Previous topic

First Steps with Celery

Next topic

Resources

This Page

Show Source

Quick search

Go

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

Next Steps

The First Steps with Celery guide is intentionally minimal. In this guide I'll demonstrate what Celery offers in more detail, including how to add Celery support for your application and library.

This document doesn't document all of Celery's features and best practices, so it's recommended that you also read the User Guide

Using Celery in your Application

Our Project

Project layout:

```
proj/__init__.py
    /celery.py
    /tasks.py
```

proj/celery.py

```
from __future__ import absolute_import, unicode_literals
from celery import Celery

app = Celery('proj',
             broker='amqp://',
             backend='amqp://',
             include=['proj.tasks'])

# Optional configuration, see the application user guide.
app.conf.update(
    result_expires=3600,
)

if __name__ == '__main__':
    app.start()
```

In this module you created our **Celery** instance (sometimes referred to as the *app*). To use Celery within your project you simply import this instance.

- The `broker` argument specifies the URL of the broker to use.

  See Choosing a Broker for more information.

- The `backend` argument specifies the result backend to use,

  It's used to keep track of task state and results. While results are disabled by default I use the RPC result backend here because I demonstrate how retrieving results work later, you may want to use a different backend for your application. They all have different strengths and weaknesses. If you don't need results it's better to disable them. Results can also be disabled for individual tasks by setting the `@task(ignore_result=True)` option.

  See Keeping Results for more information.

- The `include` argument is a list of modules to import when the worker starts. You need to add our tasks module here so that the worker is able to find our tasks.

proj/tasks.py

```
from __future__ import absolute_import, unicode_literals
from .celery import app


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


@app.task
def mul(x, y):
    return x * y


@app.task
def xsum(numbers):
    return sum(numbers)
```

Starting the worker

The **celery** program can be used to start the worker (you need to run the worker in the directory above proj):

```
$ celery -A proj worker -l info
```

When the worker starts you should see a banner and some messages:

```
-------------- celery@halcyon.local v4.0 (latentcall)
---- **** -----
--- * ***  * -- [Configuration]
-- * - **** --- . broker:      amqp://guest@localhost:5672//
- ** ---------- . app:         __main__:0x1012d8590
- ** ---------- . concurrency: 8 (processes)
- ** ---------- . events:      OFF (enable -E to monitor this worker)
- ** ----------
- *** --- * --- [Queues]
-- ******* ---- . celery:      exchange:celery(direct) binding:celery
--- ***** -----

[2012-06-08 16:23:51,078: WARNING/MainProcess] celery@halcyon.local has
```

– The *broker* is the URL you specified in the broker argument in our `celery` module, you can also specify a different broker on the command-line by using the `-b` option.

– *Concurrency* is the number of prefork worker process used to process your tasks concurrently, when all of these are busy doing work new tasks will have to wait for one of the tasks to finish before it can be processed.

The default concurrency number is the number of CPU's on that machine (including cores), you can specify a custom number using the **`celery worker -c`** option. There's no recommended value, as the optimal number depends on a number of factors, but if your tasks are mostly I/O-bound then you can try to increase it, experimentation has shown that adding more than twice the number of CPU's is rarely effective, and likely to degrade performance instead.

Including the default prefork pool, Celery also supports using Eventlet, Gevent, and running in a single thread (see Concurrency).

– *Events* is an option that when enabled causes Celery to send monitoring messages (events) for actions occurring in the worker. These can be used by monitor programs like `celery events`, and Flower - the real-time Celery monitor, that you can read about in the Monitoring and Management guide.

– *Queues* is the list of queues that the worker will consume tasks from. The worker can be told to consume from several queues at once, and this is used to route messages to specific workers as a means for Quality of Service, separation of concerns, and prioritization, all described in the Routing Guide.

You can get a complete list of command-line arguments by passing in the **`--help`** flag:

```
$ celery worker --help
```

These options are described in more detailed in the Workers Guide.

Stopping the worker

To stop the worker simply hit `Control-c`. A list of signals supported by the worker is detailed in the Workers Guide.

In the background

In production you'll want to run the worker in the background, this is described in detail in the daemonization tutorial.

The daemonization scripts uses the **celery multi** command to start one or more workers in the background:

```
$ celery multi start w1 -A proj -l info
celery multi v4.0.0 (latentcall)
> Starting nodes...
    > w1.halcyon.local: OK
```

You can restart it too:

```
$ celery  multi restart w1 -A proj -l info
celery multi v4.0.0 (latentcall)
> Stopping nodes...
    > w1.halcyon.local: TERM -> 64024
> Waiting for 1 node.....
    > w1.halcyon.local: OK
> Restarting node w1.halcyon.local: OK
celery multi v4.0.0 (latentcall)
> Stopping nodes...
    > w1.halcyon.local: TERM -> 64052
```

or stop it:

```
$ celery multi stop w1 -A proj -l info
```

The `stop` command is asynchronous so it won't wait for the worker to shutdown. You'll

probably want to use the `stopwait` command instead, this ensures all currently executing tasks are completed before exiting:

```
$ celery multi stopwait w1 -A proj -l info
```

Note:

celery multi doesn't store information about workers so you need to use the same command-line arguments when restarting. Only the same pidfile and logfile arguments must be used when stopping.

By default it'll create pid and log files in the current directory, to protect against multiple workers launching on top of each other you're encouraged to put these in a dedicated directory:

```
$ mkdir -p /var/run/celery
$ mkdir -p /var/log/celery
$ celery multi start w1 -A proj -l info --pidfile=/var/run/celery/%n.pid
                                        --logfile=/var/log/celery/%n%I.l
```

With the multi command you can start multiple workers, and there's a powerful command-line syntax to specify arguments for different workers too, for example:

```
$ celery multi start 10 -A proj -l info -Q:1-3 images,video -Q:4,5 data
    -Q default -L:4,5 debug
```

For more examples see the `multi` module in the API reference.

About the `--app` argument

The `--app` argument specifies the Celery app instance to use, it must be in the form of `module.path:attribute`

But it also supports a shortcut form If only a package name is specified, where it'll try to search for the app instance, in the following order:

With `--app=proj`:

1. an attribute named `proj.app`, or
2. an attribute named `proj.celery`, or
3. any attribute in the module `proj` where the value is a Celery application, or

If none of these are found it'll try a submodule named `proj.celery`:

4. an attribute named `proj.celery.app`, or
5. an attribute named `proj.celery.celery`, or
6. Any attribute in the module `proj.celery` where the value is a Celery application.

This scheme mimics the practices used in the documentation – that is, `proj:app` for a single contained module, and `proj.celery:app` for larger projects.

Calling Tasks

You can call a task using the `delay()` method:

```
>>> add.delay(2, 2)
```

This method is actually a star-argument shortcut to another method called

`apply_async()`:

```
>>> add.apply_async((2, 2))
```

The latter enables you to specify execution options like the time to run (countdown), the queue it should be sent to, and so on:

```
>>> add.apply_async((2, 2), queue='lopri', countdown=10)
```

In the above example the task will be sent to a queue named `lopri` and the task will execute, at the earliest, 10 seconds after the message was sent.

Applying the task directly will execute the task in the current process, so that no message is sent:

```
>>> add(2, 2)
4
```

These three methods - **`delay()`**, **`apply_async()`**, and applying (`__call__`), represents the Celery calling API, that's also used for signatures.

A more detailed overview of the Calling API can be found in the Calling User Guide.

Every task invocation will be given a unique identifier (an UUID), this is the task id.

The `delay` and `apply_async` methods return an **`AsyncResult`** instance, that can be used to keep track of the tasks execution state. But for this you need to enable a result backend so that the state can be stored somewhere.

Results are disabled by default because of the fact that there's no result backend that suits every application, so to choose one you need to consider the drawbacks of each individual backend. For many tasks keeping the return value isn't even very useful, so it's a sensible default to have. Also note that result backends aren't used for monitoring tasks and workers, for that Celery uses dedicated event messages (see Monitoring and Management Guide).

If you have a result backend configured you can retrieve the return value of a task:

```
>>> res = add.delay(2, 2)
>>> res.get(timeout=1)
4
```

You can find the task's id by looking at the **`id`** attribute:

```
>>> res.id
d6b3aea2-fb9b-4ebc-8da4-848818db9114
```

You can also inspect the exception and traceback if the task raised an exception, in fact `result.get()` will propagate any errors by default:

```
>>> res = add.delay(2)
>>> res.get(timeout=1)
```

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/opt/devel/celery/celery/result.py", line 113, in get
    interval=interval)
File "/opt/devel/celery/celery/backends/rpc.py", line 138, in wait_for
    raise meta['result']
TypeError: add() takes exactly 2 arguments (1 given)
```

If you don't wish for the errors to propagate then you can disable that by passing the

`propagate` argument:

```
>>> res.get(propagate=False)
TypeError('add() takes exactly 2 arguments (1 given)',)
```

In this case it'll return the exception instance raised instead, and so to check whether the task succeeded or failed you'll have to use the corresponding methods on the result instance:

```
>>> res.failed()
True

>>> res.successful()
False
```

So how does it know if the task has failed or not? It can find out by looking at the tasks *state*:

```
>>> res.state
'FAILURE'
```

A task can only be in a single state, but it can progress through several states. The stages of a typical task can be:

```
PENDING -> STARTED -> SUCCESS
```

The started state is a special state that's only recorded if the **`task_track_started`** setting is enabled, or if the `@task(track_started=True)` option is set for the task.

The pending state is actually not a recorded state, but rather the default state for any task id that's unknown: this you can see from this example:

```
>>> from proj.celery import app

>>> res = app.AsyncResult('this-id-does-not-exist')
>>> res.state
'PENDING'
```

If the task is retried the stages can become even more complex. To demonstrate, for a task that's retried two times the stages would be:

```
PENDING -> STARTED -> RETRY -> STARTED -> RETRY -> STARTED -> SUCCESS
```

To read more about task states you should see the States section in the tasks user guide.

Calling tasks is described in detail in the Calling Guide.

Canvas: Designing Work-flows

You just learned how to call a task using the tasks `delay` method, and this is often all you need, but sometimes you may want to pass the signature of a task invocation to another process or as an argument to another function, for this Celery uses something called *signatures*.

A signature wraps the arguments and execution options of a single task invocation in a way such that it can be passed to functions or even serialized and sent across the wire.

You can create a signature for the `add` task using the arguments `(2, 2)`, and a countdown of 10 seconds like this:

```
>>> add.signature((2, 2), countdown=10)
tasks.add(2, 2)
```

There's also a shortcut using star arguments:

```
>>> add.s(2, 2)
tasks.add(2, 2)
```

And there's that calling API again...

Signature instances also supports the calling API: meaning they have the `delay` and `apply_async` methods.

But there's a difference in that the signature may already have an argument signature specified. The `add` task takes two arguments, so a signature specifying two arguments would make a complete signature:

```
>>> s1 = add.s(2, 2)
>>> res = s1.delay()
>>> res.get()
4
```

But, you can also make incomplete signatures to create what we call *partials*:

```
# incomplete partial: add(?, 2)
>>> s2 = add.s(2)
```

`s2` is now a partial signature that needs another argument to be complete, and this can be resolved when calling the signature:

```
# resolves the partial: add(8, 2)
>>> res = s2.delay(8)
>>> res.get()
10
```

Here you added the argument 8 that was prepended to the existing argument 2 forming a complete signature of `add(8, 2)`.

Keyword arguments can also be added later, these are then merged with any existing keyword arguments, but with new arguments taking precedence:

```
>>> s3 = add.s(2, 2, debug=True)
>>> s3.delay(debug=False)   # debug is now False.
```

As stated signatures supports the calling API: meaning that;

- `sig.apply_async(args=(), kwargs={}, **options)`

  Calls the signature with optional partial arguments and partial keyword arguments. Also supports partial execution options.

- `sig.delay(*args, **kwargs)`

  Star argument version of `apply_async`. Any arguments will be prepended to the arguments in the signature, and keyword arguments is merged with any existing keys.

So this all seems very useful, but what can you actually do with these? To get to that I must introduce the canvas primitives…

The Primitives

- group
- chain
- chord
- map
- starmap
- chunks

These primitives are signature objects themselves, so they can be combined in any number of ways to compose complex work-flows.

Note:

These examples retrieve results, so to try them out you need to configure a result backend. The example project above already does that (see the backend argument to **`Celery`**).

Let's look at some examples:

Groups

A **`group`** calls a list of tasks in parallel, and it returns a special result instance that lets you inspect the results as a group, and retrieve the return values in order.

```
>>> from celery import group
>>> from proj.tasks import add

>>> group(add.s(i, i) for i in xrange(10))().get()
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

- Partial group

```
>>> g = group(add.s(i) for i in xrange(10))
>>> g(10).get()
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Chains

Tasks can be linked together so that after one task returns the other is called:

```
>>> from celery import chain
>>> from proj.tasks import add, mul

# (4 + 4) * 8
>>> chain(add.s(4, 4) | mul.s(8))().get()
64
```

or a partial chain:

```
>>> # (? + 4) * 8
>>> g = chain(add.s(4) | mul.s(8))
>>> g(4).get()
64
```

Chains can also be written like this:

```
>>> (add.s(4, 4) | mul.s(8))().get()
64
```

Chords

A chord is a group with a callback:

```
>>> from celery import chord
>>> from proj.tasks import add, xsum

>>> chord((add.s(i, i) for i in xrange(10)), xsum.s())().get()
90
```

A group chained to another task will be automatically converted to a chord:

```
>>> (group(add.s(i, i) for i in xrange(10)) | xsum.s())().get()
90
```

Since these primitives are all of the signature type they can be combined almost however you want, for example:

```
>>> upload_document.s(file) | group(apply_filter.s() for filter in filte
```

Be sure to read more about work-flows in the Canvas user guide.

Routing

Celery supports all of the routing facilities provided by AMQP, but it also supports simple routing where messages are sent to named queues.

The **`task_routes`** setting enables you to route tasks by name and keep everything centralized in one location:

```
app.conf.update(
    task_routes = {
        'proj.tasks.add': {'queue': 'hipri'},
    },
)
```

You can also specify the queue at runtime with the `queue` argument to `apply_async`:

```
>>> from proj.tasks import add
>>> add.apply_async((2, 2), queue='hipri')
```

You can then make a worker consume from this queue by specifying the **`celery worker -Q`** option:

```
$ celery -A proj worker -Q hipri
```

You may specify multiple queues by using a comma separated list, for example you can make the worker consume from both the default queue, and the `hipri` queue, where the default queue is named `celery` for historical reasons:

```
$ celery -A proj worker -Q hipri,celery
```

The order of the queues doesn't matter as the worker will give equal weight to the queues.

To learn more about routing, including taking use of the full power of AMQP routing, see the Routing Guide.

Remote Control

If you're using RabbitMQ (AMQP), Redis, or Qpid as the broker then you can control and inspect the worker at runtime.

For example you can see what tasks the worker is currently working on:

```
$ celery -A proj inspect active
```

This is implemented by using broadcast messaging, so all remote control commands are received by every worker in the cluster.

You can also specify one or more workers to act on the request using the `--destination` option. This is a comma separated list of worker host names:

```
$ celery -A proj inspect active --destination=celery@example.com
```

If a destination isn't provided then every worker will act and reply to the request.

The **celery inspect** command contains commands that doesn't change anything in the worker, it only replies information and statistics about what's going on inside the worker. For a list of inspect commands you can execute:

```
$ celery -A proj inspect --help
```

Then there's the **celery control** command, that contains commands that actually changes things in the worker at runtime:

```
$ celery -A proj control --help
```

For example you can force workers to enable event messages (used for monitoring tasks and workers):

```
$ celery -A proj control enable_events
```

When events are enabled you can then start the event dumper to see what the workers are doing:

```
$ celery -A proj events --dump
```

or you can start the curses interface:

```
$ celery -A proj events
```

when you're finished monitoring you can disable events again:

```
$ celery -A proj control disable_events
```

The **celery status** command also uses remote control commands and shows a list of online workers in the cluster:

```
$ celery -A proj status
```

You can read more about the **celery** command and monitoring in the Monitoring Guide.

Timezone

All times and dates, internally and in messages uses the UTC timezone.

When the worker receives a message, for example with a countdown set it converts that UTC time to local time. If you wish to use a different timezone than the system timezone then you must configure that using the `timezone` setting:

```
app.conf.timezone = 'Europe/London'
```

Optimization

The default configuration isn't optimized for throughput by default, it tries to walk the middle way between many short tasks and fewer long tasks, a compromise between throughput and fair scheduling.

If you have strict fair scheduling requirements, or want to optimize for throughput then you should read the Optimizing Guide.

If you're using RabbitMQ then you can install the librabbitmq module: this is an AMQP client implemented in C:

```
$ pip install librabbitmq
```

What to do now?

Now that you have read this document you should continue to the User Guide.

There's also an API reference if you're so inclined.