour
schedule('django.core.management.call_command',
         'clearsessions',
         schedule_type='H')
```

Or you can make a wrapper function which you can then 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.tasks import schedule

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

Check out the *Shell* examples if you want to schedule regular shell commands

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

'I' 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.

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.

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.

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
```

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*

An example `supervisor.conf`

```
[program:django-q]
command = python manage.py qcluster
stopasgroup = true
```

Supervisor's `stopasgroup` will ensure that the single process doesn't leave orphan process on stop or restart.

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`

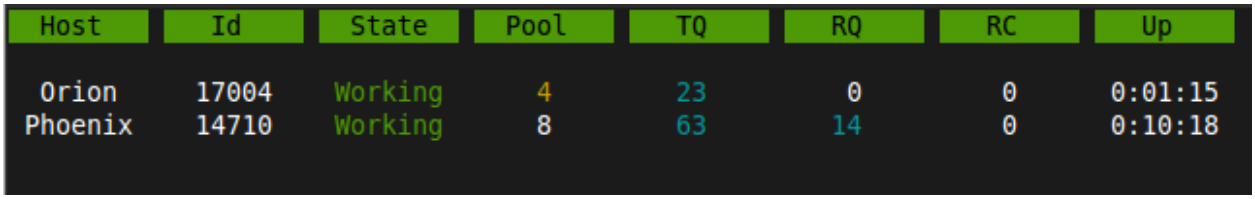
Bool. Tells you if the cluster has finished the stop procedure

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
```



For all broker types except the Redis broker, the monitor utilizes Django's cache framework to store statistics of running clusters. This can be any type of cache backend as long as it can be shared among Django instances. For this reason, the local memory backend will not work.

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[1]

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.

[1] 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

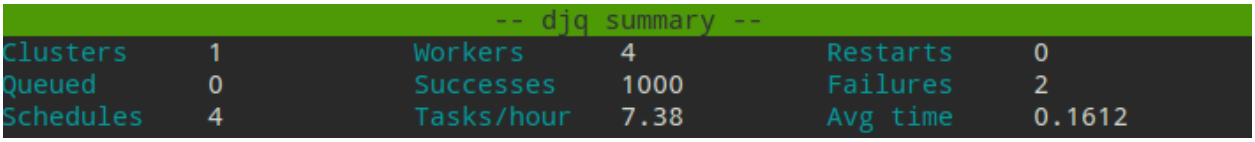
Uptime the amount of time that has passed since the cluster was started.

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

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
```



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.

Like with the monitor, these statistics come from a Redis server or Django's cache framework. So make sure you have either one configured.

To print out the current configuration run:

```
$ python manage.py qinfo --config
```

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)
```

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*, *broker=None*)

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

classmethod get_all(*broker=None*)

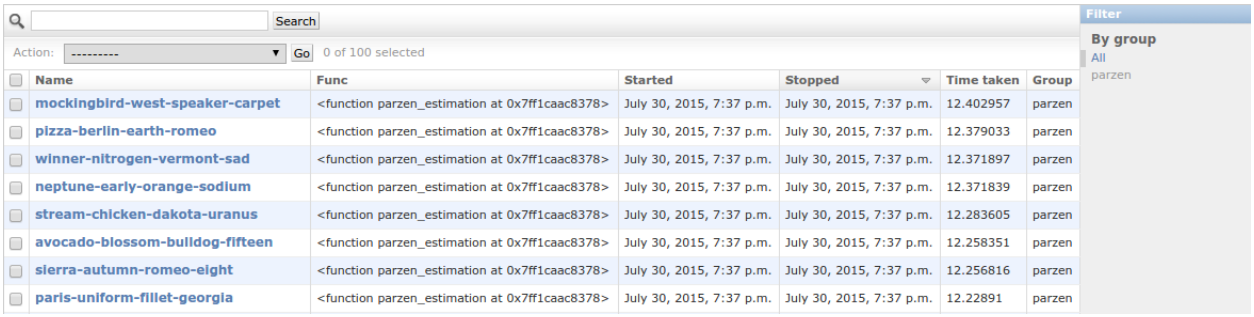
Returns a list of *Stat* objects for all active clusters. Takes an optional broker connection.

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:

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*



Uses the `Success` proxy model.

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

Failed tasks

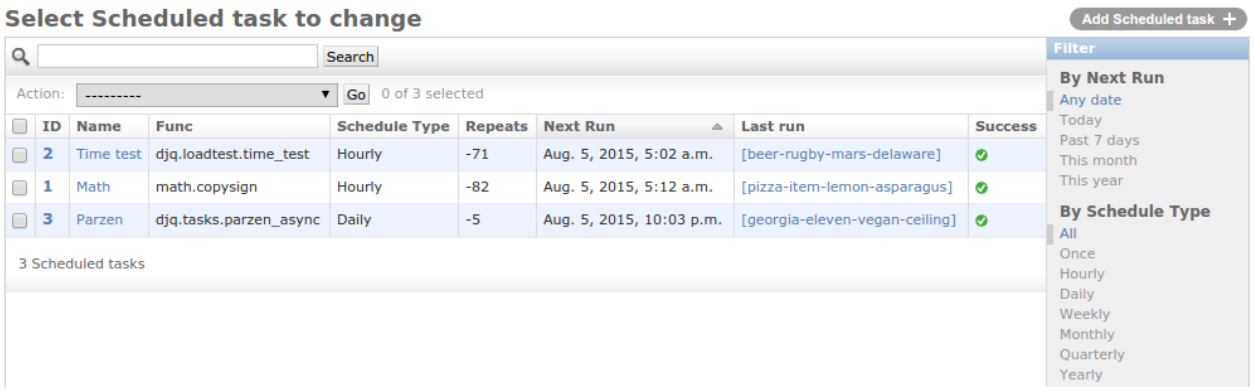
Failed tasks have encountered an error, preventing them from finishing execution. The worker will try to put the error in the *result* field of the task so you can review what happened.

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

Uses the `Failure` proxy model

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.

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

Queued tasks

This admin view is only enabled when you use the *Django ORM* broker. It shows all tasks packages currently in the broker queue. The `lock` column shows the moment at which this package was picked up by the cluster and is used to determine whether it has expired or not. For development purposes you can edit and delete queued tasks from here.

Signals

Available signals

Django Q emits the following signals during its lifecycle.

Before enqueuing a task

The `django_q.signals.pre_enqueue` signal is emitted before a task is enqueued. The task dictionary is given as the `task` argument.

Before executing a task

The `django_q.signals.pre_execute` signal is emitted before a task is executed by a worker. This signal provides two arguments:

- `task`: the task dictionary.
- `func`: the actual function that will be executed. If the task was created with a function path, this argument will be the callable function nonetheless.

Subscribing to a signal

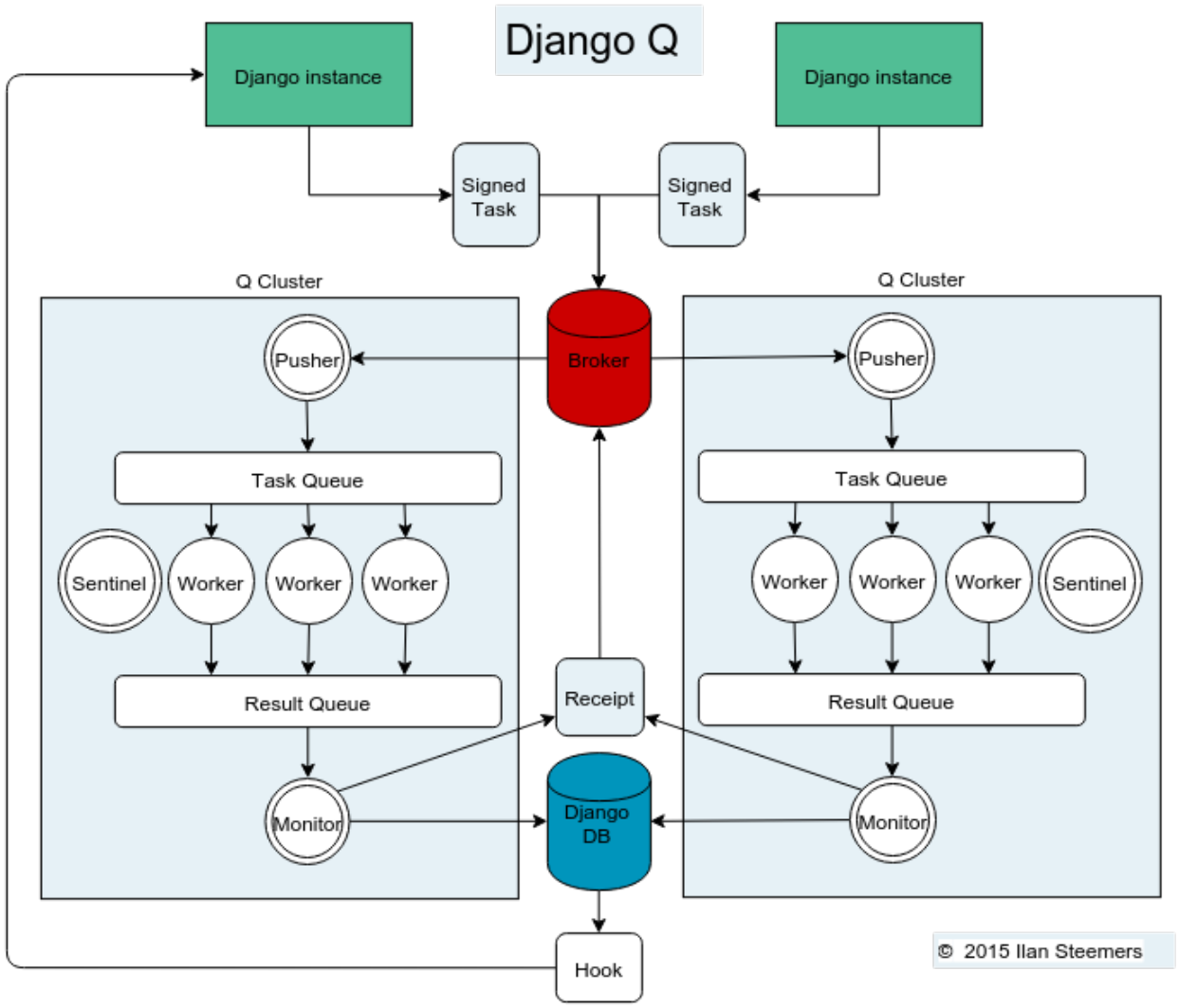
Connecting to a Django Q signal is done in the same manner as any other Django signal:

```
from django.dispatch import receiver
from django_q.signals import pre_enqueue, pre_execute

@receiver(pre_enqueue)
def my_pre_enqueue_callback(sender, task, **kwargs):
    print("Task {} will be enqueued".format(task["name"]))

@receiver(pre_execute)
def my_pre_execute_callback(sender, func, task, **kwargs):
    print("Task {} will be executed by calling {}".format(
          task["name"], func))
```

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.

Broker

The broker collects task packages from the django instances and queues them for pick up by a cluster. If the broker supports message receipts, it will keep a copy of the tasks around until a cluster acknowledges the processing of the task. Otherwise it is put back in the queue after a timeout period. This ensure at-least-once delivery. Note that even if the task errors when processed by the cluster, this is considered a successful delivery. Most failed deliveries will be the result of a worker or the cluster crashing before the task was saved.

Pusher

The pusher process continuously checks the broker for new task packages. It checks the signing and unpacks the task to the internal Task Queue. The amount of tasks in the Task Queue can be configured to control memory usage and minimize data loss in case of a failure.

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 or cache backend. If the broker supports it, a delivery receipt is sent. In case the task was part of a chain, the next task is queued.

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*.

Examples

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.tasks import async, schedule
from django_q.models import 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.

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.tasks 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.

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.tasks 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.tasks 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.

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.tasks 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.

Shell

You can execute or schedule shell commands using Pythons `subprocess` module:

```
from django_q.tasks import async, result

# make a backup copy of setup.py
async('subprocess.call', ['cp', 'setup.py', 'setup.py.bak'])

# call ls -l and dump the output
task_id=async('subprocess.check_output', ['ls', '-l'])

# get the result
dir_list = result(task_id)
```

In Python 3.5 the subprocess module has changed quite a bit and returns a `subprocess.CompletedProcess` object instead:

```
from django_q.tasks import async, result

# make a backup copy of setup.py
tid = async('subprocess.run', ['cp', 'setup.py', 'setup.py.bak'])

# get the result
r=result(tid, 500)
# we can now look at the original arguments
>>> r.args
['cp', 'setup.py', 'setup.py.bak']
# and the returncode
>>> r.returncode
0

# to capture the output we'll need a pipe
from subprocess import PIPE

# call ls -l and pipe the output
tid = async('subprocess.run', ['ls', '-l'], stdout=PIPE)
# get the result
res = result(tid, 500)
# print the output
print(res.stdout)
```

Instead of *async()* you can of course also use *schedule()* to schedule commands.

For regular Django management commands, it is easier to call them directly:

```
from django_q.tasks import async, schedule

async('django.core.management.call_command','clearsessions')

# or clear those sessions every hour

schedule('django.core.management.call_command',
     'clearsessions',
     schedule_type='H')
```

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.tasks import async, result_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 return the collated result
def parzen_async():
    # clear the previous results
    delete_group('parzen', cached=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 to the cache backend
    for w in widths:
        async(parzen_estimation, sample, x, w,
              group='parzen', cached=True)
    # return after 100 results
    return result_group('parzen', count=100, cached=True)
```

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

Alternatively the `parzen_async()` function can also be written with *async_iter()*, which automatically utilizes the cache backend and groups to return a single result from an iterable:

```
# create 100 calculations and return the collated result
def parzen_async():
    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 async iterable
    args = [(sample, x, w) for w in widths]
    result_id = async_iter(parzen_estimation, args, cached=True)
    # return the cached result or timeout after 10 seconds
    return result(result_id, wait=10000, cached=True)
```

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

- genindex
- search

Python Module Index

d

Index