Previous topic

Getting Started

Next topic

Brokers

This Page

Show Source

Quick search

Go

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

Introduction to Celery

What's a Task Queue?

Task queues are used as a mechanism to distribute work across threads or machines.

A task queue's input is a unit of work called a task. Dedicated worker processes constantly monitor task queues for new work to perform.

Celery communicates via messages, usually using a broker to mediate between clients and workers. To initiate a task the client adds a message to the queue, the broker then delivers that message to a worker.

A Celery system can consist of multiple workers and brokers, giving way to high availability and horizontal scaling.

Celery is written in Python, but the protocol can be implemented in any language. In addition to Python there's node-celery for Node.js, and a PHP client.

Language interoperability can also be achieved exposing an HTTP endpoint and having a task that requests it (webhooks).

What do I need?

Celery requires a message transport to send and receive messages. The RabbitMQ and Redis broker transports are feature complete, but there's also support for a myriad of other experimental solutions, including using SQLite for local development.

Celery can run on a single machine, on multiple machines, or even across data centers.

Version Requirements

Celery version 4.0 runs on

- Python (2.7, 3.4, 3.5)
- PyPy (5.4, 5.5)

This is the last version to support Python 2.7, and from the next version (Celery 5.x) Python 3.5 or newer is required.

If you're running an older version of Python, you need to be running an older version of Celery:

- Python 2.6: Celery series 3.1 or earlier.
- Python 2.5: Celery series 3.0 or earlier.

Get Started

If this is the first time you're trying to use Celery, or if you haven't kept up with development in the 3.1 version and are coming from previous versions, then you should read our getting started tutorials:

- First Steps with Celery
- Next Steps

- Python 2.4 was Celery series 2.2 or earlier.

Celery is a project with minimal funding, so we don't support Microsoft Windows. Please don't open any issues related to that platform.

Celery is...

- **Simple**

  Celery is easy to use and maintain, and it *doesn't need configuration files*.

  It has an active, friendly community you can talk to for support, including a mailing-list and an IRC channel.

  Here's one of the simplest applications you can make:

```
from celery import Celery

app = Celery('hello', broker='amqp://guest@localhost//')

@app.task
def hello():
    return 'hello world'
```

- **Highly Available**

  Workers and clients will automatically retry in the event of connection loss or failure, and some brokers support HA in way of *Primary/Primary* or *Primary/Replica* replication.

- **Fast**

  A single Celery process can process millions of tasks a minute, with sub-millisecond round-trip latency (using RabbitMQ, librabbitmq, and optimized settings).

- **Flexible**

  Almost every part of *Celery* can be extended or used on its own, Custom pool implementations, serializers, compression schemes, logging, schedulers, consumers, producers, broker transports, and much more.

It supports

- **Brokers**
  - RabbitMQ, Redis,
  - Amazon SQS, and more…
- **Concurrency**
  - prefork (multiprocessing),
  - Eventlet, gevent
  - *solo* (single threaded)
- **Result Stores**
  - AMQP, Redis
  - Memcached,
  - SQLAlchemy, Django ORM
  - Apache Cassandra, Elasticsearch
- **Serialization**
  - *pickle*, *json*, *yaml*, *msgpack*.
  - *zlib*, *bzip2* compression.
  - Cryptographic message signing.

Features

- **Monitoring**

  A stream of monitoring events is emitted by workers and is used by built-in and external tools to tell you what your cluster is doing – in real-time.

  Read more….

- **Work-flows**

  Simple and complex work-flows can be composed using a set of powerful primitives we call the "canvas", including grouping, chaining, chunking, and more.

  Read more….

- **Time & Rate Limits**

  You can control how many tasks can be executed per second/minute/hour, or how long a task can be allowed to run, and this can be set as a default, for a specific worker or individually for each task type.

  Read more….

- **Scheduling**

  You can specify the time to run a task in seconds or a `datetime`, or you can use periodic tasks for recurring events based on a simple interval, or Crontab expressions supporting minute, hour, day of week, day of month, and month of year.

  Read more….

- **Resource Leak Protection**

  The `--max-tasks-per-child` option is used for user tasks leaking resources, like memory or file descriptors, that are simply out of your control.

  Read more….

- **User Components**

  Each worker component can be customized, and additional components can be defined by the user. The worker is built up using "bootsteps" — a dependency graph enabling fine grained control of the worker's internals.

Framework Integration

Celery is easy to integrate with web frameworks, some of them even have integration packages:

Pyramid	pyramid_celery
Pylons	celery-pylons
Flask	not needed
web2py	web2py-celery
Tornado	tornado-celery
Tryton	celery_tryton

For Django see First steps with Django.

The integration packages aren't strictly necessary, but they can make development easier, and sometimes they add important hooks like closing database connections at *fork(2)*.

Quick Jump

I want to ⟶

- get the return value of a task
- use logging from my task
- learn about best practices
- create a custom task base class
- add a callback to a group of tasks
- split a task into several chunks
- optimize the worker
- see a list of built-in task states
- create custom task states
- set a custom task name
- track when a task starts
- retry a task when it fails
- get the id of the current task
- know what queue a task was delivered to
- see a list of running workers
- purge all messages
- inspect what the workers are doing
- see what tasks a worker has registered
- migrate tasks to a new broker
- see a list of event message types
- contribute to Celery
- learn about available configuration settings
- get a list of people and companies using Celery
- write my own remote control command
- change worker queues at runtime

Jump to ⟶

Installation

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

To install using **pip**:

```
$ pip install -U Celery
```

Bundles

Celery also defines a group of bundles that can be used to install Celery and the dependencies for a given feature.

You can specify these in your requirements or on the **pip** command-line by using brackets. Multiple bundles can be specified by separating them by commas.

```
$ pip install "celery[librabbitmq]"

$ pip install "celery[librabbitmq,redis,auth,msgpack]"
```

The following bundles are available:

Serializers

`celery[auth]`: for using the `auth` security serializer.
`celery[msgpack]`: for using the msgpack serializer.
`celery[yaml]`: for using the yaml serializer.

Concurrency

`celery[eventlet]`: for using the eventlet pool.
`celery[gevent]`: for using the gevent pool.

Transports and Backends

celery[librabbitmq]:
: for using the librabbitmq C library.

celery[redis]:
: for using Redis as a message transport or as a result backend.

celery[sqs]:
: for using Amazon SQS as a message transport (*experimental*).

celery[tblib]:
: for using the **task_remote_tracebacks** feature.

celery[memcache]:
: for using Memcached as a result backend (using pylibmc)

celery[pymemcache]:
: for using Memcached as a result backend (pure-Python implementation).

celery[cassandra]:
: for using Apache Cassandra as a result backend with DataStax driver.

celery[couchbase]:
: for using Couchbase as a result backend.

celery[elasticsearch]:
: for using Elasticsearch as a result backend.

celery[riak]:
: for using Riak as a result backend.

celery[dynamodb]:
: for using AWS DynamoDB as a result backend.

celery[zookeeper]:
: for using Zookeeper as a message transport.

celery[sqlalchemy]:
: for using SQLAlchemy as a result backend (*supported*).

celery[pyro]:
: for using the Pyro4 message transport (*experimental*).

celery[slmq]:
: for using the SoftLayer Message Queue transport (*experimental*).

celery[consul]:
: for using the Consul.io Key/Value store as a message transport or result backend (*experimental*).

celery[django]:
: specifies the lowest version possible for Django support.
You should probably not use this in your requirements, it's here for informational purposes only.

Downloading and installing from source

Download the latest version of Celery from PyPI:

https://pypi.org/project/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
```

The last command must be executed as a privileged user if you aren't currently using a virtualenv.

Using the development version

With pip

The Celery development version also requires the development versions of kombu, amqp, billiard, and vine.

You can install the latest snapshot of these using the following pip commands:

```
$ pip install https://github.com/celery/celery/zipball/master#egg=celery
$ pip install https://github.com/celery/billiard/zipball/master#egg=bill
$ pip install https://github.com/celery/py-amqp/zipball/master#egg=amqp
$ pip install https://github.com/celery/kombu/zipball/master#egg=kombu
$ pip install https://github.com/celery/vine/zipball/master#egg=vine
```

With git

Please see the Contributing section.