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 `celery_beat_` to `beat_`, `celeryd_` to `worker_`, and most of the top level `celery_` settings have been moved into a new `task_` prefix.

Warning:

Celery will still be able to read old configuration files until Celery 6.0. Afterwards, support for the old configuration files will be removed. We provide the `celery upgrade` command that should handle plenty of cases (including Django).

Please migrate to the new configuration scheme as soon as possible.

Setting name	Replace with
CELERY_ACCEPT_CONTENT	**accept_content**
CELERY_ENABLE_UTC	**enable_utc**
CELERY_IMPORTS	**imports**
CELERY_INCLUDE	**include**
CELERY_TIMEZONE	**timezone**
CELERYBEAT_MAX_LOOP_INTERVAL	**beat_max_loop_interval**
CELERYBEAT_SCHEDULE	**beat_schedule**
CELERYBEAT_SCHEDULER	**beat_scheduler**
CELERYBEAT_SCHEDULE_FILENAME	**beat_schedule_filename**
CELERYBEAT_SYNC_EVERY	**beat_sync_every**
BROKER_URL	**broker_url**
BROKER_TRANSPORT	**broker_transport**
BROKER_TRANSPORT_OPTIONS	**broker_transport_options**

CASSANDRA_COLUMN_FAMILY	cassandra_table
CASSANDRA_ENTRY_TTL	cassandra_entry_ttl
CASSANDRA_KEYSPACE	cassandra_keyspace
CASSANDRA_PORT	cassandra_port
CASSANDRA_READ_CONSISTENCY	cassandra_read_consistency
CASSANDRA_SERVERS	cassandra_servers
CASSANDRA_WRITE_CONSISTENCY	cassandra_write_consistency
CASSANDRA_OPTIONS	cassandra_options
S3_ACCESS_KEY_ID	s3_access_key_id
S3_SECRET_ACCESS_KEY	s3_secret_access_key
S3_BUCKET	s3_bucket
S3_BASE_PATH	s3_base_path
S3_ENDPOINT_URL	s3_endpoint_url
S3_REGION	s3_region
CELERY_COUCHBASE_BACKEND_SETTINGS	couchbase_backend_settings
CELERY_ARANGODB_BACKEND_SETTINGS	arangodb_backend_settings
CELERY_MONGODB_BACKEND_SETTINGS	mongodb_backend_settings
CELERY_EVENT_QUEUE_EXPIRES	event_queue_expires
CELERY_EVENT_QUEUE_TTL	event_queue_ttl
CELERY_EVENT_QUEUE_PREFIX	event_queue_prefix
CELERY_EVENT_SERIALIZER	event_serializer
CELERY_REDIS_DB	redis_db
CELERY_REDIS_HOST	redis_host
CELERY_REDIS_MAX_CONNECTIONS	redis_max_connections
CELERY_REDIS_USERNAME	redis_username

CELERY_RESULT_SERIALIZER	result_serializer
CELERY_RESULT_DBURI	Use result_backend instead.
CELERY_RESULT_ENGINE_OPTIONS	database_engine_options
[...]_DB_SHORT_LIVED_SESSIONS	database_short_lived_sessions
CELERY_RESULT_DB_TABLE_NAMES	database_db_names
CELERY_SECURITY_CERTIFICATE	security_certificate
CELERY_SECURITY_CERT_STORE	security_cert_store
CELERY_SECURITY_KEY	security_key
CELERY_SECURITY_KEY_PASSWORD	security_key_password
CELERY_ACKS_LATE	task_acks_late
CELERY_ACKS_ON_FAILURE_OR_TIMEOUT	task_acks_on_failure_or_timeout
CELERY_TASK_ALWAYS_EAGER	task_always_eager
CELERY_ANNOTATIONS	task_annotations
CELERY_COMPRESSION	task_compression
CELERY_CREATE_MISSING_QUEUES	task_create_missing_queues
CELERY_DEFAULT_DELIVERY_MODE	task_default_delivery_mode
CELERY_DEFAULT_EXCHANGE	task_default_exchange
CELERY_DEFAULT_EXCHANGE_TYPE	task_default_exchange_type
CELERY_DEFAULT_QUEUE	task_default_queue
CELERY_DEFAULT_RATE_LIMIT	task_default_rate_limit
CELERY_DEFAULT_ROUTING_KEY	task_default_routing_key
CELERY_EAGER_PROPAGATES	task_eager_propagates
CELERY_IGNORE_RESULT	task_ignore_result
CELERY_PUBLISH_RETRY	task_publish_retry
CELERY_PUBLISH_RETRY_POLICY	task_publish_retry_policy

CELERYD_AUTOSCALER	worker_autoscaler
CELERYD_CONCURRENCY	worker_concurrency
CELERYD_CONSUMER	worker_consumer
CELERY_WORKER_DIRECT	worker_direct
CELERY_DISABLE_RATE_LIMITS	worker_disable_rate_limits
CELERY_ENABLE_REMOTE_CONTROL	worker_enable_remote_control
CELERYD_HIJACK_ROOT_LOGGER	worker_hijack_root_logger
CELERYD_LOG_COLOR	worker_log_color
CELERY_WORKER_LOG_FORMAT	worker_log_format
CELERYD_WORKER_LOST_WAIT	worker_lost_wait
CELERYD_MAX_TASKS_PER_CHILD	worker_max_tasks_per_child
CELERYD_POOL	worker_pool
CELERYD_POOL_PUTLOCKS	worker_pool_putlocks
CELERYD_POOL_RESTARTS	worker_pool_restarts
CELERYD_PREFETCH_MULTIPLIER	worker_prefetch_multiplier
CELERYD_REDIRECT_STDOUTS	worker_redirect_stdouts
CELERYD_REDIRECT_STDOUTS_LEVEL	worker_redirect_stdouts_level
CELERY_SEND_EVENTS	worker_send_task_events
CELERYD_STATE_DB	worker_state_db
CELERY_WORKER_TASK_LOG_FORMAT	worker_task_log_format
CELERYD_TIMER	worker_timer
CELERYD_TIMER_PRECISION	worker_timer_precision

Configuration Directives

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

result_accept_content

Default: None (can be set, list or tuple).

New in version 4.3.

A white-list of content-types/serializers to allow for the result backend.

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

By default it is the same serializer as `accept_content`. However, a different serializer for accepted content of the result backend can be specified. Usually this is needed if signed messaging is used and the result is stored unsigned in the result backend. See Security for more.

Example:

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

# or the actual content-type (MIME)
result_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.

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:

    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.

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

will not be saved.

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.

```
header = group([t1, t2])
body = t3
c = chord(header, body)
c.link_error(error_callback_sig)
```

If *any* of the header tasks failed (`t1` or `t2`), by default, the chord body (`t3`) would **not execute**, and `error_callback_sig` will be called **once** (for the body).

Enabling this flag will change the above behavior by:

1. `error_callback_sig` will be linked to `t1` and `t2` (as well as `t3`).
2. If *any* of the header tasks failed, `error_callback_sig` will be called **for each** failed header task **and** the body (even if the body didn't run).

Consider now the following canvas with the flag enabled:

```
header = group([failingT1, failingT2])
body = t3
c = chord(header, body)
c.link_error(error_callback_sig)
```

If *all* of the header tasks failed (`failingT1` and `failingT2`), then the chord body (`t3`) would **not execute**, and `error_callback_sig` will be called **3 times** (two times for the header and one time for the body).

Lastly, consider the following canvas with the flag enabled:

```
header = group([failingT1, failingT2])
body = t3
upgraded_chord = chain(header, body)
upgraded_chord.link_error(error_callback_sig)
```

This canvas will behave exactly the same as the previous one, since the `chain` will be upgraded to a `chord` internally.

`task_soft_time_limit`

Default: Disabled.

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

See also:

FAQ: Should I use retry or acks_late?.

`task_acks_on_failure_or_timeout`

Default: Enabled

When enabled messages for all tasks will be acknowledged even if they fail or time out.

Configuring this setting only applies to tasks that are acknowledged **after** they have been executed and only if `task_acks_late` is enabled.

`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.

- `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.
- mongodb
  Use MongoDB to store the results. See MongoDB backend settings.
- `cassandra`
  Use Cassandra to store the results. See Cassandra/AstraDB 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.
- `arangodb`
  Use ArangoDB to store the results. See ArangoDB backend settings.
- `couchdb`
  Use CouchDB to store the results. See CouchDB backend settings.
- `cosmosdbsql (experimental)`
  Use the CosmosDB PaaS to store the results. See CosmosDB backend settings (experimental).
- `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.
- `azureblockblob`
  Use the AzureBlockBlob PaaS store to store the results See Azure Block Blob backend settings.
- `s3`
  Use the S3 to store the results See S3 backend settings.

This specifies the base amount of sleep time between two backend operation retry.

result_backend_max_retries

Default: Inf

This is the maximum of retries in case of recoverable exceptions.

result_backend_thread_safe

Default: False

If True, then the backend object is shared across threads. This may be useful for using a shared connection pool instead of creating a connection for every thread.

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 hours
```

result_serializer

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

Result serialization format.

See Serializers for information about supported serialization formats.

result_compression

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, Couchbase, 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.

`result_chord_join_timeout`

Default: 3.0.

The timeout in seconds (int/float) when joining a group's results within a chord.

`result_chord_retry_interval`

Default: 1.0.

Default interval for retrying chord tasks.

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 **database_engine_options** setting:

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

database_short_lived_sessions

```
    'task': 'celery',
    'group': 'celery',
}
```

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
```

Please note: using this backend could trigger the raise of `celery.backends.rpc.BacklogLimitExceeded` if the task tombstone is too *old*.

E.g.

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 in celery’s builtin backends as it’s now possible to specify the cache backend directly in the **result_backend** setting.

Note:

The django-celery-results - Using the Django ORM/Cache as a result backend library uses cache_backend for choosing django caches.

MongoDB backend settings

number of TCP connections to keep open to MongoDB at a given time. If there are more open connections than max_pool_size, sockets will be closed when they are released. Defaults to 10.

- options

  Additional keyword arguments to pass to the mongodb connection constructor. See the **pymongo** docs to see a list of arguments supported.

Example configuration

```
result_backend = 'mongodb://localhost:27017/'
mongodb_backend_settings = {
    'database': 'mydb',
    'taskmeta_collection': 'my_taskmeta_collection',
}
```

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://username:password@host:port/db'
```

For example:

```
result_backend = 'socket:///path/to/redis.sock
```

The fields of the URL are defined as follows:

1. `username`

   New in version 5.1.0.

   Username used to connect to the database.

   Note that this is only supported in Redis>=6.0 and with py-redis>=3.4.0 installed.

   If you use an older database version or an older client version you can omit the username:

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

2. `password`

   Password used to connect to the database.

3. `host`

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

4. `port`

   Port to the Redis server. Default is 6379.

5. 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?\
    ssl_cert_reqs=required\
    &ssl_ca_certs=%2Fvar%2Fssl%2Fmyca.pem\                  # /var/ssl/myca.pem
```

redis_backend_use_ssl

Default: Disabled.

The Redis backend supports SSL. This value must be set in the form of a dictionary. The valid key-value pairs are the same as the ones mentioned in the `redis` sub-section under **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.

> **Warning:**
>
> Redis will raise a *ConnectionError* if the number of concurrent connections exceeds the maximum.

redis_socket_connect_timeout

New in version 4.0.1.

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.

redis_retry_on_timeout

New in version 4.4.1.

This Cassandra backend driver requires cassandra-driver.

This backend can refer to either a regular Cassandra installation or a managed Astra DB instance. Depending on which one, exactly one between the **cassandra_servers** and **cassandra_secure_bundle_path** settings must be provided (but not both).

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. This must be provided when connecting to a Cassandra cluster. Passing this setting is strictly exclusive to **cassandra_secure_bundle_path**. Example:

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

cassandra_secure_bundle_path

Default: None.

Absolute path to the secure-connect-bundle zip file to connect to an Astra DB instance. Passing this setting is strictly exclusive to **cassandra_servers**. Example:

```
cassandra_secure_bundle_path = '/home/user/bundles/secure-connect.zip'
```

When connecting to Astra DB, it is necessary to specify the plain-text auth provider and the associated username and password, which take the value of the Client ID and the Client Secret, respectively, of a valid token generated for the Astra DB instance. See below for an Astra DB configuration example.

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

```
    'protocol_version': 3
}
```

Example configuration (Cassandra)

```
result_backend = 'cassandra://'
cassandra_servers = ['localhost']
cassandra_keyspace = 'celery'
cassandra_table = 'tasks'
cassandra_read_consistency = 'QUORUM'
cassandra_write_consistency = 'QUORUM'
cassandra_entry_ttl = 86400
```

Example configuration (Astra DB)

```
result_backend = 'cassandra://'
cassandra_keyspace = 'celery'
cassandra_table = 'tasks'
cassandra_read_consistency = 'QUORUM'
cassandra_write_consistency = 'QUORUM'
cassandra_auth_provider = 'PlainTextAuthProvider'
cassandra_auth_kwargs = {
  'username': '<<CLIENT_ID_FROM_ASTRA_DB_TOKEN>>',
  'password': '<<CLIENT_SECRET_FROM_ASTRA_DB_TOKEN>>'
}
cassandra_secure_bundle_path = '/path/to/secure-connect-bundle.zip'
cassandra_entry_ttl = 86400
```

Additional configuration

The Cassandra driver, when estabilishing the connection, undergoes a stage of negotiating the protocol version with the server(s). Similarly, a load-balancing policy is automatically supplied (by default `DCAwareRoundRobinPolicy`, which in turn has a `local_dc` setting, also determined by the driver upon connection). When possible, one should explicitly provide these in the configuration: moreover, future versions of the Cassandra driver will require at least the load-balancing policy to be specified (using execution profiles, as shown below).

```
from cassandra.cluster import ExecutionProfile
from cassandra.cluster import EXEC_PROFILE_DEFAULT
myEProfile = ExecutionProfile(
  load_balancing_policy=DCAwareRoundRobinPolicy(
    local_dc='europe-west1',  # for Astra DB, region name = dc name
  )
)
cassandra_options = {
  'protocol_version': 4,      # for Astra DB
  'execution_profiles': {EXEC_PROFILE_DEFAULT: myEProfile},
}
```

S3 backend settings

Note:

This s3 backend driver requires s3.

To install, use **s3**:

```
$ pip install celery[s3]
```

See Bundles for information on combining multiple extension requirements.

This backend requires the following configuration directives to be set.

s3_access_key_id

Default: None.

The s3 access key id. For example:

```
s3_access_key_id = 'acces_key_id'
```

s3_secret_access_key

Default: None.

A base path in the s3 bucket to use to store result keys. For example:

```
s3_base_path = '/prefix'
```

s3_endpoint_url

Default: None.

A custom s3 endpoint url. Use it to connect to a custom self-hosted s3 compatible backend (Ceph, Scality...). For example:

```
s3_endpoint_url = 'https://.s3.custom.url'
```

s3_region

Default: None.

The s3 aws region. For example:

```
s3_region = 'us-east-1'
```

Example configuration

```
s3_access_key_id = 's3-access-key-id'
s3_secret_access_key = 's3-secret-access-key'
s3_bucket = 'mybucket'
s3_base_path = '/celery_result_backend'
s3_endpoint_url = 'https://endpoint_url'
```

Azure Block Blob backend settings

To use AzureBlockBlob as the result backend you simply need to configure the **`result_backend`** setting with the correct URL.

New in version 5.1.

Default: None.

A base path in the storage container to use to store result keys. For example:

```
azureblockblob_base_path = 'prefix/'
```

azureblockblob_retry_initial_backoff_sec

Default: 2.

The initial backoff interval, in seconds, for the first retry. Subsequent retries are attempted with an exponential strategy.

azureblockblob_retry_increment_base

Default: 2.

azureblockblob_retry_max_attempts

Default: 3.

The maximum number of retry attempts.

azureblockblob_connection_timeout

Default: 20.

Timeout in seconds for establishing the azure block blob connection.

azureblockblob_read_timeout

Default: 120.

`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.

`elasticsearch_save_meta_as_text`

Default: **`True`**

Should meta saved as text or as native json. Result is always serialized as text.

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.

Warning:

The Dynamodb backend is not compatible with tables that have a sort key defined.

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.

6. `ttl_seconds`

```
$ 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

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

ArangoDB backend settings

Note:

The ArangoDB backend requires the pyArango library.

To install this package use **pip**:

```
$ pip install celery[arangodb]
```

See Bundles for instructions how to combine multiple extension requirements.

This backend can be configured via the **result_backend** set to a ArangoDB URL:

```
result_backend = 'arangodb://username:password@host:port/database/collection'
```

arangodb_backend_settings

Default: {} (empty mapping).

This is a dict supporting the following keys:

- `host`

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

- `port`

  The port the ArangoDB server is listening to. Defaults to `8529`.

- `database`

  The default database in the ArangoDB server is writing to. Defaults to `celery`.

- `collection`

  The default collection in the ArangoDB servers database is writing to. Defaults to `celery`.

To use CosmosDB as the result backend, you simply need to configure the **result_backend** setting with the correct URL.

Example configuration

```
result_backend = 'cosmosdbsql://:{InsertAccountPrimaryKeyHere}@{InsertAccountNameHere}.documer
```

`cosmosdbsql_database_name`

Default: celerydb.

The name for the database in which to store the results.

`cosmosdbsql_collection_name`

Default: celerycol.

The name of the collection in which to store the results.

`cosmosdbsql_consistency_level`

Default: Session.

Represents the consistency levels supported for Azure Cosmos DB client operations.

Consistency levels by order of strength are: Strong, BoundedStaleness, Session, ConsistentPrefix and Eventual.

`cosmosdbsql_max_retry_attempts`

Default: 9.

Maximum number of retries to be performed for a request.

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.

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

The backend will store results in the K/V store of Consul as individual keys. The backend supports auto expire of results using TTLs in Consul. The full syntax of the URL is:

```
consul://host:port[?one_client=1]
```

The URL is formed out of the following parts:

- `host`

  Host name of the Consul server.

- `port`

  The port the Consul server is listening to.

- `one_client`

  By default, for correctness, the backend uses a separate client connection per operation. In cases of extreme load, the rate of creation of new connections can cause HTTP 429 "too many connections" error responses from the Consul server when under load. The recommended way to handle this is to enable retries in `python-consul2` using the patch at https://github.com/poppyred/python-consul2/pull/31.

  Alternatively, if `one_client` is set, a single client connection will be used for all operations instead. This should eliminate the HTTP 429 errors, but the storage of results in the backend can become unreliable.

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.

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 task's settings have priority.

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_max_priority`

brokers: RabbitMQ

Default: **None**.

See RabbitMQ Message Priorities.

`task_default_priority`

brokers: RabbitMQ, Redis

Default: **None**.

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.dq2` 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 worker by specifying the hostname as the routing key and the `C.dq2` exchange:

```
task_routes = {
    'tasks.add': {'exchange': 'C.dq2', '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.

setting.

task_default_routing_key

Default: Uses the value set for **task_default_queue**.

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'
```

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`

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
}
```

New in version 5.1: Starting from Celery 5.1, py-amqp will always validate certificates received from the server and it is no longer required to manually set `cert_reqs` to `ssl.CERT_REQUIRED`.

The previous default, `ssl.CERT_NONE` is insecure and we its usage should be discouraged. If you'd like to revert to the previous insecure default set `cert_reqs` to `ssl.CERT_NONE`

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 after the initial connection is made.

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

> **Warning:**

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

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

broker_channel_error_retry

New in version 5.3.

Default: Disabled.

Automatically try to re-establish the connection to the AMQP broker if any invalid response has been returned.

The retry count and interval is the same as that of *broker_connection_retry*. Also, this option doesn't work when *broker_connection_retry* is *False*.

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
```

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_deduplicate_successful_tasks

New in version 5.1.

Default: False

Before each task execution, instruct the worker to check if this task is a duplicate message.

Deduplication occurs only with tasks that have the same identifier, enabled late acknowledgment, were re-delivered by the message broker and their state is SUCCESS in the result backend.

To avoid overflowing the result backend with queries, a local cache of successfully executed tasks is checked before querying the result backend in case the task was already successfully executed by the same worker that received the task.

This cache can be made persistent by setting the **worker_state_db** setting.

If the result backend is not persistent (the RPC backend, for example), this setting is ignored.

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.

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**.

Specify if remote control of the workers is enabled.

worker_proc_alive_timeout

Default: 4.0.

The timeout in seconds (int/float) when waiting for a new worker process to start up.

worker_cancel_long_running_tasks_on_connection_loss

New in version 5.1.

Default: Disabled by default.

Kill all long-running tasks with late acknowledgment enabled on connection loss.

Tasks which have not been acknowledged before the connection loss cannot do so anymore since their channel is gone and the task is redelivered back to the queue. This is why tasks with late acknowledged enabled must be idempotent as they may be executed more than once. In this case, the task is being executed twice per connection loss (and sometimes in parallel in other workers).

When turning this option on, those tasks which have not been completed are cancelled and their execution is terminated. Tasks which have completed in any way before the connection loss are recorded as such in the result backend as long as **`task_ignore_result`** is not enabled.

Warning:

This feature was introduced as a future breaking change. If it is turned off, Celery will emit a warning message.

In Celery 6.0, the **`worker_cancel_long_running_tasks_on_connection_loss`** will be set to `True` by default as the current behavior leads to more problems than it solves.

Events

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_exchange`

Default: `"celeryev"`.

Name of the event exchange.

> **Warning:**
>
> This option is in experimental stage, please use it with caution.

`event_serializer`

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.

`control_exchange`

Default: `"celery"`.

Name of the control command exchange.

> **Warning:**
>
> This option is in experimental stage, please use it with caution.

Logging

`worker_hijack_root_logger`

New in version 2.2.

Default: Enabled by default (hijack root logger).

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_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`).

`security_digest`

Default: **sha256**.

New in version 4.3.

A cryptography digest used to sign messages when Message Signing is used.
https://cryptography.io/en/latest/hazmat/primitives/cryptographic-hashes/#module-cryptography.hazmat.primitives.hashes

Custom Component Classes (advanced)

`worker_pool`

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

Name of the pool class used by the worker.

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"`.

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.

beat_cron_starting_deadline

New in version 5.3.

Default: None.

When using cron, the number of seconds **beat** can look back when deciding whether a cron schedule is due. When set to *None*, cronjobs that are past due will always run immediately.