Celery-RabbitMQ Documentation

Release 1.0

sivabalan

May 31, 2015

Contents

Contents:

CHAPTER 1

About

Celery is an asynchronous task queue/job queue based on distributed message passing. It is focused on real-time operation and supports scheduling as well.

The execution units, called tasks, are executed concurrently on a single or more worker servers using multiprocessing.

1.1 Get it

You can install Celery either via the Python Package Index (PyPI) or from source.

To install using pip:

$ pip install Celery

To install using easy_install:

$ easy_install Celery

1.2 Downloading and installing from source

Download the latest version of Celery from http://pypi.python.org/pypi/celery/

You can install it by doing the following:

$ tar xvfz celery-0.0.0.tar.gz

$ cd celery-0.0.0

$ python setup.py build

python setup.py install # as root

1.3 Using the development version

You can clone the repository by doing the following:

$ git clone git://github.com/celery/celery.git

CHAPTER 2

Choosing the 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:

2.1 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 are using Ubuntu or Debian install RabbitMQ by executing this command:

$ sudo apt-get install rabbitmq-server

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

And 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

CHAPTER 3

Application

The first thing you need is a Celery instance, this is called the celery application or just app in short. Since 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 you will keep everything contained in a single module.

3.1 Creating Task

Let's create the file tasks.py:

from celery import Celery

celery = Celery('tasks', broker='amqp://guest@localhost//')

@celery.task def add(x, y):

    return x + y

The first argument to Celery is the name of the current module, this is needed so that names can be automatically generated, the second argument is the broker keyword argument which specifies the URL of the message broker you want to use, using RabbitMQ here, which is already the default option.

You defined a single task, called add, which returns the sum of two numbers.

CHAPTER 4

Running the celery worker server

ou now run the worker by executing our program with the worker argument:

$ celery -A tasks worker –loglevel=info

In production you will 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 Running the worker as a daemon for more information).

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

$ celery worker –help

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

$ celery help

CHAPTER 5

Calling the task

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

This is a handy shortcut to the apply_async() method which gives greater control of the task execution:

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

The task has now been processed by the worker you started earlier, and you can verify that by looking at the workers console output.

Calling a task returns an AsyncResult instance, which 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, the exception and traceback). But this isn't enabled by default, and you have to configure Celery to use a result backend, which is not discussed as part of this tutorial.

5.1 Options

With results backend enabled, celery provides lots of options to play with.

Assuming that the result backend is 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=True)
```

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

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

CHAPTER 6

Configuration

Celery, like a consumer appliance doesn't need much to be operated. It has an input and an output, where you must connect the input to a broker and maybe the output to a result backend if so wanted. But 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 uses, but there's many things to tweak so Celery works just the way you want it to. Reading about the options available is a good idea to get familiar with what can be configured. You can read about the options in the 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 CELERY_TASK_SERIALIZER setting:

celery.conf.CELERY_TASK_SERIALIZER = 'json'

If you are configuring many settings at once you can use update:

celery.conf.update(CELERY_TASK_SERIALIZER='json', CELERY_RESULT_SERIALIZER='json', CELERY_TIMEZONE='Europe/Oslo', CELERY_ENABLE_UTC=True,

)

For larger projects using a dedicated configuration module is useful, in fact you are discouraged from hard coding periodic task intervals and task routing options, as it is much better to keep this in a centralized location, and especially for libraries it makes it possible for users to control how they want your tasks to behave, you can also imagine your SysAdmin making simple changes to the configuration in the event of system trouble.

You can tell your Celery instance to use a configuration module, by calling the config_from_object() method:

celery.config_from_object('celeryconfig')

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

A module named celeryconfig.py must then be available to load from the current directory or on the Python path, it could look like this:

celeryconfig.py:

BROKER_URL = 'amqp://' CELERY_RESULT_BACKEND = 'amqp://'

CELERY_TASK_SERIALIZER = 'json' CELERY_RESULT_SERIALIZER = 'json' CELERY_TIMEZONE = 'Europe/Oslo' CELERY_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 how you would route a misbehaving task to a dedicated queue:

celeryconfig.py:

CELERY_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:

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

}

with RabbitMQ as the broker, you can also direct the workers to set a new rate limit for the task at runtime:

$ celery 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 CELERY_ANNOTATIONS setting for more about annotations, or Remote Control Guide for more about remote control commands, and how to monitor what your workers are doing.

CHAPTER 7

The Primitives

The primitives are subtasks themselves, so that they can be combined in any number of ways to compose complex workflows.

Note:

These examples retrieve results, so to try them out you need to configure a result backend. The example project above already does that

7.1 Group

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]
```

7.2 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
```

7.3 Chord

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
```

Be sure to read more about workflows in the Canvas user guide.

CHAPTER 8

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 CELERY_ROUTES setting enables you to route tasks by name and keep everything centralized in one location:

celery.conf.update(

CELERY_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 -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.

CHAPTER 9

Remote Control

If you're using RabbitMQ (AMQP), Redis or MongoDB 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, which is a comma separated list of worker host names:

$ celery -A proj inspect active –destination=worker1.example.com

If a destination is not provided then every worker will act and reply to the request.

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

$ celery -A proj inspect –help

Then there is the celery control command, which 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.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search