Previous topic

Application

Next topic

Calling Tasks

This Page

Show Source

Quick search

Go

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

Tasks

Tasks are the building blocks of Celery applications.

A task is a class that can be created out of any callable. It performs dual roles in that it defines both what happens when a task is called (sends a message), and what happens when a worker receives that message.

Every task class has a unique name, and this name is referenced in messages so the worker can find the right function to execute.

A task message is not removed from the queue until that message has been acknowledged by a worker. A worker can reserve many messages in advance and even if the worker is killed – by power failure or some other reason – the message will be redelivered to another worker.

Ideally task functions should be idempotent: meaning the function won't cause unintended effects even if called multiple times with the same arguments. Since the worker cannot detect if your tasks are idempotent, the default behavior is to acknowledge the message in advance, just before it's executed, so that a task invocation that already started is never executed again.

If your task is idempotent you can set the **`acks_late`** option to have the worker acknowledge the message *after* the task returns instead. See also the FAQ entry Should I use retry or acks_late?.

Note that the worker will acknowledge the message if the child process executing the task is terminated (either by the task calling **`sys.exit()`**, or by signal) even when **`acks_late`** is enabled. This behavior is by purpose as…

1. We don't want to rerun tasks that forces the kernel to send a **`SIGSEGV`** (segmentation fault) or similar signals to the process.
2. We assume that a system administrator deliberately killing the task does not want it to automatically restart.
3. A task that allocates too much memory is in danger of triggering the kernel OOM killer, the same may happen again.
4. A task that always fails when redelivered may cause a high-frequency message loop taking down the system.

If you really want a task to be redelivered in these scenarios you should consider enabling the **`task_reject_on_worker_lost`** setting.

Warning:

A task that blocks indefinitely may eventually stop the worker instance from doing any other work.

If you task does I/O then make sure you add timeouts to these operations, like adding a timeout to a web request using the requests library:

```
connect_timeout, read_timeout = 5.0, 30.0
response = requests.get(URL, timeout=(connect_timeout, read_timeout))
```

Time limits are convenient for making sure all tasks return in a timely manner, but a time limit event will actually kill the process by force so only use them to detect cases where

you haven't used manual timeouts yet.

The default prefork pool scheduler is not friendly to long-running tasks, so if you have tasks that run for minutes/hours make sure you enable the `-Ofair` command-line argument to the **celery worker**. See Prefork pool prefetch settings for more information, and for the best performance route long-running and short-running tasks to dedicated workers (Automatic routing).

If your worker hangs then please investigate what tasks are running before submitting an issue, as most likely the hanging is caused by one or more tasks hanging on a network operation.

–

In this chapter you'll learn all about defining tasks, and this is the **table of contents**:

- Basics
- Names
- Task Request
- Logging
- Retrying
- List of Options
- States
- Semipredicates
- Custom task classes
- How it works
- Tips and Best Practices
- Performance and Strategies
- Example

Basics

You can easily create a task from any callable by using the `task()` decorator:

```
from .models import User

@app.task
def create_user(username, password):
    User.objects.create(username=username, password=password)
```

There are also many options that can be set for the task, these can be specified as arguments to the decorator:

```
@app.task(serializer='json')
def create_user(username, password):
    User.objects.create(username=username, password=password)
```

Multiple decorators

When using multiple decorators in combination with the task decorator you must make sure that the *task* decorator is applied last (oddly, in Python this means it must be first in the list):

How do I import the task decorator? And what's "app"?

The task decorator is available on your `Celery` application instance, if you don't know what this is then please read First Steps with Celery.

If you're using Django (see First steps with Django), or you're the author of a library then you probably want to use the

```
@app.task
@decorator2
@decorator1
def add(x, y):
    return x + y
```

`shared_task()` decorator:

```
from celery import shared_ta

@shared_task
def add(x, y):
    return x + y
```

Bound tasks

A task being bound means the first argument to the task will always be the task instance (`self`), just like Python bound methods:

```
logger = get_task_logger(__name__)

@task(bind=True)
def add(self, x, y):
    logger.info(self.request.id)
```

Bound tasks are needed for retries (using `app.Task.retry()`), for accessing information about the current task request, and for any additional functionality you add to custom task base classes.

Task inheritance

The `base` argument to the task decorator specifies the base class of the task:

```
import celery

class MyTask(celery.Task):

    def on_failure(self, exc, task_id, args, kwargs, einfo):
        print('{0!r} failed: {1!r}'.format(task_id, exc))

@task(base=MyTask)
def add(x, y):
    raise KeyError()
```

Names

Every task must have a unique name.

If no explicit name is provided the task decorator will generate one for you, and this name will be based on 1) the module the task is defined in, and 2) the name of the task function.

Example setting explicit name:

```
>>> @app.task(name='sum-of-two-numbers')
>>> def add(x, y):
...     return x + y

>>> add.name
'sum-of-two-numbers'
```

A best practice is to use the module name as a name-space, this way names won't collide if there's already a task with that name defined in another module.

```
>>> @app.task(name='tasks.add')
>>> def add(x, y):
...     return x + y
```

You can tell the name of the task by investigating its `.name` attribute:

```
>>> add.name
'tasks.add'
```

The name we specified here (`tasks.add`) is exactly the name that would've been automatically generated for us if the task was defined in a module named `tasks.py`:

`tasks.py`:

```
@app.task
def add(x, y):
    return x + y
```

```
>>> from tasks import add
>>> add.name
'tasks.add'
```

Automatic naming and relative imports

Relative imports and automatic name generation don't go well together, so if you're using relative imports you should set the name explicitly.

For example if the client imports the module `"myapp.tasks"` as `".tasks"`, and the worker imports the module as `"myapp.tasks"`, the generated names won't match and an `NotRegistered` error will be raised by the worker.

This is also the case when using Django and using `project.myapp`-style naming in `INSTALLED_APPS`:

```
INSTALLED_APPS = ['project.myapp']
```

If you install the app under the name `project.myapp` then the tasks module will be imported as `project.myapp.tasks`, so you must make sure you always import the tasks using the same name:

```
>>> from project.myapp.tasks import mytask   # << GOOD

>>> from myapp.tasks import mytask    # << BAD!!!
```

The second example will cause the task to be named differently since the worker and the client imports the modules under different names:

```
>>> from project.myapp.tasks import mytask
>>> mytask.name
'project.myapp.tasks.mytask'

>>> from myapp.tasks import mytask
>>> mytask.name
'myapp.tasks.mytask'
```

For this reason you must be consistent in how you import modules, and that is also a Python best practice.

Similarly, you shouldn't use old-style relative imports:

Absolute Imports

The best practice for developers targeting Python 2 is to add the following to the top of **every module**:

```
from __future__ import absol
```

This will force you to always use absolute imports so you will never have any problems with tasks using relative names.

Absolute imports are the default in Python 3 so you don't need this if you target that version.

```
from module import foo   # BAD!

from proj.module import foo  # GOOD!
```

New-style relative imports are fine and can be used:

```
from .module import foo  # GOOD!
```

If you want to use Celery with a project already using these patterns extensively and you don't have the time to refactor the existing code then you can consider specifying the names explicitly instead of relying on the automatic naming:

```
@task(name='proj.tasks.add')
def add(x, y):
    return x + y
```

Changing the automatic naming behavior

New in version 4.0.

There are some cases when the default automatic naming isn't suitable. Consider you have many tasks within many different modules:

```
project/
       /__init__.py
       /celery.py
       /moduleA/
               /__init__.py
               /tasks.py
       /moduleB/
               /__init__.py
               /tasks.py
```

Using the default automatic naming, each task will have a generated name like *moduleA.tasks.taskA*, *moduleA.tasks.taskB*, *moduleB.tasks.test*, and so on. You may want to get rid of having *tasks* in all task names. As pointed above, you can explicitly give names for all tasks, or you can change the automatic naming behavior by overriding `app.gen_task_name()`. Continuing with the example, *celery.py* may contain:

```
from celery import Celery

class MyCelery(Celery):

    def gen_task_name(self, name, module):
        if module.endswith('.tasks'):
            module = module[:-6]
        return super(MyCelery, self).gen_task_name(name, module)

app = MyCelery('main')
```

So each task will have a name like *moduleA.taskA*, *moduleA.taskB* and *moduleB.test*.

Warning:

Make sure that your `app.gen_task_name()` is a pure function: meaning that for the same input it must always return the same output.

Task Request

`app.Task.request` contains information and state related to the currently executing

task.

The request defines the following attributes:

id: The unique id of the executing task.
group: The unique id of the task's group, if this task is a member.
chord: The unique id of the chord this task belongs to (if the task is part of the header).
correlation_id: Custom ID used for things like de-duplication.
args: Positional arguments.
kwargs: Keyword arguments.
origin: Name of host that sent this task.
retries: How many times the current task has been retried. An integer starting at *0*.
is_eager: Set to **`True`** if the task is executed locally in the client, not by a worker.
eta: The original ETA of the task (if any). This is in UTC time (depending on the **`enable_utc`** setting).
expires: The original expiry time of the task (if any). This is in UTC time (depending on the **`enable_utc`** setting).
hostname: Node name of the worker instance executing the task.
delivery_info: Additional message delivery information. This is a mapping containing the exchange and routing key used to deliver this task. Used by for example **`app.Task.retry()`** to resend the task to the same destination queue. Availability of keys in this dict depends on the message broker used.
reply-to: Name of queue to send replies back to (used with RPC result backend for example).
called_directly: This flag is set to true if the task wasn't executed by the worker.
timelimit: A tuple of the current `(soft, hard)` time limits active for this task (if any).
callbacks: A list of signatures to be called if this task returns successfully.
errback: A list of signatures to be called if this task fails.
utc: Set to true the caller has UTC enabled (**`enable_utc`**).

New in version 3.1.

headers: Mapping of message headers sent with this task message (may be **`None`**).
reply_to: Where to send reply to (queue name).
correlation_id: Usually the same as the task id, often used in amqp to keep track of what a reply is for.

New in version 4.0.

root_id: The unique id of the first task in the workflow this task is part of (if any).
parent_id: The unique id of the task that called this task (if any).
chain: Reversed list of tasks that form a chain (if any). The last item in this list will be the next task to succeed the current task. If using version one of the task protocol the chain tasks will be in `request.callbacks` instead.

Example

An example task accessing information in the context is:

```
@app.task(bind=True)
def dump_context(self, x, y):
    print('Executing task id {0.id}, args: {0.args!r} kwargs: {0.kwargs!
            self.request))
```

The `bind` argument means that the function will be a "bound method" so that you can access attributes and methods on the task type instance.

Logging

The worker will automatically set up logging for you, or you can configure logging manually.

A special logger is available named “celery.task”, you can inherit from this logger to automatically get the task name and unique id as part of the logs.

The best practice is to create a common logger for all of your tasks at the top of your module:

```
from celery.utils.log import get_task_logger

logger = get_task_logger(__name__)

@app.task
def add(x, y):
    logger.info('Adding {0} + {1}'.format(x, y))
    return x + y
```

Celery uses the standard Python logger library, and the documentation can be found **here**.

You can also use **print()**, as anything written to standard out/-err will be redirected to the logging system (you can disable this, see **worker_redirect_stdouts**).

Note:

The worker won’t update the redirection if you create a logger instance somewhere in your task or task module.

If you want to redirect `sys.stdout` and `sys.stderr` to a custom logger you have to enable this manually, for example:

```
import sys

logger = get_task_logger(__name__)

@app.task(bind=True)
def add(self, x, y):
    old_outs = sys.stdout, sys.stderr
    rlevel = self.app.conf.worker_redirect_stdouts_level
    try:
        self.app.log.redirect_stdouts_to_logger(logger, rlevel)
        print('Adding {0} + {1}'.format(x, y))
        return x + y
    finally:
        sys.stdout, sys.stderr = old_outs
```

Argument checking

New in version 4.0.

Celery will verify the arguments passed when you call the task, just like Python does when calling a normal function:

```
>>> @app.task
... def add(x, y):
...     return x + y

# Calling the task with two arguments works:
>>> add.delay(8, 8)
<AsyncResult: f59d71ca-1549-43e0-be41-4e8821a83c0c>

# Calling the task with only one argument fails:
>>> add.delay(8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "celery/app/task.py", line 376, in delay
    return self.apply_async(args, kwargs)
  File "celery/app/task.py", line 485, in apply_async
    check_arguments(*(args or ()), **(kwargs or {}))
TypeError: add() takes exactly 2 arguments (1 given)
```

You can disable the argument checking for any task by setting its **`typing`** attribute to **`False`**:

```
>>> @app.task(typing=False)
... def add(x, y):
...     return x + y

# Works locally, but the worker reciving the task will raise an error.
>>> add.delay(8)
<AsyncResult: f59d71ca-1549-43e0-be41-4e8821a83c0c>
```

Hiding sensitive information in arguments

New in version 4.0.

When using **`task_protocol`** 2 or higher (default since 4.0), you can override how positional arguments and keyword arguments are represented in logs and monitoring events using the `argsrepr` and `kwargsrepr` calling arguments:

```
>>> add.apply_async((2, 3), argsrepr='(<secret-x>, <secret-y>)')

>>> charge.s(account, card='1234 5678 1234 5678').set(
...     kwargsrepr=repr({'card': '**** **** **** 5678'})
... ).delay()
```

Warning:

Sensitive information will still be accessible to anyone able to read your task message from the broker, or otherwise able intercept it.

For this reason you should probably encrypt your message if it contains sensitive information, or in this example with a credit card number the actual number could be stored encrypted in a secure store that you retrieve and decrypt in the task itself.

Retrying

`app.Task.retry()` can be used to re-execute the task, for example in the event of recoverable errors.

When you call `retry` it'll send a new message, using the same task-id, and it'll take care to make sure the message is delivered to the same queue as the originating task.

When a task is retried this is also recorded as a task state, so that you can track the progress of the task using the result instance (see States).

Here's an example using `retry`:

```
@app.task(bind=True)
def send_twitter_status(self, oauth, tweet):
    try:
        twitter = Twitter(oauth)
        twitter.update_status(tweet)
    except (Twitter.FailWhaleError, Twitter.LoginError) as exc:
        raise self.retry(exc=exc)
```

Note:

The **`app.Task.retry()`** call will raise an exception so any code after the retry won't be reached. This is the **`Retry`** exception, it isn't handled as an error but rather as a semi-predicate to signify to the worker that the task is to be retried, so that it can store the correct state when a result backend is enabled.

This is normal operation and always happens unless the `throw` argument to retry is set to **`False`**.

The bind argument to the task decorator will give access to `self` (the task type instance).

The `exc` method is used to pass exception information that's used in logs, and when storing task results. Both the exception and the traceback will be available in the task state (if a result backend is enabled).

If the task has a `max_retries` value the current exception will be re-raised if the max number of retries has been exceeded, but this won't happen if:

- An `exc` argument wasn't given.

  In this case the **`MaxRetriesExceededError`** exception will be raised.

- There's no current exception

  If there's no original exception to re-raise the `exc` argument will be used instead, so:

  ```
  self.retry(exc=Twitter.LoginError())
  ```

  will raise the `exc` argument given.

Using a custom retry delay

When a task is to be retried, it can wait for a given amount of time before doing so, and the default delay is defined by the **`default_retry_delay`** attribute. By default this is set to 3 minutes. Note that the unit for setting the delay is in seconds (int or float).

You can also provide the *countdown* argument to **`retry()`** to override this default.

```
@app.task(bind=True, default_retry_delay=30 * 60)  # retry in 30 minutes
def add(self, x, y):
    try:
        something_raising()
    except Exception as exc:
        # overrides the default delay to retry after 1 minute
        raise self.retry(exc=exc, countdown=60)
```

Automatic retry for known exceptions

New in version 4.0.

Sometimes you just want to retry a task whenever a particular exception is raised.

Fortunately, you can tell Celery to automatically retry a task using *autoretry_for* argument in the **task()** decorator:

```
from twitter.exceptions import FailWhaleError

@app.task(autoretry_for=(FailWhaleError,))
def refresh_timeline(user):
    return twitter.refresh_timeline(user)
```

If you want to specify custom arguments for an internal **retry()** call, pass *retry_kwargs* argument to **task()** decorator:

```
@app.task(autoretry_for=(FailWhaleError,),
          retry_kwargs={'max_retries': 5})
def refresh_timeline(user):
    return twitter.refresh_timeline(user)
```

This is provided as an alternative to manually handling the exceptions, and the example above will do the same as wrapping the task body in a **try** … **except** statement:

```
@app.task
def refresh_timeline(user):
    try:
        twitter.refresh_timeline(user)
    except FailWhaleError as exc:
        raise div.retry(exc=exc, max_retries=5)
```

If you want to automatically retry on any error, simply use:

```
@app.task(autoretry_for=(Exception,))
def x():
    ...
```

New in version 4.2.

If your tasks depend on another service, like making a request to an API, then it's a good idea to use exponential backoff to avoid overwhelming the service with your requests. Fortunately, Celery's automatic retry support makes it easy. Just specify the **retry_backoff** argument, like this:

```
from requests.exceptions import RequestException

@app.task(autoretry_for=(RequestException,), retry_backoff=True)
def x():
    ...
```

By default, this exponential backoff will also introduce random jitter to avoid having all the tasks run at the same moment. It will also cap the maximum backoff delay to 10 minutes. All these settings can be customized via options documented below.

`Task.`**autoretry_for**

A list/tuple of exception classes. If any of these exceptions are raised during the execution of the task, the task will automatically be retried. By default, no exceptions will be autoretried.

`Task.`**retry_kwargs**

A dictionary. Use this to customize how autoretries are executed. Note that if you use the exponential backoff options below, the *countdown* task option will be determined by Celery's autoretry system, and any *countdown* included in this

dictionary will be ignored.

`Task.`**`retry_backoff`**
: A boolean, or a number. If this option is set to `True`, autoretries will be delayed following the rules of exponential backoff. The first retry will have a delay of 1 second, the second retry will have a delay of 2 seconds, the third will delay 4 seconds, the fourth will delay 8 seconds, and so on. (However, this delay value is modified by **`retry_jitter`**, if it is enabled.) If this option is set to a number, it is used as a delay factor. For example, if this option is set to `3`, the first retry will delay 3 seconds, the second will delay 6 seconds, the third will delay 12 seconds, the fourth will delay 24 seconds, and so on. By default, this option is set to `False`, and autoretries will not be delayed.

`Task.`**`retry_backoff_max`**
: A number. If `retry_backoff` is enabled, this option will set a maximum delay in seconds between task autoretries. By default, this option is set to `600`, which is 10 minutes.

`Task.`**`retry_jitter`**
: A boolean. Jitter is used to introduce randomness into exponential backoff delays, to prevent all tasks in the queue from being executed simultaneously. If this option is set to `True`, the delay value calculated by **`retry_backoff`** is treated as a maximum, and the actual delay value will be a random number between zero and that maximum. By default, this option is set to `True`.

List of Options

The task decorator can take a number of options that change the way the task behaves, for example you can set the rate limit for a task using the **`rate_limit`** option.

Any keyword argument passed to the task decorator will actually be set as an attribute of the resulting task class, and this is a list of the built-in attributes.

General

`Task.`**`name`**
: The name the task is registered as.

  You can set this name manually, or a name will be automatically generated using the module and class name.

  See also Names.

`Task.`**`request`**
: If the task is being executed this will contain information about the current request. Thread local storage is used.

  See Task Request.

`Task.`**`max_retries`**
: Only applies if the task calls `self.retry` or if the task is decorated with the autoretry_for argument.

  The maximum number of attempted retries before giving up. If the number of retries exceeds this value a **`MaxRetriesExceededError`** exception will be raised.

> **Note:**
> You have to call **`retry()`** manually, as it won't automatically retry on exception..

The default is 3. A value of **None** will disable the retry limit and the task will retry forever until it succeeds.

Task.throws

Optional tuple of expected error classes that shouldn't be regarded as an actual error.

Errors in this list will be reported as a failure to the result backend, but the worker won't log the event as an error, and no traceback will be included.

Example:

```
@task(throws=(KeyError, HttpNotFound)):
def get_foo():
    something()
```

Error types:

- Expected errors (in `Task.throws`)

  Logged with severity `INFO`, traceback excluded.

- Unexpected errors

  Logged with severity `ERROR`, with traceback included.

Task.default_retry_delay

Default time in seconds before a retry of the task should be executed. Can be either **int** or **float**. Default is a three minute delay.

Task.rate_limit

Set the rate limit for this task type (limits the number of tasks that can be run in a given time frame). Tasks will still complete when a rate limit is in effect, but it may take some time before it's allowed to start.

If this is **None** no rate limit is in effect. If it is an integer or float, it is interpreted as "tasks per second".

The rate limits can be specified in seconds, minutes or hours by appending *"/s"*, *"/m"* or *"/h"* to the value. Tasks will be evenly distributed over the specified time frame.

Example: *"100/m"* (hundred tasks a minute). This will enforce a minimum delay of 600ms between starting two tasks on the same worker instance.

Default is the **task_default_rate_limit** setting: if not specified means rate limiting for tasks is disabled by default.

Note that this is a *per worker instance* rate limit, and not a global rate limit. To enforce a global rate limit (e.g., for an API with a maximum number of requests per second), you must restrict to a given queue.

Task.time_limit

The hard time limit, in seconds, for this task. When not set the workers default is used.

Task.soft_time_limit

The soft time limit for this task. When not set the workers default is used.

Task.ignore_result

Don't store task state. Note that this means you can't use **AsyncResult** to check if the task is ready, or get its return value.

Task.store_errors_even_if_ignored

If **True**, errors will be stored even if the task is configured to ignore results.

Task.serializer

A string identifying the default serialization method to use. Defaults to the **task_serializer** setting. Can be *pickle*, *json*, *yaml*, or any custom serialization methods that have been registered with **kombu.serialization.registry**.

Please see Serializers for more information.

`Task.`**compression**

A string identifying the default compression scheme to use.

Defaults to the **task_compression** setting. Can be *gzip*, or *bzip2*, or any custom compression schemes that have been registered with the **kombu.compression** registry.

Please see Compression for more information.

`Task.`**backend**

The result store backend to use for this task. An instance of one of the backend classes in *celery.backends*. Defaults to *app.backend*, defined by the **result_backend** setting.

`Task.`**acks_late**

If set to **True** messages for this task will be acknowledged **after** the task has been executed, not *just before* (the default behavior).

Note: This means the task may be executed multiple times should the worker crash in the middle of execution. Make sure your tasks are idempotent.

The global default can be overridden by the **task_acks_late** setting.

`Task.`**track_started**

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" status can be useful for when there are long running tasks and there's a need to report what task is currently running.

The host name and process id of the worker executing the task will be available in the state meta-data (e.g., *result.info['pid']*)

The global default can be overridden by the **task_track_started** setting.

See also:

The API reference for **Task**.

States

Celery can keep track of the tasks current state. The state also contains the result of a successful task, or the exception and traceback information of a failed task.

There are several *result backends* to choose from, and they all have different strengths and weaknesses (see Result Backends).

During its lifetime a task will transition through several possible states, and each state may have arbitrary meta-data attached to it. When a task moves into a new state the previous state is forgotten about, but some transitions can be deducted, (e.g., a task now in the **FAILED** state, is implied to have been in the **STARTED** state at some point).

There are also sets of states, like the set of **FAILURE_STATES**, and the set of **READY_STATES**.

The client uses the membership of these sets to decide whether the exception should be re-raised (**`PROPAGATE_STATES`**), or whether the state can be cached (it can if the task is ready).

You can also define Custom states.

Result Backends

If you want to keep track of tasks or need the return values, then Celery must store or send the states somewhere so that they can be retrieved later. There are several built-in result backends to choose from: SQLAlchemy/Django ORM, Memcached, RabbitMQ/QPid (`rpc`), and Redis – or you can define your own.

No backend works well for every use case. You should read about the strengths and weaknesses of each backend, and choose the most appropriate for your needs.

Warning:

Backends use resources to store and transmit results. To ensure that resources are released, you must eventually call **`get()`** or **`forget()`** on EVERY **`AsyncResult`** instance returned after calling a task.

See also:

Task result backend settings

RPC Result Backend (RabbitMQ/QPid)

The RPC result backend (*rpc://*) is special as it doesn't actually *store* the states, but rather sends them as messages. This is an important difference as it means that a result *can only be retrieved once*, and *only by the client that initiated the task*. Two different processes can't wait for the same result.

Even with that limitation, it is an excellent choice if you need to receive state changes in real-time. Using messaging means the client doesn't have to poll for new states.

The messages are transient (non-persistent) by default, so the results will disappear if the broker restarts. You can configure the result backend to send persistent messages using the **`result_persistent`** setting.

Database Result Backend

Keeping state in the database can be convenient for many, especially for web applications with a database already in place, but it also comes with limitations.

- Polling the database for new states is expensive, and so you should increase the polling intervals of operations, such as *result.get()*.
- Some databases use a default transaction isolation level that isn't suitable for polling tables for changes.

  In MySQL the default transaction isolation level is *REPEATABLE-READ*: meaning the transaction won't see changes made by other transactions until the current transaction is committed.

  Changing that to the *READ-COMMITTED* isolation level is recommended.

Built-in States

PENDING

Task is waiting for execution or unknown. Any task id that's not known is implied to be in the pending state.

STARTED

Task has been started. Not reported by default, to enable please see **`app.Task.track_started`**.

meta-data: *pid* and *hostname* of the worker process executing the task.

SUCCESS

Task has been successfully executed.

meta-data: *result* contains the return value of the task.
propagates: Yes
ready: Yes

FAILURE

Task execution resulted in failure.

meta-data: *result* contains the exception occurred, and *traceback* contains the backtrace of the stack at the point when the exception was raised.
propagates: Yes

RETRY

Task is being retried.

meta-data: *result* contains the exception that caused the retry, and *traceback* contains the backtrace of the stack at the point when the exceptions was raised.
propagates: No

REVOKED

Task has been revoked.

propagates: Yes

Custom states

You can easily define your own states, all you need is a unique name. The name of the state is usually an uppercase string. As an example you could have a look at the **`abortable tasks`** which defines a custom **`ABORTED`** state.

Use **`update_state()`** to update a task's state:.

```
@app.task(bind=True)
def upload_files(self, filenames):
    for i, file in enumerate(filenames):
        if not self.request.called_directly:
            self.update_state(state='PROGRESS',
                meta={'current': i, 'total': len(filenames)})
```

Here I created the state *"PROGRESS"*, telling any application aware of this state that the task is currently in progress, and also where it is in the process by having *current* and *total* counts as part of the state meta-data. This can then be used to create progress bars for

example.

Creating pickleable exceptions

A rarely known Python fact is that exceptions must conform to some simple rules to support being serialized by the pickle module.

Tasks that raise exceptions that aren't pickleable won't work properly when Pickle is used as the serializer.

To make sure that your exceptions are pickleable the exception *MUST* provide the original arguments it was instantiated with in its `.args` attribute. The simplest way to ensure this is to have the exception call `Exception.__init__`.

Let's look at some examples that work, and one that doesn't:

```
# OK:
class HttpError(Exception):
    pass

# BAD:
class HttpError(Exception):

    def __init__(self, status_code):
        self.status_code = status_code

# OK:
class HttpError(Exception):

    def __init__(self, status_code):
        self.status_code = status_code
        Exception.__init__(self, status_code)  # <-- REQUIRED
```

So the rule is: For any exception that supports custom arguments `*args`, `Exception.__init__(self, *args)` must be used.

There's no special support for *keyword arguments*, so if you want to preserve keyword arguments when the exception is unpickled you have to pass them as regular args:

```
class HttpError(Exception):

    def __init__(self, status_code, headers=None, body=None):
        self.status_code = status_code
        self.headers = headers
        self.body = body

        super(HttpError, self).__init__(status_code, headers, body)
```

Semipredicates

The worker wraps the task in a tracing function that records the final state of the task. There are a number of exceptions that can be used to signal this function to change how it treats the return of the task.

Ignore

The task may raise **`Ignore`** to force the worker to ignore the task. This means that no state will be recorded for the task, but the message is still acknowledged (removed from queue).

This can be used if you want to implement custom revoke-like functionality, or manually store the result of a task.

Example keeping revoked tasks in a Redis set:

```
from celery.exceptions import Ignore

@app.task(bind=True)
def some_task(self):
    if redis.ismember('tasks.revoked', self.request.id):
        raise Ignore()
```

Example that stores results manually:

```
from celery import states
from celery.exceptions import Ignore

@app.task(bind=True)
def get_tweets(self, user):
    timeline = twitter.get_timeline(user)
    if not self.request.called_directly:
        self.update_state(state=states.SUCCESS, meta=timeline)
    raise Ignore()
```

Reject

The task may raise **`Reject`** to reject the task message using AMQPs `basic_reject` method. This won't have any effect unless **`Task.acks_late`** is enabled.

Rejecting a message has the same effect as acking it, but some brokers may implement additional functionality that can be used. For example RabbitMQ supports the concept of Dead Letter Exchanges where a queue can be configured to use a dead letter exchange that rejected messages are redelivered to.

Reject can also be used to re-queue messages, but please be very careful when using this as it can easily result in an infinite message loop.

Example using reject when a task causes an out of memory condition:

```
import errno
from celery.exceptions import Reject

@app.task(bind=True, acks_late=True)
def render_scene(self, path):
    file = get_file(path)
    try:
        renderer.render_scene(file)

    # if the file is too big to fit in memory
    # we reject it so that it's redelivered to the dead letter exchange
    # and we can manually inspect the situation.
    except MemoryError as exc:
        raise Reject(exc, requeue=False)
    except OSError as exc:
        if exc.errno == errno.ENOMEM:
            raise Reject(exc, requeue=False)

    # For any other error we retry after 10 seconds.
    except Exception as exc:
        raise self.retry(exc, countdown=10)
```

Example re-queuing the message:

```python
from celery.exceptions import Reject

@app.task(bind=True, acks_late=True)
def requeues(self):
    if not self.request.delivery_info['redelivered']:
        raise Reject('no reason', requeue=True)
    print('received two times')
```

Consult your broker documentation for more details about the `basic_reject` method.

Retry

The `Retry` exception is raised by the `Task.retry` method to tell the worker that the task is being retried.

Custom task classes

All tasks inherit from the `app.Task` class. The `run()` method becomes the task body.

As an example, the following code,

```python
@app.task
def add(x, y):
    return x + y
```

will do roughly this behind the scenes:

```python
class _AddTask(app.Task):

    def run(self, x, y):
        return x + y
add = app.tasks[_AddTask.name]
```

Instantiation

A task is **not** instantiated for every request, but is registered in the task registry as a global instance.

This means that the `__init__` constructor will only be called once per process, and that the task class is semantically closer to an Actor.

If you have a task,

```python
from celery import Task

class NaiveAuthenticateServer(Task):

    def __init__(self):
        self.users = {'george': 'password'}

    def run(self, username, password):
        try:
            return self.users[username] == password
        except KeyError:
            return False
```

And you route every request to the same process, then it will keep state between requests.

This can also be useful to cache resources, For example, a base Task class that caches a database connection:

```python
from celery import Task

class DatabaseTask(Task):
    _db = None

    @property
    def db(self):
        if self._db is None:
            self._db = Database.connect()
        return self._db
```

that can be added to tasks like this:

```python
@app.task(base=DatabaseTask)
def process_rows():
    for row in process_rows.db.table.all():
        process_row(row)
```

The `db` attribute of the `process_rows` task will then always stay the same in each process.

Handlers

after_return(*self*, *status*, *retval*, *task_id*, *args*, *kwargs*, *einfo*)

Handler called after the task returns.

Parameters:
- **status** – Current task state.
- **retval** – Task return value/exception.
- **task_id** – Unique id of the task.
- **args** – Original arguments for the task that returned.
- **kwargs** – Original keyword arguments for the task that returned.

Keyword Arguments:
- **einfo** – **ExceptionInfo** instance, containing the traceback (if any).

The return value of this handler is ignored.

on_failure(*self*, *exc*, *task_id*, *args*, *kwargs*, *einfo*)

This is run by the worker when the task fails.

Parameters:
- **exc** – The exception raised by the task.
- **task_id** – Unique id of the failed task.
- **args** – Original arguments for the task that failed.
- **kwargs** – Original keyword arguments for the task that failed.

Keyword Arguments:
- **einfo** – **ExceptionInfo** instance, containing the traceback.

The return value of this handler is ignored.

on_retry(*self*, *exc*, *task_id*, *args*, *kwargs*, *einfo*)

This is run by the worker when the task is to be retried.

Parameters:
- **exc** – The exception sent to **retry()**.
- **task_id** – Unique id of the retried task.
- **args** – Original arguments for the retried task.
- **kwargs** – Original keyword arguments for the retried task.

Keyword Arguments:
- **einfo** – **ExceptionInfo** instance, containing the traceback.

The return value of this handler is ignored.

on_success(*self*, *retval*, *task_id*, *args*, *kwargs*)

Run by the worker if the task executes successfully.

Parameters:
- **retval** – The return value of the task.
- **task_id** – Unique id of the executed task.
- **args** – Original arguments for the executed task.
- **kwargs** – Original keyword arguments for the executed task.

The return value of this handler is ignored.

Requests and custom requests

Upon receiving a message to run a task, the worker creates a **request** to represent such demand.

Custom task classes may override which request class to use by changing the attribute **celery.app.task.Task.Request**. You may either assign the custom request class itself, or its fully qualified name.

The request has several responsibilities. Custom request classes should cover them all – they are responsible to actually run and trace the task. We strongly recommend to inherit from **celery.worker.request.Request**.

When using the pre-forking worker, the methods **on_timeout()** and **on_failure()** are executed in the main worker process. An application may leverage such facility to detect failures which are not detected using **celery.app.task.Task.on_failure()**.

As an example, the following custom request detects and logs hard time limits, and other failures.

```
import logging
from celery.worker.request import Request

logger = logging.getLogger('my.package')

class MyRequest(Request):
    'A minimal custom request to log failures and hard time limits.'

    def on_timeout(self, soft, timeout):
        super(MyRequest, self).on_timeout(soft, timeout)
        if not soft:
           logger.warning(
               'A hard timeout was enforced for task %s',
               self.task.name
           )

    def on_failure(self, exc_info, send_failed_event=True, return_ok=Fal
        super(Request, self).on_failure(
            exc_info,
            send_failed_event=send_failed_event,
            return_ok=return_ok
        )
        logger.warning(
            'Failure detected for task %s',
            self.task.name
        )

class MyTask(Task):
    Request = MyRequest  # you can use a FQN 'my.package:MyRequest'

@app.task(base=MyTask)
def some_longrunning_task():
    # use your imagination
```

How it works

Here come the technical details. This part isn't something you need to know, but you may be interested.

All defined tasks are listed in a registry. The registry contains a list of task names and their task classes. You can investigate this registry yourself:

```
>>> from proj.celery import app
>>> app.tasks
{'celery.chord_unlock':
    <@task: celery.chord_unlock>,
 'celery.backend_cleanup':
    <@task: celery.backend_cleanup>,
 'celery.chord':
    <@task: celery.chord>}
```

This is the list of tasks built-in to Celery. Note that tasks will only be registered when the module they're defined in is imported.

The default loader imports any modules listed in the **`imports`** setting.

The **`app.task()`** decorator is responsible for registering your task in the applications task registry.

When tasks are sent, no actual function code is sent with it, just the name of the task to execute. When the worker then receives the message it can look up the name in its task registry to find the execution code.

This means that your workers should always be updated with the same software as the client. This is a drawback, but the alternative is a technical challenge that's yet to be solved.

Tips and Best Practices

Ignore results you don't want

If you don't care about the results of a task, be sure to set the **`ignore_result`** option, as storing results wastes time and resources.

```
@app.task(ignore_result=True)
def mytask():
    something()
```

Results can even be disabled globally using the **`task_ignore_result`** setting.

Results can be enabled/disabled on a per-execution basis, by passing the `ignore_result` boolean parameter, when calling `apply_async` or `delay`.

```
@app.task
def mytask(x, y):
    return x + y

# No result will be stored
result = mytask.apply_async(1, 2, ignore_result=True)
print result.get() # -> None

# Result will be stored
result = mytask.apply_async(1, 2, ignore_result=False)
print result.get() # -> 3
```

By default tasks will *not ignore results* (`ignore_result=False`) when a result backend is configured.

The option precedence order is the following:

1. Global `task_ignore_result`
2. `ignore_result` option
3. Task execution option `ignore_result`

More optimization tips

You find additional optimization tips in the Optimizing Guide.

Avoid launching synchronous subtasks

Having a task wait for the result of another task is really inefficient, and may even cause a deadlock if the worker pool is exhausted.

Make your design asynchronous instead, for example by using *callbacks*.

Bad:

```
@app.task
def update_page_info(url):
    page = fetch_page.delay(url).get()
    info = parse_page.delay(url, page).get()
    store_page_info.delay(url, info)

@app.task
def fetch_page(url):
    return myhttplib.get(url)

@app.task
def parse_page(url, page):
    return myparser.parse_document(page)

@app.task
def store_page_info(url, info):
    return PageInfo.objects.create(url, info)
```

Good:

```
def update_page_info(url):
    # fetch_page -> parse_page -> store_page
    chain = fetch_page.s(url) | parse_page.s() | store_page_info.s(url)
    chain()

@app.task()
def fetch_page(url):
    return myhttplib.get(url)

@app.task()
def parse_page(page):
    return myparser.parse_document(page)

@app.task(ignore_result=True)
def store_page_info(info, url):
    PageInfo.objects.create(url=url, info=info)
```

Here I instead created a chain of tasks by linking together different `signature()`'s. You can read about chains and other powerful constructs at Canvas: Designing Work-flows.

By default celery will not enable you to run tasks within task synchronously in rare or extreme cases you might have to do so. **WARNING**: enabling subtasks run synchronously is not recommended!

```
@app.task
def update_page_info(url):
    page = fetch_page.delay(url).get(disable_sync_subtasks=False)
    info = parse_page.delay(url, page).get(disable_sync_subtasks=False)
    store_page_info.delay(url, info)

@app.task
def fetch_page(url):
    return myhttplib.get(url)

@app.task
def parse_page(url, page):
    return myparser.parse_document(page)

@app.task
def store_page_info(url, info):
    return PageInfo.objects.create(url, info)
```

Performance and Strategies

Granularity

The task granularity is the amount of computation needed by each subtask. In general it is better to split the problem up into many small tasks rather than have a few long running tasks.

With smaller tasks you can process more tasks in parallel and the tasks won't run long enough to block the worker from processing other waiting tasks.

However, executing a task does have overhead. A message needs to be sent, data may not be local, etc. So if the tasks are too fine-grained the overhead added probably removes any benefit.

See also:

The book Art of Concurrency has a section dedicated to the topic of task granularity [AOC1].

[AOC1]	Breshears, Clay. Section 2.2.1, "The Art of Concurrency". O'Reilly Media, Inc. May 15, 2009. ISBN-13 978-0-596-52153-0.

Data locality

The worker processing the task should be as close to the data as possible. The best would be to have a copy in memory, the worst would be a full transfer from another continent.

If the data is far away, you could try to run another worker at location, or if that's not possible - cache often used data, or preload data you know is going to be used.

The easiest way to share data between workers is to use a distributed cache system, like memcached.

See also:

The paper Distributed Computing Economics by Jim Gray is an excellent introduction to the topic of data locality.

State

Since celery is a distributed system, you can't know which process, or on what machine the task will be executed. You can't even know if the task will run in a timely manner.

The ancient async sayings tells us that "asserting the world is the responsibility of the task". What this means is that the world view may have changed since the task was requested, so the task is responsible for making sure the world is how it should be; If you have a task that re-indexes a search engine, and the search engine should only be re-indexed at maximum every 5 minutes, then it must be the tasks responsibility to assert that, not the callers.

Another gotcha is Django model objects. They shouldn't be passed on as arguments to tasks. It's almost always better to re-fetch the object from the database when the task is running instead, as using old data may lead to race conditions.

Imagine the following scenario where you have an article and a task that automatically expands some abbreviations in it:

```
class Article(models.Model):
    title = models.CharField()
    body = models.TextField()

@app.task
def expand_abbreviations(article):
    article.body.replace('MyCorp', 'My Corporation')
    article.save()
```

First, an author creates an article and saves it, then the author clicks on a button that initiates the abbreviation task:

```
>>> article = Article.objects.get(id=102)
>>> expand_abbreviations.delay(article)
```

Now, the queue is very busy, so the task won't be run for another 2 minutes. In the meantime another author makes changes to the article, so when the task is finally run, the body of the article is reverted to the old version because the task had the old body in its argument.

Fixing the race condition is easy, just use the article id instead, and re-fetch the article in the task body:

```
@app.task
def expand_abbreviations(article_id):
    article = Article.objects.get(id=article_id)
    article.body.replace('MyCorp', 'My Corporation')
    article.save()
```

```
>>> expand_abbreviations.delay(article_id)
```

There might even be performance benefits to this approach, as sending large messages may be expensive.

Database transactions

Let's have a look at another example:

```
from django.db import transaction

@transaction.commit_on_success
def create_article(request):
    article = Article.objects.create()
    expand_abbreviations.delay(article.pk)
```

This is a Django view creating an article object in the database, then passing the primary key to a task. It uses the *commit_on_success* decorator, that will commit the transaction when the view returns, or roll back if the view raises an exception.

There's a race condition if the task starts executing before the transaction has been committed; The database object doesn't exist yet!

The solution is to use the `on_commit` callback to launch your celery task once all transactions have been committed successfully.

```
from django.db.transaction import on_commit

def create_article(request):
    article = Article.objects.create()
    on_commit(lambda: expand_abbreviations.delay(article.pk))
```

Note:

`on_commit` is available in Django 1.9 and above, if you are using a version prior to that then the django-transaction-hooks library adds support for this.

Example

Let's take a real world example: a blog where comments posted need to be filtered for spam. When the comment is created, the spam filter runs in the background, so the user doesn't have to wait for it to finish.

I have a Django blog application allowing comments on blog posts. I'll describe parts of the models/views and tasks for this application.

`blog/models.py`

The comment model looks like this:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _


class Comment(models.Model):
    name = models.CharField(_('name'), max_length=64)
    email_address = models.EmailField(_('email address'))
    homepage = models.URLField(_('home page'),
                               blank=True, verify_exists=False)
    comment = models.TextField(_('comment'))
    pub_date = models.DateTimeField(_('Published date'),
                                    editable=False, auto_add_now=True)
    is_spam = models.BooleanField(_('spam?'),
                                  default=False, editable=False)

    class Meta:
        verbose_name = _('comment')
        verbose_name_plural = _('comments')
```

In the view where the comment is posted, I first write the comment to the database, then I launch the spam filter task in the background.

blog/views.py

```
from django import forms
from django.http import HttpResponseRedirect
from django.template.context import RequestContext
from django.shortcuts import get_object_or_404, render_to_response

from blog import tasks
from blog.models import Comment


class CommentForm(forms.ModelForm):

    class Meta:
        model = Comment


def add_comment(request, slug, template_name='comments/create.html'):
    post = get_object_or_404(Entry, slug=slug)
    remote_addr = request.META.get('REMOTE_ADDR')

    if request.method == 'post':
        form = CommentForm(request.POST, request.FILES)
        if form.is_valid():
            comment = form.save()
            # Check spam asynchronously.
            tasks.spam_filter.delay(comment_id=comment.id,
                                    remote_addr=remote_addr)
            return HttpResponseRedirect(post.get_absolute_url())
    else:
        form = CommentForm()

    context = RequestContext(request, {'form': form})
    return render_to_response(template_name, context_instance=context)
```

To filter spam in comments I use Akismet, the service used to filter spam in comments posted to the free blog platform *Wordpress*. Akismet is free for personal use, but for commercial use you need to pay. You have to sign up to their service to get an API key.

To make API calls to Akismet I use the akismet.py library written by Michael Foord.

blog/tasks.py

```python
from celery import Celery

from akismet import Akismet

from django.core.exceptions import ImproperlyConfigured
from django.contrib.sites.models import Site

from blog.models import Comment


app = Celery(broker='amqp://')


@app.task
def spam_filter(comment_id, remote_addr=None):
    logger = spam_filter.get_logger()
    logger.info('Running spam filter for comment %s', comment_id)

    comment = Comment.objects.get(pk=comment_id)
    current_domain = Site.objects.get_current().domain
    akismet = Akismet(settings.AKISMET_KEY, 'http://{0}'.format(domain))
    if not akismet.verify_key():
        raise ImproperlyConfigured('Invalid AKISMET_KEY')


    is_spam = akismet.comment_check(user_ip=remote_addr,
                        comment_content=comment.comment,
                        comment_author=comment.name,
                        comment_author_email=comment.email_address)
    if is_spam:
        comment.is_spam = True
        comment.save()

    return is_spam
```