



This document describes the current stable version of Celery (4.2). For development docs, [go here](#).

# Configuration and defaults

This document describes the configuration options available.

If you're using the default loader, you must create the `celeryconfig.py` module and make sure it's available on the Python path.

 Star 11,909

Please help support this community project with a donation:



## Previous topic

[Extensions and Bootsteps](#)

## Next topic

[Documenting Tasks with Sphinx](#)

## This Page

[Show Source](#)

## Quick search

 

Private repos and priority support

Try Read the Docs for Business Today!

*Sponsored · Ads served ethically*

- [Example configuration file](#)
- [New lowercase settings](#)
- [Configuration Directives](#)
  - [General settings](#)
  - [Time and date settings](#)
  - [Task settings](#)
  - [Task execution settings](#)
  - [Task result backend settings](#)
  - [Database backend settings](#)
  - [RPC backend settings](#)
  - [Cache backend settings](#)
  - [Redis backend settings](#)
  - [Cassandra backend settings](#)
  - [Elasticsearch backend settings](#)
  - [Riak backend settings](#)
  - [AWS DynamoDB backend settings](#)
  - [IronCache backend settings](#)
  - [Couchbase backend settings](#)
  - [CouchDB backend settings](#)
  - [File-system backend settings](#)
  - [Consul K/V store backend settings](#)
  - [Message Routing](#)
  - [Broker Settings](#)
  - [Worker](#)
  - [Events](#)
  - [Remote Control Commands](#)
  - [Logging](#)
  - [Security](#)
  - [Custom Component Classes \(advanced\)](#)
  - [Beat Settings \(\*\*celery beat\*\*\)](#)

## Example configuration file

This is an example configuration file to get you started. It should contain all you need to run a basic Celery set-up.

```
## Broker settings.
broker_url = 'amqp://guest:guest@localhost:5672//'

# List of modules to import when the Celery worker starts.
imports = ('myapp.tasks',)

## Using the database to store task state and results.
result_backend = 'db+sqlite:///results.db'

task_annotations = {'tasks.add': {'rate_limit': '10/s'}}
```

## New lowercase settings

Version 4.0 introduced new lower case settings and setting organization.

The major difference between previous versions, apart from the lower case names, are the renaming of some prefixes, like `celerybeat_` to `beat_`, `celeryd_` to `worker_`, and most of the top level `celery_` settings have been moved into a new `task_` prefix.

### Note:

Celery will still be able to read old configuration files, so there's no rush in moving to the new settings format. Furthermore, we provide the `celery upgrade` command that should handle plenty of cases (including [Django](#)).

Setting name	Replace with
CELERY_ACCEPT_CONTENT	<code>accept_content</code>
CELERY_ENABLE_UTC	<code>enable_utc</code>
CELERY_IMPORTS	<code>imports</code>
CELERY_INCLUDE	<code>include</code>
CELERY_TIMEZONE	<code>timezone</code>
CELERYBEAT_MAX_LOOP_INTERVAL	<code>beat_max_loop_interval</code>
CELERYBEAT_SCHEDULE	<code>beat_schedule</code>
CELERYBEAT_SCHEDULER	<code>beat_scheduler</code>
CELERYBEAT_SCHEDULE_FILENAME	<code>beat_schedule_filename</code>
CELERYBEAT_SYNC_EVERY	<code>beat_sync_every</code>
BROKER_URL	<code>broker_url</code>
BROKER_TRANSPORT	<code>broker_transport</code>
BROKER_TRANSPORT_OPTIONS	<code>broker_transport_options</code>
BROKER_CONNECTION_TIMEOUT	<code>broker_connection_timeout</code>
BROKER_CONNECTION_RETRY	<code>broker_connection_retry</code>
BROKER_CONNECTION_MAX_RETRIES	<code>broker_connection_max_retries</code>
BROKER_FAILOVER_STRATEGY	<code>broker_failover_strategy</code>
BROKER_HEARTBEAT	<code>broker_heartbeat</code>
BROKER_LOGIN_METHOD	<code>broker_login_method</code>
BROKER_POOL_LIMIT	<code>broker_pool_limit</code>
BROKER_USE_SSL	<code>broker_use_ssl</code>
CELERY_CACHE_BACKEND	<code>cache_backend</code>
CELERY_CACHE_BACKEND_OPTIONS	<code>cache_backend_options</code>
CASSANDRA_COLUMN_FAMILY	<code>cassandra_table</code>
CASSANDRA_ENTRY_TTL	<code>cassandra_entry_ttl</code>
CASSANDRA_KEYSPACE	<code>cassandra_keyspace</code>
CASSANDRA_PORT	<code>cassandra_port</code>
CASSANDRA_READ_CONSISTENCY	<code>cassandra_read_consistency</code>
CASSANDRA_SERVERS	<code>cassandra_servers</code>

Setting name	Replace with
CASSANDRA_WRITE_CONSISTENCY	<b>cassandra_write_consistency</b>
CASSANDRA_OPTIONS	<b>cassandra_options</b>
CELERY_COUCHBASE_BACKEND_SETTINGS	<b>couchbase_backend_settings</b>
CELERY_MONGODB_BACKEND_SETTINGS	<b>mongodb_backend_settings</b>
CELERY_EVENT_QUEUE_EXPIRES	<b>event_queue_expires</b>
CELERY_EVENT_QUEUE_TTL	<b>event_queue_ttl</b>
CELERY_EVENT_QUEUE_PREFIX	<b>event_queue_prefix</b>
CELERY_EVENT_SERIALIZER	<b>event_serializer</b>
CELERY_REDIS_DB	<b>redis_db</b>
CELERY_REDIS_HOST	<b>redis_host</b>
CELERY_REDIS_MAX_CONNECTIONS	<b>redis_max_connections</b>
CELERY_REDIS_PASSWORD	<b>redis_password</b>
CELERY_REDIS_PORT	<b>redis_port</b>
CELERY_RESULT_BACKEND	<b>result_backend</b>
CELERY_MAX_CACHED_RESULTS	<b>result_cache_max</b>
CELERY_MESSAGE_COMPRESSION	<b>result_compression</b>
CELERY_RESULT_EXCHANGE	<b>result_exchange</b>
CELERY_RESULT_EXCHANGE_TYPE	<b>result_exchange_type</b>
CELERY_TASK_RESULT_EXPIRES	<b>result_expires</b>
CELERY_RESULT_PERSISTENT	<b>result_persistent</b>
CELERY_RESULT_SERIALIZER	<b>result_serializer</b>
CELERY_RESULT_DBURI	Use <b>result_backend</b> instead.
CELERY_RESULT_ENGINE_OPTIONS	<b>database_engine_options</b>
[...]_DB_SHORT_LIVED_SESSIONS	<b>database_short_lived_sessions</b>
CELERY_RESULT_DB_TABLE_NAMES	<b>database_db_names</b>
CELERY_SECURITY_CERTIFICATE	<b>security_certificate</b>
CELERY_SECURITY_CERT_STORE	<b>security_cert_store</b>
CELERY_SECURITY_KEY	<b>security_key</b>
CELERY_ACKS_LATE	<b>task_acks_late</b>
CELERY_TASK_ALWAYS_EAGER	<b>task_always_eager</b>
CELERY_TASK_ANNOTATIONS	<b>task_annotations</b>
CELERY_TASK_COMPRESSION	<b>task_compression</b>
CELERY_TASK_CREATE_MISSING_QUEUES	<b>task_create_missing_queues</b>
CELERY_TASK_DEFAULT_DELIVERY_MODE	<b>task_default_delivery_mode</b>
CELERY_TASK_DEFAULT_EXCHANGE	<b>task_default_exchange</b>
CELERY_TASK_DEFAULT_EXCHANGE_TYPE	<b>task_default_exchange_type</b>
CELERY_TASK_DEFAULT_QUEUE	<b>task_default_queue</b>
CELERY_TASK_DEFAULT_RATE_LIMIT	<b>task_default_rate_limit</b>
CELERY_TASK_DEFAULT_ROUTING_KEY	<b>task_default_routing_key</b>
CELERY_TASK_EAGER_PROPAGATES	<b>task_eager_propagates</b>
CELERY_TASK_IGNORE_RESULT	<b>task_ignore_result</b>
CELERY_TASK_PUBLISH_RETRY	<b>task_publish_retry</b>
CELERY_TASK_PUBLISH_RETRY_POLICY	<b>task_publish_retry_policy</b>
CELERY_QUEUES	<b>task_queues</b>
CELERY_ROUTES	<b>task_routes</b>
CELERY_TASK_SEND_SENT_EVENT	<b>task_send_sent_event</b>
CELERY_TASK_SERIALIZER	<b>task_serializer</b>
CELERYD_TASK_SOFT_TIME_LIMIT	<b>task_soft_time_limit</b>
CELERYD_TASK_TIME_LIMIT	<b>task_time_limit</b>
CELERY_TRACK_STARTED	<b>task_track_started</b>
CELERYD_AGENT	<b>worker_agent</b>

Setting name	Replace with
CELERYD_AUTOSCALER	<b>worker_autoscaler</b>
CELERYD_CONCURRENCY	<b>worker_concurrency</b>
CELERYD_CONSUMER	<b>worker_consumer</b>
CELERY_WORKER_DIRECT	<b>worker_direct</b>
CELERY_DISABLE_RATE_LIMITS	<b>worker_disable_rate_limits</b>
CELERY_ENABLE_REMOTE_CONTROL	<b>worker_enable_remote_control</b>
CELERYD_HIJACK_ROOT_LOGGER	<b>worker_hijack_root_logger</b>
CELERYD_LOG_COLOR	<b>worker_log_color</b>
CELERYD_LOG_FORMAT	<b>worker_log_format</b>
CELERYD_WORKER_LOST_WAIT	<b>worker_lost_wait</b>
CELERYD_MAX_TASKS_PER_CHILD	<b>worker_max_tasks_per_child</b>
CELERYD_POOL	<b>worker_pool</b>
CELERYD_POOL_PUTLOCKS	<b>worker_pool_putlocks</b>
CELERYD_POOL_RESTARTS	<b>worker_pool_restarts</b>
CELERYD_PREFETCH_MULTIPLIER	<b>worker_prefetch_multiplier</b>
CELERYD_REDIRECT_STDOUTS	<b>worker_redirect_stdouts</b>
CELERYD_REDIRECT_STDOUTS_LEVEL	<b>worker_redirect_stdouts_level</b>
CELERYD_SEND_EVENTS	<b>worker_send_task_events</b>
CELERYD_STATE_DB	<b>worker_state_db</b>
CELERYD_TASK_LOG_FORMAT	<b>worker_task_log_format</b>
CELERYD_TIMER	<b>worker_timer</b>
CELERYD_TIMER_PRECISION	<b>worker_timer_precision</b>

## Configuration Directives

### General settings

#### `accept_content`

Default: `{ 'json' }` (set, list, or tuple).

A white-list of content-types/serializers to allow.

If a message is received that's not in this list then the message will be discarded with an error.

By default only json is enabled but any content type can be added, including pickle and yaml; when this is the case make sure untrusted parties don't have access to your broker. See [Security](#) for more.

Example:

```
# using serializer name
accept_content = ['json']

# or the actual content-type (MIME)
accept_content = ['application/json']
```

### Time and date settings

#### `enable_utc`

*New in version 2.5.*

Default: Enabled by default since version 3.0.

If enabled dates and times in messages will be converted to use the UTC timezone.

Note that workers running Celery versions below 2.5 will assume a local timezone for all messages, so only enable if all workers have been upgraded.

## timezone

*New in version 2.5.*

Default: "UTC".

Configure Celery to use a custom time zone. The timezone value can be any time zone supported by the `pytz` library.

If not set the UTC timezone is used. For backwards compatibility there's also a `enable_utc` setting, and when this is set to false the system local timezone is used instead.

## Task settings

### task\_annotations

*New in version 2.5.*

Default: **None**.

This setting can be used to rewrite any task attribute from the configuration. The setting can be a dict, or a list of annotation objects that filter for tasks and return a map of attributes to change.

This will change the `rate_limit` attribute for the `tasks.add` task:

```
task_annotations = {'tasks.add': {'rate_limit': '10/s'}}
```

or change the same for all tasks:

```
task_annotations = {'*': {'rate_limit': '10/s'}}
```

You can change methods too, for example the `on_failure` handler:

```
def my_on_failure(self, exc, task_id, args, kwargs, einfo):
    print('Oh no! Task failed: {0!r}'.format(exc))

task_annotations = {'*': {'on_failure': my_on_failure}}
```

If you need more flexibility then you can use objects instead of a dict to choose the tasks to annotate:

```
class MyAnnotate(object):

    def annotate(self, task):
        if task.name.startswith('tasks.'):
            return {'rate_limit': '10/s'}

task_annotations = (MyAnnotate(), {other,})
```

### task\_compression

Default: **None**

Default compression used for task messages. Can be `gzip`, `bzip2` (if available), or any custom compression schemes registered in the Kombu compression registry.

The default is to send uncompressed messages.

## `task_protocol`

Default: 2 (since 4.0).

Set the default task message protocol version used to send tasks. Supports protocols: 1 and 2.

Protocol 2 is supported by 3.1.24 and 4.x+.

## `task_serializer`

Default: `"json"` (since 4.0, earlier: `pickle`).

A string identifying the default serialization method to use. Can be `json` (default), `pickle`, `yaml`, `msgpack`, or any custom serialization methods that have been registered with `kombu.serialization.registry`.

## See also:

[Serializers](#).

## `task_publish_retry`

*New in version 2.2.*

Default: Enabled.

Decides if publishing task messages will be retried in the case of connection loss or other connection errors. See also [task\\_publish\\_retry\\_policy](#).

## `task_publish_retry_policy`

*New in version 2.2.*

Default: See [Message Sending Retry](#).

Defines the default policy when retrying publishing a task message in the case of connection loss or other connection errors.

# Task execution settings

## `task_always_eager`

Default: Disabled.

If this is **True**, all tasks will be executed locally by blocking until the task returns. `apply_async()` and `Task.delay()` will return an **EagerResult** instance, that emulates the API and behavior of [AsyncResult](#), except the result is already evaluated.

That is, tasks will be executed locally instead of being sent to the queue.

## task\_eager\_propagates

Default: Disabled.

If this is **True**, eagerly executed tasks (applied by `task.apply()`, or when the `task_always_eager` setting is enabled), will propagate exceptions.

It's the same as always running `apply()` with `throw=True`.

## task\_remote\_tracebacks

Default: Disabled.

If enabled task results will include the workers stack when re-raising task errors.

This requires the `tblib` library, that can be installed using `pip`:

```
$ pip install celery[tblib]
```

See [Bundles](#) for information on combining multiple extension requirements.

## task\_ignore\_result

Default: Disabled.

Whether to store the task return values or not (tombstones). If you still want to store errors, just not successful return values, you can set

`task_store_errors_even_if_ignored`.

## task\_store\_errors\_even\_if\_ignored

Default: Disabled.

If set, the worker stores all task errors in the result store even if `Task.ignore_result` is on.

## task\_track\_started

Default: Disabled.

If **True** the task will report its status as 'started' when the task is executed by a worker. The default value is **False** as the normal behavior is to not report that level of granularity. Tasks are either pending, finished, or waiting to be retried. Having a 'started' state can be useful for when there are long running tasks and there's a need to report what task is currently running.

## task\_time\_limit

Default: No time limit.

Task hard time limit in seconds. The worker processing the task will be killed and replaced with a new one when this is exceeded.

## task\_soft\_time\_limit

Default: No soft time limit.

Task soft time limit in seconds.

The `SoftTimeLimitExceeded` exception will be raised when this is exceeded. For

example, the task can catch this to clean up before the hard time limit comes:

```
from celery.exceptions import SoftTimeLimitExceeded

@app.task
def mytask():
    try:
        return do_work()
    except SoftTimeLimitExceeded:
        cleanup_in_a_hurry()
```

## task\_acks\_late

Default: Disabled.

Late ack means the task messages will be acknowledged **after** the task has been executed, not *just before* (the default behavior).

### See also:

FAQ: [Should I use retry or acks\\_late?](#).

## task\_reject\_on\_worker\_lost

Default: Disabled.

Even if [task\\_acks\\_late](#) is enabled, the worker will acknowledge tasks when the worker process executing them abruptly exits or is signaled (e.g., **KILL/INT**, etc).

Setting this to true allows the message to be re-queued instead, so that the task will execute again by the same worker, or another worker.

### Warning:

Enabling this can cause message loops; make sure you know what you're doing.

## task\_default\_rate\_limit

Default: No rate limit.

The global default rate limit for tasks.

This value is used for tasks that doesn't have a custom rate limit

### See also:

The setting;[worker\\_disable\\_rate\\_limits](#) setting can disable all rate limits.

## Task result backend settings

### result\_backend

Default: No result backend enabled by default.

The backend used to store task results (tombstones). Can be one of the following:

- `rpc`

Send results back as AMQP messages See [RPC backend settings](#).



- **database**  
Use a relational database supported by [SQLAlchemy](#). See [Database backend settings](#).
- **redis**  
Use [Redis](#) to store the results. See [Redis backend settings](#).
- **cache**  
Use [Memcached](#) to store the results. See [Cache backend settings](#).
- **cassandra**  
Use [Cassandra](#) to store the results. See [Cassandra backend settings](#).
- **elasticsearch**  
Use [Elasticsearch](#) to store the results. See [Elasticsearch backend settings](#).
- **ironcache**  
Use [IronCache](#) to store the results. See [IronCache backend settings](#).
- **couchbase**  
Use [Couchbase](#) to store the results. See [Couchbase backend settings](#).
- **couchdb**  
Use [CouchDB](#) to store the results. See [CouchDB backend settings](#).
- **filesystem**  
Use a shared directory to store the results. See [File-system backend settings](#).
- **consul**  
Use the [Consul](#) K/V store to store the results See [Consul K/V store backend settings](#).

## result\_backend\_transport\_options

Default: {} (empty mapping).

A dict of additional options passed to the underlying transport.

See your transport user manual for supported options (if any).

Example setting the visibility timeout (supported by Redis and SQS transports):

```
result_backend_transport_options = {'visibility_timeout': 18000} # 5 ho
```

## result\_serializer

Default: `json` since 4.0 (earlier: `pickle`).

Result serialization format.

See [Serializers](#) for information about supported serialization formats.

## result\_compression

Default: No compression.

Optional compression method used for task results. Supports the same options as the [task\\_serializer](#) setting.

## result\_expires

Default: Expire after 1 day.

Time (in seconds, or a `timedelta` object) for when after stored task tombstones will be deleted.

A built-in periodic task will delete the results after this time (`celery.backend_cleanup`), assuming that `celery beat` is enabled. The task runs daily at 4am.

A value of **None** or 0 means results will never expire (depending on backend specifications).

## Note:

For the moment this only works with the AMQP, database, cache, and Redis backends.

When using the database backend, `celery beat` must be running for the results to be expired.

## `result_cache_max`

Default: Disabled by default.

Enables client caching of results.

This can be useful for the old deprecated 'amqp' backend where the result is unavailable as soon as one result instance consumes it.

This is the total number of results to cache before older results are evicted. A value of 0 or None means no limit, and a value of **-1** will disable the cache.

Disabled by default.

# Database backend settings

## Database URL Examples

To use the database backend you have to configure the `result_backend` setting with a connection URL and the `db+` prefix:

```
result_backend = 'db+scheme://user:password@host:port/dbname'
```

Examples:

```
# sqlite (filename)
result_backend = 'db+sqlite:///results.sqlite'

# mysql
result_backend = 'db+mysql://scott:tiger@localhost/foo'

# postgresql
result_backend = 'db+postgresql://scott:tiger@localhost/mydatabase'

# oracle
result_backend = 'db+oracle://scott:tiger@127.0.0.1:1521/sidname'
```

Please see [Supported Databases](#) for a table of supported databases, and [Connection String](#) for more information about connection strings (this is the part of the URI that comes after the `db+` prefix).

## database\_engine\_options

Default: {} (empty mapping).

To specify additional SQLAlchemy database engine options you can use the **sqlalchemy\_engine\_options** setting:

```
# echo enables verbose logging from SQLAlchemy.  
app.conf.database_engine_options = {'echo': True}
```

## database\_short\_lived\_sessions

Default: Disabled by default.

Short lived sessions are disabled by default. If enabled they can drastically reduce performance, especially on systems processing lots of tasks. This option is useful on low-traffic workers that experience errors as a result of cached database connections going stale through inactivity. For example, intermittent errors like (*OperationalError*) (2006, 'MySQL server has gone away') can be fixed by enabling short lived sessions. This option only affects the database backend.

## database\_table\_names

Default: {} (empty mapping).

When SQLAlchemy is configured as the result backend, Celery automatically creates two tables to store result meta-data for tasks. This setting allows you to customize the table names:

```
# use custom table names for the database result backend.  
database_table_names = {  
    'task': 'myapp_taskmeta',  
    'group': 'myapp_groupmeta',  
}
```

## RPC backend settings

### result\_persistent

Default: Disabled by default (transient messages).

If set to **True**, result messages will be persistent. This means the messages won't be lost after a broker restart.

### Example configuration

```
result_backend = 'rpc://'  
result_persistent = False
```

## Cache backend settings

### Note:

The cache backend supports the **pylibmc** and **python-memcached** libraries. The latter is used only if **pylibmc** isn't installed.

Using a single Memcached server:

```
result_backend = 'cache+memcached://127.0.0.1:11211/'
```

Using multiple Memcached servers:

```
result_backend = """
cache+memcached://172.19.26.240:11211;172.19.26.242:11211/
""".strip()
```

The “memory” backend stores the cache in memory only:

```
result_backend = 'cache'
cache_backend = 'memory'
```

## cache\_backend\_options

Default: {} (empty mapping).

You can set `pylibmc` options using the `cache_backend_options` setting:

```
cache_backend_options = {
    'binary': True,
    'behaviors': {'tcp_nodelay': True},
}
```

## cache\_backend

This setting is no longer used as it’s now possible to specify the cache backend directly in the `result_backend` setting.

# Redis backend settings

## Configuring the backend URL

### Note:

The Redis backend requires the `redis` library.

To install this package use `pip`:

```
$ pip install celery[redis]
```

See [Bundles](#) for information on combining multiple extension requirements.

This backend requires the `result_backend` setting to be set to a Redis or [Redis over TLS](#) URL:

```
result_backend = 'redis://:password@host:port/db'
```

For example:

```
result_backend = 'redis://localhost/0'
```

is the same as:

```
result_backend = 'redis://'
```

Use the `rediss://` protocol to connect to redis over TLS:

```
result_backend = 'rediss://:password@host:port/db?ssl_cert_reqs=CERT_REQ
```

The fields of the URL are defined as follows:

1. `password`

Password used to connect to the database.

2. `host`

Host name or IP address of the Redis server (e.g., *localhost*).

3. `port`

Port to the Redis server. Default is 6379.

4. `db`

Database number to use. Default is 0. The db can include an optional leading slash.

When using a TLS connection (protocol is `rediss://`), you may pass in all values in `broker_use_ssl` as query parameters. Paths to certificates must be URL encoded, and `ssl_cert_reqs` is required. Example:

```
result_backend = 'rediss://:password@host:port/db?\n    ssl_cert_reqs=CERT_REQUIRED\n    &ssl_ca_certs=%2Fvar%2Fssl%2Fmyca.pem\           # /var/ssl/m\n    &ssl_certfile=%2Fvar%2Fssl%2Fredis-server-cert.pem\  # /var/ssl/r\n    &ssl_keyfile=%2Fvar%2Fssl%2Fprivate%2Fworker-key.pem' # /var/ssl/p
```

## `redis_backend_use_ssl`

Default: Disabled.

The Redis backend supports SSL. The valid values of this options are the same as `broker_use_ssl`.

## `redis_max_connections`

Default: No limit.

Maximum number of connections available in the Redis connection pool used for sending and retrieving results.

## `redis_socket_connect_timeout`

*New in version 5.0.1.*

Default: **None**

Socket timeout for connections to Redis from the result backend in seconds (int/float)

## `redis_socket_timeout`

Default: 120.0 seconds.

Socket timeout for reading/writing operations to the Redis server in seconds (int/float), used by the redis result backend.

# Cassandra backend settings

## Note:

This Cassandra backend driver requires [cassandra-driver](#).

To install, use **pip**:

```
$ pip install celery[cassandra]
```

See [Bundles](#) for information on combining multiple extension requirements.

This backend requires the following configuration directives to be set.

## cassandra\_servers

Default: `[]` (empty list).

List of `host` Cassandra servers. For example:

```
cassandra_servers = ['localhost']
```

## cassandra\_port

Default: 9042.

Port to contact the Cassandra servers on.

## cassandra\_keyspace

Default: None.

The key-space in which to store the results. For example:

```
cassandra_keyspace = 'tasks_keyspace'
```

## cassandra\_table

Default: None.

The table (column family) in which to store the results. For example:

```
cassandra_table = 'tasks'
```

## cassandra\_read\_consistency

Default: None.

The read consistency used. Values can be `ONE`, `TWO`, `THREE`, `QUORUM`, `ALL`, `LOCAL_QUORUM`, `EACH_QUORUM`, `LOCAL_ONE`.

## cassandra\_write\_consistency

Default: None.

The write consistency used. Values can be `ONE`, `TWO`, `THREE`, `QUORUM`, `ALL`, `LOCAL_QUORUM`, `EACH_QUORUM`, `LOCAL_ONE`.

## cassandra\_entry\_ttl

Default: None.

Time-to-live for status entries. They will expire and be removed after that many seconds after adding. A value of **None** (default) means they will never expire.

## cassandra\_auth\_provider

Default: **None**.

AuthProvider class within `cassandra.auth` module to use. Values can be `PlainTextAuthProvider` or `SaslAuthProvider`.

## cassandra\_auth\_kwargs

Default: {} (empty mapping).

Named arguments to pass into the authentication provider. For example:

```
cassandra_auth_kwargs = {
    username: 'cassandra',
    password: 'cassandra'
}
```

## cassandra\_options

Default: {} (empty mapping).

Named arguments to pass into the `cassandra.cluster` class.

```
cassandra_options = {
    'cql_version': '3.2.1'
    'protocol_version': 3
}
```

## Example configuration

```
cassandra_servers = ['localhost']
cassandra_keyspace = 'celery'
cassandra_table = 'tasks'
cassandra_read_consistency = 'ONE'
cassandra_write_consistency = 'ONE'
cassandra_entry_ttl = 86400
```

## Elasticsearch backend settings

To use [Elasticsearch](#) as the result backend you simply need to configure the [`result\_backend`](#) setting with the correct URL.

## Example configuration

```
result_backend = 'elasticsearch://example.com:9200/index_name/doc_type'
```

## elasticsearch\_retry\_on\_timeout

Default: **False**

Should timeout trigger a retry on different node?

## `elasticsearch_max_retries`

Default: 3.

Maximum number of retries before an exception is propagated.

## `elasticsearch_timeout`

Default: 10.0 seconds.

Global timeout, used by the elasticsearch result backend.

## Riak backend settings

### Note:

The Riak backend requires the [riak](#) library.

To install this package use **pip**:

```
$ pip install celery[riak]
```

See [Bundles](#) for information on combining multiple extension requirements.

This backend requires the `result_backend` setting to be set to a Riak URL:

```
result_backend = 'riak://host:port/bucket'
```

For example:

```
result_backend = 'riak://localhost/celery'
```

is the same as:

```
result_backend = 'riak://'
```

The fields of the URL are defined as follows:

1. `host`

Host name or IP address of the Riak server (e.g., `'localhost'`).

2. `port`

Port to the Riak server using the protobuf protocol. Default is 8087.

3. `bucket`

Bucket name to use. Default is `celery`. The bucket needs to be a string with ASCII characters only.

Alternatively, this backend can be configured with the following configuration directives.

## `riak_backend_settings`

Default: `{}` (empty mapping).

This is a dict supporting the following keys:



- `host`

The host name of the Riak server. Defaults to `"localhost"`.

- `port`

The port the Riak server is listening to. Defaults to 8087.

- `bucket`

The bucket name to connect to. Defaults to `"celery"`.

- `protocol`

The protocol to use to connect to the Riak server. This isn't configurable via `result_backend`

## AWS DynamoDB backend settings

### Note:

The Dynamodb backend requires the `boto3` library.

To install this package use `pip`:

```
$ pip install celery[dynamodb]
```

See [Bundles](#) for information on combining multiple extension requirements.

This backend requires the `result_backend` setting to be set to a DynamoDB URL:

```
result_backend = 'dynamodb://aws_access_key_id:aws_secret_access_key@reg
```

For example, specifying the AWS region and the table name:

```
result_backend = 'dynamodb://@us-east-1/celery_results
```

or retrieving AWS configuration parameters from the environment, using the default table name (`celery`) and specifying read and write provisioned throughput:

```
result_backend = 'dynamodb://@/?read=5&write=5'
```

or using the [downloadable version](#) of DynamoDB [locally](#):

```
result_backend = 'dynamodb://@localhost:8000'
```

or using downloadable version or other service with conforming API deployed on any host:

```
result_backend = 'dynamodb://@us-east-1'  
dynamodb_endpoint_url = 'http://192.168.0.40:8000'
```

The fields of the DynamoDB URL in `result_backend` are defined as follows:

1. `aws_access_key_id` & `aws_secret_access_key`

The credentials for accessing AWS API resources. These can also be resolved by the [boto3](#) library from various sources, as described [here](#).

2. `region`

The AWS region, e.g. `us-east-1` or `localhost` for the [Downloadable Version](#). See the [boto3 library documentation](#) for definition options.

### 3. port

The listening port of the local DynamoDB instance, if you are using the downloadable version. If you have not specified the `region` parameter as `localhost`, setting this parameter has **no effect**.

### 4. table

Table name to use. Default is `celery`. See the [DynamoDB Naming Rules](#) for information on the allowed characters and length.

### 5. read & write

The Read & Write Capacity Units for the created DynamoDB table. Default is `1` for both read and write. More details can be found in the [Provisioned Throughput documentation](#).

## IronCache backend settings

### Note:

The IronCache backend requires the `iron_celery` library:

To install this package use `pip`:

```
$ pip install iron_celery
```

IronCache is configured via the URL provided in `result_backend`, for example:

```
result_backend = 'ironcache://project_id:token@'
```

Or to change the cache name:

```
ironcache:://project_id:token@awesomecache
```

For more information, see: [https://github.com/iron-io/iron\\_celery](https://github.com/iron-io/iron_celery)

## Couchbase backend settings

### Note:

The Couchbase backend requires the `couchbase` library.

To install this package use `pip`:

```
$ pip install celery[couchbase]
```

See [Bundles](#) for instructions how to combine multiple extension requirements.

This backend can be configured via the `result_backend` set to a Couchbase URL:

```
result_backend = 'couchbase://username:password@host:port/bucket'
```

## couchbase\_backend\_settings

Default: {} (empty mapping).

This is a dict supporting the following keys:

- **host**

Host name of the Couchbase server. Defaults to `localhost`.

- **port**

The port the Couchbase server is listening to. Defaults to `8091`.

- **bucket**

The default bucket the Couchbase server is writing to. Defaults to `default`.

- **username**

User name to authenticate to the Couchbase server as (optional).

- **password**

Password to authenticate to the Couchbase server (optional).

## CouchDB backend settings

### Note:

The CouchDB backend requires the [pycouchdb](#) library:

To install this Couchbase package use **pip**:

```
$ pip install celery[couchdb]
```

See [Bundles](#) for information on combining multiple extension requirements.

This backend can be configured via the **result\_backend** set to a CouchDB URL:

```
result_backend = 'couchdb://username:password@host:port/container'
```

The URL is formed out of the following parts:

- **username**

User name to authenticate to the CouchDB server as (optional).

- **password**

Password to authenticate to the CouchDB server (optional).

- **host**

Host name of the CouchDB server. Defaults to `localhost`.

- **port**

The port the CouchDB server is listening to. Defaults to `8091`.

- **container**

The default container the CouchDB server is writing to. Defaults to `default`.

## File-system backend settings

This backend can be configured using a file URL, for example:

```
CELERY_RESULT_BACKEND = 'file:///var/celery/results'
```

The configured directory needs to be shared and writable by all servers using the backend.

If you're trying Celery on a single system you can simply use the backend without any further configuration. For larger clusters you could use NFS, [GlusterFS](#), CIFS, [HDFS](#) (using FUSE), or any other file-system.

## Consul K/V store backend settings

The Consul backend can be configured using a URL, for example:

```
CELERY_RESULT_BACKEND = 'consul://localhost:8500/'
```

The backend will storage results in the K/V store of Consul as individual keys.

The backend supports auto expire of results using TTLs in Consul.

## Message Routing

### task\_queues

Default: **None** (queue taken from default queue settings).

Most users will not want to specify this setting and should rather use the [automatic routing facilities](#).

If you really want to configure advanced routing, this setting should be a list of [kombu.Queue](#) objects the worker will consume from.

Note that workers can be overridden this setting via the `-Q` option, or individual queues from this list (by name) can be excluded using the `-x` option.

Also see [Basics](#) for more information.

The default is a queue/exchange/binding key of `celery`, with exchange type `direct`.

See also [task\\_routes](#)

### task\_routes

Default: **None**.

A list of routers, or a single router used to route tasks to queues. When deciding the final destination of a task the routers are consulted in order.

A router can be specified as either:

- A function with the signature `(name, args, kwargs, options, task=None, **kwargs)`
- A string providing the path to a router function.
- A dict containing router specification:  
Will be converted to a `celery.routes.MapRoute` instance.
- A list of `(pattern, route)` tuples:

Will be converted to a **celery.routes.MapRoute** instance.

Examples:

```
task_routes = {
    'celery.ping': 'default',
    'mytasks.add': 'cpu-bound',
    'feed.tasks.*': 'feeds', # <-- glob pattern
    re.compile(r'(image|video)\.tasks\.*'): 'media', # <-- regex
    'video.encode': {
        'queue': 'video',
        'exchange': 'media'
        'routing_key': 'media.video.encode',
    },
}

task_routes = ('myapp.tasks.route_task', {'celery.ping': 'default'})
```

Where `myapp.tasks.route_task` could be:

```
def route_task(self, name, args, kwargs, options, task=None, **kw):
    if task == 'celery.ping':
        return {'queue': 'default'}
```

`route_task` may return a string or a dict. A string then means it's a queue name in `task_queues`, a dict means it's a custom route.

When sending tasks, the routers are consulted in order. The first router that doesn't return `None` is the route to use. The message options is then merged with the found route settings, where the routers settings have priority.

Example if `apply_async()` has these arguments:

```
Task.apply_async(immediate=False, exchange='video',
                 routing_key='video.compress')
```

and a router returns:

```
{'immediate': True, 'exchange': 'urgent'}
```

the final message options will be:

```
immediate=True, exchange='urgent', routing_key='video.compress'
```

(and any default message options defined in the **Task** class)

Values defined in `task_routes` have precedence over values defined in `task_queues` when merging the two.

With the follow settings:

```
task_queues = {
    'cpubound': {
        'exchange': 'cpubound',
        'routing_key': 'cpubound',
    },
}

task_routes = {
    'tasks.add': {
        'queue': 'cpubound',
        'routing_key': 'tasks.add',
        'serializer': 'json',
    },
}
```

The final routing options for `tasks.add` will become:

```
{'exchange': 'cpubound',
 'routing_key': 'tasks.add',
 'serializer': 'json'}
```

See [Routers](#) for more examples.

## task\_queue\_ha\_policy

**brokers:** RabbitMQ

Default: **None**.

This will set the default HA policy for a queue, and the value can either be a string (usually `all`):

```
task_queue_ha_policy = 'all'
```

Using 'all' will replicate the queue to all current nodes, Or you can give it a list of nodes to replicate to:

```
task_queue_ha_policy = ['rabbit@host1', 'rabbit@host2']
```

Using a list will implicitly set `x-ha-policy` to 'nodes' and `x-ha-policy-params` to the given list of nodes.

See <http://www.rabbitmq.com/ha.html> for more information.

## task\_queue\_max\_priority

**brokers:** RabbitMQ

Default: **None**.

See [RabbitMQ Message Priorities](#).

## worker\_direct

Default: Disabled.

This option enables so that every worker has a dedicated queue, so that tasks can be routed to specific workers.

The queue name for each worker is automatically generated based on the worker hostname and a `.dq` suffix, using the `C.dq` exchange.

For example the queue name for the worker with node name `w1@example.com`

becomes:

```
w1@example.com.dq
```

Then you can route the task to the task by specifying the hostname as the routing key and the `C.dq` exchange:

```
task_routes = {
    'tasks.add': {'exchange': 'C.dq', 'routing_key': 'w1@example.com'}
}
```

## `task_create_missing_queues`

Default: Enabled.

If enabled (default), any queues specified that aren't defined in `task_queues` will be automatically created. See [Automatic routing](#).

## `task_default_queue`

Default: "celery".

The name of the default queue used by `.apply_async` if the message has no route or no custom queue has been specified.

This queue must be listed in `task_queues`. If `task_queues` isn't specified then it's automatically created containing one queue entry, where this name is used as the name of that queue.

## See also:

[Changing the name of the default queue](#)

## `task_default_exchange`

Default: "celery".

Name of the default exchange to use when no custom exchange is specified for a key in the `task_queues` setting.

## `task_default_exchange_type`

Default: "direct".

Default exchange type used when no custom exchange type is specified for a key in the `task_queues` setting.

## `task_default_routing_key`

Default: "celery".

The default routing key used when no custom routing key is specified for a key in the `task_queues` setting.

## `task_default_delivery_mode`

Default: "persistent".

Can be *transient* (messages not written to disk) or *persistent* (written to disk).

## Broker Settings

### `broker_url`

Default: `"amqp://"`

Default broker URL. This must be a URL in the form of:

```
transport://userid:password@hostname:port/virtual_host
```

Only the scheme part (`transport://`) is required, the rest is optional, and defaults to the specific transports default values.

The transport part is the broker implementation to use, and the default is `amqp`, (uses `librabbitmq` if installed or falls back to `pyamqp`). There are also other choices available, including: `redis://`, `sqs://`, and `qpid://`.

The scheme can also be a fully qualified path to your own transport implementation:

```
broker_url = 'proj.transports.MyTransport://localhost'
```

More than one broker URL, of the same transport, can also be specified. The broker URLs can be passed in as a single string that's semicolon delimited:

```
broker_url = 'transport://userid:password@hostname:port//;transport://us
```

Or as a list:

```
broker_url = [  
    'transport://userid:password@localhost:port//',  
    'transport://userid:password@hostname:port//'  
]
```

The brokers will then be used in the `broker_failover_strategy`.

See [URLs](#) in the Kombu documentation for more information.

### `broker_read_url` / `broker_write_url`

Default: Taken from `broker_url`.

These settings can be configured, instead of `broker_url` to specify different connection parameters for broker connections used for consuming and producing.

Example:

```
broker_read_url = 'amqp://user:pass@broker.example.com:56721'  
broker_write_url = 'amqp://user:pass@broker.example.com:56722'
```

Both options can also be specified as a list for failover alternates, see `broker_url` for more information.

### `broker_failover_strategy`

Default: `"round-robin"`.

Default failover strategy for the broker Connection object. If supplied, may map to a key



in 'kombu.connection.failover\_strategies', or be a reference to any method that yields a single item from a supplied list.

Example:

```
# Random failover strategy
def random_failover_strategy(servers):
    it = list(servers) # don't modify callers list
    shuffle = random.shuffle
    for _ in repeat(None):
        shuffle(it)
        yield it[0]

broker_failover_strategy = random_failover_strategy
```

## broker\_heartbeat

**transports supported:**

pyamqp

Default: 120.0 (negotiated by server).

Note: This value is only used by the worker, clients do not use a heartbeat at the moment.

It's not always possible to detect connection loss in a timely manner using TCP/IP alone, so AMQP defines something called heartbeats that's is used both by the client and the broker to detect if a connection was closed.

If the heartbeat value is 10 seconds, then the heartbeat will be monitored at the interval specified by the **broker\_heartbeat\_checkrate** setting (by default this is set to double the rate of the heartbeat value, so for the 10 seconds, the heartbeat is checked every 5 seconds).

## broker\_heartbeat\_checkrate

**transports supported:**

pyamqp

Default: 2.0.

At intervals the worker will monitor that the broker hasn't missed too many heartbeats. The rate at which this is checked is calculated by dividing the **broker\_heartbeat** value with this value, so if the heartbeat is 10.0 and the rate is the default 2.0, the check will be performed every 5 seconds (twice the heartbeat sending rate).

## broker\_use\_ssl

**transports supported:**

pyamqp, redis

Default: Disabled.

Toggles SSL usage on broker connection and SSL settings.

The valid values for this option vary by transport.

pyamqp

If **True** the connection will use SSL with default SSL settings. If set to a dict, will configure SSL connection according to the specified policy. The format used is Python's **ssl.wrap\_socket()** options.

Note that SSL socket is generally served on a separate port by the broker.

Example providing a client cert and validating the server cert against a custom certificate authority:

```
import ssl

broker_use_ssl = {
    'keyfile': '/var/ssl/private/worker-key.pem',
    'certfile': '/var/ssl/amqp-server-cert.pem',
    'ca_certs': '/var/ssl/myca.pem',
    'cert_reqs': ssl.CERT_REQUIRED
}
```

## Warning:

Be careful using `broker_use_ssl=True`. It's possible that your default configuration won't validate the server cert at all. Please read Python [ssl module security considerations](#).

`redis`

The setting must be a dict the keys:

- `ssl_cert_reqs` (required): one of the `SSLContext.verify_mode` values:
  - `ssl.CERT_NONE`
  - `ssl.CERT_OPTIONAL`
  - `ssl.CERT_REQUIRED`
- `ssl_ca_certs` (optional): path to the CA certificate
- `ssl_certfile` (optional): path to the client certificate
- `ssl_keyfile` (optional): path to the client key

## `broker_pool_limit`

*New in version 2.3.*

Default: 10.

The maximum number of connections that can be open in the connection pool.

The pool is enabled by default since version 2.5, with a default limit of ten connections. This number can be tweaked depending on the number of threads/green-threads (eventlet/gevent) using a connection. For example running eventlet with 1000 greenlets that use a connection to the broker, contention can arise and you should consider increasing the limit.

If set to **None** or 0 the connection pool will be disabled and connections will be established and closed for every use.

## `broker_connection_timeout`

Default: 4.0.

The default timeout in seconds before we give up establishing a connection to the AMQP server. This setting is disabled when using gevent.

## Note:

The broker connection timeout only applies to a worker attempting to connect to the broker. It does not apply to producer sending a task, see [broker\\_transport\\_options](#) for how to provide a timeout for that situation.

## broker\_connection\_retry

Default: Enabled.

Automatically try to re-establish the connection to the AMQP broker if lost.

The time between retries is increased for each retry, and is not exhausted before `broker_connection_max_retries` is exceeded.

## broker\_connection\_max\_retries

Default: 100.

Maximum number of retries before we give up re-establishing a connection to the AMQP broker.

If this is set to **0** or **None**, we'll retry forever.

## broker\_login\_method

Default: "AMQPLAIN".

Set custom amqp login method.

## broker\_transport\_options

*New in version 2.2.*

Default: {} (empty mapping).

A dict of additional options passed to the underlying transport.

See your transport user manual for supported options (if any).

Example setting the visibility timeout (supported by Redis and SQS transports):

```
broker_transport_options = {'visibility_timeout': 18000} # 5 hours
```

# Worker

## imports

Default: [] (empty list).

A sequence of modules to import when the worker starts.

This is used to specify the task modules to import, but also to import signal handlers and additional remote control commands, etc.

The modules will be imported in the original order.

## include

Default: [] (empty list).

Exact same semantics as `imports`, but can be used as a means to have different import categories.

The modules in this setting are imported after the modules in `imports`.

## `worker_concurrency`

Default: Number of CPU cores.

The number of concurrent worker processes/threads/green threads executing tasks.

If you're doing mostly I/O you can have more processes, but if mostly CPU-bound, try to keep it close to the number of CPUs on your machine. If not set, the number of CPUs/cores on the host will be used.

## `worker_prefetch_multiplier`

Default: 4.

How many messages to prefetch at a time multiplied by the number of concurrent processes. The default is 4 (four messages for each process). The default setting is usually a good choice, however – if you have very long running tasks waiting in the queue and you have to start the workers, note that the first worker to start will receive four times the number of messages initially. Thus the tasks may not be fairly distributed to the workers.

To disable prefetching, set `worker_prefetch_multiplier` to 1. Changing that setting to 0 will allow the worker to keep consuming as many messages as it wants.

For more on prefetching, read [Prefetch Limits](#)

### Note:

Tasks with ETA/countdown aren't affected by prefetch limits.

## `worker_lost_wait`

Default: 10.0 seconds.

In some cases a worker may be killed without proper cleanup, and the worker may have published a result before terminating. This value specifies how long we wait for any missing results before raising a `WorkerLostError` exception.

## `worker_max_tasks_per_child`

Maximum number of tasks a pool worker process can execute before it's replaced with a new one. Default is no limit.

## `worker_max_memory_per_child`

Default: No limit. Type: int (kilobytes)

Maximum amount of resident memory, in kilobytes, that may be consumed by a worker before it will be replaced by a new worker. If a single task causes a worker to exceed this limit, the task will be completed, and the worker will be replaced afterwards.

Example:

```
worker_max_memory_per_child = 12000 # 12MB
```

## `worker_disable_rate_limits`

Default: Disabled (rate limits enabled).

Disable all rate limits, even if tasks has explicit rate limits set.

## worker\_state\_db

Default: **None**.

Name of the file used to store persistent worker state (like revoked tasks). Can be a relative or absolute path, but be aware that the suffix `.db` may be appended to the file name (depending on Python version).

Can also be set via the `celery worker --statedb` argument.

## worker\_timer\_precision

Default: 1.0 seconds.

Set the maximum time in seconds that the ETA scheduler can sleep between rechecking the schedule.

Setting this value to 1 second means the scheduler's precision will be 1 second. If you need near millisecond precision you can set this to 0.1.

## worker\_enable\_remote\_control

Default: Enabled by default.

Specify if remote control of the workers is enabled.

# Events

## worker\_send\_task\_events

Default: Disabled by default.

Send task-related events so that tasks can be monitored using tools like *flower*. Sets the default value for the workers `-E` argument.

## task\_send\_sent\_event

*New in version 2.2.*

Default: Disabled by default.

If enabled, a `task-sent` event will be sent for every task so tasks can be tracked before they're consumed by a worker.

## event\_queue\_ttl

**transports supported:**

amqp

Default: 5.0 seconds.

Message expiry time in seconds (int/float) for when messages sent to a monitor clients event queue is deleted (`x-message-ttl`)

For example, if this value is set to 10 then a message delivered to this queue will be deleted after 10 seconds.

## event\_queue\_expires

**transports supported:**

`amqp`

Default: 60.0 seconds.

Expiry time in seconds (int/float) for when after a monitor clients event queue will be deleted (`x-expires`).

## `event_queue_prefix`

Default: "celeryev".

The prefix to use for event receiver queue names.

## `event_serializer`

Default: "json".

Message serialization format used when sending event messages.

### See also:

[Serializers](#).

## Remote Control Commands

### Note:

To disable remote control commands see the `worker_enable_remote_control` setting.

## `control_queue_ttl`

Default: 300.0

Time in seconds, before a message in a remote control command queue will expire.

If using the default of 300 seconds, this means that if a remote control command is sent and no worker picks it up within 300 seconds, the command is discarded.

This setting also applies to remote control reply queues.

## `control_queue_expires`

Default: 10.0

Time in seconds, before an unused remote control command queue is deleted from the broker.

This setting also applies to remote control reply queues.

## Logging

### `worker_hijack_root_logger`

*New in version 2.2.*

Default: Enabled by default (hijack root logger).

By default any previously configured handlers on the root logger will be removed. If you want to customize your own logging handlers, then you can disable this behavior by setting `worker_hijack_root_logger = False`.

## Note:

Logging can also be customized by connecting to the `celery.signals.setup_logging` signal.

### `worker_log_color`

Default: Enabled if app is logging to a terminal.

Enables/disables colors in logging output by the Celery apps.

### `worker_log_format`

Default:

```
"[%asctime)s: %(levelname)s/%(processName)s] %(message)s"
```

The format to use for log messages.

See the Python [logging](#) module for more information about log formats.

### `worker_task_log_format`

Default:

```
"[%asctime)s: %(levelname)s/%(processName)s  
[%task_name)s(%task_id)s] %(message)s"
```

The format to use for log messages logged in tasks.

See the Python [logging](#) module for more information about log formats.

### `worker_redirect_stdouts`

Default: Enabled by default.

If enabled `stdout` and `stderr` will be redirected to the current logger.

Used by **celery worker** and **celery beat**.

### `worker_redirect_stdouts_level`

Default: **WARNING**.

The log level output to `stdout` and `stderr` is logged as. Can be one of **DEBUG**, **INFO**, **WARNING**, **ERROR**, or **CRITICAL**.

## Security

### `security_key`

Default: **None**.

*New in version 2.5.*

The relative or absolute path to a file containing the private key used to sign messages when [Message Signing](#) is used.

## security\_certificate

Default: **None**.

*New in version 2.5.*

The relative or absolute path to an X.509 certificate file used to sign messages when [Message Signing](#) is used.

## security\_cert\_store

Default: **None**.

*New in version 2.5.*

The directory containing X.509 certificates used for [Message Signing](#). Can be a glob with wild-cards, (for example `/etc/certs/*.pem`).

# Custom Component Classes (advanced)

## worker\_pool

Default: `"prefork" (celery.concurrency.prefork:TaskPool)`.

Name of the pool class used by the worker.

### Eventlet/Gevent:

Never use this option to select the eventlet or gevent pool. You must use the `-P` option to **celery worker** instead, to ensure the monkey patches aren't applied too late, causing things to break in strange ways.

## worker\_pool\_restarts

Default: Disabled by default.

If enabled the worker pool can be restarted using the **pool\_restart** remote control command.

## worker\_autoscaler

*New in version 2.2.*

Default: `"celery.worker.autoscale:Autoscaler"`.

Name of the autoscaler class to use.

## worker\_consumer

Default: `"celery.worker.consumer:Consumer"`.

Name of the consumer class used by the worker.

## worker\_timer



Default: `"kombu.asynchronous.hub.timer:Timer"`.

Name of the ETA scheduler class used by the worker. Default is or set by the pool implementation.

## Beat Settings (`celery beat`)

### `beat_schedule`

Default: `{}` (empty mapping).

The periodic task schedule used by `beat`. See [Entries](#).

### `beat_scheduler`

Default: `"celery.beat:PersistentScheduler"`.

The default scheduler class. May be set to `"django_celery_beat.schedulers:DatabaseScheduler"` for instance, if used alongside *django-celery-beat* extension.

Can also be set via the `celery beat -S` argument.

### `beat_schedule_filename`

Default: `"celerybeat-schedule"`.

Name of the file used by *PersistentScheduler* to store the last run times of periodic tasks. Can be a relative or absolute path, but be aware that the suffix *.db* may be appended to the file name (depending on Python version).

Can also be set via the `celery beat --schedule` argument.

### `beat_sync_every`

Default: 0.

The number of periodic tasks that can be called before another database sync is issued. A value of 0 (default) means sync based on timing - default of 3 minutes as determined by `scheduler.sync_every`. If set to 1, beat will call sync after every task message sent.

### `beat_max_loop_interval`

Default: 0.

The maximum number of seconds `beat` can sleep between checking the schedule.

The default for this value is scheduler specific. For the default Celery beat scheduler the value is 300 (5 minutes), but for the *django-celery-beat* database scheduler it's 5 seconds because the schedule may be changed externally, and so it must take changes to the schedule into account.

Also when running Celery beat embedded (`-B`) on Jython as a thread the max interval is overridden and set to 1 so that it's possible to shut down in a timely manner.