Previous topic

Using Amazon SQS

Next topic

Next Steps

This Page

Show Source

Quick search

Go

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

First Steps with Celery

Celery is a task queue with batteries included. It's easy to use so that you can get started without learning the full complexities of the problem it solves. It's designed around best practices so that your product can scale and integrate with other languages, and it comes with the tools and support you need to run such a system in production.

In this tutorial you'll learn the absolute basics of using Celery.

Learn about;

- Choosing and installing a message transport (broker).
- Installing Celery and creating your first task.
- Starting the worker and calling tasks.
- Keeping track of tasks as they transition through different states, and inspecting return values.

Celery may seem daunting at first - but don't worry - this tutorial will get you started in no time. It's deliberately kept simple, so as to not confuse you with advanced features. After you have finished this tutorial, it's a good idea to browse the rest of the documentation. For example the Next Steps tutorial will showcase Celery's capabilities.

Choosing a Broker

Celery requires a solution to send and receive messages; usually this comes in the form of a separate service called a *message broker*.

There are several choices available, including:

RabbitMQ

RabbitMQ is feature-complete, stable, durable and easy to install. It's an excellent choice for a production environment. Detailed information about using RabbitMQ with Celery:

Using RabbitMQ

If you're using Ubuntu or Debian install RabbitMQ by executing this command:

```
$ sudo apt-get install rabbitmq-server
```

When the command completes, the broker will already be running in the background, ready to move messages for you: `Starting rabbitmq-server: SUCCESS`.

Don't worry if you're not running Ubuntu or Debian, you can go to this website to find similarly simple installation instructions for other platforms, including Microsoft Windows:

http://www.rabbitmq.com/download.html

Redis

Redis is also feature-complete, but is more susceptible to data loss in the event of abrupt termination or power failures. Detailed information about using Redis:

Using Redis

Other brokers

In addition to the above, there are other experimental transport implementations to choose from, including Amazon SQS.

See Broker Overview for a full list.

Installing Celery

Celery is on the Python Package Index (PyPI), so it can be installed with standard Python tools like `pip` or `easy_install`:

```
$ pip install celery
```

Application

The first thing you need is a Celery instance. We call this the *Celery application* or just *app* for short. As this instance is used as the entry-point for everything you want to do in Celery, like creating tasks and managing workers, it must be possible for other modules to import it.

In this tutorial we keep everything contained in a single module, but for larger projects you want to create a dedicated module.

Let's create the file `tasks.py`:

```
from celery import Celery

app = Celery('tasks', broker='pyamqp://guest@localhost//')

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

The first argument to **`Celery`** is the name of the current module. This is only needed so that names can be automatically generated when the tasks are defined in the *__main__* module.

The second argument is the broker keyword argument, specifying the URL of the message broker you want to use. Here using RabbitMQ (also the default option).

See Choosing a Broker above for more choices – for RabbitMQ you can use `amqp://localhost`, or for Redis you can use `redis://localhost`.

You defined a single task, called `add`, returning the sum of two numbers.

Running the Celery worker server

You can now run the worker by executing our program with the `worker` argument:

```
$ celery -A tasks worker --loglevel=info
```

> **Note:**
>
> See the Troubleshooting section if the worker doesn't start.

In production you'll want to run the worker in the background as a daemon. To do this you need to use the tools provided by your platform, or something like supervisord (see Daemonization for more information).

For a complete listing of the command-line options available, do:

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

There are also several other commands available, and help is also available:

```
$ celery help
```

Calling the task

To call our task you can use the **`delay()`** method.

This is a handy shortcut to the **`apply_async()`** method that gives greater control of the task execution (see Calling Tasks):

```
>>> from tasks import add
>>> add.delay(4, 4)
```

The task has now been processed by the worker you started earlier. You can verify this by looking at the worker's console output.

Calling a task returns an **`AsyncResult`** instance. This can be used to check the state of the task, wait for the task to finish, or get its return value (or if the task failed, to get the exception and traceback).

Results are not enabled by default. In order to do remote procedure calls or keep track of task results in a database, you will need to configure Celery to use a result backend. This is described in the next section.

Keeping Results

If you want to keep track of the tasks' states, Celery needs to store or send the states somewhere. There are several built-in result backends to choose from: SQLAlchemy/Django ORM, Memcached, Redis, RPC (RabbitMQ/AMQP), and – or you can define your own.

For this example we use the *rpc* result backend, that sends states back as transient messages. The backend is specified via the `backend` argument to **`Celery`**, (or via the

result_backend setting if you choose to use a configuration module):

```
app = Celery('tasks', backend='rpc://', broker='pyamqp://')
```

Or if you want to use Redis as the result backend, but still use RabbitMQ as the message broker (a popular combination):

```
app = Celery('tasks', backend='redis://localhost', broker='pyamqp://')
```

To read more about result backends please see Result Backends.

Now with the result backend configured, let's call the task again. This time you'll hold on to the **AsyncResult** instance returned when you call a task:

```
>>> result = add.delay(4, 4)
```

The **ready()** method returns whether the task has finished processing or not:

```
>>> result.ready()
False
```

You can wait for the result to complete, but this is rarely used since it turns the asynchronous call into a synchronous one:

```
>>> result.get(timeout=1)
8
```

In case the task raised an exception, **get()** will re-raise the exception, but you can override this by specifying the `propagate` argument:

```
>>> result.get(propagate=False)
```

If the task raised an exception, you can also gain access to the original traceback:

```
>>> result.traceback
```

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 **celery.result** for the complete result object reference.

Configuration

Celery, like a consumer appliance, doesn't need much configuration to operate. It has an input and an output. The input must be connected to a broker, and the output can be optionally connected to a result backend. However, if you look closely at the back, there's a lid revealing loads of sliders, dials, and buttons: this is the configuration.

The default configuration should be good enough for most use cases, but there are many options that can be configured to make Celery work exactly as needed. Reading about the options available is a good idea to familiarize yourself with what can be configured. You can read about the options in the Configuration and defaults reference.

The configuration can be set on the app directly or by using a dedicated configuration module. As an example you can configure the default serializer used for serializing task payloads by changing the **task_serializer** setting:

```
app.conf.task_serializer = 'json'
```

If you're configuring many settings at once you can use `update`:

```
app.conf.update(
    task_serializer='json',
    accept_content=['json'],  # Ignore other content
    result_serializer='json',
    timezone='Europe/Oslo',
    enable_utc=True,
)
```

For larger projects, a dedicated configuration module is recommended. Hard coding periodic task intervals and task routing options is discouraged. It is much better to keep these in a centralized location. This is especially true for libraries, as it enables users to control how their tasks behave. A centralized configuration will also allow your SysAdmin to make simple changes in the event of system trouble.

You can tell your Celery instance to use a configuration module by calling the **`app.config_from_object()`** method:

```
app.config_from_object('celeryconfig')
```

This module is often called "`celeryconfig`", but you can use any module name.

In the above case, a module named `celeryconfig.py` must be available to load from the current directory or on the Python path. It could look something like this:

`celeryconfig.py`:

```
broker_url = 'pyamqp://'
result_backend = 'rpc://'

task_serializer = 'json'
result_serializer = 'json'
accept_content = ['json']
timezone = 'Europe/Oslo'
enable_utc = True
```

To verify that your configuration file works properly and doesn't contain any syntax errors, you can try to import it:

```
$ python -m celeryconfig
```

For a complete reference of configuration options, see Configuration and defaults.

To demonstrate the power of configuration files, this is how you'd route a misbehaving task to a dedicated queue:

`celeryconfig.py`:

```
task_routes = {
    'tasks.add': 'low-priority',
}
```

Or instead of routing it you could rate limit the task instead, so that only 10 tasks of this type can be processed in a minute (10/m):

`celeryconfig.py`:

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

If you're using RabbitMQ or Redis as the broker then you can also direct the workers to set a new rate limit for the task at runtime:

```
$ celery -A tasks control rate_limit tasks.add 10/m
worker@example.com: OK
    new rate limit set successfully
```

See Routing Tasks to read more about task routing, and the **task_annotations** setting for more about annotations, or Monitoring and Management Guide for more about remote control commands and how to monitor what your workers are doing.

Where to go from here

If you want to learn more you should continue to the Next Steps tutorial, and after that you can read the User Guide.

Troubleshooting

There's also a troubleshooting section in the Frequently Asked Questions.

Worker doesn't start: Permission Error

- If you're using Debian, Ubuntu or other Debian-based distributions:

  Debian recently renamed the `/dev/shm` special file to `/run/shm`.

  A simple workaround is to create a symbolic link:

  ```
  # ln -s /run/shm /dev/shm
  ```

- Others:

  If you provide any of the **--pidfile**, **--logfile** or **--statedb** arguments, then you must make sure that they point to a file or directory that's writable and readable by the user starting the worker.

Result backend doesn't work or tasks are always in **PENDING** state

All tasks are **PENDING** by default, so the state would've been better named "unknown". Celery doesn't update the state when a task is sent, and any task with no history is assumed to be pending (you know the task id, after all).

1. Make sure that the task doesn't have `ignore_result` enabled.

   Enabling this option will force the worker to skip updating states.

2. Make sure the **task_ignore_result** setting isn't enabled.

3. Make sure that you don't have any old workers still running.

   It's easy to start multiple workers by accident, so make sure that the previous worker is properly shut down before you start a new one.

   An old worker that isn't configured with the expected result backend may be

running and is hijacking the tasks.

The `--pidfile` argument can be set to an absolute path to make sure this doesn't happen.

4. Make sure the client is configured with the right backend.

   If, for some reason, the client is configured to use a different backend than the worker, you won't be able to receive the result. Make sure the backend is configured correctly:

   ```
   >>> result = task.delay()
   >>> print(result.backend)
   ```