Celery Documentation

Release 4.4.1

Ask Solem
contributors

Mar 02, 2020

Contents

Celery is a simple, flexible, and reliable distributed system to process vast amounts of messages, while providing operations with the tools required to maintain such a system.

It's a task queue with focus on real-time processing, while also supporting task scheduling.

Celery has a large and diverse community of users and contributors, you should come join us *on IRC* or *our mailing-list*.

Celery is Open Source and licensed under the BSD License.

Donations

This project relies on your generous donations.

If you are using Celery to create a commercial product, please consider becoming our backer or our sponsor to ensure Celery's future.

CHAPTER 2

Getting Started

- If you're new to Celery you can get started by following the *First Steps with Celery* tutorial.

- You can also check out the *FAQ*.

Contents

3.1 Copyright

Celery User Manual

by Ask Solem

Copyright © 2009-2016, Ask Solem.

All rights reserved. This material may be copied or distributed only subject to the terms and conditions set forth in the *Creative Commons Attribution-ShareAlike 4.0 International <*https://creativecommons.org/licenses/by-sa/4.0/legalcode>'_ license.

You may share and adapt the material, even for commercial purposes, but you must give the original author credit. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same license or a license compatible to this one.

Note: While the *Celery* documentation is offered under the Creative Commons *Attribution-ShareAlike 4.0 International* license the Celery *software* is offered under the BSD License (3 Clause)

3.2 Getting Started

Release 4.4

Date Mar 02, 2020

3.2.1 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 and node-celery-ts for Node.js, and a PHP client.

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

What do I need?

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.
- 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 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.

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*

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:

  ```python
  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
- thread (multithreaded)
- *solo* (single threaded)
- **Result Stores**
- AMQP, Redis
- Memcached,
- SQLAlchemy, Django ORM
- Apache Cassandra, Elasticsearch, Riak
- MongoDB, CouchDB, Couchbase, ArangoDB
- Amazon DynamoDB, Amazon S3
- Microsoft Azure Block Blob, Microsoft Azure Cosmos DB
- File system
- **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

- *Brokers*
- *Applications*

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[arangodb] for using ArangoDB 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=billiard
$ 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.

3.2.2 Brokers

> **Release** 4.4
>
> **Date** Mar 02, 2020

Celery supports several message transport alternatives.

Broker Instructions

Using RabbitMQ

- *Installation & Configuration*
- *Installing the RabbitMQ Server*
    - *Setting up RabbitMQ*
    - *Installing RabbitMQ on macOS*
        * *Configuring the system host name*
        * *Starting/Stopping the RabbitMQ server*

Installation & Configuration

RabbitMQ is the default broker so it doesn't require any additional dependencies or initial configuration, other than the URL location of the broker instance you want to use:

```
broker_url = 'amqp://myuser:mypassword@localhost:5672/myvhost'
```

For a description of broker URLs and a full list of the various broker configuration options available to Celery, see *Broker Settings*, and see below for setting up the username, password and vhost.

Installing the RabbitMQ Server

See Installing RabbitMQ over at RabbitMQ's website. For macOS see *Installing RabbitMQ on macOS*.

Note: If you're getting *nodedown* errors after installing and using **rabbitmqctl** then this blog post can help you identify the source of the problem:

http://www.somic.org/2009/02/19/on-rabbitmqctl-and-badrpcnodedown/

Setting up RabbitMQ

To use Celery we need to create a RabbitMQ user, a virtual host and allow that user access to that virtual host:

```
$ sudo rabbitmqctl add_user myuser mypassword
```

```
$ sudo rabbitmqctl add_vhost myvhost
```

```
$ sudo rabbitmqctl set_user_tags myuser mytag
```

```
$ sudo rabbitmqctl set_permissions -p myvhost myuser ".*" ".*" ".*"
```

Substitute in appropriate values for `myuser`, `mypassword` and `myvhost` above.

See the RabbitMQ Admin Guide for more information about access control.

Installing RabbitMQ on macOS

The easiest way to install RabbitMQ on macOS is using Homebrew the new and shiny package management system for macOS.

First, install Homebrew using the one-line command provided by the Homebrew documentation:

```
ruby -e "$(curl -fsSL https://raw.github.com/Homebrew/homebrew/go/install)"
```

Finally, we can install RabbitMQ using **brew**:

```
$ brew install rabbitmq
```

After you've installed RabbitMQ with **brew** you need to add the following to your path to be able to start and stop the broker: add it to the start-up file for your shell (e.g., `.bash_profile` or `.profile`).

```
PATH=$PATH:/usr/local/sbin
```

Configuring the system host name

If you're using a DHCP server that's giving you a random host name, you need to permanently configure the host name. This is because RabbitMQ uses the host name to communicate with nodes.

Use the **scutil** command to permanently set your host name:

```
$ sudo scutil --set HostName myhost.local
```

Then add that host name to `/etc/hosts` so it's possible to resolve it back into an IP address:

```
127.0.0.1        localhost myhost myhost.local
```

If you start the **rabbitmq-server**, your rabbit node should now be *rabbit@myhost*, as verified by **rabbitmqctl**:

```
$ sudo rabbitmqctl status
Status of node rabbit@myhost ...
[{running_applications,[{rabbit,"RabbitMQ","1.7.1"},
                        {mnesia,"MNESIA  CXC 138 12","4.4.12"},
                        {os_mon,"CPO  CXC 138 46","2.2.4"},
                        {sasl,"SASL  CXC 138 11","2.1.8"},
                        {stdlib,"ERTS  CXC 138 10","1.16.4"},
                        {kernel,"ERTS  CXC 138 10","2.13.4"}]},
{nodes,[rabbit@myhost]},
{running_nodes,[rabbit@myhost]}]
...done.
```

This is especially important if your DHCP server gives you a host name starting with an IP address, (e.g., *23.10.112.31.comcast.net*). In this case RabbitMQ will try to use *rabbit@23*: an illegal host name.

Starting/Stopping the RabbitMQ server

To start the server:

```
$ sudo rabbitmq-server
```

you can also run it in the background by adding the -detached option (note: only one dash):

```
$ sudo rabbitmq-server -detached
```

Never use **kill** (*kill(1)*) to stop the RabbitMQ server, but rather use the **rabbitmqctl** command:

```
$ sudo rabbitmqctl stop
```

When the server is running, you can continue reading *Setting up RabbitMQ*.

Using Redis

Installation

For the Redis support you have to install additional dependencies. You can install both Celery and these dependencies in one go using the celery[redis] *bundle*:

```
$ pip install -U "celery[redis]"
```

Configuration

Configuration is easy, just configure the location of your Redis database:

```
app.conf.broker_url = 'redis://localhost:6379/0'
```

Where the URL is in the format of:

```
redis://:password@hostname:port/db_number
```

all fields after the scheme are optional, and will default to `localhost` on port 6379, using database 0.

If a Unix socket connection should be used, the URL needs to be in the format:

```
redis+socket:///path/to/redis.sock
```

Specifying a different database number when using a Unix socket is possible by adding the `virtual_host` parameter to the URL:

```
redis+socket:///path/to/redis.sock?virtual_host=db_number
```

It is also easy to connect directly to a list of Redis Sentinel:

```
app.conf.broker_url =↵
→'sentinel://localhost:26379;sentinel://localhost:26380;sentinel://localhost:26381'
app.conf.broker_transport_options = { 'master_name': "cluster1" }
```

Visibility Timeout

The visibility timeout defines the number of seconds to wait for the worker to acknowledge the task before the message is redelivered to another worker. Be sure to see *Caveats* below.

This option is set via the *broker_transport_options* setting:

```
app.conf.broker_transport_options = {'visibility_timeout': 3600}  # 1 hour.
```

The default visibility timeout for Redis is 1 hour.

Results

If you also want to store the state and return values of tasks in Redis, you should configure these settings:

```
app.conf.result_backend = 'redis://localhost:6379/0'
```

For a complete list of options supported by the Redis result backend, see *Redis backend settings*.

If you are using Sentinel, you should specify the master_name using the *result_backend_transport_options* setting:

```
app.conf.result_backend_transport_options = {'master_name': "mymaster"}
```

Caveats

Fanout prefix

Broadcast messages will be seen by all virtual hosts by default.

You have to set a transport option to prefix the messages so that they will only be received by the active virtual host:

```
app.conf.broker_transport_options = {'fanout_prefix': True}
```

Note that you won't be able to communicate with workers running older versions or workers that doesn't have this setting enabled.

This setting will be the default in the future, so better to migrate sooner rather than later.

Fanout patterns

Workers will receive all task related events by default.

To avoid this you must set the `fanout_patterns` fanout option so that the workers may only subscribe to worker related events:

```
app.conf.broker_transport_options = {'fanout_patterns': True}
```

Note that this change is backward incompatible so all workers in the cluster must have this option enabled, or else they won't be able to communicate.

This option will be enabled by default in the future.

Visibility timeout

If a task isn't acknowledged within the *Visibility Timeout* the task will be redelivered to another worker and executed.

This causes problems with ETA/countdown/retry tasks where the time to execute exceeds the visibility timeout; in fact if that happens it will be executed again, and again in a loop.

So you have to increase the visibility timeout to match the time of the longest ETA you're planning to use.

Note that Celery will redeliver messages at worker shutdown, so having a long visibility timeout will only delay the redelivery of 'lost' tasks in the event of a power failure or forcefully terminated workers.

Periodic tasks won't be affected by the visibility timeout, as this is a concept separate from ETA/countdown.

You can increase this timeout by configuring a transport option with the same name:

```
app.conf.broker_transport_options = {'visibility_timeout': 43200}
```

The value must be an int describing the number of seconds.

Key eviction

Redis may evict keys from the database in some situations

If you experience an error like:

```
InconsistencyError: Probably the key ('_kombu.binding.celery') has been
removed from the Redis database.
```

then you may want to configure the **redis-server** to not evict keys by setting the `timeout` parameter to 0 in the redis configuration file.

Using Amazon SQS

Installation

For the Amazon SQS support you have to install additional dependencies. You can install both Celery and these dependencies in one go using the `celery[sqs]` *bundle*:

```
$ pip install celery[sqs]
```

Configuration

You have to specify SQS in the broker URL:

```
broker_url = 'sqs://ABCDEFGHIJKLMNOPQRST:ZYXK7NiynGlTogH8Nj+P9nlE73sq3@'
```

where the URL format is:

```
sqs://aws_access_key_id:aws_secret_access_key@
```

Please note that you must remember to include the `@` sign at the end and encode the password so it can always be parsed correctly. For example:

```python
from kombu.utils.url import safequote

aws_access_key = safequote("ABCDEFGHIJKLMNOPQRST")
aws_secret_key = safequote("ZYXK7NiynG/TogH8Nj+P9nlE73sq3")

broker_url = "sqs://{aws_access_key}:{aws_secret_key}@".format(
    aws_access_key=aws_access_key, aws_secret_key=aws_secret_key,
)
```

The login credentials can also be set using the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`, in that case the broker URL may only be `sqs://`.

If you are using IAM roles on instances, you can set the BROKER_URL to: `sqs://` and kombu will attempt to retrieve access tokens from the instance metadata.

Options

Region

The default region is `us-east-1` but you can select another region by configuring the `broker_transport_options` setting:

```
broker_transport_options = {'region': 'eu-west-1'}
```

See also:

An overview of Amazon Web Services regions can be found here:

> http://aws.amazon.com/about-aws/globalinfrastructure/

Visibility Timeout

The visibility timeout defines the number of seconds to wait for the worker to acknowledge the task before the message is redelivered to another worker. Also see caveats below.

This option is set via the *broker_transport_options* setting:

```
broker_transport_options = {'visibility_timeout': 3600}  # 1 hour.
```

The default visibility timeout is 30 seconds.

Polling Interval

The polling interval decides the number of seconds to sleep between unsuccessful polls. This value can be either an int or a float. By default the value is *one second*: this means the worker will sleep for one second when there's no more messages to read.

You must note that **more frequent polling is also more expensive, so increasing the polling interval can save you money**.

The polling interval can be set via the *broker_transport_options* setting:

```
broker_transport_options = {'polling_interval': 0.3}
```

Very frequent polling intervals can cause *busy loops*, resulting in the worker using a lot of CPU time. If you need sub-millisecond precision you should consider using another transport, like *RabbitMQ <broker-amqp>*, or *Redis <broker-redis>*.

Long Polling

SQS Long Polling is enabled by default and the `WaitTimeSeconds` parameter of ReceiveMessage operation is set to 10 seconds.

The value of `WaitTimeSeconds` parameter can be set via the *broker_transport_options* setting:

```
broker_transport_options = {'wait_time_seconds': 15}
```

Valid values are 0 to 20. Note that newly created queues themselves (also if created by Celery) will have the default value of 0 set for the "Receive Message Wait Time" queue property.

Queue Prefix

By default Celery won't assign any prefix to the queue names, If you have other services using SQS you can configure it do so using the *broker_transport_options* setting:

```
broker_transport_options = {'queue_name_prefix': 'celery-'}
```

Caveats

- If a task isn't acknowledged within the `visibility_timeout`, the task will be redelivered to another worker and executed.

     This causes problems with ETA/countdown/retry tasks where the time to execute exceeds the visibility timeout; in fact if that happens it will be executed again, and again in a loop.

     So you have to increase the visibility timeout to match the time of the longest ETA you're planning to use.

Note that Celery will redeliver messages at worker shutdown, so having a long visibility timeout will only delay the redelivery of 'lost' tasks in the event of a power failure or forcefully terminated workers.

Periodic tasks won't be affected by the visibility timeout, as it is a concept separate from ETA/countdown.

The maximum visibility timeout supported by AWS as of this writing is 12 hours (43200 seconds):

```
broker_transport_options = {'visibility_timeout': 43200}
```

- SQS doesn't yet support worker remote control commands.
- SQS doesn't yet support events, and so cannot be used with **celery events**, **celerymon**, or the Django Admin monitor.

Results

Multiple products in the Amazon Web Services family could be a good candidate to store or publish results with, but there's no such result backend included at this point.

> **Warning:** Don't use the amqp result backend with SQS.
>
> It will create one queue for every task, and the queues will not be collected. This could cost you money that would be better spent contributing an AWS result store backend back to Celery :)

Broker Overview

This is comparison table of the different transports supports, more information can be found in the documentation for each individual transport (see *Broker Instructions*).

Name	Status	Monitoring	Remote Control
RabbitMQ	Stable	Yes	Yes
Redis	Stable	Yes	Yes
Amazon SQS	Stable	No	No
Zookeeper	Experimental	No	No

Experimental brokers may be functional but they don't have dedicated maintainers.

Missing monitor support means that the transport doesn't implement events, and as such Flower, *celery events*, *celerymon* and other event-based monitoring tools won't work.

Remote control means the ability to inspect and manage workers at runtime using the *celery inspect* and *celery control* commands (and other tools using the remote control API).

3.2.3 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*
  - *RabbitMQ*
  - *Redis*
  - *Other brokers*
- *Installing Celery*
- *Application*
- *Running the Celery worker server*
- *Calling the task*
- *Keeping Results*
- *Configuration*
- *Where to go from here*
- *Troubleshooting*
  - *Worker doesn't start: Permission Error*
  - *Result backend doesn't work or tasks are always in* `PENDING` *state*

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
```

Or, if you want to run it on Docker execute this:

```
$ docker run -d -p 5672:5672 rabbitmq
```

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

If you want to run it on Docker execute this:

```
$ docker run -d -p 6379:6379 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`:

```python
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)
```

3.2.4 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*
- *Calling Tasks*
- Canvas*: Designing Work-flows*
- *Routing*
- *Remote Control*
- *Timezone*
- *Optimization*
- *What to do now?*

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

```python
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 started.
```

– 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.log
```

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:

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

>>> 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, '2')
>>> res.get(timeout=1)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "celery/result.py", line 221, in get
    return self.backend.wait_for_pending(
  File "celery/backends/asynchronous.py", line 195, in wait_for_pending
    return result.maybe_throw(callback=callback, propagate=propagate)
  File "celery/result.py", line 333, in maybe_throw
    self.throw(value, self._to_remote_traceback(tb))
  File "celery/result.py", line 326, in throw
    self.on_ready.throw(*args, **kwargs)
  File "vine/promises.py", line 244, in throw
    reraise(type(exc), exc, tb)
  File "vine/five.py", line 195, in reraise
    raise value
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

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("unsupported operand type(s) for +: 'int' and 'str'")
```

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 such a way 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 support the calling API, meaning they have `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 support 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 range(10))().get()
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

- Partial group

```
>>> g = group(add.s(i) for i in range(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 range(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 range(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 filters)
```

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.

3.2.5 Resources

- *Getting Help*
  - *Mailing list*
  - *IRC*
- *Bug tracker*
- *Wiki*
- *Contributing*
- *License*

Getting Help

Mailing list

For discussions about the usage, development, and future of Celery, please join the celery-users mailing list.

IRC

Come chat with us on IRC. The **#celery** channel is located at the Freenode network.

Bug tracker

If you have any suggestions, bug reports, or annoyances please report them to our issue tracker at https://github.com/celery/celery/issues/

Wiki

https://wiki.github.com/celery/celery/

Contributing

Development of *celery* happens at GitHub: https://github.com/celery/celery

You're highly encouraged to participate in the development of *celery*. If you don't like GitHub (for some reason) you're welcome to send regular patches.

Be sure to also read the Contributing to Celery section in the documentation.

License

This software is licensed under the *New BSD License*. See the `LICENSE` file in the top distribution directory for the full license text.

3.3 User Guide

> **Release** 4.4
>
> **Date** Mar 02, 2020

3.3.1 Application

- *Main Name*
- *Configuration*
- *Laziness*
- *Breaking the chain*
- *Abstract Tasks*

The Celery library must be instantiated before use, this instance is called an application (or *app* for short).

The application is thread-safe so that multiple Celery applications with different configurations, components, and tasks can co-exist in the same process space.

Let's create one now:

```
>>> from celery import Celery
>>> app = Celery()
>>> app
<Celery __main__:0x100469fd0>
```

The last line shows the textual representation of the application: including the name of the app class (`Celery`), the name of the current main module (`__main__`), and the memory address of the object (`0x100469fd0`).

Main Name

Only one of these is important, and that's the main module name. Let's look at why that is.

When you send a task message in Celery, that message won't contain any source code, but only the name of the task you want to execute. This works similarly to how host names work on the internet: every worker maintains a mapping of task names to their actual functions, called the *task registry*.

Whenever you define a task, that task will also be added to the local registry:

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

>>> add
<@task: __main__.add>

>>> add.name
__main__.add

>>> app.tasks['__main__.add']
<@task: __main__.add>
```

and there you see that __main__ again; whenever Celery isn't able to detect what module the function belongs to, it uses the main module name to generate the beginning of the task name.

This is only a problem in a limited set of use cases:

1. If the module that the task is defined in is run as a program.

2. If the application is created in the Python shell (REPL).

For example here, where the tasks module is also used to start a worker with *app.worker_main()*:

tasks.py:

```python
from celery import Celery
app = Celery()

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

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

When this module is executed the tasks will be named starting with "__main__", but when the module is imported by another process, say to call a task, the tasks will be named starting with "tasks" (the real name of the module):

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

You can specify another name for the main module:

```python
>>> app = Celery('tasks')
>>> app.main
'tasks'

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

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

See also:

Names

Configuration

There are several options you can set that'll change how Celery works. These options can be set directly on the app instance, or you can use a dedicated configuration module.

The configuration is available as *app.conf*:

```python
>>> app.conf.timezone
'Europe/London'
```

where you can also set configuration values directly:

```python
>>> app.conf.enable_utc = True
```

or update several keys at once by using the `update` method:

```
>>> app.conf.update(
...     enable_utc=True,
...     timezone='Europe/London',
...)
```

The configuration object consists of multiple dictionaries that are consulted in order:

1. Changes made at run-time.

2. The configuration module (if any)

3. The default configuration (`celery.app.defaults`).

You can even add new default sources by using the `app.add_defaults()` method.

See also:

Go to the *Configuration reference* for a complete listing of all the available settings, and their default values.

config_from_object

The `app.config_from_object()` method loads configuration from a configuration object.

This can be a configuration module, or any object with configuration attributes.

Note that any configuration that was previously set will be reset when `config_from_object()` is called. If you want to set additional configuration you should do so after.

Example 1: Using the name of a module

The `app.config_from_object()` method can take the fully qualified name of a Python module, or even the name of a Python attribute, for example: `"celeryconfig"`, `"myproj.config.celery"`, or `"myproj.config:CeleryConfig"`:

```python
from celery import Celery

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

The `celeryconfig` module may then look like this:

celeryconfig.py:

```python
enable_utc = True
timezone = 'Europe/London'
```

and the app will be able to use it as long as `import celeryconfig` is possible.

Example 2: Passing an actual module object

You can also pass an already imported module object, but this isn't always recommended.

Tip: Using the name of a module is recommended as this means the module does not need to be serialized when the prefork pool is used. If you're experiencing configuration problems or pickle errors then please try using the name of a module instead.

```python
import celeryconfig

from celery import Celery

app = Celery()
app.config_from_object(celeryconfig)
```

Example 3: Using a configuration class/object

```python
from celery import Celery

app = Celery()

class Config:
    enable_utc = True
    timezone = 'Europe/London'

app.config_from_object(Config)
# or using the fully qualified name of the object:
#   app.config_from_object('module:Config')
```

config_from_envvar

The `app.config_from_envvar()` takes the configuration module name from an environment variable

For example – to load configuration from a module specified in the environment variable named `CELERY_CONFIG_MODULE`:

```python
import os
from celery import Celery

#: Set default configuration module name
os.environ.setdefault('CELERY_CONFIG_MODULE', 'celeryconfig')

app = Celery()
app.config_from_envvar('CELERY_CONFIG_MODULE')
```

You can then specify the configuration module to use via the environment:

```
$ CELERY_CONFIG_MODULE="celeryconfig.prod" celery worker -l info
```

Censored configuration

If you ever want to print out the configuration, as debugging information or similar, you may also want to filter out sensitive information like passwords and API keys.

Celery comes with several utilities useful for presenting the configuration, one is `humanize()`:

```
>>> app.conf.humanize(with_defaults=False, censored=True)
```

This method returns the configuration as a tabulated string. This will only contain changes to the configuration by default, but you can include the built-in default keys and values by enabling the `with_defaults` argument.

If you instead want to work with the configuration as a dictionary, you can use the `table()` method:

```
>>> app.conf.table(with_defaults=False, censored=True)
```

Please note that Celery won't be able to remove all sensitive information, as it merely uses a regular expression to search for commonly named keys. If you add custom settings containing sensitive information you should name the keys using a name that Celery identifies as secret.

A configuration setting will be censored if the name contains any of these sub-strings:

`API`, `TOKEN`, `KEY`, `SECRET`, `PASS`, `SIGNATURE`, `DATABASE`

Laziness

The application instance is lazy, meaning it won't be evaluated until it's actually needed.

Creating a `Celery` instance will only do the following:

1. Create a logical clock instance, used for events.

2. Create the task registry.

3. Set itself as the current app (but not if the `set_as_current` argument was disabled)

4. Call the `app.on_init()` callback (does nothing by default).

The `app.task()` decorators don't create the tasks at the point when the task is defined, instead it'll defer the creation of the task to happen either when the task is used, or after the application has been *finalized*,

This example shows how the task isn't created until you use the task, or access an attribute (in this case `repr()`):

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

>>> type(add)
<class 'celery.local.PromiseProxy'>

>>> add.__evaluated__()
False

>>> add          # <-- causes repr(add) to happen
<@task: __main__.add>

>>> add.__evaluated__()
True
```

Finalization of the app happens either explicitly by calling `app.finalize()` – or implicitly by accessing the `app.tasks` attribute.

Finalizing the object will:

1. Copy tasks that must be shared between apps

   Tasks are shared by default, but if the `shared` argument to the task decorator is disabled, then the task will be private to the app it's bound to.

2. Evaluate all pending task decorators.

3. Make sure all tasks are bound to the current app.

> Tasks are bound to an app so that they can read default values from the configuration.

The "default app"

Celery didn't always have applications, it used to be that there was only a module-based API, and for backwards compatibility the old API is still there until the release of Celery 5.0.

Celery always creates a special app - the "default app", and this is used if no custom application has been instantiated.

The `celery.task` module is there to accommodate the old API, and shouldn't be used if you use a custom app. You should always use the methods on the app instance, not the module based API.

For example, the old Task base class enables many compatibility features where some may be incompatible with newer features, such as task methods:

```python
from celery.task import Task   # << OLD Task base class.

from celery import Task        # << NEW base class.
```

The new base class is recommended even if you use the old module-based API.

Breaking the chain

While it's possible to depend on the current app being set, the best practice is to always pass the app instance around to anything that needs it.

I call this the "app chain", since it creates a chain of instances depending on the app being passed.

The following example is considered bad practice:

```python
from celery import current_app

class Scheduler(object):

    def run(self):
        app = current_app
```

Instead it should take the `app` as an argument:

```python
class Scheduler(object):

    def __init__(self, app):
        self.app = app
```

Internally Celery uses the `celery.app.app_or_default()` function so that everything also works in the module-based compatibility API

```python
from celery.app import app_or_default

class Scheduler(object):
    def __init__(self, app=None):
        self.app = app_or_default(app)
```

In development you can set the `CELERY_TRACE_APP` environment variable to raise an exception if the app chain breaks:

```
$ CELERY_TRACE_APP=1 celery worker -l info
```

Evolving the API

Celery has changed a lot from 2009 since it was initially created.

For example, in the beginning it was possible to use any callable as a task:

```
def hello(to):
    return 'hello {0}'.format(to)

>>> from celery.execute import apply_async

>>> apply_async(hello, ('world!',))
```

or you could also create a `Task` class to set certain options, or override other behavior

```
from celery.task import Task
from celery.registry import tasks

class Hello(Task):
    queue = 'hipri'

    def run(self, to):
        return 'hello {0}'.format(to)
tasks.register(Hello)

>>> Hello.delay('world!')
```

Later, it was decided that passing arbitrary call-able's was an anti-pattern, since it makes it very hard to use serializers other than pickle, and the feature was removed in 2.0, replaced by task decorators:

```
from celery.task import task

@task(queue='hipri')
def hello(to):
    return 'hello {0}'.format(to)
```

Abstract Tasks

All tasks created using the `task()` decorator will inherit from the application's base `Task` class.

You can specify a different base class using the `base` argument:

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

To create a custom task class you should inherit from the neutral base class: `celery.Task`.

```
from celery import Task

class DebugTask(Task):
```

(continues on next page)

```python
    def __call__(self, *args, **kwargs):
        print('TASK STARTING: {0.name}[{0.request.id}]'.format(self))
        return self.run(*args, **kwargs)
```

Tip: If you override the task's `__call__` method, then it's very important that you also call `self.run` to execute the body of the task. Do not call `super().__call__`. The `__call__` method of the neutral base class `celery.Task` is only present for reference. For optimization, this has been unrolled into `celery.app.trace.build_tracer.trace_task` which calls `run` directly on the custom task class if no `__call__` method is defined.

The neutral base class is special because it's not bound to any specific app yet. Once a task is bound to an app it'll read configuration to set default values, and so on.

To realize a base class you need to create a task using the *app.task()* decorator:

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

It's even possible to change the default base class for an application by changing its *app.Task()* attribute:

```python
>>> from celery import Celery, Task

>>> app = Celery()

>>> class MyBaseTask(Task):
...     queue = 'hipri'

>>> app.Task = MyBaseTask
>>> app.Task
<unbound MyBaseTask>

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

>>> add
<@task: __main__.add>

>>> add.__class__.mro()
[<class add of <Celery __main__:0x1012b4410>>,
 <unbound MyBaseTask>,
 <unbound Task>,
 <type 'object'>]
```

3.3.2 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 your 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:

```python
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:

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

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 `shared_task()` decorator:

```python
from celery import shared_task

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

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):

```
@app.task
@decorator2
@decorator1
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

> **Absolute Imports**
>
> The best practice for developers targeting Python 2 is to add the following to the top of **every module**:
>
> ```
> from __future__ import absolute_import
> ```
>
> 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.

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:

```
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 having 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:

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

```python
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:

```python
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
```

Note: If a specific Celery logger you need is not emitting logs, you should check that the logger is propagating properly. In this example "celery.app.trace" is enabled so that "succeeded in" logs are emitted:

```python
import celery
import logging


@celery.signals.after_setup_logger.connect
def on_after_setup_logger(**kwargs):
    logger = logging.getLogger('celery')
    logger.propagate = True
    logger = logging.getLogger('celery.app.trace')
    logger.propagate = True
```

Note: If you want to completely disable Celery logging configuration, use the *setup_logging* signal:

```python
import celery


@celery.signals.setup_logging.connect
def on_setup_logging(**kwargs):
    pass
```

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:

```python
>>> @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 receiving 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` argument 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.

```python
@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.

New in version 4.4.

You can also set *autoretry_for*, *retry_kwargs*, *retry_backoff*, *retry_backoff_max* and *retry_jitter* options in class-based tasks:

```
class BaseTaskWithRetry(Task):
    autoretry_for = (TypeError,)
    retry_kwargs = {'max_retries': 5}
    retry_backoff = True
    retry_backoff_max = 700
    retry_jitter = False
```

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 deduced, (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:.

```python
@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:

```python
# 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:

```python
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:

```python
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:

```python
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:

```python
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:

```
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,

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

will do roughly this behind the scenes:

```
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,

```
from celery import Task

class NaiveAuthenticateServer(Task):
```

```python
    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
```

Per task usage

The above can be added to each task 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.

App-wide usage

You can also use your custom class in your whole Celery app by passing it as the `task_cls` argument when instantiating the app. This argument should be either a string giving the python path to your Task class or the class itself:

```python
from celery import Celery

app = Celery('tasks', task_cls='your.module.path:DatabaseTask')
```

This will make all your tasks declared using the decorator syntax within your app to use your `DatabaseTask` class and will all have a db attribute.

The default value is the class provided by Celery: `'celery.app.task:Task'`.

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.

```python
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=False):
        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 into 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.

```python
@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`.

```python
@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:

```python
@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(page):
    return myparser.parse_document(page)

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

Good:

```python
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 allow you to run subtasks synchronously within a task, but in rare or extreme cases you might need to do so. **WARNING**: enabling subtasks to 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].

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:

```python
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:

```python
>>> 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:

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

```python
>>> 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:

```python
from django.db import transaction
from django.http import HttpResponseRedirect

@transaction.atomic
def create_article(request):
    article = Article.objects.create()
    expand_abbreviations.delay(article.pk)
    return HttpResponseRedirect('/articles/')
```

This is a Django view creating an article object in the database, then passing the primary key to a task. It uses the *transaction.atomic* 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.

```python
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:

```python
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

```python
from django import forms
from django.http import HttpResponseRedirect
from django.template.context import RequestContext
```

```python
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)
```

```python
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
```

3.3.3 Calling Tasks

Basics

This document describes Celery's uniform "Calling API" used by task instances and the *canvas*.

The API defines a standard set of execution options, as well as three methods:

- apply_async(args[, kwargs[, ...]])

    Sends a task message.

- delay(*args, **kwargs)

    Shortcut to send a task message, but doesn't support execution options.

- *calling* (`__call__`)

    Applying an object supporting the calling API (e.g., `add(2, 2)`) means that the task will not be executed by a worker, but in the current process instead (a message won't be sent).

Quick Cheat Sheet

- **T.delay(arg, kwarg=value)** Star arguments shortcut to `.apply_async`. (`.delay(*args, **kwargs)` calls `.apply_async(args, kwargs)`).

- T.apply_async((arg,), {'kwarg': value})

- **T.apply_async(countdown=10)** executes in 10 seconds from now.

- **T.apply_async(eta=now + timedelta(seconds=10))** executes in 10 seconds from now, specified using `eta`

- **T.apply_async(countdown=60, expires=120)** executes in one minute from now, but expires after 2 minutes.

- **T.apply_async(expires=now + timedelta(days=2))** expires in 2 days, set using `datetime`.

Example

The *delay()* method is convenient as it looks like calling a regular function:

```
task.delay(arg1, arg2, kwarg1='x', kwarg2='y')
```

Using *apply_async()* instead you have to write:

```
task.apply_async(args=[arg1, arg2], kwargs={'kwarg1': 'x', 'kwarg2': 'y'})
```

Tip

If the task isn't registered in the current process you can use *send_task()* to call the task by name instead.

So *delay* is clearly convenient, but if you want to set additional execution options you have to use `apply_async`.

The rest of this document will go into the task execution options in detail. All examples use a task called *add*, returning the sum of two arguments:

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

There's another way. . .

You'll learn more about this later while reading about the *Canvas*, but *signature*'s are objects used to pass around the signature of a task invocation, (for example to send it over the network), and they also support the Calling API:

```
task.s(arg1, arg2, kwarg1='x', kwargs2='y').apply_async()
```

Linking (callbacks/errbacks)

Celery supports linking tasks together so that one task follows another. The callback task will be applied with the result of the parent task as a partial argument:

```
add.apply_async((2, 2), link=add.s(16))
```

> **What's s?**
>
> The `add.s` call used here is called a signature. If you don't know what they are you should read about them in the *canvas guide*. There you can also learn about `chain`: a simpler way to chain tasks together.
>
> In practice the `link` execution option is considered an internal primitive, and you'll probably not use it directly, but use chains instead.

Here the result of the first task (4) will be sent to a new task that adds 16 to the previous result, forming the expression $(2 + 2) + 16 = 20$

You can also cause a callback to be applied if task raises an exception (*errback*), but this behaves differently from a regular callback in that it will be passed the id of the parent task, not the result. This is because it may not always be possible to serialize the exception raised, and so this way the error callback requires a result backend to be enabled, and the task must retrieve the result of the task instead.

This is an example error callback:

```python
@app.task
def error_handler(uuid):
    result = AsyncResult(uuid)
    exc = result.get(propagate=False)
    print('Task {0} raised exception: {1!r}\n{2!r}'.format(
            uuid, exc, result.traceback))
```

it can be added to the task using the `link_error` execution option:

```
add.apply_async((2, 2), link_error=error_handler.s())
```

In addition, both the `link` and `link_error` options can be expressed as a list:

```
add.apply_async((2, 2), link=[add.s(16), other_task.s()])
```

The callbacks/errbacks will then be called in order, and all callbacks will be called with the return value of the parent task as a partial argument.

On message

Celery supports catching all states changes by setting on_message callback.

For example for long-running tasks to send task progress you can do something like this:

```python
@app.task(bind=True)
def hello(self, a, b):
    time.sleep(1)
    self.update_state(state="PROGRESS", meta={'progress': 50})
    time.sleep(1)
    self.update_state(state="PROGRESS", meta={'progress': 90})
```

(continues on next page)

```
    time.sleep(1)
    return 'hello world: %i' % (a+b)
```

```
def on_raw_message(body):
    print(body)

a, b = 1, 1
r = hello.apply_async(args=(a, b))
print(r.get(on_message=on_raw_message, propagate=False))
```

Will generate output like this:

```
{'task_id': '5660d3a3-92b8-40df-8ccc-33a5d1d680d7',
 'result': {'progress': 50},
 'children': [],
 'status': 'PROGRESS',
 'traceback': None}
{'task_id': '5660d3a3-92b8-40df-8ccc-33a5d1d680d7',
 'result': {'progress': 90},
 'children': [],
 'status': 'PROGRESS',
 'traceback': None}
{'task_id': '5660d3a3-92b8-40df-8ccc-33a5d1d680d7',
 'result': 'hello world: 10',
 'children': [],
 'status': 'SUCCESS',
 'traceback': None}
hello world: 10
```

ETA and Countdown

The ETA (estimated time of arrival) lets you set a specific date and time that is the earliest time at which your task will be executed. *countdown* is a shortcut to set ETA by seconds into the future.

```
>>> result = add.apply_async((2, 2), countdown=3)
>>> result.get()    # this takes at least 3 seconds to return
20
```

The task is guaranteed to be executed at some time *after* the specified date and time, but not necessarily at that exact time. Possible reasons for broken deadlines may include many items waiting in the queue, or heavy network latency. To make sure your tasks are executed in a timely manner you should monitor the queue for congestion. Use Munin, or similar tools, to receive alerts, so appropriate action can be taken to ease the workload. See *Munin*.

While *countdown* is an integer, *eta* must be a datetime object, specifying an exact date and time (including millisecond precision, and timezone information):

```
>>> from datetime import datetime, timedelta

>>> tomorrow = datetime.utcnow() + timedelta(days=1)
>>> add.apply_async((2, 2), eta=tomorrow)
```

Expiration

The *expires* argument defines an optional expiry time, either as seconds after task publish, or a specific date and time using `datetime`:

```
>>> # Task expires after one minute from now.
>>> add.apply_async((10, 10), expires=60)

>>> # Also supports datetime
>>> from datetime import datetime, timedelta
>>> add.apply_async((10, 10), kwargs,
...                  expires=datetime.now() + timedelta(days=1)
```

When a worker receives an expired task it will mark the task as *REVOKED* (*TaskRevokedError*).

Message Sending Retry

Celery will automatically retry sending messages in the event of connection failure, and retry behavior can be configured – like how often to retry, or a maximum number of retries – or disabled all together.

To disable retry you can set the `retry` execution option to `False`:

```
add.apply_async((2, 2), retry=False)
```

Related Settings

- *task_publish_retry*
- *task_publish_retry_policy*

Retry Policy

A retry policy is a mapping that controls how retries behave, and can contain the following keys:

- *max_retries*

    Maximum number of retries before giving up, in this case the exception that caused the retry to fail will be raised.

    A value of `None` means it will retry forever.

    The default is to retry 3 times.

- *interval_start*

    Defines the number of seconds (float or integer) to wait between retries. Default is 0 (the first retry will be instantaneous).

- *interval_step*

    On each consecutive retry this number will be added to the retry delay (float or integer). Default is 0.2.

- *interval_max*

    Maximum number of seconds (float or integer) to wait between retries. Default is 0.2.

For example, the default policy correlates to:

```
add.apply_async((2, 2), retry=True, retry_policy={
    'max_retries': 3,
    'interval_start': 0,
    'interval_step': 0.2,
    'interval_max': 0.2,
})
```

the maximum time spent retrying will be 0.4 seconds. It's set relatively short by default because a connection failure could lead to a retry pile effect if the broker connection is down – For example, many web server processes waiting to retry, blocking other incoming requests.

Connection Error Handling

When you send a task and the message transport connection is lost, or the connection cannot be initiated, an `OperationalError` error will be raised:

```
>>> from proj.tasks import add
>>> add.delay(2, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "celery/app/task.py", line 388, in delay
        return self.apply_async(args, kwargs)
  File "celery/app/task.py", line 503, in apply_async
    **options
  File "celery/app/base.py", line 662, in send_task
    amqp.send_task_message(P, name, message, **options)
  File "celery/backends/rpc.py", line 275, in on_task_call
    maybe_declare(self.binding(producer.channel), retry=True)
  File "/opt/celery/kombu/kombu/messaging.py", line 204, in _get_channel
    channel = self._channel = channel()
  File "/opt/celery/py-amqp/amqp/connection.py", line 272, in connect
    self.transport.connect()
  File "/opt/celery/py-amqp/amqp/transport.py", line 100, in connect
    self._connect(self.host, self.port, self.connect_timeout)
  File "/opt/celery/py-amqp/amqp/transport.py", line 141, in _connect
    self.sock.connect(sa)
  kombu.exceptions.OperationalError: [Errno 61] Connection refused
```

If you have *retries* enabled this will only happen after retries are exhausted, or when disabled immediately.

You can handle this error too:

```
>>> from celery.utils.log import get_logger
>>> logger = get_logger(__name__)

>>> try:
...     add.delay(2, 2)
... except add.OperationalError as exc:
...     logger.exception('Sending task raised: %r', exc)
```

Serializers

> **Security**
>
> The pickle module allows for execution of arbitrary functions, please see the *security guide*.
>
> Celery also comes with a special serializer that uses cryptography to sign your messages.

Data transferred between clients and workers needs to be serialized, so every message in Celery has a `content_type` header that describes the serialization method used to encode it.

The default serializer is *JSON*, but you can change this using the `task_serializer` setting, or for each individual task, or even per message.

There's built-in support for *JSON*, `pickle`, *YAML* and `msgpack`, and you can also add your own custom serializers by registering them into the Kombu serializer registry

See also:

Message Serialization in the Kombu user guide.

Each option has its advantages and disadvantages.

json – JSON is supported in many programming languages, is now a standard part of Python (since 2.6), and is fairly fast to decode using the modern Python libraries, such as simplejson.

> The primary disadvantage to JSON is that it limits you to the following data types: strings, Unicode, floats, Boolean, dictionaries, and lists. Decimals and dates are notably missing.
>
> Binary data will be transferred using Base64 encoding, increasing the size of the transferred data by 34% compared to an encoding format where native binary types are supported.
>
> However, if your data fits inside the above constraints and you need cross-language support, the default setting of JSON is probably your best choice.
>
> See http://json.org for more information.

> **Note:** (From Python official docs https://docs.python.org/3.6/library/json.html) Keys in key/value pairs of JSON are always of the type `str`. When a dictionary is converted into JSON, all the keys of the dictionary are coerced to strings. As a result of this, if a dictionary is converted into JSON and then back into a dictionary, the dictionary may not equal the original one. That is, `loads(dumps(x)) != x` if x has non-string keys.

---

pickle – If you have no desire to support any language other than Python, then using the pickle encoding will gain you the support of all built-in Python data types (except class instances), smaller messages when sending binary files, and a slight speedup over JSON processing.

> See `pickle` for more information.

yaml – YAML has many of the same characteristics as json, except that it natively supports more data types (including dates, recursive references, etc.).

> However, the Python libraries for YAML are a good bit slower than the libraries for JSON.
>
> If you need a more expressive set of data types and need to maintain cross-language compatibility, then YAML may be a better fit than the above.
>
> See http://yaml.org/ for more information.

msgpack – msgpack is a binary serialization format that's closer to JSON in features. It's very young however, and support should be considered experimental at this point.

> See http://msgpack.org/ for more information.

The encoding used is available as a message header, so the worker knows how to deserialize any task. If you use a custom serializer, this serializer must be available for the worker.

The following order is used to decide the serializer used when sending a task:

1. The *serializer* execution option.
2. The *Task.serializer* attribute
3. The *task_serializer* setting.

Example setting a custom serializer for a single task invocation:

```
>>> add.apply_async((10, 10), serializer='json')
```

Compression

Celery can compress messages using the following builtin schemes:

- *brotli*

    brotli is optimized for the web, in particular small text documents. It is most effective for serving static content such as fonts and html pages.

    To use it, install Celery with:

    ```
    $ pip install celery[brotli]
    ```

- *bzip2*

    bzip2 creates smaller files than gzip, but compression and decompression speeds are noticeably slower than those of gzip.

    To use it, please ensure your Python executable was compiled with bzip2 support.

    If you get the following ImportError:

    ```
    >>> import bz2
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    ImportError: No module named 'bz2'
    ```

    it means that you should recompile your Python version with bzip2 support.

- *gzip*

    gzip is suitable for systems that require a small memory footprint, making it ideal for systems with limited memory. It is often used to generate files with the ".tar.gz" extension.

    To use it, please ensure your Python executable was compiled with gzip support.

    If you get the following ImportError:

    ```
    >>> import gzip
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    ImportError: No module named 'gzip'
    ```

    it means that you should recompile your Python version with gzip support.

- *lzma*

lzma provides a good compression ratio and executes with fast compression and decompression speeds at the expense of higher memory usage.

To use it, please ensure your Python executable was compiled with lzma support and that your Python version is 3.3 and above.

If you get the following `ImportError`:

```
>>> import lzma
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'lzma'
```

it means that you should recompile your Python version with lzma support.

Alternatively, you can also install a backport using:

```
$ pip install celery[lzma]
```

- *zlib*

    zlib is an abstraction of the Deflate algorithm in library form which includes support both for the gzip file format and a lightweight stream format in its API. It is a crucial component of many software systems - Linux kernel and Git VCS just to name a few.

    To use it, please ensure your Python executable was compiled with zlib support.

    If you get the following `ImportError`:

    ```
    >>> import zlib
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    ImportError: No module named 'zlib'
    ```

    it means that you should recompile your Python version with zlib support.

- *zstd*

    zstd targets real-time compression scenarios at zlib-level and better compression ratios. It's backed by a very fast entropy stage, provided by Huff0 and FSE library.

    To use it, install Celery with:

    ```
    $ pip install celery[zstd]
    ```

You can also create your own compression schemes and register them in the `kombu compression registry`.

The following order is used to decide the compression scheme used when sending a task:

1. The *compression* execution option.

2. The `Task.compression` attribute.

3. The *task_compression* attribute.

Example specifying the compression used when calling a task:

```
>>> add.apply_async((2, 2), compression='zlib')
```

Connections

> **Automatic Pool Support**
>
> Since version 2.3 there's support for automatic connection pools, so you don't have to manually handle connections and publishers to reuse connections.
>
> The connection pool is enabled by default since version 2.5.
>
> See the *broker_pool_limit* setting for more information.

You can handle the connection manually by creating a publisher:

```python
results = []
with add.app.pool.acquire(block=True) as connection:
    with add.get_publisher(connection) as publisher:
        try:
            for args in numbers:
                res = add.apply_async((2, 2), publisher=publisher)
                results.append(res)
print([res.get() for res in results])
```

Though this particular example is much better expressed as a group:

```python
>>> from celery import group

>>> numbers = [(2, 2), (4, 4), (8, 8), (16, 16)]
>>> res = group(add.s(i, j) for i, j in numbers).apply_async()

>>> res.get()
[4, 8, 16, 32]
```

Routing options

Celery can route tasks to different queues.

Simple routing (name <-> name) is accomplished using the `queue` option:

```python
add.apply_async(queue='priority.high')
```

You can then assign workers to the `priority.high` queue by using the workers *-Q* argument:

```
$ celery -A proj worker -l info -Q celery,priority.high
```

See also:

Hard-coding queue names in code isn't recommended, the best practice is to use configuration routers (*task_routes*).

To find out more about routing, please see *Routing Tasks*.

Results options

You can enable or disable result storage using the *task_ignore_result* setting or by using the `ignore_result` option:

```
>>> result = add.apply_async(1, 2, ignore_result=True)
>>> result.get()
None

>>> # Do not ignore result (default)
...
>>> result = add.apply_async(1, 2, ignore_result=False)
>>> result.get()
3
```

If you'd like to store additional metadata about the task in the result backend set the `result_extended` setting to True.

See also:

For more information on tasks, please see *Tasks*.

Advanced Options

These options are for advanced users who want to take use of AMQP's full routing capabilities. Interested parties may read the *routing guide*.

- exchange

    Name of exchange (or a `kombu.entity.Exchange`) to send the message to.

- routing_key

    Routing key used to determine.

- priority

    A number between *0* and *255*, where *255* is the highest priority.

    Supported by: RabbitMQ, Redis (priority reversed, 0 is highest).

3.3.4 Canvas: Designing Work-flows

- *Signatures*
    - *Partials*
    - *Immutability*
    - *Callbacks*
- *The Primitives*
    - *Chains*
    - *Groups*
    - *Chords*
    - *Map & Starmap*
    - *Chunks*

Signatures

New in version 2.0.

You just learned how to call a task using the tasks `delay` method in the *calling* guide, 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.

A *signature()* wraps the arguments, keyword 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 its name like this:

```
>>> from celery import signature
>>> signature('tasks.add', args=(2, 2), countdown=10)
tasks.add(2, 2)
```

  This task has a signature of arity 2 (two arguments): `(2, 2)`, and sets the countdown execution option to 10.

- or you can create one using the task's `signature` method:

```
>>> 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)
```

- Keyword arguments are also supported:

```
>>> add.s(2, 2, debug=True)
tasks.add(2, 2, debug=True)
```

- From any signature instance you can inspect the different fields:

```
>>> s = add.signature((2, 2), {'debug': True}, countdown=10)
>>> s.args
(2, 2)
>>> s.kwargs
{'debug': True}
>>> s.options
{'countdown': 10}
```

- It supports the "Calling API" of `delay`, `apply_async`, etc., including being called directly (`__call__`).

  Calling the signature will execute the task inline in the current process:

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

  `delay` is our beloved shortcut to `apply_async` taking star-arguments:

```
>>> result = add.delay(2, 2)
>>> result.get()
4
```

  `apply_async` takes the same arguments as the *app.Task.apply_async()* method:

```
>>> add.apply_async(args, kwargs, **options)
>>> add.signature(args, kwargs, **options).apply_async()

>>> add.apply_async((2, 2), countdown=1)
>>> add.signature((2, 2), countdown=1).apply_async()
```

- You can't define options with `s()`, but a chaining `set` call takes care of that:

```
>>> add.s(2, 2).set(countdown=1)
proj.tasks.add(2, 2)
```

Partials

With a signature, you can execute the task in a worker:

```
>>> add.s(2, 2).delay()
>>> add.s(2, 2).apply_async(countdown=1)
```

Or you can call it directly in the current process:

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

Specifying additional args, kwargs, or options to `apply_async`/`delay` creates partials:

- Any arguments added will be prepended to the args in the signature:

```
>>> partial = add.s(2)          # incomplete signature
>>> partial.delay(4)            # 4 + 2
>>> partial.apply_async((4,))   # same
```

- Any keyword arguments added will be merged with the kwargs in the signature, with the new keyword arguments taking precedence:

```
>>> s = add.s(2, 2)
>>> s.delay(debug=True)                     # -> add(2, 2, debug=True)
>>> s.apply_async(kwargs={'debug': True})   # same
```

- Any options added will be merged with the options in the signature, with the new options taking precedence:

```
>>> s = add.signature((2, 2), countdown=10)
>>> s.apply_async(countdown=1)  # countdown is now 1
```

You can also clone signatures to create derivatives:

```
>>> s = add.s(2)
proj.tasks.add(2)

>>> s.clone(args=(4,), kwargs={'debug': True})
proj.tasks.add(4, 2, debug=True)
```

Immutability

New in version 3.0.

Partials are meant to be used with callbacks, any tasks linked, or chord callbacks will be applied with the result of the parent task. Sometimes you want to specify a callback that doesn't take additional arguments, and in that case you can set the signature to be immutable:

```
>>> add.apply_async((2, 2), link=reset_buffers.signature(immutable=True))
```

The `.si()` shortcut can also be used to create immutable signatures:

```
>>> add.apply_async((2, 2), link=reset_buffers.si())
```

Only the execution options can be set when a signature is immutable, so it's not possible to call the signature with partial args/kwargs.

Note: In this tutorial I sometimes use the prefix operator ~ to signatures. You probably shouldn't use it in your production code, but it's a handy shortcut when experimenting in the Python shell:

```
>>> ~sig

>>> # is the same as
>>> sig.delay().get()
```

Callbacks

New in version 3.0.

Callbacks can be added to any task using the `link` argument to `apply_async`:

```
add.apply_async((2, 2), link=other_task.s())
```

The callback will only be applied if the task exited successfully, and it will be applied with the return value of the parent task as argument.

As I mentioned earlier, any arguments you add to a signature, will be prepended to the arguments specified by the signature itself!

If you have the signature:

```
>>> sig = add.s(10)
```

then *sig.delay(result)* becomes:

```
>>> add.apply_async(args=(result, 10))
```

…

Now let's call our `add` task with a callback using partial arguments:

```
>>> add.apply_async((2, 2), link=add.s(8))
```

As expected this will first launch one task calculating $2 + 2$, then another task calculating $4 + 8$.

The Primitives

New in version 3.0.

Overview

- `group`

    The group primitive is a signature that takes a list of tasks that should be applied in parallel.

- `chain`

    The chain primitive lets us link together signatures so that one is called after the other, essentially forming a *chain* of callbacks.

- `chord`

    A chord is just like a group but with a callback. A chord consists of a header group and a body, where the body is a task that should execute after all of the tasks in the header are complete.

- `map`

    The map primitive works like the built-in `map` function, but creates a temporary task where a list of arguments is applied to the task. For example, `task.map([1, 2])` – results in a single task being called, applying the arguments in order to the task function so that the result is:

    ```
    res = [task(1), task(2)]
    ```

- `starmap`

    Works exactly like map except the arguments are applied as `*args`. For example `add.starmap([(2, 2), (4, 4)])` results in a single task calling:

    ```
    res = [add(2, 2), add(4, 4)]
    ```

- `chunks`

    Chunking splits a long list of arguments into parts, for example the operation:

    ```
    >>> items = zip(range(1000), range(1000))  # 1000 items
    >>> add.chunks(items, 10)
    ```

    will split the list of items into chunks of 10, resulting in 100 tasks (each processing 10 items in sequence).

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

Here's some examples:

- Simple chain

    Here's a simple chain, the first task executes passing its return value to the next task in the chain, and so on.

    ```
    >>> from celery import chain

    >>> # 2 + 2 + 4 + 8
    >>> res = chain(add.s(2, 2), add.s(4), add.s(8))()
    >>> res.get()
    16
    ```

    This can also be written using pipes:

    ```
    >>> (add.s(2, 2) | add.s(4) | add.s(8))().get()
    16
    ```

- Immutable signatures

  Signatures can be partial so arguments can be added to the existing arguments, but you may not always want that, for example if you don't want the result of the previous task in a chain.

  In that case you can mark the signature as immutable, so that the arguments cannot be changed:

  ```
  >>> add.signature((2, 2), immutable=True)
  ```

  There's also a `.si()` shortcut for this, and this is the preferred way of creating signatures:

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

  Now you can create a chain of independent tasks instead:

  ```
  >>> res = (add.si(2, 2) | add.si(4, 4) | add.si(8, 8))()
  >>> res.get()
  16

  >>> res.parent.get()
  8

  >>> res.parent.parent.get()
  4
  ```

- Simple group

  You can easily create a group of tasks to execute in parallel:

  ```
  >>> from celery import group
  >>> res = group(add.s(i, i) for i in range(10))()
  >>> res.get(timeout=1)
  [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
  ```

- Simple chord

  The chord primitive enables us to add a callback to be called when all of the tasks in a group have finished executing. This is often required for algorithms that aren't *embarrassingly parallel*:

  ```
  >>> from celery import chord
  >>> res = chord((add.s(i, i) for i in range(10)), xsum.s())()
  >>> res.get()
  90
  ```

  The above example creates 10 task that all start in parallel, and when all of them are complete the return values are combined into a list and sent to the `xsum` task.

  The body of a chord can also be immutable, so that the return value of the group isn't passed on to the callback:

  ```
  >>> chord((import_contact.s(c) for c in contacts),
  ...         notify_complete.si(import_id)).apply_async()
  ```

  Note the use of `.si` above; this creates an immutable signature, meaning any new arguments passed (including to return value of the previous task) will be ignored.

- Blow your mind by combining

  Chains can be partial too:

```
>>> c1 = (add.s(4) | mul.s(8))

# (16 + 4) * 8
>>> res = c1(16)
>>> res.get()
160
```

this means that you can combine chains:

```
# ((4 + 16) * 2 + 4) * 8
>>> c2 = (add.s(4, 16) | mul.s(2) | (add.s(4) | mul.s(8)))

>>> res = c2()
>>> res.get()
352
```

Chaining a group together with another task will automatically upgrade it to be a chord:

```
>>> c3 = (group(add.s(i, i) for i in range(10)) | xsum.s())
>>> res = c3()
>>> res.get()
90
```

Groups and chords accepts partial arguments too, so in a chain the return value of the previous task is forwarded to all tasks in the group:

```
>>> new_user_workflow = (create_user.s() | group(
...                      import_contacts.s(),
...                      send_welcome_email.s()))
... new_user_workflow.delay(username='artv',
...                         first='Art',
...                         last='Vandelay',
...                         email='art@vandelay.com')
```

If you don't want to forward arguments to the group then you can make the signatures in the group immutable:

```
>>> res = (add.s(4, 4) | group(add.si(i, i) for i in range(10)))()
>>> res.get()
<GroupResult: de44df8c-821d-4c84-9a6a-44769c738f98 [
    bc01831b-9486-4e51-b046-480d7c9b78de,
    2650a1b8-32bf-4771-a645-b0a35dcc791b,
    dcbee2a5-e92d-4b03-b6eb-7aec60fd30cf,
    59f92e0a-23ea-41ce-9fad-8645a0e7759c,
    26e1e707-eccf-4bf4-bbd8-1e1729c3cce3,
    2d10a5f4-37f0-41b2-96ac-a973b1df024d,
    e13d3bdb-7ae3-4101-81a4-6f17ee21df2d,
    104b2be0-7b75-44eb-ac8e-f9220bdfa140,
    c5c551a5-0386-4973-aa37-b65cbeb2624b,
    83f72d71-4b71-428e-b604-6f16599a9f37]>

>>> res.parent.get()
8
```

Chains

New in version 3.0.

Tasks can be linked together: the linked task is called when the task returns successfully:

```
>>> res = add.apply_async((2, 2), link=mul.s(16))
>>> res.get()
4
```

The linked task will be applied with the result of its parent task as the first argument. In the above case where the result was 4, this will result in `mul(4, 16)`.

The results will keep track of any subtasks called by the original task, and this can be accessed from the result instance:

```
>>> res.children
[<AsyncResult: 8c350acf-519d-4553-8a53-4ad3a5c5aeb4>]

>>> res.children[0].get()
64
```

The result instance also has a `collect()` method that treats the result as a graph, enabling you to iterate over the results:

```
>>> list(res.collect())
[(<AsyncResult: 7b720856-dc5f-4415-9134-5c89def5664e>, 4),
 (<AsyncResult: 8c350acf-519d-4553-8a53-4ad3a5c5aeb4>, 64)]
```

By default `collect()` will raise an `IncompleteStream` exception if the graph isn't fully formed (one of the tasks hasn't completed yet), but you can get an intermediate representation of the graph too:

```
>>> for result, value in res.collect(intermediate=True):
....
```

You can link together as many tasks as you like, and signatures can be linked too:

```
>>> s = add.s(2, 2)
>>> s.link(mul.s(4))
>>> s.link(log_result.s())
```

You can also add *error callbacks* using the *on_error* method:

```
>>> add.s(2, 2).on_error(log_error.s()).delay()
```

This will result in the following `.apply_async` call when the signature is applied:

```
>>> add.apply_async((2, 2), link_error=log_error.s())
```

The worker won't actually call the errback as a task, but will instead call the errback function directly so that the raw request, exception and traceback objects can be passed to it.

Here's an example errback:

```
from __future__ import print_function

import os

from proj.celery import app
```

```python
@app.task
def log_error(request, exc, traceback):
    with open(os.path.join('/var/errors', request.id), 'a') as fh:
        print('--\n\n{0} {1} {2}'.format(
            task_id, exc, traceback), file=fh)
```

To make it even easier to link tasks together there's a special signature called *chain* that lets you chain tasks together:

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

>>> # (4 + 4) * 8 * 10
>>> res = chain(add.s(4, 4), mul.s(8), mul.s(10))
proj.tasks.add(4, 4) | proj.tasks.mul(8) | proj.tasks.mul(10)
```

Calling the chain will call the tasks in the current process and return the result of the last task in the chain:

```python
>>> res = chain(add.s(4, 4), mul.s(8), mul.s(10))()
>>> res.get()
640
```

It also sets `parent` attributes so that you can work your way up the chain to get intermediate results:

```python
>>> res.parent.get()
64

>>> res.parent.parent.get()
8

>>> res.parent.parent
<AsyncResult: eeaad925-6778-4ad1-88c8-b2a63d017933>
```

Chains can also be made using the `|` (pipe) operator:

```python
>>> (add.s(2, 2) | mul.s(8) | mul.s(10)).apply_async()
```

Graphs

In addition you can work with the result graph as a *DependencyGraph*:

```python
>>> res = chain(add.s(4, 4), mul.s(8), mul.s(10))()

>>> res.parent.parent.graph
285fa253-fcf8-42ef-8b95-0078897e83e6(1)
    463afec2-5ed4-4036-b22d-ba067ec64f52(0)
872c3995-6fa0-46ca-98c2-5a19155afcf0(2)
    285fa253-fcf8-42ef-8b95-0078897e83e6(1)
        463afec2-5ed4-4036-b22d-ba067ec64f52(0)
```

You can even convert these graphs to *dot* format:

```python
>>> with open('graph.dot', 'w') as fh:
...     res.parent.parent.graph.to_dot(fh)
```

and create images:

```
$ dot -Tpng graph.dot -o graph.png
```
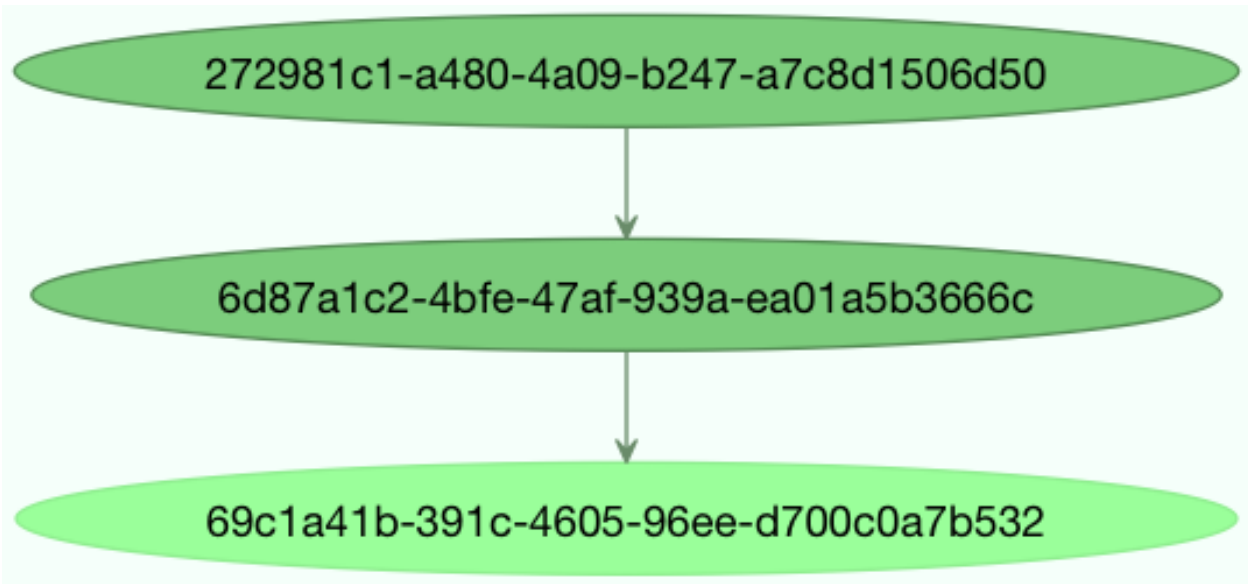

Groups

New in version 3.0.

A group can be used to execute several tasks in parallel.

The *group* function takes a list of signatures:

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

>>> group(add.s(2, 2), add.s(4, 4))
(proj.tasks.add(2, 2), proj.tasks.add(4, 4))
```

If you **call** the group, the tasks will be applied one after another in the current process, and a *GroupResult* instance is returned that can be used to keep track of the results, or tell how many tasks are ready and so on:

```
>>> g = group(add.s(2, 2), add.s(4, 4))
>>> res = g()
>>> res.get()
[4, 8]
```

Group also supports iterators:

```
>>> group(add.s(i, i) for i in range(100))()
```

A group is a signature object, so it can be used in combination with other signatures.

Group Results

The group task returns a special result too, this result works just like normal task results, except that it works on the group as a whole:

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

>>> job = group([
...             add.s(2, 2),
...             add.s(4, 4),
...             add.s(8, 8),
...             add.s(16, 16),
...             add.s(32, 32),
... ])

>>> result = job.apply_async()

>>> result.ready()  # have all subtasks completed?
True
>>> result.successful() # were all subtasks successful?
True
>>> result.get()
[4, 8, 16, 32, 64]
```

The `GroupResult` takes a list of `AsyncResult` instances and operates on them as if it was a single task.

It supports the following operations:

- `successful()`

  Return `True` if all of the subtasks finished successfully (e.g., didn't raise an exception).

- `failed()`

  Return `True` if any of the subtasks failed.

- `waiting()`

  Return `True` if any of the subtasks isn't ready yet.

- `ready()`

  Return `True` if all of the subtasks are ready.

- `completed_count()`

  Return the number of completed subtasks.

- `revoke()`

  Revoke all of the subtasks.

- `join()`

  Gather the results of all subtasks and return them in the same order as they were called (as a list).

Chords

New in version 2.3.

Note: Tasks used within a chord must *not* ignore their results. If the result backend is disabled for *any* task (header or body) in your chord you should read "*Important Notes*." Chords are not currently supported with the RPC result backend.

A chord is a task that only executes after all of the tasks in a group have finished executing.

Let's calculate the sum of the expression $1 + 1 + 2 + 2 + 3 + 3...n + n$ up to a hundred digits.

First you need two tasks, `add()` and `tsum()` (`sum()` is already a standard function):

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

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

Now you can use a chord to calculate each addition step in parallel, and then get the sum of the resulting numbers:

```python
>>> from celery import chord
>>> from tasks import add, tsum

>>> chord(add.s(i, i)
...       for i in range(100))(tsum.s()).get()
9900
```

This is obviously a very contrived example, the overhead of messaging and synchronization makes this a lot slower than its Python counterpart:

```python
>>> sum(i + i for i in range(100))
```

The synchronization step is costly, so you should avoid using chords as much as possible. Still, the chord is a powerful primitive to have in your toolbox as synchronization is a required step for many parallel algorithms.

Let's break the chord expression down:

```python
>>> callback = tsum.s()
>>> header = [add.s(i, i) for i in range(100)]
>>> result = chord(header)(callback)
>>> result.get()
9900
```

Remember, the callback can only be executed after all of the tasks in the header have returned. Each step in the header is executed as a task, in parallel, possibly on different nodes. The callback is then applied with the return value of each task in the header. The task id returned by `chord()` is the id of the callback, so you can wait for it to complete and get the final return value (but remember to *never have a task wait for other tasks*)

Error handling

So what happens if one of the tasks raises an exception?

The chord callback result will transition to the failure state, and the error is set to the *ChordError* exception:

```python
>>> c = chord([add.s(4, 4), raising_task.s(), add.s(8, 8)])
>>> result = c()
>>> result.get()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "*/celery/result.py", line 120, in get
```

```
    interval=interval)
  File "*/celery/backends/amqp.py", line 150, in wait_for
    raise meta['result']
celery.exceptions.ChordError: Dependency 97de6f3f-ea67-4517-a21c-d867c61fcb47
    raised ValueError('something something',)
```

While the traceback may be different depending on the result backend used, you can see that the error description includes the id of the task that failed and a string representation of the original exception. You can also find the original traceback in `result.traceback`.

Note that the rest of the tasks will still execute, so the third task (`add.s(8, 8)`) is still executed even though the middle task failed. Also the *ChordError* only shows the task that failed first (in time): it doesn't respect the ordering of the header group.

To perform an action when a chord fails you can therefore attach an errback to the chord callback:

```python
@app.task
def on_chord_error(request, exc, traceback):
    print('Task {0!r} raised error: {1!r}'.format(request.id, exc))
```

```python
>>> c = (group(add.s(i, i) for i in range(10)) |
...         xsum.s().on_error(on_chord_error.s())).delay()
```

Important Notes

Tasks used within a chord must *not* ignore their results. In practice this means that you must enable a `result_backend` in order to use chords. Additionally, if `task_ignore_result` is set to `True` in your configuration, be sure that the individual tasks to be used within the chord are defined with `ignore_result=False`. This applies to both Task subclasses and decorated tasks.

Example Task subclass:

```python
class MyTask(Task):
    ignore_result = False
```

Example decorated task:

```python
@app.task(ignore_result=False)
def another_task(project):
    do_something()
```

By default the synchronization step is implemented by having a recurring task poll the completion of the group every second, calling the signature when ready.

Example implementation:

```python
from celery import maybe_signature

@app.task(bind=True)
def unlock_chord(self, group, callback, interval=1, max_retries=None):
    if group.ready():
        return maybe_signature(callback).delay(group.join())
    raise self.retry(countdown=interval, max_retries=max_retries)
```

This is used by all result backends except Redis and Memcached: they increment a counter after each task in the header, then applies the callback when the counter exceeds the number of tasks in the set.

The Redis and Memcached approach is a much better solution, but not easily implemented in other backends (suggestions welcome!).

Note: Chords don't properly work with Redis before version 2.2; you'll need to upgrade to at least redis-server 2.2 to use them.

Note: If you're using chords with the Redis result backend and also overriding the `Task.after_return()` method, you need to make sure to call the super method or else the chord callback won't be applied.

```python
def after_return(self, *args, **kwargs):
    do_something()
    super(MyTask, self).after_return(*args, **kwargs)
```

Map & Starmap

`map` and `starmap` are built-in tasks that calls the task for every element in a sequence.

They differ from group in that

- only one task message is sent

- the operation is sequential.

For example using `map`:

```python
>>> from proj.tasks import add

>>> ~xsum.map([range(10), range(100)])
[45, 4950]
```

is the same as having a task doing:

```python
@app.task
def temp():
    return [xsum(range(10)), xsum(range(100))]
```

and using `starmap`:

```python
>>> ~add.starmap(zip(range(10), range(10)))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

is the same as having a task doing:

```python
@app.task
def temp():
    return [add(i, i) for i in range(10)]
```

Both `map` and `starmap` are signature objects, so they can be used as other signatures and combined in groups etc., for example to call the starmap after 10 seconds:

```python
>>> add.starmap(zip(range(10), range(10))).apply_async(countdown=10)
```

Chunks

Chunking lets you divide an iterable of work into pieces, so that if you have one million objects, you can create 10 tasks with hundred thousand objects each.

Some may worry that chunking your tasks results in a degradation of parallelism, but this is rarely true for a busy cluster and in practice since you're avoiding the overhead of messaging it may considerably increase performance.

To create a chunks signature you can use *app.Task.chunks()*:

```
>>> add.chunks(zip(range(100), range(100)), 10)
```

As with *group* the act of sending the messages for the chunks will happen in the current process when called:

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

>>> res = add.chunks(zip(range(100), range(100)), 10)()
>>> res.get()
[[0, 2, 4, 6, 8, 10, 12, 14, 16, 18],
 [20, 22, 24, 26, 28, 30, 32, 34, 36, 38],
 [40, 42, 44, 46, 48, 50, 52, 54, 56, 58],
 [60, 62, 64, 66, 68, 70, 72, 74, 76, 78],
 [80, 82, 84, 86, 88, 90, 92, 94, 96, 98],
 [100, 102, 104, 106, 108, 110, 112, 114, 116, 118],
 [120, 122, 124, 126, 128, 130, 132, 134, 136, 138],
 [140, 142, 144, 146, 148, 150, 152, 154, 156, 158],
 [160, 162, 164, 166, 168, 170, 172, 174, 176, 178],
 [180, 182, 184, 186, 188, 190, 192, 194, 196, 198]]
```

while calling `.apply_async` will create a dedicated task so that the individual tasks are applied in a worker instead:

```
>>> add.chunks(zip(range(100), range(100)), 10).apply_async()
```

You can also convert chunks to a group:

```
>>> group = add.chunks(zip(range(100), range(100)), 10).group()
```

and with the group skew the countdown of each task by increments of one:

```
>>> group.skew(start=1, stop=10)()
```

This means that the first task will have a countdown of one second, the second task a countdown of two seconds, and so on.

3.3.5 Workers Guide

- *Starting the worker*
- *Stopping the worker*
- *Restarting the worker*
- *Process Signals*
- *Variables in file paths*

Starting the worker

Daemonizing

You probably want to use a daemonization tool to start the worker in the background. See *Daemonization* for help starting the worker as a daemon using popular service managers.

You can start the worker in the foreground by executing the command:

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

For a full list of available command-line options see `worker`, or simply do:

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

You can start multiple workers on the same machine, but be sure to name each individual worker by specifying a node name with the `--hostname` argument:

```
$ celery -A proj worker --loglevel=INFO --concurrency=10 -n worker1@%h
$ celery -A proj worker --loglevel=INFO --concurrency=10 -n worker2@%h
$ celery -A proj worker --loglevel=INFO --concurrency=10 -n worker3@%h
```

The `hostname` argument can expand the following variables:

- `%h`: Hostname, including domain name.

- `%n`: Hostname only.

- `%d`: Domain name only.

If the current hostname is *george.example.com*, these will expand to:

Variable	Template	Result
%h	worker1@%h	*worker1@george.example.com*
%n	worker1@%n	*worker1@george*
%d	worker1@%d	*worker1@example.com*

Note for supervisor users

The % sign must be escaped by adding a second one: *%%h*.

Stopping the worker

Shutdown should be accomplished using the TERM signal.

When shutdown is initiated the worker will finish all currently executing tasks before it actually terminates. If these tasks are important, you should wait for it to finish before doing anything drastic, like sending the KILL signal.

If the worker won't shutdown after considerate time, for being stuck in an infinite-loop or similar, you can use the KILL signal to force terminate the worker: but be aware that currently executing tasks will be lost (i.e., unless the tasks have the *acks_late* option set).

Also as processes can't override the KILL signal, the worker will not be able to reap its children; make sure to do so manually. This command usually does the trick:

```
$ pkill -9 -f 'celery worker'
```

If you don't have the **pkill** command on your system, you can use the slightly longer version:

```
$ ps auxww | awk '/celery worker/ {print $2}' | xargs kill -9
```

Restarting the worker

To restart the worker you should send the *TERM* signal and start a new instance. The easiest way to manage workers for development is by using *celery multi*:

```
$ celery multi start 1 -A proj -l info -c4 --pidfile=/var/run/celery/%n.pid
$ celery multi restart 1 --pidfile=/var/run/celery/%n.pid
```

For production deployments you should be using init-scripts or a process supervision system (see *Daemonization*).

Other than stopping, then starting the worker to restart, you can also restart the worker using the HUP signal. Note that the worker will be responsible for restarting itself so this is prone to problems and isn't recommended in production:

```
$ kill -HUP $pid
```

Note: Restarting by HUP only works if the worker is running in the background as a daemon (it doesn't have a controlling terminal).

HUP is disabled on macOS because of a limitation on that platform.

Process Signals

The worker's main process overrides the following signals:

TERM	Warm shutdown, wait for tasks to complete.
QUIT	Cold shutdown, terminate ASAP
USR1	Dump traceback for all active threads.
USR2	Remote debug, see `celery.contrib.rdb`.

Variables in file paths

The file path arguments for `--logfile`, `--pidfile`, and `--statedb` can contain variables that the worker will expand:

Node name replacements

- `%p`: Full node name.
- `%h`: Hostname, including domain name.
- `%n`: Hostname only.
- `%d`: Domain name only.
- `%i`: Prefork pool process index or 0 if MainProcess.
- `%I`: Prefork pool process index with separator.

For example, if the current hostname is `george@foo.example.com` then these will expand to:

- `--logfile=%p.log` -> `george@foo.example.com.log`
- `--logfile=%h.log` -> `foo.example.com.log`
- `--logfile=%n.log` -> `george.log`
- `--logfile=%d.log` -> `example.com.log`

Prefork pool process index

The prefork pool process index specifiers will expand into a different filename depending on the process that'll eventually need to open the file.

This can be used to specify one log file per child process.

Note that the numbers will stay within the process limit even if processes exit or if autoscale/`maxtasksperchild`/time limits are used. That is, the number is the *process index* not the process count or pid.

- `%i` - Pool process index or 0 if MainProcess.

    Where `-n worker1@example.com -c2 -f %n-%i.log` will result in three log files:

    - `worker1-0.log` (main process)
    - `worker1-1.log` (pool process 1)
    - `worker1-2.log` (pool process 2)

- `%I` - Pool process index with separator.

Where `-n worker1@example.com -c2 -f %n%I.log` will result in three log files:

- `worker1.log` (main process)
- `worker1-1.log` (pool process 1)
- `worker1-2.log` (pool process 2)

Concurrency

By default multiprocessing is used to perform concurrent execution of tasks, but you can also use *Eventlet*. The number of worker processes/threads can be changed using the `--concurrency` argument and defaults to the number of CPUs available on the machine.

Number of processes (multiprocessing/prefork pool)

More pool processes are usually better, but there's a cut-off point where adding more pool processes affects performance in negative ways. There's even some evidence to support that having multiple worker instances running, may perform better than having a single worker. For example 3 workers with 10 pool processes each. You need to experiment to find the numbers that works best for you, as this varies based on application, work load, task run times and other factors.

Remote control

New in version 2.0.

The `celery` command

The `celery` program is used to execute remote control commands from the command-line. It supports all of the commands listed below. See *Management Command-line Utilities (inspect/control)* for more information.

---

pool support *prefork, eventlet, gevent, thread*, blocking:*solo* (see note)

broker support *amqp, redis*

Workers have the ability to be remote controlled using a high-priority broadcast message queue. The commands can be directed to all, or a specific list of workers.

Commands can also have replies. The client can then wait for and collect those replies. Since there's no central authority to know how many workers are available in the cluster, there's also no way to estimate how many workers may send a reply, so the client has a configurable timeout — the deadline in seconds for replies to arrive in. This timeout defaults to one second. If the worker doesn't reply within the deadline it doesn't necessarily mean the worker didn't reply, or worse is dead, but may simply be caused by network latency or the worker being slow at processing commands, so adjust the timeout accordingly.

In addition to timeouts, the client can specify the maximum number of replies to wait for. If a destination is specified, this limit is set to the number of destination hosts.

Note: The `solo` pool supports remote control commands, but any task executing will block any waiting control command, so it is of limited use if the worker is very busy. In that case you must increase the timeout waiting for replies in the client.

The `broadcast()` function

This is the client function used to send commands to the workers. Some remote control commands also have higher-level interfaces using *broadcast()* in the background, like *rate_limit()*, and *ping()*.

Sending the *rate_limit* command and keyword arguments:

```
>>> app.control.broadcast('rate_limit',
...                         arguments={'task_name': 'myapp.mytask',
...                                     'rate_limit': '200/m'})
```

This will send the command asynchronously, without waiting for a reply. To request a reply you have to use the *reply* argument:

```
>>> app.control.broadcast('rate_limit', {
...     'task_name': 'myapp.mytask', 'rate_limit': '200/m'}, reply=True)
[{'worker1.example.com': 'New rate limit set successfully'},
 {'worker2.example.com': 'New rate limit set successfully'},
 {'worker3.example.com': 'New rate limit set successfully'}]
```

Using the *destination* argument you can specify a list of workers to receive the command:

```
>>> app.control.broadcast('rate_limit', {
...     'task_name': 'myapp.mytask',
...     'rate_limit': '200/m'}, reply=True,
...                         destination=['worker1@example.com'])
[{'worker1.example.com': 'New rate limit set successfully'}]
```

Of course, using the higher-level interface to set rate limits is much more convenient, but there are commands that can only be requested using *broadcast()*.

Commands

`revoke`: Revoking tasks

> **pool support** all, terminate only supported by prefork
>
> **broker support** *amqp, redis*
>
> **command** `celery -A proj control revoke <task_id>`

All worker nodes keeps a memory of revoked task ids, either in-memory or persistent on disk (see *Persistent revokes*).

When a worker receives a revoke request it will skip executing the task, but it won't terminate an already executing task unless the *terminate* option is set.

Note: The terminate option is a last resort for administrators when a task is stuck. It's not for terminating the task, it's for terminating the process that's executing the task, and that process may have already started processing another task at the point when the signal is sent, so for this reason you must never call this programmatically.

If *terminate* is set the worker child process processing the task will be terminated. The default signal sent is *TERM*, but you can specify this using the *signal* argument. Signal can be the uppercase name of any signal defined in the `signal` module in the Python Standard Library.

Terminating a task also revokes it.

Example

```
>>> result.revoke()

>>> AsyncResult(id).revoke()

>>> app.control.revoke('d9078da5-9915-40a0-bfa1-392c7bde42ed')

>>> app.control.revoke('d9078da5-9915-40a0-bfa1-392c7bde42ed',
...                     terminate=True)

>>> app.control.revoke('d9078da5-9915-40a0-bfa1-392c7bde42ed',
...                     terminate=True, signal='SIGKILL')
```

Revoking multiple tasks

New in version 3.1.

The revoke method also accepts a list argument, where it will revoke several tasks at once.

Example

```
>>> app.control.revoke([
...     '7993b0aa-1f0b-4780-9af0-c47c0858b3f2',
...     'f565793e-b041-4b2b-9ca4-dca22762a55d',
...     'd9d35e03-2997-42d0-a13e-64a66b88a618',
])
```

The `GroupResult.revoke` method takes advantage of this since version 3.1.

Persistent revokes

Revoking tasks works by sending a broadcast message to all the workers, the workers then keep a list of revoked tasks in memory. When a worker starts up it will synchronize revoked tasks with other workers in the cluster.

The list of revoked tasks is in-memory so if all workers restart the list of revoked ids will also vanish. If you want to preserve this list between restarts you need to specify a file for these to be stored in by using the *–statedb* argument to **celery worker**:

```
$ celery -A proj worker -l info --statedb=/var/run/celery/worker.state
```

or if you use **celery multi** you want to create one file per worker instance so use the *%n* format to expand the current node name:

```
celery multi start 2 -l info --statedb=/var/run/celery/%n.state
```

See also *Variables in file paths*

Note that remote control commands must be working for revokes to work. Remote control commands are only supported by the RabbitMQ (amqp) and Redis at this point.

Time Limits

New in version 2.0.

> **pool support** *prefork/gevent*

Soft, or hard?

The time limit is set in two values, *soft* and *hard*. The soft time limit allows the task to catch an exception to clean up before it is killed: the hard timeout isn't catch-able and force terminates the task.

A single task can potentially run forever, if you have lots of tasks waiting for some event that'll never happen you'll block the worker from processing new tasks indefinitely. The best way to defend against this scenario happening is enabling time limits.

The time limit (*–time-limit*) is the maximum number of seconds a task may run before the process executing it is terminated and replaced by a new process. You can also enable a soft time limit (*–soft-time-limit*), this raises an exception the task can catch to clean up before the hard time limit kills it:

```python
from myapp import app
from celery.exceptions import SoftTimeLimitExceeded

@app.task
def mytask():
    try:
        do_work()
    except SoftTimeLimitExceeded:
        clean_up_in_a_hurry()
```

Time limits can also be set using the *task_time_limit* / *task_soft_time_limit* settings.

Note: Time limits don't currently work on platforms that don't support the `SIGUSR1` signal.

Changing time limits at run-time

New in version 2.3.

> **broker support** *amqp, redis*

There's a remote control command that enables you to change both soft and hard time limits for a task — named `time_limit`.

Example changing the time limit for the `tasks.crawl_the_web` task to have a soft time limit of one minute, and a hard time limit of two minutes:

```python
>>> app.control.time_limit('tasks.crawl_the_web',
                           soft=60, hard=120, reply=True)
[{'worker1.example.com': {'ok': 'time limits set successfully'}}]
```

Only tasks that starts executing after the time limit change will be affected.

Rate Limits

Changing rate-limits at run-time

Example changing the rate limit for the *myapp.mytask* task to execute at most 200 tasks of that type every minute:

```
>>> app.control.rate_limit('myapp.mytask', '200/m')
```

The above doesn't specify a destination, so the change request will affect all worker instances in the cluster. If you only want to affect a specific list of workers you can include the `destination` argument:

```
>>> app.control.rate_limit('myapp.mytask', '200/m',
...             destination=['celery@worker1.example.com'])
```

> **Warning:** This won't affect workers with the *worker_disable_rate_limits* setting enabled.

Max tasks per child setting

New in version 2.0.

> **pool support** *prefork*

With this option you can configure the maximum number of tasks a worker can execute before it's replaced by a new process.

This is useful if you have memory leaks you have no control over for example from closed source C extensions.

The option can be set using the workers *--max-tasks-per-child* argument or using the *worker_max_tasks_per_child* setting.

Max memory per child setting

New in version 4.0.

> **pool support** *prefork*

With this option you can configure the maximum amount of resident memory a worker can execute before it's replaced by a new process.

This is useful if you have memory leaks you have no control over for example from closed source C extensions.

The option can be set using the workers *--max-memory-per-child* argument or using the *worker_max_memory_per_child* setting.

Autoscaling

New in version 2.2.

> **pool support** *prefork*, *gevent*

The *autoscaler* component is used to dynamically resize the pool based on load:

- **The autoscaler adds more pool processes when there is work to do,**
    - and starts removing processes when the workload is low.

It's enabled by the *--autoscale* option, which needs two numbers: the maximum and minimum number of pool processes:

```
--autoscale=AUTOSCALE
    Enable autoscaling by providing
    max_concurrency,min_concurrency.  Example:
      --autoscale=10,3 (always keep 3 processes, but grow to
    10 if necessary).
```

You can also define your own rules for the autoscaler by subclassing `Autoscaler`. Some ideas for metrics include load average or the amount of memory available. You can specify a custom autoscaler with the `worker_autoscaler` setting.

Queues

A worker instance can consume from any number of queues. By default it will consume from all queues defined in the `task_queues` setting (that if not specified falls back to the default queue named `celery`).

You can specify what queues to consume from at start-up, by giving a comma separated list of queues to the `-Q` option:

```
$ celery -A proj worker -l info -Q foo,bar,baz
```

If the queue name is defined in `task_queues` it will use that configuration, but if it's not defined in the list of queues Celery will automatically generate a new queue for you (depending on the `task_create_missing_queues` option).

You can also tell the worker to start and stop consuming from a queue at run-time using the remote control commands `add_consumer` and `cancel_consumer`.

Queues: Adding consumers

The `add_consumer` control command will tell one or more workers to start consuming from a queue. This operation is idempotent.

To tell all workers in the cluster to start consuming from a queue named "`foo`" you can use the **celery control** program:

```
$ celery -A proj control add_consumer foo
-> worker1.local: OK
    started consuming from u'foo'
```

If you want to specify a specific worker you can use the `--destination` argument:

```
$ celery -A proj control add_consumer foo -d celery@worker1.local
```

The same can be accomplished dynamically using the `app.control.add_consumer()` method:

```
>>> app.control.add_consumer('foo', reply=True)
[{u'worker1.local': {u'ok': u"already consuming from u'foo'"}}]

>>> app.control.add_consumer('foo', reply=True,
...                          destination=['worker1@example.com'])
[{u'worker1.local': {u'ok': u"already consuming from u'foo'"}}]
```

By now we've only shown examples using automatic queues, If you need more control you can also specify the exchange, routing_key and even other options:

```
>>> app.control.add_consumer(
...      queue='baz',
...      exchange='ex',
...      exchange_type='topic',
...      routing_key='media.*',
...      options={
...          'queue_durable': False,
...          'exchange_durable': False,
...      },
...      reply=True,
...      destination=['w1@example.com', 'w2@example.com'])
```

Queues: Canceling consumers

You can cancel a consumer by queue name using the *cancel_consumer* control command.

To force all workers in the cluster to cancel consuming from a queue you can use the **celery control** program:

```
$ celery -A proj control cancel_consumer foo
```

The *--destination* argument can be used to specify a worker, or a list of workers, to act on the command:

```
$ celery -A proj control cancel_consumer foo -d celery@worker1.local
```

You can also cancel consumers programmatically using the *app.control.cancel_consumer()* method:

```
>>> app.control.cancel_consumer('foo', reply=True)
[{u'worker1.local': {u'ok': u"no longer consuming from u'foo'"}}]
```

Queues: List of active queues

You can get a list of queues that a worker consumes from by using the *active_queues* control command:

```
$ celery -A proj inspect active_queues
[...]
```

Like all other remote control commands this also supports the *--destination* argument used to specify the workers that should reply to the request:

```
$ celery -A proj inspect active_queues -d celery@worker1.local
[...]
```

This can also be done programmatically by using the `app.control.inspect.active_queues()` method:

```
>>> app.control.inspect().active_queues()
[...]

>>> app.control.inspect(['worker1.local']).active_queues()
[...]
```

Inspecting workers

app.control.inspect lets you inspect running workers. It uses remote control commands under the hood.

You can also use the `celery` command to inspect workers, and it supports the same commands as the *app.control* interface.

```
>>> # Inspect all nodes.
>>> i = app.control.inspect()

>>> # Specify multiple nodes to inspect.
>>> i = app.control.inspect(['worker1.example.com',
                             'worker2.example.com'])

>>> # Specify a single node to inspect.
>>> i = app.control.inspect('worker1.example.com')
```

Dump of registered tasks

You can get a list of tasks registered in the worker using the `registered()`:

```
>>> i.registered()
[{'worker1.example.com': ['tasks.add',
                          'tasks.sleeptask']}]
```

Dump of currently executing tasks

You can get a list of active tasks using `active()`:

```
>>> i.active()
[{'worker1.example.com':
    [{'name': 'tasks.sleeptask',
      'id': '32666e9b-809c-41fa-8e93-5ae0c80afbbf',
      'args': '(8,)',
      'kwargs': '{}'}]}]
```

Dump of scheduled (ETA) tasks

You can get a list of tasks waiting to be scheduled by using `scheduled()`:

```
>>> i.scheduled()
[{'worker1.example.com':
    [{'eta': '2010-06-07 09:07:52', 'priority': 0,
      'request': {
        'name': 'tasks.sleeptask',
        'id': '1a7980ea-8b19-413e-91d2-0b74f3844c4d',
        'args': '[1]',
        'kwargs': '{}'}},
     {'eta': '2010-06-07 09:07:53', 'priority': 0,
      'request': {
        'name': 'tasks.sleeptask',
        'id': '49661b9a-aa22-4120-94b7-9ee8031d219d',
        'args': '[2]',
        'kwargs': '{}'}}]}]
```

Note: These are tasks with an ETA/countdown argument, not periodic tasks.

Dump of reserved tasks

Reserved tasks are tasks that have been received, but are still waiting to be executed.

You can get a list of these using `reserved()`:

```
>>> i.reserved()
[{'worker1.example.com':
    [{'name': 'tasks.sleeptask',
      'id': '32666e9b-809c-41fa-8e93-5ae0c80afbbf',
      'args': '(8,)',
      'kwargs': '{}'}]}]
```

Statistics

The remote control command `inspect stats` (or `stats()`) will give you a long list of useful (or not so useful) statistics about the worker:

```
$ celery -A proj inspect stats
```

The output will include the following fields:

- `broker`

    Section for broker information.

    - `connect_timeout`

        Timeout in seconds (int/float) for establishing a new connection.

    - `heartbeat`

        Current heartbeat value (set by client).

    - `hostname`

        Node name of the remote broker.

    - `insist`

        No longer used.

    - `login_method`

        Login method used to connect to the broker.

    - `port`

        Port of the remote broker.

    - `ssl`

        SSL enabled/disabled.

    - `transport`

        Name of transport used (e.g., `amqp` or `redis`)

– transport_options

Options passed to transport.

– uri_prefix

Some transports expects the host name to be a URL.

```
redis+socket:///tmp/redis.sock
```

In this example the URI-prefix will be redis.

– userid

User id used to connect to the broker with.

– virtual_host

Virtual host used.

- clock

Value of the workers logical clock. This is a positive integer and should be increasing every time you receive statistics.

- pid

Process id of the worker instance (Main process).

- pool

Pool-specific section.

– max-concurrency

Max number of processes/threads/green threads.

– max-tasks-per-child

Max number of tasks a thread may execute before being recycled.

– processes

List of PIDs (or thread-id's).

– put-guarded-by-semaphore

Internal

– timeouts

Default values for time limits.

– writes

Specific to the prefork pool, this shows the distribution of writes to each process in the pool when using async I/O.

- prefetch_count

Current prefetch count value for the task consumer.

- rusage

System usage statistics. The fields available may be different on your platform.

From *getrusage(2)*:

– stime

Time spent in operating system code on behalf of this process.

- `utime`

    Time spent executing user instructions.

- `maxrss`

    The maximum resident size used by this process (in kilobytes).

- `idrss`

    Amount of non-shared memory used for data (in kilobytes times ticks of execution)

- `isrss`

    Amount of non-shared memory used for stack space (in kilobytes times ticks of execution)

- `ixrss`

    Amount of memory shared with other processes (in kilobytes times ticks of execution).

- `inblock`

    Number of times the file system had to read from the disk on behalf of this process.

- `oublock`

    Number of times the file system has to write to disk on behalf of this process.

- `majflt`

    Number of page faults that were serviced by doing I/O.

- `minflt`

    Number of page faults that were serviced without doing I/O.

- `msgrcv`

    Number of IPC messages received.

- `msgsnd`

    Number of IPC messages sent.

- `nvcsw`

    Number of times this process voluntarily invoked a context switch.

- `nivcsw`

    Number of times an involuntary context switch took place.

- `nsignals`

    Number of signals received.

- `nswap`

    The number of times this process was swapped entirely out of memory.

- `total`

    Map of task names and the total number of tasks with that type the worker has accepted since start-up.

Additional Commands

Remote shutdown

This command will gracefully shut down the worker remotely:

```
>>> app.control.broadcast('shutdown') # shutdown all workers
>>> app.control.broadcast('shutdown', destination='worker1@example.com')
```

Ping

This command requests a ping from alive workers. The workers reply with the string 'pong', and that's just about it. It will use the default one second timeout for replies unless you specify a custom timeout:

```
>>> app.control.ping(timeout=0.5)
[{'worker1.example.com': 'pong'},
 {'worker2.example.com': 'pong'},
 {'worker3.example.com': 'pong'}]
```

ping() also supports the *destination* argument, so you can specify the workers to ping:

```
>>> ping(['worker2.example.com', 'worker3.example.com'])
[{'worker2.example.com': 'pong'},
 {'worker3.example.com': 'pong'}]
```

Enable/disable events

You can enable/disable events by using the *enable_events*, *disable_events* commands. This is useful to temporarily monitor a worker using **celery events**/**celerymon**.

```
>>> app.control.enable_events()
>>> app.control.disable_events()
```

Writing your own remote control commands

There are two types of remote control commands:

- Inspect command

    Does not have side effects, will usually just return some value found in the worker, like the list of currently registered tasks, the list of active tasks, etc.

- Control command

    Performs side effects, like adding a new queue to consume from.

Remote control commands are registered in the control panel and they take a single argument: the current `ControlDispatch` instance. From there you have access to the active `Consumer` if needed.

Here's an example control command that increments the task prefetch count:

```python
from celery.worker.control import control_command

@control_command(
    args=[('n', int)],
    signature='[N=1]',  # <- used for help on the command-line.
)
def increase_prefetch_count(state, n=1):
    state.consumer.qos.increment_eventually(n)
    return {'ok': 'prefetch count incremented'}
```

Make sure you add this code to a module that is imported by the worker: this could be the same module as where your Celery app is defined, or you can add the module to the *imports* setting.

Restart the worker so that the control command is registered, and now you can call your command using the **celery control** utility:

```
$ celery -A proj control increase_prefetch_count 3
```

You can also add actions to the **celery inspect** program, for example one that reads the current prefetch count:

```python
from celery.worker.control import inspect_command

@inspect_command()
def current_prefetch_count(state):
    return {'prefetch_count': state.consumer.qos.value}
```

After restarting the worker you can now query this value using the **celery inspect** program:

```
$ celery -A proj inspect current_prefetch_count
```

3.3.6 Daemonization

- *Generic init-scripts*
  - *Init-script:* `celeryd`
    * *Example configuration*
    * *Using a login shell*
    * *Example Django configuration*
    * *Available options*
  - *Init-script:* `celerybeat`
    * *Example configuration*
    * *Example Django configuration*
    * *Available options*
  - *Troubleshooting*
- *Usage* `systemd`
  - *Service file: celery.service*
    * *Example configuration*

- – *Service file: celerybeat.service*
- • *Running the worker with superuser privileges (root)*
- • *supervisor*
- • `launchd (macOS)`

Most Linux distributions these days use systemd for managing the lifecycle of system and user services.

You can check if your Linux distribution uses systemd by typing:

```
$ systemd --version
systemd 237
+PAM +AUDIT +SELINUX +IMA +APPARMOR +SMACK +SYSVINIT +UTMP +LIBCRYPTSETUP +GCRYPT
→+GNUTLS +ACL +XZ +LZ4 +SECCOMP +BLKID +ELFUTILS +KMOD -IDN2 +IDN -PCRE2 default-
→hierarchy=hybrid
```

If you have output similar to the above, please refer to *our systemd documentation* for guidance.

However, the init.d script should still work in those Linux distributions as well since systemd provides the systemd-sysv compatibility layer which generates services automatically from the init.d scripts we provide.

If you package Celery for multiple Linux distributions and some do not support systemd or to other Unix systems as well, you may want to refer to *our init.d documentation*.

Generic init-scripts

See the extra/generic-init.d/ directory Celery distribution.

This directory contains generic bash init-scripts for the **celery worker** program, these should run on Linux, FreeBSD, OpenBSD, and other Unix-like platforms.

Init-script: `celeryd`

> **Usage** */etc/init.d/celeryd {start|stop|restart|status}*
>
> **Configuration file** `/etc/default/celeryd`

To configure this script to run the worker properly you probably need to at least tell it where to change directory to when it starts (to find the module containing your app, or your configuration module).

The daemonization script is configured by the file `/etc/default/celeryd`. This is a shell (**sh**) script where you can add environment variables like the configuration options below. To add real environment variables affecting the worker you must also export them (e.g., **export DISPLAY=":0"**)

Superuser privileges required

The init-scripts can only be used by root, and the shell configuration file must also be owned by root.

Unprivileged users don't need to use the init-script, instead they can use the **celery multi** utility (or **celery worker --detach**):

```
$ celery multi start worker1 \
    -A proj \
    --pidfile="$HOME/run/celery/%n.pid" \
    --logfile="$HOME/log/celery/%n%I.log"
```

(continues on next page)

```
$ celery multi restart worker1 \
    -A proj \
    --logfile="$HOME/log/celery/%n%I.log" \
    --pidfile="$HOME/run/celery/%n.pid"

$ celery multi stopwait worker1 --pidfile="$HOME/run/celery/%n.pid"
```

Example configuration

This is an example configuration for a Python project.

`/etc/default/celeryd`:

```
# Names of nodes to start
#   most people will only start one node:
CELERYD_NODES="worker1"
#   but you can also start multiple and configure settings
#   for each in CELERYD_OPTS
#CELERYD_NODES="worker1 worker2 worker3"
#   alternatively, you can specify the number of nodes to start:
#CELERYD_NODES=10

# Absolute or relative path to the 'celery' command:
CELERY_BIN="/usr/local/bin/celery"
#CELERY_BIN="/virtualenvs/def/bin/celery"

# App instance to use
# comment out this line if you don't use an app
CELERY_APP="proj"
# or fully qualified:
#CELERY_APP="proj.tasks:app"

# Where to chdir at start.
CELERYD_CHDIR="/opt/Myproject/"

# Extra command-line arguments to the worker
CELERYD_OPTS="--time-limit=300 --concurrency=8"
# Configure node-specific settings by appending node name to arguments:
#CELERYD_OPTS="--time-limit=300 -c 8 -c:worker2 4 -c:worker3 2 -Ofair:worker1"

# Set logging level to DEBUG
#CELERYD_LOG_LEVEL="DEBUG"

# %n will be replaced with the first part of the nodename.
CELERYD_LOG_FILE="/var/log/celery/%n%I.log"
CELERYD_PID_FILE="/var/run/celery/%n.pid"

# Workers should run as an unprivileged user.
#   You need to create this user manually (or you can choose
#   a user/group combination that already exists (e.g., nobody).
CELERYD_USER="celery"
CELERYD_GROUP="celery"
```

```
# If enabled pid and log directories will be created if missing,
# and owned by the userid/group configured.
CELERY_CREATE_DIRS=1
```

Using a login shell

You can inherit the environment of the `CELERYD_USER` by using a login shell:

```
CELERYD_SU_ARGS="-l"
```

Note that this isn't recommended, and that you should only use this option when absolutely necessary.

Example Django configuration

Django users now uses the exact same template as above, but make sure that the module that defines your Celery app instance also sets a default value for `DJANGO_SETTINGS_MODULE` as shown in the example Django project in *First steps with Django*.

Available options

- CELERY_APP

    App instance to use (value for `--app` argument).

- CELERY_BIN

    Absolute or relative path to the **celery** program. Examples:

    - `celery`

    - `/usr/local/bin/celery`

    - `/virtualenvs/proj/bin/celery`

    - `/virtualenvs/proj/bin/python -m celery`

- CELERYD_NODES

    List of node names to start (separated by space).

- CELERYD_OPTS

    Additional command-line arguments for the worker, see *celery worker –help* for a list. This also supports the extended syntax used by *multi* to configure settings for individual nodes. See *celery multi –help* for some multi-node configuration examples.

- CELERYD_CHDIR

    Path to change directory to at start. Default is to stay in the current directory.

- CELERYD_PID_FILE

    Full path to the PID file. Default is /var/run/celery/%n.pid

- CELERYD_LOG_FILE

Full path to the worker log file. Default is /var/log/celery/%n%I.log **Note**: Using *%I* is important when using the prefork pool as having multiple processes share the same log file will lead to race conditions.

- CELERYD_LOG_LEVEL

    Worker log level. Default is INFO.

- CELERYD_USER

    User to run the worker as. Default is current user.

- CELERYD_GROUP

    Group to run worker as. Default is current user.

- CELERY_CREATE_DIRS

    Always create directories (log directory and pid file directory). Default is to only create directories when no custom logfile/pidfile set.

- CELERY_CREATE_RUNDIR

    Always create pidfile directory. By default only enabled when no custom pidfile location set.

- CELERY_CREATE_LOGDIR

    Always create logfile directory. By default only enable when no custom logfile location set.

Init-script: `celerybeat`

Usage */etc/init.d/celerybeat {start|stop|restart}*

Configuration file /etc/default/celerybeat or /etc/default/celeryd.

Example configuration

This is an example configuration for a Python project:

/etc/default/celerybeat:

```
# Absolute or relative path to the 'celery' command:
CELERY_BIN="/usr/local/bin/celery"
#CELERY_BIN="/virtualenvs/def/bin/celery"

# App instance to use
# comment out this line if you don't use an app
CELERY_APP="proj"
# or fully qualified:
#CELERY_APP="proj.tasks:app"

# Where to chdir at start.
CELERYBEAT_CHDIR="/opt/Myproject/"

# Extra arguments to celerybeat
CELERYBEAT_OPTS="--schedule=/var/run/celery/celerybeat-schedule"
```

Example Django configuration

You should use the same template as above, but make sure the DJANGO_SETTINGS_MODULE variable is set (and exported), and that CELERYD_CHDIR is set to the projects directory:

```
export DJANGO_SETTINGS_MODULE="settings"

CELERYD_CHDIR="/opt/MyProject"
```

Available options

- CELERY_APP

    App instance to use (value for *--app* argument).

- CELERYBEAT_OPTS

    Additional arguments to **celery beat**, see **celery beat --help** for a list of available options.

- CELERYBEAT_PID_FILE

    Full path to the PID file. Default is /var/run/celeryd.pid.

- CELERYBEAT_LOG_FILE

    Full path to the log file. Default is /var/log/celeryd.log.

- CELERYBEAT_LOG_LEVEL

    Log level to use. Default is INFO.

- CELERYBEAT_USER

    User to run beat as. Default is the current user.

- CELERYBEAT_GROUP

    Group to run beat as. Default is the current user.

- CELERY_CREATE_DIRS

    Always create directories (log directory and pid file directory). Default is to only create directories when no custom logfile/pidfile set.

- CELERY_CREATE_RUNDIR

    Always create pidfile directory. By default only enabled when no custom pidfile location set.

- CELERY_CREATE_LOGDIR

    Always create logfile directory. By default only enable when no custom logfile location set.

Troubleshooting

If you can't get the init-scripts to work, you should try running them in *verbose mode*:

```
# sh -x /etc/init.d/celeryd start
```

This can reveal hints as to why the service won't start.

If the worker starts with *"OK"* but exits almost immediately afterwards and there's no evidence in the log file, then there's probably an error but as the daemons standard outputs are already closed you'll not be able to see them anywhere. For this situation you can use the `C_FAKEFORK` environment variable to skip the daemonization step:

```
# C_FAKEFORK=1 sh -x /etc/init.d/celeryd start
```

and now you should be able to see the errors.

Commonly such errors are caused by insufficient permissions to read from, or write to a file, and also by syntax errors in configuration modules, user modules, third-party libraries, or even from Celery itself (if you've found a bug you should *report it*).

Usage `systemd`

- extra/systemd/

  Usage *systemctl {start|stop|restart|status} celery.service*

  Configuration file /etc/conf.d/celery

Service file: celery.service

This is an example systemd file:

/etc/systemd/system/celery.service:

```
[Unit]
Description=Celery Service
After=network.target

[Service]
Type=forking
User=celery
Group=celery
EnvironmentFile=/etc/conf.d/celery
WorkingDirectory=/opt/celery
ExecStart=/bin/sh -c '${CELERY_BIN} multi start ${CELERYD_NODES} \
  -A ${CELERY_APP} --pidfile=${CELERYD_PID_FILE} \
  --logfile=${CELERYD_LOG_FILE} --loglevel=${CELERYD_LOG_LEVEL} ${CELERYD_OPTS}'
ExecStop=/bin/sh -c '${CELERY_BIN} multi stopwait ${CELERYD_NODES} \
  --pidfile=${CELERYD_PID_FILE}'
ExecReload=/bin/sh -c '${CELERY_BIN} multi restart ${CELERYD_NODES} \
  -A ${CELERY_APP} --pidfile=${CELERYD_PID_FILE} \
  --logfile=${CELERYD_LOG_FILE} --loglevel=${CELERYD_LOG_LEVEL} ${CELERYD_OPTS}'

[Install]
WantedBy=multi-user.target
```

Once you've put that file in `/etc/systemd/system`, you should run **systemctl daemon-reload** in order that Systemd acknowledges that file. You should also run that command each time you modify it.

To configure user, group, **chdir** change settings: `User`, `Group`, and `WorkingDirectory` defined in `/etc/systemd/system/celery.service`.

You can also use systemd-tmpfiles in order to create working directories (for logs and pid).

  file */etc/tmpfiles.d/celery.conf*

```
d /var/run/celery 0755 celery celery -
d /var/log/celery 0755 celery celery -
```

Example configuration

This is an example configuration for a Python project:

`/etc/conf.d/celery`:

```
# Name of nodes to start
# here we have a single node
CELERYD_NODES="w1"
# or we could have three nodes:
#CELERYD_NODES="w1 w2 w3"

# Absolute or relative path to the 'celery' command:
CELERY_BIN="/usr/local/bin/celery"
#CELERY_BIN="/virtualenvs/def/bin/celery"

# App instance to use
# comment out this line if you don't use an app
CELERY_APP="proj"
# or fully qualified:
#CELERY_APP="proj.tasks:app"

# How to call manage.py
CELERYD_MULTI="multi"

# Extra command-line arguments to the worker
CELERYD_OPTS="--time-limit=300 --concurrency=8"

# - %n will be replaced with the first part of the nodename.
# - %I will be replaced with the current child process index
#   and is important when using the prefork pool to avoid race conditions.
CELERYD_PID_FILE="/var/run/celery/%n.pid"
CELERYD_LOG_FILE="/var/log/celery/%n%I.log"
CELERYD_LOG_LEVEL="INFO"

# you may wish to add these options for Celery Beat
CELERYBEAT_PID_FILE="/var/run/celery/beat.pid"
CELERYBEAT_LOG_FILE="/var/log/celery/beat.log"
```

Service file: celerybeat.service

This is an example systemd file for Celery Beat:

`/etc/systemd/system/celerybeat.service`:

```
[Unit]
Description=Celery Beat Service
After=network.target

[Service]
Type=simple
```

(continues on next page)

```
User=celery
Group=celery
EnvironmentFile=/etc/conf.d/celery
WorkingDirectory=/opt/celery
ExecStart=/bin/sh -c '${CELERY_BIN} beat  \
  -A ${CELERY_APP} --pidfile=${CELERYBEAT_PID_FILE} \
  --logfile=${CELERYBEAT_LOG_FILE} --loglevel=${CELERYD_LOG_LEVEL}'

[Install]
WantedBy=multi-user.target
```

Running the worker with superuser privileges (root)

Running the worker with superuser privileges is a very dangerous practice. There should always be a workaround to avoid running as root. Celery may run arbitrary code in messages serialized with pickle - this is dangerous, especially when run as root.

By default Celery won't run workers as root. The associated error message may not be visible in the logs but may be seen if `C_FAKEFORK` is used.

To force Celery to run workers as root use `C_FORCE_ROOT`.

When running as root without `C_FORCE_ROOT` the worker will appear to start with *"OK"* but exit immediately after with no apparent errors. This problem may appear when running the project in a new development or production environment (inadvertently) as root.

supervisor

- extra/supervisord/

`launchd` (macOS)

- extra/macOS

3.3.7 Periodic Tasks

- *Introduction*
- *Time Zones*
- *Entries*
    - *Available Fields*
- *Crontab schedules*
- *Solar schedules*
- *Starting the Scheduler*
    - *Using custom scheduler classes*

Introduction

celery beat is a scheduler; It kicks off tasks at regular intervals, that are then executed by available worker nodes in the cluster.

By default the entries are taken from the *beat_schedule* setting, but custom stores can also be used, like storing the entries in a SQL database.

You have to ensure only a single scheduler is running for a schedule at a time, otherwise you'd end up with duplicate tasks. Using a centralized approach means the schedule doesn't have to be synchronized, and the service can operate without using locks.

Time Zones

The periodic task schedules uses the UTC time zone by default, but you can change the time zone used using the *timezone* setting.

An example time zone could be *Europe/London*:

```
timezone = 'Europe/London'
```

This setting must be added to your app, either by configuring it directly using (`app.conf.timezone = 'Europe/London'`), or by adding it to your configuration module if you have set one up using `app.config_from_object`. See *Configuration* for more information about configuration options.

The default scheduler (storing the schedule in the `celerybeat-schedule` file) will automatically detect that the time zone has changed, and so will reset the schedule itself, but other schedulers may not be so smart (e.g., the Django database scheduler, see below) and in that case you'll have to reset the schedule manually.

Django Users

Celery recommends and is compatible with the new `USE_TZ` setting introduced in Django 1.4.

For Django users the time zone specified in the `TIME_ZONE` setting will be used, or you can specify a custom time zone for Celery alone by using the *timezone* setting.

The database scheduler won't reset when timezone related settings change, so you must do this manually:

```
$ python manage.py shell
>>> from djcelery.models import PeriodicTask
>>> PeriodicTask.objects.update(last_run_at=None)
```

Django-Celery only supports Celery 4.0 and below, for Celery 4.0 and above, do as follow:

```
$ python manage.py shell
>>> from django_celery_beat.models import PeriodicTask
>>> PeriodicTask.objects.update(last_run_at=None)
```

Entries

To call a task periodically you have to add an entry to the beat schedule list.

```
from celery import Celery
from celery.schedules import crontab
```

(continues on next page)

```python
app = Celery()

@app.on_after_configure.connect
def setup_periodic_tasks(sender, **kwargs):
    # Calls test('hello') every 10 seconds.
    sender.add_periodic_task(10.0, test.s('hello'), name='add every 10')

    # Calls test('world') every 30 seconds
    sender.add_periodic_task(30.0, test.s('world'), expires=10)

    # Executes every Monday morning at 7:30 a.m.
    sender.add_periodic_task(
        crontab(hour=7, minute=30, day_of_week=1),
        test.s('Happy Mondays!'),
    )

@app.task
def test(arg):
    print(arg)
```

Setting these up from within the *on_after_configure* handler means that we'll not evaluate the app at module level when using test.s().

The *add_periodic_task()* function will add the entry to the *beat_schedule* setting behind the scenes, and the same setting can also be used to set up periodic tasks manually:

Example: Run the *tasks.add* task every 30 seconds.

```python
app.conf.beat_schedule = {
    'add-every-30-seconds': {
        'task': 'tasks.add',
        'schedule': 30.0,
        'args': (16, 16)
    },
}
app.conf.timezone = 'UTC'
```

Note: If you're wondering where these settings should go then please see *Configuration*. You can either set these options on your app directly or you can keep a separate module for configuration.

If you want to use a single item tuple for *args*, don't forget that the constructor is a comma, and not a pair of parentheses.

Using a timedelta for the schedule means the task will be sent in 30 second intervals (the first task will be sent 30 seconds after *celery beat* starts, and then every 30 seconds after the last run).

A Crontab like schedule also exists, see the section on *Crontab schedules*.

Like with **cron**, the tasks may overlap if the first task doesn't complete before the next. If that's a concern you should use a locking strategy to ensure only one instance can run at a time (see for example *Ensuring a task is only executed one at a time*).

Available Fields

- *task*

The name of the task to execute.

- *schedule*

    The frequency of execution.

    This can be the number of seconds as an integer, a `timedelta`, or a `crontab`. You can also define your own custom schedule types, by extending the interface of `schedule`.

- *args*

    Positional arguments (`list` or `tuple`).

- *kwargs*

    Keyword arguments (`dict`).

- *options*

    Execution options (`dict`).

    This can be any argument supported by `apply_async()` – *exchange*, *routing_key*, *expires*, and so on.

- *relative*

    If *relative* is true `timedelta` schedules are scheduled "by the clock." This means the frequency is rounded to the nearest second, minute, hour or day depending on the period of the `timedelta`.

    By default *relative* is false, the frequency isn't rounded and will be relative to the time when **celery beat** was started.

Crontab schedules

If you want more control over when the task is executed, for example, a particular time of day or day of the week, you can use the *crontab* schedule type:

```python
from celery.schedules import crontab

app.conf.beat_schedule = {
    # Executes every Monday morning at 7:30 a.m.
    'add-every-monday-morning': {
        'task': 'tasks.add',
        'schedule': crontab(hour=7, minute=30, day_of_week=1),
        'args': (16, 16),
    },
}
```

The syntax of these Crontab expressions are very flexible.

Some examples:

Example	Meaning
`crontab()`	Execute every minute.
`crontab(minute=0, hour=0)`	Execute daily at midnight.
`crontab(minute=0, hour='*/3')`	Execute every three hours: midnight, 3am, 6am, 9am, noon, 3pm, 6pm, 9pm.
`crontab(minute=0,` ` hour='0,3,6,9,12,` `   15,18,21')`	Same as previous.
`crontab(minute='*/15')`	Execute every 15 minutes.
`crontab(day_of_week='sunday')`	Execute every minute (!) at Sundays.
`crontab(minute='*',` ` hour='*',` `   day_of_week='sun')`	Same as previous.
`crontab(minute='*/10',` ` hour='3,17,` `   22', day_of_week='thu,fri')`	Execute every ten minutes, but only between 3-4 am, 5-6 pm, and 10-11 pm on Thursdays or Fridays.
`crontab(minute=0, hour='*/2,*/3')`	Execute every even hour, and every hour divisible by three. This means: at every hour *except*: 1am, 5am, 7am, 11am, 1pm, 5pm, 7pm, 11pm
`crontab(minute=0, hour='*/5')`	Execute hour divisible by 5. This means that it is triggered at 3pm, not 5pm (since 3pm equals the 24-hour clock value of "15", which is divisible by 5).
`crontab(minute=0, hour='*/3,8-17')`	Execute every hour divisible by 3, and every hour during office hours (8am-5pm).
`crontab(0, 0, day_of_month='2')`	Execute on the second day of every month.
`crontab(0, 0,` ` day_of_month='2-30/2')`	Execute on every even numbered day.
`crontab(0, 0,` ` day_of_month='1-7,` `   15-21')`	Execute on the first and third weeks of the month.
`crontab(0, 0, day_of_month='11',` `   month_of_year='5')`	Execute on the eleventh of May every year.
`crontab(0, 0,` ` month_of_year='*/3')`	Execute every day on the first month of every quarter.

See `celery.schedules.crontab` for more documentation.

Solar schedules

If you have a task that should be executed according to sunrise, sunset, dawn or dusk, you can use the `solar` schedule type:

```python
from celery.schedules import solar

app.conf.beat_schedule = {
    # Executes at sunset in Melbourne
    'add-at-melbourne-sunset': {
```

```
        'task': 'tasks.add',
        'schedule': solar('sunset', -37.81753, 144.96715),
        'args': (16, 16),
    },
}
```

The arguments are simply: `solar(event, latitude, longitude)`

Be sure to use the correct sign for latitude and longitude:

Sign	Argument	Meaning
+	`latitude`	North
−	`latitude`	South
+	`longitude`	East
−	`longitude`	West

Possible event types are:

Event	Meaning
`dawn_astronomical`	Execute at the moment after which the sky is no longer completely dark. This is when the sun is 18 degrees below the horizon.
`dawn_nautical`	Execute when there's enough sunlight for the horizon and some objects to be distinguishable; formally, when the sun is 12 degrees below the horizon.
`dawn_civil`	Execute when there's enough light for objects to be distinguishable so that outdoor activities can commence; formally, when the Sun is 6 degrees below the horizon.
`sunrise`	Execute when the upper edge of the sun appears over the eastern horizon in the morning.
`solar_noon`	Execute when the sun is highest above the horizon on that day.
`sunset`	Execute when the trailing edge of the sun disappears over the western horizon in the evening.
`dusk_civil`	Execute at the end of civil twilight, when objects are still distinguishable and some stars and planets are visible. Formally, when the sun is 6 degrees below the horizon.
`dusk_nautical`	Execute when the sun is 12 degrees below the horizon. Objects are no longer distinguishable, and the horizon is no longer visible to the naked eye.
`dusk_astronomical`	Execute at the moment after which the sky becomes completely dark; formally, when the sun is 18 degrees below the horizon.

All solar events are calculated using UTC, and are therefore unaffected by your timezone setting.

In polar regions, the sun may not rise or set every day. The scheduler is able to handle these cases (i.e., a `sunrise` event won't run on a day when the sun doesn't rise). The one exception is `solar_noon`, which is formally defined as the moment the sun transits the celestial meridian, and will occur every day even if the sun is below the horizon.

Twilight is defined as the period between dawn and sunrise; and between sunset and dusk. You can schedule an event according to "twilight" depending on your definition of twilight (civil, nautical, or astronomical), and whether you want the event to take place at the beginning or end of twilight, using the appropriate event from the list above.

See *celery.schedules.solar* for more documentation.

Starting the Scheduler

To start the **celery beat** service:

```
$ celery -A proj beat
```

You can also embed *beat* inside the worker by enabling the workers `-B` option, this is convenient if you'll never run more than one worker node, but it's not commonly used and for that reason isn't recommended for production use:

```
$ celery -A proj worker -B
```

Beat needs to store the last run times of the tasks in a local database file (named *celerybeat-schedule* by default), so it needs access to write in the current directory, or alternatively you can specify a custom location for this file:

```
$ celery -A proj beat -s /home/celery/var/run/celerybeat-schedule
```

Note: To daemonize beat see *Daemonization*.

Using custom scheduler classes

Custom scheduler classes can be specified on the command-line (the `--scheduler` argument).

The default scheduler is the `celery.beat.PersistentScheduler`, that simply keeps track of the last run times in a local `shelve` database file.

There's also the django-celery-beat extension that stores the schedule in the Django database, and presents a convenient admin interface to manage periodic tasks at runtime.

To install and use this extension:

1. Use **pip** to install the package:

   ```
   $ pip install django-celery-beat
   ```

2. Add the `django_celery_beat` module to `INSTALLED_APPS` in your Django project' `settings.py`:

   ```
   INSTALLED_APPS = (
       ...,
       'django_celery_beat',
   )
   ```

   Note that there is no dash in the module name, only underscores.

3. Apply Django database migrations so that the necessary tables are created:

   ```
   $ python manage.py migrate
   ```

4. Start the **celery beat** service using the `django_celery_beat.schedulers:DatabaseScheduler` scheduler:

   ```
   $ celery -A proj beat -l info --scheduler django_celery_beat.
   →schedulers:DatabaseScheduler
   ```

   Note: You may also add this as the `beat_scheduler` setting directly.

5. Visit the Django-Admin interface to set up some periodic tasks.

3.3.8 Routing Tasks

Note: Alternate routing concepts like topic and fanout is not available for all transports, please consult the transport comparison table.

Basics

Automatic routing

The simplest way to do routing is to use the `task_create_missing_queues` setting (on by default).

With this setting on, a named queue that's not already defined in `task_queues` will be created automatically. This makes it easy to perform simple routing tasks.

Say you have two servers, *x*, and *y* that handle regular tasks, and one server *z*, that only handles feed related tasks. You can use this configuration:

```
task_routes = {'feed.tasks.import_feed': {'queue': 'feeds'}}
```

With this route enabled import feed tasks will be routed to the *"feeds"* queue, while all other tasks will be routed to the default queue (named *"celery"* for historical reasons).

Alternatively, you can use glob pattern matching, or even regular expressions, to match all tasks in the `feed.tasks` name-space:

```
app.conf.task_routes = {'feed.tasks.*': {'queue': 'feeds'}}
```

If the order of matching patterns is important you should specify the router in *items* format instead:

```
task_routes = ([
    ('feed.tasks.*', {'queue': 'feeds'}),
    ('web.tasks.*', {'queue': 'web'}),
    (re.compile(r'(video|image)\.tasks\..*'), {'queue': 'media'}),
],)
```

Note: The `task_routes` setting can either be a dictionary, or a list of router objects, so in this case we need to specify the setting as a tuple containing a list.

After installing the router, you can start server *z* to only process the feeds queue like this:

```
user@z:/$ celery -A proj worker -Q feeds
```

You can specify as many queues as you want, so you can make this server process the default queue as well:

```
user@z:/$ celery -A proj worker -Q feeds,celery
```

Changing the name of the default queue

You can change the name of the default queue by using the following configuration:

```
app.conf.task_default_queue = 'default'
```

How the queues are defined

The point with this feature is to hide the complex AMQP protocol for users with only basic needs. However – you may still be interested in how these queues are declared.

A queue named *"video"* will be created with the following settings:

```
{'exchange': 'video',
 'exchange_type': 'direct',
 'routing_key': 'video'}
```

The non-AMQP backends like *Redis* or *SQS* don't support exchanges, so they require the exchange to have the same name as the queue. Using this design ensures it will work for them as well.

Manual routing

Say you have two servers, *x*, and *y* that handle regular tasks, and one server *z*, that only handles feed related tasks, you can use this configuration:

```
from kombu import Queue

app.conf.task_default_queue = 'default'
app.conf.task_queues = (
    Queue('default',    routing_key='task.#'),
    Queue('feed_tasks', routing_key='feed.#'),
)
app.conf.task_default_exchange = 'tasks'
app.conf.task_default_exchange_type = 'topic'
app.conf.task_default_routing_key = 'task.default'
```

task_queues is a list of Queue instances. If you don't set the exchange or exchange type values for a key, these will be taken from the *task_default_exchange* and *task_default_exchange_type* settings.

To route a task to the *feed_tasks* queue, you can add an entry in the *task_routes* setting:

```
task_routes = {
        'feeds.tasks.import_feed': {
            'queue': 'feed_tasks',
            'routing_key': 'feed.import',
        },
}
```

You can also override this using the *routing_key* argument to Task.apply_async(), or send_task():

```
>>> from feeds.tasks import import_feed
>>> import_feed.apply_async(args=['http://cnn.com/rss'],
...                         queue='feed_tasks',
...                         routing_key='feed.import')
```

To make server *z* consume from the feed queue exclusively you can start it with the *celery worker -Q* option:

```
user@z:/$ celery -A proj worker -Q feed_tasks --hostname=z@%h
```

Servers *x* and *y* must be configured to consume from the default queue:

```
user@x:/$ celery -A proj worker -Q default --hostname=x@%h
user@y:/$ celery -A proj worker -Q default --hostname=y@%h
```

If you want, you can even have your feed processing worker handle regular tasks as well, maybe in times when there's a lot of work to do:

```
user@z:/$ celery -A proj worker -Q feed_tasks,default --hostname=z@%h
```

If you have another queue but on another exchange you want to add, just specify a custom exchange and exchange type:

```
from kombu import Exchange, Queue

app.conf.task_queues = (
    Queue('feed_tasks',    routing_key='feed.#'),
    Queue('regular_tasks', routing_key='task.#'),
    Queue('image_tasks',   exchange=Exchange('mediatasks', type='direct'),
                           routing_key='image.compress'),
)
```

If you're confused about these terms, you should read up on AMQP.

See also:

In addition to the *Redis Message Priorities* below, there's Rabbits and Warrens, an excellent blog post describing queues and exchanges. There's also The *CloudAMQP tutorial*, For users of RabbitMQ the RabbitMQ FAQ could be useful as a source of information.

Special Routing Options

RabbitMQ Message Priorities

> **supported transports** RabbitMQ

New in version 4.0.

Queues can be configured to support priorities by setting the `x-max-priority` argument:

```python
from kombu import Exchange, Queue

app.conf.task_queues = [
    Queue('tasks', Exchange('tasks'), routing_key='tasks',
          queue_arguments={'x-max-priority': 10}),
]
```

A default value for all queues can be set using the *task_queue_max_priority* setting:

```python
app.conf.task_queue_max_priority = 10
```

A default priority for all tasks can also be specified using the *task_default_priority* setting:

```python
app.conf.task_default_priority = 5
```

Redis Message Priorities

> **supported transports** Redis

While the Celery Redis transport does honor the priority field, Redis itself has no notion of priorities. Please read this note before attempting to implement priorities with Redis as you may experience some unexpected behavior.

To start scheduling tasks based on priorities you need to configure queue_order_strategy transport option.

The priority support is implemented by creating n lists for each queue. This means that even though there are 10 (0-9) priority levels, these are consolidated into 4 levels by default to save resources. This means that a queue named celery will really be split into 4 queues:

```python
['celery0', 'celery3', 'celery6', 'celery9']
```

If you want more priority levels you can set the priority_steps transport option:

```python
app.conf.broker_transport_options = {
    'priority_steps': list(range(10)),
    'queue_order_strategy': 'priority',
}
```

That said, note that this will never be as good as priorities implemented at the server level, and may be approximate at best. But it may still be good enough for your application.

AMQP Primer

Messages

A message consists of headers and a body. Celery uses headers to store the content type of the message and its content encoding. The content type is usually the serialization format used to serialize the message. The body contains the name of the task to execute, the task id (UUID), the arguments to apply it with and some additional meta-data – like the number of retries or an ETA.

This is an example task message represented as a Python dictionary:

```
{'task': 'myapp.tasks.add',
 'id': '54086c5e-6193-4575-8308-dbab76798756',
 'args': [4, 4],
 'kwargs': {}}
```

Producers, consumers, and brokers

The client sending messages is typically called a *publisher*, or a *producer*, while the entity receiving messages is called a *consumer*.

The *broker* is the message server, routing messages from producers to consumers.

You're likely to see these terms used a lot in AMQP related material.

Exchanges, queues, and routing keys

1. Messages are sent to exchanges.

2. An exchange routes messages to one or more queues. Several exchange types exists, providing different ways to do routing, or implementing different messaging scenarios.

3. The message waits in the queue until someone consumes it.

4. The message is deleted from the queue when it has been acknowledged.

The steps required to send and receive messages are:

1. Create an exchange

2. Create a queue

3. Bind the queue to the exchange.

Celery automatically creates the entities necessary for the queues in `task_queues` to work (except if the queue's *auto_declare* setting is set to `False`).

Here's an example queue configuration with three queues; One for video, one for images, and one default queue for everything else:

```python
from kombu import Exchange, Queue

app.conf.task_queues = (
    Queue('default', Exchange('default'), routing_key='default'),
    Queue('videos',  Exchange('media'),   routing_key='media.video'),
    Queue('images',  Exchange('media'),   routing_key='media.image'),
)
app.conf.task_default_queue = 'default'
```

(continues on next page)

```
app.conf.task_default_exchange_type = 'direct'
app.conf.task_default_routing_key = 'default'
```

Exchange types

The exchange type defines how the messages are routed through the exchange. The exchange types defined in the standard are *direct*, *topic*, *fanout* and *headers*. Also non-standard exchange types are available as plug-ins to RabbitMQ, like the last-value-cache plug-in by Michael Bridgen.

Direct exchanges

Direct exchanges match by exact routing keys, so a queue bound by the routing key *video* only receives messages with that routing key.

Topic exchanges

Topic exchanges matches routing keys using dot-separated words, and the wild-card characters: `*` (matches a single word), and `#` (matches zero or more words).

With routing keys like `usa.news`, `usa.weather`, `norway.news`, and `norway.weather`, bindings could be `*.news` (all news), `usa.#` (all items in the USA), or `usa.weather` (all USA weather items).

Related API commands

exchange.declare(exchange_name, type, passive, durable, auto_delete, internal)
   Declares an exchange by name.

   See `amqp:Channel.exchange_declare`.

   > **Keyword Arguments**
   >
   > - **passive** – Passive means the exchange won't be created, but you can use this to check if the exchange already exists.
   >
   > - **durable** – Durable exchanges are persistent (i.e., they survive a broker restart).
   >
   > - **auto_delete** – This means the exchange will be deleted by the broker when there are no more queues using it.

queue.**declare**(*queue_name*, *passive*, *durable*, *exclusive*, *auto_delete*)
   Declares a queue by name.

   See `amqp:Channel.queue_declare`

   Exclusive queues can only be consumed from by the current connection. Exclusive also implies *auto_delete*.

queue.**bind**(*queue_name*, *exchange_name*, *routing_key*)
   Binds a queue to an exchange with a routing key.

   Unbound queues won't receive messages, so this is necessary.

   See `amqp:Channel.queue_bind`

---

queue.**delete**(*name*, *if_unused=False*, *if_empty=False*)
>   Deletes a queue and its binding.

>   See `amqp:Channel.queue_delete`

exchange.**delete**(*name*, *if_unused=False*)
>   Deletes an exchange.

>   See `amqp:Channel.exchange_delete`

Note: Declaring doesn't necessarily mean "create". When you declare you *assert* that the entity exists and that it's operable. There's no rule as to whom should initially create the exchange/queue/binding, whether consumer or producer. Usually the first one to need it will be the one to create it.

Hands-on with the API

Celery comes with a tool called **celery amqp** that's used for command line access to the AMQP API, enabling access to administration tasks like creating/deleting queues and exchanges, purging queues or sending messages. It can also be used for non-AMQP brokers, but different implementation may not implement all commands.

You can write commands directly in the arguments to **celery amqp**, or just start with no arguments to start it in shell-mode:

```
$ celery -A proj amqp
-> connecting to amqp://guest@localhost:5672/.
-> connected.
1>
```

Here `1>` is the prompt. The number 1, is the number of commands you have executed so far. Type `help` for a list of commands available. It also supports auto-completion, so you can start typing a command and then hit the *tab* key to show a list of possible matches.

Let's create a queue you can send messages to:

```
$ celery -A proj amqp
1> exchange.declare testexchange direct
ok.
2> queue.declare testqueue
ok. queue:testqueue messages:0 consumers:0.
3> queue.bind testqueue testexchange testkey
ok.
```

This created the direct exchange `testexchange`, and a queue named `testqueue`. The queue is bound to the exchange using the routing key `testkey`.

From now on all messages sent to the exchange `testexchange` with routing key `testkey` will be moved to this queue. You can send a message by using the `basic.publish` command:

```
4> basic.publish 'This is a message!' testexchange testkey
ok.
```

Now that the message is sent you can retrieve it again. You can use the `basic.get` command here, that polls for new messages on the queue in a synchronous manner (this is OK for maintenance tasks, but for services you want to use `basic.consume` instead)

Pop a message off the queue:

```
5> basic.get testqueue
{'body': 'This is a message!',
 'delivery_info': {'delivery_tag': 1,
                   'exchange': u'testexchange',
                   'message_count': 0,
                   'redelivered': False,
                   'routing_key': u'testkey'},
 'properties': {}}
```

AMQP uses acknowledgment to signify that a message has been received and processed successfully. If the message hasn't been acknowledged and consumer channel is closed, the message will be delivered to another consumer.

Note the delivery tag listed in the structure above; Within a connection channel, every received message has a unique delivery tag, This tag is used to acknowledge the message. Also note that delivery tags aren't unique across connections, so in another client the delivery tag *1* might point to a different message than in this channel.

You can acknowledge the message you received using `basic.ack`:

```
6> basic.ack 1
ok.
```

To clean up after our test session you should delete the entities you created:

```
7> queue.delete testqueue
ok. 0 messages deleted.
8> exchange.delete testexchange
ok.
```

Routing Tasks

Defining queues

In Celery available queues are defined by the *task_queues* setting.

Here's an example queue configuration with three queues; One for video, one for images, and one default queue for everything else:

```
default_exchange = Exchange('default', type='direct')
media_exchange = Exchange('media', type='direct')

app.conf.task_queues = (
    Queue('default', default_exchange, routing_key='default'),
    Queue('videos', media_exchange, routing_key='media.video'),
    Queue('images', media_exchange, routing_key='media.image')
)
app.conf.task_default_queue = 'default'
app.conf.task_default_exchange = 'default'
app.conf.task_default_routing_key = 'default'
```

Here, the *task_default_queue* will be used to route tasks that doesn't have an explicit route.

The default exchange, exchange type, and routing key will be used as the default routing values for tasks, and as the default values for entries in *task_queues*.

Multiple bindings to a single queue are also supported. Here's an example of two routing keys that are both bound to the same queue:

```
from kombu import Exchange, Queue, binding

media_exchange = Exchange('media', type='direct')

CELERY_QUEUES = (
    Queue('media', [
        binding(media_exchange, routing_key='media.video'),
        binding(media_exchange, routing_key='media.image'),
    ]),
)
```

Specifying task destination

The destination for a task is decided by the following (in order):

1. The routing arguments to `Task.apply_async()`.

2. Routing related attributes defined on the `Task` itself.

3. The *Routers* defined in `task_routes`.

It's considered best practice to not hard-code these settings, but rather leave that as configuration options by using *Routers*; This is the most flexible approach, but sensible defaults can still be set as task attributes.

Routers

A router is a function that decides the routing options for a task.

All you need to define a new router is to define a function with the signature `(name, args, kwargs, options, task=None, **kw)`:

```
def route_task(name, args, kwargs, options, task=None, **kw):
        if name == 'myapp.tasks.compress_video':
            return {'exchange': 'video',
                    'exchange_type': 'topic',
                    'routing_key': 'video.compress'}
```

If you return the `queue` key, it'll expand with the defined settings of that queue in `task_queues`:

```
{'queue': 'video', 'routing_key': 'video.compress'}
```

becomes –>

```
{'queue': 'video',
 'exchange': 'video',
 'exchange_type': 'topic',
 'routing_key': 'video.compress'}
```

You install router classes by adding them to the `task_routes` setting:

```
task_routes = (route_task,)
```

Router functions can also be added by name:

```
task_routes = ('myapp.routers.route_task',)
```

For simple task name -> route mappings like the router example above, you can simply drop a dict into *task_routes* to get the same behavior:

```
task_routes = {
    'myapp.tasks.compress_video': {
        'queue': 'video',
        'routing_key': 'video.compress',
    },
}
```

The routers will then be traversed in order, it will stop at the first router returning a true value, and use that as the final route for the task.

You can also have multiple routers defined in a sequence:

```
task_routes = [
    route_task,
    {
        'myapp.tasks.compress_video': {
            'queue': 'video',
            'routing_key': 'video.compress',
        },
    },
]
```

The routers will then be visited in turn, and the first to return a value will be chosen.

If you're using Redis or RabbitMQ you can also specify the queue's default priority in the route.

```
task_routes = {
    'myapp.tasks.compress_video': {
        'queue': 'video',
        'routing_key': 'video.compress',
        'priority': 10,
    },
}
```

Similarly, calling *apply_async* on a task will override that default priority.

```
task.apply_async(priority=0)
```

Priority Order and Cluster Responsiveness

It is important to note that, due to worker prefetching, if a bunch of tasks submitted at the same time they may be out of priority order at first. Disabling worker prefetching will prevent this issue, but may cause less than ideal performance for small, fast tasks. In most cases, simply reducing *worker_prefetch_multiplier* to 1 is an easier and cleaner way to increase the responsiveness of your system without the costs of disabling prefetching entirely.

Note that priorities values are sorted in reverse when using the redis broker: 0 being highest priority.

Broadcast

Celery can also support broadcast routing. Here is an example exchange `broadcast_tasks` that delivers copies of tasks to all workers connected to it:

```
from kombu.common import Broadcast

app.conf.task_queues = (Broadcast('broadcast_tasks'),)
app.conf.task_routes = {
    'tasks.reload_cache': {
        'queue': 'broadcast_tasks',
        'exchange': 'broadcast_tasks'
    }
}
```

Now the `tasks.reload_cache` task will be sent to every worker consuming from this queue.

Here is another example of broadcast routing, this time with a **celery beat** schedule:

```
from kombu.common import Broadcast
from celery.schedules import crontab

app.conf.task_queues = (Broadcast('broadcast_tasks'),)

app.conf.beat_schedule = {
    'test-task': {
        'task': 'tasks.reload_cache',
        'schedule': crontab(minute=0, hour='*/3'),
        'options': {'exchange': 'broadcast_tasks'}
    },
}
```

Broadcast & Results

Note that Celery result doesn't define what happens if two tasks have the same task_id. If the same task is distributed to more than one worker, then the state history may not be preserved.

It's a good idea to set the `task.ignore_result` attribute in this case.

3.3.9 Monitoring and Management Guide

- *Introduction*
- *Workers*
  - *Management Command-line Utilities (`inspect`/`control`)*
    - *Commands*
    - *Specifying destination nodes*
  - *Flower: Real-time Celery web-monitor*
    - *Features*
    - *Usage*
  - *celery events: Curses Monitor*
- *RabbitMQ*
  - *Inspecting queues*

Introduction

There are several tools available to monitor and inspect Celery clusters.

This document describes some of these, as as well as features related to monitoring, like events and broadcast commands.

Workers

Management Command-line Utilities (`inspect`/`control`)

`celery` can also be used to inspect and manage worker nodes (and to some degree tasks).

To list all the commands available do:

```
$ celery help
```

or to get help for a specific command do:

```
$ celery <command> --help
```

Commands

- **shell**: Drop into a Python shell.

  The locals will include the `celery` variable: this is the current app. Also all known tasks will be automatically added to locals (unless the `--without-tasks` flag is set).

  Uses Ipython, bpython, or regular **python** in that order if installed. You can force an implementation using `--ipython`, `--bpython`, or `--python`.

- **status**: List active nodes in this cluster

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

- **result**: Show the result of a task

  ```
  $ celery -A proj result -t tasks.add 4e196aa4-0141-4601-8138-7aa33db0f577
  ```

  Note that you can omit the name of the task as long as the task doesn't use a custom result backend.

- **purge**: Purge messages from all configured task queues.

  This command will remove all messages from queues configured in the `CELERY_QUEUES` setting:

  > **Warning:** There's no undo for this operation, and messages will be permanently deleted!

  ```
  $ celery -A proj purge
  ```

  You can also specify the queues to purge using the *-Q* option:

  ```
  $ celery -A proj purge -Q celery,foo,bar
  ```

  and exclude queues from being purged using the *-X* option:

  ```
  $ celery -A proj purge -X celery
  ```

- **inspect active**: List active tasks

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

  These are all the tasks that are currently being executed.

- **inspect scheduled**: List scheduled ETA tasks

  ```
  $ celery -A proj inspect scheduled
  ```

  These are tasks reserved by the worker when they have an *eta* or *countdown* argument set.

- **inspect reserved**: List reserved tasks

  ```
  $ celery -A proj inspect reserved
  ```

  This will list all tasks that have been prefetched by the worker, and is currently waiting to be executed (doesn't include tasks with an ETA value set).

- **inspect revoked**: List history of revoked tasks

```
$ celery -A proj inspect revoked
```

- **inspect registered**: List registered tasks

```
$ celery -A proj inspect registered
```

- **inspect stats**: Show worker statistics (see *Statistics*)

```
$ celery -A proj inspect stats
```

- **inspect query_task**: Show information about task(s) by id.

    Any worker having a task in this set of ids reserved/active will respond with status and information.

```
$ celery -A proj inspect query_task e9f6c8f0-fec9-4ae8-a8c6-cf8c8451d4f8
```

    You can also query for information about multiple tasks:

```
$ celery -A proj inspect query_task id1 id2 ... idN
```

- **control enable_events**: Enable events

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

- **control disable_events**: Disable events

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

- **migrate**: Migrate tasks from one broker to another (**EXPERIMENTAL**).

```
$ celery -A proj migrate redis://localhost amqp://localhost
```

    This command will migrate all the tasks on one broker to another. As this command is new and experimental you should be sure to have a backup of the data before proceeding.

Note: All `inspect` and `control` commands supports a `--timeout` argument, This is the number of seconds to wait for responses. You may have to increase this timeout if you're not getting a response due to latency.

Specifying destination nodes

By default the inspect and control commands operates on all workers. You can specify a single, or a list of workers by using the `--destination` argument:

```
$ celery -A proj inspect -d w1@e.com,w2@e.com reserved

$ celery -A proj control -d w1@e.com,w2@e.com enable_events
```

Flower: Real-time Celery web-monitor

Flower is a real-time web based monitor and administration tool for Celery. It's under active development, but is already an essential tool. Being the recommended monitor for Celery, it obsoletes the Django-Admin monitor, `celerymon` and the `ncurses` based monitor.

Flower is pronounced like "flow", but you can also use the botanical version if you prefer.

Features

- Real-time monitoring using Celery Events
    - Task progress and history
    - Ability to show task details (arguments, start time, run-time, and more)
    - Graphs and statistics
- Remote Control
    - View worker status and statistics
    - Shutdown and restart worker instances
    - Control worker pool size and autoscale settings
    - View and modify the queues a worker instance consumes from
    - View currently running tasks
    - View scheduled tasks (ETA/countdown)
    - View reserved and revoked tasks
    - Apply time and rate limits
    - Configuration viewer
    - Revoke or terminate tasks
- HTTP API
    - List workers
    - Shut down a worker
    - Restart worker's pool
    - Grow worker's pool
    - Shrink worker's pool
    - Autoscale worker pool
    - Start consuming from a queue
    - Stop consuming from a queue
    - List tasks
    - List (seen) task types
    - Get a task info
    - Execute a task
    - Execute a task by name
    - Get a task result
    - Change soft and hard time limits for a task
    - Change rate limit for a task
    - Revoke a task

- OpenID authentication

Screenshots

More screenshots:

Usage

You can use pip to install Flower:

```
$ pip install flower
```

Running the flower command will start a web-server that you can visit:

```
$ celery -A proj flower
```

The default port is http://localhost:5555, but you can change this using the –port argument:

```
$ celery -A proj flower --port=5555
```

Broker URL can also be passed through the `--broker` argument :

```
$ celery flower --broker=amqp://guest:guest@localhost:5672//
or
$ celery flower --broker=redis://guest:guest@localhost:6379/0
```
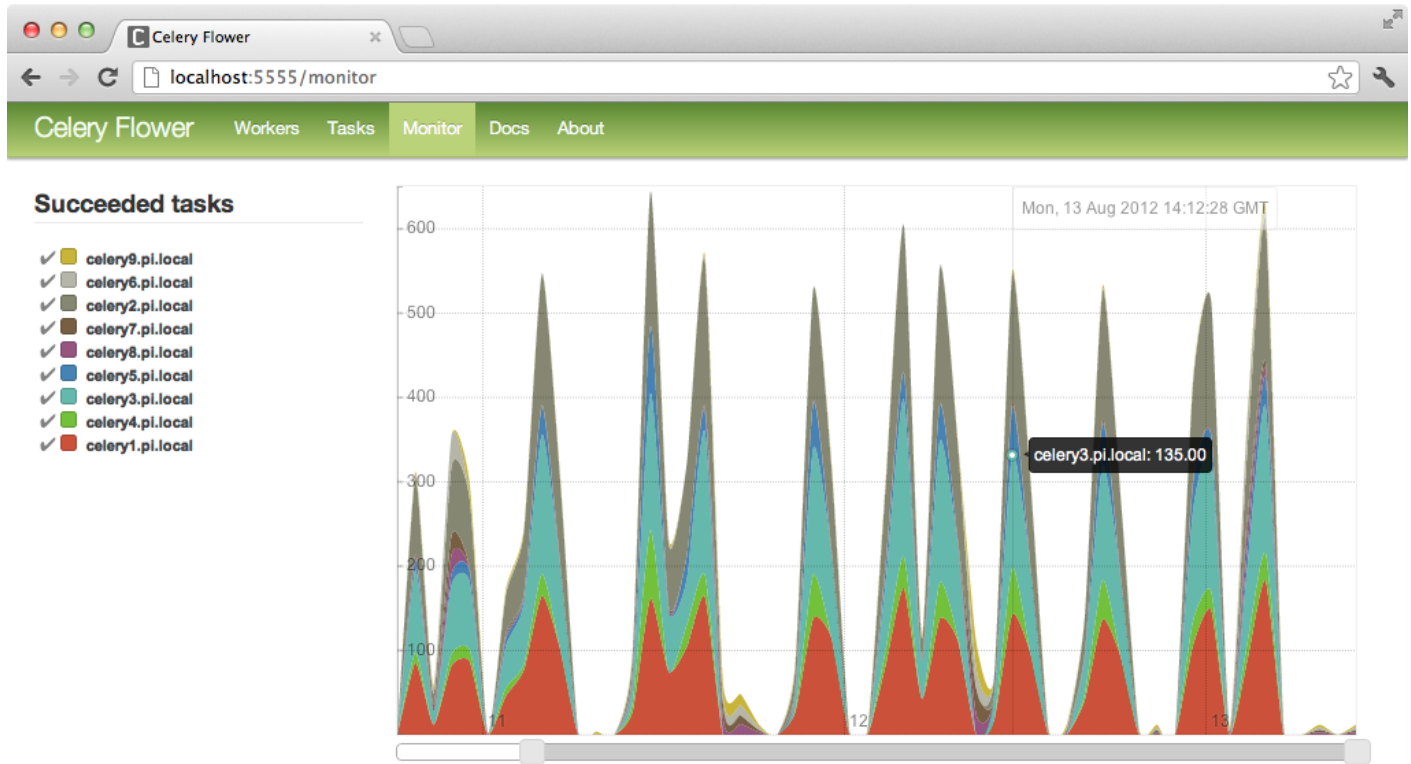
Then, you can visit flower in your web browser :

```
$ open http://localhost:5555
```

Flower has many more features than are detailed here, including authorization options. Check out the official documentation for more information.

celery events: Curses Monitor

New in version 2.0.

celery events is a simple curses monitor displaying task and worker history. You can inspect the result and traceback of tasks, and it also supports some management commands like rate limiting and shutting down workers. This monitor was started as a proof of concept, and you probably want to use Flower instead.

Starting:

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

You should see a screen like:

celery events is also used to start snapshot cameras (see *Snapshots*:

```
$ celery -A proj events --camera=<camera-class> --frequency=1.0
```

and it includes a tool to dump events to `stdout`:

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

For a complete list of options use `--help`:

```
$ celery events --help
```

RabbitMQ

To manage a Celery cluster it is important to know how RabbitMQ can be monitored.

RabbitMQ ships with the rabbitmqctl(1) command, with this you can list queues, exchanges, bindings, queue lengths, the memory usage of each queue, as well as manage users, virtual hosts and their permissions.

Note: The default virtual host (`"/"`) is used in these examples, if you use a custom virtual host you have to add the `-p` argument to the command, for example: `rabbitmqctl list_queues -p my_vhost ...`

Inspecting queues

Finding the number of tasks in a queue:

```
$ rabbitmqctl list_queues name messages messages_ready \
                          messages_unacknowledged
```

Here *messages_ready* is the number of messages ready for delivery (sent but not received), *messages_unacknowledged* is the number of messages that's been received by a worker but not acknowledged yet (meaning it is in progress, or has been reserved). *messages* is the sum of ready and unacknowledged messages.

Finding the number of workers currently consuming from a queue:

```
$ rabbitmqctl list_queues name consumers
```

Finding the amount of memory allocated to a queue:

```
$ rabbitmqctl list_queues name memory
```

> **Tip** Adding the `-q` option to rabbitmqctl(1) makes the output easier to parse.

Redis

If you're using Redis as the broker, you can monitor the Celery cluster using the *redis-cli(1)* command to list lengths of queues.

Inspecting queues

Finding the number of tasks in a queue:

```
$ redis-cli -h HOST -p PORT -n DATABASE_NUMBER llen QUEUE_NAME
```

The default queue is named *celery*. To get all available queues, invoke:

```
$ redis-cli -h HOST -p PORT -n DATABASE_NUMBER keys \*
```

Note: Queue keys only exists when there are tasks in them, so if a key doesn't exist it simply means there are no messages in that queue. This is because in Redis a list with no elements in it is automatically removed, and hence it won't show up in the *keys* command output, and *llen* for that list returns 0.

Also, if you're using Redis for other purposes, the output of the *keys* command will include unrelated values stored in the database. The recommended way around this is to use a dedicated *DATABASE_NUMBER* for Celery, you can also use database numbers to separate Celery applications from each other (virtual hosts), but this won't affect the monitoring events used by for example Flower as Redis pub/sub commands are global rather than database based.

Munin

This is a list of known Munin plug-ins that can be useful when maintaining a Celery cluster.

- `rabbitmq-munin`: Munin plug-ins for RabbitMQ.

    https://github.com/ask/rabbitmq-munin

- `celery_tasks`: Monitors the number of times each task type has been executed (requires *celerymon*).

    https://github.com/munin-monitoring/contrib/blob/master/plugins/celery/celery_tasks

- `celery_tasks_states`: Monitors the number of tasks in each state (requires *celerymon*).

    https://github.com/munin-monitoring/contrib/blob/master/plugins/celery/celery_tasks_states

Events

The worker has the ability to send a message whenever some event happens. These events are then captured by tools like Flower, and **celery events** to monitor the cluster.

Snapshots

New in version 2.1.

Even a single worker can produce a huge amount of events, so storing the history of all events on disk may be very expensive.

A sequence of events describes the cluster state in that time period, by taking periodic snapshots of this state you can keep all history, but still only periodically write it to disk.

To take snapshots you need a Camera class, with this you can define what should happen every time the state is captured; You can write it to a database, send it by email or something else entirely.

celery events is then used to take snapshots with the camera, for example if you want to capture state every 2 seconds using the camera `myapp.Camera` you run **celery events** with the following arguments:

```
$ celery -A proj events -c myapp.Camera --frequency=2.0
```

Custom Camera

Cameras can be useful if you need to capture events and do something with those events at an interval. For real-time event processing you should use `app.events.Receiver` directly, like in *Real-time processing*.

Here is an example camera, dumping the snapshot to screen:

```python
from pprint import pformat

from celery.events.snapshot import Polaroid

class DumpCam(Polaroid):
    clear_after = True  # clear after flush (incl, state.event_count).

    def on_shutter(self, state):
        if not state.event_count:
            # No new events since last snapshot.
            return
        print('Workers: {0}'.format(pformat(state.workers, indent=4)))
        print('Tasks: {0}'.format(pformat(state.tasks, indent=4)))
        print('Total: {0.event_count} events, {0.task_count} tasks'.format(
            state))
```

See the API reference for *celery.events.state* to read more about state objects.

Now you can use this cam with **celery events** by specifying it with the *-c* option:

```console
$ celery -A proj events -c myapp.DumpCam --frequency=2.0
```

Or you can use it programmatically like this:

```python
from celery import Celery
from myapp import DumpCam

def main(app, freq=1.0):
    state = app.events.State()
    with app.connection() as connection:
        recv = app.events.Receiver(connection, handlers={'*': state.event})
        with DumpCam(state, freq=freq):
            recv.capture(limit=None, timeout=None)

if __name__ == '__main__':
    app = Celery(broker='amqp://guest@localhost//')
    main(app)
```

Real-time processing

To process events in real-time you need the following

- An event consumer (this is the `Receiver`)

- A set of handlers called when events come in.

    You can have different handlers for each event type, or a catch-all handler can be used ('*')

- State (optional)

    `app.events.State` is a convenient in-memory representation of tasks and workers in the cluster that's updated as events come in.

It encapsulates solutions for many common things, like checking if a worker is still alive (by verifying heart-beats), merging event fields together as events come in, making sure time-stamps are in sync, and so on.

Combining these you can easily process events in real-time:

```python
from celery import Celery


def my_monitor(app):
    state = app.events.State()

    def announce_failed_tasks(event):
        state.event(event)
        # task name is sent only with -received event, and state
        # will keep track of this for us.
        task = state.tasks.get(event['uuid'])

        print('TASK FAILED: %s[%s] %s' % (
            task.name, task.uuid, task.info(),))

    with app.connection() as connection:
        recv = app.events.Receiver(connection, handlers={
                'task-failed': announce_failed_tasks,
                '*': state.event,
        })
        recv.capture(limit=None, timeout=None, wakeup=True)

if __name__ == '__main__':
    app = Celery(broker='amqp://guest@localhost//')
    my_monitor(app)
```

Note: The `wakeup` argument to `capture` sends a signal to all workers to force them to send a heartbeat. This way you can immediately see workers when the monitor starts.

You can listen to specific events by specifying the handlers:

```python
from celery import Celery

def my_monitor(app):
    state = app.events.State()

    def announce_failed_tasks(event):
        state.event(event)
        # task name is sent only with -received event, and state
        # will keep track of this for us.
        task = state.tasks.get(event['uuid'])

        print('TASK FAILED: %s[%s] %s' % (
            task.name, task.uuid, task.info(),))

    with app.connection() as connection:
        recv = app.events.Receiver(connection, handlers={
                'task-failed': announce_failed_tasks,
        })
        recv.capture(limit=None, timeout=None, wakeup=True)
```

```python
if __name__ == '__main__':
    app = Celery(broker='amqp://guest@localhost//')
    my_monitor(app)
```

Event Reference

This list contains the events sent by the worker, and their arguments.

Task Events

task-sent

> **signature** `task-sent(uuid, name, args, kwargs, retries, eta, expires, queue, exchange, routing_key, root_id, parent_id)`

Sent when a task message is published and the `task_send_sent_event` setting is enabled.

task-received

> **signature** `task-received(uuid, name, args, kwargs, retries, eta, hostname, timestamp, root_id, parent_id)`

Sent when the worker receives a task.

task-started

> **signature** `task-started(uuid, hostname, timestamp, pid)`

Sent just before the worker executes the task.

task-succeeded

> **signature** `task-succeeded(uuid, result, runtime, hostname, timestamp)`

Sent if the task executed successfully.

Run-time is the time it took to execute the task using the pool. (Starting from the task is sent to the worker pool, and ending when the pool result handler callback is called).

task-failed

> **signature** `task-failed(uuid, exception, traceback, hostname, timestamp)`

Sent if the execution of the task failed.

task-rejected

> **signature** `task-rejected(uuid, requeued)`

The task was rejected by the worker, possibly to be re-queued or moved to a dead letter queue.

task-revoked

> **signature** `task-revoked(uuid, terminated, signum, expired)`

Sent if the task has been revoked (Note that this is likely to be sent by more than one worker).

- **`terminated` is set to true if the task process was terminated,** and the `signum` field set to the signal used.

- `expired` is set to true if the task expired.

task-retried

> **signature** `task-retried(uuid, exception, traceback, hostname, timestamp)`

Sent if the task failed, but will be retried in the future.

Worker Events

worker-online

> **signature** `worker-online(hostname, timestamp, freq, sw_ident, sw_ver, sw_sys)`

The worker has connected to the broker and is online.

- *hostname*: Nodename of the worker.
- *timestamp*: Event time-stamp.
- *freq*: Heartbeat frequency in seconds (float).
- *sw_ident*: Name of worker software (e.g., `py-celery`).
- *sw_ver*: Software version (e.g., 2.2.0).
- *sw_sys*: Operating System (e.g., Linux/Darwin).

worker-heartbeat

> **signature** `worker-heartbeat(hostname, timestamp, freq, sw_ident, sw_ver, sw_sys, active, processed)`

Sent every minute, if the worker hasn't sent a heartbeat in 2 minutes, it is considered to be offline.

- *hostname*: Nodename of the worker.
- *timestamp*: Event time-stamp.
- *freq*: Heartbeat frequency in seconds (float).
- *sw_ident*: Name of worker software (e.g., `py-celery`).

- *sw_ver*: Software version (e.g., 2.2.0).

- *sw_sys*: Operating System (e.g., Linux/Darwin).

- *active*: Number of currently executing tasks.

- *processed*: Total number of tasks processed by this worker.

worker-offline

> **signature** `worker-offline(hostname, timestamp, freq, sw_ident, sw_ver, sw_sys)`

The worker has disconnected from the broker.

3.3.10 Security

> - *[Introduction](#)*
> - *[Areas of Concern](#)*
>   - *[Broker](#)*
>   - *[Client](#)*
>   - *[Worker](#)*
> - *[Serializers](#)*
> - *[Message Signing](#)*
> - *[Intrusion Detection](#)*
>   - *[Logs](#)*
>   - *[Tripwire](#)*

Introduction

While Celery is written with security in mind, it should be treated as an unsafe component.

Depending on your Security Policy, there are various steps you can take to make your Celery installation more secure.

Areas of Concern

Broker

It's imperative that the broker is guarded from unwanted access, especially if accessible to the public. By default, workers trust that the data they get from the broker hasn't been tampered with. See *Message Signing* for information on how to make the broker connection more trustworthy.

The first line of defense should be to put a firewall in front of the broker, allowing only white-listed machines to access it.

Keep in mind that both firewall misconfiguration, and temporarily disabling the firewall, is common in the real world. Solid security policy includes monitoring of firewall equipment to detect if they've been disabled, be it accidentally or on purpose.

In other words, one shouldn't blindly trust the firewall either.

If your broker supports fine-grained access control, like RabbitMQ, this is something you should look at enabling. See for example http://www.rabbitmq.com/access-control.html.

If supported by your broker backend, you can enable end-to-end SSL encryption and authentication using `broker_use_ssl`.

Client

In Celery, "client" refers to anything that sends messages to the broker, for example web-servers that apply tasks.

Having the broker properly secured doesn't matter if arbitrary messages can be sent through a client.

[Need more text here]

Worker

The default permissions of tasks running inside a worker are the same ones as the privileges of the worker itself. This applies to resources, such as; memory, file-systems, and devices.

An exception to this rule is when using the multiprocessing based task pool, which is currently the default. In this case, the task will have access to any memory copied as a result of the `fork()` call, and access to memory contents written by parent tasks in the same worker child process.

Limiting access to memory contents can be done by launching every task in a subprocess (`fork()` + `execve()`).

Limiting file-system and device access can be accomplished by using chroot, jail, sandboxing, virtual machines, or other mechanisms as enabled by the platform or additional software.

Note also that any task executed in the worker will have the same network access as the machine on which it's running. If the worker is located on an internal network it's recommended to add firewall rules for outbound traffic.

Serializers

The default serializer is JSON since version 4.0, but since it has only support for a restricted set of types you may want to consider using pickle for serialization instead.

The *pickle* serializer is convenient as it can serialize almost any Python object, even functions with some work, but for the same reasons *pickle* is inherently insecure*[0], and should be avoided whenever clients are untrusted or unauthenticated.

You can disable untrusted content by specifying a white-list of accepted content-types in the `accept_content` setting:

New in version 3.0.18.

Note: This setting was first supported in version 3.0.18. If you're running an earlier version it will simply be ignored, so make sure you're running a version that supports it.

---

[0] https://blog.nelhage.com/2011/03/exploiting-pickle/

```
accept_content = ['json']
```

This accepts a list of serializer names and content-types, so you could also specify the content type for json:

```
accept_content = ['application/json']
```

Celery also comes with a special *auth* serializer that validates communication between Celery clients and workers, making sure that messages originates from trusted sources. Using *Public-key cryptography* the *auth* serializer can verify the authenticity of senders, to enable this read *Message Signing* for more information.

Message Signing

Celery can use the cryptography library to sign message using *Public-key cryptography*, where messages sent by clients are signed using a private key and then later verified by the worker using a public certificate.

Optimally certificates should be signed by an official Certificate Authority, but they can also be self-signed.

To enable this you should configure the `task_serializer` setting to use the *auth* serializer. Enforcing the workers to only accept signed messages, you should set *accept_content* to *['auth']*. For additional signing of the event protocol, set *event_serializer* to *auth*. Also required is configuring the paths used to locate private keys and certificates on the file-system: the `security_key`, `security_certificate`, and `security_cert_store` settings respectively. You can tweak the signing algorithm with `security_digest`.

With these configured it's also necessary to call the `celery.setup_security()` function. Note that this will also disable all insecure serializers so that the worker won't accept messages with untrusted content types.

This is an example configuration using the *auth* serializer, with the private key and certificate files located in */etc/ssl*.

```
app = Celery()
app.conf.update(
    security_key='/etc/ssl/private/worker.key'
    security_certificate='/etc/ssl/certs/worker.pem'
    security_cert_store='/etc/ssl/certs/*.pem',
    security_digest='sha256',
    task_serializer='auth',
    event_serializer='auth',
    accept_content=['auth']
)
app.setup_security()
```

Note: While relative paths aren't disallowed, using absolute paths is recommended for these files.

Also note that the *auth* serializer won't encrypt the contents of a message, so if needed this will have to be enabled separately.

Intrusion Detection

The most important part when defending your systems against intruders is being able to detect if the system has been compromised.

Logs

Logs are usually the first place to look for evidence of security breaches, but they're useless if they can be tampered with.

A good solution is to set up centralized logging with a dedicated logging server. Access to it should be restricted. In addition to having all of the logs in a single place, if configured correctly, it can make it harder for intruders to tamper with your logs.

This should be fairly easy to setup using syslog (see also syslog-ng and rsyslog). Celery uses the `logging` library, and already has support for using syslog.

A tip for the paranoid is to send logs using UDP and cut the transmit part of the logging server's network cable :-)

Tripwire

Tripwire is a (now commercial) data integrity tool, with several open source implementations, used to keep cryptographic hashes of files in the file-system, so that administrators can be alerted when they change. This way when the damage is done and your system has been compromised you can tell exactly what files intruders have changed (password files, logs, back-doors, root-kits, and so on). Often this is the only way you'll be able to detect an intrusion.

Some open source implementations include:

- OSSEC
- Samhain
- Open Source Tripwire
- AIDE

Also, the ZFS file-system comes with built-in integrity checks that can be used.

3.3.11 Optimizing

Introduction

The default configuration makes a lot of compromises. It's not optimal for any single case, but works well enough for most situations.

There are optimizations that can be applied based on specific use cases.

Optimizations can apply to different properties of the running environment, be it the time tasks take to execute, the amount of memory used, or responsiveness at times of high load.

Ensuring Operations

In the book Programming Pearls, Jon Bentley presents the concept of back-of-the-envelope calculations by asking the question;

> How much water flows out of the Mississippi River in a day?

The point of this exercise*[0] is to show that there's a limit to how much data a system can process in a timely manner. Back of the envelope calculations can be used as a means to plan for this ahead of time.

---

[0] The chapter is available to read for free here: The back of the envelope. The book is a classic text. Highly recommended.

In Celery; If a task takes 10 minutes to complete, and there are 10 new tasks coming in every minute, the queue will never be empty. This is why it's very important that you monitor queue lengths!

A way to do this is by *using Munin*. You should set up alerts, that'll notify you as soon as any queue has reached an unacceptable size. This way you can take appropriate action like adding new worker nodes, or revoking unnecessary tasks.

General Settings

Broker Connection Pools

The broker connection pool is enabled by default since version 2.5.

You can tweak the *broker_pool_limit* setting to minimize contention, and the value should be based on the number of active threads/green-threads using broker connections.

Using Transient Queues

Queues created by Celery are persistent by default. This means that the broker will write messages to disk to ensure that the tasks will be executed even if the broker is restarted.

But in some cases it's fine that the message is lost, so not all tasks require durability. You can create a *transient* queue for these tasks to improve performance:

```python
from kombu import Exchange, Queue

task_queues = (
    Queue('celery', routing_key='celery'),
    Queue('transient', Exchange('transient', delivery_mode=1),
          routing_key='transient', durable=False),
)
```

or by using *task_routes*:

```python
task_routes = {
    'proj.tasks.add': {'queue': 'celery', 'delivery_mode': 'transient'}
}
```

The `delivery_mode` changes how the messages to this queue are delivered. A value of one means that the message won't be written to disk, and a value of two (default) means that the message can be written to disk.

To direct a task to your new transient queue you can specify the queue argument (or use the *task_routes* setting):

```python
task.apply_async(args, queue='transient')
```

For more information see the *routing guide*.

Worker Settings

Prefetch Limits

Prefetch is a term inherited from AMQP that's often misunderstood by users.

The prefetch limit is a **limit** for the number of tasks (messages) a worker can reserve for itself. If it is zero, the worker will keep consuming messages, not respecting that there may be other available worker nodes that may be able to process them sooner†[0], or that the messages may not even fit in memory.

The workers' default prefetch count is the `worker_prefetch_multiplier` setting multiplied by the number of concurrency slots‡[0] (processes/threads/green-threads).

If you have many tasks with a long duration you want the multiplier value to be *one*: meaning it'll only reserve one task per worker process at a time.

However – If you have many short-running tasks, and throughput/round trip latency is important to you, this number should be large. The worker is able to process more tasks per second if the messages have already been prefetched, and is available in memory. You may have to experiment to find the best value that works for you. Values like 50 or 150 might make sense in these circumstances. Say 64, or 128.

If you have a combination of long- and short-running tasks, the best option is to use two worker nodes that are configured separately, and route the tasks according to the run-time (see *Routing Tasks*).

Reserve one task at a time

The task message is only deleted from the queue after the task is *acknowledged*, so if the worker crashes before acknowledging the task, it can be redelivered to another worker (or the same after recovery).

When using the default of early acknowledgment, having a prefetch multiplier setting of *one*, means the worker will reserve at most one extra task for every worker process: or in other words, if the worker is started with `-c 10`, the worker may reserve at most 20 tasks (10 acknowledged tasks executing, and 10 unacknowledged reserved tasks) at any time.

Often users ask if disabling "prefetching of tasks" is possible, but what they really mean by that, is to have a worker only reserve as many tasks as there are worker processes (10 unacknowledged tasks for `-c 10`)

That's possible, but not without also enabling *late acknowledgment*. Using this option over the default behavior means a task that's already started executing will be retried in the event of a power failure or the worker instance being killed abruptly, so this also means the task must be *idempotent*

See also:

Notes at *Should I use retry or acks_late?*.

You can enable this behavior by using the following configuration options:

```
task_acks_late = True
worker_prefetch_multiplier = 1
```

Prefork pool prefetch settings

The prefork pool will asynchronously send as many tasks to the processes as it can and this means that the processes are, in effect, prefetching tasks.

This benefits performance but it also means that tasks may be stuck waiting for long running tasks to complete:

---

[0] RabbitMQ and other brokers deliver messages round-robin, so this doesn't apply to an active system. If there's no prefetch limit and you restart the cluster, there will be timing delays between nodes starting. If there are 3 offline nodes and one active node, all messages will be delivered to the active node.

[0] This is the concurrency setting; `worker_concurrency` or the `celery worker -c` option.

```
-> send task T1 to process A
# A executes T1
-> send task T2 to process B
# B executes T2
<- T2 complete sent by process B

-> send task T3 to process A
# A still executing T1, T3 stuck in local buffer and won't start until
# T1 returns, and other queued tasks won't be sent to idle processes
<- T1 complete sent by process A
# A executes T3
```

The worker will send tasks to the process as long as the pipe buffer is writable. The pipe buffer size varies based on the operating system: some may have a buffer as small as 64KB but on recent Linux versions the buffer size is 1MB (can only be changed system wide).

You can disable this prefetching behavior by enabling the `-O fair` worker option:

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

With this option enabled the worker will only write to processes that are available for work, disabling the prefetch behavior:

```
-> send task T1 to process A
# A executes T1
-> send task T2 to process B
# B executes T2
<- T2 complete sent by process B

-> send T3 to process B
# B executes T3

<- T3 complete sent by process B
<- T1 complete sent by process A
```

3.3.12 Debugging

Debugging Tasks Remotely (using pdb)

Basics

`celery.contrib.rdb` is an extended version of `pdb` that enables remote debugging of processes that doesn't have terminal access.

Example usage:

```python
from celery import task
from celery.contrib import rdb


@task()
def add(x, y):
    result = x + y
    rdb.set_trace()  # <- set break-point
    return result
```

set_trace() sets a break-point at the current location and creates a socket you can telnet into to remotely debug your task.

The debugger may be started by multiple processes at the same time, so rather than using a fixed port the debugger will search for an available port, starting from the base port (6900 by default). The base port can be changed using the environment variable *CELERY_RDB_PORT*.

By default the debugger will only be available from the local host, to enable access from the outside you have to set the environment variable *CELERY_RDB_HOST*.

When the worker encounters your break-point it'll log the following information:

```
[INFO/MainProcess] Received task:
    tasks.add[d7261c71-4962-47e5-b342-2448bedd20e8]
[WARNING/PoolWorker-1] Remote Debugger:6900:
    Please telnet 127.0.0.1 6900.  Type `exit` in session to continue.
[2011-01-18 14:25:44,119: WARNING/PoolWorker-1] Remote Debugger:6900:
    Waiting for client...
```

If you telnet the port specified you'll be presented with a *pdb* shell:

```
$ telnet localhost 6900
Connected to localhost.
Escape character is '^]'.
> /opt/devel/demoapp/tasks.py(128)add()
-> return result
(Pdb)
```

Enter `help` to get a list of available commands, It may be a good idea to read the Python Debugger Manual if you have never used *pdb* before.

To demonstrate, we'll read the value of the `result` variable, change it and continue execution of the task:

```
(Pdb) result
4
(Pdb) result = 'hello from rdb'
(Pdb) continue
Connection closed by foreign host.
```

The result of our vandalism can be seen in the worker logs:

```
[2011-01-18 14:35:36,599: INFO/MainProcess] Task
    tasks.add[d7261c71-4962-47e5-b342-2448bedd20e8] succeeded
    in 61.481s: 'hello from rdb'
```

Tips

Enabling the break-point signal

If the environment variable CELERY_RDBSIG is set, the worker will open up an rdb instance whenever the *SIGUSR2* signal is sent. This is the case for both main and worker processes.

For example starting the worker with:

```
$ CELERY_RDBSIG=1 celery worker -l info
```

You can start an rdb session for any of the worker processes by executing:

```
$ kill -USR2 <pid>
```

3.3.13 Concurrency

Release 4.4

Date Mar 02, 2020

Concurrency with Eventlet

Introduction

The Eventlet homepage describes it as a concurrent networking library for Python that allows you to change how you run your code, not how you write it.

- It uses epoll(4) or libevent for highly scalable non-blocking I/O.

- Coroutines ensure that the developer uses a blocking style of programming that's similar to threading, but provide the benefits of non-blocking I/O.

- The event dispatch is implicit: meaning you can easily use Eventlet from the Python interpreter, or as a small part of a larger application.

Celery supports Eventlet as an alternative execution pool implementation and in some cases superior to prefork. However, you need to ensure one task doesn't block the event loop too long. Generally, CPU-bound operations don't go well with Eventlet. Also note that some libraries, usually with C extensions, cannot be monkeypatched and therefore cannot benefit from using Eventlet. Please refer to their documentation if you are not sure. For example, pylibmc does not allow cooperation with Eventlet but psycopg2 does when both of them are libraries with C extensions.

The prefork pool can take use of multiple processes, but how many is often limited to a few processes per CPU. With Eventlet you can efficiently spawn hundreds, or thousands of green threads. In an informal test with a feed hub system the Eventlet pool could fetch and process hundreds of feeds every second, while the prefork pool spent 14 seconds processing 100 feeds. Note that this is one of the applications async I/O is especially good at (asynchronous HTTP requests). You may want a mix of both Eventlet and prefork workers, and route tasks according to compatibility or what works best.

Enabling Eventlet

You can enable the Eventlet pool by using the `celery worker -P` worker option.

```
$ celery -A proj worker -P eventlet -c 1000
```

Examples

See the Eventlet examples directory in the Celery distribution for some examples taking use of Eventlet support.

3.3.14 Signals

Signals allow decoupled applications to receive notifications when certain actions occur elsewhere in the application.

Celery ships with many signals that your application can hook into to augment behavior of certain actions.

Basics

Several kinds of events trigger signals, you can connect to these signals to perform actions as they trigger.

Example connecting to the *after_task_publish* signal:

```python
from celery.signals import after_task_publish

@after_task_publish.connect
def task_sent_handler(sender=None, headers=None, body=None, **kwargs):
    # information about task are located in headers for task messages
    # using the task protocol version 2.
    info = headers if 'task' in headers else body
    print('after_task_publish for task id {info[id]}'.format(
        info=info,
    ))
```

Some signals also have a sender you can filter by. For example the *after_task_publish* signal uses the task name as a sender, so by providing the sender argument to *connect* you can connect your handler to be called every time a task with name *"proj.tasks.add"* is published:

```python
@after_task_publish.connect(sender='proj.tasks.add')
def task_sent_handler(sender=None, headers=None, body=None, **kwargs):
    # information about task are located in headers for task messages
    # using the task protocol version 2.
    info = headers if 'task' in headers else body
    print('after_task_publish for task id {info[id]}'.format(
        info=info,
    ))
```

Signals use the same implementation as django.core.dispatch. As a result other keyword parameters (e.g., signal) are passed to all signal handlers by default.

The best practice for signal handlers is to accept arbitrary keyword arguments (i.e., **kwargs). That way new Celery versions can add additional arguments without breaking user code.

Signals

Task Signals

before_task_publish

New in version 3.1.

Dispatched before a task is published. Note that this is executed in the process sending the task.

Sender is the name of the task being sent.

Provides arguments:

- `body`

    Task message body.

    This is a mapping containing the task message fields, see *Version 2* and *Version 1* for a reference of possible fields that can be defined.

- `exchange`

    Name of the exchange to send to or a `Exchange` object.

- `routing_key`

    Routing key to use when sending the message.

- `headers`

    Application headers mapping (can be modified).

- `properties`

    Message properties (can be modified)

- `declare`

    List of entities (`Exchange`, `Queue`, or `binding` to declare before publishing the message. Can be modified.

- `retry_policy`

    Mapping of retry options. Can be any argument to `kombu.Connection.ensure()` and can be modified.

after_task_publish

Dispatched when a task has been sent to the broker. Note that this is executed in the process that sent the task.

Sender is the name of the task being sent.

Provides arguments:

- `headers`

    The task message headers, see *Version 2* and *Version 1* for a reference of possible fields that can be defined.

- `body`

    The task message body, see *Version 2* and *Version 1* for a reference of possible fields that can be defined.

- `exchange`

    Name of the exchange or `Exchange` object used.

- `routing_key`

Routing key used.

task_prerun

Dispatched before a task is executed.

Sender is the task object being executed.

Provides arguments:

- task_id

    Id of the task to be executed.

- task

    The task being executed.

- args

    The tasks positional arguments.

- kwargs

    The tasks keyword arguments.

task_postrun

Dispatched after a task has been executed.

Sender is the task object executed.

Provides arguments:

- task_id

    Id of the task to be executed.

- task

    The task being executed.

- args

    The tasks positional arguments.

- kwargs

    The tasks keyword arguments.

- retval

    The return value of the task.

- state

    Name of the resulting state.

task_retry

Dispatched when a task will be retried.

Sender is the task object.

Provides arguments:

- request

    The current task request.

- reason

    Reason for retry (usually an exception instance, but can always be coerced to `str`).

- einfo

    Detailed exception information, including traceback (a `billiard.einfo.ExceptionInfo` object).

task_success

Dispatched when a task succeeds.

Sender is the task object executed.

Provides arguments

- **result** Return value of the task.

task_failure

Dispatched when a task fails.

Sender is the task object executed.

Provides arguments:

- task_id

    Id of the task.

- exception

    Exception instance raised.

- args

    Positional arguments the task was called with.

- kwargs

    Keyword arguments the task was called with.

- traceback

    Stack trace object.

- einfo

    The `billiard.einfo.ExceptionInfo` instance.

task_received

Dispatched when a task is received from the broker and is ready for execution.

Sender is the consumer object.

Provides arguments:

- request

  This is a [*Request*](#) instance, and not `task.request`. When using the prefork pool this signal is dispatched in the parent process, so `task.request` isn't available and shouldn't be used. Use this object instead, as they share many of the same fields.

task_revoked

Dispatched when a task is revoked/terminated by the worker.

Sender is the task object revoked/terminated.

Provides arguments:

- request

  This is a [*Request*](#) instance, and not `task.request`. When using the prefork pool this signal is dispatched in the parent process, so `task.request` isn't available and shouldn't be used. Use this object instead, as they share many of the same fields.

- terminated

  Set to `True` if the task was terminated.

- signum

  Signal number used to terminate the task. If this is `None` and terminated is `True` then `TERM` should be assumed.

- expired

  Set to `True` if the task expired.

task_unknown

Dispatched when a worker receives a message for a task that's not registered.

Sender is the worker [*Consumer*](#).

Provides arguments:

- name

  Name of task not found in registry.

- id

  The task id found in the message.

- message

  Raw message object.

- exc

  The error that occurred.

---

task_rejected

Dispatched when a worker receives an unknown type of message to one of its task queues.

Sender is the worker *Consumer*.

Provides arguments:

- message

  Raw message object.

- exc

    The error that occurred (if any).

App Signals

import_modules

This signal is sent when a program (worker, beat, shell) etc, asks for modules in the *include* and *imports* settings to be imported.

Sender is the app instance.

Worker Signals

celeryd_after_setup

This signal is sent after the worker instance is set up, but before it calls run. This means that any queues from the *celery worker -Q* option is enabled, logging has been set up and so on.

It can be used to add custom queues that should always be consumed from, disregarding the *celery worker -Q* option. Here's an example that sets up a direct queue for each worker, these queues can then be used to route a task to any specific worker:

```python
from celery.signals import celeryd_after_setup

@celeryd_after_setup.connect
def setup_direct_queue(sender, instance, **kwargs):
    queue_name = '{0}.dq'.format(sender)  # sender is the nodename of the worker
    instance.app.amqp.queues.select_add(queue_name)
```

Provides arguments:

- sender

  Node name of the worker.

- instance

    This is the *celery.apps.worker.Worker* instance to be initialized. Note that only the *app* and *hostname* (nodename) attributes have been set so far, and the rest of __init__ hasn't been executed.

- conf

    The configuration of the current app.

celeryd_init

This is the first signal sent when **celery worker** starts up. The `sender` is the host name of the worker, so this signal can be used to setup worker specific configuration:

```python
from celery.signals import celeryd_init

@celeryd_init.connect(sender='worker12@example.com')
def configure_worker12(conf=None, **kwargs):
    conf.task_default_rate_limit = '10/m'
```

or to set up configuration for multiple workers you can omit specifying a sender when you connect:

```python
from celery.signals import celeryd_init

@celeryd_init.connect
def configure_workers(sender=None, conf=None, **kwargs):
    if sender in ('worker1@example.com', 'worker2@example.com'):
        conf.task_default_rate_limit = '10/m'
    if sender == 'worker3@example.com':
        conf.worker_prefetch_multiplier = 0
```

Provides arguments:

- `sender`

  Nodename of the worker.

- `instance`

    This is the *celery.apps.worker.Worker* instance to be initialized. Note that only the *app* and *hostname* (nodename) attributes have been set so far, and the rest of __init__ hasn't been executed.

- `conf`

    The configuration of the current app.

- `options`

    Options passed to the worker from command-line arguments (including defaults).

worker_init

Dispatched before the worker is started.

worker_ready

Dispatched when the worker is ready to accept work.

heartbeat_sent

Dispatched when Celery sends a worker heartbeat.

Sender is the *celery.worker.heartbeat.Heart* instance.

worker_shutting_down

Dispatched when the worker begins the shutdown process.

Provides arguments:

- sig

    The POSIX signal that was received.

- how

    The shutdown method, warm or cold.

- exitcode

    The exitcode that will be used when the main process exits.

worker_process_init

Dispatched in all pool child processes when they start.

Note that handlers attached to this signal mustn't be blocking for more than 4 seconds, or the process will be killed assuming it failed to start.

worker_process_shutdown

Dispatched in all pool child processes just before they exit.

Note: There's no guarantee that this signal will be dispatched, similarly to `finally` blocks it's impossible to guarantee that handlers will be called at shutdown, and if called it may be interrupted during.

Provides arguments:

- pid

    The pid of the child process that's about to shutdown.

- exitcode

    The exitcode that'll be used when the child process exits.

worker_shutdown

Dispatched when the worker is about to shut down.

Beat Signals

beat_init

Dispatched when **celery beat** starts (either standalone or embedded).

Sender is the *celery.beat.Service* instance.

beat_embedded_init

Dispatched in addition to the *beat_init* signal when **celery beat** is started as an embedded process.

Sender is the *celery.beat.Service* instance.

Eventlet Signals

eventlet_pool_started

Sent when the eventlet pool has been started.

Sender is the *celery.concurrency.eventlet.TaskPool* instance.

eventlet_pool_preshutdown

Sent when the worker shutdown, just before the eventlet pool is requested to wait for remaining workers.

Sender is the *celery.concurrency.eventlet.TaskPool* instance.

eventlet_pool_postshutdown

Sent when the pool has been joined and the worker is ready to shutdown.

Sender is the *celery.concurrency.eventlet.TaskPool* instance.

eventlet_pool_apply

Sent whenever a task is applied to the pool.

Sender is the *celery.concurrency.eventlet.TaskPool* instance.

Provides arguments:

- target

    The target function.

- args

    Positional arguments.

- kwargs

    Keyword arguments.

Logging Signals

setup_logging

Celery won't configure the loggers if this signal is connected, so you can use this to completely override the logging configuration with your own.

If you'd like to augment the logging configuration setup by Celery then you can use the *after_setup_logger* and *after_setup_task_logger* signals.

Provides arguments:

- `loglevel`

    The level of the logging object.

- `logfile`

    The name of the logfile.

- `format`

    The log format string.

- `colorize`

    Specify if log messages are colored or not.

after_setup_logger

Sent after the setup of every global logger (not task loggers). Used to augment logging configuration.

Provides arguments:

- `logger`

    The logger object.

- `loglevel`

    The level of the logging object.

- `logfile`

    The name of the logfile.

- `format`

    The log format string.

- `colorize`

    Specify if log messages are colored or not.

after_setup_task_logger

Sent after the setup of every single task logger. Used to augment logging configuration.

Provides arguments:

- `logger`

    The logger object.

- `loglevel`

    The level of the logging object.

- `logfile`

    The name of the logfile.

- `format`

    The log format string.

- colorize

    Specify if log messages are colored or not.

Command signals

user_preload_options

This signal is sent after any of the Celery command line programs are finished parsing the user preload options.

It can be used to add additional command-line arguments to the **celery** umbrella command:

```python
from celery import Celery
from celery import signals
from celery.bin.base import Option

app = Celery()
app.user_options['preload'].add(Option(
    '--monitoring', action='store_true',
    help='Enable our external monitoring utility, blahblah',
))

@signals.user_preload_options.connect
def handle_preload_options(options, **kwargs):
    if options['monitoring']:
        enable_monitoring()
```

Sender is the *Command* instance, and the value depends on the program that was called (e.g., for the umbrella command it'll be a *CeleryCommand*) object).

Provides arguments:

- app

    The app instance.

- options

    Mapping of the parsed user preload options (with default values).

Deprecated Signals

task_sent

This signal is deprecated, please use *after_task_publish* instead.

3.3.15 Testing with Celery

Tasks and unit tests

To test task behavior in unit tests the preferred method is mocking.

Eager mode

The eager mode enabled by the *task_always_eager* setting is by definition not suitable for unit tests.

When testing with eager mode you are only testing an emulation of what happens in a worker, and there are many discrepancies between the emulation and what happens in reality.

A Celery task is much like a web view, in that it should only define how to perform the action in the context of being called as a task.

This means optimally tasks only handle things like serialization, message headers, retries, and so on, with the actual logic implemented elsewhere.

Say we had a task like this:

```python
from .models import Product


@app.task(bind=True)
def send_order(self, product_pk, quantity, price):
    price = Decimal(price)  # json serializes this to string.

    # models are passed by id, not serialized.
    product = Product.objects.get(product_pk)

    try:
        product.order(quantity, price)
    except OperationalError as exc:
        raise self.retry(exc=exc)
```

Note: A task being bound means the first argument to the task will always be the task instance (self). which means you do get a self argument as the first argument and can use the Task class methods and attributes.

You could write unit tests for this task, using mocking like in this example:

```python
from pytest import raises

from celery.exceptions import Retry

# for python 2: use mock.patch from `pip install mock`.
from unittest.mock import patch

from proj.models import Product
from proj.tasks import send_order

class test_send_order:

    @patch('proj.tasks.Product.order')  # < patching Product in module above
    def test_success(self, product_order):
        product = Product.objects.create(
            name='Foo',
        )
        send_order(product.pk, 3, Decimal(30.3))
        product_order.assert_called_with(3, Decimal(30.3))

    @patch('proj.tasks.Product.order')
    @patch('proj.tasks.send_order.retry')
    def test_failure(self, send_order_retry, product_order):
        product = Product.objects.create(
            name='Foo',
        )
```

```
        # Set a side effect on the patched methods
        # so that they raise the errors we want.
        send_order_retry.side_effect = Retry()
        product_order.side_effect = OperationalError()

        with raises(Retry):
            send_order(product.pk, 3, Decimal(30.6))
```

Py.test

New in version 4.0.

Celery is also a pytest plugin that adds fixtures that you can use in your integration (or unit) test suites.

Marks

`celery` - Set test app configuration.

The `celery` mark enables you to override the configuration used for a single test case:

```python
@pytest.mark.celery(result_backend='redis://')
def test_something():
    ...
```

or for all the test cases in a class:

```python
@pytest.mark.celery(result_backend='redis://')
class test_something:

    def test_one(self):
        ...

    def test_two(self):
        ...
```

Fixtures

Function scope

`celery_app` - Celery app used for testing.

This fixture returns a Celery app you can use for testing.

Example:

```python
def test_create_task(celery_app, celery_worker):
    @celery_app.task
    def mul(x, y):
        return x * y

    assert mul.delay(4, 4).get(timeout=10) == 16
```

`celery_worker` - Embed live worker.

This fixture starts a Celery worker instance that you can use for integration tests. The worker will be started in a *separate thread* and will be shutdown as soon as the test returns.

Example:

```python
# Put this in your conftest.py
@pytest.fixture(scope='session')
def celery_config():
    return {
        'broker_url': 'amqp://',
        'result_backend': 'redis://'
    }


def test_add(celery_worker):
    mytask.delay()


# If you wish to override some setting in one test cases
# only - you can use the ``celery`` mark:
@pytest.mark.celery(result_backend='rpc')
def test_other(celery_worker):
    ...
```

Session scope

`celery_config` - Override to setup Celery test app configuration.

You can redefine this fixture to configure the test Celery app.

The config returned by your fixture will then be used to configure the `celery_app()`, and `celery_session_app()` fixtures.

Example:

```python
@pytest.fixture(scope='session')
def celery_config():
    return {
        'broker_url': 'amqp://',
        'result_backend': 'rpc',
    }
```

`celery_parameters` - Override to setup Celery test app parameters.

You can redefine this fixture to change the `__init__` parameters of test Celery app. In contrast to `celery_config()`, these are directly passed to when instantiating *Celery*.

The config returned by your fixture will then be used to configure the `celery_app()`, and `celery_session_app()` fixtures.

Example:

```python
@pytest.fixture(scope='session')
def celery_parameters():
    return {
        'task_cls':  my.package.MyCustomTaskClass,
        'strict_typing': False,
    }
```

`celery_worker_parameters` - Override to setup Celery worker parameters.

You can redefine this fixture to change the __init__ parameters of test Celery workers. These are directly passed to *WorkController* when it is instantiated.

The config returned by your fixture will then be used to configure the `celery_worker()`, and `celery_session_worker()` fixtures.

Example:

```python
@pytest.fixture(scope='session')
def celery_worker_parameters():
    return {
        'queues':  ('high-prio', 'low-prio'),
        'exclude_queues': ('celery'),
    }
```

`celery_enable_logging` - Override to enable logging in embedded workers.

This is a fixture you can override to enable logging in embedded workers.

Example:

```python
@pytest.fixture(scope='session')
def celery_enable_logging():
    return True
```

`celery_includes` - Add additional imports for embedded workers.

You can override fixture to include modules when an embedded worker starts.

You can have this return a list of module names to import, which can be task modules, modules registering signals, and so on.

Example:

```python
@pytest.fixture(scope='session')
def celery_includes():
    return [
        'proj.tests.tasks',
        'proj.tests.celery_signal_handlers',
    ]
```

`celery_worker_pool` - **Override the pool used for embedded workers.**

You can override fixture to configure the execution pool used for embedded workers.

Example:

```
@pytest.fixture(scope='session')
def celery_worker_pool():
    return 'prefork'
```

> **Warning:** You cannot use the gevent/eventlet pools, that is unless your whole test suite is running with the monkeypatches enabled.

`celery_session_worker` - **Embedded worker that lives throughout the session.**

This fixture starts a worker that lives throughout the testing session (it won't be started/stopped for every test).

Example:

```
# Add this to your conftest.py
@pytest.fixture(scope='session')
def celery_config():
    return {
        'broker_url': 'amqp://',
        'result_backend': 'rpc',
    }

# Do this in your tests.
def test_add_task(celery_session_worker):
    assert add.delay(2, 2) == 4
```

> **Warning:** It's probably a bad idea to mix session and ephemeral workers...

`celery_session_app` - **Celery app used for testing (session scope).**

This can be used by other session scoped fixtures when they need to refer to a Celery app instance.

`use_celery_app_trap` - **Raise exception on falling back to default app.**

This is a fixture you can override in your `conftest.py`, to enable the "app trap": if something tries to access the default or current_app, an exception is raised.

Example:

```
@pytest.fixture(scope='session')
def use_celery_app_trap():
    return True
```

If a test wants to access the default app, you would have to mark it using the `depends_on_current_app` fixture:

```python
@pytest.mark.usefixtures('depends_on_current_app')
def test_something():
    something()
```

3.3.16 Extensions and Bootsteps

Custom Message Consumers

You may want to embed custom Kombu consumers to manually process your messages.

For that purpose a special `ConsumerStep` bootstep class exists, where you only need to define the `get_consumers` method, that must return a list of `kombu.Consumer` objects to start whenever the connection is established:

```python
from celery import Celery
from celery import bootsteps
from kombu import Consumer, Exchange, Queue

my_queue = Queue('custom', Exchange('custom'), 'routing_key')

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


class MyConsumerStep(bootsteps.ConsumerStep):

    def get_consumers(self, channel):
        return [Consumer(channel,
```

```python
                    queues=[my_queue],
                    callbacks=[self.handle_message],
                    accept=['json'])]

    def handle_message(self, body, message):
        print('Received message: {0!r}'.format(body))
        message.ack()
app.steps['consumer'].add(MyConsumerStep)

def send_me_a_message(who, producer=None):
    with app.producer_or_acquire(producer) as producer:
        producer.publish(
            {'hello': who},
            serializer='json',
            exchange=my_queue.exchange,
            routing_key='routing_key',
            declare=[my_queue],
            retry=True,
        )

if __name__ == '__main__':
    send_me_a_message('world!')
```

Note: Kombu Consumers can take use of two different message callback dispatching mechanisms. The first one is the `callbacks` argument that accepts a list of callbacks with a `(body, message)` signature, the second one is the `on_message` argument that takes a single callback with a `(message,)` signature. The latter won't automatically decode and deserialize the payload.

```python
def get_consumers(self, channel):
    return [Consumer(channel, queues=[my_queue],
                     on_message=self.on_message)]


def on_message(self, message):
    payload = message.decode()
    print(
        'Received message: {0!r} {props!r} rawlen={s}'.format(
        payload, props=message.properties, s=len(message.body),
    ))
    message.ack()
```

Blueprints

Bootsteps is a technique to add functionality to the workers. A bootstep is a custom class that defines hooks to do custom actions at different stages in the worker. Every bootstep belongs to a blueprint, and the worker currently defines two blueprints: **Worker**, and **Consumer**

Figure A: Bootsteps in the Worker and Consumer blueprints. Starting from the bottom up the first step in the worker blueprint is the Timer, and the last step is to start the Consumer blueprint, that then establishes the broker connection and starts consuming messages.
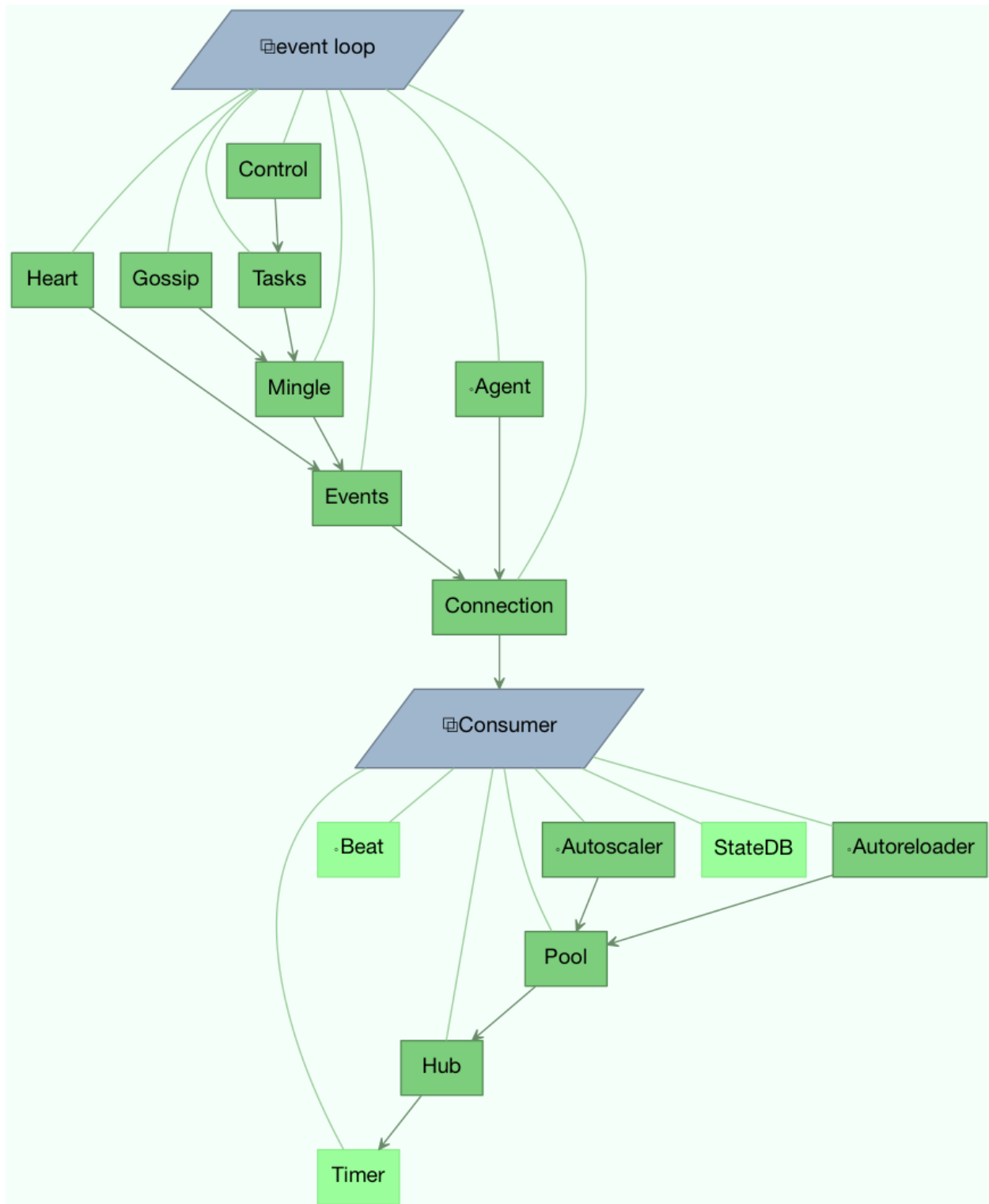
Worker

The Worker is the first blueprint to start, and with it starts major components like the event loop, processing pool, and the timer used for ETA tasks and other timed events.

When the worker is fully started it continues with the Consumer blueprint, that sets up how tasks are executed, connects to the broker and starts the message consumers.

The *WorkController* is the core worker implementation, and contains several methods and attributes that you can use in your bootstep.

Attributes

app
> The current app instance.

hostname
> The workers node name (e.g., *worker1@example.com*)

blueprint
> This is the worker *Blueprint*.

hub
> Event loop object (*Hub*). You can use this to register callbacks in the event loop.
>
> This is only supported by async I/O enabled transports (amqp, redis), in which case the *worker.use_eventloop* attribute should be set.
>
> Your worker bootstep must require the Hub bootstep to use this:

```
class WorkerStep(bootsteps.StartStopStep):
    requires = {'celery.worker.components:Hub'}
```

pool
> The current process/eventlet/gevent/thread pool. See *celery.concurrency.base.BasePool*.
>
> Your worker bootstep must require the Pool bootstep to use this:

```
class WorkerStep(bootsteps.StartStopStep):
    requires = {'celery.worker.components:Pool'}
```

timer
> *Timer* used to schedule functions.
>
> Your worker bootstep must require the Timer bootstep to use this:

```
class WorkerStep(bootsteps.StartStopStep):
    requires = {'celery.worker.components:Timer'}
```

statedb
> Database <celery.worker.state.Persistent>` to persist state between worker restarts.
>
> This is only defined if the statedb argument is enabled.
>
> Your worker bootstep must require the Statedb bootstep to use this:

```
class WorkerStep(bootsteps.StartStopStep):
    requires = {'celery.worker.components:Statedb'}
```

autoscaler

Autoscaler used to automatically grow and shrink the number of processes in the pool.

This is only defined if the autoscale argument is enabled.

Your worker bootstep must require the *Autoscaler* bootstep to use this:

```python
class WorkerStep(bootsteps.StartStopStep):
    requires = ('celery.worker.autoscaler:Autoscaler',)
```

autoreloader

Autoreloader used to automatically reload use code when the file-system changes.

This is only defined if the autoreload argument is enabled. Your worker bootstep must require the *Autoreloader* bootstep to use this;

```python
class WorkerStep(bootsteps.StartStopStep):
    requires = ('celery.worker.autoreloader:Autoreloader',)
```

Example worker bootstep

An example Worker bootstep could be:

```python
from celery import bootsteps

class ExampleWorkerStep(bootsteps.StartStopStep):
    requires = {'celery.worker.components:Pool'}

    def __init__(self, worker, **kwargs):
        print('Called when the WorkController instance is constructed')
        print('Arguments to WorkController: {0!r}'.format(kwargs))

    def create(self, worker):
        # this method can be used to delegate the action methods
        # to another object that implements ``start`` and ``stop``.
        return self

    def start(self, worker):
        print('Called when the worker is started.')

    def stop(self, worker):
        print('Called when the worker shuts down.')

    def terminate(self, worker):
        print('Called when the worker terminates')
```

Every method is passed the current WorkController instance as the first argument.

Another example could use the timer to wake up at regular intervals:

```python
from celery import bootsteps


class DeadlockDetection(bootsteps.StartStopStep):
    requires = {'celery.worker.components:Timer'}

    def __init__(self, worker, deadlock_timeout=3600):
```

<div align="right">(continues on next page)</div>

```python
        self.timeout = deadlock_timeout
        self.requests = []
        self.tref = None

    def start(self, worker):
        # run every 30 seconds.
        self.tref = worker.timer.call_repeatedly(
            30.0, self.detect, (worker,), priority=10,
        )

    def stop(self, worker):
        if self.tref:
            self.tref.cancel()
            self.tref = None

    def detect(self, worker):
        # update active requests
        for req in worker.active_requests:
            if req.time_start and time() - req.time_start > self.timeout:
                raise SystemExit()
```

Consumer

The Consumer blueprint establishes a connection to the broker, and is restarted every time this connection is lost. Consumer bootsteps include the worker heartbeat, the remote control command consumer, and importantly, the task consumer.

When you create consumer bootsteps you must take into account that it must be possible to restart your blueprint. An additional 'shutdown' method is defined for consumer bootsteps, this method is called when the worker is shutdown.

Attributes

app
> The current app instance.

controller
> The parent *WorkController* object that created this consumer.

hostname
> The workers node name (e.g., *worker1@example.com*)

blueprint
> This is the worker *Blueprint*.

hub
> Event loop object (*Hub*). You can use this to register callbacks in the event loop.
>
> This is only supported by async I/O enabled transports (amqp, redis), in which case the *worker.use_eventloop* attribute should be set.
>
> Your worker bootstep must require the Hub bootstep to use this:

```python
class WorkerStep(bootsteps.StartStopStep):
    requires = {'celery.worker.components:Hub'}
```

connection
> The current broker connection (`kombu.Connection`).
>
> A consumer bootstep must require the 'Connection' bootstep to use this:

```
class Step(bootsteps.StartStopStep):
    requires = {'celery.worker.consumer.connection:Connection'}
```

event_dispatcher
> A `app.events.Dispatcher` object that can be used to send events.
>
> A consumer bootstep must require the *Events* bootstep to use this.

```
class Step(bootsteps.StartStopStep):
    requires = {'celery.worker.consumer.events:Events'}
```

gossip
> Worker to worker broadcast communication (*Gossip*).
>
> A consumer bootstep must require the *Gossip* bootstep to use this.

```
class RatelimitStep(bootsteps.StartStopStep):
    """Rate limit tasks based on the number of workers in the
    cluster."""
    requires = {'celery.worker.consumer.gossip:Gossip'}

    def start(self, c):
        self.c = c
        self.c.gossip.on.node_join.add(self.on_cluster_size_change)
        self.c.gossip.on.node_leave.add(self.on_cluster_size_change)
        self.c.gossip.on.node_lost.add(self.on_node_lost)
        self.tasks = [
            self.app.tasks['proj.tasks.add']
            self.app.tasks['proj.tasks.mul']
        ]
        self.last_size = None

    def on_cluster_size_change(self, worker):
        cluster_size = len(list(self.c.gossip.state.alive_workers()))
        if cluster_size != self.last_size:
            for task in self.tasks:
                task.rate_limit = 1.0 / cluster_size
            self.c.reset_rate_limits()
            self.last_size = cluster_size

    def on_node_lost(self, worker):
        # may have processed heartbeat too late, so wake up soon
        # in order to see if the worker recovered.
        self.c.timer.call_after(10.0, self.on_cluster_size_change)
```

Callbacks

- `<set> gossip.on.node_join`

    Called whenever a new node joins the cluster, providing a *Worker* instance.

- `<set> gossip.on.node_leave`

    Called whenever a new node leaves the cluster (shuts down), providing a *Worker* instance.

- `<set> gossip.on.node_lost`

> Called whenever heartbeat was missed for a worker instance in the cluster (heartbeat not received or processed in time), providing a `Worker` instance.
>
> This doesn't necessarily mean the worker is actually offline, so use a time out mechanism if the default heartbeat timeout isn't sufficient.

pool

> The current process/eventlet/gevent/thread pool. See `celery.concurrency.base.BasePool`.

timer

> Timer `<celery.utils.timer2.Schedule` used to schedule functions.

heart

> Responsible for sending worker event heartbeats (`Heart`).
>
> Your consumer bootstep must require the *Heart* bootstep to use this:

```python
class Step(bootsteps.StartStopStep):
    requires = {'celery.worker.consumer.heart:Heart'}
```

task_consumer

> The `kombu.Consumer` object used to consume task messages.
>
> Your consumer bootstep must require the *Tasks* bootstep to use this:

```python
class Step(bootsteps.StartStopStep):
    requires = {'celery.worker.consumer.tasks:Tasks'}
```

strategies

> Every registered task type has an entry in this mapping, where the value is used to execute an incoming message of this task type (the task execution strategy). This mapping is generated by the Tasks bootstep when the consumer starts:

```python
for name, task in app.tasks.items():
    strategies[name] = task.start_strategy(app, consumer)
    task.__trace__ = celery.app.trace.build_tracer(
        name, task, loader, hostname
    )
```

> Your consumer bootstep must require the *Tasks* bootstep to use this:

```python
class Step(bootsteps.StartStopStep):
    requires = {'celery.worker.consumer.tasks:Tasks'}
```

task_buckets

> A `defaultdict` used to look-up the rate limit for a task by type. Entries in this dict may be None (for no limit) or a `TokenBucket` instance implementing `consume(tokens)` and `expected_time(tokens)`.
>
> TokenBucket implements the token bucket algorithm, but any algorithm may be used as long as it conforms to the same interface and defines the two methods above.

qos

> The `QoS` object can be used to change the task channels current prefetch_count value:

```python
# increment at next cycle
consumer.qos.increment_eventually(1)
# decrement at next cycle
consumer.qos.decrement_eventually(1)
consumer.qos.set(10)
```

Methods

`consumer.`**`reset_rate_limits`**`()`
> Updates the `task_buckets` mapping for all registered task types.

`consumer.`**`bucket_for_task`**`(`*type*, *Bucket=TokenBucket*`)`
> Creates rate limit bucket for a task using its `task.rate_limit` attribute.

`consumer.add_task_queue(name, exchange=None, exchange_type=None,`
`routing_key=None, **options):`
> Adds new queue to consume from. This will persist on connection restart.

`consumer.`**`cancel_task_queue`**`(`*name*`)`
> Stop consuming from queue by name. This will persist on connection restart.

`apply_eta_task``(`*request*`)`
> Schedule ETA task to execute based on the `request.eta` attribute. (*Request*)

Installing Bootsteps

`app.steps['worker']` and `app.steps['consumer']` can be modified to add new bootsteps:

```
>>> app = Celery()
>>> app.steps['worker'].add(MyWorkerStep)  # < add class, don't instantiate
>>> app.steps['consumer'].add(MyConsumerStep)

>>> app.steps['consumer'].update([StepA, StepB])

>>> app.steps['consumer']
{step:proj.StepB{()}, step:proj.MyConsumerStep{()}, step:proj.StepA{()}
```

The order of steps isn't important here as the order is decided by the resulting dependency graph (`Step.requires`).

To illustrate how you can install bootsteps and how they work, this is an example step that prints some useless debugging information. It can be added both as a worker and consumer bootstep:

```python
from celery import Celery
from celery import bootsteps


class InfoStep(bootsteps.Step):

    def __init__(self, parent, **kwargs):
        # here we can prepare the Worker/Consumer object
        # in any way we want, set attribute defaults, and so on.
        print('{0!r} is in init'.format(parent))

    def start(self, parent):
        # our step is started together with all other Worker/Consumer
        # bootsteps.
        print('{0!r} is starting'.format(parent))

    def stop(self, parent):
        # the Consumer calls stop every time the consumer is
        # restarted (i.e., connection is lost) and also at shutdown.
        # The Worker will call stop at shutdown only.
        print('{0!r} is stopping'.format(parent))
```

(continues on next page)

```python
    def shutdown(self, parent):
        # shutdown is called by the Consumer at shutdown, it's not
        # called by Worker.
        print('{0!r} is shutting down'.format(parent))

app = Celery(broker='amqp://')
app.steps['worker'].add(InfoStep)
app.steps['consumer'].add(InfoStep)
```

Starting the worker with this step installed will give us the following logs:

```
<Worker: w@example.com (initializing)> is in init
<Consumer: w@example.com (initializing)> is in init
[2013-05-29 16:18:20,544: WARNING/MainProcess]
    <Worker: w@example.com (running)> is starting
[2013-05-29 16:18:21,577: WARNING/MainProcess]
    <Consumer: w@example.com (running)> is starting
<Consumer: w@example.com (closing)> is stopping
<Worker: w@example.com (closing)> is stopping
<Consumer: w@example.com (terminating)> is shutting down
```

The `print` statements will be redirected to the logging subsystem after the worker has been initialized, so the "is starting" lines are time-stamped. You may notice that this does no longer happen at shutdown, this is because the `stop` and `shutdown` methods are called inside a *signal handler*, and it's not safe to use logging inside such a handler. Logging with the Python logging module isn't *reentrant*: meaning you cannot interrupt the function then call it again later. It's important that the `stop` and `shutdown` methods you write is also *reentrant*.

Starting the worker with *--loglevel=debug* will show us more information about the boot process:

```
[2013-05-29 16:18:20,509: DEBUG/MainProcess] | Worker: Preparing bootsteps.
[2013-05-29 16:18:20,511: DEBUG/MainProcess] | Worker: Building graph...
<celery.apps.worker.Worker object at 0x101ad8410> is in init
[2013-05-29 16:18:20,511: DEBUG/MainProcess] | Worker: New boot order:
    {Hub, Pool, Timer, StateDB, Autoscaler, InfoStep, Beat, Consumer}
[2013-05-29 16:18:20,514: DEBUG/MainProcess] | Consumer: Preparing bootsteps.
[2013-05-29 16:18:20,514: DEBUG/MainProcess] | Consumer: Building graph...
<celery.worker.consumer.Consumer object at 0x101c2d8d0> is in init
[2013-05-29 16:18:20,515: DEBUG/MainProcess] | Consumer: New boot order:
    {Connection, Mingle, Events, Gossip, InfoStep, Agent,
     Heart, Control, Tasks, event loop}
[2013-05-29 16:18:20,522: DEBUG/MainProcess] | Worker: Starting Hub
[2013-05-29 16:18:20,522: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:20,522: DEBUG/MainProcess] | Worker: Starting Pool
[2013-05-29 16:18:20,542: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:20,543: DEBUG/MainProcess] | Worker: Starting InfoStep
[2013-05-29 16:18:20,544: WARNING/MainProcess]
    <celery.apps.worker.Worker object at 0x101ad8410> is starting
[2013-05-29 16:18:20,544: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:20,544: DEBUG/MainProcess] | Worker: Starting Consumer
[2013-05-29 16:18:20,544: DEBUG/MainProcess] | Consumer: Starting Connection
[2013-05-29 16:18:20,559: INFO/MainProcess] Connected to amqp://guest@127.0.0.1:5672//
[2013-05-29 16:18:20,560: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:20,560: DEBUG/MainProcess] | Consumer: Starting Mingle
[2013-05-29 16:18:20,560: INFO/MainProcess] mingle: searching for neighbors
[2013-05-29 16:18:21,570: INFO/MainProcess] mingle: no one here
[2013-05-29 16:18:21,570: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:21,571: DEBUG/MainProcess] | Consumer: Starting Events
```

```
[2013-05-29 16:18:21,572: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:21,572: DEBUG/MainProcess] | Consumer: Starting Gossip
[2013-05-29 16:18:21,577: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:21,577: DEBUG/MainProcess] | Consumer: Starting InfoStep
[2013-05-29 16:18:21,577: WARNING/MainProcess]
    <celery.worker.consumer.Consumer object at 0x101c2d8d0> is starting
[2013-05-29 16:18:21,578: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:21,578: DEBUG/MainProcess] | Consumer: Starting Heart
[2013-05-29 16:18:21,579: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:21,579: DEBUG/MainProcess] | Consumer: Starting Control
[2013-05-29 16:18:21,583: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:21,583: DEBUG/MainProcess] | Consumer: Starting Tasks
[2013-05-29 16:18:21,606: DEBUG/MainProcess] basic.qos: prefetch_count->80
[2013-05-29 16:18:21,606: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:21,606: DEBUG/MainProcess] | Consumer: Starting event loop
[2013-05-29 16:18:21,608: WARNING/MainProcess] celery@example.com ready.
```

Command-line programs

Adding new command-line options

Command-specific options

You can add additional command-line options to the `worker`, `beat`, and `events` commands by modifying the *user_options* attribute of the application instance.

Celery commands uses the `argparse` module to parse command-line arguments, and so to add custom arguments you need to specify a callback that takes a `argparse.ArgumentParser` instance - and adds arguments. Please see the `argparse` documentation to read about the fields supported.

Example adding a custom option to the **celery worker** command:

```python
from celery import Celery

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

def add_worker_arguments(parser):
    parser.add_argument(
        '--enable-my-option', action='store_true', default=False,
        help='Enable custom option.',
    ),
app.user_options['worker'].add(add_worker_arguments)
```

All bootsteps will now receive this argument as a keyword argument to `Bootstep.__init__`:

```python
from celery import bootsteps

class MyBootstep(bootsteps.Step):

    def __init__(self, parent, enable_my_option=False, **options):
        super().__init__(parent, **options)
        if enable_my_option:
            party()

app.steps['worker'].add(MyBootstep)
```

Preload options

The **celery** umbrella command supports the concept of 'preload options'. These are special options passed to all sub-commands and parsed outside of the main parsing step.

The list of default preload options can be found in the API reference: *celery.bin.base*.

You can add new preload options too, for example to specify a configuration template:

```python
from celery import Celery
from celery import signals
from celery.bin import Option

app = Celery()

def add_preload_options(parser):
    parser.add_argument(
        '-Z', '--template', default='default',
        help='Configuration template to use.',
    )
app.user_options['preload'].add(add_preload_options)

@signals.user_preload_options.connect
def on_preload_parsed(options, **kwargs):
    use_template(options['template'])
```

Adding new `celery` sub-commands

New commands can be added to the **celery** umbrella command by using setuptools entry-points.

Entry-points is special meta-data that can be added to your packages setup.py program, and then after installation, read from the system using the pkg_resources module.

Celery recognizes celery.commands entry-points to install additional sub-commands, where the value of the entry-point must point to a valid subclass of *celery.bin.base.Command*. There's limited documentation, unfortunately, but you can find inspiration from the various commands in the celery.bin package.

This is how the Flower monitoring extension adds the **celery flower** command, by adding an entry-point in setup.py:

```python
setup(
    name='flower',
    entry_points={
        'celery.commands': [
            'flower = flower.command:FlowerCommand',
        ],
    }
)
```

The command definition is in two parts separated by the equal sign, where the first part is the name of the sub-command (flower), then the second part is the fully qualified symbol path to the class that implements the command:

```
flower.command:FlowerCommand
```

The module path and the name of the attribute should be separated by colon as above.

In the module flower/command.py, the command class is defined something like this:

```python
from celery.bin.base import Command


class FlowerCommand(Command):

    def add_arguments(self, parser):
        parser.add_argument(
            '--port', default=8888, type='int',
            help='Webserver port',
        ),
        parser.add_argument(
            '--debug', action='store_true',
        )

    def run(self, port=None, debug=False, **kwargs):
        print('Running our command')
```

Worker API

`Hub` - The workers async event loop

> **supported transports** amqp, redis

New in version 3.0.

The worker uses asynchronous I/O when the amqp or redis broker transports are used. The eventual goal is for all transports to use the event-loop, but that will take some time so other transports still use a threading-based solution.

hub.**add**(*fd*, *callback*, *flags*)

hub.**add_reader**(*fd*, *callback*, **args*)
> Add callback to be called when `fd` is readable.
>
> The callback will stay registered until explicitly removed using *hub.remove(fd)*, or the file descriptor is automatically discarded because it's no longer valid.
>
> Note that only one callback can be registered for any given file descriptor at a time, so calling `add` a second time will remove any callback that was previously registered for that file descriptor.
>
> A file descriptor is any file-like object that supports the `fileno` method, or it can be the file descriptor number (int).

hub.**add_writer**(*fd*, *callback*, **args*)
> Add callback to be called when `fd` is writable. See also notes for *hub.add_reader()* above.

hub.**remove**(*fd*)
> Remove all callbacks for file descriptor `fd` from the loop.

Timer - Scheduling events

timer.call_after(secs, callback, args=(), kwargs=(), priority=0)

timer.call_repeatedly(secs, callback, args=(), kwargs=(), priority=0)

timer.call_at(eta, callback, args=(), kwargs=(), priority=0)

3.3.17 Configuration and defaults

This document describes the configuration options available.

If you're using the default loader, you must create the `celeryconfig.py` module and make sure it's available on the Python path.

Example configuration file

This is an example configuration file to get you started. It should contain all you need to run a basic Celery set-up.

```
## Broker settings.
broker_url = 'amqp://guest:guest@localhost:5672//'

# List of modules to import when the Celery worker starts.
imports = ('myapp.tasks',)

## Using the database to store task state and results.
result_backend = 'db+sqlite:///results.db'

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

New lowercase settings

Version 4.0 introduced new lower case settings and setting organization.

The major difference between previous versions, apart from the lower case names, are the renaming of some prefixes, like `celery_beat_` to `beat_`, `celeryd_` to `worker_`, and most of the top level `celery_` settings have been moved into a new `task_` prefix.

Note: Celery will still be able to read old configuration files, so there's no rush in moving to the new settings format. Furthermore, we provide the `celery upgrade` command that should handle plenty of cases (including *Django*).

Configuration Directives

General settings

accept_content

Default: `{'json'}` (set, list, or tuple).

A white-list of content-types/serializers to allow.

If a message is received that's not in this list then the message will be discarded with an error.

By default only json is enabled but any content type can be added, including pickle and yaml; when this is the case make sure untrusted parties don't have access to your broker. See *Security* for more.

Example:

```
# using serializer name
accept_content = ['json']
```

(continues on next page)

```
# or the actual content-type (MIME)
accept_content = ['application/json']
```

result_accept_content

Default: `None` (can be set, list or tuple).

New in version 4.3.

A white-list of content-types/serializers to allow for the result backend.

If a message is received that's not in this list then the message will be discarded with an error.

By default it is the same serializer as `accept_content`. However, a different serializer for accepted content of the result backend can be specified. Usually this is needed if signed messaging is used and the result is stored unsigned in the result backend. See *Security* for more.

Example:

```
# using serializer name
result_accept_content = ['json']

# or the actual content-type (MIME)
result_accept_content = ['application/json']
```

Time and date settings

enable_utc

New in version 2.5.

Default: Enabled by default since version 3.0.

If enabled dates and times in messages will be converted to use the UTC timezone.

Note that workers running Celery versions below 2.5 will assume a local timezone for all messages, so only enable if all workers have been upgraded.

timezone

New in version 2.5.

Default: `"UTC"`.

Configure Celery to use a custom time zone. The timezone value can be any time zone supported by the pytz library.

If not set the UTC timezone is used. For backwards compatibility there's also a `enable_utc` setting, and when this is set to false the system local timezone is used instead.

Task settings

task_annotations

New in version 2.5.

Default: `None`.

This setting can be used to rewrite any task attribute from the configuration. The setting can be a dict, or a list of annotation objects that filter for tasks and return a map of attributes to change.

This will change the `rate_limit` attribute for the `tasks.add` task:

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

or change the same for all tasks:

```python
task_annotations = {'*': {'rate_limit': '10/s'}}
```

You can change methods too, for example the `on_failure` handler:

```python
def my_on_failure(self, exc, task_id, args, kwargs, einfo):
    print('Oh no! Task failed: {0!r}'.format(exc))

task_annotations = {'*': {'on_failure': my_on_failure}}
```

If you need more flexibility then you can use objects instead of a dict to choose the tasks to annotate:

```python
class MyAnnotate(object):

    def annotate(self, task):
        if task.name.startswith('tasks.'):
            return {'rate_limit': '10/s'}

task_annotations = (MyAnnotate(), {other,})
```

task_compression

Default: `None`

Default compression used for task messages. Can be `gzip`, `bzip2` (if available), or any custom compression schemes registered in the Kombu compression registry.

The default is to send uncompressed messages.

task_protocol

Default: 2 (since 4.0).

Set the default task message protocol version used to send tasks. Supports protocols: 1 and 2.

Protocol 2 is supported by 3.1.24 and 4.x+.

task_serializer

Default: `"json"` (since 4.0, earlier: pickle).

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

See also:

Serializers.

task_publish_retry

New in version 2.2.

Default: Enabled.

Decides if publishing task messages will be retried in the case of connection loss or other connection errors. See also *task_publish_retry_policy*.

task_publish_retry_policy

New in version 2.2.

Default: See *Message Sending Retry*.

Defines the default policy when retrying publishing a task message in the case of connection loss or other connection errors.

Task execution settings

task_always_eager

Default: Disabled.

If this is `True`, all tasks will be executed locally by blocking until the task returns. `apply_async()` and `Task.delay()` will return an `EagerResult` instance, that emulates the API and behavior of `AsyncResult`, except the result is already evaluated.

That is, tasks will be executed locally instead of being sent to the queue.

task_eager_propagates

Default: Disabled.

If this is `True`, eagerly executed tasks (applied by *task.apply()*, or when the *task_always_eager* setting is enabled), will propagate exceptions.

It's the same as always running `apply()` with `throw=True`.

task_remote_tracebacks

Default: Disabled.

If enabled task results will include the workers stack when re-raising task errors.

This requires the tblib library, that can be installed using **pip**:

```
$ pip install celery[tblib]
```

See *Bundles* for information on combining multiple extension requirements.

task_ignore_result

Default: Disabled.

Whether to store the task return values or not (tombstones). If you still want to store errors, just not successful return values, you can set `task_store_errors_even_if_ignored`.

task_store_errors_even_if_ignored

Default: Disabled.

If set, the worker stores all task errors in the result store even if `Task.ignore_result` is on.

task_track_started

Default: Disabled.

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

task_time_limit

Default: No time limit.

Task hard time limit in seconds. The worker processing the task will be killed and replaced with a new one when this is exceeded.

task_soft_time_limit

Default: No soft time limit.

Task soft time limit in seconds.

The *SoftTimeLimitExceeded* exception will be raised when this is exceeded. For example, the task can catch this to clean up before the hard time limit comes:

```python
from celery.exceptions import SoftTimeLimitExceeded

@app.task
def mytask():
    try:
        return do_work()
    except SoftTimeLimitExceeded:
        cleanup_in_a_hurry()
```

task_acks_late

Default: Disabled.

Late ack means the task messages will be acknowledged **after** the task has been executed, not *just before* (the default behavior).

See also:

FAQ: *Should I use retry or acks_late?*.

task_acks_on_failure_or_timeout

Default: Enabled

When enabled messages for all tasks will be acknowledged even if they fail or time out.

Configuring this setting only applies to tasks that are acknowledged **after** they have been executed and only if `task_acks_late` is enabled.

task_reject_on_worker_lost

Default: Disabled.

Even if `task_acks_late` is enabled, the worker will acknowledge tasks when the worker process executing them abruptly exits or is signaled (e.g., `KILL`/`INT`, etc).

Setting this to true allows the message to be re-queued instead, so that the task will execute again by the same worker, or another worker.

> **Warning:** Enabling this can cause message loops; make sure you know what you're doing.

task_default_rate_limit

Default: No rate limit.

The global default rate limit for tasks.

This value is used for tasks that doesn't have a custom rate limit

See also:

The setting:*worker_disable_rate_limits* setting can disable all rate limits.

Task result backend settings

result_backend

Default: No result backend enabled by default.

The backend used to store task results (tombstones). Can be one of the following:

- **rpc** Send results back as AMQP messages See *RPC backend settings*.

- **database** Use a relational database supported by SQLAlchemy. See *Database backend settings*.

- **redis** Use Redis to store the results. See *Redis backend settings*.

- **cache** Use Memcached to store the results. See *Cache backend settings*.

- **cassandra** Use Cassandra to store the results. See *Cassandra backend settings*.

- **elasticsearch** Use Elasticsearch to store the results. See *Elasticsearch backend settings*.

- **ironcache** Use IronCache to store the results. See *IronCache backend settings*.

- **couchbase** Use Couchbase to store the results. See *Couchbase backend settings*.

- **arangodb** Use ArangoDB to store the results. See *ArangoDB backend settings*.

- **couchdb** Use CouchDB to store the results. See *CouchDB backend settings*.

- **cosmosdbsql (experimental)** Use the CosmosDB PaaS to store the results. See *CosmosDB backend settings (experimental)*.

- **filesystem** Use a shared directory to store the results. See *File-system backend settings*.

- **consul** Use the Consul K/V store to store the results See *Consul K/V store backend settings*.

- **azureblockblob** Use the AzureBlockBlob PaaS store to store the results See *Azure Block Blob backend settings*.

- **s3** Use the S3 to store the results See *S3 backend settings*.

result_backend_transport_options

Default: `{}` (empty mapping).

A dict of additional options passed to the underlying transport.

See your transport user manual for supported options (if any).

Example setting the visibility timeout (supported by Redis and SQS transports):

```
result_backend_transport_options = {'visibility_timeout': 18000}  # 5 hours
```

result_serializer

Default: `json` since 4.0 (earlier: pickle).

Result serialization format.

See *Serializers* for information about supported serialization formats.

result_compression

Default: No compression.

Optional compression method used for task results. Supports the same options as the `task_serializer` setting.

result_extended

Default: `False`

Enables extended task result attributes (name, args, kwargs, worker, retries, queue, delivery_info) to be written to backend.

result_expires

Default: Expire after 1 day.

Time (in seconds, or a `timedelta` object) for when after stored task tombstones will be deleted.

A built-in periodic task will delete the results after this time (`celery.backend_cleanup`), assuming that `celery beat` is enabled. The task runs daily at 4am.

A value of `None` or 0 means results will never expire (depending on backend specifications).

Note: For the moment this only works with the AMQP, database, cache, Couchbase, and Redis backends.

When using the database backend, `celery beat` must be running for the results to be expired.

---

result_cache_max

Default: Disabled by default.

Enables client caching of results.

This can be useful for the old deprecated 'amqp' backend where the result is unavailable as soon as one result instance consumes it.

This is the total number of results to cache before older results are evicted. A value of 0 or None means no limit, and a value of `-1` will disable the cache.

Disabled by default.

result_chord_join_timeout

Default: 3.0.

The timeout in seconds (int/float) when joining a group's results within a chord.

Database backend settings

Database URL Examples

To use the database backend you have to configure the *result_backend* setting with a connection URL and the `db+` prefix:

```
result_backend = 'db+scheme://user:password@host:port/dbname'
```

Examples:

```
# sqlite (filename)
result_backend = 'db+sqlite:///results.sqlite'

# mysql
result_backend = 'db+mysql://scott:tiger@localhost/foo'

# postgresql
result_backend = 'db+postgresql://scott:tiger@localhost/mydatabase'

# oracle
result_backend = 'db+oracle://scott:tiger@127.0.0.1:1521/sidname'
```

Please see Supported Databases for a table of supported databases, and Connection String for more information about connection strings (this is the part of the URI that comes after the db+ prefix).

database_engine_options

Default: {} (empty mapping).

To specify additional SQLAlchemy database engine options you can use the sqlalchmey_engine_options setting:

```
# echo enables verbose logging from SQLAlchemy.
app.conf.database_engine_options = {'echo': True}
```

database_short_lived_sessions

Default: Disabled by default.

Short lived sessions are disabled by default. If enabled they can drastically reduce performance, especially on systems processing lots of tasks. This option is useful on low-traffic workers that experience errors as a result of cached database connections going stale through inactivity. For example, intermittent errors like *(OperationalError) (2006, 'MySQL server has gone away')* can be fixed by enabling short lived sessions. This option only affects the database backend.

database_table_schemas

Default: {} (empty mapping).

When SQLAlchemy is configured as the result backend, Celery automatically creates two tables to store result metadata for tasks. This setting allows you to customize the schema of the tables:

```
# use custom schema for the database result backend.
database_table_schemas = {
    'task': 'celery',
    'group': 'celery',
}
```

database_table_names

Default: `{}` (empty mapping).

When SQLAlchemy is configured as the result backend, Celery automatically creates two tables to store result meta-data for tasks. This setting allows you to customize the table names:

```
# use custom table names for the database result backend.
database_table_names = {
    'task': 'myapp_taskmeta',
    'group': 'myapp_groupmeta',
}
```

RPC backend settings

result_persistent

Default: Disabled by default (transient messages).

If set to `True`, result messages will be persistent. This means the messages won't be lost after a broker restart.

Example configuration

```
result_backend = 'rpc://'
result_persistent = False
```

Please note: using this backend could trigger the raise of `celery.backends.rpc.BacklogLimitExceeded` if the task tombstone is too *old*.

E.g.

```
for i in range(10000):
    r = debug_task.delay()

print(r.state)  # this would raise celery.backends.rpc.BacklogLimitExceeded
```

Cache backend settings

Note: The cache backend supports the pylibmc and python-memcached libraries. The latter is used only if pylibmc isn't installed.

Using a single Memcached server:

```
result_backend = 'cache+memcached://127.0.0.1:11211/'
```

Using multiple Memcached servers:

```
result_backend = """
    cache+memcached://172.19.26.240:11211;172.19.26.242:11211/
""".strip()
```

The "memory" backend stores the cache in memory only:

```
result_backend = 'cache'
cache_backend = 'memory'
```

cache_backend_options

Default: `{}` (empty mapping).

You can set pylibmc options using the *cache_backend_options* setting:

```
cache_backend_options = {
    'binary': True,
    'behaviors': {'tcp_nodelay': True},
}
```

cache_backend

This setting is no longer used as it's now possible to specify the cache backend directly in the *result_backend* setting.

Redis backend settings

Configuring the backend URL

Note: The Redis backend requires the redis library.

To install this package use **pip**:

```
$ pip install celery[redis]
```

See *Bundles* for information on combining multiple extension requirements.

This backend requires the *result_backend* setting to be set to a Redis or Redis over TLS URL:

```
result_backend = 'redis://:password@host:port/db'
```

For example:

```
result_backend = 'redis://localhost/0'
```

is the same as:

```
result_backend = 'redis://'
```

Use the `rediss://` protocol to connect to redis over TLS:

```
result_backend = 'rediss://:password@host:port/db?ssl_cert_reqs=required'
```

Note that the `ssl_cert_reqs` string should be one of `required`, `optional`, or `none` (though, for backwards compatibility, the string may also be one of `CERT_REQUIRED`, `CERT_OPTIONAL`, `CERT_NONE`).

If a Unix socket connection should be used, the URL needs to be in the format::

```
result_backend = 'socket:///path/to/redis.sock'
```

The fields of the URL are defined as follows:

1. `password`

    Password used to connect to the database.

2. `host`

    Host name or IP address of the Redis server (e.g., *localhost*).

3. `port`

    Port to the Redis server. Default is 6379.

4. `db`

    Database number to use. Default is 0. The db can include an optional leading slash.

When using a TLS connection (protocol is `rediss://`), you may pass in all values in *broker_use_ssl* as query parameters. Paths to certificates must be URL encoded, and `ssl_cert_reqs` is required. Example:

```
result_backend = 'rediss://:password@host:port/db?\
    ssl_cert_reqs=required\
    &ssl_ca_certs=%2Fvar%2Fssl%2Fmyca.pem\                    # /var/ssl/myca.pem
    &ssl_certfile=%2Fvar%2Fssl%2Fredis-server-cert.pem\      # /var/ssl/redis-server-
→cert.pem
    &ssl_keyfile=%2Fvar%2Fssl%2Fprivate%2Fworker-key.
→pem'    # /var/ssl/private/worker-key.pem
```

Note that the `ssl_cert_reqs` string should be one of `required`, `optional`, or `none` (though, for backwards compatibility, the string may also be one of `CERT_REQUIRED`, `CERT_OPTIONAL`, `CERT_NONE`).

redis_backend_use_ssl

Default: Disabled.

The Redis backend supports SSL. This value must be set in the form of a dictionary. The valid key-value pairs are the same as the ones mentioned in the `redis` sub-section under *broker_use_ssl*.

redis_max_connections

Default: No limit.

Maximum number of connections available in the Redis connection pool used for sending and retrieving results.

> **Warning:** Redis will raise a *ConnectionError* if the number of concurrent connections exceeds the maximum.

redis_socket_connect_timeout

New in version 4.0.1.

Default: `None`

Socket timeout for connections to Redis from the result backend in seconds (int/float)

redis_socket_timeout

Default: 120.0 seconds.

Socket timeout for reading/writing operations to the Redis server in seconds (int/float), used by the redis result backend.

redis_retry_on_timeout

Default: `False`

To retry reading/writing operations on TimeoutError to the Redis server, used by the redis result backend.

socket_keepalive

Default: `False`

Socket TCP keepalive to keep connections healthy to the Redis server, used by the redis result backend.

Cassandra backend settings

Note: This Cassandra backend driver requires cassandra-driver.

To install, use **pip**:

```
$ pip install celery[cassandra]
```

See *Bundles* for information on combining multiple extension requirements.

This backend requires the following configuration directives to be set.

cassandra_servers

Default: `[]` (empty list).

List of `host` Cassandra servers. For example:

```
cassandra_servers = ['localhost']
```

cassandra_port

Default: 9042.

Port to contact the Cassandra servers on.

cassandra_keyspace

Default: None.

The key-space in which to store the results. For example:

```
cassandra_keyspace = 'tasks_keyspace'
```

cassandra_table

Default: None.

The table (column family) in which to store the results. For example:

```
cassandra_table = 'tasks'
```

cassandra_read_consistency

Default: None.

The read consistency used. Values can be `ONE`, `TWO`, `THREE`, `QUORUM`, `ALL`, `LOCAL_QUORUM`, `EACH_QUORUM`, `LOCAL_ONE`.

cassandra_write_consistency

Default: None.

The write consistency used. Values can be `ONE`, `TWO`, `THREE`, `QUORUM`, `ALL`, `LOCAL_QUORUM`, `EACH_QUORUM`, `LOCAL_ONE`.

cassandra_entry_ttl

Default: None.

Time-to-live for status entries. They will expire and be removed after that many seconds after adding. A value of `None` (default) means they will never expire.

cassandra_auth_provider

Default: `None`.

AuthProvider class within `cassandra.auth` module to use. Values can be `PlainTextAuthProvider` or `SaslAuthProvider`.

cassandra_auth_kwargs

Default: `{}` (empty mapping).

Named arguments to pass into the authentication provider. For example:

```
cassandra_auth_kwargs = {
    username: 'cassandra',
    password: 'cassandra'
}
```

cassandra_options

Default: {} (empty mapping).

Named arguments to pass into the cassandra.cluster class.

```
cassandra_options = {
    'cql_version': '3.2.1'
    'protocol_version': 3
}
```

Example configuration

```
cassandra_servers = ['localhost']
cassandra_keyspace = 'celery'
cassandra_table = 'tasks'
cassandra_read_consistency = 'ONE'
cassandra_write_consistency = 'ONE'
cassandra_entry_ttl = 86400
```

S3 backend settings

Note: This s3 backend driver requires s3.

To install, use **s3**:

```
$ pip install celery[s3]
```

See *Bundles* for information on combining multiple extension requirements.

This backend requires the following configuration directives to be set.

s3_access_key_id

Default: None.

The s3 access key id. For example:

```
s3_access_key_id = 'acces_key_id'
```

s3_secret_access_key

Default: None.

The s3 secret access key. For example:

```
s3_secret_access_key = 'acces_secret_access_key'
```

s3_bucket

Default: None.

The s3 bucket name. For example:

```
s3_bucket = 'bucket_name'
```

s3_base_path

Default: None.

A base path in the s3 bucket to use to store result keys. For example:

```
s3_base_path = '/prefix'
```

s3_endpoint_url

Default: None.

A custom s3 endpoint url. Use it to connect to a custom self-hosted s3 compatible backend (Ceph, Scality...). For example:

```
s3_endpoint_url = 'https://.s3.custom.url'
```

s3_region

Default: None.

The s3 aws region. For example:

```
s3_region = 'us-east-1'
```

Example configuration

```
s3_access_key_id = 's3-access-key-id'
s3_secret_access_key = 's3-secret-access-key'
s3_bucket = 'mybucket'
s3_base_path = '/celery_result_backend'
s3_endpoint_url = 'https://endpoint_url'
```

Azure Block Blob backend settings

To use [AzureBlockBlob](#) as the result backend you simply need to configure the *result_backend* setting with the correct URL.

The required URL format is `azureblockblob://` followed by the storage connection string. You can find the storage connection string in the `Access Keys` pane of your storage account resource in the Azure Portal.

Example configuration

```
result_backend =␣
↪'azureblockblob://DefaultEndpointsProtocol=https;AccountName=somename;AccountKey=Lou...bzg==;Endpoi
```

azureblockblob_container_name

Default: celery.

The name for the storage container in which to store the results.

azureblockblob_retry_initial_backoff_sec

Default: 2.

The initial backoff interval, in seconds, for the first retry. Subsequent retries are attempted with an exponential strategy.

azureblockblob_retry_increment_base

Default: 2.

azureblockblob_retry_max_attempts

Default: 3.

The maximum number of retry attempts.

Elasticsearch backend settings

To use [Elasticsearch](#) as the result backend you simply need to configure the *result_backend* setting with the correct URL.

Example configuration

```
result_backend = 'elasticsearch://example.com:9200/index_name/doc_type'
```

elasticsearch_retry_on_timeout

Default: `False`

Should timeout trigger a retry on different node?

elasticsearch_max_retries

Default: 3.

Maximum number of retries before an exception is propagated.

elasticsearch_timeout

Default: 10.0 seconds.

Global timeout,used by the elasticsearch result backend.

Riak backend settings

Note: The Riak backend requires the riak library.

To install the this package use **pip**:

```
$ pip install celery[riak]
```

See *Bundles* for information on combining multiple extension requirements.

This backend requires the `result_backend` setting to be set to a Riak URL:

```
result_backend = 'riak://host:port/bucket'
```

For example:

```
result_backend = 'riak://localhost/celery'
```

is the same as:

```
result_backend = 'riak://'
```

The fields of the URL are defined as follows:

1. `host`

   Host name or IP address of the Riak server (e.g., *'localhost'*).

2. `port`

   Port to the Riak server using the protobuf protocol. Default is 8087.

3. `bucket`

   Bucket name to use. Default is *celery*. The bucket needs to be a string with ASCII characters only.

Alternatively, this backend can be configured with the following configuration directives.

riak_backend_settings

Default: `{}` (empty mapping).

This is a dict supporting the following keys:

- `host`

    The host name of the Riak server. Defaults to `"localhost"`.

- `port`

    The port the Riak server is listening to. Defaults to 8087.

- `bucket`

    The bucket name to connect to. Defaults to "celery".

- `protocol`

    The protocol to use to connect to the Riak server. This isn't configurable via *result_backend*

AWS DynamoDB backend settings

Note: The Dynamodb backend requires the boto3 library.

To install this package use **pip**:

```
$ pip install celery[dynamodb]
```

See *Bundles* for information on combining multiple extension requirements.

This backend requires the *result_backend* setting to be set to a DynamoDB URL:

```
result_backend =
→'dynamodb://aws_access_key_id:aws_secret_access_key@region:port/table?read=n&write=m'
```

For example, specifying the AWS region and the table name:

```
result_backend = 'dynamodb://@us-east-1/celery_results
```

or retrieving AWS configuration parameters from the environment, using the default table name (`celery`) and specifying read and write provisioned throughput:

```
result_backend = 'dynamodb://@/?read=5&write=5'
```

or using the downloadable version of DynamoDB locally:

```
result_backend = 'dynamodb://@localhost:8000'
```

or using downloadable version or other service with conforming API deployed on any host:

```
result_backend = 'dynamodb://@us-east-1'
dynamodb_endpoint_url = 'http://192.168.0.40:8000'
```

The fields of the DynamoDB URL in `result_backend` are defined as follows:

1. `aws_access_key_id` & `aws_secret_access_key`

The credentials for accessing AWS API resources. These can also be resolved by the boto3 library from various sources, as described here.

2. `region`

   The AWS region, e.g. `us-east-1` or `localhost` for the Downloadable Version. See the boto3 library documentation for definition options.

3. `port`

   The listening port of the local DynamoDB instance, if you are using the downloadable version. If you have not specified the `region` parameter as `localhost`, setting this parameter has **no effect**.

4. `table`

   Table name to use. Default is `celery`. See the DynamoDB Naming Rules for information on the allowed characters and length.

5. `read & write`

   The Read & Write Capacity Units for the created DynamoDB table. Default is `1` for both read and write. More details can be found in the Provisioned Throughput documentation.

6. `ttl_seconds`

   Time-to-live (in seconds) for results before they expire. The default is to not expire results, while also leaving the DynamoDB table's Time to Live settings untouched. If `ttl_seconds` is set to a positive value, results will expire after the specified number of seconds. Setting `ttl_seconds` to a negative value means to not expire results, and also to actively disable the DynamoDB table's Time to Live setting. Note that trying to change a table's Time to Live setting multiple times in quick succession will cause a throttling error. More details can be found in the DynamoDB TTL documentation

IronCache backend settings

Note: The IronCache backend requires the iron_celery library:

To install this package use **pip**:

```
$ pip install iron_celery
```

IronCache is configured via the URL provided in `result_backend`, for example:

```
result_backend = 'ironcache://project_id:token@'
```

Or to change the cache name:

```
ironcache:://project_id:token@/awesomecache
```

For more information, see: https://github.com/iron-io/iron_celery

Couchbase backend settings

Note: The Couchbase backend requires the couchbase library.

To install this package use **pip**:

```
$ pip install celery[couchbase]
```

See *Bundles* for instructions how to combine multiple extension requirements.

This backend can be configured via the *result_backend* set to a Couchbase URL:

```
result_backend = 'couchbase://username:password@host:port/bucket'
```

couchbase_backend_settings

Default: `{}` (empty mapping).

This is a dict supporting the following keys:

- `host`

    Host name of the Couchbase server. Defaults to `localhost`.

- `port`

    The port the Couchbase server is listening to. Defaults to `8091`.

- `bucket`

    The default bucket the Couchbase server is writing to. Defaults to `default`.

- `username`

    User name to authenticate to the Couchbase server as (optional).

- `password`

    Password to authenticate to the Couchbase server (optional).

ArangoDB backend settings

Note: The ArangoDB backend requires the pyArango library.

To install this package use **pip**:

```
$ pip install celery[arangodb]
```

See *Bundles* for instructions how to combine multiple extension requirements.

This backend can be configured via the *result_backend* set to a ArangoDB URL:

```
result_backend = 'arangodb://username:password@host:port/database/collection'
```

arangodb_backend_settings

Default: `{}` (empty mapping).

This is a dict supporting the following keys:

- `host`

Host name of the ArangoDB server. Defaults to `localhost`.

- `port`

  The port the ArangoDB server is listening to. Defaults to `8529`.

- `database`

  The default database in the ArangoDB server is writing to. Defaults to `celery`.

- `collection`

  The default collection in the ArangoDB servers database is writing to. Defaults to `celery`.

- `username`

  User name to authenticate to the ArangoDB server as (optional).

- `password`

  Password to authenticate to the ArangoDB server (optional).

CosmosDB backend settings (experimental)

To use CosmosDB as the result backend, you simply need to configure the *result_backend* setting with the correct URL.

Example configuration

```
result_backend =
→'cosmosdbsql://:{InsertAccountPrimaryKeyHere}@{InsertAccountNameHere}.documents.azure.com'
```

`cosmosdbsql_database_name`

Default: celerydb.

The name for the database in which to store the results.

`cosmosdbsql_collection_name`

Default: celerycol.

The name of the collection in which to store the results.

`cosmosdbsql_consistency_level`

Default: Session.

Represents the consistency levels supported for Azure Cosmos DB client operations.

Consistency levels by order of strength are: Strong, BoundedStaleness, Session, ConsistentPrefix and Eventual.

cosmosdbsql_max_retry_attempts

Default: 9.

Maximum number of retries to be performed for a request.

cosmosdbsql_max_retry_wait_time

Default: 30.

Maximum wait time in seconds to wait for a request while the retries are happening.

CouchDB backend settings

Note: The CouchDB backend requires the pycouchdb library:

To install this Couchbase package use **pip**:

```
$ pip install celery[couchdb]
```

See *Bundles* for information on combining multiple extension requirements.

This backend can be configured via the `result_backend` set to a CouchDB URL:

```
result_backend = 'couchdb://username:password@host:port/container'
```

The URL is formed out of the following parts:

- username

    User name to authenticate to the CouchDB server as (optional).

- password

    Password to authenticate to the CouchDB server (optional).

- host

    Host name of the CouchDB server. Defaults to `localhost`.

- port

    The port the CouchDB server is listening to. Defaults to `8091`.

- container

    The default container the CouchDB server is writing to. Defaults to `default`.

File-system backend settings

This backend can be configured using a file URL, for example:

```
CELERY_RESULT_BACKEND = 'file:///var/celery/results'
```

The configured directory needs to be shared and writable by all servers using the backend.

If you're trying Celery on a single system you can simply use the backend without any further configuration. For larger clusters you could use NFS, GlusterFS, CIFS, HDFS (using FUSE), or any other file-system.

Consul K/V store backend settings

The Consul backend can be configured using a URL, for example:

> CELERY_RESULT_BACKEND = 'consul://localhost:8500/'

The backend will storage results in the K/V store of Consul as individual keys.

The backend supports auto expire of results using TTLs in Consul.

Message Routing

task_queues

Default: `None` (queue taken from default queue settings).

Most users will not want to specify this setting and should rather use the *automatic routing facilities*.

If you really want to configure advanced routing, this setting should be a list of `kombu.Queue` objects the worker will consume from.

Note that workers can be overridden this setting via the `-Q` option, or individual queues from this list (by name) can be excluded using the `-X` option.

Also see *Basics* for more information.

The default is a queue/exchange/binding key of `celery`, with exchange type `direct`.

See also *task_routes*

task_routes

Default: `None`.

A list of routers, or a single router used to route tasks to queues. When deciding the final destination of a task the routers are consulted in order.

A router can be specified as either:

- A function with the signature `(name, args, kwargs, options, task=None, **kwargs)`
- A string providing the path to a router function.
- **A dict containing router specification:** Will be converted to a `celery.routes.MapRoute` instance.
- **A list of `(pattern, route)` tuples:** Will be converted to a `celery.routes.MapRoute` instance.

Examples:

```
task_routes = {
    'celery.ping': 'default',
    'mytasks.add': 'cpu-bound',
    'feed.tasks.*': 'feeds',                        # <-- glob pattern
    re.compile(r'(image|video)\.tasks\..*'): 'media',  # <-- regex
```

```
    'video.encode': {
        'queue': 'video',
        'exchange': 'media',
        'routing_key': 'media.video.encode',
    },
}

task_routes = ('myapp.tasks.route_task', {'celery.ping': 'default'})
```

Where `myapp.tasks.route_task` could be:

```python
def route_task(self, name, args, kwargs, options, task=None, **kw):
        if task == 'celery.ping':
            return {'queue': 'default'}
```

route_task may return a string or a dict. A string then means it's a queue name in *task_queues*, a dict means it's a custom route.

When sending tasks, the routers are consulted in order. The first router that doesn't return `None` is the route to use. The message options is then merged with the found route settings, where the task's settings have priority.

Example if `apply_async()` has these arguments:

```python
Task.apply_async(immediate=False, exchange='video',
                 routing_key='video.compress')
```

and a router returns:

```python
{'immediate': True, 'exchange': 'urgent'}
```

the final message options will be:

```python
immediate=False, exchange='video', routing_key='video.compress'
```

(and any default message options defined in the `Task` class)

Values defined in *task_routes* have precedence over values defined in *task_queues* when merging the two.

With the follow settings:

```python
task_queues = {
    'cpubound': {
        'exchange': 'cpubound',
        'routing_key': 'cpubound',
    },
}

task_routes = {
    'tasks.add': {
        'queue': 'cpubound',
        'routing_key': 'tasks.add',
        'serializer': 'json',
    },
}
```

The final routing options for `tasks.add` will become:

```
{'exchange': 'cpubound',
 'routing_key': 'tasks.add',
 'serializer': 'json'}
```

See *Routers* for more examples.

task_queue_ha_policy

> **brokers** RabbitMQ

Default: `None`.

This will set the default HA policy for a queue, and the value can either be a string (usually `all`):

```
task_queue_ha_policy = 'all'
```

Using 'all' will replicate the queue to all current nodes, Or you can give it a list of nodes to replicate to:

```
task_queue_ha_policy = ['rabbit@host1', 'rabbit@host2']
```

Using a list will implicitly set `x-ha-policy` to 'nodes' and `x-ha-policy-params` to the given list of nodes.

See http://www.rabbitmq.com/ha.html for more information.

task_queue_max_priority

> **brokers** RabbitMQ

Default: `None`.

See *RabbitMQ Message Priorities*.

task_default_priority

> **brokers** RabbitMQ, Redis

Default: `None`.

See *RabbitMQ Message Priorities*.

task_inherit_parent_priority

> **brokers** RabbitMQ

Default: `False`.

If enabled, child tasks will inherit priority of the parent task.

```
# The last task in chain will also have priority set to 5.
chain = celery.chain(add.s(2) | add.s(2).set(priority=5) | add.s(3))
```

Priority inheritance also works when calling child tasks from a parent task with *delay* or *apply_async*.

See *RabbitMQ Message Priorities*.

---

worker_direct

Default: Disabled.

This option enables so that every worker has a dedicated queue, so that tasks can be routed to specific workers.

The queue name for each worker is automatically generated based on the worker hostname and a `.dq` suffix, using the `C.dq` exchange.

For example the queue name for the worker with node name `w1@example.com` becomes:

```
w1@example.com.dq
```

Then you can route the task to the task by specifying the hostname as the routing key and the `C.dq` exchange:

```
task_routes = {
    'tasks.add': {'exchange': 'C.dq', 'routing_key': 'w1@example.com'}
}
```

task_create_missing_queues

Default: Enabled.

If enabled (default), any queues specified that aren't defined in *task_queues* will be automatically created. See *Automatic routing*.

task_default_queue

Default: `"celery"`.

The name of the default queue used by *.apply_async* if the message has no route or no custom queue has been specified.

This queue must be listed in *task_queues*. If *task_queues* isn't specified then it's automatically created containing one queue entry, where this name is used as the name of that queue.

See also:

Changing the name of the default queue

task_default_exchange

Default: Uses the value set for *task_default_queue*.

Name of the default exchange to use when no custom exchange is specified for a key in the *task_queues* setting.

task_default_exchange_type

Default: `"direct"`.

Default exchange type used when no custom exchange type is specified for a key in the *task_queues* setting.

task_default_routing_key

Default: Uses the value set for *task_default_queue*.

The default routing key used when no custom routing key is specified for a key in the *task_queues* setting.

task_default_delivery_mode

Default: `"persistent"`.

Can be *transient* (messages not written to disk) or *persistent* (written to disk).

Broker Settings

broker_url

Default: `"amqp://"`

Default broker URL. This must be a URL in the form of:

```
transport://userid:password@hostname:port/virtual_host
```

Only the scheme part (`transport://`) is required, the rest is optional, and defaults to the specific transports default values.

The transport part is the broker implementation to use, and the default is `amqp`, (uses `librabbitmq` if installed or falls back to `pyamqp`). There are also other choices available, including; `redis://`, `sqs://`, and `qpid://`.

The scheme can also be a fully qualified path to your own transport implementation:

```
broker_url = 'proj.transports.MyTransport://localhost'
```

More than one broker URL, of the same transport, can also be specified. The broker URLs can be passed in as a single string that's semicolon delimited:

```
broker_url =␣
→'transport://userid:password@hostname:port//;transport://userid:password@hostname:port//'
```

Or as a list:

```
broker_url = [
    'transport://userid:password@localhost:port//',
    'transport://userid:password@hostname:port//'
]
```

The brokers will then be used in the *broker_failover_strategy*.

See URLs in the Kombu documentation for more information.

broker_read_url / **broker_write_url**

Default: Taken from *broker_url*.

These settings can be configured, instead of *broker_url* to specify different connection parameters for broker connections used for consuming and producing.

Example:

```
broker_read_url = 'amqp://user:pass@broker.example.com:56721'
broker_write_url = 'amqp://user:pass@broker.example.com:56722'
```

Both options can also be specified as a list for failover alternates, see *broker_url* for more information.

broker_failover_strategy

Default: `"round-robin"`.

Default failover strategy for the broker Connection object. If supplied, may map to a key in 'kombu.connection.failover_strategies', or be a reference to any method that yields a single item from a supplied list.

Example:

```
# Random failover strategy
def random_failover_strategy(servers):
    it = list(servers)  # don't modify callers list
    shuffle = random.shuffle
    for _ in repeat(None):
        shuffle(it)
        yield it[0]

broker_failover_strategy = random_failover_strategy
```

broker_heartbeat

> **transports supported** `pyamqp`

Default: `120.0` (negotiated by server).

Note: This value is only used by the worker, clients do not use a heartbeat at the moment.

It's not always possible to detect connection loss in a timely manner using TCP/IP alone, so AMQP defines something called heartbeats that's is used both by the client and the broker to detect if a connection was closed.

If the heartbeat value is 10 seconds, then the heartbeat will be monitored at the interval specified by the *broker_heartbeat_checkrate* setting (by default this is set to double the rate of the heartbeat value, so for the 10 seconds, the heartbeat is checked every 5 seconds).

broker_heartbeat_checkrate

> **transports supported** `pyamqp`

Default: 2.0.

At intervals the worker will monitor that the broker hasn't missed too many heartbeats. The rate at which this is checked is calculated by dividing the *broker_heartbeat* value with this value, so if the heartbeat is 10.0 and the rate is the default 2.0, the check will be performed every 5 seconds (twice the heartbeat sending rate).

broker_use_ssl

> **transports supported** `pyamqp, redis`

Default: Disabled.

Toggles SSL usage on broker connection and SSL settings.

The valid values for this option vary by transport.

`pyamqp`

If `True` the connection will use SSL with default SSL settings. If set to a dict, will configure SSL connection according to the specified policy. The format used is Python's `ssl.wrap_socket()` options.

Note that SSL socket is generally served on a separate port by the broker.

Example providing a client cert and validating the server cert against a custom certificate authority:

```python
import ssl

broker_use_ssl = {
  'keyfile': '/var/ssl/private/worker-key.pem',
  'certfile': '/var/ssl/amqp-server-cert.pem',
  'ca_certs': '/var/ssl/myca.pem',
  'cert_reqs': ssl.CERT_REQUIRED
}
```

> **Warning:** Be careful using `broker_use_ssl=True`. It's possible that your default configuration won't validate the server cert at all. Please read Python ssl module security considerations.

`redis`

The setting must be a dict with the following keys:

- **`ssl_cert_reqs` (required): one of the `SSLContext.verify_mode` values:**
    - `ssl.CERT_NONE`
    - `ssl.CERT_OPTIONAL`
    - `ssl.CERT_REQUIRED`
- `ssl_ca_certs` (optional): path to the CA certificate
- `ssl_certfile` (optional): path to the client certificate
- `ssl_keyfile` (optional): path to the client key

`broker_pool_limit`

New in version 2.3.

Default: 10.

The maximum number of connections that can be open in the connection pool.

The pool is enabled by default since version 2.5, with a default limit of ten connections. This number can be tweaked depending on the number of threads/green-threads (eventlet/gevent) using a connection. For example running eventlet with 1000 greenlets that use a connection to the broker, contention can arise and you should consider increasing the limit.

If set to `None` or 0 the connection pool will be disabled and connections will be established and closed for every use.

broker_connection_timeout

Default: 4.0.

The default timeout in seconds before we give up establishing a connection to the AMQP server. This setting is disabled when using gevent.

Note: The broker connection timeout only applies to a worker attempting to connect to the broker. It does not apply to producer sending a task, see *broker_transport_options* for how to provide a timeout for that situation.

---

broker_connection_retry

Default: Enabled.

Automatically try to re-establish the connection to the AMQP broker if lost.

The time between retries is increased for each retry, and is not exhausted before *broker_connection_max_retries* is exceeded.

broker_connection_max_retries

Default: 100.

Maximum number of retries before we give up re-establishing a connection to the AMQP broker.

If this is set to `0` or `None`, we'll retry forever.

broker_login_method

Default: `"AMQPLAIN"`.

Set custom amqp login method.

broker_transport_options

New in version 2.2.

Default: `{}` (empty mapping).

A dict of additional options passed to the underlying transport.

See your transport user manual for supported options (if any).

Example setting the visibility timeout (supported by Redis and SQS transports):

```
broker_transport_options = {'visibility_timeout': 18000}  # 5 hours
```

Example setting the producer connection maximum number of retries (so producers won't retry forever if the broker isn't available at the first task execution):

```
broker_transport_options = {'max_retries': 5}
```

Worker

imports

Default: `[]` (empty list).

A sequence of modules to import when the worker starts.

This is used to specify the task modules to import, but also to import signal handlers and additional remote control commands, etc.

The modules will be imported in the original order.

include

Default: `[]` (empty list).

Exact same semantics as `imports`, but can be used as a means to have different import categories.

The modules in this setting are imported after the modules in `imports`.

worker_concurrency

Default: Number of CPU cores.

The number of concurrent worker processes/threads/green threads executing tasks.

If you're doing mostly I/O you can have more processes, but if mostly CPU-bound, try to keep it close to the number of CPUs on your machine. If not set, the number of CPUs/cores on the host will be used.

worker_prefetch_multiplier

Default: 4.

How many messages to prefetch at a time multiplied by the number of concurrent processes. The default is 4 (four messages for each process). The default setting is usually a good choice, however – if you have very long running tasks waiting in the queue and you have to start the workers, note that the first worker to start will receive four times the number of messages initially. Thus the tasks may not be fairly distributed to the workers.

To disable prefetching, set `worker_prefetch_multiplier` to 1. Changing that setting to 0 will allow the worker to keep consuming as many messages as it wants.

For more on prefetching, read *Prefetch Limits*

Note: Tasks with ETA/countdown aren't affected by prefetch limits.

---

worker_lost_wait

Default: 10.0 seconds.

In some cases a worker may be killed without proper cleanup, and the worker may have published a result before terminating. This value specifies how long we wait for any missing results before raising a *WorkerLostError* exception.

worker_max_tasks_per_child

Maximum number of tasks a pool worker process can execute before it's replaced with a new one. Default is no limit.

worker_max_memory_per_child

Default: No limit. Type: int (kilobytes)

Maximum amount of resident memory, in kilobytes, that may be consumed by a worker before it will be replaced by a new worker. If a single task causes a worker to exceed this limit, the task will be completed, and the worker will be replaced afterwards.

Example:

```
worker_max_memory_per_child = 12000  # 12MB
```

worker_disable_rate_limits

Default: Disabled (rate limits enabled).

Disable all rate limits, even if tasks has explicit rate limits set.

worker_state_db

Default: `None`.

Name of the file used to stores persistent worker state (like revoked tasks). Can be a relative or absolute path, but be aware that the suffix *.db* may be appended to the file name (depending on Python version).

Can also be set via the *celery worker --statedb* argument.

worker_timer_precision

Default: 1.0 seconds.

Set the maximum time in seconds that the ETA scheduler can sleep between rechecking the schedule.

Setting this value to 1 second means the schedulers precision will be 1 second. If you need near millisecond precision you can set this to 0.1.

worker_enable_remote_control

Default: Enabled by default.

Specify if remote control of the workers is enabled.

worker_proc_alive_timeout

Default: 4.0.

The timeout in seconds (int/float) when waiting for a new worker process to start up.

Events

worker_send_task_events

Default: Disabled by default.

Send task-related events so that tasks can be monitored using tools like *flower*. Sets the default value for the workers
`-E` argument.

task_send_sent_event

New in version 2.2.

Default: Disabled by default.

If enabled, a `task-sent` event will be sent for every task so tasks can be tracked before they're consumed by a
worker.

event_queue_ttl

> **transports supported** `amqp`

Default: 5.0 seconds.

Message expiry time in seconds (int/float) for when messages sent to a monitor clients event queue is deleted
(`x-message-ttl`)

For example, if this value is set to 10 then a message delivered to this queue will be deleted after 10 seconds.

event_queue_expires

> **transports supported** `amqp`

Default: 60.0 seconds.

Expiry time in seconds (int/float) for when after a monitor clients event queue will be deleted (`x-expires`).

event_queue_prefix

Default: `"celeryev"`.

The prefix to use for event receiver queue names.

event_exchange

Default: `"celeryev"`.

Name of the event exchange.

> **Warning:** This option is in experimental stage, please use it with caution.

event_serializer

Default: `"json"`.

Message serialization format used when sending event messages.

See also:

Serializers.

Remote Control Commands

> **Note:** To disable remote control commands see the *worker_enable_remote_control* setting.

control_queue_ttl

Default: 300.0

Time in seconds, before a message in a remote control command queue will expire.

If using the default of 300 seconds, this means that if a remote control command is sent and no worker picks it up within 300 seconds, the command is discarded.

This setting also applies to remote control reply queues.

control_queue_expires

Default: 10.0

Time in seconds, before an unused remote control command queue is deleted from the broker.

This setting also applies to remote control reply queues.

control_exchange

Default: `"celery"`.

Name of the control command exchange.

> **Warning:** This option is in experimental stage, please use it with caution.

Logging

worker_hijack_root_logger

New in version 2.2.

Default: Enabled by default (hijack root logger).

By default any previously configured handlers on the root logger will be removed. If you want to customize your own logging handlers, then you can disable this behavior by setting *worker_hijack_root_logger = False*.

> **Note:** Logging can also be customized by connecting to the `celery.signals.setup_logging` signal.

worker_log_color

Default: Enabled if app is logging to a terminal.

Enables/disables colors in logging output by the Celery apps.

worker_log_format

Default:

```
"[%(asctime)s: %(levelname)s/%(processName)s] %(message)s"
```

The format to use for log messages.

See the Python `logging` module for more information about log formats.

worker_task_log_format

Default:

```
"[%(asctime)s: %(levelname)s/%(processName)s]
    [%(task_name)s(%(task_id)s)] %(message)s"
```

The format to use for log messages logged in tasks.

See the Python `logging` module for more information about log formats.

worker_redirect_stdouts

Default: Enabled by default.

If enabled *stdout* and *stderr* will be redirected to the current logger.

Used by **celery worker** and **celery beat**.

worker_redirect_stdouts_level

Default: `WARNING`.

The log level output to *stdout* and *stderr* is logged as. Can be one of `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `CRITICAL`.

Security

security_key

Default: `None`.

New in version 2.5.

The relative or absolute path to a file containing the private key used to sign messages when *Message Signing* is used.

security_certificate

Default: `None`.

New in version 2.5.

The relative or absolute path to an X.509 certificate file used to sign messages when *Message Signing* is used.

security_cert_store

Default: `None`.

New in version 2.5.

The directory containing X.509 certificates used for *Message Signing*. Can be a glob with wild-cards, (for example `/etc/certs/*.pem`).

security_digest

Default: `sha256`.

New in version 4.3.

A cryptography digest used to sign messages when *Message Signing* is used. https://cryptography.io/en/latest/hazmat/primitives/cryptographic-hashes/#module-cryptography.hazmat.primitives.hashes

Custom Component Classes (advanced)

`worker_pool`

Default: `"prefork"` (`celery.concurrency.prefork:TaskPool`).

Name of the pool class used by the worker.

Eventlet/Gevent

Never use this option to select the eventlet or gevent pool. You must use the `-P` option to **celery worker** instead, to ensure the monkey patches aren't applied too late, causing things to break in strange ways.

`worker_pool_restarts`

Default: Disabled by default.

If enabled the worker pool can be restarted using the `pool_restart` remote control command.

`worker_autoscaler`

New in version 2.2.

Default: `"celery.worker.autoscale:Autoscaler"`.

Name of the autoscaler class to use.

`worker_consumer`

Default: `"celery.worker.consumer:Consumer"`.

Name of the consumer class used by the worker.

`worker_timer`

Default: `"kombu.asynchronous.hub.timer:Timer"`.

Name of the ETA scheduler class used by the worker. Default is or set by the pool implementation.

Beat Settings (`celery beat`)

`beat_schedule`

Default: `{}` (empty mapping).

The periodic task schedule used by *beat*. See *Entries*.

beat_scheduler

Default: `"celery.beat:PersistentScheduler"`.

The default scheduler class. May be set to `"django_celery_beat.schedulers:DatabaseScheduler"` for instance, if used alongside django-celery-beat extension.

Can also be set via the `celery beat -S` argument.

beat_schedule_filename

Default: `"celerybeat-schedule"`.

Name of the file used by *PersistentScheduler* to store the last run times of periodic tasks. Can be a relative or absolute path, but be aware that the suffix *.db* may be appended to the file name (depending on Python version).

Can also be set via the `celery beat --schedule` argument.

beat_sync_every

Default: 0.

The number of periodic tasks that can be called before another database sync is issued. A value of 0 (default) means sync based on timing - default of 3 minutes as determined by scheduler.sync_every. If set to 1, beat will call sync after every task message sent.

beat_max_loop_interval

Default: 0.

The maximum number of seconds *beat* can sleep between checking the schedule.

The default for this value is scheduler specific. For the default Celery beat scheduler the value is 300 (5 minutes), but for the django-celery-beat database scheduler it's 5 seconds because the schedule may be changed externally, and so it must take changes to the schedule into account.

Also when running Celery beat embedded (`-B`) on Jython as a thread the max interval is overridden and set to 1 so that it's possible to shut down in a timely manner.

3.3.18 Documenting Tasks with Sphinx

This document describes how auto-generate documentation for Tasks using Sphinx.

celery.contrib.sphinx

Sphinx documentation plugin used to document tasks.

Introduction

Usage

Add the extension to your `docs/conf.py` configuration module:

```
extensions = (...,
              'celery.contrib.sphinx')
```

If you'd like to change the prefix for tasks in reference documentation then you can change the `celery_task_prefix` configuration value:

```
celery_task_prefix = '(task)'  # < default
```

With the extension installed *autodoc* will automatically find task decorated objects (e.g. when using the automodule directive) and generate the correct (as well as add a `(task)` prefix), and you can also refer to the tasks using *:task:proj.tasks.add* syntax.

Use `.. autotask::` to alternatively manually document a task.

class `celery.contrib.sphinx.`**`TaskDirective`**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

> Sphinx task directive.
>
> **`get_signature_prefix`**(*sig*)
>> May return a prefix to put before the object name in the signature.

class `celery.contrib.sphinx.`**`TaskDocumenter`**(*directive*, *name*, *indent=''*)

> Document task definitions.
>
> **classmethod `can_document_member`**(*member*, *membername*, *isattr*, *parent*)
>> Called to see if a member can be documented by this documenter.
>
> **`check_module`**()
>> Check if *self.object* is really defined in the module given by *self.modname*.
>
> **`document_members`**(*all_members=False*)
>> Generate reST for member documentation.
>>
>> If *all_members* is True, do all members, else those given by *self.options.members*.
>
> **`format_args`**()
>> Format the argument signature of *self.object*.
>>
>> Should return None if the object does not have a signature.

`celery.contrib.sphinx.`**`autodoc_skip_member_handler`**(*app*, *what*, *name*, *obj*, *skip*, *options*)

> Handler for autodoc-skip-member event.

`celery.contrib.sphinx.`**`setup`**(*app*)

> Setup Sphinx extension.

3.4 Django

> **Release** 4.4
>
> **Date** Mar 02, 2020

3.4.1 First steps with Django

Using Celery with Django

Note: Previous versions of Celery required a separate library to work with Django, but since 3.1 this is no longer the case. Django is supported out of the box now so this document only contains a basic way to integrate Celery and Django. You'll use the same API as non-Django users so you're recommended to read the *First Steps with Celery* tutorial first and come back to this tutorial. When you have a working example you can continue to the *Next Steps* guide.

Note: Celery 4.0 supports Django 1.8 and newer versions. Please use Celery 3.1 for versions older than Django 1.8.

To use Celery with your Django project you must first define an instance of the Celery library (called an "app")

If you have a modern Django project layout like:

```
– proj/
  – manage.py
  – proj/
    – __init__.py
    – settings.py
    – urls.py
```

then the recommended way is to create a new *proj/proj/celery.py* module that defines the Celery instance:

> **file** *proj/proj/celery.py*

```python
from __future__ import absolute_import, unicode_literals

import os

from celery import Celery

# set the default Django settings module for the 'celery' program.
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'proj.settings')

app = Celery('proj')

# Using a string here means the worker doesn't have to serialize
# the configuration object to child processes.
# - namespace='CELERY' means all celery-related configuration keys
#   should have a `CELERY_` prefix.
app.config_from_object('django.conf:settings', namespace='CELERY')

# Load task modules from all registered Django app configs.
app.autodiscover_tasks()


@app.task(bind=True)
def debug_task(self):
    print('Request: {0!r}'.format(self.request))
```

Then you need to import this app in your `proj/proj/__init__.py` module. This ensures that the app is loaded when Django starts so that the `@shared_task` decorator (mentioned later) will use it:

`proj/proj/__init__.py`:

```python
from __future__ import absolute_import, unicode_literals
```

(continues on next page)

```python
# This will make sure the app is always imported when
# Django starts so that shared_task will use this app.
from .celery import app as celery_app

__all__ = ('celery_app',)
```

Note that this example project layout is suitable for larger projects, for simple projects you may use a single contained module that defines both the app and tasks, like in the *First Steps with Celery* tutorial.

Let's break down what happens in the first module, first we import absolute imports from the future, so that our `celery.py` module won't clash with the library:

```python
from __future__ import absolute_import
```

Then we set the default `DJANGO_SETTINGS_MODULE` environment variable for the **celery** command-line program:

```python
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'proj.settings')
```

You don't need this line, but it saves you from always passing in the settings module to the `celery` program. It must always come before creating the app instances, as is what we do next:

```python
app = Celery('proj')
```

This is our instance of the library, you can have many instances but there's probably no reason for that when using Django.

We also add the Django settings module as a configuration source for Celery. This means that you don't have to use multiple configuration files, and instead configure Celery directly from the Django settings; but you can also separate them if wanted.

```python
app.config_from_object('django.conf:settings', namespace='CELERY')
```

The uppercase name-space means that all *Celery configuration options* must be specified in uppercase instead of lowercase, and start with CELERY_, so for example the *task_always_eager* setting becomes CELERY_TASK_ALWAYS_EAGER, and the *broker_url* setting becomes CELERY_BROKER_URL. This also applies to the workers settings, for instance, the *worker_concurrency* setting becomes CELERY_WORKER_CONCURRENCY.

You can pass the settings object directly instead, but using a string is better since then the worker doesn't have to serialize the object. The CELERY_ namespace is also optional, but recommended (to prevent overlap with other Django settings).

Next, a common practice for reusable apps is to define all tasks in a separate `tasks.py` module, and Celery does have a way to auto-discover these modules:

```python
app.autodiscover_tasks()
```

With the line above Celery will automatically discover tasks from all of your installed apps, following the `tasks.py` convention:

```
- app1/
    - tasks.py
    - models.py
- app2/
    - tasks.py
    - models.py
```

This way you don't have to manually add the individual modules to the *CELERY_IMPORTS* setting.

Finally, the `debug_task` example is a task that dumps its own request information. This is using the new `bind=True` task option introduced in Celery 3.1 to easily refer to the current task instance.

Using the `@shared_task` decorator

The tasks you write will probably live in reusable apps, and reusable apps cannot depend on the project itself, so you also cannot import your app instance directly.

The `@shared_task` decorator lets you create tasks without having any concrete app instance:

`demoapp/tasks.py`:

```python
# Create your tasks here
from __future__ import absolute_import, unicode_literals

from celery import shared_task
from demoapp.models import Widget


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


@shared_task
def mul(x, y):
    return x * y


@shared_task
def xsum(numbers):
    return sum(numbers)


@shared_task
def count_widgets():
    return Widget.objects.count()


@shared_task
def rename_widget(widget_id, name):
    w = Widget.objects.get(id=widget_id)
    w.name = name
    w.save()
```

See also:

You can find the full source code for the Django example project at: https://github.com/celery/celery/tree/master/examples/django/

Relative Imports

You have to be consistent in how you import the task module. For example, if you have `project.app` in `INSTALLED_APPS`, then you must also import the tasks `from project.app` or else the names of the tasks will end up being different.

See *Automatic naming and relative imports*

Extensions

`django-celery-results` - Using the Django ORM/Cache as a result backend

The django-celery-results extension provides result backends using either the Django ORM, or the Django Cache framework.

To use this with your project you need to follow these steps:

1. Install the django-celery-results library:

   ```
   $ pip install django-celery-results
   ```

2. Add `django_celery_results` to `INSTALLED_APPS` in your Django project's `settings.py`:

   ```
   INSTALLED_APPS = (
       ...,
       'django_celery_results',
   )
   ```

   Note that there is no dash in the module name, only underscores.

3. Create the Celery database tables by performing a database migrations:

   ```
   $ python manage.py migrate django_celery_results
   ```

4. Configure Celery to use the django-celery-results backend.

   Assuming you are using Django's `settings.py` to also configure Celery, add the following settings:

   ```
   CELERY_RESULT_BACKEND = 'django-db'
   ```

   For the cache backend you can use:

   ```
   CELERY_CACHE_BACKEND = 'django-cache'
   ```

   We can also use the cache defined in the CACHES setting in django.

   ```
   # celery setting.
   CELERY_CACHE_BACKEND = 'default'

   # django setting.
   CACHES = {
       'default': {
           'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
           'LOCATION': 'my_cache_table',
       }
   }
   ```

   For additional configuration options, view the *Task result backend settings* reference.

`django-celery-beat` **- Database-backed Periodic Tasks with Admin interface.**

See *Using custom scheduler classes* for more information.

Starting the worker process

In a production environment you'll want to run the worker in the background as a daemon - see *Daemonization* - but for testing and development it is useful to be able to start a worker instance by using the **celery worker** manage command, much as you'd use Django's **manage.py runserver**:

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

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

```
$ celery help
```

Known Issues

CONN_MAX_AGE other than zero is known to cause issues according to bug #4878. Until this is fixed, please set CONN_MAX_AGE to zero.

Where to go from here

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

3.5 Contributing

Welcome!

This document is fairly extensive and you aren't really expected to study this in detail for small contributions;

> The most important rule is that contributing must be easy and that the community is friendly and not nitpicking on details, such as coding style.

If you're reporting a bug you should read the Reporting bugs section below to ensure that your bug report contains enough information to successfully diagnose the issue, and if you're contributing code you should try to mimic the conventions you see surrounding the code you're working on, but in the end all patches will be cleaned up by the person merging the changes so don't worry too much.

- *Community Code of Conduct*
    - *Be considerate*
    - *Be respectful*
    - *Be collaborative*
    - *When you disagree, consult others*
    - *When you're unsure, ask for help*
    - *Step down considerately*

3.5.1 Community Code of Conduct

The goal is to maintain a diverse community that's pleasant for everyone. That's why we would greatly appreciate it if everyone contributing to and interacting with the community also followed this Code of Conduct.

The Code of Conduct covers our behavior as members of the community, in any forum, mailing list, wiki, website, Internet relay chat (IRC), public meeting or private correspondence.

The Code of Conduct is heavily based on the Ubuntu Code of Conduct, and the Pylons Code of Conduct.

Be considerate

Your work will be used by other people, and you in turn will depend on the work of others. Any decision you take will affect users and colleagues, and we expect you to take those consequences into account when making decisions. Even if it's not obvious at the time, our contributions to Celery will impact the work of others. For example, changes to code, infrastructure, policy, documentation and translations during a release may negatively impact others' work.

Be respectful

The Celery community and its members treat one another with respect. Everyone can make a valuable contribution to Celery. We may not always agree, but disagreement is no excuse for poor behavior and poor manners. We might all experience some frustration now and then, but we cannot allow that frustration to turn into a personal attack. It's important to remember that a community where people feel uncomfortable or threatened isn't a productive one. We expect members of the Celery community to be respectful when dealing with other contributors as well as with people outside the Celery project and with users of Celery.

Be collaborative

Collaboration is central to Celery and to the larger free software community. We should always be open to collaboration. Your work should be done transparently and patches from Celery should be given back to the community when they're made, not just when the distribution releases. If you wish to work on new code for existing upstream projects, at least keep those projects informed of your ideas and progress. It many not be possible to get consensus from upstream, or even from your colleagues about the correct implementation for an idea, so don't feel obliged to have that agreement before you begin, but at least keep the outside world informed of your work, and publish your work in a way that allows outsiders to test, discuss, and contribute to your efforts.

When you disagree, consult others

Disagreements, both political and technical, happen all the time and the Celery community is no exception. It's important that we resolve disagreements and differing views constructively and with the help of the community and community process. If you really want to go a different way, then we encourage you to make a derivative distribution or alternate set of packages that still build on the work we've done to utilize as common of a core as possible.

When you're unsure, ask for help

Nobody knows everything, and nobody is expected to be perfect. Asking questions avoids many problems down the road, and so questions are encouraged. Those who are asked questions should be responsive and helpful. However, when asking a question, care must be taken to do so in an appropriate forum.

Step down considerately

Developers on every project come and go and Celery is no different. When you leave or disengage from the project, in whole or in part, we ask that you do so in a way that minimizes disruption to the project. This means you should tell people you're leaving and take the proper steps to ensure that others can pick up where you left off.

3.5.2 Reporting Bugs

Security

You must never report security related issues, vulnerabilities or bugs including sensitive information to the bug tracker, or elsewhere in public. Instead sensitive bugs must be sent by email to `security@celeryproject.org`.

If you'd like to submit the information encrypted our PGP key is:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.15 (Darwin)
```

(continues on next page)

```
mQENBFJpWDkBCADFIc9/Fpgse4owLNvsTC7GYfnJL19XO0hnL99sPx+DPbfr+cSE
9wiU+Wp2TfUX7pCLEGrODiEP6ZCZbgtiPgId+JYvMxpP6GXbjiIlHRw1EQNH8RlX
cVxy3rQfVv8PGGiJuyBBjxzvETHW25htVAZ5TI1+CkxmuyyEYqgZN2fNd0wEU19D
+c10G1gSECbCQTCbacLSzdpngAt1Gkrc96r7wGHBBSvDaGDD2pFSkVuTLMbIRrVp
lnKOPMsUijiip2EMr2DvfuXiUIUvaqInTPNWkDynLoh69ib5xC19CSVLONjkKBsr
Pe+qAY29liBatatpXsydY7GIUzyBT3MzgMJlABEBAAG0MUNlbGVyeSBTZWN1cml0
eSBUZWFtIDxzZWN1cml0eUBjZWxlcnlwcm9qZWN0Lm9yZz6JATgEEwECACIFAlJp
WDkCGwMGCwkIBwMCBhUIAgkKCwQWAgMBAh4BAheAAAoJEOArFOUDCicIw1IH/26f
CViDC7/P13jr+srRdjAsWvQztia9HmTlY8cUnbmkR9w6b6j3F2ayw8VhkyFWgYEJ
wtPBv8mHKADiVSFARS+0yGsfCkia5wDSQuIv6XqRlIrXUyqJbmF4NUFTyCZYoh+C
ZiQpN9xGhFPr5QDlMx2izWg1rvWlG1jY2Es1v/xED3AeCOB1eUGvRe/uJHKjGv7J
rj0pFcptZX+WDF22AN235WYwgJM6TrNfSu8sv8vNAQOVnsKcgsqhuwomSGsOfMQj
LFzIn95MKBBU1G5wOs7JtwiV9jefGqJGBO2FAvOVbvPdK/saSnB+7K36dQcIHqms
5hU4Xj0RIJiod5idlRC5AQ0EUmlYOQEIAJs8OwHMkrdcvy9kk2HBVbdqhgAREMKy
gmphDp7prRL9FqSY/dKpCbG0u82zyJypdb7QiaQ5pfPzPpQcd2dIcohkkh7G3E+e
hS2L9AXHpwR26/PzMBXyr2iNnNc4vTksHvGVDxzFnRpka6vbI/hrrZmYNYh9EAiv
uhE54b3/XhXwFgHjZXb9i8hgJ3nsO0pRwvUAM1bRGMbvf8e9F+kqgV0yWYNnh6QL
4Vpl1+epqp2RKPHyNQftbQyrAHXT9kQF9pPlx013MKYaFTADscuAp4T3dy7xmiwS
crqMbZLzfrxfFOsNxTUGE5vmJCcm+mybAtRo4aV6ACohAO9NevMx8pUAEQEAAYkB
HwQYAQIACQUCUmlYOQIbDAAKCRDgKxTlAwonCNFbB/9esir/f7TufE+isNqErzR/
aZKZo2WzZR9c75kbqo6J6DYuUHe6xI0OZ2qZ60iABDEZAiNXGulysFLCiPdatQ8x
8zt3DF9BMkEck54ZvAjpNSern6zfZb1jPYWZq3TKxlTs/GuCgBAuV4i5vDTZ7xK/
aF+OFY5zN7ciZHkqLgMiTZ+RhqRcK6FhVBP/Y7d9NlBOcDBTxxE1ZO1ute6n7guJ
ciw4hfoRk8qNN19szZuq3UU64zpkM2sBsIFM9tGF2FADRxiOaOWZHmIyVZriPFqW
RUwjSjs7jBVNq0Vy4fCu/5+e+XLOUBOoqtM5W7ELt0t1w9tXebtPEetV86in8fU2
=0chn
-----END PGP PUBLIC KEY BLOCK-----
```

Other bugs

Bugs can always be described to the *Mailing list*, but the best way to report an issue and to ensure a timely response is to use the issue tracker.

1) **Create a GitHub account**.

You need to create a GitHub account to be able to create new issues and participate in the discussion.

2) **Determine if your bug is really a bug**.

You shouldn't file a bug if you're requesting support. For that you can use the *Mailing list*, or *IRC*. If you still need support you can open a github issue, please prepend the title with [QUESTION].

3) **Make sure your bug hasn't already been reported**.

Search through the appropriate Issue tracker. If a bug like yours was found, check if you have new information that could be reported to help the developers fix the bug.

4) **Check if you're using the latest version**.

A bug could be fixed by some other improvements and fixes - it might not have an existing report in the bug tracker. Make sure you're using the latest releases of celery, billiard, kombu, amqp, and vine.

5) **Collect information about the bug**.

To have the best chance of having a bug fixed, we need to be able to easily reproduce the conditions that caused it. Most of the time this information will be from a Python traceback message, though some bugs might be in design, spelling or other errors on the website/docs/code.

A) If the error is from a Python traceback, include it in the bug report.

B) We also need to know what platform you're running (Windows, macOS, Linux, etc.), the version of your Python interpreter, and the version of Celery, and related packages that you were running when the bug occurred.

C) If you're reporting a race condition or a deadlock, tracebacks can be hard to get or might not be that useful. Try to inspect the process to get more diagnostic data. Some ideas:

  - Enable Celery's *breakpoint signal* and use it to inspect the process's state. This will allow you to open a `pdb` session.

  - Collect tracing data using *strace'_(Linux), :command:'dtruss* (macOS), and **ktrace** (BSD), ltrace, and lsof.

D) Include the output from the **celery report** command:

```
$ celery -A proj report
```

This will also include your configuration settings and it will try to remove values for keys known to be sensitive, but make sure you also verify the information before submitting so that it doesn't contain confidential information like API tokens and authentication credentials.

E) Your issue might be tagged as *Needs Test Case*. A test case represents all the details needed to reproduce what your issue is reporting. A test case can be some minimal code that reproduces the issue or detailed instructions and configuration values that reproduces said issue.

6) **Submit the bug**.

By default GitHub will email you to let you know when new comments have been made on your bug. In the event you've turned this feature off, you should check back on occasion to ensure you don't miss any questions a developer trying to fix the bug might ask.

Issue Trackers

Bugs for a package in the Celery ecosystem should be reported to the relevant issue tracker.

  - celery: https://github.com/celery/celery/issues/

  - kombu: https://github.com/celery/kombu/issues

  - amqp: https://github.com/celery/py-amqp/issues

  - vine: https://github.com/celery/vine/issues

  - librabbitmq: https://github.com/celery/librabbitmq/issues

  - django-celery-beat: https://github.com/celery/django-celery-beat/issues

  - django-celery-results: https://github.com/celery/django-celery-results/issues

If you're unsure of the origin of the bug you can ask the *Mailing list*, or just use the Celery issue tracker.

3.5.3 Contributors guide to the code base

There's a separate section for internal details, including details about the code base and a style guide.

Read *Contributors Guide to the Code* for more!

3.5.4 Versions

Version numbers consists of a major version, minor version and a release number. Since version 2.1.0 we use the versioning semantics described by SemVer: http://semver.org.

Stable releases are published at PyPI while development releases are only available in the GitHub git repository as tags. All version tags starts with "v", so version 0.8.0 has the tag v0.8.0.

3.5.5 Branches

Current active version branches:

- dev (which git calls "master") (https://github.com/celery/celery/tree/master)

- 4.2 (https://github.com/celery/celery/tree/4.2)

- 4.1 (https://github.com/celery/celery/tree/4.1)

- 3.1 (https://github.com/celery/celery/tree/3.1)

You can see the state of any branch by looking at the Changelog:

> https://github.com/celery/celery/blob/master/Changelog

If the branch is in active development the topmost version info should contain meta-data like:

```
4.3.0
======
:release-date: TBA
:status: DEVELOPMENT
:branch: dev (git calls this master)
```

The `status` field can be one of:

- PLANNING

   The branch is currently experimental and in the planning stage.

- DEVELOPMENT

   The branch is in active development, but the test suite should be passing and the product should be working and possible for users to test.

- FROZEN

   The branch is frozen, and no more features will be accepted. When a branch is frozen the focus is on testing the version as much as possible before it is released.

dev branch

The dev branch (called "master" by git), is where development of the next version happens.

Maintenance branches

Maintenance branches are named after the version – for example, the maintenance branch for the 2.2.x series is named `2.2`.

Previously these were named `releaseXX-maint`.

The versions we currently maintain is:

- 4.2

  This is the current series.

- 4.1

  Drop support for python 2.6. Add support for python 3.4, 3.5 and 3.6.

- 3.1

  Official support for python 2.6, 2.7 and 3.3, and also supported on PyPy.

Archived branches

Archived branches are kept for preserving history only, and theoretically someone could provide patches for these if they depend on a series that's no longer officially supported.

An archived version is named `X.Y-archived`.

To maintain a cleaner history and drop compatibility to continue improving the project, we **do not have any archived version** right now.

Feature branches

Major new features are worked on in dedicated branches. There's no strict naming requirement for these branches.

Feature branches are removed once they've been merged into a release branch.

3.5.6 Tags

- Tags are used exclusively for tagging releases. A release tag is named with the format `vX.Y.Z` – for example `v2.3.1`.
- Experimental releases contain an additional identifier `vX.Y.Z-id` – for example `v3.0.0-rc1`.
- Experimental tags may be removed after the official release.

3.5.7 Working on Features & Patches

Note: Contributing to Celery should be as simple as possible, so none of these steps should be considered mandatory.

You can even send in patches by email if that's your preferred work method. We won't like you any less, any contribution you make is always appreciated!

However, following these steps may make maintainer's life easier, and may mean that your changes will be accepted sooner.

Forking and setting up the repository

First you need to fork the Celery repository, a good introduction to this is in the GitHub Guide: Fork a Repo.

After you have cloned the repository, you should checkout your copy to a directory on your machine:

```
$ git clone git@github.com:username/celery.git
```

When the repository is cloned, enter the directory to set up easy access to upstream changes:

```
$ cd celery
$ git remote add upstream git://github.com/celery/celery.git
$ git fetch upstream
```

If you need to pull in new changes from upstream you should always use the `--rebase` option to `git pull`:

```
git pull --rebase upstream master
```

With this option, you don't clutter the history with merging commit notes. See Rebasing merge commits in git. If you want to learn more about rebasing, see the Rebase section in the GitHub guides.

If you need to work on a different branch than the one git calls `master`, you can fetch and checkout a remote branch like this:

```
git checkout --track -b 5.0-devel upstream/5.0-devel
```

Note: Any feature or fix branch should be created from `upstream/master`.

Developing and Testing with Docker

Because of the many components of Celery, such as a broker and backend, Docker and docker-compose can be utilized to greatly simplify the development and testing cycle. The Docker configuration here requires a Docker version of at least 17.13.0 and *docker-compose* 1.13.0+.

The Docker components can be found within the `docker/` folder and the Docker image can be built via:

```
$ docker-compose build celery
```

and run via:

```
$ docker-compose run --rm celery <command>
```

where <command> is a command to execute in a Docker container. The *–rm* flag indicates that the container should be removed after it is exited and is useful to prevent accumulation of unwanted containers.

Some useful commands to run:

- `bash`

    To enter the Docker container like a normal shell

- `make test`

    To run the test suite. **Note:** This will run tests using python 3.8 by default.

- `tox`

    To run tox and test against a variety of configurations. **Note:** This command will run tests for every environment defined in `tox.ini`. It takes a while.

- `pyenv exec python{2.7,3.5,3.6,3.7,3.8} -m pytest t/unit`

    To run unit tests using pytest.

    Note: `{2.7,3.5,3.6,3.7,3.8}` means you can use any of those options. e.g. `pyenv exec python3.6 -m pytest t/unit`

- `pyenv exec python{2.7,3.5,3.6,3.7,3.8} -m pytest t/integration`

> To run integration tests using pytest
>
> **Note:** `{2.7,3.5,3.6,3.7,3.8}` means you can use any of those options. e.g. `pyenv exec python3.6 -m pytest t/unit`

By default, docker-compose will mount the Celery and test folders in the Docker container, allowing code changes and testing to be immediately visible inside the Docker container. Environment variables, such as the broker and backend to use are also defined in the `docker/docker-compose.yml` file.

By running `docker-compose build celery` an image will be created with the name `celery/celery:dev`. This docker image has every dependency needed for development installed. `pyenv` is used to install multiple python versions, the docker image offers python 2.7, 3.5, 3.6, 3.7 and 3.8. The default python version is set to 3.8.

The `docker-compose.yml` file defines the necessary environment variables to run integration tests. The `celery` service also mounts the codebase and sets the `PYTHONPATH` environment variable to `/home/developer/celery`. By setting `PYTHONPATH` the service allows to use the mounted codebase as global module for development. If you prefer, you can also run `python -m pip install -e .` to install the codebase in development mode.

If you would like to run a Django or stand alone project to manually test or debug a feature, you can use the image built by *docker-compose* and mount your custom code. Here's an example:

Assuming a folder structure such as:

```
+ celery_project
  + celery # repository cloned here.
  + my_project
    - manage.py
    + my_project
      - views.py
```

```yaml
version: "3"

services:
    celery:
        image: celery/celery:dev
        environment:
            TEST_BROKER: amqp://rabbit:5672
            TEST_BACKEND: redis://redis
        volumes:
            - ../../celery:/home/developer/celery
            - ../my_project:/home/developer/my_project
        depends_on:
            - rabbit
            - redis
    rabbit:
        image: rabbitmq:latest
    redis:
        image: redis:latest
```

In the previous example, we are using the image that we can build from this repository and mounting the celery code base as well as our custom project.

Running the unit test suite

If you like to develop using virtual environments or just outside docker, you must make sure all necessary dependencies are installed. There are multiple requirements files to make it easier to install all dependencies. You do not have to use every requirements file but you must use *default.txt*.

```
# pip install -U -r requirements/default.txt
```

To run the Celery test suite you need to install `requirements/test.txt`.

```
$ pip install -U -r requirements/test.txt
$ pip install -U -r requirements/default.txt
```

After installing the dependencies required, you can now execute the test suite by calling py.test:

```
$ py.test t/unit
$ py.test t/integration
```

Some useful options to **py.test** are:

- `-x`

    Stop running the tests at the first test that fails.

- `-s`

    Don't capture output

- `-v`

    Run with verbose output.

If you want to run the tests for a single test file only you can do so like this:

```
$ py.test t/unit/worker/test_worker.py
```

Calculating test coverage

To calculate test coverage you must first install the pytest-cov module.

Installing the pytest-cov module:

```
$ pip install -U pytest-cov
```

Code coverage in HTML format

1. Run **py.test** with the `--cov-report=html` argument enabled:

    ```
    $ py.test --cov=celery --cov-report=html
    ```

2. The coverage output will then be located in the `htmlcov/` directory:

    ```
    $ open htmlcov/index.html
    ```

Code coverage in XML (Cobertura-style)

1. Run **py.test** with the `--cov-report=xml` argument enabled:

```
$ py.test --cov=celery --cov-report=xml
```

1. The coverage XML output will then be located in the `coverage.xml` file.

Running the tests on all supported Python versions

There's a tox configuration file in the top directory of the distribution.

To run the tests for all supported Python versions simply execute:

```
$ tox
```

Use the `tox -e` option if you only want to test specific Python versions:

```
$ tox -e 2.7
```

Building the documentation

To build the documentation, you need to install the dependencies listed in `requirements/docs.txt` and `requirements/default.txt`:

```
$ pip install -U -r requirements/docs.txt
$ pip install -U -r requirements/default.txt
```

Additionally, to build with no warnings, you will need to install the following packages:

```
$ apt-get install texlive texlive-latex-extra dvipng
```

After these dependencies are installed, you should be able to build the docs by running:

```
$ cd docs
$ rm -rf _build
$ make html
```

Make sure there are no errors or warnings in the build output. After building succeeds, the documentation is available at `_build/html`.

Verifying your contribution

To use these tools, you need to install a few dependencies. These dependencies can be found in `requirements/pkgutils.txt`.

Installing the dependencies:

```
$ pip install -U -r requirements/pkgutils.txt
```

pyflakes & PEP-8

To ensure that your changes conform to **PEP 8** and to run pyflakes execute:

```
$ make flakecheck
```

To not return a negative exit code when this command fails, use the `flakes` target instead:

```
$ make flakes
```

API reference

To make sure that all modules have a corresponding section in the API reference, please execute:

```
$ make apicheck
```

If files are missing, you can add them by copying an existing reference file.

If the module is internal, it should be part of the internal reference located in `docs/internals/reference/`. If the module is public, it should be located in `docs/reference/`.

For example, if reference is missing for the module `celery.worker.awesome` and this module is considered part of the public API, use the following steps:

Use an existing file as a template:

```
$ cd docs/reference/
$ cp celery.schedules.rst celery.worker.awesome.rst
```

Edit the file using your favorite editor:

```
$ vim celery.worker.awesome.rst

    # change every occurrence of ``celery.schedules`` to
    # ``celery.worker.awesome``
```

Edit the index using your favorite editor:

```
$ vim index.rst

    # Add ``celery.worker.awesome`` to the index.
```

Commit your changes:

```
# Add the file to git
$ git add celery.worker.awesome.rst
$ git add index.rst
$ git commit celery.worker.awesome.rst index.rst \
    -m "Adds reference for celery.worker.awesome"
```

Isort

Isort is a python utility to help sort imports alphabetically and separated into sections. The Celery project uses isort to better maintain imports on every module. Please run isort if there are any new modules or the imports on an existent module had to be modified.

```
$ isort my_module.py # Run isort for one file
$ isort -rc . # Run it recursively
$ isort m_module.py --diff # Do a dry-run to see the proposed changes
```

Creating pull requests

When your feature/bugfix is complete, you may want to submit a pull request, so that it can be reviewed by the maintainers.

Before submitting a pull request, please make sure you go through this checklist to make it easier for the maintainers to accept your proposed changes:

- [ ] **Make sure any change or new feature has a unit and/or integration test.** If a test is not written, a label will be assigned to your PR with the name `Needs Test Coverage`.

- [ ] **Make sure unit test coverage does not decrease.** `py.test -xv --cov=celery --cov-report=xml --cov-report term.` You can check the current test coverage here: https://codecov.io/gh/celery/celery

- [ ] **Run `flake8` against the code. The following commands are valid** and equivalent.:

  ```
  $ flake8 -j 2 celery/ t/
  $ make flakecheck
  $ tox -e flake8
  ```

- [ ] **Run `flakeplus` against the code. The following commands are valid** and equivalent.:

  ```
  $ flakeplus --2.7 celery/ t/
  $ make flakes
  $ tox -e flakeplus
  ```

- [ ] **Run `pydocstyle` against the code. The following commands are valid** and equivalent.:

  ```
  $ pydocstyle celery/
  $ tox -e pydocstyle
  ```

- [ ] **Build api docs to make sure everything is OK. The following commands are valid** and equivalent.:

  ```
  $ make apicheck
  $ cd docs && sphinx-build -b apicheck -d _build/doctrees . _build/apicheck
  $ tox -e apicheck
  ```

- [ ] **Build configcheck. The following commands are valid** and equivalent.:

  ```
  $ make configcheck
  $ cd docs && sphinx-build -b configcheck -d _build/doctrees  . _build/
  →configcheck
  $ tox -e configcheck
  ```

- [ ] **Run `bandit` to make sure there's no security issues. The following commands are valid** and equivalent.:

  ```
  $ pip install -U bandit
  $ bandit -b bandit.json celery/
  $ tox -e bandit
  ```

- [ ] **Run unit and integration tests for every python version. The following commands are valid** and equivalent.:

  ```
  $ tox -v
  ```

- [ ] Confirm `isort` on any new or modified imports:

  ```
  $ isort my_module.py --diff
  ```

Creating pull requests is easy, and they also let you track the progress of your contribution. Read the Pull Requests section in the GitHub Guide to learn how this is done.

You can also attach pull requests to existing issues by following the steps outlined here: https://bit.ly/koJoso

You can also use hub to create pull requests. Example: https://theiconic.tech/git-hub-fbe2e13ef4d1

Status Labels

There are different labels used to easily manage github issues and PRs. Most of these labels make it easy to categorize each issue with important details. For instance, you might see a `Component:canvas` label on an issue or PR. The `Component:canvas` label means the issue or PR corresponds to the canvas functionality. These labels are set by the maintainers and for the most part external contributors should not worry about them. A subset of these labels are prepended with **Status:**. Usually the **Status:** labels show important actions which the issue or PR needs. Here is a summary of such statuses:

- **Status: Cannot Reproduce**

  One or more Celery core team member has not been able to reproduce the issue.

- **Status: Confirmed**

  The issue or PR has been confirmed by one or more Celery core team member.

- **Status: Duplicate**

  A duplicate issue or PR.

- **Status: Feedback Needed**

  One or more Celery core team member has asked for feedback on the issue or PR.

- **Status: Has Testcase**

  It has been confirmed the issue or PR includes a test case. This is particularly important to correctly write tests for any new feature or bug fix.

- **Status: In Progress**

  The PR is still in progress.

- **Status: Invalid**

  The issue reported or the PR is not valid for the project.

- **Status: Needs Documentation**

  The PR does not contain documentation for the feature or bug fix proposed.

- **Status: Needs Rebase**

  The PR has not been rebased with `master`. It is very important to rebase PRs before they can be merged to `master` to solve any merge conflicts.

- **Status: Needs Test Coverage**

  Celery uses codecov to verify code coverage. Please make sure PRs do not decrease code coverage. This label will identify PRs which need code coverage.

- **Status: Needs Test Case**

  The issue or PR needs a test case. A test case can be a minimal code snippet that reproduces an issue or a detailed set of instructions and configuration values that reproduces the issue reported. If possible a test case can be submitted in the form of a PR to Celery's integration suite. The test case will be marked as failed until the bug is fixed. When a test case cannot be run by Celery's integration suite, then it's better to describe in the issue itself.

- **Status: Needs Verification**

  This label is used to notify other users we need to verify the test case offered by the reporter and/or we need to include the test in our integration suite.

- **Status: Not a Bug**

  It has been decided the issue reported is not a bug.

- **Status: Won't Fix**

  It has been decided the issue will not be fixed. Sadly the Celery project does not have unlimited resources and sometimes this decision has to be made. Although, any external contributors are invited to help out even if an issue or PR is labeled as `Status:  Won't Fix`.

- **Status: Works For Me**

  One or more Celery core team members have confirmed the issue reported works for them.

3.5.8 Coding Style

You should probably be able to pick up the coding style from surrounding code, but it is a good idea to be aware of the following conventions.

- All Python code must follow the **PEP 8** guidelines.

pep8 is a utility you can use to verify that your code is following the conventions.

- Docstrings must follow the **PEP 257** conventions, and use the following style.

    Do this:

    ```python
    def method(self, arg):
        """Short description.

        More details.

        """
    ```

    or:

    ```python
    def method(self, arg):
        """Short description."""
    ```

    but not this:

    ```python
    def method(self, arg):
        """
        Short description.
        """
    ```

- Lines shouldn't exceed 78 columns.

    You can enforce this in **vim** by setting the `textwidth` option:

    ```
    set textwidth=78
    ```

    If adhering to this limit makes the code less readable, you have one more character to go on. This means 78 is a soft limit, and 79 is the hard limit :)

- Import order

- Python standard library (*import xxx*)
- Python standard library (*from xxx import*)
- Third-party packages.
- Other modules from the current package.

or in case of code using Django:

- Python standard library (*import xxx*)
- Python standard library (*from xxx import*)
- Third-party packages.
- Django packages.
- Other modules from the current package.

Within these sections the imports should be sorted by module name.

Example:

```python
import threading
import time

from collections import deque
from Queue import Queue, Empty

from .platforms import Pidfile
from .five import zip_longest, items, range
from .utils.time import maybe_timedelta
```

- Wild-card imports must not be used (*from xxx import **).

- For distributions where Python 2.5 is the oldest support version, additional rules apply:

  - Absolute imports must be enabled at the top of every module:

    ```python
    from __future__ import absolute_import
    ```

  - If the module uses the `with` statement and must be compatible with Python 2.5 (celery isn't), then it must also enable that:

    ```python
    from __future__ import with_statement
    ```

  - Every future import must be on its own line, as older Python 2.5 releases didn't support importing multiple features on the same future import line:

    ```python
    # Good
    from __future__ import absolute_import
    from __future__ import with_statement

    # Bad
    from __future__ import absolute_import, with_statement
    ```

    (Note that this rule doesn't apply if the package doesn't include support for Python 2.5)

- Note that we use "new-style" relative imports when the distribution doesn't support Python versions below 2.5

  This requires Python 2.5 or later:

```
from . import submodule
```

3.5.9 Contributing features requiring additional libraries

Some features like a new result backend may require additional libraries that the user must install.

We use setuptools *extra_requires* for this, and all new optional features that require third-party libraries must be added.

1) Add a new requirements file in *requirements/extras*

   For the Cassandra backend this is `requirements/extras/cassandra.txt`, and the file looks like this:

   ```
   pycassa
   ```

   These are pip requirement files, so you can have version specifiers and multiple packages are separated by newline. A more complex example could be:

   ```
   # pycassa 2.0 breaks Foo
   pycassa>=1.0,<2.0
   thrift
   ```

2) Modify `setup.py`

   After the requirements file is added, you need to add it as an option to `setup.py` in the `extras_require` section:

   ```
   extra['extras_require'] = {
       # ...
       'cassandra': extras('cassandra.txt'),
   }
   ```

3) Document the new feature in `docs/includes/installation.txt`

   You must add your feature to the list in the *Bundles* section of `docs/includes/installation.txt`.

   After you've made changes to this file, you need to render the distro `README` file:

   ```
   $ pip install -U requirements/pkgutils.txt
   $ make readme
   ```

That's all that needs to be done, but remember that if your feature adds additional configuration options, then these needs to be documented in `docs/configuration.rst`. Also, all settings need to be added to the `celery/app/defaults.py` module.

Result backends require a separate section in the `docs/configuration.rst` file.

3.5.10 Contacts

This is a list of people that can be contacted for questions regarding the official git repositories, PyPI packages Read the Docs pages.

If the issue isn't an emergency then it's better to *report an issue*.

Committers

Ask Solem

github https://github.com/ask

twitter https://twitter.com/#!/asksol

Asif Saif Uddin

github https://github.com/auvipy

twitter https://twitter.com/#!/auvipy

Dmitry Malinovsky

github https://github.com/malinoff

twitter https://twitter.com/__malinoff__

Ionel Cristian Mărieș

github https://github.com/ionelmc

twitter https://twitter.com/ionelmc

Mher Movsisyan

github https://github.com/mher

twitter https://twitter.com/#!/movsm

Omer Katz

github https://github.com/thedrow

twitter https://twitter.com/the_drow

Steeve Morin

github https://github.com/steeve

twitter https://twitter.com/#!/steeve

Josue Balandrano Coronel

github https://github.com/xirdneh

twitter https://twitter.com/eusoj_xirdneh

Website

The Celery Project website is run and maintained by

Mauro Rocco

> **github** https://github.com/fireantology
>
> **twitter** https://twitter.com/#!/fireantology

with design by:

Jan Henrik Helmers

> **web** http://www.helmersworks.com
>
> **twitter** https://twitter.com/#!/helmers

3.5.11 Packages

`celery`

> **git** https://github.com/celery/celery
>
> **CI** https://travis-ci.org/#!/celery/celery
>
> **Windows-CI** https://ci.appveyor.com/project/ask/celery
>
> **PyPI** celery
>
> **docs** http://docs.celeryproject.org

`kombu`

Messaging library.

> **git** https://github.com/celery/kombu
>
> **CI** https://travis-ci.org/#!/celery/kombu
>
> **Windows-CI** https://ci.appveyor.com/project/ask/kombu
>
> **PyPI** kombu
>
> **docs** https://kombu.readthedocs.io

`amqp`

Python AMQP 0.9.1 client.

> **git** https://github.com/celery/py-amqp
>
> **CI** https://travis-ci.org/#!/celery/py-amqp
>
> **Windows-CI** https://ci.appveyor.com/project/ask/py-amqp
>
> **PyPI** amqp

    docs https://amqp.readthedocs.io

vine

Promise/deferred implementation.

    git https://github.com/celery/vine/

    CI https://travis-ci.org/#!/celery/vine/

    Windows-CI https://ci.appveyor.com/project/ask/vine

    PyPI vine

    docs https://vine.readthedocs.io

billiard

Fork of multiprocessing containing improvements that'll eventually be merged into the Python stdlib.

    git https://github.com/celery/billiard

    CI https://travis-ci.org/#!/celery/billiard/

    Windows-CI https://ci.appveyor.com/project/ask/billiard

    PyPI billiard

django-celery-beat

Database-backed Periodic Tasks with admin interface using the Django ORM.

    git https://github.com/celery/django-celery-beat

    CI https://travis-ci.org/#!/celery/django-celery-beat

    Windows-CI https://ci.appveyor.com/project/ask/django-celery-beat

    PyPI django-celery-beat

django-celery-results

Store task results in the Django ORM, or using the Django Cache Framework.

    git https://github.com/celery/django-celery-results

    CI https://travis-ci.org/#!/celery/django-celery-results

    Windows-CI https://ci.appveyor.com/project/ask/django-celery-results

    PyPI django-celery-results

librabbitmq

Very fast Python AMQP client written in C.

    git https://github.com/celery/librabbitmq

    PyPI librabbitmq

cell

Actor library.

> **git** https://github.com/celery/cell
>
> **PyPI** cell

cyme

Distributed Celery Instance manager.

> **git** https://github.com/celery/cyme
>
> **PyPI** cyme
>
> **docs** https://cyme.readthedocs.io/

Deprecated

- `django-celery`

  git https://github.com/celery/django-celery

  PyPI django-celery

  docs http://docs.celeryproject.org/en/latest/django

- `Flask-Celery`

  git https://github.com/ask/Flask-Celery

  PyPI Flask-Celery

- `celerymon`

  git https://github.com/celery/celerymon

  PyPI celerymon

- `carrot`

  git https://github.com/ask/carrot

  PyPI carrot

- `ghettoq`

  git https://github.com/ask/ghettoq

  PyPI ghettoq

- `kombu-sqlalchemy`

  git https://github.com/ask/kombu-sqlalchemy

  PyPI kombu-sqlalchemy

- `django-kombu`

  git https://github.com/ask/django-kombu

  PyPI django-kombu

- `pylibrabbitmq`

Old name for librabbitmq.

> **git** `None`
>
> **PyPI** pylibrabbitmq

3.5.12 Release Procedure

Updating the version number

The version number must be updated in three places:

- `celery/__init__.py`
- `docs/include/introduction.txt`
- `README.rst`

The changes to the previous files can be handled with the [*bumpversion* command line tool] (https://pypi.org/project/bumpversion/). The corresponding configuration lives in `.bumpversion.cfg`. To do the necessary changes, run:

```
$ bumpversion
```

After you have changed these files, you must render the `README` files. There's a script to convert sphinx syntax to generic reStructured Text syntax, and the make target *readme* does this for you:

```
$ make readme
```

Now commit the changes:

```
$ git commit -a -m "Bumps version to X.Y.Z"
```

and make a new version tag:

```
$ git tag vX.Y.Z
$ git push --tags
```

Releasing

Commands to make a new public stable release:

```
$ make distcheck  # checks pep8, autodoc index, runs tests and more
$ make dist  # NOTE: Runs git clean -xdf and removes files not in the repo.
$ python setup.py sdist upload --sign --identity='Celery Security Team'
$ python setup.py bdist_wheel upload --sign --identity='Celery Security Team'
```

If this is a new release series then you also need to do the following:

- **Go to the Read The Docs management interface at:** https://readthedocs.org/projects/celery/?fromdocs= celery

- Enter "Edit project"

    Change default branch to the branch of this series, for example, use the `2.4` branch for the 2.4 series.

- Also add the previous version under the "versions" tab.

3.6 Community Resources

This is a list of external blog posts, tutorials, and slides related to Celery. If you have a link that's missing from this list, please contact the mailing-list or submit a patch.

- *Resources*
  - *Who's using Celery*
  - *Wiki*
  - *Celery questions on Stack Overflow*
  - *Mailing-list Archive: celery-users*
- *News*

3.6.1 Resources

Who's using Celery

https://github.com/celery/celery/wiki#companieswebsites-using-celery

Wiki

https://github.com/celery/celery/wiki

Celery questions on Stack Overflow

https://stackoverflow.com/search?q=celery&tab=newest

Mailing-list Archive: celery-users

http://blog.gmane.org/gmane.comp.python.amqp.celery.user

3.6.2 News

This section has moved to the Celery homepage: http://celeryproject.org/community/

3.7 Tutorials

Release 4.4

Date Mar 02, 2020

3.7.1 Task Cookbook

- *Ensuring a task is only executed one at a time*

Ensuring a task is only executed one at a time

You can accomplish this by using a lock.

In this example we'll be using the cache framework to set a lock that's accessible for all workers.

It's part of an imaginary RSS feed importer called *djangofeeds*. The task takes a feed URL as a single argument, and imports that feed into a Django model called *Feed*. We ensure that it's not possible for two or more workers to import the same feed at the same time by setting a cache key consisting of the MD5 check-sum of the feed URL.

The cache key expires after some time in case something unexpected happens, and something always will. . .

For this reason your tasks run-time shouldn't exceed the timeout.

Note: In order for this to work correctly you need to be using a cache backend where the `.add` operation is atomic. `memcached` is known to work well for this purpose.

```python
from celery import task
from celery.five import monotonic
from celery.utils.log import get_task_logger
from contextlib import contextmanager
from django.core.cache import cache
from hashlib import md5
from djangofeeds.models import Feed

logger = get_task_logger(__name__)

LOCK_EXPIRE = 60 * 10  # Lock expires in 10 minutes


@contextmanager
def memcache_lock(lock_id, oid):
    timeout_at = monotonic() + LOCK_EXPIRE - 3
    # cache.add fails if the key already exists
    status = cache.add(lock_id, oid, LOCK_EXPIRE)
    try:
        yield status
    finally:
        # memcache delete is very slow, but we have to use it to take
        # advantage of using add() for atomic locking
        if monotonic() < timeout_at and status:
            # don't release the lock if we exceeded the timeout
            # to lessen the chance of releasing an expired lock
            # owned by someone else
            # also don't release the lock if we didn't acquire it
            cache.delete(lock_id)


@task(bind=True)
def import_feed(self, feed_url):
    # The cache key consists of the task name and the MD5 digest
```

```
# of the feed URL.
feed_url_hexdigest = md5(feed_url).hexdigest()
lock_id = '{0}-lock-{1}'.format(self.name, feed_url_hexdigest)
logger.debug('Importing feed: %s', feed_url)
with memcache_lock(lock_id, self.app.oid) as acquired:
    if acquired:
        return Feed.objects.import_feed(feed_url).url
logger.debug(
    'Feed %s is already being imported by another worker', feed_url)
```

3.8 Frequently Asked Questions

3.8.1 General

What kinds of things should I use Celery for?

Answer: Queue everything and delight everyone is a good article describing why you'd use a queue in a web context.

These are some common use cases:

- Running something in the background. For example, to finish the web request as soon as possible, then update the users page incrementally. This gives the user the impression of good performance and "snappiness", even though the real work might actually take some time.

- Running something after the web request has finished.

- Making sure something is done, by executing it asynchronously and using retries.

- Scheduling periodic work.

And to some degree:

- Distributed computing.

- Parallel execution.

3.8.2 Misconceptions

Does Celery really consist of 50.000 lines of code?

Answer: No, this and similarly large numbers have been reported at various locations.

The numbers as of this writing are:

- core: 7,141 lines of code.

- tests: 14,209 lines.

- backends, contrib, compat utilities: 9,032 lines.

Lines of code isn't a useful metric, so even if Celery did consist of 50k lines of code you wouldn't be able to draw any conclusions from such a number.

Does Celery have many dependencies?

A common criticism is that Celery uses too many dependencies. The rationale behind such a fear is hard to imagine, especially considering code reuse as the established way to combat complexity in modern software development, and that the cost of adding dependencies is very low now that package managers like pip and PyPI makes the hassle of installing and maintaining dependencies a thing of the past.

Celery has replaced several dependencies along the way, and the current list of dependencies are:

celery

- kombu

Kombu is part of the Celery ecosystem and is the library used to send and receive messages. It's also the library that enables us to support many different message brokers. It's also used by the OpenStack project, and many others, validating the choice to separate it from the Celery code-base.

- billiard

Billiard is a fork of the Python multiprocessing module containing many performance and stability improvements. It's an eventual goal that these improvements will be merged back into Python one day.

It's also used for compatibility with older Python versions that don't come with the multiprocessing module.

- pytz

The pytz module provides timezone definitions and related tools.

kombu

Kombu depends on the following packages:

- amqp

The underlying pure-Python amqp client implementation. AMQP being the default broker this is a natural dependency.

Note: To handle the dependencies for popular configuration choices Celery defines a number of "bundle" packages, see *Bundles*.

Is Celery heavy-weight?

Celery poses very little overhead both in memory footprint and performance.

But please note that the default configuration isn't optimized for time nor space, see the *Optimizing* guide for more information.

Is Celery dependent on pickle?

Answer: No, Celery can support any serialization scheme.

We have built-in support for JSON, YAML, Pickle, and msgpack. Every task is associated with a content type, so you can even send one task using pickle, another using JSON.

The default serialization support used to be pickle, but since 4.0 the default is now JSON. If you require sending complex Python objects as task arguments, you can use pickle as the serialization format, but see notes in *Serializers*.

If you need to communicate with other languages you should use a serialization format suited to that task, which pretty much means any serializer that's not pickle.

You can set a global default serializer, the default serializer for a particular Task, or even what serializer to use when sending a single task instance.

Is Celery for Django only?

Answer: No, you can use Celery with any framework, web or otherwise.

Do I have to use AMQP/RabbitMQ?

Answer: No, although using RabbitMQ is recommended you can also use Redis, SQS, or Qpid.

See *Brokers* for more information.

Redis as a broker won't perform as well as an AMQP broker, but the combination RabbitMQ as broker and Redis as a result store is commonly used. If you have strict reliability requirements you're encouraged to use RabbitMQ or

another AMQP broker. Some transports also use polling, so they're likely to consume more resources. However, if you for some reason aren't able to use AMQP, feel free to use these alternatives. They will probably work fine for most use cases, and note that the above points are not specific to Celery; If using Redis/database as a queue worked fine for you before, it probably will now. You can always upgrade later if you need to.

Is Celery multilingual?

Answer: Yes.

`worker` is an implementation of Celery in Python. If the language has an AMQP client, there shouldn't be much work to create a worker in your language. A Celery worker is just a program connecting to the broker to process messages.

Also, there's another way to be language-independent, and that's to use REST tasks, instead of your tasks being functions, they're URLs. With this information you can even create simple web servers that enable preloading of code. Simply expose an endpoint that performs an operation, and create a task that just performs an HTTP request to that endpoint.

You can also use Flower's REST API to invoke tasks.

3.8.3 Troubleshooting

MySQL is throwing deadlock errors, what can I do?

Answer: MySQL has default isolation level set to *REPEATABLE-READ*, if you don't really need that, set it to *READ-COMMITTED*. You can do that by adding the following to your `my.cnf`:

```
[mysqld]
transaction-isolation = READ-COMMITTED
```

For more information about InnoDB's transaction model see MySQL - The InnoDB Transaction Model and Locking in the MySQL user manual.

(Thanks to Honza Kral and Anton Tsigularov for this solution)

The worker isn't doing anything, just hanging

Answer: See *MySQL is throwing deadlock errors, what can I do?*, or *Why is Task.delay/apply*/the worker just hanging?*.

Task results aren't reliably returning

Answer: If you're using the database backend for results, and in particular using MySQL, see *MySQL is throwing deadlock errors, what can I do?*.

Why is Task.delay/apply*/the worker just hanging?

Answer: There's a bug in some AMQP clients that'll make it hang if it's not able to authenticate the current user, the password doesn't match or the user doesn't have access to the virtual host specified. Be sure to check your broker logs (for RabbitMQ that's `/var/log/rabbitmq/rabbit.log` on most systems), it usually contains a message describing the reason.

Does it work on FreeBSD?

Answer: Depends;

When using the RabbitMQ (AMQP) and Redis transports it should work out of the box.

For other transports the compatibility prefork pool is used and requires a working POSIX semaphore implementation, this is enabled in FreeBSD by default since FreeBSD 8.x. For older version of FreeBSD, you have to enable POSIX semaphores in the kernel and manually recompile billiard.

Luckily, Viktor Petersson has written a tutorial to get you started with Celery on FreeBSD here: http://www.playingwithwire.com/2009/10/how-to-get-celeryd-to-work-on-freebsd/

I'm having *IntegrityError: Duplicate Key* errors. Why?

Answer: See *MySQL is throwing deadlock errors, what can I do?*. Thanks to @@howsthedotcom.

Why aren't my tasks processed?

Answer: With RabbitMQ you can see how many consumers are currently receiving tasks by running the following command:

```
$ rabbitmqctl list_queues -p <myvhost> name messages consumers
Listing queues ...
celery     2891    2
```

This shows that there's 2891 messages waiting to be processed in the task queue, and there are two consumers processing them.

One reason that the queue is never emptied could be that you have a stale worker process taking the messages hostage. This could happen if the worker wasn't properly shut down.

When a message is received by a worker the broker waits for it to be acknowledged before marking the message as processed. The broker won't re-send that message to another consumer until the consumer is shut down properly.

If you hit this problem you have to kill all workers manually and restart them:

```
$ pkill 'celery worker'

$ # - If you don't have pkill use:
$ # ps auxww | awk '/celery worker/ {print $2}' | xargs kill
```

You may have to wait a while until all workers have finished executing tasks. If it's still hanging after a long time you can kill them by force with:

```
$ pkill -9 'celery worker'

$ # - If you don't have pkill use:
$ # ps auxww | awk '/celery worker/ {print $2}' | xargs kill -9
```

Why won't my Task run?

Answer: There might be syntax errors preventing the tasks module being imported.

You can find out if Celery is able to run the task by executing the task manually:

```
>>> from myapp.tasks import MyPeriodicTask
>>> MyPeriodicTask.delay()
```

Watch the workers log file to see if it's able to find the task, or if some other error is happening.

Why won't my periodic task run?

Answer: See *Why won't my Task run?*.

How do I purge all waiting tasks?

Answer: You can use the `celery purge` command to purge all configured task queues:

```
$ celery -A proj purge
```

or programmatically:

```
>>> from proj.celery import app
>>> app.control.purge()
1753
```

If you only want to purge messages from a specific queue you have to use the AMQP API or the **celery amqp** utility:

```
$ celery -A proj amqp queue.purge <queue name>
```

The number 1753 is the number of messages deleted.

You can also start the worker with the `--purge` option enabled to purge messages when the worker starts.

I've purged messages, but there are still messages left in the queue?

Answer: Tasks are acknowledged (removed from the queue) as soon as they're actually executed. After the worker has received a task, it will take some time until it's actually executed, especially if there are a lot of tasks already waiting for execution. Messages that aren't acknowledged are held on to by the worker until it closes the connection to the broker (AMQP server). When that connection is closed (e.g., because the worker was stopped) the tasks will be re-sent by the broker to the next available worker (or the same worker when it has been restarted), so to properly purge the queue of waiting tasks you have to stop all the workers, and then purge the tasks using `celery.control.purge()`.

3.8.4 Results

How do I get the result of a task if I have the ID that points there?

Answer: Use *task.AsyncResult*:

```
>>> result = my_task.AsyncResult(task_id)
>>> result.get()
```

This will give you a *AsyncResult* instance using the tasks current result backend.

If you need to specify a custom result backend, or you want to use the current application's default backend you can use *app.AsyncResult*:

```
>>> result = app.AsyncResult(task_id)
>>> result.get()
```

3.8.5 Security

Isn't using *pickle* a security concern?

Answer: Indeed, since Celery 4.0 the default serializer is now JSON to make sure people are choosing serializers consciously and aware of this concern.

It's essential that you protect against unauthorized access to your broker, databases and other services transmitting pickled data.

Note that this isn't just something you should be aware of with Celery, for example also Django uses pickle for its cache client.

For the task messages you can set the `task_serializer` setting to "json" or "yaml" instead of pickle.

Similarly for task results you can set `result_serializer`.

For more details of the formats used and the lookup order when checking what format to use for a task see *Serializers*

Can messages be encrypted?

Answer: Some AMQP brokers supports using SSL (including RabbitMQ). You can enable this using the `broker_use_ssl` setting.

It's also possible to add additional encryption and security to messages, if you have a need for this then you should contact the *Mailing list*.

Is it safe to run `celery worker` as root?

Answer: No!

We're not currently aware of any security issues, but it would be incredibly naive to assume that they don't exist, so running the Celery services (**celery worker**, **celery beat**, **celeryev**, etc) as an unprivileged user is recommended.

3.8.6 Brokers

Why is RabbitMQ crashing?

Answer: RabbitMQ will crash if it runs out of memory. This will be fixed in a future release of RabbitMQ. please refer to the RabbitMQ FAQ: https://www.rabbitmq.com/faq.html#node-runs-out-of-memory

---

Note: This is no longer the case, RabbitMQ versions 2.0 and above includes a new persister, that's tolerant to out of memory errors. RabbitMQ 2.1 or higher is recommended for Celery.

If you're still running an older version of RabbitMQ and experience crashes, then please upgrade!

Misconfiguration of Celery can eventually lead to a crash on older version of RabbitMQ. Even if it doesn't crash, this can still consume a lot of resources, so it's important that you're aware of the common pitfalls.

• Events.

Running *worker* with the *-E* option will send messages for events happening inside of the worker.

Events should only be enabled if you have an active monitor consuming them, or if you purge the event queue periodically.

• AMQP backend results.

When running with the AMQP result backend, every task result will be sent as a message. If you don't collect these results, they will build up and RabbitMQ will eventually run out of memory.

This result backend is now deprecated so you shouldn't be using it. Use either the RPC backend for rpc-style calls, or a persistent backend if you need multi-consumer access to results.

Results expire after 1 day by default. It may be a good idea to lower this value by configuring the *result_expires* setting.

If you don't use the results for a task, make sure you set the *ignore_result* option:

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


class MyTask(Task):
    ignore_result = True
```

Can I use Celery with ActiveMQ/STOMP?

Answer: No. It used to be supported by Carrot (our old messaging library) but isn't currently supported in Kombu (our new messaging library).

What features aren't supported when not using an AMQP broker?

This is an incomplete list of features not available when using the virtual transports:

• Remote control commands (supported only by Redis).

• Monitoring with events may not work in all virtual transports.

• **The *header* and *fanout* exchange types** (*fanout* is supported by Redis).

3.8.7 Tasks

How can I reuse the same connection when calling tasks?

Answer: See the *broker_pool_limit* setting. The connection pool is enabled by default since version 2.5.

`sudo` in a `subprocess` returns `None`

There's a **sudo** configuration option that makes it illegal for process without a tty to run **sudo**:

```
Defaults requiretty
```

If you have this configuration in your `/etc/sudoers` file then tasks won't be able to call **sudo** when the worker is running as a daemon. If you want to enable that, then you need to remove the line from `/etc/sudoers`.

See: http://timelordz.com/wiki/Apache_Sudo_Commands

Why do workers delete tasks from the queue if they're unable to process them?

Answer:

The worker rejects unknown tasks, messages with encoding errors and messages that don't contain the proper fields (as per the task message protocol).

If it didn't reject them they could be redelivered again and again, causing a loop.

Recent versions of RabbitMQ has the ability to configure a dead-letter queue for exchange, so that rejected messages is moved there.

Can I call a task by name?

Answer: Yes, use `app.send_task()`.

You can also call a task by name, from any language, using an AMQP client:

```
>>> app.send_task('tasks.add', args=[2, 2], kwargs={})
<AsyncResult: 373550e8-b9a0-4666-bc61-ace01fa4f91d>
```

To use `chain`, `chord` or `group` with tasks called by name, use the `Celery.signature()` method:

```
>>> chain(
...     app.signature('tasks.add', args=[2, 2], kwargs={}),
...     app.signature('tasks.add', args=[1, 1], kwargs={})
... ).apply_async()
<AsyncResult: e9d52312-c161-46f0-9013-2713e6df812d>
```

Can I get the task id of the current task?

Answer: Yes, the current id and more is available in the task request:

```
@app.task(bind=True)
def mytask(self):
    cache.set(self.request.id, "Running")
```

For more information see *Task Request*.

If you don't have a reference to the task instance you can use `app.current_task`:

```
>>> app.current_task.request.id
```

But note that this will be any task, be it one executed by the worker, or a task called directly by that task, or a task called eagerly.

To get the current task being worked on specifically, use `current_worker_task`:

```
>>> app.current_worker_task.request.id
```

Note: Both *current_task*, and *current_worker_task* can be None.

Can I specify a custom task_id?

Answer: Yes, use the *task_id* argument to Task.apply_async():

```
>>> task.apply_async(args, kwargs, task_id='...')
```

Can I use decorators with tasks?

Answer: Yes, but please see note in the sidebar at *Basics*.

Can I use natural task ids?

Answer: Yes, but make sure it's unique, as the behavior for two tasks existing with the same id is undefined.

The world will probably not explode, but they can definitely overwrite each others results.

Can I run a task once another task has finished?

Answer: Yes, you can safely launch a task inside a task.

A common pattern is to add callbacks to tasks:

```python
from celery.utils.log import get_task_logger

logger = get_task_logger(__name__)

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

@app.task(ignore_result=True)
def log_result(result):
    logger.info("log_result got: %r", result)
```

Invocation:

```
>>> (add.s(2, 2) | log_result.s()).delay()
```

See *Canvas: Designing Work-flows* for more information.

Can I cancel the execution of a task?

Answer: Yes, Use *result.revoke()*:

```
>>> result = add.apply_async(args=[2, 2], countdown=120)
>>> result.revoke()
```

or if you only have the task id:

```
>>> from proj.celery import app
>>> app.control.revoke(task_id)
```

The latter also support passing a list of task-ids as argument.

Why aren't my remote control commands received by all workers?

Answer: To receive broadcast remote control commands, every worker node creates a unique queue name, based on the nodename of the worker.

If you have more than one worker with the same host name, the control commands will be received in round-robin between them.

To work around this you can explicitly set the nodename for every worker using the `-n` argument to `worker`:

```
$ celery -A proj worker -n worker1@%h
$ celery -A proj worker -n worker2@%h
```

where `%h` expands into the current hostname.

Can I send some tasks to only some servers?

Answer: Yes, you can route tasks to one or more workers, using different message routing topologies, and a worker instance can bind to multiple queues.

See *Routing Tasks* for more information.

Can I disable prefetching of tasks?

Answer: Maybe! The AMQP term "prefetch" is confusing, as it's only used to describe the task prefetching *limit*. There's no actual prefetching involved.

Disabling the prefetch limits is possible, but that means the worker will consume as many tasks as it can, as fast as possible.

A discussion on prefetch limits, and configuration settings for a worker that only reserves one task at a time is found here: *Prefetch Limits*.

Can I change the interval of a periodic task at runtime?

Answer: Yes, you can use the Django database scheduler, or you can create a new schedule subclass and override `is_due()`:

```python
from celery.schedules import schedule

class my_schedule(schedule):

    def is_due(self, last_run_at):
        return run_now, next_time_to_check
```

Does Celery support task priorities?

Answer: Yes, RabbitMQ supports priorities since version 3.5.0, and the Redis transport emulates priority support.

You can also prioritize work by routing high priority tasks to different workers. In the real world this usually works better than per message priorities. You can use this in combination with rate limiting, and per message priorities to achieve a responsive system.

Should I use retry or acks_late?

Answer: Depends. It's not necessarily one or the other, you may want to use both.

Task.retry is used to retry tasks, notably for expected errors that is catch-able with the `try` block. The AMQP transaction isn't used for these errors: **if the task raises an exception it's still acknowledged!**

The *acks_late* setting would be used when you need the task to be executed again if the worker (for some reason) crashes mid-execution. It's important to note that the worker isn't known to crash, and if it does it's usually an unrecoverable error that requires human intervention (bug in the worker, or task code).

In an ideal world you could safely retry any task that's failed, but this is rarely the case. Imagine the following task:

```
@app.task
def process_upload(filename, tmpfile):
    # Increment a file count stored in a database
    increment_file_counter()
    add_file_metadata_to_db(filename, tmpfile)
    copy_file_to_destination(filename, tmpfile)
```

If this crashed in the middle of copying the file to its destination the world would contain incomplete state. This isn't a critical scenario of course, but you can probably imagine something far more sinister. So for ease of programming we have less reliability; It's a good default, users who require it and know what they are doing can still enable acks_late (and in the future hopefully use manual acknowledgment).

In addition *Task.retry* has features not available in AMQP transactions: delay between retries, max retries, etc.

So use retry for Python errors, and if your task is idempotent combine that with *acks_late* if that level of reliability is required.

Can I schedule tasks to execute at a specific time?

Answer: Yes. You can use the *eta* argument of `Task.apply_async()`.

See also *Periodic Tasks*.

Can I safely shut down the worker?

Answer: Yes, use the `TERM` signal.

This will tell the worker to finish all currently executing jobs and shut down as soon as possible. No tasks should be lost even with experimental transports as long as the shutdown completes.

You should never stop `worker` with the `KILL` signal (`kill -9`), unless you've tried `TERM` a few times and waited a few minutes to let it get a chance to shut down.

Also make sure you kill the main worker process only, not any of its child processes. You can direct a kill signal to a specific child process if you know the process is currently executing a task the worker shutdown is depending on, but this also means that a `WorkerLostError` state will be set for the task so the task won't run again.

Identifying the type of process is easier if you have installed the setproctitle module:

```
$ pip install setproctitle
```

With this library installed you'll be able to see the type of process in **ps** listings, but the worker must be restarted for this to take effect.

See also:

Stopping the worker

Can I run the worker in the background on [platform]?

Answer: Yes, please see *Daemonization*.

3.8.8 Django

What purpose does the database tables created by `django-celery-beat` have?

When the database-backed schedule is used the periodic task schedule is taken from the `PeriodicTask` model, there are also several other helper tables (`IntervalSchedule`, `CrontabSchedule`, `PeriodicTasks`).

What purpose does the database tables created by `django-celery-results` have?

The Django database result backend extension requires two extra models: `TaskResult` and `GroupResult`.

3.8.9 Windows

Does Celery support Windows?

Answer: No.

Since Celery 4.x, Windows is no longer supported due to lack of resources.

But it may still work and we are happy to accept patches.

3.9 Change history

This document contains change notes for bugfix & new features in the 4.x series, please see *What's new in Celery 4.4 (Cliffs)* for an overview of what's new in Celery 4.4.

3.9.1 4.4.0

release-date 2019-12-16 9.45 A.M UTC+6:00

release-by Asif Saif Uddin

- This version is officially supported on CPython 2.7, 3.5, 3.6, 3.7 & 3.8 and is also supported on PyPy2 & PyPy3.
- Kombu 4.6.7
- Task class definitions can have retry attributes (#5869)

3.9.2 4.4.0rc5

release-date 2019-12-07 21.05 A.M UTC+6:00

release-by Asif Saif Uddin

- Kombu 4.6.7

- Events bootstep disabled if no events (#5807)

- SQS - Reject on failure (#5843)

- Add a concurrency model with ThreadPoolExecutor (#5099)

- Add auto expiry for DynamoDB backend (#5805)

- Store extending result in all backends (#5661)

- Fix a race condition when publishing a very large chord header (#5850)

- Improve docs and test matrix

3.9.3 4.4.0rc4

release-date 2019-11-11 00.45 A.M UTC+6:00

release-by Asif Saif Uddin

- Kombu 4.6.6

- Py-AMQP 2.5.2

- Python 3.8

- Numerious bug fixes

- PyPy 7.2

3.9.4 4.4.0rc3

release-date 2019-08-14 23.00 P.M UTC+6:00

release-by Asif Saif Uddin

- Kombu 4.6.4

- Billiard 3.6.1

- Py-AMQP 2.5.1

- Avoid serializing datetime (#5606)

- Fix: (group() | group()) not equals single group (#5574)

- Revert "Broker connection uses the heartbeat setting from app config.

- Additional file descriptor safety checks.

- fixed call for null args (#5631)

- Added generic path for cache backend.

- Fix Nested group(chain(group)) fails (#5638)

- Use self.run() when overriding __call__ (#5652)

- Fix termination of asyncloop (#5671)

- Fix migrate task to work with both v1 and v2 of the message protocol.

- Updating task_routes config during runtime now have effect.

3.9.5 4.4.0rc2

release-date 2019-06-15 4:00 A.M UTC+6:00

release-by Asif Saif Uddin

- Many bugs and regressions fixed.

- Kombu 4.6.3

3.9.6 4.4.0rc1

release-date 2019-06-06 1:00 P.M UTC+6:00

release-by Asif Saif Uddin

- Python 3.4 drop

- Kombu 4.6.1

- Replace deprecated PyMongo methods usage (#5443)

- Pass task request when calling update_state (#5474)

- Fix bug in remaining time calculation in case of DST time change (#5411)

- Fix missing task name when requesting extended result (#5439)

- Fix *collections* import issue on Python 2.7 (#5428)

- handle *AttributeError* in base backend exception deserializer (#5435)

- Make *AsynPool*'s *proc_alive_timeout* configurable (#5476)

- AMQP Support for extended result (#5495)

- Fix SQL Alchemy results backend to work with extended result (#5498)

- Fix restoring of exceptions with required param (#5500)

- Django: Re-raise exception if *ImportError* not caused by missing tasks module (#5211)

- Django: fixed a regression putting DB connections in invalid state when *CONN_MAX_AGE != 0* (#5515)

- Fixed *OSError* leading to lost connection to broker (#4457)

- Fixed an issue with inspect API unable get details of Request

- Fix mogodb backend authentication (#5527)

- Change column type for Extended Task Meta args/kwargs to LargeBinary

- Handle http_auth in Elasticsearch backend results (#5545)

- Fix task serializer being ignored with *task_always_eager=True* (#5549)

- Fix *task.replace* to work in *.apply() as well as '.apply_async()* (#5540)

- Fix sending of *worker_process_init* signal for solo worker (#5562)

- Fix exception message upacking (#5565)

- Add delay parameter function to beat_schedule (#5558)

- Multiple documentation updates

3.9.7 4.3.0

release-date 2019-03-31 7:00 P.M UTC+3:00

release-by Omer Katz

- Added support for broadcasting using a regular expression pattern or a glob pattern to multiple Pidboxes.

  This allows you to inspect or ping multiple workers at once.

  Contributed by **Dmitry Malinovsky** & **Jason Held**

- Added support for PEP 420 namespace packages.

  This allows you to load tasks from namespace packages.

  Contributed by **Colin Watson**

- Added `acks_on_failure_or_timeout` as a setting instead of a task only option.

  This was missing from the original PR but now added for completeness.

  Contributed by **Omer Katz**

- Added the `task_received` signal.

  Contributed by **Omer Katz**

- Fixed a crash of our CLI that occurred for everyone using Python < 3.6.

  The crash was introduced in acd6025 by using the `ModuleNotFoundError` exception which was introduced in Python 3.6.

  Contributed by **Omer Katz**

- Fixed a crash that occurred when using the Redis result backend while the `result_expires` is set to None.

  Contributed by **Toni Ruža** & **Omer Katz**

- Added support the DNS seedlist connection format for the MongoDB result backend.

  This requires the *dnspython* package which will be installed by default when installing the dependencies for the MongoDB result backend.

  Contributed by **George Psarakis**

- Bump the minimum eventlet version to 0.24.1.

  Contributed by **George Psarakis**

- Replace the *msgpack-python* package with *msgpack*.

  We're no longer using the deprecated package. See our important notes for this release for further details on how to upgrade.

  Contributed by **Daniel Hahler**

- Allow scheduling error handlers which are not registered tasks in the current worker.

  These kind of error handlers are now possible:

```python
from celery import Signature
Signature(
  'bar', args=['foo'],
  link_error=Signature('msg.err', queue='msg')
).apply_async()
```

- Additional fixes and enhancements to the SSL support of the Redis broker and result backend.

  Contributed by **Jeremy Cohen**

Code Cleanups, Test Coverage & CI Improvements by:

- **Omer Katz**

- **Florian Chardin**

Documentation Fixes by:

- **Omer Katz**

- **Samuel Huang**

- **Amir Hossein Saeid Mehr**

- **Dmytro Litvinov**

3.9.8 4.3.0 RC2

release-date 2019-03-03 9:30 P.M UTC+2:00

release-by Omer Katz

- **Filesystem Backend**: Added meaningful error messages for filesystem backend.

  Contributed by **Lars Rinn**

- **New Result Backend**: Added the ArangoDB backend.

  Contributed by **Dilip Vamsi Moturi**

- **Django**: Prepend current working directory instead of appending so that the project directory will have precedence over system modules as expected.

  Contributed by **Antonin Delpeuch**

- Bump minimum py-redis version to 3.2.0.

  Due to multiple bugs in earlier versions of py-redis that were causing issues for Celery, we were forced to bump the minimum required version to 3.2.0.

  Contributed by **Omer Katz**

- **Dependencies**: Bump minimum required version of Kombu to 4.4

  Contributed by **Omer Katz**

3.9.9 4.3.0 RC1

release-date 2019-02-20 5:00 PM IST

release-by Omer Katz

- **Canvas**: `celery.chain.apply()` does not ignore keyword arguments anymore when applying the chain.

  Contributed by **Korijn van Golen**

- **Result Set**: Don't attempt to cache results in a `celery.result.ResultSet`.

  During a join, the results cache was populated using `celery.result.ResultSet.get()`, if one of the results contains an exception, joining unexpectedly failed.

  The results cache is now removed.

  Contributed by **Derek Harland**

- **Application**: `celery.Celery.autodiscover_tasks()` now attempts to import the package itself when the *related_name* keyword argument is *None*.

  Contributed by **Alex Ioannidis**

- **Windows Support**: On Windows 10, stale PID files prevented celery beat to run. We now remove them when a `SystemExit` is raised.

  Contributed by **:github_user:'na387'**

- **Task**: Added the new `task_acks_on_failure_or_timeout` setting.

  Acknowledging SQS messages on failure or timing out makes it impossible to use dead letter queues.

  We introduce the new option acks_on_failure_or_timeout, to ensure we can totally fallback on native SQS message lifecycle, using redeliveries for retries (in case of slow processing or failure) and transitions to dead letter queue after defined number of times.

  Contributed by **Mario Kostelac**

- **RabbitMQ Broker**: Adjust HA headers to work on RabbitMQ 3.x.

  This change also means we're ending official support for RabbitMQ 2.x.

  Contributed by **Asif Saif Uddin**

- **Command Line**: Improve **celery update** error handling.

  Contributed by **Federico Bond**

- **Canvas**: Support chords with `task_always_eager` set to *True*.

  Contributed by **Axel Haustant**

- **Result Backend**: Optionally store task properties in result backend.

  Setting the `result_extended` configuration option to *True* enables storing additional task properties in the result backend.

  Contributed by **John Arnold**

- **Couchbase Result Backend**: Allow the Couchbase result backend to automatically detect the serialization format.

  Contributed by **Douglas Rohde**

- **New Result Backend**: Added the Azure Block Blob Storage result backend.

  The backend is implemented on top of the azure-storage library which uses Azure Blob Storage for a scalable low-cost PaaS backend.

  The backend was load tested via a simple nginx/gunicorn/sanic app hosted on a DS4 virtual machine (4 vCores, 16 GB RAM) and was able to handle 600+ concurrent users at ~170 RPS.

The commit also contains a live end-to-end test to facilitate verification of the backend functionality. The test is activated by setting the *AZUREBLOCKBLOB_URL* environment variable to *azureblock-blob://{ConnectionString}* where the value for *ConnectionString* can be found in the *Access Keys* pane of a Storage Account resources in the Azure Portal.

Contributed by **Clemens Wolff**

- **Task**: `celery.app.task.update_state()` now accepts keyword arguments.

  This allows passing extra fields to the result backend. These fields are unused by default but custom result backends can use them to determine how to store results.

  Contributed by **Christopher Dignam**

- Gracefully handle consumer `kombu.exceptions.DecodeError`.

  When using the v2 protocol the worker no longer crashes when the consumer encounters an error while decoding a message.

  Contributed by **Steven Sklar**

- **Deployment**: Fix init.d service stop.

  Contributed by **Marcus McHale**

- **Django**: Drop support for Django < 1.11.

  Contributed by **Asif Saif Uddin**

- **Django**: Remove old djcelery loader.

  Contributed by **Asif Saif Uddin**

- **Result Backend**: *`celery.worker.request.Request`* now passes *`celery.app.task.Context`* to the backend's store_result functions.

  Since the class currently passes *self* to these functions, revoking a task resulted in corrupted task result data when django-celery-results was used.

  Contributed by **Kiyohiro Yamaguchi**

- **Worker**: Retry if the heartbeat connection dies.

  Previously, we keep trying to write to the broken connection. This results in a memory leak because the event dispatcher will keep appending the message to the outbound buffer.

  Contributed by **Raf Geens**

- **Celery Beat**: Handle microseconds when scheduling.

  Contributed by **K Davis**

- **Asynpool**: Fixed deadlock when closing socket.

  Upon attempting to close a socket, `celery.concurrency.asynpool.AsynPool` only removed the queue writer from the hub but did not remove the reader. This led to a deadlock on the file descriptor and eventually the worker stopped accepting new tasks.

  We now close both the reader and the writer file descriptors in a single loop iteration which prevents the deadlock.

  Contributed by **Joshua Engelman**

- **Celery Beat**: Correctly consider timezone when calculating timestamp.

  Contributed by **:github_user:'yywing'**

- **Celery Beat**: `celery.beat.Scheduler.schedules_equal()` can now handle either arguments being a *None* value.

  Contributed by **:github_user:' ratson'**

- **Documentation/Sphinx**: Fixed Sphinx support for shared_task decorated functions.

  Contributed by **Jon Banafato**

- **New Result Backend**: Added the CosmosDB result backend.

  This change adds a new results backend. The backend is implemented on top of the pydocumentdb library which uses Azure CosmosDB for a scalable, globally replicated, high-performance, low-latency and high-throughput PaaS backend.

  Contributed by **Clemens Wolff**

- **Application**: Added configuration options to allow separate multiple apps to run on a single RabbitMQ vhost.

  The newly added `event_exchange` and `control_exchange` configuration options allow users to use separate Pidbox exchange and a separate events exchange.

  This allow different Celery applications to run separately on the same vhost.

  Contributed by **Artem Vasilyev**

- **Result Backend**: Forget parent result metadata when forgetting a result.

  Contributed by **:github_user:'tothegump'**

- **Task** Store task arguments inside `celery.exceptions.MaxRetriesExceededError`.

  Contributed by **Anthony Ruhier**

- **Result Backend**: Added the `result_accept_content` setting.

  This feature allows to configure different accepted content for the result backend.

  A special serializer (*auth*) is used for signed messaging, however the result_serializer remains in json, because we don't want encrypted content in our result backend.

  To accept unsigned content from the result backend, we introduced this new configuration option to specify the accepted content from the backend.

  Contributed by **Benjamin Pereto**

- **Canvas**: Fixed error callback processing for class based tasks.

  Contributed by **Victor Mireyev**

- **New Result Backend**: Added the S3 result backend.

  Contributed by **Florian Chardin**

- **Task**: Added support for Cythonized Celery tasks.

  Contributed by **Andrey Skabelin**

- **Riak Result Backend**: Warn Riak backend users for possible Python 3.7 incompatibilities.

  Contributed by **George Psarakis**

- **Python Runtime**: Added Python 3.7 support.

  Contributed by **Omer Katz** & **Asif Saif Uddin**

- **Auth Serializer**: Revamped the auth serializer.

  The auth serializer received a complete overhaul. It was previously horribly broken.

We now depend on cryptography instead of pyOpenSSL for this serializer.

Contributed by **Benjamin Pereto**

- **Command Line**: `celery report` now reports kernel version along with other platform details.

  Contributed by **Omer Katz**

- **Canvas**: Fixed chords with chains which include sub chords in a group.

  Celery now correctly executes the last task in these types of canvases:

```
c = chord(
  group([
      chain(
          dummy.si(),
          chord(
              group([dummy.si(), dummy.si()]),
              dummy.si(),
          ),
      ),
      chain(
          dummy.si(),
          chord(
              group([dummy.si(), dummy.si()]),
              dummy.si(),
          ),
      ),
  ]),
  dummy.si()
)

c.delay().get()
```

  Contributed by **Maximilien Cuony**

- **Canvas**: Complex canvases with error callbacks no longer raises an `AttributeError`.

  Very complex canvases such as this no longer raise an `AttributeError` which prevents constructing them.

  We do not know why this bug occurs yet.

  Contributed by **Manuel Vázquez Acosta**

- **Command Line**: Added proper error messages in cases where app cannot be loaded.

  Previously, celery crashed with an exception.

  We now print a proper error message.

  Contributed by **Omer Katz**

- **Task**: Added the `task_default_priority` setting.

  You can now set the default priority of a task using the `task_default_priority` setting. The setting's value will be used if no priority is provided for a specific task.

  Contributed by **:github_user:'madprogrammer'**

- **Dependencies**: Bump minimum required version of Kombu to 4.3 and Billiard to 3.6.

  Contributed by **Asif Saif Uddin**

- **Result Backend**: Fix memory leak.

We reintroduced weak references to bound methods for AsyncResult callback promises, after adding full weakref support for Python 2 in vine. More details can be found in celery/celery#4839.

Contributed by **George Psarakis** and **:github_user:'monsterxx03'**.

- **Task Execution**: Fixed roundtrip serialization for eager tasks.

  When doing the roundtrip serialization for eager tasks, the task serializer will always be JSON unless the *serializer* argument is present in the call to `celery.app.task.Task.apply_async()`. If the serializer argument is present but is *'pickle'*, an exception will be raised as pickle-serialized objects cannot be deserialized without specifying to *serialization.loads* what content types should be accepted. The Producer's *serializer* seems to be set to *None*, causing the default to JSON serialization.

  We now continue to use (in order) the *serializer* argument to `celery.app.task.Task.apply_async()`, if present, or the *Producer*'s serializer if not *None*. If the *Producer*'s serializer is *None*, it will use the Celery app's *task_serializer* configuration entry as the serializer.

  Contributed by **Brett Jackson**

- **Redis Result Backend**: The `celery.backends.redis.ResultConsumer` class no longer assumes `celery.backends.redis.ResultConsumer.start()` to be called before `celery.backends.redis.ResultConsumer.drain_events()`.

  This fixes a race condition when using the Gevent workers pool.

  Contributed by **Noam Kush**

- **Task**: Added the `task_inherit_parent_priority` setting.

  Setting the `task_inherit_parent_priority` configuration option to *True* will make Celery tasks inherit the priority of the previous task linked to it.

  Examples:

```
c = celery.chain(
  add.s(2), # priority=None
  add.s(3).set(priority=5), # priority=5
  add.s(4), # priority=5
  add.s(5).set(priority=3), # priority=3
  add.s(6), # priority=3
)
```

```
@app.task(bind=True)
def child_task(self):
  pass

@app.task(bind=True)
def parent_task(self):
  child_task.delay()

# child_task will also have priority=5
parent_task.apply_async(args=[], priority=5)
```

  Contributed by **:github_user:'madprogrammer'**

- **Canvas**: Added the `result_chord_join_timeout` setting.

  Previously, `celery.result.GroupResult.join()` had a fixed timeout of 3 seconds.

  The `result_chord_join_timeout` setting now allows you to change it.

  Contributed by **:github_user:'srafehi'**

Code Cleanups, Test Coverage & CI Improvements by:

- **Jon Dufresne**
- **Asif Saif Uddin**
- **Omer Katz**
- **Brett Jackson**
- **Bruno Alla**
- **:github_user:'tothegump'**
- **Bojan Jovanovic**
- **Florian Chardin**
- **:github_user:'walterqian'**
- **Fabian Becker**
- **Lars Rinn**
- **:github_user:'madprogrammer'**
- **Ciaran Courtney**

Documentation Fixes by:

- **Lewis M. Kabui**
- **Dash Winterson**
- **Shanavas M**
- **Brett Randall**
- **Przemysław Suliga**
- **Joshua Schmid**
- **Asif Saif Uddin**
- **Xiaodong**
- **Vikas Prasad**
- **Jamie Alessio**
- **Lars Kruse**
- **Guilherme Caminha**
- **Andrea Rabbaglietti**
- **Itay Bittan**
- **Noah Hall**
- **Peng Weikang**
- **Mariatta Wijaya**
- **Ed Morley**
- **Paweł Adamczak**
- **:github_user:'CoffeeExpress'**
- **:github_user:'aviadatsnyk'**

- **Brian Schrader**

- **Josue Balandrano Coronel**

- **Tom Clancy**

- **Sebastian Wojciechowski**

- **Meysam Azad**

- **Willem Thiart**

- **Charles Chan**

- **Omer Katz**

- **Milind Shakya**

3.10 What's new in Celery 4.4 (Cliffs)

> **Author** Asif Saif Uddin (`auvipy at gmail.com`)

> **Change history**
>
> What's new documents describe the changes in major versions, we also have a *Change history* that lists the changes in bugfix releases (0.0.x), while older series are archived under the *History* section.

Celery is a simple, flexible, and reliable distributed programming framework to process vast amounts of messages, while providing operations with the tools required to maintain a distributed system with python.

It's a task queue with focus on real-time processing, while also supporting task scheduling.

Celery has a large and diverse community of users and contributors, you should come join us *on IRC* or *our mailing-list*.

To read more about Celery you should go read the *introduction*.

While this version is backward compatible with previous versions it's important that you read the following section.

This version is officially supported on CPython 2.7, 3.5, 3.6, 3.7 & 3.8 and is also supported on PyPy2 & PyPy3.

> **Table of Contents**
>
> Make sure you read the important notes before upgrading to this version.

3.10.1 Preface

The 4.4.0 release continues to improve our efforts to provide you with the best task execution platform for Python.

This release has been codenamed Cliffs which is one of my favorite tracks.

This release focuses on mostly bug fixes and usability improvement for developers. Many long standing bugs, usability issues, documentation issues & minor ehancement issues were squashed which improve the overall developers experience.

Celery 4.4 is the first release to support Python 3.8 & pypy36-7.2.

As we now begin to work on Celery 5, the next generation of our task execution platform, at least another 4.x is expected before Celery 5 stable release & will get support for at least 1 years depending on community demand and support.

We have also focused on reducing contribution friction and updated the contributing tools.

— *Asif Saif Uddin*

Wall of Contributors

Note: This wall was automatically generated from git history, so sadly it doesn't not include the people who help with more important things like answering mailing-list questions.

3.10.2 Upgrading from Celery 4.3

Please read the important notes below as there are several breaking changes.

3.10.3 Important Notes

Supported Python Versions

The supported Python Versions are:

- CPython 2.7
- CPython 3.5
- CPython 3.6
- CPython 3.7
- CPython 3.8
- PyPy2.7 7.2 (`pypy2`)
- PyPy3.5 7.1 (`pypy3`)
- PyPy3.6 7.2 (`pypy3`)

Dropped support for Python 3.4

Celery now requires either Python 2.7 or Python 3.5 and above.

Python 3.4 has reached EOL in March 2019. In order to focus our efforts we have dropped support for Python 3.4 in this version.

If you still require to run Celery using Python 3.4 you can still use Celery 4.3. However we encourage you to upgrade to a supported Python version since no further security patches will be applied for Python 3.4.

Kombu

Starting from this release, the minimum required version is Kombu 4.6.6.

Billiard

Starting from this release, the minimum required version is Billiard 3.6.1.

Redis Message Broker

Due to multiple bugs in earlier versions of redis-py that were causing issues for Celery, we were forced to bump the minimum required version to 3.3.0.

Redis Result Backend

Due to multiple bugs in earlier versions of redis-py that were causing issues for Celery, we were forced to bump the minimum required version to 3.3.0.

DynamoDB Result Backend

The DynamoDB result backend has gained TTL support. As a result the minimum boto3 version was bumped to 1.9.178 which is the first version to support TTL for DynamoDB.

S3 Results Backend

To keep up with the current AWS API changes the minimum boto3 version was bumped to 1.9.125.

SQS Message Broker

To keep up with the current AWS API changes the minimum boto3 version was bumped to 1.9.125.

Configuration

CELERY_TASK_RESULT_EXPIRES has been replaced with *CELERY_RESULT_EXPIRES*.

3.10.4 News

Task Pools

Threaded Tasks Pool

We reintroduced a threaded task pool using *concurrent.futures.ThreadPoolExecutor*.

The previous threaded task pool was experimental. In addition it was based on the threadpool package which is obsolete.

You can use the new threaded task pool by setting `worker_pool` to 'threads' or by passing *–pool threads* to the *celery worker* command.

Result Backends

ElasticSearch Results Backend

HTTP Basic Authentication Support

You can now use HTTP Basic Authentication when using the ElasticSearch result backend by providing the username and the password in the URI.

Previously, they were ignored and only unauthenticated requests were issued.

MongoDB Results Backend

Support for Authentication Source and Authentication Method

You can now specify the authSource and authMethod for the MongoDB using the URI options. The following URI does just that:

```
mongodb://user:password@example.com/?authSource=the_database&authMechanism=SCRAM-SHA-
→256
```

Refer to the documentation for details about the various options.

Tasks

Task class definitions can now have retry attributes

You can now use *autoretry_for*, *retry_kwargs*, *retry_backoff*, *retry_backoff_max* and *retry_jitter* in class-based tasks:

```python
class BaseTaskWithRetry(Task):
    autoretry_for = (TypeError,)
    retry_kwargs = {'max_retries': 5}
    retry_backoff = True
    retry_backoff_max = 700
    retry_jitter = False
```

Canvas

Replacing Tasks Eagerly

You can now call *self.replace()* on tasks which are run eagerly. They will work exactly the same as tasks which are run asynchronously.

Chaining Groups

Chaining groups no longer result in a single group.

The following used to join the two groups into one. Now they correctly execute one after another:

```python
>>> result = group(add.si(1, 2), add.si(1, 2)) | group(tsum.s(), tsum.s()).delay()
>>> result.get()
[6, 6]
```

3.11 API Reference

Release 4.4

Date Mar 02, 2020

3.11.1 `celery` — Distributed processing

This module is the main entry-point for the Celery API. It includes commonly needed things for calling tasks, and creating Celery applications.

Celery	Celery application instance
group	group tasks together
chain	chain tasks together
chord	chords enable callbacks for groups
signature()	create a new task signature
Signature	object describing a task invocation
current_app	proxy to the current application instance
current_task	proxy to the currently executing task

`Celery` application objects

New in version 2.5.

class `celery.Celery`(*main=None*, *loader=None*, *backend=None*, *amqp=None*, *events=None*, *log=None*, *control=None*, *set_as_current=True*, *tasks=None*, *broker=None*, *include=None*, *changes=None*, *config_source=None*, *fixups=None*, *task_cls=None*, *autofinalize=True*, *namespace=None*, *strict_typing=True*, ***kwargs*)

> Celery application.
>
> > **Parameters main** (*str*) – Name of the main module if running as *__main__*. This is used as the prefix for auto-generated task names.
> >
> > **Keyword Arguments**
> >
> > - **broker** (*str*) – URL of the default broker used.
> >
> > - **backend** (*Union[str, Type[celery.backends.base.Backend]]*) – The result store backend class, or the name of the backend class to use.
> >
> >   Default is the value of the *result_backend* setting.
> >
> > - **autofinalize** (*bool*) – If set to False a `RuntimeError` will be raised if the task registry or tasks are used before the app is finalized.
> >
> > - **set_as_current** (*bool*) – Make this the global current app.
> >
> > - **include** (*List[str]*) – List of modules every worker should import.
> >
> > - **amqp** (*Union[str, Type[AMQP]]*) – AMQP object or class name.
> >
> > - **events** (*Union[str, Type[celery.app.events.Events]]*) – Events object or class name.
> >
> > - **log** (*Union[str, Type[Logging]]*) – Log object or class name.
> >
> > - **control** (*Union[str, Type[celery.app.control.Control]]*) – Control object or class name.
> >
> > - **tasks** (*Union[str, Type[TaskRegistry]]*) – A task registry, or the name of a registry class.
> >
> > - **fixups** (*List[str]*) – List of fix-up plug-ins (e.g., see `celery.fixups.django`).
> >
> > - **config_source** (*Union[str, class]*) – Take configuration from a class, or object. Attributes may include any settings described in the documentation.
> >
> > - **task_cls** (*Union[str, Type[celery.app.task.Task]]*) – base task class to use. See *this section* for usage.

> **user_options = None**
> > Custom options for command-line programs. See *Adding new command-line options*

> **steps = None**
> > Custom bootsteps to extend and modify the worker. See *Installing Bootsteps*.

> **current_task**
> > Instance of task being executed, or `None`.

> **current_worker_task**
> > The task currently being executed by a worker or `None`.
> >
> > Differs from *current_task* in that it's not affected by tasks calling other tasks directly, or eagerly.

amqp
> *amqp*.

>> **Type** AMQP related functionality

backend
> Current backend instance.

loader
> Current loader instance.

control
> *control*.

>> **Type** Remote control

events
> events.

>> **Type** Consuming and sending events

log
> *log*.

>> **Type** *Logging*

tasks
> Task registry.

> ---
> **Warning:** Accessing this attribute will also auto-finalize the app.
> ---

pool
> pool.

> ---
> **Note:** This attribute is not related to the workers concurrency pool.
> ---

>> **Type** Broker connection pool

producer_pool

Task
> Base task class for this app.

timezone
> Current timezone for this app.
>
> This is a cached property taking the time zone from the *timezone* setting.

builtin_fixups = {'celery.fixups.django:fixup'}

oid
> Universally unique identifier for this app.

close()
> Clean up after the application.
>
> Only necessary for dynamically created apps, and you should probably use the with statement instead.

Example

```
>>> with Celery(set_as_current=False) as app:
...     with app.connection_for_write() as conn:
...         pass
```

signature(**args*, ***kwargs*)
Return a new [Signature](#) bound to this app.

bugreport()
Return information useful in bug reports.

config_from_object(*obj*, *silent=False*, *force=False*, *namespace=None*)
Read configuration from object.

Object is either an actual object or the name of a module to import.

Example

```
>>> celery.config_from_object('myapp.celeryconfig')
```

```
>>> from myapp import celeryconfig
>>> celery.config_from_object(celeryconfig)
```

> Parameters
> * **silent** ([bool](#)) – If true then import errors will be ignored.
> * **force** ([bool](#)) – Force reading configuration immediately. By default the configuration will be read only when required.

config_from_envvar(*variable_name*, *silent=False*, *force=False*)
Read configuration from environment variable.

The value of the environment variable must be the name of a module to import.

Example

```
>>> os.environ['CELERY_CONFIG_MODULE'] = 'myapp.celeryconfig'
>>> celery.config_from_envvar('CELERY_CONFIG_MODULE')
```

autodiscover_tasks(*packages=None*, *related_name='tasks'*, *force=False*)
Auto-discover task modules.

Searches a list of packages for a "tasks.py" module (or use related_name argument).

If the name is empty, this will be delegated to fix-ups (e.g., Django).

For example if you have a directory layout like this:

```
foo/__init__.py
    tasks.py
    models.py

bar/__init__.py
    tasks.py
```

(continues on next page)

```
    models.py

baz/__init__.py
    models.py
```

Then calling `app.autodiscover_tasks(['foo', 'bar', 'baz'])` will result in the modules `foo.tasks` and `bar.tasks` being imported.

> **Parameters**
>
> - **packages** (`List[str]`) – List of packages to search. This argument may also be a callable, in which case the value returned is used (for lazy evaluation).
>
> - **related_name** (`str`) – The name of the module to find. Defaults to "tasks": meaning "look for 'module.tasks' for every module in `packages`.". If `None` will only try to import the package, i.e. "look for 'module'".
>
> - **force** (`bool`) – By default this call is lazy so that the actual auto-discovery won't happen until an application imports the default modules. Forcing will cause the auto-discovery to happen immediately.

add_defaults(*fun*)

Add default configuration from dict `d`.

If the argument is a callable function then it will be regarded as a promise, and it won't be loaded until the configuration is actually needed.

This method can be compared to:

```
>>> celery.conf.update(d)
```

with a difference that 1) no copy will be made and 2) the dict will not be transferred when the worker spawns child processes, so it's important that the same configuration happens at import time when pickle restores the object on the other side.

add_periodic_task(*schedule*, *sig*, *args=()*, *kwargs=()*, *name=None*, ***opts*)

setup_security(*allowed_serializers=None*, *key=None*, *cert=None*, *store=None*, *digest='sha256'*,
    serializer='json')

Setup the message-signing serializer.

This will affect all application instances (a global operation).

Disables untrusted serializers and if configured to use the `auth` serializer will register the `auth` serializer with the provided settings into the Kombu serializer registry.

> **Parameters**
>
> - **allowed_serializers** (`Set[str]`) – List of serializer names, or content_types that should be exempt from being disabled.
>
> - **key** (`str`) – Name of private key file to use. Defaults to the `security_key` setting.
>
> - **cert** (`str`) – Name of certificate file to use. Defaults to the `security_certificate` setting.
>
> - **store** (`str`) – Directory containing certificates. Defaults to the `security_cert_store` setting.
>
> - **digest** (`str`) – Digest algorithm used when signing messages. Default is `sha256`.

- **serializer** (*str*) – Serializer used to encode messages after they've been signed. See *task_serializer* for the serializers supported. Default is `json`.

start (*argv=None*)

Run **celery** using *argv*.

Uses `sys.argv` if *argv* is not specified.

task (**args*, ***opts*)

Decorator to create a task class out of any callable.

Examples

```python
@app.task
def refresh_feed(url):
    store_feed(feedparser.parse(url))
```

with setting extra options:

```python
@app.task(exchange='feeds')
def refresh_feed(url):
    return store_feed(feedparser.parse(url))
```

Note: App Binding: For custom apps the task decorator will return a proxy object, so that the act of creating the task is not performed until the task is used or the task registry is accessed.

If you're depending on binding to be deferred, then you must not access any attributes on the returned object until the application is fully set up (finalized).

---

send_task (*name*, *args=None*, *kwargs=None*, *countdown=None*, *eta=None*, *task_id=None*, *producer=None*, *connection=None*, *router=None*, *result_cls=None*, *expires=None*, *publisher=None*, *link=None*, *link_error=None*, *add_to_parent=True*, *group_id=None*, *retries=0*, *chord=None*, *reply_to=None*, *time_limit=None*, *soft_time_limit=None*, *root_id=None*, *parent_id=None*, *route_name=None*, *shadow=None*, *chain=None*, *task_type=None*, ***options*)

Send task by name.

Supports the same arguments as *Task.apply_async()*.

Parameters

- **name** (*str*) – Name of task to call (e.g., *"tasks.add"*).

- **result_cls** (*AsyncResult*) – Specify custom result class.

gen_task_name (*name*, *module*)

AsyncResult

Create new result instance.

See also:

celery.result.AsyncResult.

GroupResult

Create new group result instance.

See also:

celery.result.GroupResult.

---

worker_main(*argv=None*)
> Run **celery worker** using *argv*.
>
> Uses `sys.argv` if *argv* is not specified.

Worker
> Worker application.
>
> See also:
>
> *Worker*.

WorkController
> Embeddable worker.
>
> See also:
>
> *WorkController*.

Beat
> **celery beat** scheduler application.
>
> See also:
>
> *Beat*.

connection_for_read(*url=None*, ***kwargs*)
> Establish connection used for consuming.
>
> See also:
>
> *connection()* for supported arguments.

connection_for_write(*url=None*, ***kwargs*)
> Establish connection used for producing.
>
> See also:
>
> *connection()* for supported arguments.

connection(*hostname=None*, *userid=None*, *password=None*, *virtual_host=None*, *port=None*, *ssl=None*, *connect_timeout=None*, *transport=None*, *transport_options=None*, *heartbeat=None*, *login_method=None*, *failover_strategy=None*, ***kwargs*)
> Establish a connection to the message broker.
>
> Please use *connection_for_read()* and *connection_for_write()* instead, to convey the intent of use for this connection.
>
> > **Parameters**
> >
> > - **url** – Either the URL or the hostname of the broker to use.
> >
> > - **hostname** (*str*) – URL, Hostname/IP-address of the broker. If a URL is used, then the other argument below will be taken from the URL instead.
> >
> > - **userid** (*str*) – Username to authenticate as.
> >
> > - **password** (*str*) – Password to authenticate with
> >
> > - **virtual_host** (*str*) – Virtual host to use (domain).
> >
> > - **port** (*int*) – Port to connect to.
> >
> > - **ssl** (*bool, Dict*) – Defaults to the *broker_use_ssl* setting.
> >
> > - **transport** (*str*) – defaults to the `broker_transport` setting.
> >
> > - **transport_options** (*Dict*) – Dictionary of transport specific options.

- **heartbeat** (`int`) – AMQP Heartbeat in seconds (`pyamqp` only).

- **login_method** (`str`) – Custom login method to use (AMQP only).

- **failover_strategy** (`str, Callable`) – Custom failover strategy.

- ****kwargs** – Additional arguments to `kombu.Connection`.

> **Returns** the lazy connection instance.
>
> **Return type** [kombu.Connection](#)

connection_or_acquire(*connection=None*, *pool=True*, **_*, ***__*)
    Context used to acquire a connection from the pool.

    For use within a `with` statement to get a connection from the pool if one is not already provided.

> **Parameters connection** (`kombu.Connection`) – If not provided, a connection will be acquired from the connection pool.

producer_or_acquire(*producer=None*)
    Context used to acquire a producer from the pool.

    For use within a `with` statement to get a producer from the pool if one is not already provided

> **Parameters producer** (`kombu.Producer`) – If not provided, a producer will be acquired from the producer pool.

select_queues(*queues=None*)
    Select subset of queues.

> **Parameters queues** (`Sequence[str]`) – a list of queue names to keep.

now()
    Return the current time and date as a datetime.

set_current()
    Make this the current app for this thread.

set_default()
    Make this the default app for all threads.

finalize(*auto=False*)
    Finalize the app.

    This loads built-in tasks, evaluates pending task decorators, reads configuration, etc.

on_init()
    Optional callback called at init.

prepare_config(*c*)
    Prepare configuration before it is merged with the defaults.

on_configure
    Signal sent when app is loading configuration.

on_after_configure
    Signal sent after app has prepared the configuration.

on_after_finalize
    Signal sent after app has been finalized.

on_after_fork
    Signal sent in child process after fork.

Canvas primitives

See *Canvas: Designing Work-flows* for more about creating task work-flows.

class `celery.group`(**tasks*, ***options*)

Creates a group of tasks to be executed in parallel.

A group is lazy so you must call it to take action and evaluate the group.

Note: If only one argument is passed, and that argument is an iterable then that'll be used as the list of tasks instead: this allows us to use `group` with generator expressions.

Example

```
>>> lazy_group = group([add.s(2, 2), add.s(4, 4)])
>>> promise = lazy_group()  # <-- evaluate: returns lazy result.
>>> promise.get()  # <-- will wait for the task to return
[4, 8]
```

> **Parameters**
>
> - ***tasks** (*List[*`Signature`*]*) – A list of signatures that this group will call. If there's only one argument, and that argument is an iterable, then that'll define the list of signatures instead.
> - ****options** (*Any*) – Execution options applied to all tasks in the group.
>
> **Returns**
>
> > **signature that when called will then call all of the** tasks in the group (and return a `GroupResult` instance that can be used to inspect the state of the group).
>
> **Return type** *group*

class `celery.chain`(**tasks*, ***options*)

Chain tasks together.

Each tasks follows one another, by being applied as a callback of the previous task.

Note: If called with only one argument, then that argument must be an iterable of tasks to chain: this allows us to use generator expressions.

Example

This is effectively $((2 + 2) + 4)$:

```
>>> res = chain(add.s(2, 2), add.s(4))()
>>> res.get()
8
```

Calling a chain will return the result of the last task in the chain. You can get to the other tasks by following the `result.parent`'s:

```
>>> res.parent.get()
4
```

Using a generator expression:

```
>>> lazy_chain = chain(add.s(i) for i in range(10))
>>> res = lazy_chain(3)
```

> **Parameters *tasks** (`Signature`) – List of task signatures to chain. If only one argument is
> passed and that argument is an iterable, then that'll be used as the list of signatures to chain
> instead. This means that you can use a generator expression.
>
> **Returns**
>
> > **A lazy signature that can be called to apply the first** task in the chain. When that task succeeds the next task in the chain is applied, and so on.
>
> **Return type** *chain*

class `celery.`**chord**(*header*, *body=None*, *task='celery.chord'*, *args=None*, *kwargs=None*, *app=None*,
    **options)

Barrier synchronization primitive.

A chord consists of a header and a body.

The header is a group of tasks that must complete before the callback is called. A chord is essentially a callback
for a group of tasks.

The body is applied with the return values of all the header tasks as a list.

Example

The chord:

```
>>> res = chord([add.s(2, 2), add.s(4, 4)])(sum_task.s())
```

is effectively $\Sigma((2 + 2) + (4 + 4))$:

```
>>> res.get()
12
```

`celery.`**signature**(*varies*, **args*, ***kwargs*)

Create new signature.

- if the first argument is a signature already then it's cloned.
- if the first argument is a dict, then a Signature version is returned.

> **Returns** The resulting signature.
>
> **Return type** *Signature*

class `celery.`**Signature**(*task=None*, *args=None*, *kwargs=None*, *options=None*, *type=None*, *sub-
    task_type=None*, *immutable=False*, *app=None*, ***ex*)

Task Signature.

Class that wraps the arguments and execution options for a single task invocation.

Used as the parts in a *group* and other constructs, or to pass tasks around as callbacks while being compatible with serializers with a strict type subset.

Signatures can also be created from tasks:

- Using the `.signature()` method that has the same signature as `Task.apply_async`:

```
>>> add.signature(args=(1,), kwargs={'kw': 2}, options={})
```

- or the `.s()` shortcut that works for star arguments:

```
>>> add.s(1, kw=2)
```

- the `.s()` shortcut does not allow you to specify execution options but there's a chaning *.set* method that returns the signature:

```
>>> add.s(2, 2).set(countdown=10).set(expires=30).delay()
```

Note: You should use *signature()* to create new signatures. The `Signature` class is the type returned by that function and should be used for `isinstance` checks for signatures.

See also:

Canvas: Designing Work-flows for the complete guide.

> **Parameters**
>
> - **task** (*Union[Type[celery.app.task.Task], str]*) – Either a task class/instance, or the name of a task.
>
> - **args** (*Tuple*) – Positional arguments to apply.
>
> - **kwargs** (*Dict*) – Keyword arguments to apply.
>
> - **options** (*Dict*) – Additional options to `Task.apply_async()`.

Note: If the first argument is a `dict`, the other arguments will be ignored and the values in the dict will be used instead:

```
>>> s = signature('tasks.add', args=(2, 2))
>>> signature(s)
{'task': 'tasks.add', args=(2, 2), kwargs={}, options={}}
```

Proxies

`celery.``current_app`
    The currently set app for this thread.

`celery.``current_task`
    The task currently being executed (only set in the worker, or when eager/apply is used).

Celery Application.

3.11.2 Proxies

celery.app.**default_app = <Celery default>**
    Proxy always returning the app set as default.

3.11.3 Functions

celery.app.**app_or_default**(*app=None*)
    Function returning the app provided or the default app if none.

    The environment variable CELERY_TRACE_APP is used to trace app leaks. When enabled an exception is raised if there is no active app.

celery.app.**enable_trace**()
    Enable tracing of app instances.

celery.app.**disable_trace**()
    Disable tracing of app instances.

3.11.4 `celery.app.task`

Task implementation: request context and the task base class.

class celery.app.task.**Task**
    Task base class.

Note: When called tasks apply the *run()* method. This method must be defined by all tasks (that is unless the __call__() method is overridden).

AsyncResult(*task_id*, ***kwargs*)
        Get AsyncResult instance for the specified task.

                Parameters **task_id** (*str*) – Task id to get result for.

exception MaxRetriesExceededError(**args*, ***kwargs*)
        The tasks max restart limit has been exceeded.

exception OperationalError
        Recoverable message transport connection error.

Request = 'celery.worker.request:Request'
        Request class used, or the qualified name of one.

Strategy = 'celery.worker.strategy:default'
        Execution strategy used, or the qualified name of one.

abstract = True
        Deprecated attribute abstract here for compatibility.

acks_late = False
> When enabled messages for this task will be acknowledged **after** the task has been executed, and not *just before* (the default behavior).

> Please note that this means the task may be executed twice if the worker crashes mid execution.

> The application default can be overridden with the `task_acks_late` setting.

acks_on_failure_or_timeout = True
> When enabled messages for this task will be acknowledged even if it fails or times out.

> Configuring this setting only applies to tasks that are acknowledged **after** they have been executed and only if `task_acks_late` is enabled.

> The application default can be overridden with the `task_acks_on_failure_or_timeout` setting.

add_to_chord(*sig*, *lazy=False*)
> Add signature to the chord the current task is a member of.

> New in version 4.0.

> Currently only supported by the Redis result backend.

> > **Parameters**
> >
> > - **sig** (*~@Signature*) – Signature to extend chord with.
> >
> > - **lazy** (*bool*) – If enabled the new task won't actually be called, and `sig.delay()` must be called manually.

after_return(*status*, *retval*, *task_id*, *args*, *kwargs*, *einfo*)
> Handler called after the task returns.

> > **Parameters**
> >
> > - **status** (*str*) – Current task state.
> >
> > - **retval** (*Any*) – Task return value/exception.
> >
> > - **task_id** (*str*) – Unique id of the task.
> >
> > - **args** (*Tuple*) – Original arguments for the task.
> >
> > - **kwargs** (*Dict*) – Original keyword arguments for the task.
> >
> > - **einfo** (*ExceptionInfo*) – Exception information.
> >
> > **Returns** The return value of this handler is ignored.
> >
> > **Return type** None

apply(*args=None*, *kwargs=None*, *link=None*, *link_error=None*, *task_id=None*, *retries=None*, *throw=None*, *logfile=None*, *loglevel=None*, *headers=None*, ***options*)
> Execute this task locally, by blocking until the task returns.

> > **Parameters**
> >
> > - **args** (*Tuple*) – positional arguments passed on to the task.
> >
> > - **kwargs** (*Dict*) – keyword arguments passed on to the task.
> >
> > - **throw** (*bool*) – Re-raise task exceptions. Defaults to the `task_eager_propagates` setting.
> >
> > **Returns** pre-evaluated result.
> >
> > **Return type** *celery.result.EagerResult*

apply_async(*args=None,   kwargs=None,   task_id=None,   producer=None,   link=None,   link_error=None, shadow=None, **options*)
Apply tasks asynchronously by sending a message.

> **Parameters**
>
> - **args** (`Tuple`) – The positional arguments to pass on to the task.
>
> - **kwargs** (`Dict`) – The keyword arguments to pass on to the task.
>
> - **countdown** (`float`) – Number of seconds into the future that the task should execute. Defaults to immediate execution.
>
> - **eta** (`datetime`) – Absolute time and date of when the task should be executed. May not be specified if *countdown* is also supplied.
>
> - **expires** (`float, datetime`) – Datetime or seconds in the future for the task should expire. The task won't be executed after the expiration time.
>
> - **shadow** (`str`) – Override task name used in logs/monitoring. Default is retrieved from `shadow_name()`.
>
> - **connection** (`kombu.Connection`) – Re-use existing broker connection instead of acquiring one from the connection pool.
>
> - **retry** (`bool`) – If enabled sending of the task message will be retried in the event of connection loss or failure. Default is taken from the `task_publish_retry` setting. Note that you need to handle the producer/connection manually for this to work.
>
> - **retry_policy** (`Mapping`) – Override the retry policy used. See the `task_publish_retry_policy` setting.
>
> - **queue** (`str, kombu.Queue`) – The queue to route the task to. This must be a key present in `task_queues`, or `task_create_missing_queues` must be enabled. See *Routing Tasks* for more information.
>
> - **exchange** (`str, kombu.Exchange`) – Named custom exchange to send the task to. Usually not used in combination with the `queue` argument.
>
> - **routing_key** (`str`) – Custom routing key used to route the task to a worker server. If in combination with a `queue` argument only used to specify custom routing keys to topic exchanges.
>
> - **priority** (`int`) – The task priority, a number between 0 and 9. Defaults to the `priority` attribute.
>
> - **serializer** (`str`) – Serialization method to use. Can be *pickle*, *json*, *yaml*, *msgpack* or any custom serialization method that's been registered with `kombu.serialization.registry`. Defaults to the `serializer` attribute.
>
> - **compression** (`str`) – Optional compression method to use. Can be one of `zlib`, `bzip2`, or any custom compression methods registered with `kombu.compression.register()`. Defaults to the `task_compression` setting.
>
> - **link** (`Signature`) – A single, or a list of tasks signatures to apply if the task returns successfully.
>
> - **link_error** (`Signature`) – A single, or a list of task signatures to apply if an error occurs while executing the task.
>
> - **producer** (`kombu.Producer`) – custom producer to use when publishing the task.

- **add_to_parent** (*bool*) – If set to True (default) and the task is applied while executing another task, then the result will be appended to the parent tasks `request.children` attribute. Trailing can also be disabled by default using the *trail* attribute

- **publisher** (*kombu.Producer*) – Deprecated alias to `producer`.

- **headers** (*Dict*) – Message headers to be included in the message.

Returns Promise of future evaluation.

Return type *celery.result.AsyncResult*

Raises

- `TypeError` – If not enough arguments are passed, or too many arguments are passed. Note that signature checks may be disabled by specifying `@task(typing=False)`.

- `kombu.exceptions.OperationalError` – If a connection to the transport cannot be made, or if the connection is lost.

Note: Also supports all keyword arguments supported by `kombu.Producer.publish()`.

---

autoregister = True
    If disabled this task won't be registered automatically.

backend
    The result store backend used for this task.

chunks (*it*, *n*)
    Create a `chunks` task for this task.

default_retry_delay = 180
    Default time in seconds before a retry of the task should be executed. 3 minutes by default.

delay (**args*, ***kwargs*)
    Star argument version of *apply_async()*.

    Does not support the extra options enabled by *apply_async()*.

    Parameters

    - ***args** (*Any*) – Positional arguments passed on to the task.

    - ****kwargs** (*Any*) – Keyword arguments passed on to the task.

    Returns Future promise.

    Return type *celery.result.AsyncResult*

expires = None
    Default task expiry time.

ignore_result = False
    If enabled the worker won't store task state and return values for this task. Defaults to the *task_ignore_result* setting.

map (*it*)
    Create a `xmap` task from `it`.

max_retries = 3
    Maximum number of retries before giving up. If set to `None`, it will **never** stop retrying.

name = None
    Name of the task.

classmethod on_bound(*app*)

Called when the task is bound to an app.

Note: This class method can be defined to do additional actions when the task class is bound to an app.

---

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

Error handler.

This is run by the worker when the task fails.

> **Parameters**
>
> - **exc** (*Exception*) – The exception raised by the task.
> - **task_id** (*str*) – Unique id of the failed task.
> - **args** (*Tuple*) – Original arguments for the task that failed.
> - **kwargs** (*Dict*) – Original keyword arguments for the task that failed.
> - **einfo** (*ExceptionInfo*) – Exception information.
>
> **Returns** The return value of this handler is ignored.
>
> **Return type** None

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

Retry handler.

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

> **Parameters**
>
> - **exc** (*Exception*) – The exception sent to *retry()*.
> - **task_id** (*str*) – Unique id of the retried task.
> - **args** (*Tuple*) – Original arguments for the retried task.
> - **kwargs** (*Dict*) – Original keyword arguments for the retried task.
> - **einfo** (*ExceptionInfo*) – Exception information.
>
> **Returns** The return value of this handler is ignored.
>
> **Return type** None

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

Success handler.

Run by the worker if the task executes successfully.

> **Parameters**
>
> - **retval** (*Any*) – The return value of the task.
> - **task_id** (*str*) – Unique id of the executed task.
> - **args** (*Tuple*) – Original arguments for the executed task.
> - **kwargs** (*Dict*) – Original keyword arguments for the executed task.
>
> **Returns** The return value of this handler is ignored.
>
> **Return type** None

priority = None
> Default task priority.

rate_limit = None
> `None` (no rate limit), *'100/s'* (hundred tasks a second), *'100/m'* (hundred tasks a minute),"100/h'' (hundred tasks an hour)
>
> > **Type** Rate limit for this task type. Examples

reject_on_worker_lost = None
> Even if *acks_late* is enabled, the worker will acknowledge tasks when the worker process executing them abruptly exits or is signaled (e.g., KILL/INT, etc).
>
> Setting this to true allows the message to be re-queued instead, so that the task will execute again by the same worker, or another worker.
>
> Warning: Enabling this can cause message loops; make sure you know what you're doing.

replace(*sig*)
> Replace this task, with a new task inheriting the task id.
>
> Execution of the host task ends immediately and no subsequent statements will be run.
>
> New in version 4.0.
>
> > **Parameters sig** (~@Signature) – signature to replace with.
> >
> > **Raises**
> >
> > > - *Ignore* – This is always raised when called in asynchrous context.
> > > - It is best to always use `return self.replace(...)` to convey
> > > - to the reader that the task won't continue after being replaced.

request
> Get current request object.

request_stack = <celery.utils.threads._LocalStack object>
> Task request stack, the current request will be the topmost.

resultrepr_maxsize = 1024
> Max length of result representation used in logs and events.

retry(*args=None*, *kwargs=None*, *exc=None*, *throw=True*, *eta=None*, *countdown=None*, *max_retries=None*, ***options*)
> Retry the task, adding it to the back of the queue.

Example

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

```
>>> @app.task(bind=True)
... def tweet(self, auth, message):
...     twitter = Twitter(oauth=auth)
...     try:
...         twitter.post_status_update(message)
...     except twitter.FailWhale as exc:
...         # Retry in 5 minutes.
...         self.retry(countdown=60 * 5, exc=exc)
```

Note: Although the task will never return above as *retry* raises an exception to notify the worker, we use *raise* in front of the retry to convey that the rest of the block won't be executed.

> **Parameters**
>
> - **args** (`Tuple`) – Positional arguments to retry with.
>
> - **kwargs** (`Dict`) – Keyword arguments to retry with.
>
> - **exc** (`Exception`) – Custom exception to report when the max retry limit has been exceeded (default: `MaxRetriesExceededError`).
>
>   If this argument is set and retry is called while an exception was raised (`sys. exc_info()` is set) it will attempt to re-raise the current exception.
>
>   If no exception was raised it will raise the `exc` argument provided.
>
> - **countdown** (`float`) – Time in seconds to delay the retry for.
>
> - **eta** (`datetime`) – Explicit time and date to run the retry at.
>
> - **max_retries** (`int`) – If set, overrides the default retry limit for this execution. Changes to this parameter don't propagate to subsequent task retry attempts. A value of `None`, means "use the default", so if you want infinite retries you'd have to set the `max_retries` attribute of the task to `None` first.
>
> - **time_limit** (`int`) – If set, overrides the default time limit.
>
> - **soft_time_limit** (`int`) – If set, overrides the default soft time limit.
>
> - **throw** (`bool`) – If this is `False`, don't raise the `Retry` exception, that tells the worker to mark the task as being retried. Note that this means the task will be marked as failed if the task raises an exception, or successful if it returns after the retry call.
>
> - ****options** (`Any`) – Extra options to pass on to `apply_async()`.
>
> **Raises** `celery.exceptions.Retry` – To tell the worker that the task has been re-sent for retry. This always happens, unless the *throw* keyword argument has been explicitly set to `False`, and is considered normal operation.

run (**args*, ***kwargs*)
    The body of the task executed by workers.

s (**args*, ***kwargs*)
    Create signature.

    Shortcut for `.s(*a, **k) -> .signature(a, k)`.

send_event (*type_*, *retry=True*, *retry_policy=None*, ***fields*)
    Send monitoring event message.

This can be used to add custom event types in Flower and other monitors.

> **Parameters** **type** (`str`) – Type of event, e.g. `"task-failed"`.
>
> **Keyword Arguments**
>
> - **retry** (`bool`) – Retry sending the message if the connection is lost. Default is taken from the `task_publish_retry` setting.
>
> - **retry_policy** (`Mapping`) – Retry settings. Default is taken from the `task_publish_retry_policy` setting.

- ****fields** (*Any*) – Map containing information about the event. Must be JSON serializable.

send_events = True
> If enabled the worker will send monitoring events related to this task (but only if the worker is configured to send task related events). Note that this has no effect on the task-failure event case where a task is not registered (as it will have no task class to check this flag).

serializer = 'json'
> The name of a serializer that are registered with `kombu.serialization.registry`. Default is *'json'*.

shadow_name(*args*, *kwargs*, *options*)
> Override for custom task name in worker logs/monitoring.

Example

```python
from celery.utils.imports import qualname

def shadow_name(task, args, kwargs, options):
    return qualname(args[0])

@app.task(shadow_name=shadow_name, serializer='pickle')
def apply_function_async(fun, *args, **kwargs):
    return fun(*args, **kwargs)
```

> #### Parameters
>
> - **args** (*Tuple*) – Task positional arguments.
>
> - **kwargs** (*Dict*) – Task keyword arguments.
>
> - **options** (*Dict*) – Task execution options.

si(**args*, ***kwargs*)
> Create immutable signature.
>
> Shortcut for `.si(*a, **k)` -> `.signature(a, k, immutable=True)`.

signature(*args=None*, **starargs*, ***starkwargs*)
> Create signature.
>
> > **Returns**
> >
> > > **object for** this task, wrapping arguments and execution options for a single task invocation.
> >
> > **Return type** *signature*

soft_time_limit = None
> Soft time limit. Defaults to the *task_soft_time_limit* setting.

starmap(*it*)
> Create a `xstarmap` task from `it`.

store_errors_even_if_ignored = False
> When enabled errors will be stored even if the task is otherwise configured to ignore results.

subtask(*args=None*, **starargs*, ***starkwargs*)
> Create signature.
>
> > **Returns**

> **object for** this task, wrapping arguments and execution options for a single task invocation.
>
> > **Return type** *signature*

throws = ()
  Tuple of expected exceptions.

  These are errors that are expected in normal operation and that shouldn't be regarded as a real error by the worker. Currently this means that the state will be updated to an error state, but the worker won't log the event as an error.

time_limit = None
  Hard time limit. Defaults to the *task_time_limit* setting.

track_started = False
  If enabled the task will report its status as 'started' when the task is executed by a worker. Disabled by default 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 application default can be overridden using the *task_track_started* setting.

trail = True
  If enabled the request will keep track of subtasks started by this task, and this information will be sent with the result (`result.children`).

typing = True
  Enable argument checking. You can set this to false if you don't want the signature to be checked when calling the task. Defaults to `Celery.strict_typing`.

update_state(*task_id=None*, *state=None*, *meta=None*, ***kwargs*)
  Update task state.

  > **Parameters**
  >
  > > - **task_id** (*str*) – Id of the task to update. Defaults to the id of the current task.
  > >
  > > - **state** (*str*) – New state.
  > >
  > > - **meta** (*Dict*) – State meta-data.

class celery.app.task.**Context**(**args*, ***kwargs*)
  Task request variables (Task.request).

celery.app.task.**TaskType**
  alias of `builtins.type`

Sending/Receiving Messages (Kombu integration).

- *AMQP*

- *Queues*

3.11.5 AMQP

class celery.app.amqp.**AMQP**(*app*)
  App AMQP API: app.amqp.

Connection
> Broker connection class used. Default is `kombu.Connection`.

Consumer
> Base Consumer class used. Default is `kombu.Consumer`.

Producer
> Base Producer class used. Default is `kombu.Producer`.

queues
> All currently defined task queues (a `Queues` instance).

argsrepr_maxsize
> Max size of positional argument representation used for logging purposes. Default is 1024.

kwargsrepr_maxsize
> Max size of keyword argument representation used for logging purposes. Default is 1024.

Queues(*queues*, *create_missing=None*, *ha_policy=None*, *autoexchange=None*, *max_priority=None*)

Router(*queues=None*, *create_missing=None*)
> Return the current task router.

flush_routes()

create_task_message

send_task_message

default_queue

default_exchange

producer_pool

router

routes

3.11.6 Queues

class `celery.app.amqp.`**Queues**(*queues=None*, *default_exchange=None*, *create_missing=True*, *ha_policy=None*, *autoexchange=None*, *max_priority=None*, *default_routing_key=None*)
Queue name declaration mapping.

> **Parameters**
>
> - **queues** (`Iterable`) – Initial list/tuple or dict of queues.
> - **create_missing** (`bool`) – By default any unknown queues will be added automatically, but if this flag is disabled the occurrence of unknown queues in *wanted* will raise `KeyError`.
> - **ha_policy** (`Sequence, str`) – Default HA policy for queues with none set.
> - **max_priority** (`int`) – Default x-max-priority for queues with none set.

add(*queue*, ***kwargs*)
> Add new queue.
>
> The first argument can either be a `kombu.Queue` instance, or the name of a queue. If the former the rest of the keyword arguments are ignored, and options are simply taken from the queue instance.
>
> > **Parameters**

- **queue** (*kombu.Queue, str*) – Queue to add.

- **exchange** (*kombu.Exchange, str*) – if queue is str, specifies exchange name.

- **routing_key** (*str*) – if queue is str, specifies binding key.

- **exchange_type** (*str*) – if queue is str, specifies type of exchange.

- ****options** (*Any*) – Additional declaration options used when queue is a str.

add_compat(*name*, ***options*)

consume_from

deselect(*exclude*)
> Deselect queues so that they won't be consumed from.

>> Parameters **exclude** (*Sequence[str], str*) – Names of queues to avoid consuming from.

format(*indent=0*, *indent_first=True*)
> Format routing table into string for log dumps.

new_missing(*name*)

select(*include*)
> Select a subset of currently defined queues to consume from.

>> Parameters **include** (*Sequence[str], str*) – Names of queues to consume from.

select_add(*queue*, ***kwargs*)
> Add new task queue that'll be consumed from.

> The queue will be active even when a subset has been selected using the *celery worker -Q* option.

3.11.7 `celery.app.defaults`

Configuration introspection and defaults.

class celery.app.defaults.**Option**(*default=None*, **args*, ***kwargs*)
> Describes a Celery configuration option.

> **alt = None**

> **deprecate_by = None**

> **old = {}**

> **remove_by = None**

> **to_python**(*value*)

> **typemap = {'any': <function Option.<lambda>>, 'bool': <function strtobool>, 'dict':**

celery.app.defaults.**flatten**(*d*, *root=''*, *keyfilter=<function _flatten_keys>*)
> Flatten settings.

celery.app.defaults.**find**(*name*, *namespace='celery'*)
> Find setting by name.

3.11.8 `celery.app.control`

Worker Remote Control Client.

Client for worker remote control commands. Server implementation is in `celery.worker.control`.

class celery.app.control.**Inspect**(*destination=None*, *timeout=1.0*, *callback=None*, *connection=None*, *app=None*, *limit=None*, *pattern=None*, *matcher=None*)

> API for app.control.inspect.
>
> **active**(*safe=None*)
>
> **active_queues**()
>
> **app = None**
>
> **clock**()
>
> **conf**(*with_defaults=False*)
>
> **hello**(*from_node*, *revoked=None*)
>
> **memdump**(*samples=10*)
>
> **memsample**()
>
> **objgraph**(*type='Request'*, *n=200*, *max_depth=10*)
>
> **ping**(*destination=None*)
>
> **query_task**(**ids*)
>
> **registered**(**taskinfoitems*)
>
> **registered_tasks**(**taskinfoitems*)
>
> **report**()
>
> **reserved**(*safe=None*)
>
> **revoked**()
>
> **scheduled**(*safe=None*)
>
> **stats**()

class celery.app.control.**Control**(*app=None*)

> Worker remote control client.
>
> **class Mailbox**(*namespace*, *type='direct'*, *connection=None*, *clock=None*, *accept=None*, *serializer=None*, *producer_pool=None*, *queue_ttl=None*, *queue_expires=None*, *reply_queue_ttl=None*, *reply_queue_expires=10.0*)
>
> > Process Mailbox.
> >
> > **Node**(*hostname=None*, *state=None*, *channel=None*, *handlers=None*)
> >
> > **abcast**(*command*, *kwargs=None*)
> >
> > **accept = ['json']**
> >
> > **call**(*destination*, *command*, *kwargs=None*, *timeout=None*, *callback=None*, *channel=None*)
> >
> > **cast**(*destination*, *command*, *kwargs=None*)
> >
> > **connection = None**
> >
> > **exchange = None**

exchange_fmt = '%s.pidbox'

get_queue(*hostname*)

get_reply_queue()

multi_call(*command*, *kwargs=None*, *timeout=1*, *limit=None*, *callback=None*, *channel=None*)

namespace = None

node_cls
    alias of [`Node`](#)

oid

producer_or_acquire(*producer=None*, *channel=None*)

producer_pool

reply_exchange = None

reply_exchange_fmt = 'reply.%s.pidbox'

reply_queue

serializer = None

type = 'direct'

add_consumer(*queue*, *exchange=None*, *exchange_type='direct'*, *routing_key=None*, *options=None*,
                destination=None, ***kwargs*)
Tell all (or specific) workers to start consuming from a new queue.

Only the queue name is required as if only the queue is specified then the exchange/routing key will be set
to the same name (like automatic queues do).

Note: This command does not respect the default queue/exchange options in the configuration.

Parameters

- **queue** ([`str`](#)) – Name of queue to start consuming from.

- **exchange** ([`str`](#)) – Optional name of exchange.

- **exchange_type** ([`str`](#)) – Type of exchange (defaults to 'direct') command to, when
  empty broadcast to all workers.

- **routing_key** ([`str`](#)) – Optional routing key.

- **options** (*Dict*) – Additional options as supported by `kombu.entity.Queue.`
  `from_dict()`.

See also:

[`broadcast()`](#) for supported keyword arguments.

autoscale(*max*, *min*, *destination=None*, ***kwargs*)
Change worker(s) autoscale setting.

See also:

Supports the same arguments as [`broadcast()`](#).

broadcast (*command*, *arguments=None*, *destination=None*, *connection=None*, *reply=False*, *time-out=1.0*, *limit=None*, *callback=None*, *channel=None*, *pattern=None*, *matcher=None*, ***extra_kwargs*)

   Broadcast a control command to the celery workers.

   Parameters

   - **command** (`str`) – Name of command to send.

   - **arguments** (`Dict`) – Keyword arguments for the command.

   - **destination** (`List`) – If set, a list of the hosts to send the command to, when empty broadcast to all workers.

   - **connection** (`kombu.Connection`) – Custom broker connection to use, if not set, a connection will be acquired from the pool.

   - **reply** (`bool`) – Wait for and return the reply.

   - **timeout** (`float`) – Timeout in seconds to wait for the reply.

   - **limit** (`int`) – Limit number of replies.

   - **callback** (`Callable`) – Callback called immediately for each reply received.

   - **pattern** (`str`) – Custom pattern string to match

   - **matcher** (`Callable`) – Custom matcher to run the pattern to match

cancel_consumer (*queue*, *destination=None*, ***kwargs*)

   Tell all (or specific) workers to stop consuming from `queue`.

   See also:

   Supports the same arguments as `broadcast()`.

disable_events (*destination=None*, ***kwargs*)

   Tell all (or specific) workers to disable events.

   See also:

   Supports the same arguments as `broadcast()`.

discard_all (*connection=None*)

   Discard all waiting tasks.

   This will ignore all tasks waiting for execution, and they will be deleted from the messaging server.

   Parameters connection (`kombu.Connection`) – Optional specific connection instance to use. If not provided a connection will be acquired from the connection pool.

   Returns the number of tasks discarded.

   Return type int

election (*id*, *topic*, *action=None*, *connection=None*)

enable_events (*destination=None*, ***kwargs*)

   Tell all (or specific) workers to enable events.

   See also:

   Supports the same arguments as `broadcast()`.

heartbeat (*destination=None*, ***kwargs*)

   Tell worker(s) to send a heartbeat immediately.

   See also:

Supports the same arguments as *broadcast()*

inspect

ping(*destination=None*, *timeout=1.0*, ***kwargs*)

> Ping all (or specific) workers.
>
> > **Returns** List of {`'hostname'`: `reply`} dictionaries.
> >
> > **Return type** List[Dict]
>
> **See also:**
>
> *broadcast()* for supported keyword arguments.

pool_grow(*n=1*, *destination=None*, ***kwargs*)

> Tell all (or specific) workers to grow the pool by n.
>
> **See also:**
>
> Supports the same arguments as *broadcast()*.

pool_restart(*modules=None*, *reload=False*, *reloader=None*, *destination=None*, ***kwargs*)

> Restart the execution pools of all or specific workers.
>
> > **Keyword Arguments**
> >
> > - **modules** (*Sequence[str]*) – List of modules to reload.
> >
> > - **reload** (*bool*) – Flag to enable module reloading. Default is False.
> >
> > - **reloader** (*Any*) – Function to reload a module.
> >
> > - **destination** (*Sequence[str]*) – List of worker names to send this command to.
>
> **See also:**
>
> Supports the same arguments as *broadcast()*

pool_shrink(*n=1*, *destination=None*, ***kwargs*)

> Tell all (or specific) workers to shrink the pool by n.
>
> **See also:**
>
> Supports the same arguments as *broadcast()*.

purge(*connection=None*)

> Discard all waiting tasks.
>
> This will ignore all tasks waiting for execution, and they will be deleted from the messaging server.
>
> > **Parameters** **connection** (*kombu.Connection*) – Optional specific connection instance to use. If not provided a connection will be acquired from the connection pool.
> >
> > **Returns** the number of tasks discarded.
> >
> > **Return type** int

rate_limit(*task_name*, *rate_limit*, *destination=None*, ***kwargs*)

> Tell workers to set a new rate limit for task by type.
>
> > **Parameters**
> >
> > - **task_name** (*str*) – Name of task to change rate limit for.
> >
> > - **rate_limit** (*int, str*) – The rate limit as tasks per second, or a rate limit string (*'100/m'*, etc. see `celery.task.base.Task.rate_limit` for more information).

See also:

`broadcast()` for supported keyword arguments.

revoke (*task_id*, *destination=None*, *terminate=False*, *signal='SIGTERM'*, ***kwargs*)
Tell all (or specific) workers to revoke a task by id.

If a task is revoked, the workers will ignore the task and not execute it after all.

> **Parameters**
>
> - **task_id** (*str*) – Id of the task to revoke.
>
> - **terminate** (*bool*) – Also terminate the process currently working on the task (if any).
>
> - **signal** (*str*) – Name of signal to send to process if terminate. Default is TERM.

See also:

`broadcast()` for supported keyword arguments.

shutdown (*destination=None*, ***kwargs*)
Shutdown worker(s).

See also:

Supports the same arguments as `broadcast()`

terminate (*task_id*, *destination=None*, *signal='SIGTERM'*, ***kwargs*)
Tell all (or specific) workers to terminate a task by id.

See also:

This is just a shortcut to `revoke()` with the terminate argument enabled.

time_limit (*task_name*, *soft=None*, *hard=None*, *destination=None*, ***kwargs*)
Tell workers to set time limits for a task by type.

> **Parameters**
>
> - **task_name** (*str*) – Name of task to change time limits for.
>
> - **soft** (*float*) – New soft time limit (in seconds).
>
> - **hard** (*float*) – New hard time limit (in seconds).
>
> - ****kwargs** (*Any*) – arguments passed on to `broadcast()`.

celery.app.control.**flatten_reply** (*reply*)
Flatten node replies.

Convert from a list of replies in this format:

```
[{'a@example.com': reply},
 {'b@example.com': reply}]
```

into this format:

```
{'a@example.com': reply,
 'b@example.com': reply}
```

3.11.9 `celery.app.registry`

Registry of available tasks.

class celery.app.registry.**TaskRegistry**
> Map of registered tasks.

> **exception NotRegistered**
>> The task ain't registered.

> **filter_types**(*type*)

> **periodic**()

> **register**(*task*)
>> Register a task in the task registry.

>> The task will be automatically instantiated if not already an instance. Name must be configured prior to registration.

> **regular**()

> **unregister**(*name*)
>> Unregister task by name.

>>> **Parameters name** ([*str*](#)) – name of the task to unregister, or a celery.task.base.Task with a valid *name* attribute.

>>> **Raises** *celery.exceptions.NotRegistered* – if the task is not registered.

3.11.10 `celery.app.backends`

Backend selection.

celery.app.backends.**by_name**(*backend=None*,   *loader=None*,   *extension_namespace='celery.result_backends'*)
> Get backend class by name/alias.

celery.app.backends.**by_url**(*backend=None*, *loader=None*)
> Get backend class by URL.

3.11.11 `celery.app.builtins`

Built-in Tasks.

The built-in tasks are always available in all app instances.

3.11.12 `celery.app.events`

Implementation for the app.events shortcuts.

class celery.app.events.**Events**(*app=None*)
> Implements app.events.

> **Dispatcher**

> **Receiver**

> **State**

> **default_dispatcher**(*hostname=None*, *enabled=True*, *buffer_while_offline=False*)

> **dispatcher_cls = 'celery.events.dispatcher:EventDispatcher'**

> **receiver_cls = 'celery.events.receiver:EventReceiver'**

```
state_cls = 'celery.events.state:State'
```

3.11.13 `celery.app.log`

Logging configuration.

The Celery instances logging section: `Celery.log`.

Sets up logging for the worker and other programs, redirects standard outs, colors log output, patches logging related compatibility fixes, and so on.

class `celery.app.log.`**`TaskFormatter`**(*fmt=None*, *use_color=True*)

Formatter for tasks, adding the task name and id.

> **`format`**(*record*)
>
> Format the specified record as text.
>
> The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

class `celery.app.log.`**`Logging`**(*app*)

Application logging setup (app.log).

> **`already_setup = False`**
>
> **`colored`**(*logfile=None*, *enabled=None*)
>
> **`get_default_logger`**(*name='celery'*, ***kwargs*)
>
> **`redirect_stdouts`**(*loglevel=None*, *name='celery.redirected'*)
>
> **`redirect_stdouts_to_logger`**(*logger*, *loglevel=None*, *stdout=True*, *stderr=True*)
>
> Redirect `sys.stdout` and `sys.stderr` to logger.
>
> > **Parameters**
> >
> > • **logger** (`logging.Logger`) – Logger instance to redirect to.
> >
> > • **loglevel** (`int, str`) – The loglevel redirected message will be logged as.
>
> **`setup`**(*loglevel=None*, *logfile=None*, *redirect_stdouts=False*, *redirect_level='WARNING'*, *colorize=None*, *hostname=None*)
>
> **`setup_handlers`**(*logger*, *logfile*, *format*, *colorize*, *formatter=<class 'celery.utils.log.ColorFormatter'>*, ***kwargs*)
>
> **`setup_logger`**(*name='celery'*, **args*, ***kwargs*)
>
> Deprecated: No longer used.
>
> **`setup_logging_subsystem`**(*loglevel=None*, *logfile=None*, *format=None*, *colorize=None*, *hostname=None*, ***kwargs*)
>
> **`setup_task_loggers`**(*loglevel=None*, *logfile=None*, *format=None*, *colorize=None*, *propagate=False*, ***kwargs*)
>
> Setup the task logger.
>
> If *logfile* is not specified, then *sys.stderr* is used.
>
> Will return the base task logger object.
>
> **`supports_color`**(*colorize=None*, *logfile=None*)

3.11.14 `celery.app.utils`

App utilities: Compat settings, bug-report tool, pickling apps.

class `celery.app.utils.`**`Settings`**(*changes*, *defaults=None*, *keys=None*, *prefix=None*)
    Celery settings object.

    `broker_read_url`

    `broker_url`

    `broker_write_url`

    `finalize`()

    `find_option`(*name*, *namespace=''*)
        Search for option by name.

        Example

```
>>> from proj.celery import app
>>> app.conf.find_option('disable_rate_limits')
('worker', 'prefetch_multiplier',
 <Option: type->bool default->False>))
```

            Parameters

                • **name** (*str*) – Name of option, cannot be partial.

                • **namespace** (*str*) – Preferred name-space (`None` by default).

            Returns of (`namespace, key, type`).

            Return type Tuple

    `find_value_for_key`(*name*, *namespace='celery'*)
        Shortcut to `get_by_parts(*find_option(name)[:-1])`.

    `get_by_parts`(**parts*)
        Return the current value for setting specified as a path.

        Example

```
>>> from proj.celery import app
>>> app.conf.get_by_parts('worker', 'disable_rate_limits')
False
```

    `humanize`(*with_defaults=False*, *censored=True*)
        Return a human readable text showing configuration changes.

    `result_backend`

    `table`(*with_defaults=False*, *censored=True*)

    `task_default_exchange`

    `task_default_routing_key`

    `timezone`

value_set_for(*key*)

without_defaults()
>   Return the current configuration, but without defaults.

celery.app.utils.**appstr**(*app*)
>   String used in __repr__ etc, to id app instances.

celery.app.utils.**bugreport**(*app*)
>   Return a string containing information useful in bug-reports.

celery.app.utils.**filter_hidden_settings**(*conf*)
>   Filter sensitive settings.

celery.app.utils.**find_app**(*app*, *symbol_by_name=<function symbol_by_name>*, *imp=<function import_from_cwd>*)
>   Find app by name.

3.11.15 `celery.bootsteps`

A directed acyclic graph of reusable components.

class celery.bootsteps.**Blueprint**(*steps=None*, *name=None*, *on_start=None*, *on_close=None*, *on_stopped=None*)
>   Blueprint containing bootsteps that can be applied to objects.
>
>   **Parameters**
>
>   - **Sequence[Union[str, Step]]** (*steps*) – List of steps.
>
>   - **name** (*str*) – Set explicit name for this blueprint.
>
>   - **on_start** (*Callable*) – Optional callback applied after blueprint start.
>
>   - **on_close** (*Callable*) – Optional callback applied before blueprint close.
>
>   - **on_stopped** (*Callable*) – Optional callback applied after blueprint stopped.

>   **GraphFormatter**
>   >   alias of StepFormatter

>   **alias**

>   **apply**(*parent*, ***kwargs*)
>   >   Apply the steps in this blueprint to an object.
>
>   >   This will apply the __init__ and include methods of each step, with the object as argument:
>
>    >    ```
>    >    step = Step(obj)
>    >    ...
>    >    step.include(obj)
>    >    ```
>
>   >   For *StartStopStep* the services created will also be added to the objects steps attribute.

>   **claim_steps**()

>   **close**(*parent*)

>   **connect_with**(*other*)

>   **default_steps = {}**

>   **human_state**()

>   **info**(*parent*)

---

join (*timeout=None*)

load_step (*step*)

name = None

restart (*parent*, *method='stop'*, *description='restarting'*, *propagate=False*)

send_all (*parent*, *method*, *description=None*, *reverse=True*, *propagate=True*, *args=()*)

start (*parent*)

started = 0

state = None

state_to_name = {0: 'initializing', 1: 'running', 2: 'closing', 3: 'terminating'}

stop (*parent*, *close=True*, *terminate=False*)

class `celery.bootsteps.`**Step** (*parent*, ***kwargs*)

A Bootstep.

The __init__() method is called when the step is bound to a parent object, and can as such be used to initialize attributes in the parent object at parent instantiation-time.

alias

conditional = False

Set this to true if the step is enabled based on some condition.

create (*parent*)

Create the step.

enabled = True

This provides the default for *include_if()*.

include (*parent*)

include_if (*parent*)

Return true if bootstep should be included.

You can define this as an optional predicate that decides whether this step should be created.

info (*obj*)

instantiate (*name*, **args*, ***kwargs*)

label = None

Optional short name used for graph outputs and in logs.

last = False

This flag is reserved for the workers Consumer, since it is required to always be started last. There can only be one object marked last in every blueprint.

name = 'celery.bootsteps.Step'

Optional step name, will use qualname if not specified.

requires = ()

List of other steps that that must be started before this step. Note that all dependencies must be in the same blueprint.

class `celery.bootsteps.`**StartStopStep** (*parent*, ***kwargs*)

Bootstep that must be started and stopped in order.

close (*parent*)

---

include(*parent*)

name = 'celery.bootsteps.StartStopStep'

obj = None
> Optional obj created by the create() method. This is used by *StartStopStep* to keep the original service object.

start(*parent*)

stop(*parent*)

terminate(*parent*)

class celery.bootsteps.**ConsumerStep**(*parent*, ***kwargs*)
> Bootstep that starts a message consumer.

consumers = None

get_consumers(*channel*)

name = 'celery.bootsteps.ConsumerStep'

requires = ('celery.worker.consumer:Connection',)

shutdown(*c*)

start(*c*)

stop(*c*)

3.11.16 `celery.result`

Task results/state and results for groups of tasks.

class celery.result.**ResultBase**
> Base class for results.

parent = None
> Parent result (if part of a chain)

class celery.result.**AsyncResult**(*id*, *backend=None*, *task_name=None*, *app=None*, *parent=None*)
> Query task state.

> **Parameters**
> - **id** (*str*) – See *id*.
> - **backend** (*Backend*) – See *backend*.

exception TimeoutError
> The operation timed out.

app = None

args

as_tuple()

backend = None
> The task result backend to use.

build_graph(*intermediate=False*, *formatter=None*)

children

collect(*intermediate=False*, ***kwargs*)

Collect results as they return.

Iterator, like *get()* will wait for the task to complete, but will also follow *AsyncResult* and *ResultSet* returned by the task, yielding (`result,` `value`) tuples for each result in the tree.

An example would be having the following tasks:

```python
from celery import group
from proj.celery import app

@app.task(trail=True)
def A(how_many):
    return group(B.s(i) for i in range(how_many))()

@app.task(trail=True)
def B(i):
    return pow2.delay(i)

@app.task(trail=True)
def pow2(i):
    return i ** 2
```

```python
>>> from celery.result import ResultBase
>>> from proj.tasks import A

>>> result = A.delay(10)
>>> [v for v in result.collect()
...  if not isinstance(v, (ResultBase, tuple))]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note: The `Task.trail` option must be enabled so that the list of children is stored in `result.children`. This is the default but enabled explicitly for illustration.

> **Yields** *Tuple[AsyncResult, Any]* – tuples containing the result instance of the child task, and the return value of that task.

date_done

UTC date and time.

failed()

Return `True` if the task failed.

forget()

Forget the result of this task and its parents.

get(*timeout=None*, *propagate=True*, *interval=0.5*, *no_ack=True*, *follow_parents=True*, *callback=None*, *on_message=None*, *on_interval=None*, *disable_sync_subtasks=True*, *EXCEPTION_STATES=frozenset({'FAILURE', 'REVOKED', 'RETRY'})*, *PROPA-GATE_STATES=frozenset({'FAILURE', 'REVOKED'})*)

Wait until task is ready, and return its result.

Warning: Waiting for tasks within a task may lead to deadlocks. Please read *Avoid launching synchronous subtasks*.

> **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.

> **Parameters**
>
> - **timeout** (`float`) – How long to wait, in seconds, before the operation times out.
>
> - **propagate** (`bool`) – Re-raise exception if the task failed.
>
> - **interval** (`float`) – Time to wait (in seconds) before retrying to retrieve the result. Note that this does not have any effect when using the RPC/redis result store backends, as they don't use polling.
>
> - **no_ack** (`bool`) – Enable amqp no ack (automatically acknowledge message). If this is `False` then the message will **not be acked**.
>
> - **follow_parents** (`bool`) – Re-raise any exception raised by parent tasks.
>
> - **disable_sync_subtasks** (`bool`) – Disable tasks to wait for sub tasks this is the default configuration. CAUTION do not enable this unless you must.
>
> **Raises**
>
> - `celery.exceptions.TimeoutError` – if *timeout* isn't `None` and the result does not arrive within *timeout* seconds.
>
> - `Exception` – If the remote call raised an exception then that exception will be re-raised in the caller process.

get_leaf()

graph

id = None
> The task's UUID.

ignored
> If True, task result retrieval is disabled.

info
> Task return value.

> **Note:** When the task has been executed, this contains the return value. If the task raised an exception, this will be the exception instance.

iterdeps(*intermediate=False*)

kwargs

maybe_reraise(*propagate=True*, *callback=None*)

maybe_throw(*propagate=True*, *callback=None*)

name

queue

ready()
> Return `True` if the task has executed.

If the task is still running, pending, or is waiting for retry then `False` is returned.

result
> Task return value.

> **Note:** When the task has been executed, this contains the return value. If the task raised an exception, this will be the exception instance.

---

retries

revoke(*connection=None*, *terminate=False*, *signal=None*, *wait=False*, *timeout=None*)
> Send revoke signal to all workers.
>
> Any worker receiving the task, or having reserved the task, *must* ignore it.
>
> > **Parameters**
> >
> > - **terminate** (*bool*) – Also terminate the process currently working on the task (if any).
> >
> > - **signal** (*str*) – Name of signal to send to process if terminate. Default is TERM.
> >
> > - **wait** (*bool*) – Wait for replies from workers. The `timeout` argument specifies the seconds to wait. Disabled by default.
> >
> > - **timeout** (*float*) – Time in seconds to wait for replies when `wait` is enabled.

state
> The tasks current state.
>
> Possible values includes:
>
> > *PENDING*
> >
> > > The task is waiting for execution.
> >
> > *STARTED*
> >
> > > The task has been started.
> >
> > *RETRY*
> >
> > > The task is to be retried, possibly because of failure.
> >
> > *FAILURE*
> >
> > > The task raised an exception, or has exceeded the retry limit. The *result* attribute then contains the exception raised by the task.
> >
> > *SUCCESS*
> >
> > > The task executed successfully. The *result* attribute then contains the tasks return value.

status
> The tasks current state.
>
> Possible values includes:
>
> > *PENDING*
> >
> > > The task is waiting for execution.
> >
> > *STARTED*
> >
> > > The task has been started.

RETRY

The task is to be retried, possibly because of failure.

FAILURE

The task raised an exception, or has exceeded the retry limit. The `result` attribute then contains the exception raised by the task.

SUCCESS

The task executed successfully. The `result` attribute then contains the tasks return value.

successful()
Return `True` if the task executed successfully.

supports_native_join

task_id
Compat. alias to `id`.

then(*callback*, *on_error=None*, *weak=False*)

throw(**args*, ***kwargs*)

traceback
Get the traceback of a failed task.

wait(*timeout=None*, *propagate=True*, *interval=0.5*, *no_ack=True*, *follow_parents=True*, *callback=None*, *on_message=None*, *on_interval=None*, *disable_sync_subtasks=True*, *EXCEPTION_STATES=frozenset({'FAILURE', 'REVOKED', 'RETRY'})*, *PROPA-GATE_STATES=frozenset({'FAILURE', 'REVOKED'})*)
Wait until task is ready, and return its result.

> **Warning:** Waiting for tasks within a task may lead to deadlocks. Please read *Avoid launching synchronous subtasks*.

> **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.

Parameters

- **timeout** (`float`) – How long to wait, in seconds, before the operation times out.

- **propagate** (`bool`) – Re-raise exception if the task failed.

- **interval** (`float`) – Time to wait (in seconds) before retrying to retrieve the result. Note that this does not have any effect when using the RPC/redis result store backends, as they don't use polling.

- **no_ack** (`bool`) – Enable amqp no ack (automatically acknowledge message). If this is `False` then the message will **not be acked**.

- **follow_parents** (`bool`) – Re-raise any exception raised by parent tasks.

- **disable_sync_subtasks** (`bool`) – Disable tasks to wait for sub tasks this is the default configuration. CAUTION do not enable this unless you must.

Raises

- *celery.exceptions.TimeoutError* – if *timeout* isn't `None` and the result does not arrive within *timeout* seconds.

- `Exception` – If the remote call raised an exception then that exception will be re-raised in the caller process.

worker

class celery.result.**ResultSet**(*results*, *app=None*, *ready_barrier=None*, ***kwargs*)
    A collection of results.

> **Parameters results** (*Sequence[AsyncResult]*) – List of result instances.

add(*result*)
    Add *AsyncResult* as a new member of the set.

    Does nothing if the result is already a member.

app

backend

clear()
    Remove all results from this set.

completed_count()
    Task completion count.

> **Returns** the number of tasks completed.

> **Return type** int

discard(*result*)
    Remove result from the set if it is a member.

    Does nothing if it's not a member.

failed()
    Return true if any of the tasks failed.

> **Returns**

> > **true if one of the tasks failed.** (i.e., raised an exception)

> **Return type** bool

forget()
    Forget about (and possible remove the result of) all the tasks.

get(*timeout=None*, *propagate=True*, *interval=0.5*, *callback=None*, *no_ack=True*, *on_message=None*, *disable_sync_subtasks=True*, *on_interval=None*)
    See *join()*.

    This is here for API compatibility with *AsyncResult*, in addition it uses *join_native()* if available for the current result backend.

iter_native(*timeout=None*, *interval=0.5*, *no_ack=True*, *on_message=None*, *on_interval=None*)
    Backend optimized version of *iterate()*.

    New in version 2.2.

    Note that this does not support collecting the results for different task types using different backends.

    This is currently only supported by the amqp, Redis and cache result backends.

iterate (*timeout=None*, *propagate=True*, *interval=0.5*)
Deprecated method, use `get()` with a callback argument.

join (*timeout=None*, *propagate=True*, *interval=0.5*, *callback=None*, *no_ack=True*, *on_message=None*,
disable_sync_subtasks=True, *on_interval=None*)
Gather the results of all tasks as a list in order.

Note: This can be an expensive operation for result store backends that must resort to polling (e.g., database).

You should consider using `join_native()` if your backend supports it.

Warning: Waiting for tasks within a task may lead to deadlocks. Please see *Avoid launching synchronous subtasks*.

Parameters

- **timeout** (*float*) – The number of seconds to wait for results before the operation times out.

- **propagate** (*bool*) – If any of the tasks raises an exception, the exception will be reraised when this flag is set.

- **interval** (*float*) – Time to wait (in seconds) before retrying to retrieve a result from the set. Note that this does not have any effect when using the amqp result store backend, as it does not use polling.

- **callback** (*Callable*) – Optional callback to be called for every result received. Must have signature `(task_id, value)` No results will be returned by this function if a callback is specified. The order of results is also arbitrary when a callback is used. To get access to the result object for a particular id you'll have to generate an index first: `index = {r.id:  r for r in gres.results.values()}` Or you can create new result objects on the fly: `result = app.AsyncResult(task_id)` (both will take advantage of the backend cache anyway).

- **no_ack** (*bool*) – Automatic message acknowledgment (Note that if this is set to `False` then the messages *will not be acknowledged*).

- **disable_sync_subtasks** (*bool*) – Disable tasks to wait for sub tasks this is the default configuration. CAUTION do not enable this unless you must.

Raises `celery.exceptions.TimeoutError` – if `timeout` isn't `None` and the operation takes longer than `timeout` seconds.

join_native (*timeout=None*, *propagate=True*, *interval=0.5*, *callback=None*, *no_ack=True*,
on_message=None, *on_interval=None*, *disable_sync_subtasks=True*)
Backend optimized version of `join()`.

New in version 2.2.

Note that this does not support collecting the results for different task types using different backends.

This is currently only supported by the amqp, Redis and cache result backends.

maybe_reraise (*callback=None*, *propagate=True*)

maybe_throw (*callback=None*, *propagate=True*)

---

ready()
> Did all of the tasks complete? (either by success of failure).
>
> > **Returns** true if all of the tasks have been executed.
> >
> > **Return type** [bool](#)

remove(*result*)
> Remove result from the set; it must be a member.
>
> > **Raises** [KeyError](#) – if the result isn't a member.

results = None
> List of results in in the set.

revoke(*connection=None*, *terminate=False*, *signal=None*, *wait=False*, *timeout=None*)
> Send revoke signal to all workers for all tasks in the set.
>
> > **Parameters**
> >
> > - **terminate** ([bool](#)) – Also terminate the process currently working on the task (if any).
> >
> > - **signal** ([str](#)) – Name of signal to send to process if terminate. Default is TERM.
> >
> > - **wait** ([bool](#)) – Wait for replies from worker. The timeout argument specifies the number of seconds to wait. Disabled by default.
> >
> > - **timeout** ([float](#)) – Time in seconds to wait for replies when the wait argument is enabled.

successful()
> Return true if all tasks successful.
>
> > **Returns**
> >
> > > **true if all of the tasks finished** successfully (i.e. didn't raise an exception).
> >
> > **Return type** [bool](#)

supports_native_join

then(*callback*, *on_error=None*, *weak=False*)

update(*results*)
> Extend from iterable of results.

waiting()
> Return true if any of the tasks are incomplete.
>
> > **Returns**
> >
> > > **true if one of the tasks are still** waiting for execution.
> >
> > **Return type** [bool](#)

class celery.result.**GroupResult**(*id=None*, *results=None*, *parent=None*, ***kwargs*)
> Like [ResultSet](#), but with an associated id.
>
> This type is returned by [group](#).
>
> It enables inspection of the tasks state and return values as a single entity.
>
> > **Parameters**
> >
> > - **id** ([str](#)) – The id of the group.
> >
> > - **results** (*Sequence[AsyncResult]*) – List of result instances.

- **parent** (`ResultBase`) – Parent result of this group.

as_tuple()

children

delete(*backend=None*)
    Remove this result if it was previously saved.

id = None
    The UUID of the group.

classmethod restore(*id*, *backend=None*, *app=None*)
    Restore previously saved group result.

results = None
    List/iterator of results in the group

save(*backend=None*)
    Save group-result for later retrieval using `restore()`.

Example

```
>>> def save_and_restore(result):
...     result.save()
...     result = GroupResult.restore(result.id)
```

class celery.result.**EagerResult**(*id*, *ret_value*, *state*, *traceback=None*)
    Result that we know has already been executed.

forget()
    Forget the result of this task and its parents.

get(*timeout=None*, *propagate=True*, *disable_sync_subtasks=True*, ***kwargs*)
    Wait until task is ready, and return its result.

> **Warning:** Waiting for tasks within a task may lead to deadlocks. Please read *Avoid launching synchronous subtasks*.

> **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.

> **Parameters**
>
> - **timeout** (`float`) – How long to wait, in seconds, before the operation times out.
>
> - **propagate** (`bool`) – Re-raise exception if the task failed.
>
> - **interval** (`float`) – Time to wait (in seconds) before retrying to retrieve the result. Note that this does not have any effect when using the RPC/redis result store backends, as they don't use polling.
>
> - **no_ack** (`bool`) – Enable amqp no ack (automatically acknowledge message). If this is `False` then the message will **not be acked**.
>
> - **follow_parents** (`bool`) – Re-raise any exception raised by parent tasks.

- **disable_sync_subtasks** (*bool*) – Disable tasks to wait for sub tasks this is the default configuration. CAUTION do not enable this unless you must.

Raises

- *celery.exceptions.TimeoutError* – if *timeout* isn't None and the result does not arrive within *timeout* seconds.

- Exception – If the remote call raised an exception then that exception will be re-raised in the caller process.

ready()
> Return True if the task has executed.

> If the task is still running, pending, or is waiting for retry then False is returned.

result
> The tasks return value.

revoke(**args*, ***kwargs*)
> Send revoke signal to all workers.

> Any worker receiving the task, or having reserved the task, *must* ignore it.

> **Parameters**

> - **terminate** (*bool*) – Also terminate the process currently working on the task (if any).

> - **signal** (*str*) – Name of signal to send to process if terminate. Default is TERM.

> - **wait** (*bool*) – Wait for replies from workers. The timeout argument specifies the seconds to wait. Disabled by default.

> - **timeout** (*float*) – Time in seconds to wait for replies when wait is enabled.

state
> The tasks state.

status
> The tasks state.

supports_native_join

then(*callback*, *on_error=None*, *weak=False*)

traceback
> The traceback if the task failed.

wait(*timeout=None*, *propagate=True*, *disable_sync_subtasks=True*, ***kwargs*)
> Wait until task is ready, and return its result.

> **Warning:** Waiting for tasks within a task may lead to deadlocks. Please read *Avoid launching synchronous subtasks*.

> **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.

> **Parameters**

- **timeout** (*float*) – How long to wait, in seconds, before the operation times out.

- **propagate** (*bool*) – Re-raise exception if the task failed.

- **interval** (*float*) – Time to wait (in seconds) before retrying to retrieve the result. Note that this does not have any effect when using the RPC/redis result store backends, as they don't use polling.

- **no_ack** (*bool*) – Enable amqp no ack (automatically acknowledge message). If this is `False` then the message will **not be acked**.

- **follow_parents** (*bool*) – Re-raise any exception raised by parent tasks.

- **disable_sync_subtasks** (*bool*) – Disable tasks to wait for sub tasks this is the default configuration. CAUTION do not enable this unless you must.

Raises

- *celery.exceptions.TimeoutError* – if *timeout* isn't `None` and the result does not arrive within *timeout* seconds.

- `Exception` – If the remote call raised an exception then that exception will be re-raised in the caller process.

celery.result.**result_from_tuple**(*r*, *app=None*)
    Deserialize result from tuple.

3.11.17 `celery.schedules`

Schedules define the intervals at which periodic tasks run.

exception celery.schedules.**ParseException**
    Raised by *crontab_parser* when the input can't be parsed.

class celery.schedules.**schedule**(*run_every=None*, *relative=False*, *nowfun=None*, *app=None*)
    Schedule for periodic task.

Parameters

- **run_every** (*float, timedelta*) – Time interval.

- **relative** (*bool*) – If set to True the run time will be rounded to the resolution of the interval.

- **nowfun** (*Callable*) – Function returning the current date and time (*datetime*).

- **app** (*Celery*) – Celery app instance.

human_seconds

is_due(*last_run_at*)
    Return tuple of (is_due, next_time_to_check).

Notes

- next time to check is in seconds.

- **(True, 20), means the task should be run now, and the next** time to check is in 20 seconds.

- (False, 12.3), means the task is not due, but that the scheduler should check again in 12.3 seconds.

The next time to check is used to save energy/CPU cycles, it does not need to be accurate but will influence the precision of your schedule. You must also keep in mind the value of *beat_max_loop_interval*, that decides the maximum number of seconds the scheduler can sleep between re-checking the periodic task intervals. So if you have a task that changes schedule at run-time then your next_run_at check will decide how long it will take before a change to the schedule takes effect. The max loop interval takes precedence over the next check at value returned.

Scheduler max interval variance

The default max loop interval may vary for different schedulers. For the default scheduler the value is 5 minutes, but for example the django-celery-beat database scheduler the value is 5 seconds.

---

relative = False

remaining_estimate(*last_run_at*)

seconds

class celery.schedules.**crontab**(*minute='*'*, *hour='*'*, *day_of_week='*'*, *day_of_month='*'*, *month_of_year='*'*, ***kwargs*)

Crontab schedule.

A Crontab can be used as the run_every value of a periodic task entry to add *crontab(5)*-like scheduling.

Like a *cron(5)*-job, you can specify units of time of when you'd like the task to execute. It's a reasonably complete implementation of **cron**'s features, so it should provide a fair degree of scheduling needs.

You can specify a minute, an hour, a day of the week, a day of the month, and/or a month in the year in any of the following formats:

minute

- A (list of) integers from 0-59 that represent the minutes of an hour of when execution should occur; or

- A string representing a Crontab pattern. This may get pretty advanced, like minute='*/15' (for every quarter) or minute='1,13,30-45,50-59/2'.

hour

- A (list of) integers from 0-23 that represent the hours of a day of when execution should occur; or

- A string representing a Crontab pattern. This may get pretty advanced, like hour='*/3' (for every three hours) or hour='0,8-17/2' (at midnight, and every two hours during office hours).

day_of_week

- A (list of) integers from 0-6, where Sunday = 0 and Saturday = 6, that represent the days of a week that execution should occur.

- A string representing a Crontab pattern. This may get pretty advanced, like day_of_week='mon-fri' (for weekdays only). (Beware that day_of_week='*/2' does not literally mean 'every two days', but 'every day that is divisible by two'!)

day_of_month

- A (list of) integers from 1-31 that represents the days of the month that execution should occur.

- A string representing a Crontab pattern. This may get pretty advanced, such as day_of_month='2-30/2' (for every even numbered day) or day_of_month='1-7, 15-21' (for the first and third weeks of the month).

month_of_year

- A (list of) integers from 1-12 that represents the months of the year during which execution can occur.

- A string representing a Crontab pattern. This may get pretty advanced, such as `month_of_year='*/3'` (for the first month of every quarter) or `month_of_year='2-12/2'` (for every even numbered month).

nowfun
> Function returning the current date and time (`datetime`).

app
> The Celery app instance.

It's important to realize that any day on which execution should occur must be represented by entries in all three of the day and month attributes. For example, if `day_of_week` is 0 and `day_of_month` is every seventh day, only months that begin on Sunday and are also in the `month_of_year` attribute will have execution events. Or, `day_of_week` is 1 and `day_of_month` is '1-7,15-21' means every first and third Monday of every month present in `month_of_year`.

is_due(*last_run_at*)
> Return tuple of (`is_due, next_time_to_run`).

Note: Next time to run is in seconds.

> **SeeAlso:** `celery.schedules.schedule.is_due()` for more information.

remaining_delta(*last_run_at*, *tz=None*, *ffwd=<class 'celery.utils.time.ffwd'>*)

remaining_estimate(*last_run_at*, *ffwd=<class 'celery.utils.time.ffwd'>*)
> Estimate of next run time.

> Returns when the periodic task should run next as a `timedelta`.

class celery.schedules.**crontab_parser**(*max_=60*, *min_=0*)
> Parser for Crontab expressions.

> Any expression of the form 'groups' (see BNF grammar below) is accepted and expanded to a set of numbers. These numbers represent the units of time that the Crontab needs to run on:

```
digit   :: '0'..'9'
dow     :: 'a'..'z'
number  :: digit+ | dow+
steps   :: number
range   :: number ( '-' number ) ?
numspec :: '*' | range
expr    :: numspec ( '/' steps ) ?
groups  :: expr ( ',' expr ) *
```

> The parser is a general purpose one, useful for parsing hours, minutes and day of week expressions. Example usage:

```
>>> minutes = crontab_parser(60).parse('*/15')
[0, 15, 30, 45]
>>> hours = crontab_parser(24).parse('*/4')
[0, 4, 8, 12, 16, 20]
>>> day_of_week = crontab_parser(7).parse('*')
[0, 1, 2, 3, 4, 5, 6]
```

> It can also parse day of month and month of year expressions if initialized with a minimum of 1. Example usage:

```
>>> days_of_month = crontab_parser(31, 1).parse('*/3')
[1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31]
>>> months_of_year = crontab_parser(12, 1).parse('*/2')
[1, 3, 5, 7, 9, 11]
>>> months_of_year = crontab_parser(12, 1).parse('2-12/2')
[2, 4, 6, 8, 10, 12]
```

The maximum possible expanded value returned is found by the formula:

$$max_+min_-1$$

exception ParseException
 Raised by *crontab_parser* when the input can't be parsed.

parse(*spec*)

celery.schedules.**maybe_schedule**(*s*, *relative=False*, *app=None*)
 Return schedule from number, timedelta, or actual schedule.

class celery.schedules.**solar**(*event*, *lat*, *lon*, ***kwargs*)
 Solar event.

A solar event can be used as the run_every value of a periodic task entry to schedule based on certain solar events.

Notes

Available event valus are:

- dawn_astronomical
- dawn_nautical
- dawn_civil
- sunrise
- solar_noon
- sunset
- dusk_civil
- dusk_nautical
- dusk_astronomical

 Parameters

- **event** (*str*) – Solar event that triggers this task. See note for available values.
- **lat** (*int*) – The latitude of the observer.
- **lon** (*int*) – The longitude of the observer.
- **nowfun** (*Callable*) – Function returning the current date and time as a class:~*datetime.datetime*.
- **app** (*Celery*) – Celery app instance.

is_due(*last_run_at*)
 Return tuple of (is_due, next_time_to_run).

Note: next time to run is in seconds.

See also:

`celery.schedules.schedule.is_due()` for more information.

remaining_estimate(*last_run_at*)
Return estimate of next time to run.

> **Returns**
>
> > **when the periodic task should** run next, or if it shouldn't run today (e.g., the sun does not rise today), returns the time when the next check should take place.
>
> **Return type** timedelta

3.11.18 `celery.signals`

Celery Signals.

This module defines the signals (Observer pattern) sent by both workers and clients.

Functions can be connected to these signals, and connected functions are called whenever a signal is called.

See also:

Signals for more information.

3.11.19 `celery.security`

Message Signing Serializer.

celery.security.**setup_security**(*allowed_serializers=None*, *key=None*, *cert=None*, *store=None*, *digest=None*, *serializer='json'*, *app=None*)
See `Celery.setup_security()`.

3.11.20 `celery.utils.debug`

- *Sampling Memory Usage*
- *API Reference*

Sampling Memory Usage

This module can be used to diagnose and sample the memory usage used by parts of your application.

For example, to sample the memory usage of calling tasks you can do this:

```
from celery.utils.debug import sample_mem, memdump

from tasks import add
```

(continues on next page)

```python
try:
    for i in range(100):
        for j in range(100):
            add.delay(i, j)
        sample_mem()
finally:
    memdump()
```

API Reference

Utilities for debugging memory usage, blocking calls, etc.

celery.utils.debug.**sample_mem**()
> Sample RSS memory usage.
>
> Statistics can then be output by calling *memdump()*.

celery.utils.debug.**memdump**(*samples=10*, *file=None*)
> Dump memory statistics.
>
> Will print a sample of all RSS memory samples added by calling *sample_mem()*, and in addition print used RSS memory after `gc.collect()`.

celery.utils.debug.**sample**(*x*, *n*, *k=0*)
> Given a list *x* a sample of length `n` of that list is returned.
>
> For example, if *n* is 10, and *x* has 100 items, a list of every tenth. item is returned.
>
> `k` can be used as offset.

celery.utils.debug.**mem_rss**()
> Return RSS memory usage as a humanized string.

celery.utils.debug.**ps**()
> Return the global `psutil.Process` instance.

> **Note:** Returns `None` if psutil is not installed.

3.11.21 `celery.exceptions`

- *Error Hierarchy*

Celery error types.

Error Hierarchy

- **Exception**
    - *celery.exceptions.CeleryError*
        * *ImproperlyConfigured*
        * *SecurityError*

– **kombu.exceptions.KombuError**

  * *OperationalError*

  Raised when a transport connection error occurs while sending a message (be it a task, remote control command error).

  Note: This exception does not inherit from *CeleryError*.

– **billiard errors (prefork pool)**

exception `celery.exceptions.`**`CeleryWarning`**
    Base class for all Celery warnings.

exception `celery.exceptions.`**`AlwaysEagerIgnored`**
    send_task ignores *task_always_eager* option.

exception `celery.exceptions.`**`DuplicateNodenameWarning`**
    Multiple workers are using the same nodename.

exception `celery.exceptions.`**`FixupWarning`**
    Fixup related warning.

exception `celery.exceptions.`**`NotConfigured`**
    Celery hasn't been configured, as no config module has been found.

exception `celery.exceptions.`**`CeleryError`**
    Base class for all Celery errors.

exception `celery.exceptions.`**`ImproperlyConfigured`**
    Celery is somehow improperly configured.

exception `celery.exceptions.`**`SecurityError`**
    Security related exception.

exception `celery.exceptions.`**`OperationalError`**
    Recoverable message transport connection error.

exception `celery.exceptions.`**`TaskPredicate`**
    Base class for task-related semi-predicates.

exception `celery.exceptions.`**`Ignore`**
    A task can raise this to ignore doing state updates.

exception `celery.exceptions.`**`Reject`**(*reason=None*, *requeue=False*)
    A task can raise this if it wants to reject/re-queue the message.

exception `celery.exceptions.`**`Retry`**(*message=None*, *exc=None*, *when=None*, ***kwargs*)
    The task is to be retried later.

   exc = None
        Exception (if any) that caused the retry to happen.

   humanize()

   message = None
        Optional message describing context of retry.

   when = None
        Time of retry (ETA), either `numbers.Real` or `datetime`.

exception `celery.exceptions.`**`TaskError`**
    Task related errors.

exception `celery.exceptions.`**`QueueNotFound`**
    Task routed to a queue not in `conf.queues`.

exception `celery.exceptions.`**`IncompleteStream`**
    Found the end of a stream of data, but the data isn't complete.

exception `celery.exceptions.`**`NotRegistered`**
    The task ain't registered.

exception `celery.exceptions.`**`AlreadyRegistered`**
    The task is already registered.

exception `celery.exceptions.`**`TimeoutError`**
    The operation timed out.

exception `celery.exceptions.`**`MaxRetriesExceededError`**(**args*, ***kwargs*)
    The tasks max restart limit has been exceeded.

exception `celery.exceptions.`**`TaskRevokedError`**
    The task has been revoked, so no result available.

exception `celery.exceptions.`**`InvalidTaskError`**
    The task has invalid data or ain't properly constructed.

exception `celery.exceptions.`**`ChordError`**
    A task part of the chord raised an exception.

exception `celery.exceptions.`**`SoftTimeLimitExceeded`**
    The soft time limit has been exceeded. This exception is raised to give the task a chance to clean up.

exception `celery.exceptions.`**`TimeLimitExceeded`**
    The time limit has been exceeded and the job has been terminated.

exception `celery.exceptions.`**`WorkerLostError`**
    The worker processing a job has exited prematurely.

exception `celery.exceptions.`**`Terminated`**
    The worker processing a job has been terminated by user request.

exception `celery.exceptions.`**`CPendingDeprecationWarning`**
    Warning of pending deprecation.

exception `celery.exceptions.`**`CDeprecationWarning`**
    Warning of deprecation.

exception `celery.exceptions.`**`WorkerShutdown`**
    Signals that the worker should perform a warm shutdown.

exception `celery.exceptions.`**`WorkerTerminate`**
    Signals that the worker should terminate immediately.

3.11.22 `celery.loaders`

Get loader by name.

Loaders define how configuration is read, what happens when workers start, when tasks are executed and so on.

`celery.loaders.`**`get_loader_cls`**(*loader*)
    Get loader class by name/alias.

3.11.23 `celery.loaders.app`

The default loader used with custom app instances.

class `celery.loaders.app.`**`AppLoader`**(*app*, ***kwargs*)
    Default loader used when an app is specified.

3.11.24 `celery.loaders.default`

The default loader used when no custom app has been initialized.

class `celery.loaders.default.`**Loader**(*app*, ***kwargs*)
  The loader used by the default app.

  read_configuration(*fail_silently=True*)
    Read configuration from `celeryconfig.py`.

  setup_settings(*settingsdict*)

3.11.25 `celery.loaders.base`

Loader base class.

class `celery.loaders.base.`**BaseLoader**(*app*, ***kwargs*)
  Base class for loaders.

  Loaders handles,

  - Reading celery client/worker configurations.
  - **What happens when a task starts?** See *on_task_init()*.
  - **What happens when the worker starts?** See *on_worker_init()*.
  - **What happens when the worker shuts down?** See *on_worker_shutdown()*.
  - What modules are imported to find tasks?

  autodiscover_tasks(*packages*, *related_name='tasks'*)

  builtin_modules = frozenset()

  cmdline_config_parser(*args*, *namespace='celery'*, *re_type=re.compile('\\(\(\\w+\)\\)')*, *extra_types=None*, *override_types=None*)

  conf
    Loader configuration.

  config_from_object(*obj*, *silent=False*)

  configured = False

  default_modules

  find_module(*module*)

  import_default_modules()

  import_from_cwd(*module*, *imp=None*, *package=None*)

  import_module(*module*, *package=None*)

  import_task_module(*module*)

  init_worker()

  init_worker_process()

  now(*utc=True*)

  on_process_cleanup()
    Called after a task is executed.

  on_task_init(*task_id*, *task*)
    Called before a task is executed.

  on_worker_init()
    Called when the worker (**celery worker**) starts.

on_worker_process_init()
> Called when a child process starts.

on_worker_shutdown()
> Called when the worker (**celery worker**) shuts down.

override_backends = {}

read_configuration(*env='CELERY_CONFIG_MODULE'*)

shutdown_worker()

worker_initialized = False

- *States*
- *Sets*
  - *READY_STATES*
  - *UNREADY_STATES*
  - *EXCEPTION_STATES*
  - *PROPAGATE_STATES*
  - *ALL_STATES*
- *Misc*

Built-in task states.

3.11.26 States

See *States*.

3.11.27 Sets

READY_STATES

Set of states meaning the task result is ready (has been executed).

UNREADY_STATES

Set of states meaning the task result is not ready (hasn't been executed).

EXCEPTION_STATES

Set of states meaning the task returned an exception.

PROPAGATE_STATES

Set of exception states that should propagate exceptions to the user.

ALL_STATES

Set of all possible states.

3.11.28 Misc

`celery.states.`**`PENDING = 'PENDING'`**
    Task state is unknown (assumed pending since you know the id).

`celery.states.`**`RECEIVED = 'RECEIVED'`**
    Task was received by a worker (only used in events).

`celery.states.`**`STARTED = 'STARTED'`**
    Task was started by a worker (*task_track_started*).

`celery.states.`**`SUCCESS = 'SUCCESS'`**
    Task succeeded

`celery.states.`**`FAILURE = 'FAILURE'`**
    Task failed

`celery.states.`**`REVOKED = 'REVOKED'`**
    Task was revoked.

`celery.states.`**`RETRY = 'RETRY'`**
    Task is waiting for retry.

`celery.states.`**`precedence`**(*state*)
    Get the precedence index for state.

    Lower index means higher precedence.

class `celery.states.`**`state`**
    Task state.

    State is a subclass of `str`, implementing comparison methods adhering to state precedence rules:

    ```
    >>> from celery.states import state, PENDING, SUCCESS

    >>> state(PENDING) < state(SUCCESS)
    True
    ```

    Any custom state is considered to be lower than *FAILURE* and *SUCCESS*, but higher than any of the other built-in states:

    ```
    >>> state('PROGRESS') > state(STARTED)
    True

    >>> state('PROGRESS') > state('SUCCESS')
    False
    ```

3.11.29 `celery.contrib.abortable`

- *Abortable tasks overview*
    - *Usage example*

Abortable Tasks.

Abortable tasks overview

For long-running `Task`'s, it can be desirable to support aborting during execution. Of course, these tasks should be built to support abortion specifically.

The *AbortableTask* serves as a base class for all `Task` objects that should support abortion by producers.

- Producers may invoke the `abort()` method on *AbortableAsyncResult* instances, to request abortion.

- Consumers (workers) should periodically check (and honor!) the `is_aborted()` method at controlled points in their task's `run()` method. The more often, the better.

The necessary intermediate communication is dealt with by the *AbortableTask* implementation.

Usage example

In the consumer:

```python
from __future__ import absolute_import

from celery.contrib.abortable import AbortableTask
from celery.utils.log import get_task_logger

from proj.celery import app

logger = get_logger(__name__)

@app.task(bind=True, base=AbortableTask)
def long_running_task(self):
    results = []
    for i in range(100):
        # check after every 5 iterations...
        # (or alternatively, check when some timer is due)
        if not i % 5:
            if self.is_aborted():
                # respect aborted state, and terminate gracefully.
                logger.warning('Task aborted')
                return
            value = do_something_expensive(i)
            results.append(y)
    logger.info('Task complete')
    return results
```

In the producer:

```python
from __future__ import absolute_import

import time

from proj.tasks import MyLongRunningTask

def myview(request):
    # result is of type AbortableAsyncResult
    result = long_running_task.delay()
```

(continues on next page)

```
    # abort the task after 10 seconds
    time.sleep(10)
    result.abort()
```

After the *result.abort()* call, the task execution isn't aborted immediately. In fact, it's not guaranteed to abort at all. Keep checking *result.state* status, or call *result.get(timeout=)* to have it block until the task is finished.

Note: In order to abort tasks, there needs to be communication between the producer and the consumer. This is currently implemented through the database backend. Therefore, this class will only work with the database backends.

class celery.contrib.abortable.**AbortableAsyncResult**(*id*, *backend=None*, *task_name=None*, *app=None*, *parent=None*)

    Represents an abortable result.

    Specifically, this gives the *AsyncResult* a *abort()* method, that sets the state of the underlying Task to *'ABORTED'*.

    abort()

        Set the state of the task to ABORTED.

        Abortable tasks monitor their state at regular intervals and terminate execution if so.

        Warning: Be aware that invoking this method does not guarantee when the task will be aborted (or even if the task will be aborted at all).

    is_aborted()

        Return True if the task is (being) aborted.

class celery.contrib.abortable.**AbortableTask**

    Task that can be aborted.

    This serves as a base class for all Task's that support aborting during execution.

    All subclasses of *AbortableTask* must call the *is_aborted()* method periodically and act accordingly when the call evaluates to True.

    AsyncResult(*task_id*)

        Return the accompanying AbortableAsyncResult instance.

    abstract = True

    is_aborted(***kwargs*)

        Return true if task is aborted.

        Checks against the backend whether this *AbortableAsyncResult* is ABORTED.

        Always return False in case the *task_id* parameter refers to a regular (non-abortable) Task.

        Be aware that invoking this method will cause a hit in the backend (for example a database query), so find a good balance between calling it regularly (for responsiveness), but not too often (for performance).

3.11.30 `celery.contrib.migrate`

Message migration tools (Broker <-> Broker).

exception `celery.contrib.migrate.`**`StopFiltering`**
    Semi-predicate used to signal filter stop.

class `celery.contrib.migrate.`**`State`**
    Migration progress state.

    `count = 0`

    `filtered = 0`

    `strtotal`

    `total_apx = 0`

`celery.contrib.migrate.`**`republish`**(*producer*, *message*, *exchange=None*, *routing_key=None*, *re-move_props=None*)
    Republish message.

`celery.contrib.migrate.`**`migrate_task`**(*producer*, *body_*, *message*, *queues=None*)
    Migrate single task message.

`celery.contrib.migrate.`**`migrate_tasks`**(*source*, *dest*, *migrate=<function migrate_task>*, *app=None*, *queues=None*, ***kwargs*)
    Migrate tasks from one broker to another.

`celery.contrib.migrate.`**`move`**(*predicate*, *connection=None*, *exchange=None*, *routing_key=None*, *source=None*, *app=None*, *callback=None*, *limit=None*, *trans-form=None*, ***kwargs*)
    Find tasks by filtering them and move the tasks to a new queue.

        Parameters

- **`predicate`** (`Callable`) – Filter function used to decide the messages to move. Must accept the standard signature of (`body`, `message`) used by Kombu consumer callbacks. If the predicate wants the message to be moved it must return either:

    1) a tuple of (`exchange`, `routing_key`), or

    2) a `Queue` instance, or

    3) **any other true value means the specified** `exchange` and `routing_key` arguments will be used.

- **`connection`** (`kombu.Connection`) – Custom connection to use.

- **`source`** – List[Union[str, kombu.Queue]]: Optional list of source queues to use instead of the default (queues in `task_queues`). This list can also contain `Queue` instances.

- **`exchange`** (`str`, `kombu.Exchange`) – Default destination exchange.

- **`routing_key`** (`str`) – Default destination routing key.

- **`limit`** (`int`) – Limit number of messages to filter.

- **`callback`** (`Callable`) – Callback called after message moved, with signature (`state`, `body`, `message`).

- **`transform`** (`Callable`) – Optional function to transform the return value (destination) of the filter function.

    Also supports the same keyword arguments as *start_filter()*.

    To demonstrate, the *move_task_by_id()* operation can be implemented like this:

```
def is_wanted_task(body, message):
    if body['id'] == wanted_id:
        return Queue('foo', exchange=Exchange('foo'),
                     routing_key='foo')

move(is_wanted_task)
```

or with a transform:

```
def transform(value):
    if isinstance(value, string_t):
        return Queue(value, Exchange(value), value)
    return value

move(is_wanted_task, transform=transform)
```

Note: The predicate may also return a tuple of (exchange, routing_key) to specify the destination to where the task should be moved, or a Queue instance. Any other true value means that the task will be moved to the default exchange/routing_key.

---

celery.contrib.migrate.**task_id_eq**(*task_id*, *body*, *message*)
> Return true if task id equals task_id'.

celery.contrib.migrate.**task_id_in**(*ids*, *body*, *message*)
> Return true if task id is member of set ids'.

celery.contrib.migrate.**start_filter**(*app*, *conn*, *filter*, *limit=None*, *timeout=1.0*, *ack_messages=False*, *tasks=None*, *queues=None*, *callback=None*, *forever=False*, *on_declare_queue=None*, *consume_from=None*, *state=None*, *accept=None*, ***kwargs*)
> Filter tasks.

celery.contrib.migrate.**move_task_by_id**(*task_id*, *dest*, ***kwargs*)
> Find a task by id and move it to another queue.

> > **Parameters**

> > > • **task_id** (*str*) – Id of task to find and move.

> > > • **dest** – (str, kombu.Queue): Destination queue.

> > > • **transform** (*Callable*) – Optional function to transform the return value (destination) of the filter function.

> > > • ****kwargs** (*Any*) – Also supports the same keyword arguments as *move()*.

celery.contrib.migrate.**move_by_idmap**(*map*, ***kwargs*)
> Move tasks by matching from a task_id: queue mapping.

> Where queue is a queue to move the task to.

Example

```
>>> move_by_idmap({
...     '5bee6e82-f4ac-468e-bd3d-13e8600250bc': Queue('name'),
...     'ada8652d-aef3-466b-abd2-becdaf1b82b3': Queue('name'),
```

(continues on next page)

```
...        '3a2b140d-7db1-41ba-ac90-c36a0ef4ab1f': Queue('name')},
...     queues=['hipri'])
```

celery.contrib.migrate.**move_by_taskmap**(*map*, ***kwargs*)
    Move tasks by matching from a `task_name:   queue` mapping.

    `queue` is the queue to move the task to.

Example

```
>>> move_by_taskmap({
...     'tasks.add': Queue('name'),
...     'tasks.mul': Queue('name'),
... })
```

celery.contrib.migrate.**move_direct**(*predicate*, *connection=None*, *exchange=None*, *rout-ing_key=None*, *source=None*, *app=None*, *callback=None*, *limit=None*, ***, *transform=<function worker_direct>*, ***kwargs*)
    Find tasks by filtering them and move the tasks to a new queue.

> **Parameters**
>
> - **predicate** (`Callable`) – Filter function used to decide the messages to move. Must accept the standard signature of (`body, message`) used by Kombu consumer callbacks. If the predicate wants the message to be moved it must return either:
>
>   1) a tuple of (`exchange, routing_key`), or
>
>   2) a `Queue` instance, or
>
>   3) **any other true value means the specified** `exchange` and `routing_key` arguments will be used.
>
> - **connection** (`kombu.Connection`) – Custom connection to use.
>
> - **source** – List[Union[str, kombu.Queue]]: Optional list of source queues to use instead of the default (queues in `task_queues`). This list can also contain `Queue` instances.
>
> - **exchange** (`str, kombu.Exchange`) – Default destination exchange.
>
> - **routing_key** (`str`) – Default destination routing key.
>
> - **limit** (`int`) – Limit number of messages to filter.
>
> - **callback** (`Callable`) – Callback called after message moved, with signature (`state, body, message`).
>
> - **transform** (`Callable`) – Optional function to transform the return value (destination) of the filter function.

    Also supports the same keyword arguments as `start_filter()`.

    To demonstrate, the `move_task_by_id()` operation can be implemented like this:

```python
def is_wanted_task(body, message):
    if body['id'] == wanted_id:
        return Queue('foo', exchange=Exchange('foo'),
                     routing_key='foo')
```

```
move(is_wanted_task)
```

or with a transform:

```python
def transform(value):
    if isinstance(value, string_t):
        return Queue(value, Exchange(value), value)
    return value

move(is_wanted_task, transform=transform)
```

Note: The predicate may also return a tuple of (exchange, routing_key) to specify the destination to where the task should be moved, or a Queue instance. Any other true value means that the task will be moved to the default exchange/routing_key.

---

celery.contrib.migrate.**move_direct_by_id**(*task_id*, *dest*, ***kwargs*)
> Find a task by id and move it to another queue.

> > **Parameters**

> > > * **task_id** (*str*) – Id of task to find and move.

> > > * **dest** – (str, kombu.Queue): Destination queue.

> > > * **transform** (*Callable*) – Optional function to transform the return value (destination) of the filter function.

> > > * ****kwargs** (*Any*) – Also supports the same keyword arguments as *move()*.

3.11.31 `celery.contrib.pytest`

> * *API Reference*

API Reference

Fixtures and testing utilities for py.test.

celery.contrib.pytest.**celery_app**(*request*, *celery_config*, *celery_parameters*, *celery_enable_logging*, *use_celery_app_trap*)
> Fixture creating a Celery application instance.

celery.contrib.pytest.**celery_config**()
> Redefine this fixture to configure the test Celery app.

> The config returned by your fixture will then be used to configure the *celery_app()* fixture.

celery.contrib.pytest.**celery_enable_logging**()
> You can override this fixture to enable logging.

celery.contrib.pytest.**celery_includes**()
> You can override this include modules when a worker start.

You can have this return a list of module names to import, these can be task modules, modules registering signals, and so on.

celery.contrib.pytest.**celery_parameters**()
> Redefine this fixture to change the init parameters of test Celery app.
>
> The dict returned by your fixture will then be used as parameters when instantiating *Celery*.

celery.contrib.pytest.**celery_session_app**(*request*, *celery_config*, *celery_parameters*, *celery_enable_logging*, *use_celery_app_trap*)
> Session Fixture: Return app for session fixtures.

celery.contrib.pytest.**celery_session_worker**(*request*, *celery_session_app*, *celery_includes*, *celery_worker_pool*, *celery_worker_parameters*)
> Session Fixture: Start worker that lives throughout test suite.

celery.contrib.pytest.**celery_worker**(*request*, *celery_app*, *celery_includes*, *celery_worker_pool*, *celery_worker_parameters*)
> Fixture: Start worker in a thread, stop it when the test returns.

celery.contrib.pytest.**celery_worker_parameters**()
> Redefine this fixture to change the init parameters of Celery workers.
>
> This can be used e. g. to define queues the worker will consume tasks from.
>
> The dict returned by your fixture will then be used as parameters when instantiating *WorkController*.

celery.contrib.pytest.**celery_worker_pool**()
> You can override this fixture to set the worker pool.
>
> The "solo" pool is used by default, but you can set this to return e.g. "prefork".

celery.contrib.pytest.**depends_on_current_app**(*celery_app*)
> Fixture that sets app as current.

celery.contrib.pytest.**pytest_configure**(*config*)
> Register additional pytest configuration.

celery.contrib.pytest.**use_celery_app_trap**()
> You can override this fixture to enable the app trap.
>
> The app trap raises an exception whenever something attempts to use the current or default apps.

3.11.32 celery.contrib.sphinx

Sphinx documentation plugin used to document tasks.

Introduction

Usage

Add the extension to your docs/conf.py configuration module:

```
extensions = (...,
              'celery.contrib.sphinx')
```

If you'd like to change the prefix for tasks in reference documentation then you can change the celery_task_prefix configuration value:

```
celery_task_prefix = '(task)'  # < default
```

With the extension installed *autodoc* will automatically find task decorated objects (e.g. when using the automodule directive) and generate the correct (as well as add a (task) prefix), and you can also refer to the tasks using *:task:proj.tasks.add* syntax.

Use .. autotask:: to alternatively manually document a task.

class celery.contrib.sphinx.**TaskDirective**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Sphinx task directive.

    get_signature_prefix(*sig*)
        May return a prefix to put before the object name in the signature.

class celery.contrib.sphinx.**TaskDocumenter**(*directive*, *name*, *indent=''*)
Document task definitions.

    classmethod can_document_member(*member*, *membername*, *isattr*, *parent*)
        Called to see if a member can be documented by this documenter.

    check_module()
        Check if *self.object* is really defined in the module given by *self.modname*.

    document_members(*all_members=False*)
        Generate reST for member documentation.

        If *all_members* is True, do all members, else those given by *self.options.members*.

    format_args()
        Format the argument signature of *self.object*.

        Should return None if the object does not have a signature.

celery.contrib.sphinx.**autodoc_skip_member_handler**(*app*, *what*, *name*, *obj*, *skip*, *options*)
Handler for autodoc-skip-member event.

celery.contrib.sphinx.**setup**(*app*)
Setup Sphinx extension.

3.11.33 `celery.contrib.testing.worker`

> • *API Reference*

API Reference

Embedded workers for integration tests.

class celery.contrib.testing.worker.**TestWorkController**(**args*, ***kwargs*)
Worker that can synchronize on being fully started.

    ensure_started()
        Wait for worker to be fully up and running.

> **Warning:** Worker must be started within a thread for this to work, or it will block forever.

> **on_consumer_ready**(*consumer*)
> > Callback called when the Consumer blueprint is fully started.

celery.contrib.testing.worker.**setup_app_for_worker**(*app*, *loglevel*, *logfile*)
> Setup the app to be used for starting an embedded worker.

celery.contrib.testing.worker.**start_worker**(*app*, *concurrency=1*, *pool='solo'*, *loglevel='error'*, *logfile=None*, *perform_ping_check=True*, *ping_task_timeout=10.0*, ***kwargs*)
> Start embedded worker.

> > **Yields** *celery.app.worker.Worker* – worker instance.

3.11.34 `celery.contrib.testing.app`

> - *API Reference*

API Reference

Create Celery app instances used for testing.

celery.contrib.testing.app.**DEFAULT_TEST_CONFIG = {'accept_content': {'json'}, 'broker_hea**
> Contains the default configuration values for the test app.

celery.contrib.testing.app.**TestApp**(*name=None*, *config=None*, *enable_logging=False*, *set_as_current=False*, *log=<class 'celery.contrib.testing.app.UnitLogging'>*, *backend=None*, *broker=None*, ***kwargs*)
> App used for testing.

class celery.contrib.testing.app.**Trap**
> Trap that pretends to be an app but raises an exception instead.

> This to protect from code that does not properly pass app instances, then falls back to the current_app.

class celery.contrib.testing.app.**UnitLogging**(**args*, ***kwargs*)
> Sets up logging for the test application.

celery.contrib.testing.app.**set_trap**(*app*)
> Contextmanager that installs the trap app.

> The trap means that anything trying to use the current or default app will raise an exception.

celery.contrib.testing.app.**setup_default_app**(*app*, *use_trap=False*)
> Setup default app for testing.

> Ensures state is clean after the test returns.

3.11.35 `celery.contrib.testing.manager`

- *API Reference*

API Reference

Integration testing utilities.

class celery.contrib.testing.manager.**Manager**(*app*, ***kwargs*)
    Test helpers for task integration tests.

class celery.contrib.testing.manager.**ManagerMixin**
    Mixin that adds *Manager* capabilities.

    assert_accepted(*ids*, *interval=0.5*, *desc='waiting for tasks to be accepted'*, ***policy*)

    assert_received(*ids*, *interval=0.5*, *desc='waiting for tasks to be received'*, ***policy*)

    assert_result_tasks_in_progress_or_completed(*async_results*, *interval=0.5*, *desc='waiting for tasks to be started or completed'*, ***policy*)

    assert_task_state_from_result(*fun*, *results*, *interval=0.5*, ***policy*)

    assert_task_worker_state(*fun*, *ids*, *interval=0.5*, ***policy*)

    ensure_not_for_a_while(*fun*, *catch*, *desc='thing'*, *max_retries=20*, *interval_start=0.1*, *interval_step=0.02*, *interval_max=1.0*, *emit_warning=False*, ***options*)
        Make sure something does not happen (at least for a while).

    inspect(*timeout=3.0*)

    is_accepted(*ids*, ***kwargs*)

    is_received(*ids*, ***kwargs*)

    static is_result_task_in_progress(*results*, ***kwargs*)

    join(*r*, *propagate=False*, *max_retries=10*, ***kwargs*)

    missing_results(*r*)

    query_task_states(*ids*, *timeout=0.5*)

    query_tasks(*ids*, *timeout=0.5*)

    remark(*s*, *sep='-'*)

    retry_over_time(**args*, ***kwargs*)

    true_or_raise(*fun*, **args*, ***kwargs*)

    wait_for(*fun*, *catch*, *desc='thing'*, *args=()*, *kwargs=None*, *errback=None*, *max_retries=10*, *interval_start=0.1*, *interval_step=0.5*, *interval_max=5.0*, *emit_warning=False*, ***options*)
        Wait for event to happen.

        The *catch* argument specifies the exception that means the event has not happened yet.

exception celery.contrib.testing.manager.**Sentinel**
    Signifies the end of something.

celery.contrib.testing.manager.**humanize_seconds**(*secs*, *prefix=''*, *sep=''*, *now='now'*, ***, *microseconds=True*)
    Show seconds in human form.

    For example, 60 becomes "1 minute", and 7200 becomes "2 hours".

Parameters

- **prefix** (*str*) – can be used to add a preposition to the output (e.g., 'in' will give 'in 1 second', but add nothing to 'now').

- **now** (*str*) – Literal 'now'.

- **microseconds** (*bool*) – Include microseconds.

3.11.36 `celery.contrib.testing.mocks`

- *API Reference*

API Reference

Useful mocks for unit testing.

`celery.contrib.testing.mocks.`**`TaskMessage`**(*name*, *id=None*, *args=()*, *kwargs=None*, *callbacks=None*, *errbacks=None*, *chain=None*, *shadow=None*, *utc=None*, ***options*)

   Create task message in protocol 2 format.

`celery.contrib.testing.mocks.`**`TaskMessage1`**(*name*, *id=None*, *args=()*, *kwargs=None*, *callbacks=None*, *errbacks=None*, *chain=None*, ***options*)

   Create task message in protocol 1 format.

`celery.contrib.testing.mocks.`**`task_message_from_sig`**(*app*, *sig*, *utc=True*, *TaskMessage=<function TaskMessage>*)

   Create task message from *celery.Signature*.

Example

```
>>> m = task_message_from_sig(app, add.s(2, 2))
>>> amqp_client.basic_publish(m, exchange='ex', routing_key='rkey')
```

3.11.37 `celery.contrib.rdb`

Remote Debugger.

Introduction

This is a remote debugger for Celery tasks running in multiprocessing pool workers. Inspired by a lost post on dzone.com.

Usage

```
from celery.contrib import rdb
from celery import task

@task()
def add(x, y):
    result = x + y
    rdb.set_trace()
    return result
```

Environment Variables

CELERY_RDB_HOST

CELERY_RDB_HOST

> Hostname to bind to. Default is '127.0.0.1' (only accessible from localhost).

CELERY_RDB_PORT

CELERY_RDB_PORT

> Base port to bind to. Default is 6899. The debugger will try to find an available port starting from the base port. The selected port will be logged by the worker.

`celery.contrib.rdb.`**`set_trace`**(*frame=None*)
> Set break-point at current location, or a specified frame.

`celery.contrib.rdb.`**`debugger`**()
> Return the current debugger instance, or create if none.

class `celery.contrib.rdb.`**Rdb**(*host='127.0.0.1'*, *port=6899*, *port_search_limit=100*, *port_skew=0*, *out=<_io.TextIOWrapper* *name='<stdout>'* *mode='w' encoding='UTF-8'>*)
> Remote debugger.

3.11.38 `celery.events`

Monitoring Event Receiver+Dispatcher.

Events is a stream of messages sent for certain actions occurring in the worker (and clients if *task_send_sent_event* is enabled), used for monitoring purposes.

`celery.events.`**Event**(*type*, *_fields=None*, *__dict__=<class 'dict'>*, *__now__=<built-in function time>*, ***fields*)
> Create an event.

> ### Notes

> > An event is simply a dictionary: the only required field is `type`. A `timestamp` field will be set to the current time if not provided.

class `celery.events.`**`EventDispatcher`**(*connection=None, hostname=None, enabled=True, channel=None, buffer_while_offline=True, app=None, serializer=None, groups=None, delivery_mode=1, buffer_group=None, buffer_limit=24, on_send_buffered=None*)

Dispatches event messages.

> **Parameters**
>
> > * **connection** (*kombu.Connection*) – Connection to the broker.
> >
> > * **hostname** (*str*) – Hostname to identify ourselves as, by default uses the hostname returned by `anon_nodename()`.
> >
> > * **groups** (*Sequence[str]*) – List of groups to send events for. *send()* will ignore send requests to groups not in this list. If this is `None`, all events will be sent. Example groups include `"task"` and `"worker"`.
> >
> > * **enabled** (*bool*) – Set to `False` to not actually publish any events, making *send()* a no-op.
> >
> > * **channel** (*kombu.Channel*) – Can be used instead of *connection* to specify an exact channel to use when sending events.
> >
> > * **buffer_while_offline** (*bool*) – If enabled events will be buffered while the connection is down. *flush()* must be called as soon as the connection is re-established.

Note: You need to *close()* this after use.

`DISABLED_TRANSPORTS = {'sql'}`

`app = None`

`close`()
> Close the event dispatcher.

`disable`()

`enable`()

`extend_buffer`(*other*)
> Copy the outbound buffer of another instance.

`flush`(*errors=True*, *groups=True*)
> Flush the outbound buffer.

`on_disabled = None`

`on_enabled = None`

`publish`(*type*, *fields*, *producer*, *blind=False*, *Event=<function Event>*, ***kwargs*)
> Publish event using custom `Producer`.
>
> > **Parameters**
> >
> > > * **type** (*str*) – Event type name, with group separated by dash (-). fields: Dictionary of event fields, must be json serializable.
> > >
> > > * **producer** (*kombu.Producer*) – Producer instance to use: only the `publish` method will be called.
> > >
> > > * **retry** (*bool*) – Retry in the event of connection failure.

- **retry_policy** (*Mapping*) – Map of custom retry policy options. See `ensure()`.

- **blind** (*bool*) – Don't set logical clock value (also don't forward the internal logical clock).

- **Event** (*Callable*) – Event type used to create event. Defaults to *Event()*.

- **utcoffset** (*Callable*) – Function returning the current utc offset in hours.

publisher

send(*type*, *blind=False*, *utcoffset=<function utcoffset>*, *retry=False*, *retry_policy=None*, *Event=<function Event>*, ***fields*)
   Send event.

   Parameters

   - **type** (*str*) – Event type name, with group separated by dash (-).

   - **retry** (*bool*) – Retry in the event of connection failure.

   - **retry_policy** (*Mapping*) – Map of custom retry policy options. See `ensure()`.

   - **blind** (*bool*) – Don't set logical clock value (also don't forward the internal logical clock).

   - **Event** (*Callable*) – Event type used to create event, defaults to *Event()*.

   - **utcoffset** (*Callable*) – unction returning the current utc offset in hours.

   - ****fields** (*Any*) – Event fields – must be json serializable.

class celery.events.**EventReceiver**(*channel*, *handlers=None*, *routing_key='#'*, *node_id=None*, *app=None*, *queue_prefix=None*, *accept=None*, *queue_ttl=None*, *queue_expires=None*)
   Capture events.

   Parameters

   - **connection** (*kombu.Connection*) – Connection to the broker.

   - **handlers** (*Mapping[Callable]*) – Event handlers. This is a map of event type names and their handlers. The special handler "*" captures all events that don't have a handler.

app = None

capture(*limit=None*, *timeout=None*, *wakeup=True*)
   Open up a consumer capturing events.

   This has to run in the main process, and it will never stop unless `EventDispatcher.should_stop` is set to True, or forced via `KeyboardInterrupt` or `SystemExit`.

connection

event_from_message(*body*, *localize=True*, *now=<built-in function time>*, *tzfields=operator.itemgetter('utcoffset', 'timestamp')*, *adjust_timestamp=<function adjust_timestamp>*, *CLIENT_CLOCK_SKEW=-1*)

get_consumers(*Consumer*, *channel*)

itercapture(*limit=None*, *timeout=None*, *wakeup=True*)

on_consume_ready(*connection*, *channel*, *consumers*, *wakeup=True*, ***kwargs*)

---

process(*type*, *event*)
> Process event by dispatching to configured handler.

wakeup_workers(*channel=None*)

celery.events.**get_exchange**(*conn*, *name='celeryev'*)
> Get exchange used for sending events.

> **Parameters**
>> • **conn** (`kombu.Connection`) – Connection used for sending/receiving events.
>>
>> • **name** (`str`) – Name of the exchange. Default is `celeryev`.

> **Note:** The event type changes if Redis is used as the transport (from topic -> fanout).

celery.events.**group_from**(*type*)
> Get the group part of an event type name.

Example

```
>>> group_from('task-sent')
'task'
```

```
>>> group_from('custom-my-event')
'custom'
```

3.11.39 `celery.events.receiver`

Event receiver implementation.

class celery.events.receiver.**EventReceiver**(*channel*, *handlers=None*, *routing_key='#'*, *node_id=None*, *app=None*, *queue_prefix=None*, *accept=None*, *queue_ttl=None*, *queue_expires=None*)
> Capture events.

> **Parameters**
>> • **connection** (`kombu.Connection`) – Connection to the broker.
>>
>> • **handlers** (`Mapping[Callable]`) – Event handlers. This is a map of event type names and their handlers. The special handler "*" captures all events that don't have a handler.

app = None

capture(*limit=None*, *timeout=None*, *wakeup=True*)
> Open up a consumer capturing events.

> This has to run in the main process, and it will never stop unless `EventDispatcher.should_stop` is set to True, or forced via `KeyboardInterrupt` or `SystemExit`.

connection

event_from_message(*body*, *localize=True*, *now=<built-in function time>*, *tz-fields=operator.itemgetter('utcoffset', 'timestamp')*, *adjust_timestamp=<function adjust_timestamp>*, *CLIENT_CLOCK_SKEW=-1*)

get_consumers(*Consumer*, *channel*)

itercapture(*limit=None*, *timeout=None*, *wakeup=True*)

on_consume_ready(*connection*, *channel*, *consumers*, *wakeup=True*, ***kwargs*)

process(*type*, *event*)
  Process event by dispatching to configured handler.

wakeup_workers(*channel=None*)

3.11.40 `celery.events.state`

Event dispatcher sends events.

class `celery.events.dispatcher.`**EventDispatcher**(*connection=None*, *hostname=None*, *enabled=True*, *channel=None*, *buffer_while_offline=True*, *app=None*, *serializer=None*, *groups=None*, *delivery_mode=1*, *buffer_group=None*, *buffer_limit=24*, *on_send_buffered=None*)

  Dispatches event messages.

  Parameters

  - **connection** (*kombu.Connection*) – Connection to the broker.

  - **hostname** (*str*) – Hostname to identify ourselves as, by default uses the hostname returned by `anon_nodename()`.

  - **groups** (*Sequence[str]*) – List of groups to send events for. *send()* will ignore send requests to groups not in this list. If this is `None`, all events will be sent. Example groups include `"task"` and `"worker"`.

  - **enabled** (*bool*) – Set to `False` to not actually publish any events, making *send()* a no-op.

  - **channel** (*kombu.Channel*) – Can be used instead of *connection* to specify an exact channel to use when sending events.

  - **buffer_while_offline** (*bool*) – If enabled events will be buffered while the connection is down. *flush()* must be called as soon as the connection is re-established.

Note: You need to *close()* this after use.

DISABLED_TRANSPORTS = {'sql'}

app = None

close()
  Close the event dispatcher.

disable()

enable()

extend_buffer(*other*)
    Copy the outbound buffer of another instance.

flush(*errors=True*, *groups=True*)
    Flush the outbound buffer.

on_disabled = None

on_enabled = None

publish(*type*, *fields*, *producer*, *blind=False*, *Event=<function Event>*, ***kwargs*)
    Publish event using custom `Producer`.

        Parameters

- **type** (`str`) – Event type name, with group separated by dash (-). fields: Dictionary of event fields, must be json serializable.

- **producer** (`kombu.Producer`) – Producer instance to use: only the `publish` method will be called.

- **retry** (`bool`) – Retry in the event of connection failure.

- **retry_policy** (`Mapping`) – Map of custom retry policy options. See `ensure()`.

- **blind** (`bool`) – Don't set logical clock value (also don't forward the internal logical clock).

- **Event** (`Callable`) – Event type used to create event. Defaults to `Event()`.

- **utcoffset** (`Callable`) – Function returning the current utc offset in hours.

publisher

send(*type*, *blind=False*, *utcoffset=<function utcoffset>*, *retry=False*, *retry_policy=None*, *Event=<function Event>*, ***fields*)
    Send event.

        Parameters

- **type** (`str`) – Event type name, with group separated by dash (-).

- **retry** (`bool`) – Retry in the event of connection failure.

- **retry_policy** (`Mapping`) – Map of custom retry policy options. See `ensure()`.

- **blind** (`bool`) – Don't set logical clock value (also don't forward the internal logical clock).

- **Event** (`Callable`) – Event type used to create event, defaults to `Event()`.

- **utcoffset** (`Callable`) – unction returning the current utc offset in hours.

- ****fields** (`Any`) – Event fields – must be json serializable.

3.11.41 `celery.events.event`

Creating events, and event exchange definition.

`celery.events.event.`**Event**(*type*, *_fields=None*, *__dict__=<class 'dict'>*, *__now__=<built-in function time>*, ***fields*)
    Create an event.

Notes

An event is simply a dictionary: the only required field is `type`. A `timestamp` field will be set to the current time if not provided.

celery.events.event.**event_exchange = <unbound Exchange celeryev(topic)>**
Exchange used to send events on. Note: Use *get_exchange()* instead, as the type of exchange will vary depending on the broker connection.

celery.events.event.**get_exchange**(*conn*, *name='celeryev'*)
Get exchange used for sending events.

> **Parameters**
> - **conn** (*kombu.Connection*) – Connection used for sending/receiving events.
> - **name** (*str*) – Name of the exchange. Default is `celeryev`.

Note: The event type changes if Redis is used as the transport (from topic -> fanout).

---

celery.events.event.**group_from**(*type*)
Get the group part of an event type name.

Example

```
>>> group_from('task-sent')
'task'
```

```
>>> group_from('custom-my-event')
'custom'
```

3.11.42 `celery.events.state`

In-memory representation of cluster state.

This module implements a data-structure used to keep track of the state of a cluster of workers and the tasks it is working on (by consuming events).

For every event consumed the state is updated, so the state represents the state of the cluster at the time of the last event.

Snapshots (*celery.events.snapshot*) can be used to take "pictures" of this state at regular intervals to for example, store that in a database.

class celery.events.state.**Worker**(*hostname=None*, *pid=None*, *freq=60*, *heartbeats=None*, *clock=0*, *active=None*, *processed=None*, *loadavg=None*, *sw_ident=None*, *sw_ver=None*, *sw_sys=None*)
Worker State.

active

alive

clock

event

expire_window = 200

---

freq

heartbeat_expires

heartbeat_max = 4

heartbeats

hostname

id

loadavg

pid

processed

status_string

sw_ident

sw_sys

sw_ver

update (*f*, ***kw*)

class `celery.events.state.`**Task** (*uuid=None*, *cluster_state=None*, *children=None*, ***kwargs*)
  Task State.

  args = None

  as_dict ()

  client = None

  clock = 0

  eta = None

  event (*type_*, *timestamp=None*, *local_received=None*, *fields=None*, *precedence=<function precedence>*, *items=<function items>*, *setattr=<built-in function setattr>*, *task_event_to_state=<built-in method get of dict object>*, *RETRY='RETRY'*)

  exception = None

  exchange = None

  expires = None

  failed = None

  id

  info (*fields=None*, *extra=None*)
    Information about this task suitable for on-screen display.

  kwargs = None

  merge_rules = {'RECEIVED': ('name', 'args', 'kwargs', 'parent_id', 'root_id', 'retries
    How to merge out of order events. Disorder is detected by logical ordering (e.g., *task-received* must've happened before a *task-failed* event).

    A merge rule consists of a state and a list of fields to keep from that state. (RECEIVED, ('name', 'args')), means the name and args fields are always taken from the RECEIVED state, and any values for these fields received before or after is simply ignored.

  name = None

origin

parent

parent_id = None

ready

received = None

rejected = None

result = None

retried = None

retries = None

revoked = None

root

root_id = None

routing_key = None

runtime = None

sent = None

started = None

state = 'PENDING'

succeeded = None

timestamp = None

traceback = None

worker = None

class celery.events.state.**State**(*callback=None,   workers=None,   tasks=None,*
taskheap=None,   max_workers_in_memory=5000,
max_tasks_in_memory=10000,   on_node_join=None,
on_node_leave=None,   tasks_by_type=None,
tasks_by_worker=None)

Records clusters state.

class Task(*uuid=None, cluster_state=None, children=None, **kwargs*)

Task State.

args = None

as_dict()

client = None

clock = 0

eta = None

event(*type_, timestamp=None, local_received=None, fields=None, precedence=<function*
precedence>, items=<function items>, setattr=<built-in function setattr>,
task_event_to_state=<built-in method get of dict object>, RETRY='RETRY')

exception = None

exchange = None

expires = None

failed = None

id

info (*fields=None*, *extra=None*)
Information about this task suitable for on-screen display.

kwargs = None

merge_rules = {'RECEIVED': ('name', 'args', 'kwargs', 'parent_id', 'root_id', 'ret

name = None

origin

parent

parent_id = None

ready

received = None

rejected = None

result = None

retried = None

retries = None

revoked = None

root

root_id = None

routing_key = None

runtime = None

sent = None

started = None

state = 'PENDING'

succeeded = None

timestamp = None

traceback = None

worker = None

class Worker (*hostname=None*, *pid=None*, *freq=60*, *heartbeats=None*, *clock=0*, *active=None*, *processed=None*, *loadavg=None*, *sw_ident=None*, *sw_ver=None*, *sw_sys=None*)
Worker State.

active

alive

clock

event

expire_window = 200

> > **freq**
>
> > **heartbeat_expires**
>
> > **heartbeat_max = 4**
>
> > **heartbeats**
>
> > **hostname**
>
> > **id**
>
> > **loadavg**
>
> > **pid**
>
> > **processed**
>
> > **status_string**
>
> > **sw_ident**
>
> > **sw_sys**
>
> > **sw_ver**
>
> > **update**(*f*, ***kw*)

alive_workers()
> Return a list of (seemingly) alive workers.

clear(*ready=True*)

clear_tasks(*ready=True*)

event(*event*)

event_count = 0

freeze_while(*fun*, **args*, ***kwargs*)

get_or_create_task(*uuid*)
> Get or create task by uuid.

get_or_create_worker(*hostname*, ***kwargs*)
> Get or create worker by hostname.

> > **Returns** of (`worker, was_created`) pairs.

> > **Return type** Tuple

heap_multiplier = 4

itertasks(*limit=None*)

rebuild_taskheap(*timetuple=<class 'kombu.clocks.timetuple'>*)

task_count = 0

task_event(*type_*, *fields*)
> Deprecated, use *event()*.

task_types()
> Return a list of all seen task types.

tasks_by_time(*limit=None*, *reverse=True*)
> Generator yielding tasks ordered by time.

> > **Yields** Tuples of (`uuid, Task`).

tasks_by_timestamp(*limit=None*, *reverse=True*)
  Generator yielding tasks ordered by time.

  > **Yields** Tuples of (uuid, Task).

worker_event(*type_*, *fields*)
  Deprecated, use *event()*.

celery.events.state.**heartbeat_expires**(*timestamp*, *freq=60*, *expire_window=200*, *Decimal=<class 'decimal.Decimal'>*, *float=<class 'float'>*, *isinstance=<built-in function isinstance>*)
  Return time when heartbeat expires.

3.11.43 `celery.beat`

The periodic task scheduler.

exception celery.beat.**SchedulingError**
  An error occurred while scheduling a task.

class celery.beat.**ScheduleEntry**(*name=None*, *task=None*, *last_run_at=None*, *total_run_count=None*, *schedule=None*, *args=()*, *kwargs=None*, *options=None*, *relative=False*, *app=None*)
  An entry in the scheduler.

  > **Parameters**
  >
  > - **name** (*str*) – see *name*.
  > - **schedule** (*schedule*) – see *schedule*.
  > - **args** (*Tuple*) – see *args*.
  > - **kwargs** (*Dict*) – see *kwargs*.
  > - **options** (*Dict*) – see *options*.
  > - **last_run_at** (*datetime*) – see *last_run_at*.
  > - **total_run_count** (*int*) – see *total_run_count*.
  > - **relative** (*bool*) – Is the time relative to when the server starts?

args = None
  Positional arguments to apply.

default_now()

editable_fields_equal(*other*)

is_due()
  See is_due().

kwargs = None
  Keyword arguments to apply.

last_run_at = None
  The time and date of when this task was last scheduled.

name = None
  The task name

next(*last_run_at=None*)
  Return new instance, with date and count fields updated.

options = None
> Task execution options.

schedule = None
> The schedule (*schedule*)

total_run_count = 0
> Total number of times this task has been scheduled.

update(*other*)
> Update values from another entry.
>
> **Will only update "editable" fields:** `task`, `schedule`, `args`, `kwargs`, `options`.

class `celery.beat.`**Scheduler**(*app*,   *schedule=None*,   *max_interval=None*,   *Producer=None*,
> *lazy=False*, *sync_every_tasks=None*, ***kwargs*)
>
> Scheduler for periodic tasks.
>
> The **celery beat** program may instantiate this class multiple times for introspection purposes, but then with the `lazy` argument set. It's important for subclasses to be idempotent when this argument is set.
>
> > **Parameters**
> >
> > - **schedule** (*schedule*) – see *schedule*.
> > - **max_interval** (*int*) – see *max_interval*.
> > - **lazy** (*bool*) – Don't set up the schedule.
>
> **Entry**
> > alias of *ScheduleEntry*
>
> **add**(***kwargs*)
>
> **adjust**(*n*, *drift=-0.01*)
>
> **apply_async**(*entry*, *producer=None*, *advance=True*, ***kwargs*)
>
> **apply_entry**(*entry*, *producer=None*)
>
> **close**()
>
> **connection**
>
> **get_schedule**()
>
> **info**
>
> **install_default_entries**(*data*)
>
> **is_due**(*entry*)
>
> **logger = <Logger celery.beat (WARNING)>**
>
> **max_interval = 300**
> > Maximum time to sleep between re-checking the schedule.
>
> **merge_inplace**(*b*)
>
> **populate_heap**(*event_t=<class 'celery.beat.event_t'>*, *heapify=<built-in function heapify>*)
> > Populate the heap with the data contained in the schedule.
>
> **producer**
>
> **reserve**(*entry*)
>
> **schedule**
> > The schedule dict/shelve.

schedules_equal(*old_schedules*, *new_schedules*)

send_task(**args*, ***kwargs*)

set_schedule(*schedule*)

setup_schedule()

should_sync()

sync()

sync_every = 180
    How often to sync the schedule (3 minutes by default)

sync_every_tasks = None
    How many tasks can be called before a sync is forced.

tick(*event_t=<class 'celery.beat.event_t'>*, *min=<built-in function min>*, *heappop=<built-in function heappop>*, *heappush=<built-in function heappush>*)
    Run a tick - one iteration of the scheduler.

    Executes one due task per call.

        Returns preferred delay in seconds for next call.

        Return type float

update_from_dict(*dict_*)

class celery.beat.**PersistentScheduler**(**args*, ***kwargs*)
    Scheduler backed by shelve database.

    close()

    get_schedule()

    info

    known_suffixes = ('', '.db', '.dat', '.bak', '.dir')

    persistence = <module 'shelve' from '/home/docs/.pyenv/versions/3.7.3/lib/python3.7/sh

    schedule

    set_schedule(*schedule*)

    setup_schedule()

    sync()

class celery.beat.**Service**(*app*, *max_interval=None*, *schedule_filename=None*, *scheduler_cls=None*)
    Celery periodic task service.

    get_scheduler(*lazy=False*, *extension_namespace='celery.beat_schedulers'*)

    scheduler

    scheduler_cls
        alias of *PersistentScheduler*

    start(*embedded_process=False*)

    stop(*wait=False*)

    sync()

`celery.beat.`**`EmbeddedService`**(*app*, *max_interval=None*, ***kwargs*)
    Return embedded clock service.

> **Parameters thread** ([*bool*](#)) – Run threaded instead of as a separate process. Uses [`multiprocessing`](#) by default, if available.

3.11.44 `celery.apps.worker`

Worker command-line program.

This module is the 'program-version' of [`celery.worker`](#).

It does everything necessary to run that module as an actual application, like installing signal handlers, platform tweaks, and so on.

class `celery.apps.worker.`**`Worker`**(*app=None*, *hostname=None*, ***kwargs*)
    Worker as a program.

`emit_banner`()

`extra_info`()

`install_platform_tweaks`(*worker*)
    Install platform specific tweaks and workarounds.

`macOS_proxy_detection_workaround`()
    See https://github.com/celery/celery/issues#issue/161.

`on_after_init`(*purge=False*, *no_color=None*, *redirect_stdouts=None*, *redirect_stdouts_level=None*, ***kwargs*)

`on_before_init`(*quiet=False*, ***kwargs*)

`on_consumer_ready`(*consumer*)

`on_init_blueprint`()

`on_start`()

`purge_messages`()

`set_process_status`(*info*)

`setup_logging`(*colorize=None*)

`startup_info`(*artlines=True*)

`tasklist`(*include_builtins=True*, *sep='\n'*, *int_='celery.'*)

3.11.45 `celery.apps.beat`

Beat command-line program.

This module is the 'program-version' of [`celery.beat`](#).

It does everything necessary to run that module as an actual application, like installing signal handlers and so on.

class `celery.apps.beat.`**`Beat`**(*max_interval=None*, *app=None*, *socket_timeout=30*, *pidfile=None*, *no_color=None*, *loglevel='WARN'*, *logfile=None*, *schedule=None*, *scheduler=None*, *scheduler_cls=None*, *redirect_stdouts=None*, *redirect_stdouts_level=None*, ***kwargs*)
    Beat as a service.

class Service (*app*, *max_interval=None*, *schedule_filename=None*, *scheduler_cls=None*)
Celery periodic task service.

> **get_scheduler** (*lazy=False*, *extension_namespace='celery.beat_schedulers'*)
>
> **scheduler**
>
> **scheduler_cls**
> alias of `PersistentScheduler`
>
> **start** (*embedded_process=False*)
>
> **stop** (*wait=False*)
>
> **sync** ()

app = None

banner (*service*)

init_loader ()

install_sync_handler (*service*)
Install a *SIGTERM + SIGINT* handler saving the schedule.

run ()

set_process_title ()

setup_logging (*colorize=None*)

start_scheduler ()

startup_info (*service*)

3.11.46 `celery.apps.multi`

Start/stop/manage workers.

class `celery.apps.multi.`**Cluster** (*nodes*, *cmd=None*, *env=None*, *on_stopping_preamble=None*, *on_send_signal=None*, *on_still_waiting_for=None*, *on_still_waiting_progress=None*, *on_still_waiting_end=None*, *on_node_start=None*, *on_node_restart=None*, *on_node_shutdown_ok=None*, *on_node_status=None*, *on_node_signal=None*, *on_node_signal_dead=None*, *on_node_down=None*, *on_child_spawn=None*, *on_child_signalled=None*, *on_child_failure=None*)
Represent a cluster of workers.

data

find (*name*)

getpids (*on_down=None*)

kill ()

restart (*sig=<Signals.SIGTERM: 15>*)

send_all (*sig*)

shutdown_nodes (*nodes*, *sig=<Signals.SIGTERM: 15>*, *retry=None*)

start ()

start_node (*node*)

stop (*retry=None*, *callback=None*, *sig=<Signals.SIGTERM: 15>*)

stopwait (*retry=2*, *callback=None*, *sig=<Signals.SIGTERM: 15>*)

class celery.apps.multi.**Node** (*name*, *cmd=None*, *append=None*, *options=None*, *extra_args=None*)

Represents a node in a cluster.

alive ()

argv_with_executable

executable

classmethod from_kwargs (*name*, ***kwargs*)

getopt (**alt*)

handle_process_exit (*retcode*, *on_signalled=None*, *on_failure=None*)

logfile

pid

pidfile

prepare_argv (*argv*, *path*)

send (*sig*, *on_error=None*)

start (*env=None*, ***kwargs*)

3.11.47 `celery.worker`

Worker implementation.

class celery.worker.**WorkController** (*app=None*, *hostname=None*, ***kwargs*)

Unmanaged worker instance.

class Blueprint (*steps=None*, *name=None*, *on_start=None*, *on_close=None*, *on_stopped=None*)

Worker bootstep blueprint.

default_steps = {'celery.worker.autoscale:WorkerComponent', 'celery.worker.compone

name = 'Worker'

app = None

blueprint = None

exitcode = None

contains the exit code if a SystemExit event is handled.

info ()

on_after_init (***kwargs*)

on_before_init (***kwargs*)

on_close ()

on_consumer_ready (*consumer*)

on_init_blueprint ()

on_start ()

on_stopped ()

pidlock = None

pool = None

prepare_args(***kwargs*)

register_with_event_loop(*hub*)

reload(*modules=None*, *reload=False*, *reloader=None*)

rusage()

semaphore = None

setup_defaults(*concurrency=None*, *loglevel='WARN'*, *logfile=None*, *task_events=None*, *pool=None*, *consumer_cls=None*, *timer_cls=None*, *timer_precision=None*, *autoscaler_cls=None*, *pool_putlocks=None*, *pool_restarts=None*, *optimization=None*, *O=None*, *statedb=None*, *time_limit=None*, *soft_time_limit=None*, *scheduler=None*, *pool_cls=None*, *state_db=None*, *task_time_limit=None*, *task_soft_time_limit=None*, *scheduler_cls=None*, *schedule_filename=None*, *max_tasks_per_child=None*, *prefetch_multiplier=None*, *disable_rate_limits=None*, *worker_lost_wait=None*, *max_memory_per_child=None*, ***_kw*)

setup_includes(*includes*)

setup_instance(*queues=None*, *ready_callback=None*, *pidfile=None*, *include=None*, *use_eventloop=None*, *exclude_queues=None*, ***kwargs*)

setup_queues(*include*, *exclude=None*)

should_use_eventloop()

signal_consumer_close()

start()

state

stats()

stop(*in_sighandler=False*, *exitcode=None*)
> Graceful shutdown of the worker server.

terminate(*in_sighandler=False*)
> Not so graceful shutdown of the worker server.

3.11.48 `celery.worker.request`

Task request.

This module defines the [`Request`](#) class, that specifies how tasks are executed.

class celery.worker.request.**Request**(*message*, *on_ack=<function noop>*, *hostname=None*, *eventer=None*, *app=None*, *connection_errors=None*, *request_dict=None*, *task=None*, *on_reject=<function noop>*, *body=None*, *headers=None*, *decoded=False*, *utc=True*, *maybe_make_aware=<function maybe_make_aware>*, *maybe_iso8601=<function maybe_iso8601>*, ***opts*)
> A request for task execution.

> **acknowledge**()
> > Acknowledge task.

acknowledged = False

app

args

argsrepr

body

chord

connection_errors

content_encoding

content_type

correlation_id

delivery_info

errbacks

eta

eventer

execute(*loglevel=None*, *logfile=None*)
> Execute the task in a `trace_task()`.

>> **Parameters**

>>> • **loglevel** (`int`) – The loglevel used by the task.

>>> • **logfile** (`str`) – The logfile used by the task.

execute_using_pool(*pool*, ***kwargs*)
> Used by the worker to send this task to the pool.

>> **Parameters** **pool** (`TaskPool`) – The execution pool used to execute this request.

>> **Raises** `celery.exceptions.TaskRevokedError` – if the task was revoked.

expires

group

hostname

humaninfo()

id

info(*safe=False*)

kwargs

kwargsrepr

maybe_expire()
> If expired, mark the task as revoked.

message

name

on_accepted(*pid*, *time_accepted*)
> Handler called when task is accepted by worker pool.

on_ack

on_failure(*exc_info*, *send_failed_event=True*, *return_ok=False*)
    Handler called if the task raised an exception.

on_reject

on_retry(*exc_info*)
    Handler called if the task should be retried.

on_success(*failed__retval__runtime*, ***kwargs*)
    Handler called if the task was successfully processed.

on_timeout(*soft*, *timeout*)
    Handler called if the task times out.

parent_id

reject(*requeue=False*)

reply_to

request_dict

revoked()
    If revoked, skip task and mark state.

root_id

send_event(*type*, ***fields*)

store_errors

task

task_id

task_name

terminate(*pool*, *signal=None*)

time_limits = (None, None)

time_start = None

type

tzlocal

utc

worker_pid = None

3.11.49 `celery.worker.state`

Internal worker state (global).

This includes the currently active and reserved tasks, statistics, and revoked tasks.

celery.worker.state.**SOFTWARE_INFO = {'sw_ident': 'py-celery', 'sw_sys': 'Linux', 'sw_ver**
    Worker software/platform information.

celery.worker.state.**reserved_requests = <_weakrefset.WeakSet object>**
    set of all reserved *Request*'s.

celery.worker.state.**active_requests = <_weakrefset.WeakSet object>**
    set of currently active *Request*'s.

`celery.worker.state.`**`total_count = {}`**
    count of tasks accepted by the worker, sorted by type.

`celery.worker.state.`**`revoked = <LimitedSet(0):  maxlen=50000, expires=10800, minlen=0>`**
    the list of currently revoked tasks. Persistent if `statedb` set.

`celery.worker.state.`**`task_reserved`**(*request*, *add_request=<method-wrapper '__setitem__' of dict object>*, *add_reserved_request=<bound method Weak-Set.add of <_weakrefset.WeakSet object>>*)
    Update global state when a task has been reserved.

`celery.worker.state.`**`maybe_shutdown`**()
    Shutdown if flags have been set.

`celery.worker.state.`**`task_accepted`**(*request*, *_all_total_count=None*, *add_active_request=<bound method Weak-Set.add of <_weakrefset.WeakSet object>>*, *add_to_total_count=<bound method Counter.update of Counter()>*)
    Update global state when a task has been accepted.

`celery.worker.state.`**`task_ready`**(*request*, *remove_request=<built-in method pop of dict object>*, *discard_active_request=<bound method Weak-Set.discard of <_weakrefset.WeakSet object>>*, *discard_reserved_request=<bound method WeakSet.discard of <_weakrefset.WeakSet object>>*)
    Update global state when a task is ready.

class `celery.worker.state.`**`Persistent`**(*state*, *filename*, *clock=None*)
    Stores worker state between restarts.

    This is the persistent data stored by the worker when *celery worker --statedb* is enabled.

    Currently only stores revoked task id's.

    `close`()

    `compress`()
        Returns a bytes object containing compressed data.

        data Binary data to be compressed.

        level Compression level, in 0-9 or -1.

    `db`

    `decompress`()
        Returns a bytes object containing the uncompressed data.

        data Compressed data.

        wbits The window buffer size and container format.

        bufsize The initial output buffer size.

    `merge`()

    `open`()

    `protocol = 2`

    `save`()

    `storage = <module 'shelve' from '/home/docs/.pyenv/versions/3.7.3/lib/python3.7/shelve`

    `sync`()

3.11.50 `celery.worker.strategy`

Task execution strategy (optimization).

`celery.worker.strategy.`**`default`**(*task*, *app*, *consumer*, *info=<bound method Logger.info of <Logger celery.worker.strategy (WARNING)>>*, *error=<bound method Logger.error of <Logger celery.worker.strategy (WARNING)>>*, *task_reserved=<function task_reserved>*, *to_system_tz=<bound method _Zone.to_system of <celery.utils.time._Zone object>>*, *bytes=<class 'bytes'>*, *buffer_t=<class 'vine.five.buffer_t'>*, *proto1_to_proto2=<function proto1_to_proto2>*)

Default task execution strategy.

Note: Strategies are here as an optimization, so sadly it's not very easy to override.

3.11.51 `celery.worker.consumer`

Worker consumer.

`class` `celery.worker.consumer.`**`Consumer`**(*on_task_request*, *init_callback=<function noop>*, *hostname=None*, *pool=None*, *app=None*, *timer=None*, *controller=None*, *hub=None*, *amqheartbeat=None*, *worker_options=None*, *disable_rate_limits=False*, *initial_prefetch_count=2*, *prefetch_multiplier=1*, ***kwargs*)

Consumer blueprint.

    `class Blueprint`(*steps=None*, *name=None*, *on_start=None*, *on_close=None*, *on_stopped=None*)
        Consumer blueprint.

        `default_steps = ['celery.worker.consumer.connection:Connection', 'celery.worker.co`

        `name = 'Consumer'`

        `shutdown`(*parent*)

    `Strategies`
        alias of `builtins.dict`

    `add_task_queue`(*queue*, *exchange=None*, *exchange_type=None*, *routing_key=None*, ***options*)

    `apply_eta_task`(*task*)
        Method called by the timer to apply a task with an ETA/countdown.

    `bucket_for_task`(*type*)

    `call_soon`(*p*, **args*, ***kwargs*)

    `cancel_task_queue`(*queue*)

    `connect`()
        Establish the broker connection used for consuming tasks.

        Retries establishing the connection if the *broker_connection_retry* setting is enabled

    `connection_for_read`(*heartbeat=None*)

    `connection_for_write`(*heartbeat=None*)

create_task_handler(*promise=<class 'vine.promises.promise'>*)

ensure_connected(*conn*)

init_callback = None
> Optional callback called the first time the worker is ready to receive tasks.

loop_args()

on_close()

on_connection_error_after_connected(*exc*)

on_connection_error_before_connected(*exc*)

on_decode_error(*message*, *exc*)
> Callback called if an error occurs while decoding a message.
>
> Simply logs the error and acknowledges the message so it doesn't enter a loop.
>
> > **Parameters**
> >
> > - **message** (*kombu.Message*) – The message received.
> >
> > - **exc** ([*Exception*](#)) – The exception being handled.

on_invalid_task(*body*, *message*, *exc*)

on_ready()

on_send_event_buffered()

on_unknown_message(*body*, *message*)

on_unknown_task(*body*, *message*, *exc*)

perform_pending_operations()

pool = None
> The current worker pool instance.

register_with_event_loop(*hub*)

reset_rate_limits()

restart_count = -1

shutdown()

start()

stop()

timer = None
> A timer used for high-priority internal tasks, such as sending heartbeats.

update_strategies()

class celery.worker.consumer.**Agent**(*c*, ***kwargs*)
> Agent starts [cell](#) actors.

conditional = True

create(*c*)
> Create the step.

name = 'celery.worker.consumer.agent.Agent'

requires = (step:celery.worker.consumer.connection.Connection{()},)

class celery.worker.consumer.**Connection**(*c*, ***kwargs*)
  Service managing the consumer broker connection.

  info(*c*)

  name = 'celery.worker.consumer.connection.Connection'

  shutdown(*c*)

  start(*c*)

class celery.worker.consumer.**Control**(*c*, ***kwargs*)
  Remote control command service.

  include_if(*c*)
    Return true if bootstep should be included.

    You can define this as an optional predicate that decides whether this step should be created.

  name = 'celery.worker.consumer.control.Control'

  requires = (step:celery.worker.consumer.tasks.Tasks{(step:celery.worker.consumer.mingl

class celery.worker.consumer.**Events**(*c*, *task_events=True*, *without_heartbeat=False*, *without_gossip=False*, ***kwargs*)
  Service used for sending monitoring events.

  name = 'celery.worker.consumer.events.Events'

  requires = (step:celery.worker.consumer.connection.Connection{()},)

  shutdown(*c*)

  start(*c*)

  stop(*c*)

class celery.worker.consumer.**Gossip**(*c*, *without_gossip=False*, *interval=5.0*, *heartbeat_interval=2.0*, ***kwargs*)
  Bootstep consuming events from other workers.

  This keeps the logical clock value up to date.

  call_task(*task*)

  compatible_transport(*app*)

  compatible_transports = {'amqp', 'redis'}

  election(*id*, *topic*, *action=None*)

  get_consumers(*channel*)

  label = 'Gossip'

  name = 'celery.worker.consumer.gossip.Gossip'

  on_elect(*event*)

  on_elect_ack(*event*)

  on_message(*prepare*, *message*)

  on_node_join(*worker*)

  on_node_leave(*worker*)

  on_node_lost(*worker*)

  periodic()

register_timer()

requires = (step:celery.worker.consumer.mingle.Mingle{(step:celery.worker.consumer.even

start(*c*)

class celery.worker.consumer.**Heart**(*c*, *without_heartbeat=False*, *heartbeat_interval=None*,
**kwargs)

Bootstep sending event heartbeats.

This service sends a `worker-heartbeat` message every n seconds.

Note: Not to be confused with AMQP protocol level heartbeats.

---

name = 'celery.worker.consumer.heart.Heart'

requires = (step:celery.worker.consumer.events.Events{(step:celery.worker.consumer.con

shutdown(*c*)

start(*c*)

stop(*c*)

class celery.worker.consumer.**Mingle**(*c*, *without_mingle=False*, ***kwargs*)

Bootstep syncing state with neighbor workers.

At startup, or upon consumer restart, this will:

> • Sync logical clocks.
>
> • Sync revoked tasks.

compatible_transport(*app*)

compatible_transports = {'amqp', 'redis'}

label = 'Mingle'

name = 'celery.worker.consumer.mingle.Mingle'

on_clock_event(*c*, *clock*)

on_node_reply(*c*, *nodename*, *reply*)

on_revoked_received(*c*, *revoked*)

requires = (step:celery.worker.consumer.events.Events{(step:celery.worker.consumer.con

send_hello(*c*)

start(*c*)

sync(*c*)

sync_with_node(*c*, *clock=None*, *revoked=None*, ***kwargs*)

class celery.worker.consumer.**Tasks**(*c*, ***kwargs*)

Bootstep starting the task message consumer.

info(*c*)
Return task consumer info.

name = 'celery.worker.consumer.tasks.Tasks'

requires = (step:celery.worker.consumer.mingle.Mingle{(step:celery.worker.consumer.even

---

shutdown(*c*)
> Shutdown task consumer.

start(*c*)
> Start task consumer.

stop(*c*)
> Stop task consumer.

3.11.52 `celery.worker.consumer.agent`

Celery + cell integration.

class `celery.worker.consumer.agent.`**Agent**(*c*, ***kwargs*)
> Agent starts cell actors.

> **conditional = True**

> **create**(*c*)
> > Create the step.

> **name = 'celery.worker.consumer.agent.Agent'**

> **requires = (step:celery.worker.consumer.connection.Connection{()},)**

3.11.53 `celery.worker.consumer.connection`

Consumer Broker Connection Bootstep.

class `celery.worker.consumer.connection.`**Connection**(*c*, ***kwargs*)
> Service managing the consumer broker connection.

> **info**(*c*)

> **name = 'celery.worker.consumer.connection.Connection'**

> **shutdown**(*c*)

> **start**(*c*)

3.11.54 `celery.worker.consumer.consumer`

Worker Consumer Blueprint.

This module contains the components responsible for consuming messages from the broker, processing the messages and keeping the broker connections up and running.

class `celery.worker.consumer.consumer.`**Consumer**(*on_task_request*,
init_callback=<function noop>,
hostname=None, *pool=None*,
app=None, *timer=None*, *controller=None*, *hub=None*, *amqheartbeat=None*, *worker_options=None*,
disable_rate_limits=False,
initial_prefetch_count=2,
prefetch_multiplier=1, ***kwargs*)
> Consumer blueprint.

class Blueprint (*steps=None*, *name=None*, *on_start=None*, *on_close=None*, *on_stopped=None*)
Consumer blueprint.

default_steps = ['celery.worker.consumer.connection:Connection', 'celery.worker.co

name = 'Consumer'

shutdown (*parent*)

Strategies
alias of `builtins.dict`

add_task_queue (*queue*, *exchange=None*, *exchange_type=None*, *routing_key=None*, ***options*)

apply_eta_task (*task*)
Method called by the timer to apply a task with an ETA/countdown.

bucket_for_task (*type*)

call_soon (*p*, **args*, ***kwargs*)

cancel_task_queue (*queue*)

connect ()
Establish the broker connection used for consuming tasks.

Retries establishing the connection if the `broker_connection_retry` setting is enabled

connection_for_read (*heartbeat=None*)

connection_for_write (*heartbeat=None*)

create_task_handler (*promise=<class 'vine.promises.promise'>*)

ensure_connected (*conn*)

init_callback = None
Optional callback called the first time the worker is ready to receive tasks.

loop_args ()

on_close ()

on_connection_error_after_connected (*exc*)

on_connection_error_before_connected (*exc*)

on_decode_error (*message*, *exc*)
Callback called if an error occurs while decoding a message.

Simply logs the error and acknowledges the message so it doesn't enter a loop.

> **Parameters**
>
> > - **message** (`kombu.Message`) – The message received.
> >
> > - **exc** ([`Exception`]) – The exception being handled.

on_invalid_task (*body*, *message*, *exc*)

on_ready ()

on_send_event_buffered ()

on_unknown_message (*body*, *message*)

on_unknown_task (*body*, *message*, *exc*)

perform_pending_operations ()

pool = None
>   The current worker pool instance.

register_with_event_loop(*hub*)

reset_rate_limits()

restart_count = -1

shutdown()

start()

stop()

timer = None
>   A timer used for high-priority internal tasks, such as sending heartbeats.

update_strategies()

class celery.worker.consumer.consumer.**Evloop**(*parent*, ***kwargs*)
>   Event loop service.

>   **Note:** This is always started last.

---

label = 'event loop'

last = True

name = 'celery.worker.consumer.consumer.Evloop'

patch_all(*c*)

start(*c*)

celery.worker.consumer.consumer.**dump_body**(*m*, *body*)
>   Format message body for debugging purposes.

3.11.55 `celery.worker.consumer.control`

Worker Remote Control Bootstep.

Control -> *celery.worker.pidbox* -> kombu.pidbox.

The actual commands are implemented in *celery.worker.control*.

class celery.worker.consumer.control.**Control**(*c*, ***kwargs*)
>   Remote control command service.

include_if(*c*)
>   Return true if bootstep should be included.

>   You can define this as an optional predicate that decides whether this step should be created.

name = 'celery.worker.consumer.control.Control'

requires = (step:celery.worker.consumer.tasks.Tasks{(step:celery.worker.consumer.mingle

3.11.56 `celery.worker.consumer.events`

Worker Event Dispatcher Bootstep.

Events -> `celery.events.EventDispatcher`.

class celery.worker.consumer.events.**Events**(*c*, *task_events=True*, *without_heartbeat=False*, *without_gossip=False*, ***kwargs*)

    Service used for sending monitoring events.

    name = 'celery.worker.consumer.events.Events'

    requires = (step:celery.worker.consumer.connection.Connection{()},)

    shutdown(*c*)

    start(*c*)

    stop(*c*)

3.11.57 `celery.worker.consumer.gossip`

Worker <-> Worker communication Bootstep.

class celery.worker.consumer.gossip.**Gossip**(*c*, *without_gossip=False*, *interval=5.0*, *heartbeat_interval=2.0*, ***kwargs*)

    Bootstep consuming events from other workers.

    This keeps the logical clock value up to date.

    call_task(*task*)

    compatible_transport(*app*)

    compatible_transports = {'amqp', 'redis'}

    election(*id*, *topic*, *action=None*)

    get_consumers(*channel*)

    label = 'Gossip'

    name = 'celery.worker.consumer.gossip.Gossip'

    on_elect(*event*)

    on_elect_ack(*event*)

    on_message(*prepare*, *message*)

    on_node_join(*worker*)

    on_node_leave(*worker*)

    on_node_lost(*worker*)

    periodic()

    register_timer()

    requires = (step:celery.worker.consumer.mingle.Mingle{(step:celery.worker.consumer.even

    start(*c*)

3.11.58 `celery.worker.consumer.heart`

Worker Event Heartbeat Bootstep.

class celery.worker.consumer.heart.**Heart**(*c*, *without_heartbeat=False*, *heartbeat_interval=None*, ***kwargs*)

> Bootstep sending event heartbeats.

> This service sends a `worker-heartbeat` message every n seconds.

> ---

> **Note:** Not to be confused with AMQP protocol level heartbeats.

> ---

> **name = 'celery.worker.consumer.heart.Heart'**

> **requires = (step:celery.worker.consumer.events.Events{(step:celery.worker.consumer.con**

> **shutdown**(*c*)

> **start**(*c*)

> **stop**(*c*)

3.11.59 `celery.worker.consumer.mingle`

Worker <-> Worker Sync at startup (Bootstep).

class celery.worker.consumer.mingle.**Mingle**(*c*, *without_mingle=False*, ***kwargs*)

> Bootstep syncing state with neighbor workers.

> At startup, or upon consumer restart, this will:

> > • Sync logical clocks.

> > • Sync revoked tasks.

> **compatible_transport**(*app*)

> **compatible_transports = {'amqp', 'redis'}**

> **label = 'Mingle'**

> **name = 'celery.worker.consumer.mingle.Mingle'**

> **on_clock_event**(*c*, *clock*)

> **on_node_reply**(*c*, *nodename*, *reply*)

> **on_revoked_received**(*c*, *revoked*)

> **requires = (step:celery.worker.consumer.events.Events{(step:celery.worker.consumer.con**

> **send_hello**(*c*)

> **start**(*c*)

> **sync**(*c*)

> **sync_with_node**(*c*, *clock=None*, *revoked=None*, ***kwargs*)

3.11.60 `celery.worker.consumer.tasks`

Worker Task Consumer Bootstep.

class `celery.worker.consumer.tasks.`**`Tasks`**(*c*, ***kwargs*)
> Bootstep starting the task message consumer.

> **`info`**(*c*)
>> Return task consumer info.

> **`name = 'celery.worker.consumer.tasks.Tasks'`**

> **`requires = (step:celery.worker.consumer.mingle.Mingle{(step:celery.worker.consumer.eve`**

> **`shutdown`**(*c*)
>> Shutdown task consumer.

> **`start`**(*c*)
>> Start task consumer.

> **`stop`**(*c*)
>> Stop task consumer.

3.11.61 `celery.worker.worker`

WorkController can be used to instantiate in-process workers.

The command-line interface for the worker is in `celery.bin.worker`, while the worker program is in `celery.apps.worker`.

The worker program is responsible for adding signal handlers, setting up logging, etc. This is a bare-bones worker without global side-effects (i.e., except for the global state stored in `celery.worker.state`).

The worker consists of several components, all managed by bootsteps (mod:*celery.bootsteps*).

class `celery.worker.worker.`**`WorkController`**(*app=None*, *hostname=None*, ***kwargs*)
> Unmanaged worker instance.

> **class Blueprint**(*steps=None*, *name=None*, *on_start=None*, *on_close=None*, *on_stopped=None*)
>> Worker bootstep blueprint.

>> **`default_steps = {'celery.worker.autoscale:WorkerComponent', 'celery.worker.compone`**

>> **`name = 'Worker'`**

> **`app = None`**

> **`blueprint = None`**

> **`exitcode = None`**
>> contains the exit code if a `SystemExit` event is handled.

> **`info`**()

> **`on_after_init`**(***kwargs*)

> **`on_before_init`**(***kwargs*)

> **`on_close`**()

> **`on_consumer_ready`**(*consumer*)

> **`on_init_blueprint`**()

> **`on_start`**()

on_stopped()

pidlock = None

pool = None

prepare_args(***kwargs*)

register_with_event_loop(*hub*)

reload(*modules=None*, *reload=False*, *reloader=None*)

rusage()

semaphore = None

setup_defaults(*concurrency=None*, *loglevel='WARN'*, *logfile=None*, *task_events=None*, *pool=None*, *consumer_cls=None*, *timer_cls=None*, *timer_precision=None*, *autoscaler_cls=None*, *pool_putlocks=None*, *pool_restarts=None*, *optimization=None*, *O=None*, *statedb=None*, *time_limit=None*, *soft_time_limit=None*, *scheduler=None*, *pool_cls=None*, *state_db=None*, *task_time_limit=None*, *task_soft_time_limit=None*, *scheduler_cls=None*, *schedule_filename=None*, *max_tasks_per_child=None*, *prefetch_multiplier=None*, *disable_rate_limits=None*, *worker_lost_wait=None*, *max_memory_per_child=None*, ***_kw*)

setup_includes(*includes*)

setup_instance(*queues=None*, *ready_callback=None*, *pidfile=None*, *include=None*, *use_eventloop=None*, *exclude_queues=None*, ***kwargs*)

setup_queues(*include*, *exclude=None*)

should_use_eventloop()

signal_consumer_close()

start()

state

stats()

stop(*in_sighandler=False*, *exitcode=None*)
    Graceful shutdown of the worker server.

terminate(*in_sighandler=False*)
    Not so graceful shutdown of the worker server.

3.11.62 `celery.bin.base`

Base command-line interface.

exception celery.bin.base.**Error**(*reason*, *status=None*)
    Exception raised by commands.

    status = 1

exception celery.bin.base.**UsageError**(*reason*, *status=None*)
    Exception raised for malformed arguments.

    status = 64

class celery.bin.base.**Extensions**(*namespace*, *register*)
    Loads extensions from setuptools entrypoints.

add(*cls*, *name*)

load()

class celery.bin.base.**Command**(*app=None*, *get_app=None*, *no_color=False*, *std-out=None*, *stderr=None*, *quiet=False*, *on_error=None*, *on_usage_error=None*)

Base class for command-line applications.

> **Parameters**
>
> - **app** (`Celery`) – The app to use.
>
> - **get_app** (`Callable`) – Fucntion returning the current app when no app provided.

exception Error(*reason*, *status=None*)

Exception raised by commands.

> **status = 1**

Parser

alias of `argparse.ArgumentParser`

exception UsageError(*reason*, *status=None*)

Exception raised for malformed arguments.

> **status = 64**

add_append_opt(*acc*, *opt*, *value*)

add_arguments(*parser*)

add_compat_options(*parser*, *options*)

add_preload_arguments(*parser*)

args = ''

Arg list used in help.

args_name = 'args'

Name of argparse option used for parsing positional args.

ask(*q*, *choices*, *default=None*)

Prompt user to choose from a tuple of string values.

If a default is not specified the question will be repeated until the user gives a valid choice.

Matching is case insensitive.

> **Parameters**
>
> - **q** (`str`) – the question to ask (don't include questionark)
>
> - **choice** (`Tuple[str]`) – tuple of possible choices, must be lowercase.
>
> - **default** (`Any`) – Default value if any.

check_args(*args*)

colored

create_parser(*prog_name*, *command=None*)

description = ''

Text to print in –help before option list.

die(*msg*, *status=1*)

doc = None

early_version (*argv*)

enable_config_from_cmdline = False
> Enable if the application should support config from the cmdline.

epilog = None
> Text to print at end of –help

error (*s*)

execute_from_commandline (*argv=None*)
> Execute application from command-line.

> > **Parameters argv** (`List[str]`) – The list of command-line arguments. Defaults to `sys.argv`.

expanduser (*value*)

find_app (*app*)

get_cls_by_name (*name*, *imp=<function import_from_cwd>*)

get_options ()

handle_argv (*prog_name*, *argv*, *command=None*)
> Parse arguments from argv and dispatch to `run()`.

> > **Warning:** Exits with an error message if `supports_args` is disabled and `argv` contains positional arguments.

> > **Parameters**
> > - **prog_name** (`str`) – The program name (`argv[0]`).
> > - **argv** (`List[str]`) – Rest of command-line arguments.

host_format (*s*, ***extra*)

leaf = True
> Set to true if this command doesn't have sub-commands

maybe_patch_concurrency (*argv=None*)

namespace = None
> Default configuration name-space.

no_color

node_format (*s*, *nodename*, ***extra*)

on_concurrency_setup ()

on_error (*exc*)

on_usage_error (*exc*)

option_list = None
> List of options (without preload options).

out (*s*, *fh=None*)

parse_doc (*doc*)

parse_options(*prog_name*, *arguments*, *command=None*)
   Parse the available options.

parse_preload_options(*args*)

prepare_args(*options*, *args*)

prepare_arguments(*parser*)

prepare_parser(*parser*)

pretty(*n*)

pretty_dict_ok_error(*n*)

pretty_list(*n*)

process_cmdline_config(*argv*)

prog_name = 'celery'

respects_app_option = True

run(**args*, ***options*)

run_from_argv(*prog_name*, *argv=None*, *command=None*)

say_chat(*direction*, *title*, *body=''*)

say_remote_command_reply(*replies*)

setup_app_from_commandline(*argv*)

show_body = True

show_reply = True

supports_args = True
   If false the parser will raise an exception if positional args are provided.

symbol_by_name(*name*, *imp=<function import_from_cwd>*)

usage(*command*)

verify_args(*given*, *_index=0*)

version = '4.4.1 (cliffs)'
   Application version.

with_pool_option(*argv*)
   Return tuple of (`short_opts`, `long_opts`).

   Returns only if the command supports a pool argument, and used to monkey patch eventlet/gevent environments as early as possible.

Example

```
>>> has_pool_option = (['-P'], ['--pool'])
```

class celery.bin.base.**Option**(**opts*, ***attrs*)

   Instance attributes: _short_opts : [string] _long_opts : [string]

   action : string type : string dest : string default : any nargs : int const : any choices : [string] callback :
   function callback_args : (any*) callback_kwargs : { string : any } help : string metavar : string

```
    ACTIONS = ('store', 'store_const', 'store_true', 'store_false', 'append', 'append_const
    ALWAYS_TYPED_ACTIONS = ('store', 'append')
    ATTRS = ['action', 'type', 'dest', 'default', 'nargs', 'const', 'choices', 'callback',
    CHECK_METHODS = [<function Option._check_action>, <function Option._check_type>, <funct
    CONST_ACTIONS = ('store_const', 'append_const')
    STORE_ACTIONS = ('store', 'store_const', 'store_true', 'store_false', 'append', 'append
    TYPED_ACTIONS = ('store', 'append', 'callback')
    TYPES = ('string', 'int', 'long', 'float', 'complex', 'choice')
    TYPE_CHECKER = {'choice':  <function check_choice>, 'complex':  <function check_builti
    check_value(opt, value)
    convert_value(opt, value)
    get_opt_string()
    process(opt, value, values, parser)
    take_action(action, dest, opt, value, values, parser)
    takes_value()
```

celery.bin.base.**daemon_options**(*parser*, *default_pidfile=None*, *default_logfile=None*)
    Add daemon options to argparse parser.

3.11.63 `celery.bin.celery`

- *Preload Options*
- *Daemon Options*
- `celery inspect`
- `celery control`
- `celery migrate`
- `celery upgrade`
- `celery shell`
- `celery result`
- `celery purge`
- `celery call`

The **celery** umbrella command.

Preload Options

These options are supported by all commands, and usually parsed before command-specific arguments.

-A, --app
    app instance to use (e.g., `module.attr_name`)

-b, --broker
    URL to broker. default is `amqp://guest@localhost//`

--loader
    name of custom loader class to use.

--config
    Name of the configuration module

-C, --no-color
    Disable colors in output.

-q, --quiet
    Give less verbose output (behavior depends on the sub command).

--help
    Show help and exit.

Daemon Options

These options are supported by commands that can detach into the background (daemon). They will be present in any command that also has a *–detach* option.

-f, --logfile
    Path to log file. If no logfile is specified, *stderr* is used.

--pidfile
    Optional file used to store the process pid.

    The program won't start if this file already exists and the pid is still alive.

--uid
    User id, or user name of the user to run as after detaching.

--gid
    Group id, or group name of the main group to change to after detaching.

--umask
    Effective umask (in octal) of the process after detaching. Inherits the umask of the parent process by default.

--workdir
    Optional directory to change to after detaching.

--executable
    Executable to use for the detached process.

celery inspect

-t, --timeout
    Timeout in seconds (float) waiting for reply

-d, --destination
    Comma separated list of destination node names.

-j, --json
    Use json as output format.

celery control

-t, --timeout
Timeout in seconds (float) waiting for reply

-d, --destination
Comma separated list of destination node names.

-j, --json
Use json as output format.

celery migrate

-n, --limit
Number of tasks to consume (int).

-t, -timeout
Timeout in seconds (float) waiting for tasks.

-a, --ack-messages
Ack messages from source broker.

-T, --tasks
List of task names to filter on.

-Q, --queues
List of queues to migrate.

-F, --forever
Continually migrate tasks until killed.

celery upgrade

--django
Upgrade a Django project.

--compat
Maintain backwards compatibility.

--no-backup
Don't backup original files.

celery shell

-I, --ipython
Force iPython implementation.

-B, --bpython
Force bpython implementation.

-P, --python
Force default Python shell.

-T, --without-tasks
Don't add tasks to locals.

--eventlet
Use eventlet monkey patches.

--gevent
    Use gevent monkey patches.

celery result

-t, --task
    Name of task (if custom backend).

--traceback
    Show traceback if any.

celery purge

-f, --force
    Don't prompt for verification before deleting messages (DANGEROUS)

celery call

-a, --args
    Positional arguments (json format).

-k, --kwargs
    Keyword arguments (json format).

--eta
    Scheduled time in ISO-8601 format.

--countdown
    ETA in seconds from now (float/int).

--expires
    Expiry time in float/int seconds, or a ISO-8601 date.

--serializer
    Specify serializer to use (default is json).

--queue
    Destination queue.

--exchange
    Destination exchange (defaults to the queue exchange).

--routing-key
    Destination routing key (defaults to the queue routing key).

class celery.bin.celery.**CeleryCommand**(*app=None*, *get_app=None*, *no_color=False*, *stdout=None*, *stderr=None*, *quiet=False*, *on_error=None*, *on_usage_error=None*)
    Base class for commands.

commands = {'amqp': <class 'celery.bin.amqp.amqp'>, 'beat': <class 'celery.bin.beat.

enable_config_from_cmdline = True

execute(*command*, *argv=None*)

execute_from_commandline(*argv=None*)
    Execute application from command-line.

> > Parameters **argv** (`List[str]`) – The list of command-line arguments. Defaults to `sys.argv`.

ext_fmt = '{self.namespace}.commands'

classmethod get_command_info(*command*, *indent=0*, *color=None*, *colored=None*, *app=None*)

handle_argv(*prog_name*, *argv*, ***kwargs*)

Parse arguments from argv and dispatch to `run()`.

> **Warning:** Exits with an error message if `supports_args` is disabled and `argv` contains positional arguments.

> > Parameters
> >
> > - **prog_name** (`str`) – The program name (`argv[0]`).
> >
> > - **argv** (`List[str]`) – Rest of command-line arguments.

classmethod list_commands(*indent=0*, *colored=None*, *app=None*)

load_extension_commands()

namespace = 'celery'

on_concurrency_setup()

on_usage_error(*exc*, *command=None*)

prepare_prog_name(*name*)

prog_name = 'celery'

classmethod register_command(*fun*, *name=None*)

with_pool_option(*argv*)

Return tuple of (`short_opts`, `long_opts`).

Returns only if the command supports a pool argument, and used to monkey patch eventlet/gevent environments as early as possible.

Example

```
>>> has_pool_option = (['-P'], ['--pool'])
```

celery.bin.celery.**main**(*argv=None*)

Start celery umbrella command.

3.11.64 `celery.bin.worker`

Program used to start a Celery worker instance.

The **celery worker** command (previously known as `celeryd`)

See also:

See *Preload Options*.

-c, --concurrency
    Number of child processes processing the queue. The default is the number of CPUs available on your system.

-P, --pool
    Pool implementation:

    prefork (default), eventlet, gevent, threads or solo.

-n, --hostname
    Set custom hostname (e.g., 'w1@%%h'). Expands: %%h (hostname), %%n (name) and %%d, (domain).

-B, --beat
    Also run the *celery beat* periodic task scheduler. Please note that there must only be one instance of this service.

    Note: -B is meant to be used for development purposes. For production environment, you need to start **celery beat** separately.

-Q, --queues
    List of queues to enable for this worker, separated by comma. By default all configured queues are enabled.
    Example: *-Q video,image*

-X, --exclude-queues
    List of queues to disable for this worker, separated by comma. By default all configured queues are enabled.
    Example: *-X video,image*.

-I, --include
    Comma separated list of additional modules to import. Example: -I foo.tasks,bar.tasks

-s, --schedule
    Path to the schedule database if running with the *-B* option. Defaults to *celerybeat-schedule*. The extension
    ".db" may be appended to the filename.

-O
    Apply optimization profile. Supported: default, fair

--prefetch-multiplier
    Set custom prefetch multiplier value for this worker instance.

--scheduler
    Scheduler class to use. Default is *celery.beat.PersistentScheduler*

-S, --statedb
    Path to the state database. The extension '.db' may be appended to the filename. Default: {default}

-E, --task-events
    Send task-related events that can be captured by monitors like **celery events**, *celerymon*, and others.

--without-gossip
    Don't subscribe to other workers events.

--without-mingle
    Don't synchronize with other workers at start-up.

--without-heartbeat
    Don't send event heartbeats.

--heartbeat-interval
    Interval in seconds at which to send worker heartbeat

--purge
> Purges all waiting tasks before the daemon is started. **WARNING**: This is unrecoverable, and the tasks will be deleted from the messaging server.

--time-limit
> Enables a hard time limit (in seconds int/float) for tasks.

--soft-time-limit
> Enables a soft time limit (in seconds int/float) for tasks.

--max-tasks-per-child
> Maximum number of tasks a pool worker can execute before it's terminated and replaced by a new worker.

--max-memory-per-child
> Maximum amount of resident memory, in KiB, that may be consumed by a child process before it will be replaced by a new one. If a single task causes a child process to exceed this limit, the task will be completed and the child process will be replaced afterwards. Default: no limit.

--autoscale
> Enable autoscaling by providing max_concurrency, min_concurrency. Example:

```
--autoscale=10,3
```

> (always keep 3 processes, but grow to 10 if necessary)

--detach
> Start worker as a background process.

-f, --logfile
> Path to log file. If no logfile is specified, *stderr* is used.

-l, --loglevel
> Logging level, choose between *DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*, or *FATAL*.

--pidfile
> Optional file used to store the process pid.
>
> The program won't start if this file already exists and the pid is still alive.

--uid
> User id, or user name of the user to run as after detaching.

--gid
> Group id, or group name of the main group to change to after detaching.

--umask
> Effective `umask(1)` (in octal) of the process after detaching. Inherits the `umask(1)` of the parent process by default.

--workdir
> Optional directory to change to after detaching.

--executable
> Executable to use for the detached process.

class `celery.bin.worker.`**`worker`**(*app=None*, *get_app=None*, *no_color=False*, *stdout=None*, *stderr=None*, *quiet=False*, *on_error=None*, *on_usage_error=None*)
> Start worker instance.

Examples

```
$ celery worker --app=proj -l info
$ celery worker -A proj -l info -Q hipri,lopri

$ celery worker -A proj --concurrency=4
$ celery worker -A proj --concurrency=1000 -P eventlet
$ celery worker --autoscale=10,0
```

add_arguments(*parser*)

doc = 'Program used to start a Celery worker instance.\n\nThe :program:`celery worker`

enable_config_from_cmdline = True

maybe_detach(*argv*, *dopts=None*)

namespace = 'worker'

removed_flags = {'--force-execv', '--no-execv'}

run(*hostname=None*, *pool_cls=None*, *app=None*, *uid=None*, *gid=None*, *loglevel=None*, *logfile=None*, *pidfile=None*, *statedb=None*, ***kwargs*)

run_from_argv(*prog_name*, *argv=None*, *command=None*)

supports_args = False

with_pool_option(*argv*)
Return tuple of (short_opts, long_opts).

Returns only if the command supports a pool argument, and used to monkey patch eventlet/gevent environments as early as possible.

Example

```
>>> has_pool_option = (['-P'], ['--pool'])
```

celery.bin.worker.**main**(*app=None*)
Start worker.

3.11.65 `celery.bin.beat`

The **celery beat** command.

See also:

See *Preload Options* and *Daemon Options*.

--detach
Detach and run in the background as a daemon.

-s, --schedule
Path to the schedule database. Defaults to *celerybeat-schedule*. The extension '.db' may be appended to the filename. Default is {default}.

-S, --scheduler
Scheduler class to use. Default is {default}.

--max-interval
Max seconds to sleep between schedule iterations.

-f, --logfile
    Path to log file. If no logfile is specified, *stderr* is used.

-l, --loglevel
    Logging level, choose between *DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*, or *FATAL*.

--pidfile
    File used to store the process pid. Defaults to *celerybeat.pid*.

    The program won't start if this file already exists and the pid is still alive.

--uid
    User id, or user name of the user to run as after detaching.

--gid
    Group id, or group name of the main group to change to after detaching.

--umask
    Effective umask (in octal) of the process after detaching. Inherits the umask of the parent process by default.

--workdir
    Optional directory to change to after detaching.

--executable
    Executable to use for the detached process.

class `celery.bin.beat`.**beat**(*app=None,   get_app=None,   no_color=False,   stdout=None,*
                                stderr=None, quiet=False, on_error=None, on_usage_error=None)
    Start the beat periodic task scheduler.

Examples

```
$ celery beat -l info
$ celery beat -s /var/run/celery/beat-schedule --detach
$ celery beat -S django
```

The last example requires the django-celery-beat extension package found on PyPI.

add_arguments(*parser*)

doc = "The :program:`celery beat` command.\n\n.. program:: celery beat\n\n.. seealso:

enable_config_from_cmdline = True

run(*detach=False, logfile=None, pidfile=None, uid=None, gid=None, umask=None, workdir=None,*
       ***kwargs*)

supports_args = False

3.11.66 `celery.bin.events`

The **celery events** command.

See also:

See *Preload Options* and *Daemon Options*.

-d, --dump
    Dump events to stdout.

-c, --camera
    Take snapshots of events using this camera.

--detach
> Camera: Detach and run in the background as a daemon.

-F, --freq, --frequency
> Camera: Shutter frequency. Default is every 1.0 seconds.

-r, --maxrate
> Camera: Optional shutter rate limit (e.g., 10/m).

-l, --loglevel
> Logging level, choose between *DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*, or *FATAL*. Default is INFO.

-f, --logfile
> Path to log file. If no logfile is specified, *stderr* is used.

--pidfile
> Optional file used to store the process pid.
>
> The program won't start if this file already exists and the pid is still alive.

--uid
> User id, or user name of the user to run as after detaching.

--gid
> Group id, or group name of the main group to change to after detaching.

--umask
> Effective umask (in octal) of the process after detaching. Inherits the umask of the parent process by default.

--workdir
> Optional directory to change to after detaching.

--executable
> Executable to use for the detached process.

class celery.bin.events.**events**(*app=None*, *get_app=None*, *no_color=False*, *stdout=None*, *stderr=None*, *quiet=False*, *on_error=None*, *on_usage_error=None*)
> Event-stream utilities.

> ### Notes

> ```
> # - Start graphical monitor (requires curses)
> $ celery events --app=proj
> $ celery events -d --app=proj
> # - Dump events to screen.
> $ celery events -b amqp://
> # - Run snapshot camera.
> $ celery events -c <camera> [options]
> ```

> ### Examples

> ```
> $ celery events
> $ celery events -d
> $ celery events -c mod.attr -F 1.0 --detach --maxrate=100/m -l info
> ```

> **add_arguments**(*parser*)

> **doc** = "The :program:`celery events` command.\n\n.. program:: celery events\n\n.. seea

run(*dump=False*, *camera=None*, *frequency=1.0*, *maxrate=None*, *loglevel='INFO'*, *logfile=None*, *prog_name='celery events'*, *pidfile=None*, *uid=None*, *gid=None*, *umask=None*, *workdir=None*, *detach=False*, ***kwargs*)

run_evcam(*camera*, *logfile=None*, *pidfile=None*, *uid=None*, *gid=None*, *umask=None*, *workdir=None*, *detach=False*, ***kwargs*)

run_evdump()

run_evtop()

set_process_status(*prog*, *info="*)

supports_args = False

3.11.67 `celery.bin.logtool`

The **celery logtool** command.

class celery.bin.logtool.**logtool**(*app=None*, *get_app=None*, *no_color=False*, *stdout=None*, *stderr=None*, *quiet=False*, *on_error=None*, *on_usage_error=None*)

The celery logtool command.

args = '<action> [arguments]\n ..... stats [file1|- [file2 [...]]]\n ..... traces [fil

debug(*files*)

errors(*files*)

incomplete(*files*)

run(*what=None*, **files*, ***kwargs*)

say1(*line*, **_*)

stats(*files*)

traces(*files*)

3.11.68 `celery.bin.amqp`

The **celery amqp** command.

class celery.bin.amqp.**AMQPAdmin**(**args*, ***kwargs*)

The celery **celery amqp** utility.

Shell

alias of *AMQShell*

connect(*conn=None*)

note(*m*)

run()

class celery.bin.amqp.**AMQShell**(**args*, ***kwargs*)

AMQP API Shell.

Parameters

- **connect** (*Callable*) – Function used to connect to the server. Must return kombu.Connection object.

- **silent** (*[bool](#)*) – If enabled, the commands won't have annoying output not relevant
  when running in non-shell mode.

amqp = {'basic.ack': <celery.bin.amqp.Spec object>, 'basic.get': <celery.bin.amqp.Sp
  Map of AMQP API commands and their *[Spec](#)*.

builtins = {'EOF': 'do_exit', 'exit': 'do_exit', 'help': 'do_help'}
  Map of built-in command names -> method names

chan = None

completenames(*text*, **ignored*)
  Return all commands starting with *text*, for tab-completion.

conn = None

counter = 1

default(*line*)
  Called on an input line when the command prefix is not recognized.

  If this method is not overridden, it prints an error message and returns.

dispatch(*cmd*, *arglist*)
  Dispatch and execute the command.

  Look-up order is: *[builtins](#)* -> *[amqp](#)*.

display_command_help(*cmd*, *short=False*)

do_exit(**args*)
  The *'exit'* command.

do_help(**args*)
  List available commands with "help" or detailed help with "help cmd".

get_amqp_api_command(*cmd*, *arglist*)
  Get AMQP command wrapper.

  With a command name and a list of arguments, convert the arguments to Python values and find the
  corresponding method on the AMQP channel object.

  > **Returns** of *(method, processed_args)* pairs.

  > **Return type** Tuple

get_names()

identchars = '.'

inc_counter = count(2)

needs_reconnect = False

note(*m*)
  Say something to the user. Disabled if `silent`.

onecmd(*line*)
  Parse line and execute command.

parseline(*parts*)
  Parse input line.

  > **Returns**

  > > **of three items:** *(command_name, arglist, original_line)*

> > **Return type** Tuple

> **prompt**
>> str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

>> Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

> **prompt_fmt = '{self.counter}> '**

> **respond**(*retval*)
>> What to do with the return value of a command.

> **say**(*m*)

class `celery.bin.amqp.`**Spec**(**args*, ***kwargs*)
> AMQP Command specification.

> Used to convert arguments to Python values and display various help and tool-tips.

> > **Parameters**

> > > • **args** (*Sequence*) – see *args*.

> > > • **returns** (*str*) – see *returns*.

> **args = None**
>> List of arguments this command takes. Should contain (`argument_name`, `argument_type`) tuples.

> **coerce**(*index*, *value*)
>> Coerce value for argument at index.

> **format_arg**(*name*, *type*, *default_value=None*)

> **format_response**(*response*)
>> Format the return value of this command in a human-friendly way.

> **format_signature**()

> **returns = None**
>> Helpful human string representation of what this command returns. May be `None`, to signify the return type is unknown.

> **str_args_to_python**(*arglist*)
>> Process list of string arguments to values according to spec.

> > ### Example

> > ```
> > >>> spec = Spec([('queue', str), ('if_unused', bool)])
> > >>> spec.str_args_to_python('pobox', 'true')
> > ('pobox', True)
> > ```

class `celery.bin.amqp.`**amqp**(*app=None*, *get_app=None*, *no_color=False*, *stdout=None*, *stderr=None*, *quiet=False*, *on_error=None*, *on_usage_error=None*)
> AMQP Administration Shell.

> Also works for non-AMQP transports (but not ones that store declarations in memory).

Examples

```
$ # start shell mode
$ celery amqp
$ # show list of commands
$ celery amqp help

$ celery amqp exchange.delete name
$ celery amqp queue.delete queue
$ celery amqp queue.delete queue yes yes
```

run(**args*, ***options*)

3.11.69 `celery.bin.graph`

The **celery graph** command.

class celery.bin.graph.**graph**(*app=None*, *get_app=None*, *no_color=False*, *std-out=None*, *stderr=None*, *quiet=False*, *on_error=None*, *on_usage_error=None*)

The celery graph command.

args = '<TYPE> [arguments]\n ..... bootsteps [worker] [consumer]\n ..... workers [enum

bootsteps(**args*, ***kwargs*)

run(*what=None*, **args*, ***kwargs*)

workers(**args*, ***kwargs*)

3.11.70 `celery.bin.multi`

Start multiple worker instances from the command-line.

Examples

```
$ # Single worker with explicit name and events enabled.
$ celery multi start Leslie -E

$ # Pidfiles and logfiles are stored in the current directory
$ # by default.  Use --pidfile and --logfile argument to change
$ # this.  The abbreviation %n will be expanded to the current
$ # node name.
$ celery multi start Leslie -E --pidfile=/var/run/celery/%n.pid
                           --logfile=/var/log/celery/%n%I.log

$ # You need to add the same arguments when you restart,
$ # as these aren't persisted anywhere.
$ celery multi restart Leslie -E --pidfile=/var/run/celery/%n.pid
                             --logfile=/var/log/celery/%n%I.log

$ # To stop the node, you need to specify the same pidfile.
$ celery multi stop Leslie --pidfile=/var/run/celery/%n.pid

$ # 3 workers, with 3 processes each
```

(continues on next page)

```
$ celery multi start 3 -c 3
celery worker -n celery1@myhost -c 3
celery worker -n celery2@myhost -c 3
celery worker -n celery3@myhost -c 3

$ # start 3 named workers
$ celery multi start image video data -c 3
celery worker -n image@myhost -c 3
celery worker -n video@myhost -c 3
celery worker -n data@myhost -c 3

$ # specify custom hostname
$ celery multi start 2 --hostname=worker.example.com -c 3
celery worker -n celery1@worker.example.com -c 3
celery worker -n celery2@worker.example.com -c 3

$ # specify fully qualified nodenames
$ celery multi start foo@worker.example.com bar@worker.example.com -c 3

$ # fully qualified nodenames but using the current hostname
$ celery multi start foo@%h bar@%h

$ # Advanced example starting 10 workers in the background:
$ #   * Three of the workers processes the images and video queue
$ #   * Two of the workers processes the data queue with loglevel DEBUG
$ #   * the rest processes the default' queue.
$ celery multi start 10 -l INFO -Q:1-3 images,video -Q:4,5 data
    -Q default -L:4,5 DEBUG

$ # You can show the commands necessary to start the workers with
$ # the 'show' command:
$ celery multi show 10 -l INFO -Q:1-3 images,video -Q:4,5 data
    -Q default -L:4,5 DEBUG

$ # Additional options are added to each celery worker' comamnd,
$ # but you can also modify the options for ranges of, or specific workers

$ # 3 workers: Two with 3 processes, and one with 10 processes.
$ celery multi start 3 -c 3 -c:1 10
celery worker -n celery1@myhost -c 10
celery worker -n celery2@myhost -c 3
celery worker -n celery3@myhost -c 3

$ # can also specify options for named workers
$ celery multi start image video data -c 3 -c:image 10
celery worker -n image@myhost -c 10
celery worker -n video@myhost -c 3
celery worker -n data@myhost -c 3

$ # ranges and lists of workers in options is also allowed:
$ # (-c:1-3 can also be written as -c:1,2,3)
$ celery multi start 5 -c 3  -c:1-3 10
celery worker -n celery1@myhost -c 10
celery worker -n celery2@myhost -c 10
celery worker -n celery3@myhost -c 10
celery worker -n celery4@myhost -c 3
celery worker -n celery5@myhost -c 3
```

```
$ # lists also works with named workers
$ celery multi start foo bar baz xuzzy -c 3 -c:foo,bar,baz 10
celery worker -n foo@myhost -c 10
celery worker -n bar@myhost -c 10
celery worker -n baz@myhost -c 10
celery worker -n xuzzy@myhost -c 3
```

class celery.bin.multi.**MultiTool**(*env=None*, *cmd=None*, *fh=None*, *stdout=None*, *stderr=None*, ***kwargs*)

The celery multi program.

Cluster(*nodes*, *cmd=None*)

DOWN

FAILED

class MultiParser(*cmd='celery worker'*, *append=''*, *prefix=''*, *suffix=''*, *range_prefix='celery'*)

class Node(*name*, *cmd=None*, *append=None*, *options=None*, *extra_args=None*)

Represents a node in a cluster.

alive()

argv_with_executable

executable

classmethod from_kwargs(*name*, ***kwargs*)

getopt(**alt*)

handle_process_exit(*retcode*, *on_signalled=None*, *on_failure=None*)

logfile

pid

pidfile

prepare_argv(*argv*, *path*)

send(*sig*, *on_error=None*)

start(*env=None*, ***kwargs*)

parse(*p*)

OK

OptionParser

alias of celery.apps.multi.NamespacedOptionParser

call_command(*command*, *argv*)

cluster_from_argv(*argv*, *cmd=None*)

execute_from_commandline(*argv*, *cmd=None*)

expand(*template*, **argv*)

get(*wanted*, **argv*)

help(**argv*)

kill(*cluster*)

names(*cluster*)

on_child_failure(*node*, *retcode*)

on_child_signalled(*node*, *signum*)

on_child_spawn(*node*, *argstr*, *env*)

on_node_down(*node*)

on_node_restart(*node*)

on_node_shutdown_ok(*node*)

on_node_signal(*node*, *sig*)

on_node_signal_dead(*node*)

on_node_start(*node*)

on_node_status(*node*, *retval*)

on_send_signal(*node*, *sig*)

on_still_waiting_end()

on_still_waiting_for(*nodes*)

on_still_waiting_progress(*nodes*)

on_stopping_preamble(*nodes*)

reserved_options = [('--nosplash', 'nosplash'), ('--quiet', 'quiet'), ('-q', 'quiet'),

restart(*cluster*, *sig*, ***kwargs*)

show(*cluster*)

start(*cluster*)

stop(*cluster*, *sig*, ***kwargs*)

stop_verify(*cluster*, *sig*, ***kwargs*)

stopwait(*cluster*, *sig*, ***kwargs*)

validate_arguments(*argv*)

3.11.71 `celery.bin.call`

The `celery call` program used to send tasks from the command-line.

class `celery.bin.call.`**call**(*app=None*, *get_app=None*, *no_color=False*, *stdout=None*, *stderr=None*, *quiet=False*, *on_error=None*, *on_usage_error=None*)

Call a task by name.

Examples

```
$ celery call tasks.add --args='[2, 2]'
$ celery call tasks.add --args='[2, 2]' --countdown=10
```

add_arguments(*parser*)

```
args = '<task_name>'
args_name = 'posargs'
```

run(*name*, **_*, ***kwargs*)

3.11.72 `celery.bin.control`

The `celery control,` `. inspect` and `. status` programs.

class `celery.bin.control.`**`control`**(**args*, ***kwargs*)

Workers remote control.

Availability: RabbitMQ (AMQP), Redis, and MongoDB transports.

Examples

```
$ celery control enable_events --timeout=5
$ celery control -d worker1@example.com enable_events
$ celery control -d w1.e.com,w2.e.com enable_events

$ celery control -d w1.e.com add_consumer queue_name
$ celery control -d w1.e.com cancel_consumer queue_name

$ celery control add_consumer queue exchange direct rkey
```

call(*method*, *arguments*, ***options*)

```
control_group = 'control'
name = 'control'
```

class `celery.bin.control.`**`inspect`**(**args*, ***kwargs*)

Inspect the worker at runtime.

Availability: RabbitMQ (AMQP) and Redis transports.

Examples

```
$ celery inspect active --timeout=5
$ celery inspect scheduled -d worker1@example.com
$ celery inspect revoked -d w1@e.com,w2@e.com
```

call(*method*, *arguments*, ***options*)

```
control_group = 'inspect'
name = 'inspect'
```

class `celery.bin.control.`**`status`**(*app=None*, *get_app=None*, *no_color=False*, *std-out=None*, *stderr=None*, *quiet=False*, *on_error=None*, *on_usage_error=None*)

Show list of workers that are online.

```
option_list = None
```

run(**args*, ***kwargs*)

3.11.73 `celery.bin.list`

The `celery list bindings` command, used to inspect queue bindings.

class `celery.bin.list.`**`list_`**(*app=None*, *get_app=None*, *no_color=False*, *stdout=None*, *stderr=None*, *quiet=False*, *on_error=None*, *on_usage_error=None*)
Get info from broker.

Note: For RabbitMQ the management plugin is required.

Example

```
$ celery list bindings
```

`args = '[bindings]'`

`list_bindings`(*management*)

`run`(*what=None*, **_*, ***kw*)

3.11.74 `celery.bin.migrate`

The `celery migrate` command, used to filter and move messages.

class `celery.bin.migrate.`**`migrate`**(*app=None*, *get_app=None*, *no_color=False*, *stdout=None*, *stderr=None*, *quiet=False*, *on_error=None*, *on_usage_error=None*)
Migrate tasks from one broker to another.

Warning: This command is experimental, make sure you have a backup of the tasks before you continue.

Example

```
$ celery migrate amqp://A.example.com amqp://guest@B.example.com//
$ celery migrate redis://localhost amqp://guest@localhost//
```

`add_arguments`(*parser*)

`args = '<source_url> <dest_url>'`

`on_migrate_task`(*state*, *body*, *message*)

`progress_fmt = 'Migrating task {state.count}/{state.strtotal}:  {body[task]}[{body[id]`

`run`(*source*, *destination*, ***kwargs*)

3.11.75 `celery.bin.purge`

The `celery purge` program, used to delete messages from queues.

class celery.bin.purge.**purge**(*app=None*, *get_app=None*, *no_color=False*, *std-out=None*, *stderr=None*, *quiet=False*, *on_error=None*, *on_usage_error=None*)

Erase all messages from all known task queues.

> **Warning:** There's no undo operation for this command.

add_arguments(*parser*)

fmt_empty = 'No messages purged from {qnum} {queues}'

fmt_purged = 'Purged {mnum} {messages} from {qnum} known task {queues}.'

run(*force=False*, *queues=None*, *exclude_queues=None*, ***kwargs*)

warn_prelude = '{warning}: This will remove all tasks from {queues}: {names}.\n Ther

warn_prompt = 'Are you sure you want to delete all tasks'

3.11.76 `celery.bin.result`

The `celery result` program, used to inspect task results.

class celery.bin.result.**result**(*app=None*, *get_app=None*, *no_color=False*, *std-out=None*, *stderr=None*, *quiet=False*, *on_error=None*, *on_usage_error=None*)

Gives the return value for a given task id.

Examples

```
$ celery result 8f511516-e2f5-4da4-9d2f-0fb83a86e500
$ celery result 8f511516-e2f5-4da4-9d2f-0fb83a86e500 -t tasks.add
$ celery result 8f511516-e2f5-4da4-9d2f-0fb83a86e500 --traceback
```

add_arguments(*parser*)

args = '<task_id>'

run(*task_id*, **args*, ***kwargs*)

3.11.77 `celery.bin.shell`

The `celery shell` program, used to start a REPL.

class celery.bin.shell.**shell**(*app=None*, *get_app=None*, *no_color=False*, *std-out=None*, *stderr=None*, *quiet=False*, *on_error=None*, *on_usage_error=None*)

Start shell session with convenient access to celery symbols.

The following symbols will be added to the main globals:

- `celery`: the current application.
- `chord`, `group`, `chain`, `chunks`, `xmap`, `xstarmap` `subtask`, `Task`
- all registered tasks.

add_arguments(*parser*)

invoke_bpython_shell()

invoke_default_shell()

invoke_fallback_shell()

invoke_ipython_shell()

run(**args*, ***kwargs*)

3.11.78 `celery.bin.upgrade`

The `celery upgrade` command, used to upgrade from previous versions.

class `celery.bin.upgrade.`**upgrade**(*app=None*, *get_app=None*, *no_color=False*, *stdout=None*, *stderr=None*, *quiet=False*, *on_error=None*, *on_usage_error=None*)

Perform upgrade between versions.

add_arguments(*parser*)

choices = {'settings'}

run(**args*, ***kwargs*)

settings(*command*, *filename=None*, *no_backup=False*, *django=False*, *compat=False*, ***kwargs*)

usage(*command*)

3.12 Internals

Release 4.4

Date Mar 02, 2020

3.12.1 Contributors Guide to the Code

Philosophy

The API>RCP Precedence Rule

- The API is more important than Readability

- Readability is more important than Convention

- **Convention is more important than Performance**

    – ...unless the code is a proven hot-spot.

More important than anything else is the end-user API. Conventions must step aside, and any suffering is always
alleviated if the end result is a better API.

Conventions and Idioms Used

Classes

Naming

- Follows **PEP 8**.

- Class names must be *CamelCase*.

- but not if they're verbs, verbs shall be *lower_case*:

```python
# - test case for a class
class TestMyClass(Case):              # BAD
    pass

class test_MyClass(Case):             # GOOD
    pass

# - test case for a function
class TestMyFunction(Case):           # BAD
    pass

class test_my_function(Case):         # GOOD
    pass

# - "action" class (verb)
class UpdateTwitterStatus(object):    # BAD
    pass

class update_twitter_status(object):    # GOOD
    pass
```

Note: Sometimes it makes sense to have a class mask as a function, and there's precedence for this
in the Python standard library (e.g., `contextmanager`). Celery examples include *signature*,
chord, `inspect`, `promise` and more..

- Factory functions and methods must be *CamelCase* (excluding verbs):

```python
class Celery(object):

    def consumer_factory(self):     # BAD
        ...

    def Consumer(self):             # GOOD
        ...
```

Default values

Class attributes serve as default values for the instance, as this means that they can be set by either instantiation or inheritance.

Example:

```python
class Producer(object):
    active = True
    serializer = 'json'

    def __init__(self, serializer=None, active=None):
        self.serializer = serializer or self.serializer

        # must check for None when value can be false-y
        self.active = active if active is not None else self.active
```

A subclass can change the default value:

```python
TaskProducer(Producer):
    serializer = 'pickle'
```

and the value can be set at instantiation:

```python
>>> producer = TaskProducer(serializer='msgpack')
```

Exceptions

Custom exceptions raised by an objects methods and properties should be available as an attribute and documented in the method/property that throw.

This way a user doesn't have to find out where to import the exception from, but rather use `help(obj)` and access the exception class from the instance directly.

Example:

```python
class Empty(Exception):
    pass

class Queue(object):
    Empty = Empty

    def get(self):
        """Get the next item from the queue.

        :raises Queue.Empty: if there are no more items left.
```

```
        """
        try:
            return self.queue.popleft()
        except IndexError:
            raise self.Empty()
```

Composites

Similarly to exceptions, composite classes should be override-able by inheritance and/or instantiation. Common sense can be used when selecting what classes to include, but often it's better to add one too many: predicting what users need to override is hard (this has saved us from many a monkey patch).

Example:

```python
class Worker(object):
    Consumer = Consumer

    def __init__(self, connection, consumer_cls=None):
        self.Consumer = consumer_cls or self.Consumer

    def do_work(self):
        with self.Consumer(self.connection) as consumer:
            self.connection.drain_events()
```

Applications vs. "single mode"

In the beginning Celery was developed for Django, simply because this enabled us get the project started quickly, while also having a large potential user base.

In Django there's a global settings object, so multiple Django projects can't co-exist in the same process space, this later posed a problem for using Celery with frameworks that don't have this limitation.

Therefore the app concept was introduced. When using apps you use 'celery' objects instead of importing things from Celery sub-modules, this (unfortunately) also means that Celery essentially has two API's.

Here's an example using Celery in single-mode:

```python
from celery import task
from celery.task.control import inspect

from .models import CeleryStats


@task
def write_stats_to_db():
    stats = inspect().stats(timeout=1)
    for node_name, reply in stats:
        CeleryStats.objects.update_stat(node_name, stats)
```

and here's the same using Celery app objects:

```python
from .celery import celery
from .models import CeleryStats
```

```
@app.task
def write_stats_to_db():
    stats = celery.control.inspect().stats(timeout=1)
    for node_name, reply in stats:
        CeleryStats.objects.update_stat(node_name, stats)
```

In the example above the actual application instance is imported from a module in the project, this module could look something like this:

```
from celery import Celery

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

Module Overview

- celery.app

    This is the core of Celery: the entry-point for all functionality.

- celery.loaders

    Every app must have a loader. The loader decides how configuration is read; what happens when the worker starts; when a task starts and ends; and so on.

    The loaders included are:

    - app

        Custom Celery app instances uses this loader by default.

    - default

        "single-mode" uses this loader by default.

    Extension loaders also exist, for example celery-pylons.

- celery.worker

    This is the worker implementation.

- celery.backends

    Task result backends live here.

- celery.apps

    Major user applications: worker and beat. The command-line wrappers for these are in celery.bin (see below)

- celery.bin

    Command-line applications. `setup.py` creates setuptools entry-points for these.

- celery.concurrency

    Execution pool implementations (prefork, eventlet, gevent, solo, thread).

- celery.db

    Database models for the SQLAlchemy database result backend. (should be moved into `celery.backends.database`)

- celery.events

Sending and consuming monitoring events, also includes curses monitor, event dumper and utilities to work with in-memory cluster state.

- celery.execute.trace

  How tasks are executed and traced by the worker, and in eager mode.

- celery.security

  Security related functionality, currently a serializer using cryptographic digests.

- celery.task

  single-mode interface to creating tasks, and controlling workers.

- t.unit (int distribution)

  The unit test suite.

- celery.utils

  Utility functions used by the Celery code base. Much of it is there to be compatible across Python versions.

- celery.contrib

  Additional public code that doesn't fit into any other name-space.

Worker overview

- *celery.bin.worker:Worker*

  This is the command-line interface to the worker.

  Responsibilities:

  - Daemonization when `--detach` set,
  - dropping privileges when using `--uid`/`--gid` arguments
  - Installs "concurrency patches" (eventlet/gevent monkey patches).

  `app.worker_main(argv)` calls `instantiate('celery.bin.worker:Worker')(app).execute_from_commandline(argv)`

- *app.Worker -> celery.apps.worker:Worker*

  Responsibilities: * sets up logging and redirects standard outs * installs signal handlers (*TERM*/*HUP*/*STOP*/*USR1* (cry)/*USR2* (rdb)) * prints banner and warnings (e.g., pickle warning) * handles the `celery worker --purge` argument

- *app.WorkController -> celery.worker.WorkController*

  This is the real worker, built up around bootsteps.

3.12.2 Celery Deprecation Time-line

Removals for version 5.0

Old Task API

Compat Task Modules

- Module `celery.decorators` will be removed:

    This means you need to change:

    ```python
    from celery.decorators import task
    ```

    Into:

    ```python
    from celery import task
    ```

- Module `celery.task` *may* be removed (not decided)

    This means you should change:

    ```python
    from celery.task import task
    ```

    into:

    ```python
    from celery import task
    ```

    —and:

    ```python
    from celery.task import Task
    ```

    into:

    ```python
    from celery import Task
    ```

Note that the new `Task` class no longer uses `classmethod()` for these methods:

- delay

- apply_async

- retry

- apply

- AsyncResult

- subtask

This also means that you can't call these methods directly on the class, but have to instantiate the task first:

```
>>> MyTask.delay()          # NO LONGER WORKS


>>> MyTask().delay()        # WORKS!
```

Task attributes

The task attributes:

- `queue`

- `exchange`

- `exchange_type`

- `routing_key`

- `delivery_mode`

- `priority`

is deprecated and must be set by *task_routes* instead.

Modules to Remove

- `celery.execute`

    This module only contains `send_task`: this must be replaced with *app.send_task* instead.

- `celery.decorators`

    See *Compat Task Modules*

- `celery.log`

    Use *app.log* instead.

- `celery.messaging`

    Use *app.amqp* instead.

- `celery.registry`

    Use *celery.app.registry* instead.

- `celery.task.control`

    Use *app.control* instead.

- `celery.task.schedules`

    Use *celery.schedules* instead.

- `celery.task.chords`

    Use *celery.chord()* instead.

Settings

BROKER Settings

Setting name	Replace with
BROKER_HOST	*broker_url*
BROKER_PORT	*broker_url*
BROKER_USER	*broker_url*
BROKER_PASSWORD	*broker_url*
BROKER_VHOST	*broker_url*

REDIS Result Backend Settings

Setting name	Replace with
CELERY_REDIS_HOST	*result_backend*
CELERY_REDIS_PORT	*result_backend*
CELERY_REDIS_DB	*result_backend*
CELERY_REDIS_PASSWORD	*result_backend*
REDIS_HOST	*result_backend*
REDIS_PORT	*result_backend*
REDIS_DB	*result_backend*
REDIS_PASSWORD	*result_backend*

Task_sent signal

The *task_sent* signal will be removed in version 4.0. Please use the *before_task_publish* and *after_task_publish* signals instead.

Result

Apply to: *AsyncResult*, *EagerResult*:

- Result.wait() -> Result.get()

- Result.task_id() -> Result.id

- Result.status -> Result.state.

Settings

Setting name	Replace with
CELERY_AMQP_TASK_RESULT_EXPIRES	*result_expires*

Removals for version 2.0

- The following settings will be removed:

Setting name	Replace with
CELERY_AMQP_CONSUMER_QUEUES	*task_queues*
CELERY_AMQP_CONSUMER_QUEUES	*task_queues*
CELERY_AMQP_EXCHANGE	*task_default_exchange*
CELERY_AMQP_EXCHANGE_TYPE	*task_default_exchange_type*
CELERY_AMQP_CONSUMER_ROUTING_KEY	*task_queues*
CELERY_AMQP_PUBLISHER_ROUTING_KEY	*task_default_routing_key*

- `CELERY_LOADER` definitions without class name.

  For example,, *celery.loaders.default*, needs to include the class name: *celery.loaders.default.Loader*.

- **TaskSet.run().** Use **celery.task.base.TaskSet.apply_async()** instead.

3.12.3 Internals: The worker

- *Introduction*
- *Data structures*
    - *timer*
- *Components*
    - *Consumer*
    - *Timer*
    - *TaskPool*

Introduction

The worker consists of 4 main components: the consumer, the scheduler, the mediator and the task pool. All these components runs in parallel working with two data structures: the ready queue and the ETA schedule.

Data structures

timer

The timer uses `heapq` to schedule internal functions. It's very efficient and can handle hundred of thousands of entries.

Components

Consumer

Receives messages from the broker using Kombu.

When a message is received it's converted into a `celery.worker.request.Request` object.

Tasks with an ETA, or rate-limit are entered into the *timer*, messages that can be immediately processed are sent to the execution pool.

ETA and rate-limit when used together will result in the rate limit being observed with the task being scheduled after the ETA.

Timer

The timer schedules internal functions, like cleanup and internal monitoring, but also it schedules ETA tasks and rate limited tasks. If the scheduled tasks ETA has passed it is moved to the execution pool.

TaskPool

This is a slightly modified `multiprocessing.Pool`. It mostly works the same way, except it makes sure all of the workers are running at all times. If a worker is missing, it replaces it with a new one.

3.12.4 Message Protocol

Task messages

Version 2

Definition

```
properties = {
    'correlation_id': uuid task_id,
    'content_type': string mimetype,
```

(continues on next page)

```
    'content_encoding': string encoding,

    # optional
    'reply_to': string queue_or_url,
}
headers = {
    'lang': string 'py'
    'task': string task,
    'id': uuid task_id,
    'root_id': uuid root_id,
    'parent_id': uuid parent_id,
    'group': uuid group_id,

    # optional
    'meth': string method_name,
    'shadow': string alias_name,
    'eta': iso8601 ETA,
    'expires': iso8601 expires,
    'retries': int retries,
    'timelimit': (soft, hard),
    'argsrepr': str repr(args),
    'kwargsrepr': str repr(kwargs),
    'origin': str nodename,
}

body = (
    object[] args,
    Mapping kwargs,
    Mapping embed {
        'callbacks': Signature[] callbacks,
        'errbacks': Signature[] errbacks,
        'chain': Signature[] chain,
        'chord': Signature chord_callback,
    }
)
```

Example

This example sends a task message using version 2 of the protocol:

```python
# chain: add(add(add(2, 2), 4), 8) == 2 + 2 + 4 + 8

import json
import os
import socket

task_id = uuid()
args = (2, 2)
kwargs = {}
basic_publish(
    message=json.dumps((args, kwargs, None)),
    application_headers={
        'lang': 'py',
        'task': 'proj.tasks.add',
        'argsrepr': repr(args),
```

```
        'kwargsrepr': repr(kwargs),
        'origin': '@'.join([os.getpid(), socket.gethostname()])
    }
    properties={
        'correlation_id': task_id,
        'content_type': 'application/json',
        'content_encoding': 'utf-8',
    }
)
```

Changes from version 1

- Protocol version detected by the presence of a `task` message header.

- Support for multiple languages via the `lang` header.

     Worker may redirect the message to a worker that supports the language.

- Meta-data moved to headers.

     This means that workers/intermediates can inspect the message and make decisions based on the headers without decoding the payload (that may be language specific, for example serialized by the Python specific pickle serializer).

- Always UTC

     There's no `utc` flag anymore, so any time information missing timezone will be expected to be in UTC time.

- Body is only for language specific data.

     – Python stores args/kwargs and embedded signatures in body.

     – If a message uses raw encoding then the raw data will be passed as a single argument to the function.

     – Java/C, etc. can use a Thrift/protobuf document as the body

- `origin` is the name of the node sending the task.

- Dispatches to actor based on `task`, `meth` headers

     `meth` is unused by Python, but may be used in the future to specify class+method pairs.

- Chain gains a dedicated field.

     Reducing the chain into a recursive `callbacks` argument causes problems when the recursion limit is exceeded.

     This is fixed in the new message protocol by specifying a list of signatures, each task will then pop a task off the list when sending the next message:

```
execute_task(message)
chain = embed['chain']
if chain:
    sig = maybe_signature(chain.pop())
    sig.apply_async(chain=chain)
```

- `correlation_id` replaces `task_id` field.

- `root_id` and `parent_id` fields helps keep track of work-flows.

- `shadow` lets you specify a different name for logs, monitors can be used for concepts like tasks that calls a function specified as argument:

```python
from celery.utils.imports import qualname

class PickleTask(Task):

    def unpack_args(self, fun, args=()):
        return fun, args

    def apply_async(self, args, kwargs, **options):
        fun, real_args = self.unpack_args(*args)
        return super(PickleTask, self).apply_async(
            (fun, real_args, kwargs), shadow=qualname(fun), **options
        )

@app.task(base=PickleTask)
def call(fun, args, kwargs):
    return fun(*args, **kwargs)
```

Version 1

In version 1 of the protocol all fields are stored in the message body: meaning workers and intermediate consumers must deserialize the payload to read the fields.

Message body

- **task**

    string

    Name of the task. **required**

- **id**

    string

    Unique id of the task (UUID). **required**

- **args**

    list

    List of arguments. Will be an empty list if not provided.

- **kwargs**

    dictionary

    Dictionary of keyword arguments. Will be an empty dictionary if not provided.

- **retries**

    int

    Current number of times this task has been retried. Defaults to *0* if not specified.

- **eta**

    string **(ISO 8601)**

Estimated time of arrival. This is the date and time in ISO 8601 format. If not provided the message isn't scheduled, but will be executed asap.

- **expires**

    string **(ISO 8601)**

    New in version 2.0.2.

    Expiration date. This is the date and time in ISO 8601 format. If not provided the message will never expire. The message will be expired when the message is received and the expiration date has been exceeded.

- **taskset**

    string

    The group this task is part of (if any).

- **chord**

    Signature

    New in version 2.3.

    Signifies that this task is one of the header parts of a chord. The value of this key is the body of the cord that should be executed when all of the tasks in the header has returned.

- **utc**

    bool

    New in version 2.5.

    If true time uses the UTC timezone, if not the current local timezone should be used.

- **callbacks**

    <list>Signature

    New in version 3.0.

    A list of signatures to call if the task exited successfully.

- **errbacks**

    <list>Signature

    New in version 3.0.

    A list of signatures to call if an error occurs while executing the task.

- **timelimit**

    <tuple>(float, float)

    New in version 3.1.

    Task execution time limit settings. This is a tuple of hard and soft time limit value (*int/float* or `None` for no limit).

    Example value specifying a soft time limit of 3 seconds, and a hard time limit of 10 seconds:

    ```
    {'timelimit': (3.0, 10.0)}
    ```

Example message

This is an example invocation of a *celery.task.ping* task in json format:

```
{"id": "4cc7438e-afd4-4f8f-a2f3-f46567e7ca77",
 "task": "celery.task.PingTask",
 "args": [],
 "kwargs": {},
 "retries": 0,
 "eta": "2009-11-17T12:30:56.527191"}
```

Task Serialization

Several types of serialization formats are supported using the *content_type* message header.

The MIME-types supported by default are shown in the following table.

Scheme	MIME Type
json	application/json
yaml	application/x-yaml
pickle	application/x-python-serialize
msgpack	application/x-msgpack

Event Messages

Event messages are always JSON serialized and can contain arbitrary message body fields.

Since version 4.0. the body can consist of either a single mapping (one event), or a list of mappings (multiple events).

There are also standard fields that must always be present in an event message:

Standard body fields

- *string* type

    The type of event. This is a string containing the *category* and *action* separated by a dash delimiter (e.g., task-succeeded).

- *string* hostname

    The fully qualified hostname of where the event occurred at.

- *unsigned long long* clock

    The logical clock value for this event (Lamport time-stamp).

- *float* timestamp

    The UNIX time-stamp corresponding to the time of when the event occurred.

- *signed short* utcoffset

    This field describes the timezone of the originating host, and is specified as the number of hours ahead of/behind UTC (e.g., -2 or +1).

- *unsigned long long* pid

The process id of the process the event originated in.

Standard event types

For a list of standard event types and their fields see the *Event Reference*.

Example message

This is the message fields for a `task-succeeded` event:

```
properties = {
    'routing_key': 'task.succeeded',
    'exchange': 'celeryev',
    'content_type': 'application/json',
    'content_encoding': 'utf-8',
    'delivery_mode': 1,
}
headers = {
    'hostname': 'worker1@george.vandelay.com',
}
body = {
    'type': 'task-succeeded',
    'hostname': 'worker1@george.vandelay.com',
    'pid': 6335,
    'clock': 393912923921,
    'timestamp': 1401717709.101747,
    'utcoffset': -1,
    'uuid': '9011d855-fdd1-4f8f-adb3-a413b499eafb',
    'retval': '4',
    'runtime': 0.0003212,
)
```

3.12.5 "The Big Instance" Refactor

The *app* branch is a work-in-progress to remove the use of a global configuration in Celery.

Celery can now be instantiated and several instances of Celery may exist in the same process space. Also, large parts can be customized without resorting to monkey patching.

Examples

Creating a Celery instance:

```
>>> from celery import Celery
>>> app = Celery()
>>> app.config_from_object('celeryconfig')
>>> #app.config_from_envvar('CELERY_CONFIG_MODULE')
```

Creating tasks:

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

Creating custom Task subclasses:

```
Task = celery.create_task_cls()


class DebugTask(Task):

    def on_failure(self, *args, **kwargs):
        import pdb
        pdb.set_trace()


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

Starting a worker:

```
worker = celery.Worker(loglevel='INFO')
```

Getting access to the configuration:

```
celery.conf.task_always_eager = True
celery.conf['task_always_eager'] = True
```

Controlling workers:

```
>>> celery.control.inspect().active()
>>> celery.control.rate_limit(add.name, '100/m')
>>> celery.control.broadcast('shutdown')
>>> celery.control.discard_all()
```

Other interesting attributes:

```
# Establish broker connection.
>>> celery.broker_connection()

# AMQP Specific features.
>>> celery.amqp
>>> celery.amqp.Router
>>> celery.amqp.get_queues()
>>> celery.amqp.get_task_consumer()

# Loader
>>> celery.loader

# Default backend
>>> celery.backend
```

As you can probably see, this really opens up another dimension of customization abilities.

Deprecated

- `celery.task.ping` `celery.task.PingTask`

  Inferior to the ping remote control command. Will be removed in Celery 2.3.

Aliases (Pending deprecation)

- **`celery.task.base`**

    - `.Task` -> {`app.Task` / *`celery.app.task.Task`*}

- **`celery.task.sets`**

    - `.TaskSet` -> {`app.TaskSet`}

- **`celery.decorators`** / **`celery.task`**

    - `.task` -> {`app.task`}

- **`celery.execute`**

    - `.apply_async` -> {`task.apply_async`}

    - `.apply` -> {`task.apply`}

    - `.send_task` -> {`app.send_task`}

    - `.delay_task` -> *no alternative*

- **`celery.log`**

    - `.get_default_logger` -> {`app.log.get_default_logger`}

    - `.setup_logger` -> {`app.log.setup_logger`}

    - `.get_task_logger` -> {`app.log.get_task_logger`}

    - `.setup_task_logger` -> {`app.log.setup_task_logger`}

    - `.setup_logging_subsystem` -> {`app.log.setup_logging_subsystem`}

    - `.redirect_stdouts_to_logger` -> {`app.log.redirect_stdouts_to_logger`}

- **`celery.messaging`**

    - `.establish_connection` -> {`app.broker_connection`}

    - `.with_connection` -> {`app.with_connection`}

    - `.get_consumer_set` -> {`app.amqp.get_task_consumer`}

    - `.TaskPublisher` -> {`app.amqp.TaskPublisher`}

    - `.TaskConsumer` -> {`app.amqp.TaskConsumer`}

    - `.ConsumerSet` -> {`app.amqp.ConsumerSet`}

- celery.conf.* -> {`app.conf`}

    NOTE: All configuration keys are now named the same as in the configuration. So the key `task_always_eager` is accessed as:

    ```
    >>> app.conf.task_always_eager
    ```

    instead of:

    ```
    >>> from celery import conf
    >>> conf.always_eager
    ```

    - `.get_queues` -> {`app.amqp.get_queues`}

- **`celery.task.control`**

- .broadcast -> {app.control.broadcast}

- .rate_limit -> {app.control.rate_limit}

- .ping -> {app.control.ping}

- .revoke -> {app.control.revoke}

- .discard_all -> {app.control.discard_all}

- .inspect -> {app.control.inspect}

- **celery.utils.info**

    - .humanize_seconds -> celery.utils.time.humanize_seconds

    - .textindent -> celery.utils.textindent

    - .get_broker_info -> {app.amqp.get_broker_info}

    - .format_broker_info -> {app.amqp.format_broker_info}

    - .format_queues -> {app.amqp.format_queues}

Default App Usage

To be backward compatible, it must be possible to use all the classes/functions without passing an explicit app instance.

This is achieved by having all app-dependent objects use *default_app* if the app instance is missing.

```python
from celery.app import app_or_default

class SomeClass(object):

    def __init__(self, app=None):
        self.app = app_or_default(app)
```

The problem with this approach is that there's a chance that the app instance is lost along the way, and everything seems to be working normally. Testing app instance leaks is hard. The environment variable CELERY_TRACE_APP can be used, when this is enabled *celery.app.app_or_default()* will raise an exception whenever it has to go back to the default app instance.

App Dependency Tree

- **{app}**

    - celery.loaders.base.BaseLoader

    - celery.backends.base.BaseBackend

    - **{app.TaskSet}**

        * celery.task.sets.TaskSet (app.TaskSet)

    - **[app.TaskSetResult]**

        * celery.result.TaskSetResult (app.TaskSetResult)

- **{app.AsyncResult}**

    - celery.result.BaseAsyncResult / celery.result.AsyncResult

- **celery.bin.worker.WorkerCommand**

> > > – **celery.apps.worker.Worker**

> > > > ∗ **celery.worker.WorkerController**

> > > > > · **celery.worker.consumer.Consumer**

> > > > > > celery.worker.request.Request

> > > > > > celery.events.EventDispatcher

> > > > > > **celery.worker.control.ControlDispatch**

> > > > > > > celery.worker.control.registry.Panel

> > > > > > > celery.pidbox.BroadcastPublisher

> > > > > > celery.pidbox.BroadcastConsumer

> > > > > · celery.beat.EmbeddedService

> • **celery.bin.events.EvCommand**

> > > – **celery.events.snapshot.evcam**

> > > > ∗ celery.events.snapshot.Polaroid

> > > > ∗ celery.events.EventReceiver

> > > – **celery.events.cursesmon.evtop**

> > > > ∗ celery.events.EventReceiver

> > > > ∗ celery.events.cursesmon.CursesMonitor

> > > – **celery.events.dumper**

> > > > ∗ celery.events.EventReceiver

> • celery.bin.amqp.AMQPAdmin

> • **celery.bin.beat.BeatCommand**

> > > – **celery.apps.beat.Beat**

> > > > ∗ **celery.beat.Service**

> > > > > · celery.beat.Scheduler

3.12.6 Internal Module Reference

Release 4.4

Date Mar 02, 2020

celery.worker.components

Worker-level Bootsteps.

class celery.worker.components.**Timer**(*parent*, ***kwargs*)
  Timer bootstep.

  create(*w*)
    Create the step.

  name = 'celery.worker.components.Timer'

---

on_timer_error(*exc*)

on_timer_tick(*delay*)

class `celery.worker.components.`**Hub**(*w*, ***kwargs*)
Worker starts the event loop.

> **create**(*w*)
> Create the step.
>
> **include_if**(*w*)
> Return true if bootstep should be included.
>
> You can define this as an optional predicate that decides whether this step should be created.
>
> **name = 'celery.worker.components.Hub'**
>
> **requires = (step:celery.worker.components.Timer{()},)**
>
> **start**(*w*)
>
> **stop**(*w*)
>
> **terminate**(*w*)

class `celery.worker.components.`**Pool**(*w*, *autoscale=None*, ***kwargs*)
Bootstep managing the worker pool.

> Describes how to initialize the worker pool, and starts and stops the pool during worker start-up/shutdown.
>
> Adds attributes:
> - autoscale
> - pool
> - max_concurrency
> - min_concurrency
>
> **close**(*w*)
>
> **create**(*w*)
> Create the step.
>
> **info**(*w*)
>
> **name = 'celery.worker.components.Pool'**
>
> **register_with_event_loop**(*w*, *hub*)
>
> **requires = (step:celery.worker.components.Hub{(step:celery.worker.components.Timer{()}**
>
> **terminate**(*w*)

class `celery.worker.components.`**Beat**(*w*, *beat=False*, ***kwargs*)
Step used to embed a beat process.

> Enabled when the `beat` argument is set.
>
> **conditional = True**
>
> **create**(*w*)
> Create the step.
>
> **label = 'Beat'**
>
> **name = 'celery.worker.components.Beat'**

class `celery.worker.components.`**StateDB**(*w*, ***kwargs*)
Bootstep that sets up between-restart state database file.

create(*w*)
    Create the step.

name = 'celery.worker.components.StateDB'

class celery.worker.components.**Consumer**(*parent*, ***kwargs*)
    Bootstep starting the Consumer blueprint.

create(*w*)
    Create the step.

last = True

name = 'celery.worker.components.Consumer'

celery.worker.loops

The consumers highly-optimized inner loop.

celery.worker.loops.**asynloop**(*obj*, *connection*, *consumer*, *blueprint*, *hub*, *qos*, *heartbeat*, *clock*, *hbrate=2.0*)
    Non-blocking event loop.

celery.worker.loops.**synloop**(*obj*, *connection*, *consumer*, *blueprint*, *hub*, *qos*, *heartbeat*, *clock*, *hbrate=2.0*, ***kwargs*)
    Fallback blocking event loop for transports that doesn't support AIO.

celery.worker.heartbeat

Heartbeat service.

This is the internal thread responsible for sending heartbeat events at regular intervals (may not be an actual thread).

class celery.worker.heartbeat.**Heart**(*timer*, *eventer*, *interval=None*)
    Timer sending heartbeats at regular intervals.
        Parameters
            • **timer** (*kombu.asynchronous.timer.Timer*) – Timer to use.
            • **eventer** (*celery.events.EventDispatcher*) – Event dispatcher to use.
            • **interval** (*float*) – Time in seconds between sending heartbeats. Default is 2 seconds.
    start()

    stop()

celery.worker.control

Worker remote control command implementations.

class celery.worker.control.**Panel**(***kwargs*)
    Global registry of remote control commands.

data = {'active': <function active>, 'active_queues': <function active_queues>, 'add

meta = {'active': controller_info_t(alias='dump_active', type='inspect', visible=True

classmethod register(**args*, ***kwargs*)
    Register a virtual subclass of an ABC.

    Returns the subclass, to allow usage as a class decorator.

celery.worker.pidbox

Worker Pidbox (remote control).

class celery.worker.pidbox.**Pidbox**(*c*)
Worker mailbox.

consumer = None

on_message(*body*, *message*)

on_stop()

reset()

shutdown(*c*)

start(*c*)

stop(*c*)

class celery.worker.pidbox.**gPidbox**(*c*)
Worker pidbox (greenlet).

loop(*c*)

on_stop()

reset()

start(*c*)

celery.worker.autoscale

Pool Autoscaling.

This module implements the internal thread responsible for growing and shrinking the pool according to the current autoscale settings.

The autoscale thread is only enabled if the *celery worker --autoscale* option is used.

class celery.worker.autoscale.**Autoscaler**(*pool*, *max_concurrency*, *min_concurrency=0*, *worker=None*, *keepalive=30.0*, *mutex=None*)
Background thread to autoscale pool workers.

body()

force_scale_down(*n*)

force_scale_up(*n*)

info()

maybe_scale(*req=None*)

processes

qty

scale_down(*n*)

scale_up(*n*)

update(*max=None*, *min=None*)

class celery.worker.autoscale.**WorkerComponent**(*w*, ***kwargs*)
Bootstep that starts the autoscaler thread/timer in the worker.

```
conditional = True
```

create(*w*)
    Create the step.

info(*w*)
    Return *Autoscaler* info.

```
label = 'Autoscaler'
```

```
name = 'celery.worker.autoscale.WorkerComponent'
```

register_with_event_loop(*w*, *hub*)

```
requires = (step:celery.worker.components.Pool{(step:celery.worker.components.Hub{(step
```

celery.concurrency

Pool implementation abstract factory, and alias definitions.

celery.concurrency.**get_implementation**(*cls*)
    Return pool implementation by name.

celery.concurrency.solo

Single-threaded execution pool.

class celery.concurrency.solo.**TaskPool**(**args*, ***kwargs*)
    Solo task pool (blocking, inline, fast).

    ```
body_can_be_buffer = True
```

celery.concurrency.prefork

Prefork execution pool.

Pool implementation using multiprocessing.

class celery.concurrency.prefork.**TaskPool**(*limit=None*,   *putlocks=True*,   *fork-ing_enable=True*,   *callbacks_propagate=()*, *app=None*, ***options*)
    Multiprocessing Pool implementation.

    BlockingPool
        alias of billiard.pool.Pool

    Pool
        alias of celery.concurrency.asynpool.AsynPool

    did_start_ok()

    num_processes

    on_close()

    on_start()

    on_stop()
        Gracefully stop the pool.

    on_terminate()
        Force terminate the pool.

register_with_event_loop(*loop*)

restart()

uses_semaphore = True

write_stats = None

celery.concurrency.prefork.**process_initializer**(*app*, *hostname*)
    Pool child process initializer.

    Initialize the child pool process to ensure the correct app instance is used and things like logging works.

celery.concurrency.prefork.**process_destructor**(*pid*, *exitcode*)
    Pool child process destructor.

    Dispatch the *worker_process_shutdown* signal.

celery.concurrency.eventlet

Eventlet execution pool.

class celery.concurrency.eventlet.**TaskPool**(**args*, ***kwargs*)
    Eventlet Task Pool.

    class Timer(**args*, ***kwargs*)
        Eventlet Timer.

        cancel(*tref*)

        clear()

        queue
            Snapshot of underlying datastructure.

    grow(*n=1*)

    is_green = True

    on_apply(*target*, *args=None*, *kwargs=None*, *callback=None*, *accept_callback=None*, ***_*)

    on_start()

    on_stop()

    shrink(*n=1*)

    signal_safe = False

    task_join_will_block = False

celery.concurrency.gevent

Gevent execution pool.

class celery.concurrency.gevent.**TaskPool**(**args*, ***kwargs*)
    GEvent Pool.

    class Timer(**args*, ***kwargs*)

        clear()

        queue
            Snapshot of underlying datastructure.

grow(*n=1*)

is_green = True

num_processes

on_apply(*target*, *args=None*, *kwargs=None*, *callback=None*, *accept_callback=None*, *timeout=None*, *timeout_callback=None*, *apply_target=<function apply_target>*, ***_*)

on_start()

on_stop()

shrink(*n=1*)

signal_safe = False

task_join_will_block = False

celery.concurrency.thread

Thread execution pool.

class celery.concurrency.thread.**TaskPool**(**args*, ***kwargs*)
    Thread Task Pool.

    body_can_be_buffer = True

    on_apply(*target*, *args=None*, *kwargs=None*, *callback=None*, *accept_callback=None*, ***_*)

    on_stop()

    signal_safe = False

celery.concurrency.base

Base Execution Pool.

class celery.concurrency.base.**BasePool**(*limit=None*, *putlocks=True*, *forking_enable=True*, *callbacks_propagate=()*, *app=None*, ***options*)
    Task pool.

    CLOSE = 2

    RUN = 1

    TERMINATE = 3

    class Timer(*schedule=None*, *on_error=None*, *on_tick=None*, *on_start=None*, *max_interval=None*, ***kwargs*)
        Timer thread.

        Note: This is only used for transports not supporting AsyncIO.

        ---

        class Entry(*fun*, *args=None*, *kwargs=None*)
            Schedule Entry.

            args

            cancel()

            canceled

> > > **cancelled**
> > >
> > > **fun**
> > >
> > > **kwargs**
> > >
> > > **tref**
>
> > **Schedule**
> >   alias of `kombu.asynchronous.timer.Timer`
>
> > **call_after**(**args*, ***kwargs*)
>
> > **call_at**(**args*, ***kwargs*)
>
> > **call_repeatedly**(**args*, ***kwargs*)
>
> > **cancel**(*tref*)
>
> > **clear**()
>
> > **empty**()
>
> > **ensure_started**()
>
> > **enter**(*entry*, *eta*, *priority=None*)
>
> > **enter_after**(**args*, ***kwargs*)
>
> > **exit_after**(*secs*, *priority=10*)
>
> > **next**()
>
> > **on_tick = None**
>
> > **queue**
>
> > **run**()
> >   Method representing the thread's activity.
> >
> >   You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.
>
> > **running = False**
>
> > **stop**()

active

apply_async(*target*, *args=None*, *kwargs=None*, ***options*)
    Equivalent of the `apply()` built-in function.

    Callbacks should optimally return as soon as possible since otherwise the thread which handles the result will get blocked.

body_can_be_buffer = False

close()

did_start_ok()

flush()

info

is_green = False
    set to true if pool uses greenlets.

---

maintain_pool(**args*, ***kwargs*)

num_processes

on_apply(**args*, ***kwargs*)

on_close()

on_hard_timeout(*job*)

on_soft_timeout(*job*)

on_start()

on_stop()

on_terminate()

register_with_event_loop(*loop*)

restart()

signal_safe = True
> set to true if the pool can be shutdown from within a signal handler.

start()

stop()

task_join_will_block = True

terminate()

terminate_job(*pid*, *signal=None*)

uses_semaphore = False
> only used by multiprocessing pool

celery.concurrency.base.**apply_target**(*target*, *args=()*, *kwargs=None*, *callback=None*, *accept_callback=None*, *pid=None*, *getpid=<built-in function getpid>*, *propagate=()*, *monotonic=<built-in function monotonic>*, ***_*)
> Apply function within pool context.

celery.backends

Result Backends.

celery.backends.**get_backend_by_url**(*backend=None*, *loader=None*)
> Deprecated alias to *celery.app.backends.by_url()*.

celery.backends.**get_backend_cls**(*backend=None*, *loader=None*, ***kwargs*)
> Deprecated alias to *celery.app.backends.by_name()*.

celery.backends.base

Result backend base classes.

- *BaseBackend* defines the interface.

- **KeyValueStoreBackend is a common base class** using K/V semantics like _get and _put.

class `celery.backends.base.`**BaseBackend**(*app*, *serializer=None*, *max_cached_results=None*,
                                            accept=None, *expires=None*, *expires_type=None*,
                                            url=None, ***kwargs*)

    Base (synchronous) result backend.

class `celery.backends.base.`**KeyValueStoreBackend**(**args*, ***kwargs*)

    Result backend base class for key/value stores.

class `celery.backends.base.`**DisabledBackend**(*app*,   *serializer=None*,
                                            max_cached_results=None,   *accept=None*,
                                            expires=None, *expires_type=None*, *url=None*,
                                            **kwargs)

    Dummy result backend.

    as_uri(**args*, ***kwargs*)
        Return the backend as an URI, sanitizing the password or not.

    ensure_chords_allowed()

    get_many(**args*, ***kwargs*)

    get_result(**args*, ***kwargs*)

    get_state(**args*, ***kwargs*)

    get_status(**args*, ***kwargs*)

    get_task_meta_for(**args*, ***kwargs*)

    get_traceback(**args*, ***kwargs*)

    store_result(**args*, ***kwargs*)
        Update task state and result.

    wait_for(**args*, ***kwargs*)

celery.backends.asynchronous

Async I/O backend support utilities.

class `celery.backends.asynchronous.`**AsyncBackendMixin**

    Mixin for backends that enables the async API.

    add_pending_result(*result*, *weak=False*, *start_drainer=True*)

    add_pending_results(*results*, *weak=False*)

    is_async

    iter_native(*result*, *no_ack=True*, ***kwargs*)

    on_result_fulfilled(*result*)

    remove_pending_result(*result*)

    wait_for_pending(*result*, *callback=None*, *propagate=True*, ***kwargs*)

class `celery.backends.asynchronous.`**BaseResultConsumer**(*backend*,   *app*,   *accept*,
                                            pending_results,   *pend-
                                            ing_messages*)

    Manager responsible for consuming result messages.

    cancel_for(*task_id*)

    consume_from(*task_id*)

drain_events(*timeout=None*)

drain_events_until(*p*, *timeout=None*, *on_interval=None*)

on_after_fork()

on_out_of_band_result(*message*)

on_state_change(*meta*, *message*)

on_wait_for_pending(*result*, *timeout=None*, ***kwargs*)

start(*initial_task_id*, ***kwargs*)

stop()

class celery.backends.asynchronous.**Drainer**(*result_consumer*)
    Result draining service.

drain_events_until(*p*, *timeout=None*, *interval=1*, *on_interval=None*, *wait=None*)

start()

stop()

wait_for(*p*, *wait*, *timeout=None*)

celery.backends.asynchronous.**register_drainer**(*name*)
    Decorator used to register a new result drainer type.

celery.backends.azureblockblob

The Azure Storage Block Blob backend for Celery.

class celery.backends.azureblockblob.**AzureBlockBlobBackend**(*url=None*, *container_name=None*, *retry_initial_backoff_sec=None*, *retry_increment_base=None*, *retry_max_attempts=None*, **args*, ***kwargs*)

Azure Storage Block Blob backend for Celery.

delete(*key*)
    Delete the value at a given key.
        Parameters key – The key of the value to delete.

get(*key*)
    Read the value stored at the given key.
        Parameters key – The key for which to read the value.

mget(*keys*)
    Read all the values for the provided keys.
        Parameters keys – The list of keys to read.

set(*key*, *value*)
    Store a value for a given key.
        Parameters
                • **key** – The key at which to store the value.
                • **value** – The value to store.

`celery.backends.rpc`

The `RPC` result backend for AMQP brokers.

RPC-style result backend, using reply-to and one queue per client.

exception `celery.backends.rpc.`**`BacklogLimitExceeded`**
    Too much state history to fast-forward.

class `celery.backends.rpc.`**`RPCBackend`**(*app,   connection=None,   exchange=None,   ex-
                                          change_type=None, persistent=None, serializer=None,
                                          auto_delete=True, **kwargs*)
    Base class for the RPC result backend.

    exception `BacklogLimitExceeded`
        Exception raised when there are too many messages for a task id.

    class `Consumer`(*channel,   queues=None,   no_ack=None,   auto_declare=None,   call-
                    backs=None,   on_decode_error=None,   on_message=None,   accept=None,
                    prefetch_count=None, tag_prefix=None*)
        Consumer that requires manual declaration of queues.

        `auto_declare = False`

    class `Exchange`(*name='', type='', channel=None, **kwargs*)
        An Exchange declaration.
            Parameters
                • **`name`** (`str`) – See *name*.
                • **`type`** (`str`) – See *type*.
                • **`channel`** (`kombu.Connection, ChannelT`) – See *channel*.
                • **`durable`** (`bool`) – See *durable*.
                • **`auto_delete`** (`bool`) – See *auto_delete*.
                • **`delivery_mode`** (`enum`) – See *delivery_mode*.
                • **`arguments`** (`Dict`) – See *arguments*.
                • **`no_declare`** (`bool`) – See *no_declare*

        `name`
            Name of the exchange. Default is no name (the default exchange).
                Type str

        `type`
            *This description of AMQP exchange types was shamelessly stolen from the blog post 'AMQP in 10
            minutes: Part 4'_ by Rajith Attapattu. Reading this article is recommended if you're new to amqp.*

            "AMQP defines four default exchange types (routing algorithms) that covers most of the common
            messaging use cases. An AMQP broker can also define additional exchange types, so see your
            broker manual for more information about available exchange types.
                    • *direct* (*default*)

                        Direct match between the routing key in the message, and the rout-
                        ing criteria used when a queue is bound to this exchange.
                    • *topic*

                        Wildcard match between the routing key and the routing pattern
                        specified in the exchange/queue binding. The routing key is treated
                        as zero or more words delimited by *"."* and supports special wild-
                        card characters. *"*"* matches a single word and *"#"* matches zero
                        or more words.
                    • *fanout*

> > > Queues are bound to this exchange with no arguments. Hence any
> > > message sent to this exchange will be forwarded to all queues bound
> > > to this exchange.

> > - *headers*

> > > Queues are bound to this exchange with a table of arguments containing headers and values (optional). A special argument named "x-match" determines the matching algorithm, where *"all"* implies an *AND* (all pairs must match) and *"any"* implies *OR* (at least one pair must match).

> > > [`arguments`](#) is used to specify the arguments.

> > **Type** str

channel

> The channel the exchange is bound to (if bound).
>
> > **Type** ChannelT

durable

> Durable exchanges remain active when a server restarts. Non-durable exchanges (transient exchanges) are purged when a server restarts. Default is `True`.
>
> > **Type** [bool](#)

auto_delete

> If set, the exchange is deleted when all queues have finished using it. Default is `False`.
>
> > **Type** [bool](#)

delivery_mode

> The default delivery mode used for messages. The value is an integer, or alias string.
>
> - 1 or *"transient"*
>
>   > The message is transient. Which means it is stored in memory only, and is lost
>   > if the server dies or restarts.
>
> - **2 or "persistent"** (*default*) The message is persistent. Which means the message is stored
>   both in-memory, and on disk, and therefore preserved if the server dies or restarts.
>
> The default value is 2 (persistent).
>
> > **Type** enum

arguments

> Additional arguments to specify when the exchange is declared.
>
> > **Type** Dict

no_declare

> Never declare this exchange ([`declare()`](#) does nothing).
>
> > **Type** [bool](#)

Message (*body*, *delivery_mode=None*, *properties=None*, ***kwargs*)

> Create message instance to be sent with [`publish()`](#).
>
> > **Parameters**

> > - **body** (*Any*) – Message body.

> > - **delivery_mode** ([*bool*](#)) – Set custom delivery mode. Defaults to
> >   [*delivery_mode*](#).

> > - **priority** ([*int*](#)) – Message priority, 0 to broker configured max priority,
> >   where higher is better.

> > - **content_type** ([*str*](#)) – The messages content_type. If content_type
> >   is set, no serialization occurs as it is assumed this is either a binary ob-

> > > > ject, or you've done your own serialization. Leave blank if using built-in
> > > > serialization as our library properly sets content_type.
> > >
> > > - **content_encoding** (*str*) – The character set in which this object is
> > >   encoded. Use "binary" if sending in raw binary objects. Leave blank if
> > >   using built-in serialization as our library properly sets content_encoding.
> > >
> > > - **properties** (*Dict*) – Message properties.
> > >
> > > - **headers** (*Dict*) – Message headers.

PERSISTENT_DELIVERY_MODE = 2

TRANSIENT_DELIVERY_MODE = 1

attrs = (('name', None), ('type', None), ('arguments', None), ('durable', <class '

auto_delete = False

bind_to(*exchange="*, *routing_key="*, *arguments=None*, *nowait=False*, *channel=None*,
**kwargs)
   Bind the exchange to another exchange.
   > **Parameters nowait** (*bool*) – If set the server will not respond, and the call will not
   > block waiting for a response. Default is `False`.

binding(*routing_key="*, *arguments=None*, *unbind_arguments=None*)

can_cache_declaration
   bool(x) -> bool

   Returns True when the argument x is true, False otherwise. The builtins True and False are the
   only two instances of the class bool. The class bool is a subclass of the class int, and cannot be
   subclassed.

declare(*nowait=False*, *passive=None*, *channel=None*)
   Declare the exchange.

   Creates the exchange on the broker, unless passive is set in which case it will only assert that the
   exchange exists.
   Argument:
   > nowait (bool): **If set the server will not respond, and a** response will not be waited for.
   > Default is `False`.

delete(*if_unused=False*, *nowait=False*)
   Delete the exchange declaration on server.
   > **Parameters**
   >
   > - **if_unused** (*bool*) – Delete only if the exchange has no bindings. De-
   >   fault is `False`.
   >
   > - **nowait** (*bool*) – If set the server will not respond, and a response will
   >   not be waited for. Default is `False`.

delivery_mode = None

durable = True

name = ''

no_declare = False

passive = False

publish(*message*, *routing_key=None*, *mandatory=False*, *immediate=False*, *exchange=None*)
   Publish message.

Parameters

- **message** (*Union[kombu.Message, str, bytes]*) – Message to publish.

- **routing_key** (*str*) – Message routing key.

- **mandatory** (*bool*) – Currently not supported.

- **immediate** (*bool*) – Currently not supported.

type = 'direct'

unbind_from(*source=''*, *routing_key=''*, *nowait=False*, *arguments=None*, *channel=None*)
Delete previously created exchange binding from the server.

class Producer(*channel*, *exchange=None*, *routing_key=None*, *serializer=None*, *auto_declare=None*, *compression=None*, *on_return=None*)
Message Producer.

Parameters

- **channel** (*kombu.Connection, ChannelT*) – Connection or channel.
- **exchange** (*kombu.entity.Exchange, str*) – Optional default exchange.
- **routing_key** (*str*) – Optional default routing key.
- **serializer** (*str*) – Default serializer. Default is *"json"*.
- **compression** (*str*) – Default compression method. Default is no compression.
- **auto_declare** (*bool*) – Automatically declare the default exchange at instantiation. Default is True.
- **on_return** (*Callable*) – Callback to call for undeliverable messages, when the *mandatory* or *immediate* arguments to *publish()* is used. This callback needs the following signature: *(exception, exchange, routing_key, message)*. Note that the producer needs to drain events to use this feature.

auto_declare = True

channel

close()

compression = None

connection

declare()
Declare the exchange.

Note: This happens automatically at instantiation when the *auto_declare* flag is enabled.

---

exchange = None

maybe_declare(*entity*, *retry=False*, ***retry_policy*)
Declare exchange if not already declared during this session.

on_return = None

publish(*body*, *routing_key=None*, *delivery_mode=None*, *mandatory=False*, *immediate=False*, *priority=0*, *content_type=None*, *content_encoding=None*, *serializer=None*, *headers=None*, *compression=None*, *exchange=None*, *retry=False*, *retry_policy=None*, *declare=None*, *expiration=None*, ***properties*)
Publish message to the specified exchange.

Parameters

- **body** (*Any*) – Message body.
- **routing_key** (*str*) – Message routing key.
- **delivery_mode** (*enum*) – See `delivery_mode`.
- **mandatory** (*bool*) – Currently not supported.
- **immediate** (*bool*) – Currently not supported.
- **priority** (*int*) – Message priority. A number between 0 and 9.
- **content_type** (*str*) – Content type. Default is auto-detect.
- **content_encoding** (*str*) – Content encoding. Default is auto-detect.
- **serializer** (*str*) – Serializer to use. Default is auto-detect.
- **compression** (*str*) – Compression method to use. Default is none.
- **headers** (*Dict*) – Mapping of arbitrary headers to pass along with the message body.
- **exchange** (*kombu.entity.Exchange, str*) – Override the exchange. Note that this exchange must have been declared.
- **declare** (*Sequence[EntityT]*) – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using `maybe_declare()`.
- **retry** (*bool*) – Retry publishing, or declaring entities if the connection is lost.
- **retry_policy** (*Dict*) – Retry configuration, this is the keywords supported by `ensure()`.
- **expiration** (*float*) – A TTL in seconds can be specified per message. Default is no expiration.
- ****properties** (*Any*) – Additional message properties, see AMQP spec.

release()

revive(*channel*)
    Revive the producer after connection loss.

routing_key = ''

serializer = None

class Queue(*name=''*, *exchange=None*, *routing_key=''*, *channel=None*, *bindings=None*, *on_declared=None*, ***kwargs*)
    Queue that never caches declaration.

can_cache_declaration = False

class ResultConsumer(**args*, ***kwargs*)

class Consumer(*channel*, *queues=None*, *no_ack=None*, *auto_declare=None*, *callbacks=None*, *on_decode_error=None*, *on_message=None*, *accept=None*, *prefetch_count=None*, *tag_prefix=None*)
    Message consumer.
        Parameters

- **channel** (*kombu.Connection, ChannelT*) – see `channel`.

- **queues** (*Sequence[kombu.Queue]*) – see *queues*.

- **no_ack** (*bool*) – see *no_ack*.

- **auto_declare** (*bool*) – see *auto_declare*

- **callbacks** (*Sequence[Callable]*) – see *callbacks*.

- **on_message** (*Callable*) – See *on_message*

- **on_decode_error** (*Callable*) – see *on_decode_error*.

- **prefetch_count** (*int*) – see *prefetch_count*.

exception ContentDisallowed
> Consumer does not allow this content-type.

accept = None

add_queue(*queue*)
> Add a queue to the list of queues to consume from.

> **Note:** This will not start consuming from the queue, for that you will have to call *consume()* after.

---

auto_declare = True

callbacks = None

cancel()
> End all active queue consumers.

> **Note:** This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

---

cancel_by_queue(*queue*)
> Cancel consumer by queue name.

channel = None

close()
> End all active queue consumers.

> **Note:** This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

---

connection

consume(*no_ack=None*)
> Start consuming messages.

> Can be called multiple times, but note that while it will consume from new queues added since the last call, it will not cancel consuming from removed queues (use *cancel_by_queue()*).
>> **Parameters no_ack** (*bool*) – See *no_ack*.

consuming_from(*queue*)
> Return True if currently consuming from queue'.

declare()
> Declare queues, exchanges and bindings.

> ---
> **Note:** This is done automatically at instantiation when *auto_declare* is set.
> ---

flow(*active*)
> Enable/disable flow from peer.

> This is a simple flow-control mechanism that a peer can use to avoid overflowing its queues or otherwise finding itself receiving more messages than it can process.

> The peer that receives a request to stop sending content will finish sending the current content (if any), and then wait until flow is reactivated.

no_ack = None

on_decode_error = None

on_message = None

prefetch_count = None

purge()
> Purge messages from all queues.

> ---
> **Warning:** This will *delete all ready messages*, there is no undo operation.
> ---

qos(*prefetch_size=0*, *prefetch_count=0*, *apply_global=False*)
> Specify quality of service.

> The client can request that messages should be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

> The prefetch window is Ignored if the *no_ack* option is set.
> > **Parameters**
> > - **prefetch_size** (*int*) – Specify the prefetch window in octets. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls within other prefetch limits). May be set to zero, meaning "no specific limit", although other prefetch limits may still apply.
> > - **prefetch_count** (*int*) – Specify the prefetch window in terms of whole messages.
> > - **apply_global** (*bool*) – Apply new settings globally on all channels.

queues

receive(*body*, *message*)
> Method called when a message is received.

> This dispatches to the registered *callbacks*.
> > **Parameters**
> > - **body** (*Any*) – The decoded message body.
> > - **message** (*Message*) – The message instance.

> > **Raises** `NotImplementedError` – If no consumer callbacks have been registered.

> **recover**(*requeue=False*)
> > Redeliver unacknowledged messages.

> > Asks the broker to redeliver all unacknowledged messages on the specified channel.
> > > **Parameters requeue** (`bool`) – By default the messages will be redelivered to the original recipient. With *requeue* set to true, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

> **register_callback**(*callback*)
> > Register a new callback to be called when a message is received.

> > ---

> > **Note:** The signature of the callback needs to accept two arguments: *(body, message)*, which is the decoded message body and the `Message` instance.

> > ---

> **revive**(*channel*)
> > Revive consumer after connection loss.

cancel_for(*task_id*)

consume_from(*task_id*)

drain_events(*timeout=None*)

on_after_fork()

start(*initial_task_id*, *no_ack=True*, ***kwargs*)

stop()

as_uri(*include_password=True*)
> Return the backend as an URI, sanitizing the password or not.

binding

delete_group(*group_id*)

destination_for(*task_id*, *request*)
> Get the destination for result by task id.
> > **Returns** tuple of (`reply_to`, `correlation_id`).
> > **Return type** Tuple[str, str]

ensure_chords_allowed()

get_task_meta(*task_id*, *backlog_limit=1000*)

oid

on_out_of_band_result(*task_id*, *message*)

on_reply_declare(*task_id*)

on_result_fulfilled(*result*)

on_task_call(*producer*, *task_id*)

persistent = False

poll(*task_id*, *backlog_limit=1000*)

reload_group_result(*task_id*)
>   Reload group result, even if it has been previously fetched.

reload_task_result(*task_id*)
>   Reload task result, even if it has been previously fetched.

restore_group(*group_id*, *cache=True*)
>   Get the result for a group.

retry_policy = {'interval_max': 1, 'interval_start': 0, 'interval_step': 1, 'max_re

revive(*channel*)

save_group(*group_id*, *result*)
>   Store the result of an executed group.

store_result(*task_id*, *result*, *state*, *traceback=None*, *request=None*, ***kwargs*)
>   Send task return value and state.

supports_autoexpire = True

supports_native_join = True

celery.backends.database

SQLAlchemy result store backend.

class celery.backends.database.**DatabaseBackend**(*dburi=None*,   *engine_options=None*,
>   *url=None*, ***kwargs*)
>   The database result backend.

>   **ResultSession**(*session_manager=<celery.backends.database.session.SessionManager object>*)

>   **cleanup**()
>   >   Delete expired meta-data.

>   **extended_result**

>   **subpolling_interval = 0.5**

>   **task_cls**
>   >   alias of *celery.backends.database.models.Task*

>   **taskset_cls**
>   >   alias of *celery.backends.database.models.TaskSet*

celery.backends.amqp

The old AMQP result backend, deprecated and replaced by the RPC backend.

exception celery.backends.amqp.**BacklogLimitExceeded**
>   Too much state history to fast-forward.

class celery.backends.amqp.**AMQPBackend**(*app*,   *connection=None*,   *exchange=None*,   *ex-
>   change_type=None*,   *persistent=None*,   *serial-
>   izer=None*, *auto_delete=True*, ***kwargs*)
>   The AMQP result backend.

>   Deprecated: Please use the RPC backend or a persistent backend.

>   **exception BacklogLimitExceeded**
>   >   Too much state history to fast-forward.

class Consumer(*channel*, *queues=None*, *no_ack=None*, *auto_declare=None*, *call-backs=None*, *on_decode_error=None*, *on_message=None*, *accept=None*, *prefetch_count=None*, *tag_prefix=None*)

    Message consumer.

        Parameters
- **channel** (*kombu.Connection, ChannelT*) – see *channel*.
- **queues** (*Sequence[kombu.Queue]*) – see *queues*.
- **no_ack** (*bool*) – see *no_ack*.
- **auto_declare** (*bool*) – see *auto_declare*
- **callbacks** (*Sequence[Callable]*) – see *callbacks*.
- **on_message** (*Callable*) – See *on_message*
- **on_decode_error** (*Callable*) – see *on_decode_error*.
- **prefetch_count** (*int*) – see *prefetch_count*.

exception ContentDisallowed

    Consumer does not allow this content-type.

accept = None

add_queue(*queue*)

    Add a queue to the list of queues to consume from.

    Note: This will not start consuming from the queue, for that you will have to call *consume()* after.

---

auto_declare = True

callbacks = None

cancel()

    End all active queue consumers.

    Note: This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

---

cancel_by_queue(*queue*)

    Cancel consumer by queue name.

channel = None

close()

    End all active queue consumers.

    Note: This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

---

connection

consume(*no_ack=None*)

    Start consuming messages.

    Can be called multiple times, but note that while it will consume from new queues added since the last call, it will not cancel consuming from removed queues (use *cancel_by_queue()*).

        Parameters no_ack (*bool*) – See *no_ack*.

consuming_from(*queue*)

    Return True if currently consuming from queue'.

---

declare()
> Declare queues, exchanges and bindings.

> **Note:** This is done automatically at instantiation when *auto_declare* is set.

flow (*active*)
> Enable/disable flow from peer.

> This is a simple flow-control mechanism that a peer can use to avoid overflowing its queues or otherwise finding itself receiving more messages than it can process.

> The peer that receives a request to stop sending content will finish sending the current content (if any), and then wait until flow is reactivated.

no_ack = None

on_decode_error = None

on_message = None

prefetch_count = None

purge()
> Purge messages from all queues.

> **Warning:** This will *delete all ready messages*, there is no undo operation.

qos (*prefetch_size=0*, *prefetch_count=0*, *apply_global=False*)
> Specify quality of service.

> The client can request that messages should be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

> The prefetch window is Ignored if the *no_ack* option is set.
> > **Parameters**
> > - **prefetch_size** (*int*) – Specify the prefetch window in octets. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls within other prefetch limits). May be set to zero, meaning "no specific limit", although other prefetch limits may still apply.
> > - **prefetch_count** (*int*) – Specify the prefetch window in terms of whole messages.
> > - **apply_global** (*bool*) – Apply new settings globally on all channels.

queues

receive (*body*, *message*)
> Method called when a message is received.

> This dispatches to the registered *callbacks*.
> > **Parameters**
> > - **body** (*Any*) – The decoded message body.
> > - **message** (*Message*) – The message instance.
> > **Raises** NotImplementedError – If no consumer callbacks have been registered.

---

recover (*requeue=False*)
> Redeliver unacknowledged messages.

> Asks the broker to redeliver all unacknowledged messages on the specified channel.
>> **Parameters requeue** (*bool*) – By default the messages will be redelivered to the original recipient. With *requeue* set to true, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

register_callback (*callback*)
> Register a new callback to be called when a message is received.

> **Note:** The signature of the callback needs to accept two arguments: *(body, message)*, which is the decoded message body and the `Message` instance.

---

revive (*channel*)
> Revive consumer after connection loss.

class Exchange (*name=''*, *type=''*, *channel=None*, ***kwargs*)
> An Exchange declaration.
>> **Parameters**
>>> - **name** (*str*) – See *name*.
>>> - **type** (*str*) – See *type*.
>>> - **channel** (*kombu.Connection, ChannelT*) – See *channel*.
>>> - **durable** (*bool*) – See *durable*.
>>> - **auto_delete** (*bool*) – See *auto_delete*.
>>> - **delivery_mode** (*enum*) – See *delivery_mode*.
>>> - **arguments** (*Dict*) – See *arguments*.
>>> - **no_declare** (*bool*) – See *no_declare*

name
> Name of the exchange. Default is no name (the default exchange).
>> **Type** str

type
> *This description of AMQP exchange types was shamelessly stolen from the blog post 'AMQP in 10 minutes: Part 4'_ by Rajith Attapattu. Reading this article is recommended if you're new to amqp.*

> "AMQP defines four default exchange types (routing algorithms) that covers most of the common messaging use cases. An AMQP broker can also define additional exchange types, so see your broker manual for more information about available exchange types.
>> - *direct* (*default*)
>>
>>> Direct match between the routing key in the message, and the routing criteria used when a queue is bound to this exchange.
>> - *topic*
>>
>>> Wildcard match between the routing key and the routing pattern specified in the exchange/queue binding. The routing key is treated as zero or more words delimited by "." and supports special wildcard characters. "*" matches a single word and "#" matches zero or more words.
>> - *fanout*
>>
>>> Queues are bound to this exchange with no arguments. Hence any message sent to this exchange will be forwarded to all queues bound to this exchange.
>> - *headers*

> Queues are bound to this exchange with a table of arguments containing headers and values (optional). A special argument named "x-match" determines the matching algorithm, where *"all"* implies an *AND* (all pairs must match) and *"any"* implies *OR* (at least one pair must match).
>
> `arguments` is used to specify the arguments.
>
> > **Type** str

channel

> The channel the exchange is bound to (if bound).
> > **Type** ChannelT

durable

> Durable exchanges remain active when a server restarts. Non-durable exchanges (transient exchanges) are purged when a server restarts. Default is `True`.
> > **Type** bool

auto_delete

> If set, the exchange is deleted when all queues have finished using it. Default is `False`.
> > **Type** bool

delivery_mode

> The default delivery mode used for messages. The value is an integer, or alias string.
> - 1 or *"transient"*
>   > The message is transient. Which means it is stored in memory only, and is lost if the server dies or restarts.
> - **2 or "persistent"** (*default*) The message is persistent. Which means the message is stored both in-memory, and on disk, and therefore preserved if the server dies or restarts.
>
> The default value is 2 (persistent).
> > **Type** enum

arguments

> Additional arguments to specify when the exchange is declared.
> > **Type** Dict

no_declare

> Never declare this exchange (`declare()` does nothing).
> > **Type** bool

Message (*body*, *delivery_mode=None*, *properties=None*, ***kwargs*)

> Create message instance to be sent with `publish()`.
> > **Parameters**
> >
> > - **body** (*Any*) – Message body.
> >
> > - **delivery_mode** (*bool*) – Set custom delivery mode. Defaults to `delivery_mode`.
> >
> > - **priority** (*int*) – Message priority, 0 to broker configured max priority, where higher is better.
> >
> > - **content_type** (*str*) – The messages content_type. If content_type is set, no serialization occurs as it is assumed this is either a binary object, or you've done your own serialization. Leave blank if using built-in serialization as our library properly sets content_type.
> >
> > - **content_encoding** (*str*) – The character set in which this object is encoded. Use "binary" if sending in raw binary objects. Leave blank if using built-in serialization as our library properly sets content_encoding.

- **properties** (`Dict`) – Message properties.

- **headers** (`Dict`) – Message headers.

PERSISTENT_DELIVERY_MODE = 2

TRANSIENT_DELIVERY_MODE = 1

attrs = (('name', None), ('type', None), ('arguments', None), ('durable', <class '

auto_delete = False

bind_to(*exchange=''*, *routing_key=''*, *arguments=None*, *nowait=False*, *channel=None*, ***kwargs*)
> Bind the exchange to another exchange.
>> **Parameters nowait** (`bool`) – If set the server will not respond, and the call will not block waiting for a response. Default is `False`.

binding(*routing_key=''*, *arguments=None*, *unbind_arguments=None*)

can_cache_declaration
> bool(x) -> bool

> Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

declare(*nowait=False*, *passive=None*, *channel=None*)
> Declare the exchange.

> Creates the exchange on the broker, unless passive is set in which case it will only assert that the exchange exists.
> **Argument:**
>> nowait (bool): **If set the server will not respond, and a** response will not be waited for. Default is `False`.

delete(*if_unused=False*, *nowait=False*)
> Delete the exchange declaration on server.
>> **Parameters**

- **if_unused** (`bool`) – Delete only if the exchange has no bindings. Default is `False`.

- **nowait** (`bool`) – If set the server will not respond, and a response will not be waited for. Default is `False`.

delivery_mode = None

durable = True

name = ''

no_declare = False

passive = False

publish(*message*, *routing_key=None*, *mandatory=False*, *immediate=False*, *exchange=None*)
> Publish message.
>> **Parameters**

- **message** (`Union[kombu.Message,` `str,` `bytes]`) – Message to publish.

- **routing_key** (`str`) – Message routing key.

- **mandatory** (`bool`) – Currently not supported.

> • **immediate** (*bool*) – Currently not supported.

type = 'direct'

unbind_from(*source=''*, *routing_key=''*, *nowait=False*, *arguments=None*, *channel=None*)
> Delete previously created exchange binding from the server.

class Producer(*channel*, *exchange=None*, *routing_key=None*, *serializer=None*, *auto_declare=None*, *compression=None*, *on_return=None*)
> Message Producer.
>
> > **Parameters**
> > > • **channel** (*kombu.Connection, ChannelT*) – Connection or channel.
> > > • **exchange** (*kombu.entity.Exchange, str*) – Optional default exchange.
> > > • **routing_key** (*str*) – Optional default routing key.
> > > • **serializer** (*str*) – Default serializer. Default is *"json"*.
> > > • **compression** (*str*) – Default compression method. Default is no compression.
> > > • **auto_declare** (*bool*) – Automatically declare the default exchange at instantiation. Default is `True`.
> > > • **on_return** (*Callable*) – Callback to call for undeliverable messages, when the *mandatory* or *immediate* arguments to *publish()* is used. This callback needs the following signature: *(exception, exchange, routing_key, message)*. Note that the producer needs to drain events to use this feature.

auto_declare = True

channel

close()

compression = None

connection

declare()
> Declare the exchange.

> **Note:** This happens automatically at instantiation when the *auto_declare* flag is enabled.

---

exchange = None

maybe_declare(*entity*, *retry=False*, ***retry_policy*)
> Declare exchange if not already declared during this session.

on_return = None

publish(*body*, *routing_key=None*, *delivery_mode=None*, *mandatory=False*, *immediate=False*, *priority=0*, *content_type=None*, *content_encoding=None*, *serializer=None*, *headers=None*, *compression=None*, *exchange=None*, *retry=False*, *retry_policy=None*, *declare=None*, *expiration=None*, ***properties*)
> Publish message to the specified exchange.
>
> > **Parameters**
> > > • **body** (*Any*) – Message body.
> > >
> > > • **routing_key** (*str*) – Message routing key.
> > >
> > > • **delivery_mode** (*enum*) – See `delivery_mode`.
> > >
> > > • **mandatory** (*bool*) – Currently not supported.

- **immediate** (`bool`) – Currently not supported.

- **priority** (`int`) – Message priority. A number between 0 and 9.

- **content_type** (`str`) – Content type. Default is auto-detect.

- **content_encoding** (`str`) – Content encoding. Default is auto-detect.

- **serializer** (`str`) – Serializer to use. Default is auto-detect.

- **compression** (`str`) – Compression method to use. Default is none.

- **headers** (`Dict`) – Mapping of arbitrary headers to pass along with the message body.

- **exchange** (`kombu.entity.Exchange, str`) – Override the exchange. Note that this exchange must have been declared.

- **declare** (`Sequence[EntityT]`) – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using `maybe_declare()`.

- **retry** (`bool`) – Retry publishing, or declaring entities if the connection is lost.

- **retry_policy** (`Dict`) – Retry configuration, this is the keywords supported by `ensure()`.

- **expiration** (`float`) – A TTL in seconds can be specified per message. Default is no expiration.

- ****properties** (`Any`) – Additional message properties, see AMQP spec.

release()

revive(*channel*)

    Revive the producer after connection loss.

routing_key = ''

serializer = None

Queue

    alias of `NoCacheQueue`

as_uri(*include_password=True*)

    Return the backend as an URI, sanitizing the password or not.

consume(*task_id*, *timeout=None*, *no_ack=True*, *on_interval=None*)

delete_group(*group_id*)

destination_for(*task_id*, *request*)

drain_events(*connection*, *consumer*, *timeout=None*, *on_interval=None*, *now=<built-in function monotonic>*, *wait=None*)

get_many(*task_ids*, *timeout=None*, *no_ack=True*, *on_message=None*, *on_interval=None*, *now=<built-in function monotonic>*, *getfields=operator.itemgetter('status', 'task_id')*, *READY_STATES=frozenset({'FAILURE', 'SUCCESS', 'REVOKED'})*, *PROPAGATE_STATES=frozenset({'FAILURE', 'REVOKED'})*, ***kwargs*)

get_task_meta(*task_id*, *backlog_limit=1000*)

on_reply_declare(*task_id*)

> **persistent = True**
>
> **poll**(*task_id*, *backlog_limit=1000*)
>
> **reload_group_result**(*task_id*)
>   Reload group result, even if it has been previously fetched.
>
> **reload_task_result**(*task_id*)
>   Reload task result, even if it has been previously fetched.
>
> **restore_group**(*group_id*, *cache=True*)
>   Get the result for a group.
>
> **retry_policy = {'interval_max': 1, 'interval_start': 0, 'interval_step': 1, 'max_ret**
>
> **revive**(*channel*)
>
> **rkey**(*task_id*)
>
> **save_group**(*group_id*, *result*)
>   Store the result of an executed group.
>
> **store_result**(*task_id*, *result*, *state*, *traceback=None*, *request=None*, ***kwargs*)
>   Send task return value and state.
>
> **supports_autoexpire = True**
>
> **supports_native_join = True**
>
> **wait_for**(*task_id*, *timeout=None*, *cache=True*, *no_ack=True*, *on_interval=None*, *READY_STATES=frozenset({'FAILURE', 'SUCCESS', 'REVOKED'})*, *PROPA-GATE_STATES=frozenset({'FAILURE', 'REVOKED'})*, ***kwargs*)
>   Wait for task and return its result.
>
>   If the task raises an exception, this exception will be re-raised by *wait_for()*.
>   **Raises** *celery.exceptions.TimeoutError* – If *timeout* is not None, and the operation takes longer than *timeout* seconds.

celery.backends.cache

Memcached and in-memory cache result backend.

class celery.backends.cache.**CacheBackend**(*app*, *expires=None*, *backend=None*, *options=None*, *url=None*, ***kwargs*)
>   Cache result backend.
>
> **as_uri**(**args*, ***kwargs*)
>   Return the backend as an URI.
>
>   This properly handles the case of multiple servers.
>
> **client**
>
> **delete**(*key*)
>
> **expire**(*key*, *value*)
>
> **get**(*key*)
>
> **implements_incr = True**
>
> **incr**(*key*)
>
> **mget**(*keys*)
>
> **servers = None**

set(*key*, *value*)

supports_autoexpire = True

supports_native_join = True

celery.backends.consul

Consul result store backend.

- *ConsulBackend* **implements KeyValueStoreBackend to store results** in the key-value store of Consul.

class celery.backends.consul.**ConsulBackend**(**args*, ***kwargs*)

Consul.io K/V store backend for Celery.

client = None

consistency = 'consistent'

consul = None

delete(*key*)

get(*key*)

mget(*keys*)

path = None

set(*key*, *value*)

Set a key in Consul.

Before creating the key it will create a session inside Consul where it creates a session with a TTL

The key created afterwards will reference to the session's ID.

If the session expires it will remove the key so that results can auto expire from the K/V store

supports_autoexpire = True

celery.backends.couchdb

CouchDB result store backend.

class celery.backends.couchdb.**CouchBackend**(*url=None*, **args*, ***kwargs*)

CouchDB backend.

> **Raises** *celery.exceptions.ImproperlyConfigured* – if module pycouchdb is not
> available.

connection

container = 'default'

delete(*key*)

get(*key*)

host = 'localhost'

mget(*keys*)

password = None

port = 5984

scheme = 'http'

set (*key*, *value*)

username = None

celery.backends.mongodb

MongoDB result store backend.

class celery.backends.mongodb.**MongoBackend**(*app=None*, ***kwargs*)

MongoDB result backend.

> **Raises** *celery.exceptions.ImproperlyConfigured* – if module pymongo is not available.

as_uri(*include_password=False*)

Return the backend as an URI.

> **Parameters include_password** (*bool*) – Password censored if disabled.

cleanup()

Delete expired meta-data.

collection

Get the meta-data task collection.

database

Get database from MongoDB connection.

performs authentication if necessary.

database_name = 'celery'

decode(*data*)

encode(*data*)

expires_delta

group_collection

Get the meta-data task collection.

groupmeta_collection = 'celery_groupmeta'

host = 'localhost'

max_pool_size = 10

mongo_host = None

options = None

password = None

port = 27017

supports_autoexpire = False

taskmeta_collection = 'celery_taskmeta'

user = None

celery.backends.elasticsearch

Elasticsearch result store backend.

class celery.backends.elasticsearch.**ElasticsearchBackend**(*url=None*, **args*, ***kwargs*)

    Elasticsearch Backend.

        Raises *celery.exceptions.ImproperlyConfigured* – if module elasticsearch is not available.

    delete(*key*)

    doc_type = 'backend'

    es_max_retries = 3

    es_retry_on_timeout = False

    es_timeout = 10

    get(*key*)

    host = 'localhost'

    index = 'celery'

    mget(*keys*)

    password = None

    port = 9200

    scheme = 'http'

    server

    set(*key*, *value*)

    username = None

celery.backends.redis

Redis result store backend.

class celery.backends.redis.**RedisBackend**(*host=None*, *port=None*, *db=None*, *password=None*, *max_connections=None*, *url=None*, *connection_pool=None*, ***kwargs*)

    Redis task result store.

    It makes use of the following commands: GET, MGET, DEL, INCRBY, EXPIRE, SET, SETEX

    ConnectionPool

    class ResultConsumer(**args*, ***kwargs*)

        cancel_for(*task_id*)

        consume_from(*task_id*)

        drain_events(*timeout=None*)

        on_after_fork()

        on_state_change(*meta*, *message*)

        on_wait_for_pending(*result*, ***kwargs*)

        reconnect_on_error()

        start(*initial_task_id*, ***kwargs*)

> **stop**()

add_to_chord(*group_id*, *result*)

apply_chord(*header_result*, *body*, ***kwargs*)

client

db

delete(*key*)

ensure(*fun*, *args*, ***policy*)

expire(*key*, *value*)

forget(*task_id*)

get(*key*)

host

incr(*key*)

max_connections = None
> Maximum number of connections in the pool.

mget(*keys*)

on_chord_part_return(*request*, *state*, *result*, *propagate=None*, ***kwargs*)

on_connection_error(*max_retries*, *exc*, *intervals*, *retries*)

on_task_call(*producer*, *task_id*)

password

port

redis = None
> redis client module.

set(*key*, *value*, ***retry_policy*)

supports_autoexpire = True

supports_native_join = True

class celery.backends.redis.**SentinelBackend**(**args*, ***kwargs*)
> Redis sentinel task result store.

sentinel = None

celery.backends.riak

Riak result store backend.

class celery.backends.riak.**RiakBackend**(*host=None*, *port=None*, *bucket_name=None*, *proto-col=None*, *url=None*, **args*, ***kwargs*)
> Riak result backend.
>> **Raises** *celery.exceptions.ImproperlyConfigured* – if module riak is not available.

bucket

bucket_name = 'celery'
> default Riak bucket name (*default*)

client

delete(*key*)

get(*key*)

host = 'localhost'
> default Riak server hostname (*localhost*)

mget(*keys*)

port = 8087
> default Riak server port (8087)

protocol = 'pbc'
> default protocol used to connect to Riak, might be *http* or *pbc*

set(*key*, *value*)

celery.backends.cassandra

Apache Cassandra result store backend using the DataStax driver.

class celery.backends.cassandra.**CassandraBackend**(*servers=None,   keyspace=None,*
table=None,   entry_ttl=None,
port=9042, **kwargs)

> Cassandra backend utilizing DataStax driver.
> > **Raises** *celery.exceptions.ImproperlyConfigured* – if module cassandra-driver is
> > not available, or if the *cassandra_servers* setting is not set.
>
> **as_uri**(*include_password=True*)
> > Return the backend as an URI, sanitizing the password or not.
>
> **process_cleanup**()
> > Cleanup actions to do at the end of a task worker process.
>
> **servers = None**
> > hostname.
> > > **Type** List of Cassandra servers with format
>
> **supports_autoexpire = True**

celery.backends.couchbase

Couchbase result store backend.

class celery.backends.couchbase.**CouchbaseBackend**(*url=None, *args, **kwargs*)
> Couchbase backend.
> > **Raises** *celery.exceptions.ImproperlyConfigured* – if module couchbase is not
> > available.
>
> **bucket = 'default'**
>
> **connection**
>
> **delete**(*key*)
>
> **get**(*key*)
>
> **host = 'localhost'**
>
> **key_t**
> > alias of builtins.str
>
> **mget**(*keys*)

password = None

port = 8091

quiet = False

set(*key*, *value*)

supports_autoexpire = True

timeout = 2.5

username = None

celery.backends.arangodb

ArangoDb result store backend.

class celery.backends.arangodb.**ArangoDbBackend**(*url=None*, **args*, ***kwargs*)
   ArangoDb backend.

   Sample url "arangodb://username:password@host:port/database/collection" *arangodb_backend_settings* is where the settings are present (in the app.conf) Settings should contain the host, port, username, password, database name, collection name else the default will be chosen. Default database name and collection name is celery.
   > **Raises** celery.exceptions.ImproperlyConfigured: – if module pyArango is not available.

   collection = 'celery'

   connection
      Connect to the arangodb server.

   database = 'celery'

   db
      Database Object to the given database.

   delete(*key*)

   get(*key*)

   host = '127.0.0.1'

   http_protocol = 'http'

   key_t
      alias of `builtins.str`

   mget(*keys*)

   password = None

   port = '8529'

   set(*key*, *value*)
      Insert a doc with value into task attribute and _key as key.

   username = None

celery.backends.dynamodb

AWS DynamoDB result store backend.

class celery.backends.dynamodb.**DynamoDBBackend**(*url=None*, *table_name=None*, **args*, ***kwargs*)

    AWS DynamoDB result backend.

        Raises *celery.exceptions.ImproperlyConfigured* – if module boto3 is not available.

    aws_region = None

        AWS region (*default*)

    client

    delete(*key*)

    endpoint_url = None

        The endpoint URL that is passed to boto3 (local DynamoDB) (*default*)

    get(*key*)

    mget(*keys*)

    read_capacity_units = 1

        Read Provisioned Throughput (*default*)

    set(*key*, *value*)

    supports_autoexpire = True

    table_name = 'celery'

        default DynamoDB table name (*default*)

    time_to_live_seconds = None

        Item time-to-live in seconds (*default*)

    write_capacity_units = 1

        Write Provisioned Throughput (*default*)

celery.backends.filesystem

File-system result store backend.

class celery.backends.filesystem.**FilesystemBackend**(*url=None*, *open=<built-in function open>*, *unlink=<built-in function unlink>*, *sep='/'*, *encoding='UTF-8'*, **args*, ***kwargs*)

    File-system result backend.

        Parameters

            • **url** (*str*) – URL to the directory we should use

            • **open** (*Callable*) – open function to use when opening files

            • **unlink** (*Callable*) – unlink function to use when deleting files

            • **sep** (*str*) – directory separator (to join the directory with the key)

            • **encoding** (*str*) – encoding used on the file-system

    delete(*key*)

    get(*key*)

    mget(*keys*)

    set(*key*, *value*)

`celery.backends.cosmosdbsql`

The CosmosDB/SQL backend for Celery (experimental).

class `celery.backends.cosmosdbsql.`**`CosmosDBSQLBackend`**(*url=None*,
database_name=None,
collection_name=None,
consistency_level=None,
max_retry_attempts=None,
max_retry_wait_time=None,
*args, ***kwargs*)

> CosmosDB/SQL backend for Celery.
>
> **delete**(*key*)
> > Delete the value at a given key.
> > > Parameters **key** – The key of the value to delete.
>
> **get**(*key*)
> > Read the value stored at the given key.
> > > Parameters **key** – The key for which to read the value.
>
> **mget**(*keys*)
> > Read all the values for the provided keys.
> > > Parameters **keys** – The list of keys to read.
>
> **set**(*key*, *value*)
> > Store a value for a given key.
> > > **Parameters**
> > > - **key** – The key at which to store the value.
> > > - **value** – The value to store.

`celery.backends.s3`

s3 result store backend.

class `celery.backends.s3.`**`S3Backend`**(***kwargs*)

> An S3 task result store.
> > Raises *`celery.exceptions.ImproperlyConfigured`* – if module boto3 is not available, if the `aws_access_key_id` or setting:*aws_secret_access_key* are not set, or it the `bucket` is not set.
>
> **delete**(*key*)
>
> **get**(*key*)
>
> **set**(*key*, *value*)

`celery.app.trace`

Trace task execution.

This module defines how the task execution is traced: errors are recorded, handlers are applied and so on.

class `celery.app.trace.`**`TraceInfo`**(*state*, *retval=None*)
> Information about task execution.
>
> **handle_error_state**(*task*, *req*, *eager=False*, *call_errbacks=True*)
>
> **handle_failure**(*task*, *req*, *store_errors=True*, *call_errbacks=True*)
> > Handle exception.

handle_ignore(*task*, *req*, ***kwargs*)

handle_reject(*task*, *req*, ***kwargs*)

handle_retry(*task*, *req*, *store_errors=True*, ***kwargs*)
> Handle retry exception.

retval

state

celery.app.trace.**build_tracer**(*name*, *task*, *loader=None*, *hostname=None*, *store_errors=True*, *Info=<class 'celery.app.trace.TraceInfo'>*, *eager=False*, *propagate=False*, *app=None*, *monotonic=<built-in function monotonic>*, *trace_ok_t=<class 'celery.app.trace.trace_ok_t'>*, *IGNORE_STATES=frozenset({'RETRY', 'REJECTED', 'IGNORED'})*)
> Return a function that traces task execution.
>
> Catches all exceptions and updates result backend with the state and result.
>
> If the call was successful, it saves the result to the task result backend, and sets the task status to *"SUCCESS"*.
>
> If the call raises *Retry*, it extracts the original exception, uses that as the result and sets the task state to *"RETRY"*.
>
> If the call results in an exception, it saves the exception as the task result, and sets the task state to *"FAILURE"*.
>
> Return a function that takes the following arguments:
>> **param uuid** The id of the task.
>> **param args** List of positional args to pass on to the function.
>> **param kwargs** Keyword arguments mapping to pass on to the function.
>> **keyword request** Request dict.

celery.app.trace.**trace_task**(*task*, *uuid*, *args*, *kwargs*, *request=None*, ***opts*)
> Trace task execution.

celery.app.trace.**setup_worker_optimizations**(*app*, *hostname=None*)
> Setup worker related optimizations.

celery.app.trace.**reset_worker_optimizations**()
> Reset previously configured optimizations.

celery.app.annotations

Task Annotations.

Annotations is a nice term for monkey-patching task classes in the configuration.

This prepares and performs the annotations in the *task_annotations* setting.

class celery.app.annotations.**MapAnnotation**
> Annotation map: task_name => attributes.

> **annotate**(*task*)

> **annotate_any**()

celery.app.annotations.**prepare**(*annotations*)
> Expand the *task_annotations* setting.

celery.app.annotations.**resolve_all**(*anno*, *task*)
> Resolve all pending annotations.

`celery.app.routes`

Task Routing.

Contains utilities for working with task routers, (`task_routes`).

class `celery.app.routes.`**`MapRoute`**(*map*)

    Creates a router out of a `dict`.

class `celery.app.routes.`**`Router`**(*routes=None*,   *queues=None*,   *create_missing=False*,   *app=None*)

    Route tasks based on the `task_routes` setting.

    `expand_destination`(*route*)

    `lookup_route`(*name*, *args=None*, *kwargs=None*, *options=None*, *task_type=None*)

    `query_router`(*router*, *task*, *args*, *kwargs*, *options*, *task_type*)

    `route`(*options*, *name*, *args=()*, *kwargs=None*, *task_type=None*)

`celery.app.routes.`**`prepare`**(*routes*)

    Expand the `task_routes` setting.

`celery.security.certificate`

X.509 certificates.

class `celery.security.certificate.`**`Certificate`**(*cert*)

    X.509 certificate.

    `get_id`()

        Serial number/issuer pair uniquely identifies a certificate.

    `get_issuer`()

        Return issuer (CA) as a string.

    `get_pubkey`()

        Get public key from certificate.

    `get_serial_number`()

        Return the serial number in the certificate.

    `has_expired`()

        Check if the certificate has expired.

    `verify`(*data*, *signature*, *digest*)

        Verify signature for string containing data.

class `celery.security.certificate.`**`CertStore`**

    Base class for certificate stores.

    `add_cert`(*cert*)

    `itercerts`()

        Return certificate iterator.

class `celery.security.certificate.`**`FSCertStore`**(*path*)

    File system certificate store.

`celery.security.key`

Private keys for the security serializer.

class `celery.security.key.`**`PrivateKey`**(*key*, *password=None*)
> Represents a private key.

> **`sign`**(*data*, *digest*)
> > Sign string containing data.

`celery.security.serialization`

Secure serializer.

class `celery.security.serialization.`**`SecureSerializer`**(*key=None*, *cert=None*, *cert_store=None*, *digest='sha256'*, *serializer='json'*)

> Signed serializer.

> **`deserialize`**(*data*)
> > Deserialize data structure from string.

> **`serialize`**(*data*)
> > Serialize data structure into string.

`celery.security.serialization.`**`register_auth`**(*key=None*, *cert=None*, *store=None*, *digest='sha256'*, *serializer='json'*)
> Register security serializer.

`celery.security.utils`

Utilities used by the message signing serializer.

`celery.security.utils.`**`get_digest_algorithm`**(*digest='sha256'*)
> Convert string to hash object of cryptography library.

`celery.security.utils.`**`reraise_errors`**(*msg='{0!r}'*, *errors=None*)
> Context reraising crypto errors as `SecurityError`.

`celery.events.snapshot`

Periodically store events in a database.

Consuming the events as a stream isn't always suitable so this module implements a system to take snapshots of the state of a cluster at regular intervals. There's a full implementation of this writing the snapshots to a database in `djcelery.snapshots` in the *django-celery* distribution.

class `celery.events.snapshot.`**`Polaroid`**(*state*, *freq=1.0*, *maxrate=None*, *cleanup_freq=3600.0*, *timer=None*, *app=None*)
> Record event snapshots.

> **`cancel`**()

> **`capture`**()

> **`cleanup`**()

> **`cleanup_signal`** `= <Signal:  cleanup_signal providing_args=set()>`

```
clear_after = False
```

install()

on_cleanup()

on_shutter(*state*)

shutter()

```
shutter_signal = <Signal:  shutter_signal providing_args={'state'}>
```

```
timer = None
```

celery.events.snapshot.**evcam**(*camera*, *freq=1.0*, *maxrate=None*, *loglevel=0*, *logfile=None*, *pid-file=None*, *timer=None*, *app=None*)

Start snapshot recorder.

celery.events.cursesmon

Graphical monitor of Celery events using curses.

class celery.events.cursesmon.**CursesMonitor**(*state*, *app*, *keymap=None*)

A curses based Celery task monitor.

alert(*callback*, *title=None*)

alert_remote_control_reply(*reply*)

```
background = 7
```

display_height

display_task_row(*lineno*, *task*)

display_width

draw()

find_position()

```
foreground = 0
```

format_row(*uuid*, *task*, *worker*, *timestamp*, *state*)

```
greet = 'celery events 4.4.1 (cliffs)'
```

handle_keypress()

```
help = 'j:down k:up i:info t:traceback r:result c:revoke ^c:  quit'
```

```
help_title = 'Keys:  '
```

```
info_str = 'Info:  '
```

init_screen()

```
keyalias = {258:  'J', 259:  'K', 343:  'I'}
```

```
keymap = {}
```

limit

move_selection(*direction=1*)

move_selection_down()

move_selection_up()

nap()

online_str = 'Workers online: '

readline(*x*, *y*)

resetscreen()

revoke_selection()

safe_add_str(*y*, *x*, *string*, **args*, ***kwargs*)

screen_delay = 10

screen_height

screen_width

selected_position = 0

selected_str = 'Selected: '

selected_task = None

selection_info()

selection_rate_limit()

selection_result()

selection_traceback()

tasks

win = None

workers

celery.events.cursesmon.**evtop**(*app=None*)
    Start curses monitor.

celery.events.dumper

Utility to dump events to screen.

This is a simple program that dumps events to the console as they happen. Think of it like a *tcpdump* for Celery events.

class celery.events.dumper.**Dumper**(*out=<_io.TextIOWrapper   name='<stdout>'   mode='w'*
                                        encoding='UTF-8'>)
    Monitor events.

    format_task_event(*hostname*, *timestamp*, *type*, *task*, *event*)

    on_event(*ev*)

    say(*msg*)

celery.events.dumper.**evdump**(*app=None*, *out=<_io.TextIOWrapper name='<stdout>' mode='w'*
                                 encoding='UTF-8'>)
    Start event dump.

`celery.backends.database.models`

Database models used by the SQLAlchemy result store backend.

class `celery.backends.database.models.`**`Task`**(*task_id*)
  Task result/status.

  classmethod configure(*schema=None*, *name=None*)

  date_done

  id

  result

  status

  task_id

  to_dict()

  traceback

class `celery.backends.database.models.`**`TaskExtended`**(*task_id*)
  For the extend result.

  args

  date_done

  id

  kwargs

  name

  queue

  result

  retries

  status

  task_id

  to_dict()

  traceback

  worker

class `celery.backends.database.models.`**`TaskSet`**(*taskset_id*, *result*)
  TaskSet result.

  classmethod configure(*schema=None*, *name=None*)

  date_done

  id

  result

  taskset_id

  to_dict()

`celery.backends.database.session`

SQLAlchemy session.

class `celery.backends.database.session.`**`SessionManager`**
    Manage SQLAlchemy sessions.

> **`create_session`**(*dburi*, *short_lived_sessions=False*, ***kwargs*)

> **`get_engine`**(*dburi*, ***kwargs*)

> **`prepare_models`**(*engine*)

> **`session_factory`**(*dburi*, ***kwargs*)

`celery.utils`

Utility functions.

Don't import from here directly anymore, as these are only here for backwards compatibility.

class `celery.utils.`**`cached_property`**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)
    Cached property descriptor.

> Caches the return value of the get method on first call.

> ### Examples

```python
@cached_property
def connection(self):
    return Connection()

@connection.setter  # Prepares stored value
def connection(self, value):
    if value is None:
        raise TypeError('Connection must be a connection')
    return value

@connection.deleter
def connection(self, value):
    # Additional action to do at del(self.attr)
    if value is not None:
        print('Connection {0!r} deleted'.format(value)
```

> **`deleter`**(*fdel*)

> **`setter`**(*fset*)

`celery.utils.`**`chunks`**(*it*, *n*)
    Split an iterator into chunks with *n* elements each.

> **Warning:** `it` must be an actual iterator, if you pass this a concrete sequence will get you repeating elements.
>
> So `chunks(iter(range(1000)), 10)` is fine, but `chunks(range(1000), 10)` is not.

Example

n == 2 >>> x = chunks(iter([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]), 2) >>> list(x) [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9], [10]]

n == 3 >>> x = chunks(iter([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]), 3) >>> list(x) [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10]]

celery.utils.**gen_task_name**(*app*, *name*, *module_name*)
> Generate task name from name/module pair.

celery.utils.**gen_task_name**(*app*, *name*, *module_name*)
> Generate task name from name/module pair.

celery.utils.**gen_unique_id**(*_uuid=<function uuid4>*)
> Generate unique id in UUID4 format.

> **See also:**

> For now this is provided by uuid.uuid4().

celery.utils.**get_cls_by_name**(*name*, *aliases=None*, *imp=None*, *package=None*, *sep='.'*, *default=None*, ***kwargs*)
> Get symbol by qualified name.

> The name should be the full dot-separated path to the class:

```
modulename.ClassName
```

> Example:

```
celery.concurrency.processes.TaskPool
                          ^- class name
```

> or using ':' to separate module and symbol:

```
celery.concurrency.processes:TaskPool
```

> If *aliases* is provided, a dict containing short name/long name mappings, the name is looked up in the aliases first.

Examples

```
>>> symbol_by_name('celery.concurrency.processes.TaskPool')
<class 'celery.concurrency.processes.TaskPool'>
```

```
>>> symbol_by_name('default', {
...     'default': 'celery.concurrency.processes.TaskPool'})
<class 'celery.concurrency.processes.TaskPool'>
```

> # Does not try to look up non-string names. >>> from celery.concurrency.processes import TaskPool >>> symbol_by_name(TaskPool) is TaskPool True

celery.utils.**get_full_cls_name**(*obj*)
> Return object name.

celery.utils.**import_from_cwd**(*module*, *imp=None*, *package=None*)
> Import module, temporarily including modules in the current directory.

> Modules located in the current directory has precedence over modules located in *sys.path*.

`celery.utils.`**`instantiate`**(*name*, **args*, ***kwargs*)
  Instantiate class by name.

  See also:

  `symbol_by_name()`.

`celery.utils.`**`memoize`**(*maxsize=None*,   *keyfun=None*,   *Cache=<class 'kombu.utils.functional.LRUCache'>*)
  Decorator to cache function return value.

`celery.utils.`**`nodename`**(*name*, *hostname*)
  Create node name from name/hostname pair.

`celery.utils.`**`nodesplit`**(*name*)
  Split node name into tuple of name/hostname.

`celery.utils.`**`noop`**(**args*, ***kwargs*)
  No operation.

  Takes any arguments/keyword arguments and does nothing.

`celery.utils.`**`uuid`**(*_uuid=<function uuid4>*)
  Generate unique id in UUID4 format.

  See also:

  For now this is provided by `uuid.uuid4()`.

`celery.utils.`**`worker_direct`**(*hostname*)
  Return the `kombu.Queue` being a direct route to a worker.
      Parameters hostname (`str, Queue`) – The fully qualified node name of a worker (e.g., w1@example.com). If passed a `kombu.Queue` instance it will simply return that instead.

`celery.utils.abstract`

Abstract classes.

class `celery.utils.abstract.`**`CallableTask`**
  Task interface.

  `apply`(**args*, ***kwargs*)

  `apply_async`(**args*, ***kwargs*)

  `delay`(**args*, ***kwargs*)

class `celery.utils.abstract.`**`CallableSignature`**
  Celery Signature interface.

  `app`

  `args`

  `chord_size`

  `clone`(*args=None*, *kwargs=None*)

  `freeze`(*id=None*, *group_id=None*, *chord=None*, *root_id=None*)

  `id`

  `immutable`

  `kwargs`

link(*callback*)

link_error(*errback*)

name

options

set(*immutable=None*, ***options*)

subtask_type

task

type

celery.utils.collections

Custom maps, sets, sequences, and other data structures.

class celery.utils.collections.**AttributeDictMixin**
    Mixin for Mapping interface that adds attribute access.

    I.e., *d.key -> d[key])*.

class celery.utils.collections.**AttributeDict**
    Dict subclass with attribute access.

class celery.utils.collections.**BufferMap**(*maxsize*, *iterable=None*, *bufmaxsize=1000*)
    Map of buffers.

    Buffer
        alias of *Messagebuffer*

    exception Empty
        Exception raised by Queue.get(block=0)/get_nowait().

    bufmaxsize = None

    extend(*key*, *it*)

    maxsize = None

    put(*key*, *item*)

    take(*key*, **default*)

    total = 0

class celery.utils.collections.**ChainMap**(**maps*, ***kwargs*)
    Key lookup on a sequence of maps.

    add_defaults(*d*)

    bind_to(*callback*)

    changes = None

    clear() → None. Remove all items from D.

    copy()

    defaults = None

    classmethod fromkeys(*iterable*, **args*)
        Create a ChainMap with a single dict created from the iterable.

get $(k[, d]) \rightarrow$ D[k] if k in D, else d. d defaults to None.

items ()

iteritems ()

iterkeys ()

itervalues ()

key_t = None

keys ()

maps = None

pop $(k[, d]) \rightarrow$ v, remove specified key and return the corresponding value.
    If key is not found, d is returned if given, otherwise KeyError is raised.

setdefault $(k[, d]) \rightarrow$ D.get(k,d), also set D[k]=d if k not in D

update $([E], **F) \rightarrow$ None. Update D from mapping/iterable E and F.
    If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method,
    does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values ()

class celery.utils.collections.**ConfigurationView**(*changes*, *defaults=None*, *keys=None*,
                                                            prefix=None)
    A view over an applications configuration dictionaries.

    Custom (but older) version of `collections.ChainMap`.

    If the key does not exist in `changes`, the `defaults` dictionaries are consulted.
        Parameters
            • **changes** (`Mapping`) – Map of configuration changes.
            • **defaults** (`List[Mapping]`) – List of dictionaries containing the default config-
              uration.
    clear ()
        Remove all changes, but keep defaults.

    first (**keys*)

    get $(k[, d]) \rightarrow$ D[k] if k in D, else d. d defaults to None.

    swap_with (*other*)

class celery.utils.collections.**DictAttribute**(*obj*)
    Dict interface to attributes.

    obj[k] -> obj.k obj[k] = val -> obj.k = val

    get (*key*, *default=None*)

    items ()

    iteritems ()

    iterkeys ()

    itervalues ()

    keys ()

    obj = None

    setdefault (*key*, *default=None*)

> **values**()

class celery.utils.collections.**Evictable**

> Mixin for classes supporting the evict method.
>
> > **exception Empty**
> >
> > > Exception raised by Queue.get(block=0)/get_nowait().
> >
> > **evict**()
> >
> > > Force evict until maxsize is enforced.

class celery.utils.collections.**LimitedSet**(*maxlen=0*, *expires=0*, *data=None*, *minlen=0*)

> Kind-of Set (or priority queue) with limitations.
>
> Good for when you need to test for membership (*a in set*), but the set should not grow unbounded.
>
> maxlen is enforced at all times, so if the limit is reached we'll also remove non-expired items.
>
> You can also configure minlen: this is the minimal residual size of the set.
>
> All arguments are optional, and no limits are enabled by default.
>
> > **Parameters**
> >
> > - **maxlen** (*int*) – Optional max number of items. Adding more items than maxlen will result in immediate removal of items sorted by oldest insertion time.
> > - **expires** (*float*) – TTL for all items. Expired items are purged as keys are inserted.
> > - **minlen** (*int*) – Minimal residual size of this set. .. versionadded:: 4.0
> >
> >   Value must be less than maxlen if both are configured.
> >
> >   Older expired items will be deleted, only after the set exceeds minlen number of items.
> > - **data** (*Sequence*) – Initial data to initialize set with. Can be an iterable of (key, value) pairs, a dict ({key: insertion_time}), or another instance of *LimitedSet*.

Example

```
>>> s = LimitedSet(maxlen=50000, expires=3600, minlen=4000)
>>> for i in range(60000):
...     s.add(i)
...     s.add(str(i))
...
>>> 57000 in s  # last 50k inserted values are kept
True
>>> '10' in s  # '10' did expire and was purged from set.
False
>>> len(s)  # maxlen is reached
50000
>>> s.purge(now=monotonic() + 7200)  # clock + 2 hours
>>> len(s)  # now only minlen items are cached
4000
>>>> 57000 in s  # even this item is gone now
False
```

> **add**(*item*, *now=None*)
>
> > Add a new item, or reset the expiry time of an existing item.
>
> **as_dict**()
>
> > Whole set as serializable dictionary.

Example

```
>>> s = LimitedSet(maxlen=200)
>>> r = LimitedSet(maxlen=200)
>>> for i in range(500):
...     s.add(i)
...
>>> r.update(s.as_dict())
>>> r == s
True
```

clear()
> Clear all data, start from scratch again.

discard(*item*)

max_heap_percent_overload = 15

pop(*default=None*)
> Remove and return the oldest item, or `None` when empty.

pop_value(*item*)

purge(*now=None*)
> Check oldest items and remove them if needed.
>> **Parameters now** (*float*) – Time of purging – by default right now. This can be useful for unit testing.

update(*other*)
> Update this set from other LimitedSet, dict or iterable.

class celery.utils.collections.**Messagebuffer**(*maxsize*, *iterable=None*, *deque=<class 'collections.deque'>*)
> A buffer of pending messages.

> **exception Empty**
>> Exception raised by Queue.get(block=0)/get_nowait().

> **extend**(*it*)

> **put**(*item*)

> **take**(**default*)

class celery.utils.collections.**OrderedDict**
> Dict where insertion order matters.

celery.utils.collections.**force_mapping**(*m*)
> Wrap object into supporting the mapping interface if necessary.

celery.utils.collections.**lpmerge**(*L*, *R*)
> In place left precedent dictionary merge.

> Keeps values from *L*, if the value in *R* is `None`.

celery.utils.nodenames

Worker name utilities.

celery.utils.nodenames.**worker_direct**(*hostname*)
> Return the `kombu.Queue` being a direct route to a worker.

Parameters **hostname** (*[str,](Queue)* *[Queue](str)*) – The fully qualified node name of a worker (e.g., w1@example.com). If passed a `kombu.Queue` instance it will simply return that instead.

celery.utils.nodenames.**gethostname**() → string
    Return the current host name.

celery.utils.nodenames.**nodename**(*name*, *hostname*)
    Create node name from name/hostname pair.

celery.utils.nodenames.**anon_nodename**(*hostname=None*, *prefix='gen'*)
    Return the nodename for this process (not a worker).

    This is used for e.g. the origin task message field.

celery.utils.nodenames.**nodesplit**(*name*)
    Split node name into tuple of name/hostname.

celery.utils.nodenames.**default_nodename**(*hostname*)
    Return the default nodename for this process.

celery.utils.nodenames.**node_format**(*s*, *name*, ***extra*)
    Format worker node name (name@host.com).

celery.utils.nodenames.**host_format**(*s*, *host=None*, *name=None*, ***extra*)
    Format host %x abbreviations.

celery.utils.deprecated

Deprecation utilities.

celery.utils.deprecated.**Callable**(*deprecation=None*, *removal=None*, *alternative=None*, *description=None*)
    Decorator for deprecated functions.

    A deprecation warning will be emitted when the function is called.
        Parameters
            • **deprecation** (*[str](str)*) – Version that marks first deprecation, if this argument isn't set a `PendingDeprecationWarning` will be emitted instead.
            • **removal** (*[str](str)*) – Future version when this feature will be removed.
            • **alternative** (*[str](str)*) – Instructions for an alternative solution (if any).
            • **description** (*[str](str)*) – Description of what's being deprecated.

celery.utils.deprecated.**Property**(*deprecation=None*, *removal=None*, *alternative=None*, *description=None*)
    Decorator for deprecated properties.

celery.utils.deprecated.**warn**(*description=None*, *deprecation=None*, *removal=None*, *alternative=None*, *stacklevel=2*)
    Warn of (pending) deprecation.

celery.utils.functional

Functional-style utilties.

class celery.utils.functional.**LRUCache**(*limit=None*)
    LRU Cache implementation using a doubly linked list to track access.
        Parameters **limit** (*[int](int)*) – The maximum number of keys to keep in the cache. When a new key is inserted and the limit has been exceeded, the *Least Recently Used* key will be discarded from the cache.
        incr(*key*, *delta=1*)

items()

iteritems()

iterkeys()

itervalues()

keys()

popitem() → (k, v), remove and return some (key, value) pair
>   as a 2-tuple; but raise KeyError if D is empty.

update($[E]$, ***F*) → None. Update D from mapping/iterable E and F.
>   If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method,
>   does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values()

celery.utils.functional.**is_list**(*l*, *scalars=(<class 'collections.abc.Mapping'>, <class 'str'>),*
>   *iters=(<class 'collections.abc.Iterable'>,)*)
>   Return true if the object is iterable.

Note: Returns false if object is a mapping or string.

---

celery.utils.functional.**maybe_list**(*l*, *scalars=(<class 'collections.abc.Mapping'>, <class*
>   *'str'>)*)
>   Return list of one element if `l` is a scalar.

celery.utils.functional.**memoize**(*maxsize=None*, *keyfun=None*, *Cache=<class*
>   *'kombu.utils.functional.LRUCache'>*)
>   Decorator to cache function return value.

class celery.utils.functional.**mlazy**(*fun*, **args*, ***kwargs*)
>   Memoized lazy evaluation.

>   The function is only evaluated once, every subsequent access will return the same value.

>   **evaluate**()

>   **evaluated = False**
>   >   Set to `True` after the object has been evaluated.

celery.utils.functional.**noop**(**args*, ***kwargs*)
>   No operation.

>   Takes any arguments/keyword arguments and does nothing.

celery.utils.functional.**first**(*predicate*, *it*)
>   Return the first element in `it` that `predicate` accepts.

>   If `predicate` is None it will return the first item that's not `None`.

celery.utils.functional.**firstmethod**(*method*, *on_call=None*)
>   Multiple dispatch.

>   Return a function that with a list of instances, finds the first instance that gives a value for the given method.

>   The list can also contain lazy instances (`lazy`.)

celery.utils.functional.**chunks**(*it*, *n*)
>   Split an iterator into chunks with *n* elements each.

> **Warning:** `it` must be an actual iterator, if you pass this a concrete sequence will get you repeating elements.
>
> So `chunks(iter(range(1000)), 10)` is fine, but `chunks(range(1000), 10)` is not.

Example

n == 2 >>> x = chunks(iter([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]), 2) >>> list(x) [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9], [10]]

n == 3 >>> x = chunks(iter([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]), 3) >>> list(x) [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10]]

celery.utils.functional.**padlist**(*container*, *size*, *default=None*)
    Pad list with default elements.

Example

```python
>>> first, last, city = padlist(['George', 'Costanza', 'NYC'], 3)
('George', 'Costanza', 'NYC')
>>> first, last, city = padlist(['George', 'Costanza'], 3)
('George', 'Costanza', None)
>>> first, last, city, planet = padlist(
...     ['George', 'Costanza', 'NYC'], 4, default='Earth',
... )
('George', 'Costanza', 'NYC', 'Earth')
```

celery.utils.functional.**mattrgetter**(**attrs*)
    Get attributes, ignoring attribute errors.

    Like `operator.itemgetter()` but return `None` on missing attributes instead of raising `AttributeError`.

celery.utils.functional.**uniq**(*it*)
    Return all unique elements in `it`, preserving order.

celery.utils.functional.**regen**(*it*)
    Convert iterator to an object that can be consumed multiple times.

    `Regen` takes any iterable, and if the object is an generator it will cache the evaluated list on first access, so that the generator can be "consumed" multiple times.

celery.utils.functional.**dictfilter**(*d=None*, ***kw*)
    Remove all keys from dict `d` whose value is `None`.

class celery.utils.functional.**lazy**(*fun*, **args*, ***kwargs*)
    Holds lazy evaluation.

    Evaluated when called or if the `evaluate()` method is called. The function is re-evaluated on every call.
    Overloaded operations that will evaluate the promise: `__str__()`, `__repr__()`, `__cmp__()`.
    evaluate()

celery.utils.functional.**maybe_evaluate**(*value*)
    Evaluate value only if value is a *lazy* instance.

celery.utils.functional.**head_from_fun**(*fun*, *bound=False*, *debug=False*)
    Generate signature function from actual function.

celery.utils.functional.**maybe**(*typ*, *val*)
    Call typ on value if val is defined.

`celery.utils.functional.`**`fun_accepts_kwargs`**(*fun*)
> Return true if function accepts arbitrary keyword arguments.

`celery.utils.graph`

Dependency graph implementation.

class `celery.utils.graph.`**`DOT`**
> Constants related to the dot format.
>
> **`ATTR = '{name}={value}'`**
>
> **`ATTRSEP = ', '`**
>
> **`DIRS = {'digraph': '->', 'graph': '--'}`**
>
> **`EDGE = '{INp}"{0}" {dir} "{1}" [{attrs}]'`**
>
> **`HEAD = '\n{IN}{type} {id} {{\n{INp}graph [{attrs}]\n'`**
>
> **`NODE = '{INp}"{0}" [{attrs}]'`**
>
> **`TAIL = '{IN}}}'`**

exception `celery.utils.graph.`**`CycleError`**
> A cycle was detected in an acyclic graph.

class `celery.utils.graph.`**`DependencyGraph`**(*it=None*, *formatter=None*)
> A directed acyclic graph of objects and their dependencies.
>
> Supports a robust topological sort to detect the order in which they must be handled.
>
> Takes an optional iterator of (`obj, dependencies`) tuples to build the graph from.
>
> > **Warning:** Does not support cycle detection.
>
> **`add_arc`**(*obj*)
> > Add an object to the graph.
>
> **`add_edge`**(*A*, *B*)
> > Add an edge from object `A` to object `B`.
> >
> > I.e. `A` depends on `B`.
>
> **`connect`**(*graph*)
> > Add nodes from another graph.
>
> **`edges`**()
> > Return generator that yields for all edges in the graph.
>
> **`format`**(*obj*)
>
> **`items`**()
>
> **`iteritems`**()
>
> **`repr_node`**(*obj*, *level=1*, *fmt='{0}({1})'*)
>
> **`to_dot`**(*fh*, *formatter=None*)
> > Convert the graph to DOT format.
> > > **Parameters**
> > > > • **fh** (`IO`) – A file, or a file-like object to write the graph to.

- **formatter** (celery.utils.graph.GraphFormatter) – Custom graph formatter to use.

topsort()

    Sort the graph topologically.

        Returns of objects in the order in which they must be handled.

        Return type List

update(*it*)

    Update graph with data from a list of (obj, deps) tuples.

valency_of(*obj*)

    Return the valency (degree) of a vertex in the graph.

class celery.utils.graph.**GraphFormatter**(*root=None*, *type=None*, *id=None*, *indent=0*, *inw='
'*, ***scheme*)

    Format dependency graphs.

    FMT(*fmt*, **args*, ***kwargs*)

    attr(*name*, *value*)

    attrs(*d*, *scheme=None*)

    draw_edge(*a*, *b*, *scheme=None*, *attrs=None*)

    draw_node(*obj*, *scheme=None*, *attrs=None*)

    edge(*a*, *b*, ***attrs*)

    edge_scheme = {'arrowcolor': 'black', 'arrowsize': 0.7, 'color': 'darkseagreen4'}

    graph_scheme = {'bgcolor': 'mintcream'}

    head(***attrs*)

    label(*obj*)

    node(*obj*, ***attrs*)

    node_scheme = {'color': 'palegreen4', 'fillcolor': 'palegreen3'}

    scheme = {'arrowhead': 'vee', 'fontname': 'HelveticaNeue', 'shape': 'box', 'style':

    tail()

    term_scheme = {'color': 'palegreen2', 'fillcolor': 'palegreen1'}

    terminal_node(*obj*, ***attrs*)

celery.utils.objects

Object related utilities, including introspection, etc.

class celery.utils.objects.**Bunch**(***kwargs*)

    Object that enables you to modify attributes.

class celery.utils.objects.**FallbackContext**(*provided*, *fallback*, **fb_args*, ***fb_kwargs*)

    Context workaround.

    The built-in @contextmanager utility does not work well when wrapping other contexts, as the traceback is wrong when the wrapped context raises.

    This solves this problem and can be used instead of @contextmanager in this example:

```
@contextmanager
def connection_or_default_connection(connection=None):
    if connection:
        # user already has a connection, shouldn't close
        # after use
        yield connection
    else:
        # must've new connection, and also close the connection
        # after the block returns
        with create_new_connection() as connection:
            yield connection
```

This wrapper can be used instead for the above like this:

```
def connection_or_default_connection(connection=None):
    return FallbackContext(connection, create_new_connection)
```

class celery.utils.objects.**getitem_property**(*keypath*, *doc=None*)

Attribute -> dict key descriptor.

The target object must support __getitem__, and optionally __setitem__.

Example

```
>>> from collections import defaultdict
```

```
>>> class Me(dict):
...     deep = defaultdict(dict)
...
...     foo = _getitem_property('foo')
...     deep_thing = _getitem_property('deep.thing')
```

```
>>> me = Me()
>>> me.foo
None
```

```
>>> me.foo = 10
>>> me.foo
10
>>> me['foo']
10
```

```
>>> me.deep_thing = 42
>>> me.deep_thing
42
>>> me.deep
defaultdict(<type 'dict'>, {'thing': 42})
```

celery.utils.objects.**mro_lookup**(*cls*, *attr*, *stop=None*, *monkey_patched=None*)

Return the first node by MRO order that defines an attribute.

> **Parameters**
> * **cls** (*Any*) – Child class to traverse.
> * **attr** (*str*) – Name of attribute to find.
> * **stop** (*Set[Any]*) – A set of types that if reached will stop the search.

- • **monkey_patched** (*Sequence*) – Use one of the stop classes if the attributes module origin isn't in this list. Used to detect monkey patched attributes.

  Returns The attribute value, or `None` if not found.

  Return type Any

celery.utils.term

Terminals and colors.

class celery.utils.term.**colored**(**s*, ***kwargs*)

Terminal colored text.

Example

```
>>> c = colored(enabled=True)
>>> print(str(c.red('the quick '), c.blue('brown ', c.bold('fox ')),
...       c.magenta(c.underline('jumps over')),
...       c.yellow(' the lazy '),
...       c.green('dog ')))
```

black(**s*)

blink(**s*)

blue(**s*)

bold(**s*)

bright(**s*)

cyan(**s*)

embed()

green(**s*)

iblue(**s*)

icyan(**s*)

igreen(**s*)

imagenta(**s*)

ired(**s*)

iwhite(**s*)

iyellow(**s*)

magenta(**s*)

no_color()

node(*s*, *op*)

red(**s*)

reset(**s*)

reverse(**s*)

underline(**s*)

white(**s*)

yellow(**s*)

Utilities related to dates, times, intervals, and timezones.

class celery.utils.time.**LocalTimezone**
Local time implementation.

Note: Used only when the `enable_utc` setting is disabled.

---

dst(*dt*)
datetime -> DST offset as timedelta positive east of UTC.

fromutc(*dt*)
datetime in UTC -> datetime in local time.

tzname(*dt*)
datetime -> string name of time zone.

utcoffset(*dt*)
datetime -> timedelta showing offset from UTC, negative values indicating West of UTC

celery.utils.time.**maybe_timedelta**(*delta*)
Convert integer to timedelta, if argument is an integer.

celery.utils.time.**delta_resolution**(*dt*, *delta*)
Round a `datetime` to the resolution of timedelta.

If the `timedelta` is in days, the `datetime` will be rounded to the nearest days, if the `timedelta` is in hours the `datetime` will be rounded to the nearest hour, and so on until seconds, which will just return the original `datetime`.

celery.utils.time.**remaining**(*start*, *ends_in*, *now=None*, *relative=False*)
Calculate the remaining time for a start date and a timedelta.

For example, "how many seconds left for 30 seconds after start?"
> **Parameters**
> - **start** (`datetime`) – Starting date.
> - **ends_in** (`timedelta`) – The end delta.
> - **relative** (`bool`) – If enabled the end time will be calculated using `delta_resolution()` (i.e., rounded to the resolution of *ends_in*).
> - **now** (`Callable`) – Function returning the current time and date. Defaults to `datetime.utcnow()`.
>
> **Returns** Remaining time.
> **Return type** timedelta

celery.utils.time.**rate**(*r*)
Convert rate string (*"100/m"*, *"2/h"* or *"0.5/s"*) to seconds.

celery.utils.time.**weekday**(*name*)
Return the position of a weekday: 0 - 7, where 0 is Sunday.

Example

```python
>>> weekday('sunday'), weekday('sun'), weekday('mon')
(0, 0, 1)
```

celery.utils.time.**humanize_seconds**(*secs*, *prefix=''*, *sep=''*, *now='now'*, *microseconds=False*)
    Show seconds in human form.

    For example, 60 becomes "1 minute", and 7200 becomes "2 hours".

        Parameters
            • **prefix** (`str`) – can be used to add a preposition to the output (e.g., 'in' will give 'in 1 second', but add nothing to 'now').
            • **now** (`str`) – Literal 'now'.
            • **microseconds** (`bool`) – Include microseconds.

celery.utils.time.**maybe_iso8601**(*dt*)
    Either `datetime | str -> datetime` or `None -> None`.

celery.utils.time.**is_naive**(*dt*)
    Return `True` if `datetime` is naive.

celery.utils.time.**make_aware**(*dt*, *tz*)
    Set timezone for a `datetime` object.

celery.utils.time.**localize**(*dt*, *tz*)
    Convert aware `datetime` to another timezone.

celery.utils.time.**to_utc**(*dt*)
    Convert naive `datetime` to UTC.

celery.utils.time.**maybe_make_aware**(*dt*, *tz=None*)
    Convert dt to aware datetime, do nothing if dt is already aware.

class celery.utils.time.**ffwd**(*year=None*, *month=None*, *weeks=0*, *weekday=None*, *day=None*, *hour=None*, *minute=None*, *second=None*, *microsecond=None*, ***kwargs*)
    Version of `dateutil.relativedelta` that only supports addition.

celery.utils.time.**utcoffset**(*time=<module 'time' (built-in)>*, *localtime=<built-in function localtime>*)
    Return the current offset to UTC in hours.

celery.utils.time.**adjust_timestamp**(*ts*, *offset*, *here=<function utcoffset>*)
    Adjust timestamp based on provided utcoffset.

celery.utils.time.**get_exponential_backoff_interval**(*factor*, *retries*, *maximum*, *full_jitter=False*)
    Calculate the exponential backoff wait time.

celery.utils.iso8601

Parse ISO8601 dates.

Originally taken from pyiso8601 (https://bitbucket.org/micktwomey/pyiso8601)

Modified to match the behavior of `dateutil.parser`:

  • raise `ValueError` instead of `ParseError`

  • return naive `datetime` by default

  • uses `pytz.FixedOffset`

This is the original License:

Copyright (c) 2007 Michael Twomey

celery.utils.iso8601.**parse_iso8601**(*datestring*)
>   Parse and convert ISO-8601 string to datetime.

celery.utils.saferepr

Streaming, truncating, non-recursive version of `repr()`.

Differences from regular `repr()`:

- Sets are represented the Python 3 way: `{1, 2}` vs `set([1, 2])`.

- Unicode strings does not have the `u'` prefix, even on Python 2.

- Empty set formatted as `set()` (Python 3), not `set([])` (Python 2).

- Longs don't have the `L` suffix.

Very slow with no limits, super quick with limits.

celery.utils.saferepr.**saferepr**(*o*, *maxlen=None*, *maxlevels=3*, *seen=None*)
>   Safe version of `repr()`.

> **Warning:** Make sure you set the maxlen argument, or it will be very slow for recursive objects. With the maxlen set, it's often faster than built-in repr.

celery.utils.saferepr.**reprstream**(*stack*, *seen=None*, *maxlevels=3*, *level=0*, *isinstance=<built-in function isinstance>*)
>   Streaming repr, yielding tokens.

celery.utils.serialization

Utilities for safely pickling exceptions.

exception celery.utils.serialization.**UnpickleableExceptionWrapper**(*exc_module*, *exc_cls_name*, *exc_args*, *text=None*)

>   Wraps unpickleable exceptions.
>   >   **Parameters**

- **exc_module** (*str*) – See *exc_module*.
- **exc_cls_name** (*str*) – See *exc_cls_name*.
- **exc_args** (*Tuple[Any, ..]*) – See *exc_args*.

Example

```
>>> def pickle_it(raising_function):
...     try:
...         raising_function()
...     except Exception as e:
...         exc = UnpickleableExceptionWrapper(
...             e.__class__.__module__,
...             e.__class__.__name__,
...             e.args,
...         )
...         pickle.dumps(exc)  # Works fine.
```

exc_args = None
    The arguments for the original exception.

exc_cls_name = None
    The name of the original exception class.

exc_module = None
    The module of the original exception.

classmethod from_exception(*exc*)

restore()

celery.utils.serialization.**subclass_exception**(*name*, *parent*, *module*)
    Create new exception class.

celery.utils.serialization.**find_pickleable_exception**(*exc*, *loads=<built-in function loads>*, *dumps=<built-in function dumps>*)

Find first pickleable exception base class.

With an exception instance, iterate over its super classes (by MRO) and find the first super exception that's pickleable. It does not go below Exception (i.e., it skips Exception, BaseException and object). If that happens you should use UnpickleableException instead.

Parameters
- **exc** (*BaseException*) – An exception instance.
- **loads** – decoder to use.
- **dumps** – encoder to use

Returns
    Nearest pickleable parent exception class (except Exception and parents), or if the exception is pickleable it will return None.
Return type Exception

celery.utils.serialization.**create_exception_cls**(*name*, *module*, *parent=None*)
    Dynamically create an exception class.

celery.utils.serialization.**get_pickleable_exception**(*exc*)
    Make sure exception is pickleable.

celery.utils.serialization.**get_pickleable_etype**(*cls*, *loads=<built-in function loads>*, *dumps=<built-in function dumps>*)
    Get pickleable exception type.

---

celery.utils.serialization.**get_pickled_exception**(*exc*)
   Reverse of *get_pickleable_exception()*.

celery.utils.serialization.**strtobool**(*term*, *table=None*)
   Convert common terms for true/false to bool.

   Examples (true/false/yes/no/on/off/1/0).

celery.utils.sysinfo

System information utilities.

celery.utils.sysinfo.**load_average**()
   Return system load average as a triple.

class celery.utils.sysinfo.**df**(*path*)
   Disk information.

   available

   capacity

   stat

   total_blocks

celery.utils.threads

Threading primitives and utilities.

class celery.utils.threads.**bgThread**(*name=None*, ***kwargs*)
   Background service thread.

   body()

   on_crash(*msg*, **fmt*, ***kwargs*)

   run()
      Method representing the thread's activity.

      You may override this method in a subclass. The standard run() method invokes the callable object passed
      to the object's constructor as the target argument, if any, with sequential and keyword arguments taken
      from the args and kwargs arguments, respectively.

   stop()
      Graceful shutdown.

class celery.utils.threads.**Local**
   Local object.

celery.utils.threads.**LocalStack**
   alias of celery.utils.threads._LocalStack

class celery.utils.threads.**LocalManager**(*locals=None*, *ident_func=None*)
   Local objects cannot manage themselves.

   For that you need a local manager. You can pass a local manager multiple locals or add them later by appending
   them to manager.locals. Every time the manager cleans up, it will clean up all the data left in the locals
   for this context.

   The ident_func parameter can be added to override the default ident function for the wrapped locals.

cleanup()
> Manually clean up the data in the locals for this context.
>
> Call this at the end of the request or use `make_middleware()`.

get_ident()
> Return context identifier.
>
> This is the indentifer the local objects use internally for this context. You cannot override this method to change the behavior but use it to link other context local objects (such as SQLAlchemy's scoped sessions) to the Werkzeug locals.

`celery.utils.threads.`**get_ident**() → integer
> Return a non-zero integer that uniquely identifies the current thread amongst other threads that exist simultaneously. This may be used to identify per-thread resources. Even though on some platforms threads identities may appear to be allocated consecutive numbers starting at 1, this behavior should not be relied upon, and the number should be seen purely as a magic cookie. A thread's identity may be reused for another thread after it exits.

`celery.utils.threads.`**default_socket_timeout**(*timeout*)
> Context temporarily setting the default socket timeout.

celery.utils.timer2

Scheduler for Python functions.

Note: This is used for the thread-based worker only, not for amqp/redis/sqs/qpid where `kombu.asynchronous.timer` is used.

---

class `celery.utils.timer2.`**Entry**(*fun*, *args=None*, *kwargs=None*)
> Schedule Entry.
>
> **args**
>
> **cancel**()
>
> **canceled**
>
> **cancelled**
>
> **fun**
>
> **kwargs**
>
> **tref**

`celery.utils.timer2.`**Schedule**
> alias of `kombu.asynchronous.timer.Timer`

class `celery.utils.timer2.`**Timer**(*schedule=None*, *on_error=None*, *on_tick=None*, *on_start=None*, *max_interval=None*, ***kwargs*)
> Timer thread.

Note: This is only used for transports not supporting AsyncIO.

> **class Entry**(*fun*, *args=None*, *kwargs=None*)
> > Schedule Entry.
> >
> > **args**

> > **cancel**()
>
> > **canceled**
>
> > **cancelled**
>
> > **fun**
>
> > **kwargs**
>
> > **tref**
>
> **Schedule**
>
> > alias of `kombu.asynchronous.timer.Timer`
>
> **call_after**(*args*, **kwargs*)
>
> **call_at**(*args*, **kwargs*)
>
> **call_repeatedly**(*args*, **kwargs*)
>
> **cancel**(*tref*)
>
> **clear**()
>
> **empty**()
>
> **ensure_started**()
>
> **enter**(*entry*, *eta*, *priority=None*)
>
> **enter_after**(*args*, **kwargs*)
>
> **exit_after**(*secs*, *priority=10*)
>
> **next**()
>
> **on_tick = None**
>
> **queue**
>
> **run**()
>
> > Method representing the thread's activity.
> >
> > You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.
>
> **running = False**
>
> **stop**()

celery.utils.timer2.**to_timestamp**(*d*, *default_timezone=<UTC>*, *time=<built-in function mono-tonic>*)

> Convert datetime to timestamp.
>
> If d' is already a timestamp, then that will be used.

celery.utils.imports

Utilities related to importing modules and symbols by name.

exception celery.utils.imports.**NotAPackage**

> Raised when importing a package, but it's not a package.

celery.utils.imports.**qualname**(*obj*)

> Return object name.

`celery.utils.imports.`**`instantiate`**(*name*, **args*, ***kwargs*)

> Instantiate class by name.
>
> **See also:**
>
> *symbol_by_name()*.

`celery.utils.imports.`**`symbol_by_name`**(*name*, *aliases=None*, *imp=None*, *package=None*, *sep='.'*, *default=None*, ***kwargs*)

> Get symbol by qualified name.
>
> The name should be the full dot-separated path to the class:

```
modulename.ClassName
```

> Example:

```
celery.concurrency.processes.TaskPool
                          ^- class name
```

> or using ':' to separate module and symbol:

```
celery.concurrency.processes:TaskPool
```

> If *aliases* is provided, a dict containing short name/long name mappings, the name is looked up in the aliases first.

Examples

```
>>> symbol_by_name('celery.concurrency.processes.TaskPool')
<class 'celery.concurrency.processes.TaskPool'>
```

```
>>> symbol_by_name('default', {
...     'default': 'celery.concurrency.processes.TaskPool'})
<class 'celery.concurrency.processes.TaskPool'>
```

> # Does not try to look up non-string names. >>> from celery.concurrency.processes import TaskPool >>> symbol_by_name(TaskPool) is TaskPool True

`celery.utils.imports.`**`cwd_in_path`**()

> Context adding the current working directory to sys.path.

`celery.utils.imports.`**`find_module`**(*module*, *path=None*, *imp=None*)

> Version of `imp.find_module()` supporting dots.

`celery.utils.imports.`**`import_from_cwd`**(*module*, *imp=None*, *package=None*)

> Import module, temporarily including modules in the current directory.
>
> Modules located in the current directory has precedence over modules located in *sys.path*.

`celery.utils.imports.`**`reload_from_cwd`**(*module*, *reloader=None*)

> Reload module (ensuring that CWD is in sys.path).

`celery.utils.imports.`**`module_file`**(*module*)

> Return the correct original file name of a module.

`celery.utils.imports.`**`gen_task_name`**(*app*, *name*, *module_name*)

> Generate task name from name/module pair.

`celery.utils.log`

Logging utilities.

class `celery.utils.log.`**`ColorFormatter`**(*fmt=None*, *use_color=True*)
    Logging formatter that adds colors based on severity.

    `COLORS = {'black':  <bound method colored.black of ''>, 'blue':  <bound method colored`
        Loglevel -> Color mapping.

    `colors = {'CRITICAL': <bound method colored.magenta of ''>, 'DEBUG': <bound method col`

    `format`(*record*)
        Format the specified record as text.

        The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

    `formatException`(*ei*)
        Format and return the specified exception information as a string.

        This default implementation just uses traceback.print_exception()

class `celery.utils.log.`**`LoggingProxy`**(*logger*, *loglevel=None*)
    Forward file object to `logging.Logger` instance.
        Parameters
                • **logger** (*Logger*) – Logger instance to forward to.
                • **loglevel** (*int, str*) – Log level to use when logging messages.

    `close`()

    `closed = False`

    `flush`()

    `isatty`()
        Here for file support.

    `loglevel = 40`

    `mode = 'w'`

    `name = None`

    `write`(*data*)
        Write message to logging object.

    `writelines`(*sequence*)
        Write list of strings to file.

        The sequence can be any iterable object producing strings. This is equivalent to calling *write()* for each string.

`celery.utils.log.`**`set_in_sighandler`**(*value*)
    Set flag signifiying that we're inside a signal handler.

`celery.utils.log.`**`in_sighandler`**()
    Context that records that we are in a signal handler.

`celery.utils.log.`**`get_logger`**(*name*)
    Get logger by name.

celery.utils.log.**get_task_logger**(*name*)
    Get logger for task module by name.

celery.utils.log.**mlevel**(*level*)
    Convert level name/int to log level.

celery.utils.log.**get_multiprocessing_logger**()
    Return the multiprocessing logger.

celery.utils.log.**reset_multiprocessing_logger**()
    Reset multiprocessing logging setup.

celery.utils.text

Text formatting utilities.

celery.utils.text.**abbr**(*S*, *max*, *ellipsis='...'*)
    Abbreviate word.

celery.utils.text.**abbrtask**(*S*, *max*)
    Abbreviate task name.

celery.utils.text.**dedent**(*s*, *n=4*, *sep='\n'*)
    Remove identation.

celery.utils.text.**dedent_initial**(*s*, *n=4*)
    Remove identation from first line of text.

celery.utils.text.**ensure_newlines**(*s*, *n=2*)
    Ensure text s ends in separator sep'.

celery.utils.text.**ensure_sep**(*sep*, *s*, *n=2*)
    Ensure text s ends in separator sep'.

celery.utils.text.**fill_paragraphs**(*s*, *width*, *sep='\n'*)
    Fill paragraphs with newlines (or custom separator).

celery.utils.text.**indent**(*t*, *indent=0*, *sep='\n'*)
    Indent text.

celery.utils.text.**join**(*l*, *sep='\n'*)
    Concatenate list of strings.

celery.utils.text.**pluralize**(*n*, *text*, *suffix='s'*)
    Pluralize term when n is greater than one.

celery.utils.text.**pretty**(*value*, *width=80*, *nl_width=80*, *sep='\n'*, ***kw*)
    Format value for printing to console.

celery.utils.text.**str_to_list**(*s*)
    Convert string to list.

celery.utils.text.**simple_format**(*s*, *keys*, *pattern=re.compile('%(\w)')*, *expand='\\1'*)
    Format string, expanding abbreviations in keys'.

celery.utils.text.**truncate**(*s*, *maxlen=128*, *suffix='...'*)
    Truncate text to a maximum number of characters.

celery.utils.dispatch

Observer pattern.

class celery.utils.dispatch.**Signal**(*providing_args=None*, *use_caching=False*, *name=None*)
    Create new signal.

        Keyword Arguments

- **providing_args** (*List*) – A list of the arguments this signal can pass along in a `send()` call.
- **use_caching** (*bool*) – Enable receiver cache.
- **name** (*str*) – Name of signal, used for debugging purposes.

    connect(**args*, ***kwargs*)
        Connect receiver to sender for signal.

            Parameters

- **receiver** (*Callable*) – A function or an instance method which is to receive signals. Receivers must be hashable objects.

  if weak is `True`, then receiver must be weak-referenceable.

  Receivers must be able to accept keyword arguments.

  If receivers have a *dispatch_uid* attribute, the receiver will not be added if another receiver already exists with that *dispatch_uid*.
- **sender** (*Any*) – The sender to which the receiver should respond. Must either be a Python object, or `None` to receive events from any sender.
- **weak** (*bool*) – Whether to use weak references to the receiver. By default, the module will attempt to use weak references to the receiver objects. If this parameter is false, then strong references will be used.
- **dispatch_uid** (*Hashable*) – An identifier used to uniquely identify a particular instance of a receiver. This will usually be a string, though it may be anything hashable.
- **retry** (*bool*) – If the signal receiver raises an exception (e.g. ConnectionError), the receiver will be retried until it runs successfully. A strong ref to the receiver will be stored and the *weak* option will be ignored.

    disconnect(*receiver=None*, *sender=None*, *weak=None*, *dispatch_uid=None*)
        Disconnect receiver from sender for signal.

        If weak references are used, disconnect needn't be called. The receiver will be removed from dispatch automatically.

            Parameters

- **receiver** (*Callable*) – The registered receiver to disconnect. May be none if *dispatch_uid* is specified.
- **sender** (*Any*) – The registered sender to disconnect.
- **weak** (*bool*) – The weakref state to disconnect.
- **dispatch_uid** (*Hashable*) – The unique identifier of the receiver to disconnect.

    has_listeners(*sender=None*)

    receivers = None
        Holds a dictionary of `{receiverkey (id): weakref(receiver)}` mappings.

    send(*sender*, ***named*)
        Send signal from sender to all connected receivers.

        If any receiver raises an error, the error propagates back through send, terminating the dispatch loop, so it is quite possible to not have all receivers called if a raises an error.

            Parameters

- **sender** (*Any*) – The sender of the signal. Either a specific object or `None`.
- ****named** (*Any*) – Named arguments which will be passed to receivers.

            Returns of tuple pairs: *[(receiver, response), ...]*.

> **Return type** List

send_robust(*sender*, ***named*)
> Send signal from sender to all connected receivers.

> If any receiver raises an error, the error propagates back through send, terminating the dispatch loop, so it is quite possible to not have all receivers called if a raises an error.
> > **Parameters**
> > * **sender** (`Any`) – The sender of the signal. Either a specific object or `None`.
> > * ****named** (`Any`) – Named arguments which will be passed to receivers.
> > **Returns** of tuple pairs: *[(receiver, response), . . .].*
> > **Return type** List

celery.utils.dispatch.signal

Implementation of the Observer pattern.

class celery.utils.dispatch.signal.**Signal**(*providing_args=None*, *use_caching=False*, *name=None*)
> Create new signal.
> > **Keyword Arguments**
> > * **providing_args** (`List`) – A list of the arguments this signal can pass along in a *send()* call.
> > * **use_caching** (`bool`) – Enable receiver cache.
> > * **name** (`str`) – Name of signal, used for debugging purposes.

> **connect**(**args*, ***kwargs*)
> > Connect receiver to sender for signal.
> > > **Parameters**
> > > * **receiver** (`Callable`) – A function or an instance method which is to receive signals. Receivers must be hashable objects.
> > >
> > >   if weak is `True`, then receiver must be weak-referenceable.
> > >
> > >   Receivers must be able to accept keyword arguments.
> > >
> > >   If receivers have a *dispatch_uid* attribute, the receiver will not be added if another receiver already exists with that *dispatch_uid*.
> > > * **sender** (`Any`) – The sender to which the receiver should respond. Must either be a Python object, or `None` to receive events from any sender.
> > > * **weak** (`bool`) – Whether to use weak references to the receiver. By default, the module will attempt to use weak references to the receiver objects. If this parameter is false, then strong references will be used.
> > > * **dispatch_uid** (`Hashable`) – An identifier used to uniquely identify a particular instance of a receiver. This will usually be a string, though it may be anything hashable.
> > > * **retry** (`bool`) – If the signal receiver raises an exception (e.g. Connection-Error), the receiver will be retried until it runs successfully. A strong ref to the receiver will be stored and the *weak* option will be ignored.

> **disconnect**(*receiver=None*, *sender=None*, *weak=None*, *dispatch_uid=None*)
> > Disconnect receiver from sender for signal.

> > If weak references are used, disconnect needn't be called. The receiver will be removed from dispatch automatically.
> > > **Parameters**
> > > * **receiver** (`Callable`) – The registered receiver to disconnect. May be none if *dispatch_uid* is specified.

> - **sender** (*Any*) – The registered sender to disconnect.
> - **weak** ([*bool*](#)) – The weakref state to disconnect.
> - **dispatch_uid** (*Hashable*) – The unique identifier of the receiver to disconnect.

has_listeners(*sender=None*)

receivers = None
: Holds a dictionary of `{receiverkey (id): weakref(receiver)}` mappings.

send(*sender*, ***named*)
: Send signal from sender to all connected receivers.

    If any receiver raises an error, the error propagates back through send, terminating the dispatch loop, so it is quite possible to not have all receivers called if a raises an error.
    > **Parameters**
    > - **sender** (*Any*) – The sender of the signal. Either a specific object or `None`.
    > - ****named** (*Any*) – Named arguments which will be passed to receivers.
    >
    > **Returns** of tuple pairs: *[(receiver, response), . . .].*
    > **Return type** List

send_robust(*sender*, ***named*)
: Send signal from sender to all connected receivers.

    If any receiver raises an error, the error propagates back through send, terminating the dispatch loop, so it is quite possible to not have all receivers called if a raises an error.
    > **Parameters**
    > - **sender** (*Any*) – The sender of the signal. Either a specific object or `None`.
    > - ****named** (*Any*) – Named arguments which will be passed to receivers.
    >
    > **Returns** of tuple pairs: *[(receiver, response), . . .].*
    > **Return type** List

celery.utils.dispatch.weakref_backports

Weakref compatibility.

weakref_backports is a partial backport of the weakref module for Python versions below 3.4.

Copyright (C) 2013 Python Software Foundation, see LICENSE.python for details.

The following changes were made to the original sources during backporting:

- Added `self` to `super` calls.

- Removed `from None` when raising exceptions.

class celery.utils.dispatch.weakref_backports.**WeakMethod**
: Weak reference to bound method.

    A custom [weakref.ref](#) subclass which simulates a weak reference to a bound method, working around the lifetime problem of bound methods.

celery.platforms

Platforms.

Utilities dealing with platform specifics: signals, daemonization, users, groups, and so on.

celery.platforms.**pyimplementation**()
: Return string identifying the current Python implementation.

exception `celery.platforms.`**`LockFailed`**
> Raised if a PID lock can't be acquired.

`celery.platforms.`**`get_fdmax`**(*default=None*)
> Return the maximum number of open file descriptors on this system.
>> **Keyword Arguments** **`default`** – Value returned if there's no file descriptor limit.

class `celery.platforms.`**`Pidfile`**(*path*)
> Pidfile.
>
> This is the type returned by *`create_pidlock()`*.
>
> **See also:**
>
> Best practice is to not use this directly but rather use the *`create_pidlock()`* function instead: more convenient and also removes stale pidfiles (when the process holding the lock is no longer running).
>
> **`acquire`**()
>> Acquire lock.
>
> **`is_locked`**()
>> Return true if the pid lock exists.
>
> **`path = None`**
>> Path to the pid lock file.
>
> **`read_pid`**()
>> Read and return the current pid.
>
> **`release`**(**args*)
>> Release lock.
>
> **`remove`**()
>> Remove the lock.
>
> **`remove_if_stale`**()
>> Remove the lock if the process isn't running.
>>
>> I.e. process does not respons to signal.
>
> **`write_pid`**()

`celery.platforms.`**`create_pidlock`**(*pidfile*)
> Create and verify pidfile.
>
> If the pidfile already exists the program exits with an error message, however if the process it refers to isn't running anymore, the pidfile is deleted and the program continues.
>
> This function will automatically install an `atexit` handler to release the lock at exit, you can skip this by calling `_create_pidlock()` instead.
>> **Returns** used to manage the lock.
>> **Return type** *Pidfile*

> **Example**

```
>>> pidlock = create_pidlock('/var/run/app.pid')
```

`celery.platforms.`**`close_open_fds`**(*keep=None*)

class celery.platforms.**DaemonContext**(*pidfile=None*, *workdir=None*, *umask=None*, *fake=False*, *after_chdir=None*, *after_forkers=True*, ***kwargs*)

 Context manager daemonizing the process.

 close(**args*)

 open()

 redirect_to_null(*fd*)

celery.platforms.**detached**(*logfile=None*, *pidfile=None*, *uid=None*, *gid=None*, *umask=0*, *workdir=None*, *fake=False*, ***opts*)

 Detach the current process in the background (daemonize).

  Parameters

- **logfile** (*str*) – Optional log file. The ability to write to this file will be verified before the process is detached.
- **pidfile** (*str*) – Optional pid file. The pidfile won't be created, as this is the responsibility of the child. But the process will exit if the pid lock exists and the pid written is still running.
- **uid** (*int, str*) – Optional user id or user name to change effective privileges to.
- **gid** (*int, str*) – Optional group id or group name to change effective privileges to.
- **umask** (*str, int*) – Optional umask that'll be effective in the child process.
- **workdir** (*str*) – Optional new working directory.
- **fake** (*bool*) – Don't actually detach, intended for debugging purposes.
- ****opts** (*Any*) – Ignored.

Example

```
>>> from celery.platforms import detached, create_pidlock
>>> with detached(
...         logfile='/var/log/app.log',
...         pidfile='/var/run/app.pid',
...         uid='nobody'):
... # Now in detached child process with effective user set to nobody,
... # and we know that our logfile can be written to, and that
... # the pidfile isn't locked.
... pidlock = create_pidlock('/var/run/app.pid')
...
... # Run the program
... program.run(logfile='/var/log/app.log')
```

celery.platforms.**parse_uid**(*uid*)

 Parse user id.

  Parameters uid (*str, int*) – Actual uid, or the username of a user.
  Returns The actual uid.
  Return type int

celery.platforms.**parse_gid**(*gid*)

 Parse group id.

  Parameters gid (*str, int*) – Actual gid, or the name of a group.
  Returns The actual gid of the group.
  Return type int

celery.platforms.**setgroups**(*groups*)

 Set active groups from a list of group ids.

celery.platforms.**initgroups**(*uid*, *gid*)
  Init process group permissions.

  Compat version of `os.initgroups()` that was first added to Python 2.7.

celery.platforms.**setgid**(*gid*)
  Version of `os.setgid()` supporting group names.

celery.platforms.**setuid**(*uid*)
  Version of `os.setuid()` supporting usernames.

celery.platforms.**maybe_drop_privileges**(*uid=None*, *gid=None*)
  Change process privileges to new user/group.

  If UID and GID is specified, the real user/group is changed.

  If only UID is specified, the real user is changed, and the group is changed to the users primary group.

  If only GID is specified, only the group is changed.

celery.platforms.**signal_name**(*signum*)
  Return name of signal from signal number.

celery.platforms.**set_process_title**(*progname*, *info=None*)
  Set the **ps** name for the currently running process.

  Only works if setproctitle is installed.

celery.platforms.**set_mp_process_title**(*progname*, *info=None*, *hostname=None*)
  Set the **ps** name from the current process name.

  Only works if setproctitle is installed.

celery.platforms.**get_errno_name**(*n*)
  Get errno for string (e.g., ENOENT).

celery.platforms.**ignore_errno**(**errnos*, ***kwargs*)
  Context manager to ignore specific POSIX error codes.

  Takes a list of error codes to ignore: this can be either the name of the code, or the code integer itself:

```python
>>> with ignore_errno('ENOENT'):
...     with open('foo', 'r') as fh:
...         return fh.read()

>>> with ignore_errno(errno.ENOENT, errno.EPERM):
...     pass
```

  Parameters types (*Tuple[Exception]*) – A tuple of exceptions to ignore (when the errno matches). Defaults to `Exception`.

celery.platforms.**fd_by_path**(*paths*)
  Return a list of file descriptors.

  This method returns list of file descriptors corresponding to file paths passed in paths variable.
  Parameters paths – List[str]: List of file paths.
  Returns List of file descriptors.
  Return type List[int]

Example

```
>>> keep = fd_by_path(['/dev/urandom', '/my/precious/'])
```

celery.platforms.**isatty**(*fh*)
Return true if the process has a controlling terminal.

celery._state

Internal state.

This is an internal module containing thread state like the current_app, and current_task.

This module shouldn't be used directly.

celery._state.**set_default_app**(*app*)
Set default app.

celery._state.**get_current_app**()

celery._state.**get_current_task**()
Currently executing task.

celery._state.**get_current_worker_task**()
Currently executing task, that was applied by the worker.

This is used to differentiate between the actual task executed by the worker and any task that was called within a task (using task.__call__ or task.apply)

celery._state.**current_app = <Celery default>**
Proxy to current app.

celery._state.**current_task = None**
Proxy to current task.

celery._state.**connect_on_app_finalize**(*callback*)
Connect callback to be called when any app is finalized.

3.13 History

This section contains historical change histories, for the latest version please visit *Change history*.

> **Release** 4.4
>
> **Date** Mar 02, 2020

3.13.1 What's new in Celery 4.2 (windowlicker)

> **Author** Omer Katz (omer.drow at gmail.com)

Change history

What's new documents describe the changes in major versions, we also have a *Change history* that lists the changes in bugfix releases (0.0.x), while older series are archived under the *History* section.

Celery is a simple, flexible, and reliable distributed system to process vast amounts of messages, while providing operations with the tools required to maintain such a system.

It's a task queue with focus on real-time processing, while also supporting task scheduling.

Celery has a large and diverse community of users and contributors, you should come join us *on IRC* or *our mailing-list*.

To read more about Celery you should go read the *introduction*.

While this version is backward compatible with previous versions it's important that you read the following section.

This version is officially supported on CPython 2.7, 3.4, 3.5 & 3.6 and is also supported on PyPy.

Table of Contents

Make sure you read the important notes before upgrading to this version.

Preface

The 4.2.0 release continues to improve our efforts to provide you with the best task execution platform for Python.

This release is mainly a bug fix release, ironing out some issues and regressions found in Celery 4.0.0.

Traditionally, releases were named after Autechre's track names. This release continues this tradition in a slightly different way. Each major version of Celery will use a different artist's track names as codenames.

From now on, the 4.x series will be codenamed after Aphex Twin's track names. This release is codenamed after his very famous track, Windowlicker.

Thank you for your support!

— *Omer Katz*

Wall of Contributors

Aaron Harnly <aharnly@wgen.net> Aaron Harnly <github.com@bulk.harnly.net> Aaron McMillin <github@aaron.mcmillinclan.org> Aaron Ross <aaronelliotross@gmail.com> Aaron Ross <aaron@wawd.com> Aaron Schumacher <ajschumacher@gmail.com> abecciu <augusto@becciu.org> abhinav nilaratna <anilaratna2@bloomberg.net> Acey9 <huiwang.e@gmail.com> Acey <huiwang.e@gmail.com> aclowes

---

<aclowes@gmail.com> Adam Chainz <adam@adamj.eu> Adam DePue <adepue@hearsaycorp.com> Adam Endicott <adam@zoey.local> Adam Renberg <tgwizard@gmail.com> Adam Venturella <aventurella@gmail.com> Adaptification <Adaptification@users.noreply.github.com> Adrian <adrian@planetcoding.net> adriano petrich <petrich@gmail.com> Adrian Rego <arego320@gmail.com> Adrien Guinet <aguinet@quarkslab.com> Agris Ameriks <ameriks@gmail.com> Ahmet Demir <ahmet2mir+github@gmail.com> air-upc <xin.shli@ele.me> Aitor Gómez-Goiri <aitor@gomezgoiri.net> Akira Matsuzaki <akira.matsuzaki.1977@gmail.com> Akshar Raaj <akshar@agiliq.com> Alain Masiero <amasiero@ocs.online.net> Alan Hamlett <alan.hamlett@prezi.com> Alan Hamlett <alanhamlett@users.noreply.github.com> Alan Justino <alan.justino@yahoo.com.br> Alan Justino da Silva <alan.justino@yahoo.com.br> Albert Wang <albert@zerocater.com> Alcides Viamontes Esquivel <a.viamontes.esquivel@gmail.com> Alec Clowes <aclowes@gmail.com> Alejandro Pernin <ale.pernin@gmail.com> Alejandro Varas <alej0varas@gmail.com> Aleksandr Kuznetsov <aku.ru.kz@gmail.com> Ales Zoulek <ales.zoulek@gmail.com> Alexander <a.a.lebedev@gmail.com> Alexander A. Sosnovskiy <alecs.box@gmail.com> Alexander Koshelev <daevaorn@gmail.com> Alexander Koval <kovalidis@gmail.com> Alexander Oblovatniy <oblalex@users.noreply.github.com> Alexander Oblovatniy <oblovatniy@gmail.com> Alexander Ovechkin <frostoov@gmail.com> Alexander Smirnov <asmirnov@five9.com> Alexandru Chirila <alex@alexkiro.com> Alexey Kotlyarov <alexey@infoxchange.net.au> Alexey Zatelepin <ztlpn@yandex-team.ru> Alex Garel <alex@garel.org> Alex Hill <alex@hill.net.au> Alex Kiriukha <akiriukha@cogniance.com> Alex Koshelev <daevaorn@gmail.com> Alex Rattray <rattray.alex@gmail.com> Alex Williams <alex.williams@skyscanner.net> Alex Zaitsev <azaitsev@gmail.com> Ali Bozorgkhan <alibozorgkhan@gmail.com> Allan Caffee <allan.caffee@gmail.com> Allard Hoeve <allard@byte.nl> allenling <lingyiwang@haomaiyi.com> Alli <alzeih@users.noreply.github.com> Alman One <alman@laptop.home> Alman One <alman-one@laptop.home> alman-one <masiero.alain@gmail.com> Amir Rustamzadeh <amirrustam@users.noreply.github.com> anand21nanda@gmail.com <anand21nanda@gmail.com> Anarchist666 <Anarchist666@yandex.ru> Anders Pearson <anders@columbia.edu> Andrea Rabbaglietti <silverfix@gmail.com> Andreas Pelme <andreas@pelme.se> Andreas Savvides <andreas@editd.com> Andrei Fokau <andrei.fokau@neutron.kth.se> Andrew de Quincey <adq@lidskialf.net> Andrew Kittredge <andrewlkittredge@gmail.com> Andrew McFague <amcfague@wgen.net> Andrew Stewart <astewart@twistbioscience.com> Andrew Watts <andrewwatts@gmail.com> Andrew Wong <argsno@gmail.com> Andrey Voronov <eyvoro@users.noreply.github.com> Andriy Yurchuk <ayurchuk@minuteware.net> Aneil Mallavarapu <aneil.mallavar@gmail.com> anentropic <ego@anentropic.com> anh <anhlh2@gmail.com> Ankur Dedania <AbsoluteMSTR@gmail.com> Anthony Lukach <anthony-lukach@gmail.com> antlegrand <2t.antoine@gmail.com> Antoine Legrand <antoine.legrand@smartjog.com> Anton <anton.gladkov@gmail.com> Anton Gladkov <atn18@yandex-team.ru> Antonin Delpeuch <antonin@delpeuch.eu> Arcadiy Ivanov <arcadiy@ivanov.biz> areski <areski@gmail.com> Armenak Baburyan <kanemra@gmail.com> Armin Ronacher <armin.ronacher@active-4.com> armo <kanemra@gmail.com> Arnaud Rocher <caillouma-jor@users.noreply.github.com> arpanshah29 <ashah29@stanford.edu> Arsenio Santos <arsenio@gmail.com> Arthur Vigil <ahvigil@mail.sfsu.edu> Arthur Vuillard <arthur@hashbang.fr> Ashish Dubey <ashish.dubey91@gmail.com> Asif Saifuddin Auvi <auvipy@gmail.com> Asif Saifuddin Auvi <auvipy@users.noreply.github.com> ask <ask@0x61736b.net> Ask Solem <ask@celeryproject.org> Ask Solem <askh@opera.com> Ask Solem Hoel <ask@celeryproject.org> aydin <adigeaydin@gmail.com> baeuml <baeuml@kit.edu> Balachandran C <balachandran.c@gramvaani.org> Balthazar Rouberol <balthazar.rouberol@mapado.com> Balthazar Rouberol <balthazar.rouberol@ubertas.co.uk> bartloop <38962178+bartloop@users.noreply.github.com> Bartosz Ptaszynski <> Batiste Bieler <batiste.bieler@pix4d.com> bee-keeper <ricbottomley@gmail.com> Bence Tamas <mr.bence.tamas@gmail.com> Ben Firshman <ben@firshman.co.uk> Ben Welsh <ben.welsh@gmail.com> Berker Peksag <berker.peksag@gmail.com> Bert Vanderbauwhede <batlock666@gmail.com> Bert Vanderbauwhede <bert.vanderbauwhede@ugent.be> BLAGA Razvan-Paul <razvan.paul.blaga@gmail.com> bobbybeever <bobby.beever@yahoo.com> bobby <bobby.beever@yahoo.com> Bobby Powers <bobbypowers@gmail.com> Bohdan Rybak <bohdan.rybak@gmail.com> Brad Jasper <bjasper@gmail.com> Branko Čibej <brane@apache.org> BR <b.rabiega@gmail.com> Brendan MacDonell <macdonellba@gmail.com> Brendon Crawford <brendon@aphexcreations.net> Brent Watson <brent@brentwatson.com> Brian Bouterse <bmbouter@gmail.com> Brian Dixon <bjdixon@gmail.com> Brian Luan <jznight@gmail.com> Brian May <brian@linuxpenguins.xyz> Brian Peiris <brianpeiris@gmail.com> Brian Rosner <brosner@gmail.com> Brodie Rao <brodie@sf.io> Bruno Alla <browniebroke@users.noreply.github.com> Bryan Berg <bdb@northeastham.org> Bryan Berg <bryan@mixedmedialabs.com> Bryan Bishop <kanzure@gmail.com> Bryan Helmig <bryan@bryanhelmig.com> Bryce Groff <bgroff@hawaii.edu> Caleb Mingle <mingle@uber.com> Carlos Garcia-

Dubus <carlos.garciadm@gmail.com> Catalin Iacob <iacobcatalin@gmail.com> Charles McLaughlin <mclaughlinct@gmail.com> Chase Seibert <chase.seibert+github@gmail.com> ChillarAnand <anand21nanda@gmail.com> Chris Adams <chris@improbable.org> Chris Angove <cangove@wgen.net> Chris Chamberlin <chamberlincd@gmail.com> chrisclark <chris@untrod.com> Chris Harris <chris.harris@kitware.com> Chris Kuehl <chris@techxonline.net> Chris Martin <ch.martin@gmail.com> Chris Mitchell <chris.mit7@gmail.com> Chris Rose <offby1@offby1.net> Chris St. Pierre <chris.a.st.pierre@gmail.com> Chris Streeter <chris@chrisstreeter.com> Christian <github@penpal4u.net> Christoph Burgmer <christoph@nwebs.de> Christopher Hoskin <mans0954@users.noreply.github.com> Christopher Lee <chris@cozi.com> Christopher Peplin <github@rhubarbtech.com> Christopher Peplin <peplin@bueda.com> Christoph Krybus <ckrybus@googlemail.com> clayg <clay.gerrard@gmail.com> Clay Gerrard <clayg@clayg-desktop.(none)> Clemens Wolff <clemens@justamouse.com> cmclaughlin <mclaughlinct@gmail.com> Codeb Fan <codeb2cc@gmail.com> Colin McIntosh <colin@colinmcintosh.com> Conrad Kramer <ckrames1234@gmail.com> Corey Farwell <coreyf@rwell.org> Craig Younkins <cyounkins@Craigs-MacBook-Pro.local> csfeathers <csfeathers@users.noreply.github.com> Cullen Rhodes <rhodes.cullen@yahoo.co.uk> daftshady <daftonshady@gmail.com> Dan <dmtaub@gmail.com> Dan Hackner <dan.hackner@gmail.com> Daniel Devine <devine@ddevnet.net> Daniele Procida <daniele@vurt.org> Daniel Hahler <github@thequod.de> Daniel Hepper <daniel.hepper@gmail.com> Daniel Huang <dxhuang@gmail.com> Daniel Lundin <daniel.lundin@trioptima.com> Daniel Lundin <dln@eintr.org> Daniel Watkins <daniel@daniel-watkins.co.uk> Danilo Bargen <mail@dbrgn.ch> Dan McGee <dan@archlinux.org> Dan McGee <dpmcgee@gmail.com> Dan Wilson <danjwilson@gmail.com> Daodao <daodaod@gmail.com> Dave Smith <dave@thesmithfam.org> Dave Smith <dsmith@hirevue.com> David Arthur <darthur@digitalsmiths.com> David Arthur <mumrah@gmail.com> David Baumgold <david@davidbaumgold.com> David Cramer <dcramer@gmail.com> David Davis <daviddavis@users.noreply.github.com> David Harrigan <dharrigan118@gmail.com> David Harrigan <dharrigan@dyn.com> David Markey <dmarkey@localhost.localdomain> David Miller <david@deadpansincerity.com> David Miller <il.livid.dream@gmail.com> David Pravec <David.Pravec@danix.org> David Pravec <david.pravec@nethost.cz> David Strauss <david@davidstrauss.net> David White <dpwhite2@ncsu.edu> DDevine <devine@ddevnet.net> Denis Podlesniy <Haos616@Gmail.com> Denis Shirokov <dan@rexuni.com> Dennis Brakhane <dennis.brakhane@inoio.de> Derek Harland <donkopotamus@users.noreply.github.com> derek_kim <bluewhale8202@gmail.com> dessant <dessant@users.noreply.github.com> Dieter Adriaenssens <ruleant@users.sourceforge.net> Dima Kurguzov <koorgoo@gmail.com> dimka665 <dimka665@gmail.com> dimlev <dimlev@gmail.com> dmarkey <david@dmarkey.com> Dmitry Malinovsky <damalinov@gmail.com> Dmitry Malinovsky <dmalinovsky@thumbtack.net> dmollerm <d.moller.m@gmail.com> Dmytro Petruk <bavaria95@gmail.com> dolugen <dolugen@gmail.com> dongweiming <ciici1234@hotmail.com> dongweiming <ciici123@gmail.com> Dongweiming <ciici123@gmail.com> dtheodor <dimitris.theodorou@gmail.com> Dudás Ádám <sir.dudas.adam@gmail.com> Dustin J. Mitchell <dustin@mozilla.com> D. Yu <darylyu@users.noreply.github.com> Ed Morley <edmorley@users.noreply.github.com> Eduardo Ramírez <ejramire@uc.cl> Edward Betts <edward@4angle.com> Emil Stanchev <stanchev.emil@gmail.com> Eran Rundstein <eran@sandsquid.(none)> ergo <ergo@debian.Belkin> Eric Poelke <epoelke@gmail.com> Eric Zarowny <ezarowny@gmail.com> ernop <ernestfrench@gmail.com> Evgeniy <quick.es@gmail.com> evildmp <daniele@apple-juice.co.uk> fatihsucu <fatihsucu0@gmail.com> Fatih Sucu <fatihsucu@users.noreply.github.com> Feanil Patel <feanil@edx.org> Felipe <fcoelho@users.noreply.github.com> Felipe Godói Rosário <felipe.rosario@geru.com.br> Felix Berger <bflat1@gmx.net> Fengyuan Chen <cfy1990@gmail.com> Fernando Rocha <fernandogrd@gmail.com> ffeast <ffeast@gmail.com> Flavio Percoco Premoli <flaper87@gmail.com> Florian Apolloner <apollo13@apolloner.eu> Florian Apolloner <florian@apollo13.(none)> Florian Demmer <fdemmer@gmail.com> flyingfoxlee <lingyunzhi312@gmail.com> Francois Visconte <f.visconte@gmail.com> François Voron <fvoron@gmail.com> Frédéric Junod <frederic.junod@camptocamp.com> fredj <frederic.junod@camptocamp.com> frol <frolvlad@gmail.com> Gabriel <gabrielpjordao@gmail.com> Gao Jiangmiao <gao.jiangmiao@h3c.com> GDR! <gdr@gdr.name> GDvalle <GDvalle@users.noreply.github.com> Geoffrey Bauduin <bauduin.geo@gmail.com> georgepsarakis <giwrgos.psarakis@gmail.com> George Psarakis <giwrgos.psarakis@gmail.com> George Sibble <gsibble@gmail.com> George Tantiras <raratiru@users.noreply.github.com> Georgy Cheshkov <medoslav@gmail.com> Gerald Manipon <pymonger@gmail.com> German M. Bravo <german.mb@deipi.com> Gert Van Gool <gertvangool@gmail.com> Gilles Dartiguelongue <gilles.dartiguelongue@esiee.org> Gino Ledesma <gledesma@apple.com> gmanipon <gmanipon@jpl.nasa.gov> Grant Thomas <jgrantthomas@gmail.com> Greg Haskins <greg@greghaskins.com> gregoire <gregoire@audacy.fr> Greg Taylor <gtaylor@duointeractive.com>

Greg Wilbur <gwilbur@bloomberg.net> Guillaume Gauvrit <guillaume@gandi.net> Guillaume Gendre <dzb.rtz@gmail.com> Gun.io Whitespace Robot <contact@gun.io> Gunnlaugur Thor Briem <gunnlaugur@gmail.com> harm <harm.verhagen@gmail.com> Harm Verhagen <harm.verhagen@gmail.com> Harry Moreno <morenoh149@gmail.com> hclihn <23141651+hclihn@users.noreply.github.com> hekevintran <hekevintran@gmail.com> honux <atoahp@hotmail.com> Honza Kral <honza.kral@gmail.com> Honza Král <Honza.Kral@gmail.com> Hooksie <me@matthooks.com> Hsiaoming Yang <me@lepture.com> Huang Huang <mozillazg101@gmail.com> Hynek Schlawack <hs@ox.cx> Hynek Schlawack <schlawack@variomedia.de> Ian Dees <ian.dees@gmail.com> Ian McCracken <ian.mccracken@gmail.com> Ian Wilson <ian.owings@gmail.com> Idan Kamara <idankk86@gmail.com> Ignas Mikalajūnas <ignas.mikalajunas@gmail.com> Igor Kasianov <super.hang.glider@gmail.com> illes <illes.solt@gmail.com> Ilya <4beast@gmail.com> Ilya Georgievsky <i.georgievsky@drweb.com> Ionel Cristian Mărieș <contact@ionelmc.ro> Ionel Maries Cristian <contact@ionelmc.ro> Ionut Turturica <jonozzz@yahoo.com> Iurii Kriachko <iurii.kriachko@gmail.com> Ivan Metzlar <metzlar@gmail.com> Ivan Virabyan <i.virabyan@gmail.com> j0hnsmith <info@whywouldwe.com> Jackie Leng <Jackie.Leng@nelen-schuurmans.nl> J Alan Brogan <jalanb@users.noreply.github.com> Jameel Al-Aziz <me@jalaziz.net> James M. Allen <james.m.allen@gmail.com> James Michael DuPont <JamesMikeDuPont@gmail.com> James Pulec <jpulec@gmail.com> James Remeika <james@remeika.us> Jamie Alessio <jamie@stoic.net> Jannis Leidel <jannis@leidel.info> Jared Biel <jared.biel@bolderthinking.com> Jason Baker <amnorvend@gmail.com> Jason Baker <jason@ubuntu.ubuntu-domain> Jason Veatch <jtveatch@gmail.com> Jasper Bryant-Greene <jbg@rf.net.nz> Javier Domingo Cansino <javierdo1@gmail.com> Javier Martin Montull <javier.martin.montull@cern.ch> Jay Farrimond <jay@instaedu.com> Jay McGrath <jaymcgrath@users.noreply.github.com> jbiel <jared.biel@bolderthinking.com> jbochi <jbochi@gmail.com> Jed Smith <jed@jedsmith.org> Jeff Balogh <github@jeffbalogh.org> Jeff Balogh <me@jeffbalogh.org> Jeff Terrace <jterrace@gmail.com> Jeff Widman <jeff@jeffwidman.com> Jelle Verstraaten <jelle.verstraaten@xs4all.nl> Jeremy Cline <jeremy@jcline.org> Jeremy Zafran <jeremy.zafran@cloudlock.com> jerry <jerry@stellaservice.com> Jerzy Kozera <jerzy.kozera@gmail.com> Jerzy Kozera <jerzy.kozera@sensisoft.com> jespern <jesper@noehr.org> Jesper Noehr <jespern@jesper-noehrs-macbook-pro.local> Jesse <jvanderdoes@gmail.com> jess <jessachandler@gmail.com> Jess Johnson <jess@grokcode.com> Jian Yu <askingyj@gmail.com> JJ <jairojair@gmail.com> João Ricardo <joaoricardo000@gmail.com> Jocelyn Delalande <jdelalande@oasiswork.fr> JocelynDelalande <JocelynDelalande@users.noreply.github.com> Joe Jevnik <JoeJev@gmail.com> Joe Sanford <joe@cs.tufts.edu> Joe Sanford <josephsanford@gmail.com> Joey Wilhelm <tarkatronic@gmail.com> John Anderson <sontek@gmail.com> John Arnold <johnar@microsoft.com> John Barham <jbarham@gmail.com> John Watson <john@dctrwatson.com> John Watson <john@disqus.com> John Watson <johnw@mahalo.com> John Whitlock <John-Whitlock@ieee.org> Jonas Haag <jonas@lophus.org> Jonas Obrist <me@ojii.ch> Jonatan Heyman <jonatan@heyman.info> Jonathan Jordan <jonathan@metaltoad.com> Jonathan Sundqvist <sundqvist.jonathan@gmail.com> jonathan vanasco <jonathan@2xlp.com> Jon Chen <bsd@voltaire.sh> Jon Dufresne <jon.dufresne@gmail.com> Josh <kaizoku@phear.cc> Josh Kupershmidt <schmiddy@gmail.com> Joshua "jag" Ginsberg <jag@flowtheory.net> Josue Balandrano Coronel <xirdneh@gmail.com> Jozef <knaperek@users.noreply.github.com> jpellerin <jpellerin@jpdesk.(none)> jpellerin <none@none> JP <jpellerin@gmail.com> JTill <jtillman@hearsaycorp.com> Juan Gutierrez <juanny.gee@gmail.com> Juan Ignacio Catalano <catalanojuan@gmail.com> Juan Rossi <juan@getmango.com> Juarez Bochi <jbochi@gmail.com> Jude Nagurney <jude@pwan.org> Julien Deniau <julien@sitioweb.fr> julienp <julien@caffeine.lu> Julien Poissonnier <julien@caffeine.lu> Jun Sakai <jsakai@splunk.com> Justin Patrin <jpatrin@skyhighnetworks.com> Justin Patrin <papercrane@reversefold.com> Kalle Bronsen <bronsen@nrrd.de> kamalgill <kamalgill@mac.com> Kamil Breguła <mik-laj@users.noreply.github.com> Kanan Rahimov <mail@kenanbek.me> Kareem Zidane <kzidane@cs50.harvard.edu> Keith Perkins <keith@tasteoftheworld.us> Ken Fromm <ken@frommworldwide.com> Ken Reese <krrg@users.noreply.github.com> keves <e@keves.org> Kevin Gu <guqi@reyagroup.com> Kevin Harvey <kharvey@axialhealthcare.com> Kevin McCarthy <me@kevinmccarthy.org> Kevin Richardson <kevin.f.richardson@gmail.com> Kevin Richardson <kevin@kevinrichardson.co> Kevin Tran <hekevintran@gmail.com> Kieran Brownlees <kbrownlees@users.noreply.github.com> Kirill Pavlov <pavlov99@yandex.ru> Kirill Romanov <djaler1@gmail.com> komu <komuw05@gmail.com> Konstantinos Koukopoulos <koukopoulos@gmail.com> Konstantin Podshumok <kpp.live@gmail.com> Kornelijus Survila <kornholijo@gmail.com> Kouhei Maeda <mkouhei@gmail.com> Kracekumar Ramaraju <me@kracekumar.com> Krzysztof Bujniewicz <k.bujniewicz@bankier.pl> kuno <neokuno@gmail.com> Kxrr <Hi@Kxrr.Us> Kyle Kelley <rgbkrk@gmail.com> Laurent Peuch <cortex@worlddomination.be> lead2gold

<caronc@users.noreply.github.com> Leo Dirac <leo@banyanbranch.com> Leo Singer <leo.singer@ligo.org> Lewis M. Kabui <lewis.maina@andela.com> llllllllll <joejev@gmail.com> Locker537 <Locker537@gmail.com> Loic Bistuer <loic.bistuer@sixmedia.com> Loisaida Sam <sam.sandberg@gmail.com> lookfwd <lookfwd@gmail.com> Loren Abrams <labrams@hearsaycorp.com> Loren Abrams <loren.abrams@gmail.com> Lucas Wiman <lucaswiman@counsyl.com> lucio <lucio@prometeo.spirit.net.ar> Luis Clara Gomez <ekkolabs@gmail.com> Lukas Linhart <lukas.linhart@centrumholdings.com> Łukasz Kożuchowski <lukasz.kozuchowski@10clouds.com> Łukasz Langa <lukasz@langa.pl> Łukasz Oleś <lukaszoles@gmail.com> Luke Burden <lukeburden@gmail.com> Luke Hutscal <luke@creaturecreative.com> Luke Plant <L.Plant.98@cantab.net> Luke Pomfrey <luke.pomfrey@titanemail.com> Luke Zapart <drx@drx.pl> mabouels <abouelsaoud@gmail.com> Maciej Obuchowski <obuchowski.maciej@gmail.com> Mads Jensen <mje@inducks.org> Manuel Kaufmann <humitos@gmail.com> Manuel Vázquez Acosta <mvaled@users.noreply.github.com> Marat Sharafutdinov <decaz89@gmail.com> Marcelo Da Cruz Pinto <Marcelo_DaCruzPinto@McAfee.com> Marc Gibbons <marc_gibbons@rogers.com> Marc Hörsken <mback2k@users.noreply.github.com> Marcin Kuźmiński <marcin@python-blog.com> marcinkuzminski <marcin@python-works.com> Marcio Ribeiro <binary@b1n.org> Marco Buttu <marco.buttu@gmail.com> Marco Schweighauser <marco@mailrelay.ch> mariia-zelenova <32500603+mariia-zelenova@users.noreply.github.com> Marin Atanasov Nikolov <dnaeon@gmail.com> Marius Gedminas <marius@gedmin.as> mark hellewell <mark.hellewell@gmail.com> Mark Lavin <markdlavin@gmail.com> Mark Lavin <mlavin@caktusgroup.com> Mark Parncutt <me@markparncutt.com> Mark Story <mark@freshbooks.com> Mark Stover <stovenator@gmail.com> Mark Thurman <mthurman@gmail.com> Markus Kaiserswerth <github@sensun.org> Markus Ullmann <mail@markus-ullmann.de> martialp <martialp@users.noreply.github.com> Martin Davidsson <martin@dropcam.com> Martin Galpin <m@66laps.com> Martin Melin <git@martinmelin.com> Matt Davis <matteius@gmail.com> Matthew Duggan <mgithub@guarana.org> Matthew J Morrison <mattj.morrison@gmail.com> Matthew Miller <matthewgarrettmiller@gmail.com> Matthew Schinckel <matt@schinckel.net> mattlong <matt@crocodoc.com> Matt Long <matt@crocodoc.com> Matt Robenolt <matt@ydekproductions.com> Matt Robenolt <m@robenolt.com> Matt Williamson <dawsdesign@gmail.com> Matt Williamson <matt@appdelegateinc.com> Matt Wise <matt@nextdoor.com> Matt Woodyard <matt@mattwoodyard.com> Mauro Rocco <fireantology@gmail.com> Maxim Bodyansky <maxim@viking.(none)> Maxime Beauchemin <maxime.beauchemin@apache.org> Maxime Vdb <mvergerdelbove@work4labs.com> Mayflower <fucongwang@gmail.com> mbacho <mbacho@users.noreply.github.com> mher <mher.movsisyan@gmail.com> Mher Movsisyan <mher.movsisyan@gmail.com> Michael Aquilina <michaelaquilina@gmail.com> Michael Duane Mooring <mikeumus@gmail.com> Michael Elsdoerfer michael@elsdoerfer.com <michael@puppetmaster.(none)> Michael Elsdorfer <michael@elsdoerfer.com> Michael Elsdörfer <michael@elsdoerfer.com> Michael Fladischer <FladischerMichael@fladi.at> Michael Floering <michaelfloering@gmail.com> Michael Howitz <mh@gocept.com> michael <michael@giver.dpool.org> Michael <michael-k@users.noreply.github.com> michael <michael@puppetmaster.(none)> Michael Peake <michaeljpeake@icloud.com> Michael Permana <michael@origamilogic.com> Michael Permana <mpermana@hotmail.com> Michael Robellard <mikerobellard@onshift.com> Michael Robellard <mrobellard@onshift.com> Michal Kuffa <beezz@users.noreply.github.com> Miguel Hernandez Martos <enlavin@gmail.com> Mike Attwood <mike@cybersponse.com> Mike Chen <yi.chen.it@gmail.com> Mike Helmick <michaelhelmick@users.noreply.github.com> mikemccabe <mike@mcca.be> Mikhail Gusarov <dottedmag@dottedmag.net> Mikhail Korobov <kmike84@gmail.com> Mikołaj <mikolevy1@gmail.com> Milen Pavlov <milen.pavlov@gmail.com> Misha Wolfson <myw@users.noreply.github.com> Mitar <mitar.github@tnode.com> Mitar <mitar@tnode.com> Mitchel Humpherys <mitch.special@gmail.com> mklauber <matt+github@mklauber.com> mlissner <mlissner@michaeljaylissner.com> monkut <nafein@hotmail.com> Morgan Doocy <morgan@doocy.net> Morris Tweed <tweed.morris@gmail.com> Morton Fox <github@qslw.com> Môshe van der Sterre <me@moshe.nl> Moussa Taifi <moutai10@gmail.com> mozillazg <opensource.mozillazg@gmail.com> mpavlov <milen.pavlov@gmail.com> mperice <mperice@users.noreply.github.com> mrmmm <mohammad.almeer@gmail.com> Muneyuki Noguchi <nogu.dev@gmail.com> m-vdb <mvergerdelbove@work4labs.com> nadad <nadad6@gmail.com> Nathaniel Varona <nathaniel.varona@gmail.com> Nathan Van Gheem <vangheem@gmail.com> Nat Williams <nat.williams@gmail.com> Neil Chintomby <mace033@gmail.com> Neil Chintomby <neil@mochimedia.com> Nicholas Pilon <npilon@gmail.com> nicholsonjf <nicholsonjf@gmail.com> Nick Eaket <4418194+neaket360pi@users.noreply.github.com> Nick Johnson <njohnson@limcollective.com> Nicolas Mota <nicolas_mota@live.com> nicolasunravel <nicolas@unravel.ie> Niklas Aldergren <niklas@aldergren.com> Noah Kantrowitz <noah@coderanger.net> Noel

Remy <mocramis@gmail.com> NoKriK <nokrik@nokrik.net> Norman Richards <orb@nostacktrace.com> NotSqrt <notsqrt@gmail.com> nott <reg@nott.cc> ocean1 <ocean1@users.noreply.github.com> ocean1 <ocean_ieee@yahoo.it> ocean1 <ocean.kuzuri@gmail.com> OddBloke <daniel.watkins@glassesdirect.com> Oleg Anashkin <oleg.anashkin@gmail.com> Olivier Aubert <contact@olivieraubert.net> Omar Khan <omar@omarkhan.me> Omer Katz <omer.drow@gmail.com> Omer Korner <omerkorner@gmail.com> orarbel <orarbel@gmail.com> orf <tom@tomforb.es> Ori Hoch <ori@uumpa.com> outself <yura.nevsky@gmail.com> Pablo Marti <pmargam@gmail.com> pachewise <pachewise@users.noreply.github.com> partizan <serg.partizan@gmail.com> Pär Wieslander <wieslander@gmail.com> Patrick Altman <paltman@gmail.com> Patrick Cloke <clokep@users.noreply.github.com> Patrick <paltman@gmail.com> Patrick Stegmann <code@patrick-stegmann.de> Patrick Stegmann <wonderb0lt@users.noreply.github.com> Patrick Zhang <pat-dujour@gmail.com> Paul English <paul@onfrst.com> Paul Jensen <pjensen@interactdirect.com> Paul Kilgo <pkilgo@clemson.edu> Paul McMillan <paul.mcmillan@nebula.com> Paul McMillan <Paul@McMillan.ws> Paulo <PauloPeres@users.noreply.github.com> Paul Pearce <pearce@cs.berkeley.edu> Pavel Savchenko <pavel@modlinltd.com> Pavlo Kapyshin <i@93z.org> pegler <pegler@gmail.com> Pepijn de Vos <pepi-jndevos@gmail.com> Peter Bittner <django@bittner.it> Peter Brook <peter.d.brook@gmail.com> Philip Garnero <philip.garnero@corp.ovh.com> Pierre Fersing <pierref@pierref.org> Piotr Maślanka <pi-otr.maslanka@henrietta.com.pl> Piotr Sikora <piotr.sikora@frickle.com> PMickael <exploze@gmail.com> PMick-ael <mickael.penhard@gmail.com> Polina Giralt <polina.giralt@gmail.com> precious <vs.kulaga@gmail.com> Preston Moore <prestonkmoore@gmail.com> Primož Kerin <kerin.primoz@gmail.com> Pysaoke <pysaoke@gmail.com> Rachel Johnson <racheljohnson457@gmail.com> Rachel Willmer <rachel@willmer.org> raducc <raducc@users.noreply.github.com> Raf Geens <rafgeens@gmail.com> Raghuram Srinivasan <raghu@set.tv> Raphaël Riel <raphael.riel@gmail.com> Raphaël Slinckx <rslinckx@gmail.com> Régis B <github@behmo.com> Remigiusz Modrzejewski <lrem@maxnet.org.pl> Rémi Marenco <remi.marenco@gmail.com> rfkrocktk <rfkrocktk@gmail.com> Rick van Hattem <rick.van.hattem@fawo.nl> Rick Wargo <rickwargo@users.noreply.github.com> Rico Moorman <rico.moorman@gmail.com> Rik <gi-taarik@gmail.com> Rinat Shigapov <rinatshigapov@gmail.com> Riyad Parvez <social.riyad@gmail.com> rlotun <rlotun@gmail.com> rnoel <rnoel@ltutech.com> Robert Knight <robertknight@gmail.com> Roberto Gaiser <gaiser@geekbunker.org> roderick <mail@roderick.de> Rodolphe Quiedeville <rodolphe@quiedeville.org> Roger Hu <rhu@hearsaycorp.com> Roger Hu <roger.hu@gmail.com> Roman Imankulov <roman@netangels.ru> Roman Sichny <roman@sichnyi.com> Romuald Brunet <romuald@gandi.net> Ronan Amicel <ro-nan.amicel@gmail.com> Ross Deane <ross.deane@gmail.com> Ross Lawley <ross.lawley@gmail.com> Ross Patterson <me@rpatterson.net> Ross <ross@duedil.com> Rudy Attias <rudy.attias@gmail.com> rumyana neykova <rumi.neykova@gmail.com> Rumyana Neykova <rumi.neykova@gmail.com> Rune Halvorsen <runefh@gmail.com> Rune Halvorsen <runeh@vorkosigan.(none)> runeh <runeh@vorkosigan.(none)> Russell Keith-Magee <russell@keith-magee.com> Ryan Guest <ryanguest@gmail.com> Ryan Hiebert <ryan@ryanhiebert.com> Ryan Kelly <rkelly@truveris.com> Ryan Luckie <rtluckie@gmail.com> Ryan Pe-trello <lists@ryanpetrello.com> Ryan P. Kelly <rpkelly@cpan.org> Ryan P Kilby <rpkilby@ncsu.edu> Salvatore Rinchiera <srinchiera@college.harvard.edu> Sam Cooke <sam@mixcloud.com> samjy <sam+git@samjy.com> Sammie S. Taunton <diemuzi@gmail.com> Samuel Dion-Girardeau <samueldg@users.noreply.github.com> Samuel Dion-Girardeau <samuel.diongirardeau@gmail.com> Samuel GIFFARD <samuel@giffard.co> Scott Cooper <scttcper@gmail.com> screeley <screeley@screeley-laptop.(none)> sdcooke <sam@mixcloud.com> Sean O'Connor <sean@seanoc.com> Sean Wang <seanw@patreon.com> Sebastian Kalinowski <sebastian@kalinowski.eu> Sébastien Fievet <zyegfryed@gmail.com> Seong Won Mun <longfinfunnel@gmail.com> Sergey Fursov <GeyseR85@gmail.com> Sergey Tikhonov <zimbler@gmail.com> Sergi Almacellas Abellana <sergi@koolpi.com> Sergio Fernandez <ElAutoestopista@users.noreply.github.com> Seungha Kim <seungha.dev@gmail.com> shalev67 <shalev67@gmail.com> Shitikanth <golu3990@gmail.com> Silas Sewell <silas@sewell.org> Simon Charette <charette.s@gmail.com> Simon Engledew <simon@engledew.com> Simon Josi <simon.josi@atizo.com> Simon Legner <Simon.Legner@gmail.com> Simon Peeters <peeters.simon@gmail.com> Simon Schmidt <schmidt.simon@gmail.com> skovorodkin <sergey@skovorodkin.com> Slam <3lnc.slam@gmail.com> Smirl <smirlie@googlemail.com> squfrans <frans@squla.com> Srinivas Garlapati <srinivasa.b.garlapati@gmail.com> Stas Rudakou <stas@garage22.net> Static <staticfox@staticfox.net> Steeve Morin <steeve.morin@gmail.com> Stefan hr Berder <stefan.berder@ledapei.com> Stefan Kjartansson <esteban.supreme@gmail.com> Steffen All-ner <sa@gocept.com> Stephen Weber <mordel@gmail.com> Steven Johns <duoi@users.noreply.github.com> Steven Parker <voodoonofx@gmail.com> Steven <rh0dium@users.noreply.github.com> Steven Sklar

<steve@predata.com> Steven Skoczen <steven@aquameta.com> Steven Skoczen <steven@quantumimagery.com> Steve Peak <steve@stevepeak.net> stipa <stipa@debian.local.local> sukrit007 <sukrit007@gmail.com> Sukrit Khera <sukrit007@gmail.com> Sundar Raman <cybertoast@gmail.com> sunfinite <sunfinite@gmail.com> sww <sww@users.noreply.github.com> Tadej Janež <tadej.janez@tadej.hicsalta.si> Taha Jahangir <mtjahangir@gmail.com> Takeshi Kanemoto <tak.kanemoto@gmail.com> TakesxiSximada <takesxi.sximada@gmail.com> Tamer Sherif <tamer.sherif@flyingelephantlab.com> Tao Qingyun <845767657@qq.com> Tarun Bhardwaj <mailme@tarunbhardwaj.com> Tayfun Sen <tayfun.sen@markafoni.com> Tayfun Sen <tayfun.sen@skyscanner.net> Tayfun Sen <totayfun@gmail.com> tayfun <tayfun.sen@markafoni.com> Taylor C. Richberger <taywee@gmx.com> taylornelson <taylor@sourcedna.com> Theodore Dubois <tbodt@users.noreply.github.com> Theo Spears <github@theos.me.uk> Thierry RAMORASOAVINA <thierry.ramorasoavina@orange.com> Thijs Triemstra <info@collab.nl> Thomas French <thomas@sandtable.com> Thomas Grainger <tagrain@gmail.com> Thomas Johansson <prencher@prencher.dk> Thomas Meson <zllak@hycik.org> Thomas Minor <sxeraverx@gmail.com> Thomas Wright <tom.tdw@gmail.com> Timo Sugliani <timo.sugliani@gmail.com> Timo Sugliani <tsugliani@tsugliani-desktop.(none)> Titusz <tp@py7.de> tnir <tnir@users.noreply.github.com> Tobias Kunze <rixx@cutebit.de> Tocho Tochev <tocho@tochev.net> Tomas Machalek <tomas.machalek@gmail.com> Tomasz Święcicki <tomislater@gmail.com> Tom 'Biwaa' Riat <riat.tom@gmail.com> Tomek Święcicki <tomislater@gmail.com> Tom S <scytale@gmail.com> tothegump <tothegump@gmail.com> Travis Swicegood <development@domain51.com> Travis Swicegood <travis@domain51.com> Travis <treeder@gmail.com> Trevor Skaggs <skaggs.trevor@gmail.com> Ujjwal Ojha <ojhaujjwal@users.noreply.github.com> unknown <Jonatan@.(none)> Valentyn Klindukh <vklindukh@cogniance.com> Viktor Holmqvist <viktorholmqvist@gmail.com> Vincent Barbaresi <vbarbaresi@users.noreply.github.com> Vincent Driessen <vincent@datafox.nl> Vinod Chandru <vinod.chandru@gmail.com> Viraj <vnavkal0@gmail.com> Vitaly Babiy <vbabiy86@gmail.com> Vitaly <olevinsky.v.s@gmail.com> Vivek Anand <vivekanand1101@users.noreply.github.com> Vlad <frolvlad@gmail.com> Vladimir Gorbunov <vsg@suburban.me> Vladimir Kryachko <v.kryachko@gmail.com> Vladimir Rutsky <iamironbob@gmail.com> Vladislav Stepanov <8uk.8ak@gmail.com> Vsevolod <Vsevolod@zojax.com> Wes Turner <wes.turner@gmail.com> wes <wes@policystat.com> Wes Winham <winhamwr@gmail.com> w- <github@wangsanata.com> whendrik <whendrik@gmail.com> Wido den Hollander <wido@widodh.nl> Wieland Hoffmann <mineo@users.noreply.github.com> Wiliam Souza <wiliamsouza83@gmail.com> Wil Langford <wil.langford+github@gmail.com> William King <willtrking@gmail.com> Will <paradox41@users.noreply.github.com> Will Thompson <will@willthompson.co.uk> winhamwr <winhamwr@gmail.com> Wojciech Zywno <w.zywno@gmail.com> W. Trevor King <wking@tremily.us> wyc <wayne@neverfear.org> wyc <wyc@fastmail.fm> xando <sebastian.pawlus@gmail.com> Xavier Damman <xdamman@gmail.com> Xavier Hardy <xavierhardy@users.noreply.github.com> Xavier Ordoquy <xordoquy@linovia.com> xin li <xin.shli@ele.me> xray7224 <xray7224@googlemail.com> y0ngdi <36658095+y0ngdi@users.noreply.github.com> Yan Kalchevskiy <yan.kalchevskiy@gmail.com> Yohann Rebattu <yohann@rebattu.fr> Yoichi NAKAYAMA <yoichi.nakayama@gmail.com> Yuhannaa <yuhannaa@gmail.com> YuLun Shih <shih@yulun.me> Yury V. Zaytsev <yury@shurup.com> Yuval Greenfield <ubershmekel@gmail.com> Zach Smith <zmsmith27@gmail.com> Zhang Chi <clvrobj@gmail.com> Zhaorong Ma <mazhaorong@gmail.com> Zoran Pavlovic <xcepticzoki@gmail.com> ztlpn <mvzp10@gmail.com> (Sean Ho) <h1x2y3awalm@gmail.com> <wdv4758h@gmail.com>

Note: This wall was automatically generated from git history, so sadly it doesn't not include the people who help with more important things like answering mailing-list questions.

Important Notes

Supported Python Versions

The supported Python Versions are:

- CPython 2.7

- CPython 3.4
- CPython 3.5
- CPython 3.6
- PyPy 5.8 (`pypy2`)

News

Result Backends

New Redis Sentinel Results Backend

Redis Sentinel provides high availability for Redis. A new result backend supporting it was added.

Cassandra Results Backend

A new *cassandra_options* configuration option was introduced in order to configure the cassandra client.

See *Cassandra backend settings* for more information.

DynamoDB Results Backend

A new *dynamodb_endpoint_url* configuration option was introduced in order to point the result backend to a local endpoint during development or testing.

See *AWS DynamoDB backend settings* for more information.

Python 2/3 Compatibility Fixes

Both the CouchDB and the Consul result backends accepted byte strings without decoding them to Unicode first. This is now no longer the case.

Canvas

Multiple bugs were resolved resulting in a much smoother experience when using Canvas.

Tasks

Bound Tasks as Error Callbacks

We fixed a regression that occurred when bound tasks are used as error callbacks. This used to work in Celery 3.x but raised an exception in 4.x until this release.

In both 4.0 and 4.1 the following code wouldn't work:

```
@app.task(name="raise_exception", bind=True)
def raise_exception(self):
    raise Exception("Bad things happened")


@app.task(name="handle_task_exception", bind=True)
def handle_task_exception(self):
    print("Exception detected")

subtask = raise_exception.subtask()

subtask.apply_async(link_error=handle_task_exception.s())
```

Task Representation

- Shadowing task names now works as expected. The shadowed name is properly presented in flower, the logs and the traces.
- *argsrepr* and *kwargsrepr* were previously not used even if specified. They now work as expected. See *Hiding sensitive information in arguments* for more information.

Custom Requests

We now allow tasks to use custom *request* classes for custom task classes.

See *Requests and custom requests* for more information.

Retries with Exponential Backoff

Retries can now be performed with exponential backoffs to avoid overwhelming external services with requests.

See *Automatic retry for known exceptions* for more information.

Sphinx Extension

Tasks were supposed to be automatically documented when using Sphinx's Autodoc was used. The code that would have allowed automatic documentation had a few bugs which are now fixed.

Also, The extension is now documented properly. See *Documenting Tasks with Sphinx* for more information.

3.13.2 Change history

This document contains change notes for bugfix releases in the 4.x series, please see *What's new in Celery 4.2 (windowlicker)* for an overview of what's new in Celery 4.2.

4.2.1

> **release-date** 2018-07-18 11:00 AM IST
>
> **release-by** Omer Katz

- **Result Backend**: Fix deserialization of exceptions that are present in the producer codebase but not in the consumer codebase.

  Contributed by **John Arnold**

- **Message Protocol Compatibility**: Fix error caused by an invalid (None) timelimit value in the message headers when migrating messages from 3.x to 4.x.

  Contributed by **Robert Kopaczewski**

- **Result Backend**: Fix serialization of exception arguments when exception arguments are not JSON serializable by default.

  Contributed by **Tom Booth**

- **Worker**: Fixed multiple issues with rate limited tasks

  Maintain scheduling order. Fix possible scheduling of a `celery.worker.request.Request` with the wrong `kombu.utils.limits.TokenBucket` which could cause tasks' rate limit to behave incorrectly. Fix possible duplicated execution of tasks that were rate limited or if ETA/Countdown was provided for them.

  Contributed by @ideascf

- **Worker**: Defensively handle invalid timelimit header values in requests.

  Contributed by **Omer Katz**

Documentation fixes:

- **Matt Wiens**
- **Seunghun Lee**
- **Lewis M. Kabui**
- **Prathamesh Salunkhe**

4.2.0

release-date 2018-06-10 21:30 PM IST

release-by Omer Katz

- **Task**: Add `ignore_result` as task execution option (#4709, #3834)

    Contributed by **Andrii Kostenko** and **George Psarakis**.

- **Redis Result Backend**: Do not create PubSub subscriptions when results are ignored (#4709, #3834)

    Contributed by **Andrii Kostenko** and **George Psarakis**.

- **Redis Result Backend**: Result consumer always unsubscribes when task state is ready (#4666)

    Contributed by **George Psarakis**.

- **Development/Testing**: Add docker-compose and base Dockerfile for development (#4482)

    Contributed by **Chris Mitchell**.

- **Documentation/Sphinx**: Teach autodoc to document tasks if undoc-members is not set (#4588)

    Contributed by **Leo Singer**.

- **Documentation/Sphinx**: Put back undoc-members option in sphinx test (#4586)

    Contributed by **Leo Singer**.

- **Documentation/Sphinx**: Sphinx autodoc picks up tasks automatically only if *undoc-members* is set (#4584)

Contributed by **Leo Singer**.

- **Task**: Fix shadow_name issue when using previous version Task class (#4572)

    Contributed by @pachewise.

- **Task**: Add support for bound tasks as *link_error* parameter (Fixes #3723) (#4545)

    Contributed by @brabiega.

- **Deployment**: Add a command line option for setting the Result Backend URL (#4549)

    Contributed by @y0ngdi.

- **CI**: Enable pip cache in appveyor build (#4546)

    Contributed by **Thijs Triemstra**.

- **Concurrency/Asynpool**: Fix errno property name shadowing.

    Contributed by **Omer Katz**.

- **DynamoDB Backend**: Configurable endpoint URL (#4532)

    Contributed by **Bohdan Rybak**.

- **Timezones**: Correctly detect UTC timezone and timezone from settings (Fixes #4517) (#4519)

    Contributed by @last-partizan.

- **Control**: Cleanup the mailbox's producer pool after forking (#4472)

    Contributed by **Nick Eaket**.

- **Documentation**: Start Celery and Celery Beat on Azure WebJob (#4484)

    Contributed by **PauloPeres**.

- **Celery Beat**: Schedule due tasks on startup, after Beat restart has occurred (#4493)

    Contributed by **Igor Kasianov**.

- **Worker**: Use absolute time when task is accepted by worker pool (#3684)

    Contributed by **Régis Behmo**.

- **Canvas**: Propagate arguments to chains inside groups (#4481)

    Contributed by **Chris Mitchell**.

- **Canvas**: Fix *Task.replace* behavior in nested chords (fixes #4368) (#4369)

    Contributed by **Denis Shirokov** & **Alex Hill**.

- **Installation**: Pass python_requires argument to setuptools (#4479)

    Contributed by **Jon Dufresne**.

- **Message Protocol Compatibility**: Handle "hybrid" messages that have moved between Celery versions (#4358) (Issue #4356)

    Contributed by **Russell Keith-Magee**.

- **Canvas**: request on_timeout now ignores soft time limit exception (fixes #4412) (#4473)

    Contributed by **Alex Garel**.

- **Redis Result Backend**: Integration test to verify PubSub unsubscriptions (#4468)

    Contributed by **George Psarakis**.

- **Message Protocol Properties**: Allow the shadow keyword argument and the shadow_name method to set shadow properly (#4381)

    Contributed by @hclihn.

- **Canvas**: Run chord_unlock on same queue as chord body (#4448) (Issue #4337)

    Contributed by **Alex Hill**.

- **Canvas**: Support chords with empty header group (#4443)

    Contributed by **Alex Hill**.

- **Timezones**: make astimezone call in localize more safe (#4324)

    Contributed by **Matt Davis**.

- **Canvas**: Fix length-1 and nested chords (#4437) (Issues #4393, #4055, #3885, #3597, #3574, #3323, #4301)

    Contributed by **Alex Hill**.

- **CI**: Run Openstack Bandit in Travis CI in order to detect security issues.

    Contributed by **Omer Katz**.

- **CI**: Run isort in Travis CI in order to lint Python **import** statements.

    Contributed by **Omer Katz**.

- **Canvas**: Resolve TypeError on *.get* from nested groups (#4432) (Issue #4274)

    Contributed by **Misha Wolfson**.

- **CouchDB Backend**: Correct CouchDB key string type for Python 2/3 compatibility (#4166)

    Contributed by @fmind && **Omer Katz**.

- **Group Result**: Fix current_app fallback in GroupResult.restore() (#4431)

    Contributed by **Alex Hill**.

- **Consul Backend**: Correct key string type for Python 2/3 compatibility (#4416)

    Contributed by **Wido den Hollander**.

- **Group Result**: Correctly restore an empty GroupResult (#2202) (#4427)

    Contributed by **Alex Hill** & **Omer Katz**.

- **Result**: Disable synchronous waiting for sub-tasks on eager mode(#4322)

    Contributed by **Denis Podlesniy**.

- **Celery Beat**: Detect timezone or Daylight Saving Time changes (#1604) (#4403)

    Contributed by **Vincent Barbaresi**.

- **Canvas**: Fix append to an empty chain. Fixes #4047. (#4402)

    Contributed by **Omer Katz**.

- **Task**: Allow shadow to override task name in trace and logging messages. (#4379)

    Contributed by @hclihn.

- **Documentation/Sphinx**: Fix getfullargspec Python 2.x compatibility in contrib/sphinx.py (#4399)

    Contributed by **Javier Martin Montull**.

- **Documentation**: Updated installation instructions for SQS broker (#4382)

> Contributed by **Sergio Fernandez**.

- **Celery Beat**: Better equality comparison for ScheduleEntry instances (#4312)

  > Contributed by @mariia-zelenova.

- **Task**: Adding 'shadow' property to as_task_v2 (#4350)

  > Contributed by **Marcelo Da Cruz Pinto**.

- Try to import directly, do not use deprecated imp method (#4216)

  > Contributed by **Tobias Kunze**.

- **Task**: Enable *kwargsrepr* and *argsrepr* override for modifying task argument representation (#4260)

  > Contributed by **James M. Allen**.

- **Result Backend**: Add Redis Sentinel backend (#4144)

  > Contributed by **Geoffrey Bauduin**.

- Use unique time values for Collections/LimitedSet (#3879 and #3891) (#3892)

  > Contributed by @lead2gold.

- **CI**: Report coverage for all result backends.

  > Contributed by **Omer Katz**.

- **Django**: Use Django DB max age connection setting (fixes #4116) (#4292)

  > Contributed by **Marco Schweighauser**.

- **Canvas**: Properly take into account chain tasks link_error (#4240)

  > Contributed by @agladkov.

- **Canvas**: Allow to create group with single task (fixes issue #4255) (#4280)

  > Contributed by @agladkov.

- **Canvas**: Copy dictionary parameter in chord.from_dict before modifying (fixes issue #4223) (#4278)

  > Contributed by @agladkov.

- **Results Backend**: Add Cassandra options (#4224)

  > Contributed by **Scott Cooper**.

- **Worker**: Apply rate limiting for tasks with ETA (#4251)

  > Contributed by @arpanshah29.

- **Celery Beat**: support scheduler entries without a schedule (#4235)

  > Contributed by **Markus Kaiserswerth**.

- **SQS Broker**: Updated SQS requirements file with correct boto3 version (#4231)

  > Contributed by **Alejandro Varas**.

- Remove unused code from _create_app contextmanager (#4204)

  > Contributed by **Ryan P Kilby**.

- **Group Result**: Modify GroupResult.as_tuple() to include parent (fixes #4106) (#4205)

  > Contributed by @pachewise.

- **Beat**: Set default scheduler class in beat command. (#4189)

Contributed by @Kxrr.

- **Worker**: Retry signal receiver after raised exception (#4192)

    Contributed by **David Davis**.

- **Task**: Allow custom Request class for tasks (#3977)

    Contributed by **Manuel Vázquez Acosta**.

- **Django**: Django fixup should close all cache backends (#4187)

    Contributed by **Raphaël Riel**.

- **Deployment**: Adds stopasgroup to the supervisor scripts (#4200)

    Contributed by @martialp.

- Using Exception.args to serialize/deserialize exceptions (#4085)

    Contributed by **Alexander Ovechkin**.

- **Timezones**: Correct calculation of application current time with timezone (#4173)

    Contributed by **George Psarakis**.

- **Remote Debugger**: Set the SO_REUSEADDR option on the socket (#3969)

    Contributed by **Theodore Dubois**.

- **Django**: Celery ignores exceptions raised during *django.setup()* (#4146)

    Contributed by **Kevin Gu**.

- Use heartbeat setting from application configuration for Broker connection (#4148)

    Contributed by @mperice.

- **Celery Beat**: Fixed exception caused by next_transit receiving an unexpected argument. (#4103)

    Contributed by **DDevine**.

- **Task** Introduce exponential backoff with Task auto-retry (#4101)

    Contributed by **David Baumgold**.

- **AsyncResult**: Remove weak-references to bound methods in AsyncResult promises. (#4131)

    Contributed by **Vinod Chandru**.

- **Development/Testing**: Allow eager application of canvas structures (#4576)

    Contributed by **Nicholas Pilon**.

- **Command Line**: Flush stderr before exiting with error code 1.

    Contributed by **Antonin Delpeuch**.

- **Task**: Escapes single quotes in kwargsrepr strings.

    Contributed by **Kareem Zidane**

- **AsyncResult**: Restore ability to join over ResultSet after fixing celery/#3818.

    Contributed by **Derek Harland**

- **Redis Results Backend**: Unsubscribe on message success.

    Previously Celery would leak channels, filling the memory of the Redis instance.

    Contributed by **George Psarakis**

- **Task**: Only convert eta to isoformat when it is not already a string.

  Contributed by **Omer Katz**

- **Redis Results Backend**: The result_backend setting now supports rediss:// URIs

  Contributed by **James Remeika**

- **Canvas** Keyword arguments are passed to tasks in chain as expected.

  Contributed by [@tothegump](#)

- **Django** Fix a regression causing Celery to crash when using Django.

  Contributed by **Jonas Haag**

- **Canvas** Chain with one task now runs as expected.

  Contributed by [@tothegump](#)

- **Kombu** Celery 4.2 now requires Kombu 4.2 or better.

  Contributed by **Omer Katz & Asif Saifuddin Auvi**

- *GreenletExit* is not in *__all__* in greenlet.py which can not be imported by Python 3.6.

  The import was adjusted to work on Python 3.6 as well.

  Contributed by **Hsiaoming Yang**

- Fixed a regression that occurred during the development of Celery 4.2 which caused *celery report* to crash when Django is installed.

  Contributed by **Josue Balandrano Coronel**

- Matched the behavior of *GroupResult.as_tuple()* to that of *AsyncResult.as_tuple()*.

  The group's parent is now serialized correctly.

  Contributed by **Josue Balandrano Coronel**

- Use Redis coercion mechanism for converting URI query parameters.

  Contributed by **Justin Patrin**

- Fixed the representation of *GroupResult*.

  The dependency graph is now presented correctly.

  Contributed by **Josue Balandrano Coronel**

Documentation, CI, Installation and Tests fixes:

- **Sammie S. Taunton**
- **Dan Wilson**
- [@pachewise](#)
- **Sergi Almacellas Abellana**
- **Omer Katz**
- **Alex Zaitsev**
- **Leo Singer**
- **Rachel Johnson**
- **Jon Dufresne**

- **Samuel Dion-Girardeau**

- **Ryan Guest**

- **Huang Huang**

- **Geoffrey Bauduin**

- **Andrew Wong**

- **Mads Jensen**

- **Jackie Leng**

- **Harry Moreno**

- @michael-k

- **Nicolas Mota**

- **Armenak Baburyan**

- **Patrick Zhang**

- @anentropic

- @jairojair

- **Ben Welsh**

- **Michael Peake**

- **Fengyuan Chen**

- @arpanshah29

- **Xavier Hardy**

- **Shitikanth**

- **Igor Kasianov**

- **John Arnold**

- @dmollerm

- **Robert Knight**

- **Asif Saifuddin Auvi**

- **Eduardo Ramírez**

- **Kamil Breguła**

- **Juan Gutierrez**

3.13.3 What's new in Celery 4.1 (latentcall)

Author Omer Katz (`omer.drow at gmail.com`)

Change history

What's new documents describe the changes in major versions, we also have a *Change history* that lists the changes in bugfix releases (0.0.x), while older series are archived under the *History* section.

Celery is a simple, flexible, and reliable distributed system to process vast amounts of messages, while providing operations with the tools required to maintain such a system.

It's a task queue with focus on real-time processing, while also supporting task scheduling.

Celery has a large and diverse community of users and contributors, you should come join us *on IRC* or *our mailing-list*.

To read more about Celery you should go read the *introduction*.

While this version is backward compatible with previous versions it's important that you read the following section.

This version is officially supported on CPython 2.7, 3.4, 3.5 & 3.6 and is also supported on PyPy.

Table of Contents

Make sure you read the important notes before upgrading to this version.

Preface

The 4.1.0 release continues to improve our efforts to provide you with the best task execution platform for Python.

This release is mainly a bug fix release, ironing out some issues and regressions found in Celery 4.0.0.

We added official support for Python 3.6 and PyPy 5.8.0.

This is the first time we release without Ask Solem as an active contributor. We'd like to thank him for his hard work in creating and maintaining Celery over the years.

Since Ask Solem was not involved there were a few kinks in the release process which we promise to resolve in the next release. This document was missing when we did release Celery 4.1.0. Also, we did not update the release codename as we should have. We apologize for the inconvenience.

For the time being, I, Omer Katz will be the release manager.

Thank you for your support!

— *Omer Katz*

Wall of Contributors

Acey <huiwang.e@gmail.com> Acey9 <huiwang.e@gmail.com> Alan Hamlett <alanhamlett@users.noreply.github.com> Alan Justino da Silva <alan.justino@yahoo.com.br> Alejandro Pernin <ale.pernin@gmail.com> Alli <alzeih@users.noreply.github.com> Andreas Pelme <andreas@pelme.se> Andrew de Quincey <adq@lidskialf.net> Anthony Lukach <anthonylukach@gmail.com> Arcadiy Ivanov <arcadiy@ivanov.biz> Arnaud Rocher <caillemajor@users.noreply.github.com> Arthur Vigil <ahvigil@mail.sfsu.edu> Asif Saifuddin Auvi <auvipy@users.noreply.github.com> Ask Solem <ask@celeryproject.org> BLAGA Razvan-Paul <razvan.paul.blaga@gmail.com> Brendan MacDonell <macdonellba@gmail.com> Brian Luan <jznight@gmail.com> Brian May <brian@linuxpenguins.xyz> Bruno Alla <browniebroke@users.noreply.github.com> Chris Kuehl <chris@techxonline.net> Christian <github@penpal4u.net> Christopher Hoskin <mans0954@users.noreply.github.com> Daniel Hahler <github@thequod.de> Daniel Huang <dxhuang@gmail.com> Derek Harland <donkopotamus@users.noreply.github.com> Dmytro Petruk <bavaria95@gmail.com> Ed Morley <edmorley@users.noreply.github.com> Eric Poelke <epoelke@gmail.com> Felipe <fcoelho@users.noreply.github.com> François Voron <fvoron@gmail.com> GDR! <gdr@gdr.name> George Psarakis <giwrgos.psarakis@gmail.com> J Alan Brogan <jalanb@users.noreply.github.com> James Michael DuPont <JamesMikeDuPont@gmail.com> Jamie Alessio <jamie@stoic.net> Javier Domingo Cansino <javierdo1@gmail.com> Jay McGrath <jaymcgrath@users.noreply.github.com> Jian Yu <askingyj@gmail.com> Joey Wilhelm <tarkatronic@gmail.com> Jon Dufresne <jon.dufresne@gmail.com> Kalle Bronsen <bronsen@nrrd.de> Kirill Romanov <djaler1@gmail.com> Laurent Peuch <cortex@worlddomination.be> Luke Plant <L.Plant.98@cantab.net> Marat Sharafutdinov <decaz89@gmail.com> Marc Gibbons <marc_gibbons@rogers.com> Marc Hörsken <mback2k@users.noreply.github.com> Michael <michael-k@users.noreply.github.com> Michael Howitz <mh@gocept.com> Michal Kuffa <beezz@users.noreply.github.com> Mike Chen <yi.chen.it@gmail.com> Mike Helmick <michaelhelmick@users.noreply.github.com> Morgan Doocy <morgan@doocy.net> Moussa Taifi <moutai10@gmail.com> Omer Katz <omer.drow@gmail.com> Patrick Cloke <clokep@users.noreply.github.com> Peter Bittner <django@bittner.it> Preston Moore <prestonkmoore@gmail.com> Primož Kerin <kerin.primoz@gmail.com> Pysaoke <pysaoke@gmail.com> Rick Wargo <rickwargo@users.noreply.github.com> Rico Moorman <rico.moorman@gmail.com> Roman Sichny <roman@sichnyi.com> Ross Patterson <me@rpatterson.net> Ryan Hiebert <ryan@ryanhiebert.com> Rémi Marenco <remi.marenco@gmail.com> Salvatore Rinchiera <srinchiera@college.harvard.edu> Samuel Dion-Girardeau <samuel.diongirardeau@gmail.com> Sergey Fursov <GeyseR85@gmail.com> Simon Legner <Simon.Legner@gmail.com> Simon Schmidt <schmidt.simon@gmail.com> Slam <3lnc.slam@gmail.com> Static <staticfox@staticfox.net> Steffen Allner <sa@gocept.com> Steven <rh0dium@users.noreply.github.com> Steven Johns <duoi@users.noreply.github.com> Tamer Sherif <tamer.sherif@flyingelephantlab.com> Tao Qingyun <845767657@qq.com> Tayfun Sen <totayfun@gmail.com> Taylor C. Richberger <taywee@gmx.com> Thierry RAMORASOAVINA <thierry.ramorasoavina@orange.com> Tom 'Biwaa' Riat <riat.tom@gmail.com> Viktor Holmqvist <viktorholmqvist@gmail.com> Viraj <vnavkal0@gmail.com> Vivek Anand <vivekanand1101@users.noreply.github.com> Will <paradox41@users.noreply.github.com> Wojciech Zywno <w.zywno@gmail.com> Yoichi NAKAYAMA <yoichi.nakayama@gmail.com> YuLun Shih <shih@yulun.me> Yuhannaa <yuhannaa@gmail.com> abhinav nilaratna <anilaratna2@bloomberg.net> aydin <adigeaydin@gmail.com> csfeathers <csfeathers@users.noreply.github.com> georgepsarakis <giwrgos.psarakis@gmail.com> orf <tom@tomforb.es> shalev67 <shalev67@gmail.com> sww <sww@users.noreply.github.com> tnir <tnir@users.noreply.github.com> (Sean Ho) <h1x2y3awalm@gmail.com>

Note: This wall was automatically generated from git history, so sadly it doesn't not include the people who help with more important things like answering mailing-list questions.

Important Notes

Added support for Python 3.6 & PyPy 5.8.0

We now run our unit test suite and integration test suite on Python 3.6.x and PyPy 5.8.0.

We expect newer versions of PyPy to work but unfortunately we do not have the resources to test PyPy with those versions.

The supported Python Versions are:

- CPython 2.7
- CPython 3.4
- CPython 3.5
- CPython 3.6
- PyPy 5.8 (`pypy2`)

News

Result Backends

New DynamoDB Results Backend

We added a new results backend for those of you who are using DynamoDB.

If you are interested in using this results backend, refer to *AWS DynamoDB backend settings* for more information.

Elasticsearch

The Elasticsearch results backend is now more robust and configurable.

See *Elasticsearch backend settings* for more information about the new configuration options.

Redis

The Redis results backend can now use TLS to encrypt the communication with the Redis database server.

See *Redis backend settings*.

MongoDB

The MongoDB results backend can now handle binary-encoded task results.

This was a regression from 4.0.0 which resulted in a problem using serializers such as MsgPack or Pickle in conjunction with the MongoDB results backend.

Periodic Tasks

The task schedule now updates automatically when new tasks are added. Now if you use the Django database scheduler, you can add and remove tasks from the schedule without restarting Celery beat.

Tasks

The `disable_sync_subtasks` argument was added to allow users to override disabling synchronous subtasks.
See *Avoid launching synchronous subtasks*

Canvas

Multiple bugs were resolved resulting in a much smoother experience when using Canvas.

3.13.4 Change history

4.1.1

release-date 2018-05-21 12:48 PM PST

release-by Omer Katz

Important: Please upgrade as soon as possible or pin Kombu to 4.1.0.

- **Breaking Change**: The module *async* in Kombu changed to *asynchronous*.

Contributed by **Omer Katz & Asif Saifuddin Auvi**

4.1.0

release-date 2017-07-25 00:00 PM PST

release-by Omer Katz

- **Configuration**: CELERY_SEND_EVENTS instead of CELERYD_SEND_EVENTS for 3.1.x compatibility (#3997)

Contributed by **abhinav nilaratna**.

- **App**: Restore behavior so Broadcast queues work. (#3934)

Contributed by **Patrick Cloke**.

- **Sphinx**: Make appstr use standard format (#4134) (#4139)

Contributed by **Preston Moore**.

- **App**: Make id, name always accessible from logging.Formatter via extra (#3994)

Contributed by **Yoichi NAKAYAMA**.

- **Worker**: Add worker_shutting_down signal (#3998)

Contributed by **Daniel Huang**.

- **PyPy**: Support PyPy version 5.8.0 (#4128)

Contributed by **Omer Katz**.

- **Results**: Elasticsearch: Fix serializing keys (#3924)

Contributed by @staticfox.

- **Canvas**: Deserialize all tasks in a chain (#4015)

  Contributed by @fcoelho.

- **Systemd**: Recover loglevel for ExecStart in systemd config (#4023)

  Contributed by **Yoichi NAKAYAMA**.

- **Sphinx**: Use the Sphinx add_directive_to_domain API. (#4037)

  Contributed by **Patrick Cloke**.

- **App**: Pass properties to before_task_publish signal (#4035)

  Contributed by **Javier Domingo Cansino**.

- **Results**: Add SSL option for Redis backends (#3831)

  Contributed by **Chris Kuehl**.

- **Beat**: celery.schedule.crontab: fix reduce (#3826) (#3827)

  Contributed by **Taylor C. Richberger**.

- **State**: Fix celery issues when using flower REST API

  Contributed by **Thierry RAMORASOAVINA**.

- **Results**: Elasticsearch: Fix serializing document id.

  Contributed by **Acey9**.

- **Beat**: Make shallow copy of schedules dictionary

  Contributed by **Brian May**.

- **Beat**: Populate heap when periodic tasks are changed

  Contributed by **Wojciech Zywno**.

- **Task**: Allow class methods to define tasks (#3952)

  Contributed by **georgepsarakis**.

- **Platforms**: Always return boolean value when checking if signal is supported (#3962).

  Contributed by **Jian Yu**.

- **Canvas**: Avoid duplicating chains in chords (#3779)

  Contributed by **Ryan Hiebert**.

- **Canvas**: Lookup task only if list has items (#3847)

  Contributed by **Marc Gibbons**.

- **Results**: Allow unicode message for exception raised in task (#3903)

  Contributed by **George Psarakis**.

- **Python3**: Support for Python 3.6 (#3904, #3903, #3736)

  Contributed by **Jon Dufresne**, **George Psarakis**, **Asif Saifuddin Auvi**, **Omer Katz**.

- **App**: Fix retried tasks with expirations (#3790)

  Contributed by **Brendan MacDonell**.

-
  - Fixes items format route in docs (#3875)

  Contributed by **Slam**.

- **Utils**: Fix maybe_make_aware (#3850)

  Contributed by **Taylor C. Richberger**.

- **Task**: Fix task ETA issues when timezone is defined in configuration (#3867)

  Contributed by **George Psarakis**.

- **Concurrency**: Consumer does not shutdown properly when embedded in gevent application (#3746)

  Contributed by **Arcadiy Ivanov**.

- **Canvas**: Fix #3725: Task replaced with group does not complete (#3731)

  Contributed by **Morgan Doocy**.

- **Task**: Correct order in chains with replaced tasks (#3730)

  Contributed by **Morgan Doocy**.

- **Result**: Enable synchronous execution of sub-tasks (#3696)

  Contributed by **shalev67**.

- **Task**: Fix request context for blocking task apply (added hostname) (#3716)

  Contributed by **Marat Sharafutdinov**.

- **Utils**: Fix task argument handling (#3678) (#3693)

  Contributed by **Roman Sichny**.

- **Beat**: Provide a transparent method to update the Scheduler heap (#3721)

  Contributed by **Alejandro Pernin**.

- **Beat**: Specify default value for pidfile option of celery beat. (#3722)

  Contributed by **Arnaud Rocher**.

- **Results**: Elasticsearch: Stop generating a new field every time when a new result is being put (#3708)

  Contributed by **Mike Chen**.

- **Requirements**

  - Now depends on Kombu 4.1.0.

- **Results**: Elasticsearch now reuses fields when new results are added.

    Contributed by **Mike Chen**.

- **Results**: Fixed MongoDB integration when using binary encodings (Issue #3575).

    Contributed by **Andrew de Quincey**.

- **Worker**: Making missing `*args` and `**kwargs` in Task protocol 1 return empty value in protocol 2 (Issue #3687).

    Contributed by **Roman Sichny**.

- **App**: Fixed `TypeError` in AMQP when using deprecated signal (Issue #3707).

    Contributed by @michael-k.

- **Beat**: Added a transparent method to update the scheduler heap.

    Contributed by **Alejandro Pernin**.

- **Task**: Fixed handling of tasks with keyword arguments on Python 3 (Issue #3657).

Contributed by **Roman Sichny**.

- **Task**: Fixed request context for blocking task apply by adding missing hostname attribute.

    Contributed by **Marat Sharafutdinov**.

- **Task**: Added option to run subtasks synchronously with `disable_sync_subtasks` argument.

    Contributed by @shalev67.

- **App**: Fixed chaining of replaced tasks (Issue #3726).

    Contributed by **Morgan Doocy**.

- **Canvas**: Fixed bug where replaced tasks with groups were not completing (Issue #3725).

    Contributed by **Morgan Doocy**.

- **Worker**: Fixed problem where consumer does not shutdown properly when embedded in a gevent application (Issue #3745).

    Contributed by **Arcadiy Ivanov**.

- **Results**: Added support for using AWS DynamoDB as a result backend (#3736).

    Contributed by **George Psarakis**.

- **Testing**: Added caching on pip installs.

    Contributed by @orf.

- **Worker**: Prevent consuming queue before ready on startup (Issue #3620).

    Contributed by **Alan Hamlett**.

- **App**: Fixed task ETA issues when timezone is defined in configuration (Issue #3753).

    Contributed by **George Psarakis**.

- **Utils**: `maybe_make_aware` should not modify datetime when it is already timezone-aware (Issue #3849).

    Contributed by **Taylor C. Richberger**.

- **App**: Fixed retrying tasks with expirations (Issue #3734).

    Contributed by **Brendan MacDonell**.

- **Results**: Allow unicode message for exceptions raised in task (Issue #3858).

    Contributed by @staticfox.

- **Canvas**: Fixed `IndexError` raised when chord has an empty header.

    Contributed by **Marc Gibbons**.

- **Canvas**: Avoid duplicating chains in chords (Issue #3771).

    Contributed by **Ryan Hiebert** and **George Psarakis**.

- **Utils**: Allow class methods to define tasks (Issue #3863).

    Contributed by **George Psarakis**.

- **Beat**: Populate heap when periodic tasks are changed.

    Contributed by @wzywno and **Brian May**.

- **Results**: Added support for Elasticsearch backend options settings.

    Contributed by @Acey9.

- **Events**: Ensure `Task.as_dict()` works when not all information about task is available.

    Contributed by @tramora.

- **Schedules**: Fixed pickled crontab schedules to restore properly (Issue #3826).

    Contributed by **Taylor C. Richberger**.

- **Results**: Added SSL option for redis backends (Issue #3830).

    Contributed by **Chris Kuehl**.

- Documentation and examples improvements by:

    – **Bruno Alla**

    – **Jamie Alessio**

    – **Vivek Anand**

    – **Peter Bittner**

    – **Kalle Bronsen**

    – **Jon Dufresne**

    – **James Michael DuPont**

    – **Sergey Fursov**

    – **Samuel Dion-Girardeau**

    – **Daniel Hahler**

    – **Mike Helmick**

    – **Marc Hörsken**

    – **Christopher Hoskin**

    – **Daniel Huang**

    – **Primož Kerin**

    – **Michal Kuffa**

    – **Simon Legner**

    – **Anthony Lukach**

    – **Ed Morley**

    – **Jay McGrath**

    – **Rico Moorman**

    – **Viraj Navkal**

    – **Ross Patterson**

    – **Dmytro Petruk**

    – **Luke Plant**

    – **Eric Poelke**

    – **Salvatore Rinchiera**

    – **Arnaud Rocher**

    – **Kirill Romanov**

- **– Simon Schmidt**
- **– Tamer Sherif**
- **– YuLun Shih**
- **– Ask Solem**
- **– Tom 'Biwaa' Riat**
- **– Arthur Vigil**
- **– Joey Wilhelm**
- **– Jian Yu**
- **– YuLun Shih**
- **– Arthur Vigil**
- **– Joey Wilhelm**
- – @baixuexue123
- – @bronsen
- – @michael-k
- – @orf
- – @3lnc

3.13.5 What's new in Celery 4.0 (latentcall)

> **Author** Ask Solem (`ask at celeryproject.org`)

Change history

What's new documents describe the changes in major versions, we also have a *Change history* that lists the changes in bugfix releases (0.0.x), while older series are archived under the *History* section.

Celery is a simple, flexible, and reliable distributed system to process vast amounts of messages, while providing operations with the tools required to maintain such a system.

It's a task queue with focus on real-time processing, while also supporting task scheduling.

Celery has a large and diverse community of users and contributors, you should come join us *on IRC* or *our mailing-list*.

To read more about Celery you should go read the *introduction*.

While this version is backward compatible with previous versions it's important that you read the following section.

This version is officially supported on CPython 2.7, 3.4, and 3.5. and also supported on PyPy.

Table of Contents

Make sure you read the important notes before upgrading to this version.

Preface

Welcome to Celery 4!

This is a massive release with over two years of changes. Not only does it come with many new features, but it also fixes a massive list of bugs, so in many ways you could call it our "Snow Leopard" release.

The next major version of Celery will support Python 3.5 only, where we are planning to take advantage of the new asyncio library.

This release would not have been possible without the support of my employer, Robinhood (we're hiring!).

> • Ask Solem

Dedicated to Sebastian "Zeb" Bjørnerud (RIP), with special thanks to Ty Wilkins, for designing our new logo, all the contributors who help make this happen, and my colleagues at Robinhood.

Wall of Contributors

Aaron McMillin, Adam Chainz, Adam Renberg, Adriano Martins de Jesus, Adrien Guinet, Ahmet Demir, Aitor Gómez-Goiri, Alan Justino, Albert Wang, Alex Koshelev, Alex Rattray, Alex Williams, Alexander Koshelev, Alexander Lebedev, Alexander Oblovatniy, Alexey Kotlyarov, Ali Bozorgkhan, Alice Zoë Bevan–McGregor, Allard Hoeve, Alman One, Amir Rustamzadeh, Andrea Rabbaglietti, Andrea Rosa, Andrei Fokau, Andrew Rodionoff, Andrew Stewart, Andriy Yurchuk, Aneil Mallavarapu, Areski Belaid, Armenak Baburyan, Arthur Vuillard, Artyom Koval, Asif Saifuddin Auvi, Ask Solem, Balthazar Rouberol, Batiste Bieler, Berker Peksag, Bert Vanderbauwhede, Brendan Smithyman, Brian Bouterse, Bryce Groff, Cameron Will, ChangBo Guo, Chris Clark, Chris Duryee, Chris Erway, Chris Harris, Chris Martin, Chillar Anand, Colin McIntosh, Conrad Kramer, Corey Farwell, Craig Jellick, Cullen Rhodes, Dallas Marlow, Daniel Devine, Daniel Wallace, Danilo Bargen, Davanum Srinivas, Dave Smith, David Baumgold, David Harrigan, David Pravec, Dennis Brakhane, Derek Anderson, Dmitry Dygalo, Dmitry Malinovsky, Dongweiming, Dudás Ádám, Dustin J. Mitchell, Ed Morley, Edward Betts, Éloi Rivard, Emmanuel Cazenave, Fahad Siddiqui, Fatih Sucu, Feanil Patel, Federico Ficarelli, Felix Schwarz, Felix Yan, Fernando Rocha, Flavio Grossi, Frantisek Holop, Gao Jiangmiao, George Whewell, Gerald Manipon, Gilles Dartiguelongue, Gino Ledesma, Greg Wilbur, Guillaume Seguin, Hank John, Hogni Gylfason, Ilya Georgievsky, Ionel Cristian Mărieș, Ivan Larin, James Pulec, Jared Lewis, Jason Veatch, Jasper Bryant-Greene, Jeff Widman, Jeremy Tillman, Jeremy Zafran, Jocelyn Delalande, Joe Jevnik, Joe Sanford, John Anderson, John Barham, John Kirkham, John Whitlock, Jonathan Vanasco, Joshua Harlow, João Ricardo, Juan Carlos Ferrer, Juan Rossi, Justin Patrin, Kai Groner, Kevin Harvey, Kevin Richardson, Komu Wairagu, Konstantinos Koukopoulos, Kouhei Maeda, Kracekumar Ramaraju, Krzysztof Bujniewicz, Latitia M. Haskins, Len Buckens, Lev Berman, lidongming, Lorenzo Mancini, Lucas Wiman, Luke Pomfrey, Luyun Xie, Maciej Obuchowski, Manuel Kaufmann, Marat Sharafutdinov, Marc Sibson, Marcio Ribeiro, Marin Atanasov Nikolov, Mathieu Fenniak, Mark Parncutt, Mauro Rocco, Maxime Beauchemin, Maxime Vdb, Mher Movsisyan, Michael Aquilina, Michael Duane Mooring, Michael Permana, Mickaël Penhard, Mike Attwood, Mitchel Humpherys, Mohamed Abouelsaoud, Morris Tweed, Morton Fox, Môshe van der Sterre, Nat Williams, Nathan Van Gheem, Nicolas Unravel, Nik Nyby, Omer Katz, Omer Korner, Ori Hoch, Paul Pearce, Paulo Bu, Pavlo Kapyshin, Philip Garnero, Pierre Fersing, Piotr Kilczuk, Piotr Maślanka, Quentin Pradet, Radek Czajka, Raghuram Srinivasan, Randy Barlow, Raphael Michel, Rémy Léone, Robert Coup, Robert Kolba, Rockallite Wulf, Rodolfo Carvalho, Roger Hu, Romuald Brunet, Rongze Zhu, Ross Deane, Ryan Luckie, Rémy Greinhofer, Samuel Giffard, Samuel Jaillet, Sergey Azovskov, Sergey Tikhonov, Seungha Kim, Simon Peeters, Spencer E. Olson, Srinivas Garlapati, Stephen Milner, Steve Peak, Steven Sklar, Stuart Axon, Sukrit Khera, Tadej Janež, Taha Jahangir, Takeshi Kanemoto, Tayfun Sen, Tewfik Sadaoui, Thomas French, Thomas Grainger, Tomas Machalek, Tobias Schottdorf, Tocho Tochev, Valentyn Klindukh, Vic Kumar, Vladimir Bolshakov, Vladimir Gorbunov, Wayne Chang, Wieland Hoffmann, Wido den Hollander, Wil Langford, Will Thompson, William King, Yury Selivanov, Vytis Banaitis, Zoran Pavlovic, Xin Li, , @allenling, @alzeih, @bastb, @bee-keeper, @ffeast, @firefly4268, @flyingfoxlee, @gdw2, @gitaarik, @hankjin, @lvh, @m-vdb, @kindule, @mdk:, @michael-k, @mozillazg, @nokrik, @ocean1, @orlo666, @raducc, @wanglei, @worldexception, @xBeAsTx.

Note: This wall was automatically generated from git history, so sadly it doesn't not include the people who help with more important things like answering mailing-list questions.

Upgrading from Celery 3.1

Step 1: Upgrade to Celery 3.1.25

If you haven't already, the first step is to upgrade to Celery 3.1.25.

This version adds forward compatibility to the new message protocol, so that you can incrementally upgrade from 3.1 to 4.0.

Deploy the workers first by upgrading to 3.1.25, this means these workers can process messages sent by clients using both 3.1 and 4.0.

After the workers are upgraded you can upgrade the clients (e.g. web servers).

Step 2: Update your configuration with the new setting names

This version radically changes the configuration setting names, to be more consistent.

The changes are fully backwards compatible, so you have the option to wait until the old setting names are deprecated, but to ease the transition we have included a command-line utility that rewrites your settings automatically.

See *Lowercase setting names* for more information.

Step 3: Read the important notes in this document

Make sure you are not affected by any of the important upgrade notes mentioned in the following section.

An especially important note is that Celery now checks the arguments you send to a task by matching it to the signature (*Task argument checking*).

Step 4: Upgrade to Celery 4.0

At this point you can upgrade your workers and clients with the new version.

Important Notes

Dropped support for Python 2.6

Celery now requires Python 2.7 or later, and also drops support for Python 3.3 so supported versions are:

- CPython 2.7
- CPython 3.4
- CPython 3.5
- PyPy 5.4 (`pypy2`)
- PyPy 5.5-alpha (`pypy3`)

Last major version to support Python 2

Starting from Celery 5.0 only Python 3.5+ will be supported.

To make sure you're not affected by this change you should pin the Celery version in your requirements file, either to a specific version: `celery==4.0.0`, or a range: `celery>=4.0,<5.0`.

Dropping support for Python 2 will enable us to remove massive amounts of compatibility code, and going with Python 3.5 allows us to take advantage of typing, async/await, asyncio, and similar concepts there's no alternative for in older versions.

Celery 4.x will continue to work on Python 2.7, 3.4, 3.5; just as Celery 3.x still works on Python 2.6.

Django support

Celery 4.x requires Django 1.8 or later, but we really recommend using at least Django 1.9 for the new `transaction.on_commit` feature.

A common problem when calling tasks from Django is when the task is related to a model change, and you wish to cancel the task if the transaction is rolled back, or ensure the task is only executed after the changes have been written to the database.

`transaction.atomic` enables you to solve this problem by adding the task as a callback to be called only when the transaction is committed.

Example usage:

```python
from functools import partial
from django.db import transaction

from .models import Article, Log
from .tasks import send_article_created_notification

def create_article(request):
    with transaction.atomic():
        article = Article.objects.create(**request.POST)
        # send this task only if the rest of the transaction succeeds.
        transaction.on_commit(partial(
            send_article_created_notification.delay, article_id=article.pk))
        Log.objects.create(type=Log.ARTICLE_CREATED, object_pk=article.pk)
```

Removed features

- Microsoft Windows is no longer supported.

  The test suite is passing, and Celery seems to be working with Windows, but we make no guarantees as we are unable to diagnose issues on this platform. If you are a company requiring support on this platform, please get in touch.

- Jython is no longer supported.

Features removed for simplicity

- Webhook task machinery (`celery.task.http`) has been removed.

  Nowadays it's easy to use the requests module to write webhook tasks manually. We would love to use requests but we are simply unable to as there's a very vocal 'anti-dependency' mob in the Python community

  If you need backwards compatibility you can simply copy + paste the 3.1 version of the module and make sure it's imported by the worker: https://github.com/celery/celery/blob/3.1/celery/task/http.py

- Tasks no longer sends error emails.

  This also removes support for `app.mail_admins`, and any functionality related to sending emails.

- `celery.contrib.batches` has been removed.

  This was an experimental feature, so not covered by our deprecation timeline guarantee.

  You can copy and pase the existing batches code for use within your projects: https://github.com/celery/celery/blob/3.1/celery/contrib/batches.py

Features removed for lack of funding

We announced with the 3.1 release that some transports were moved to experimental status, and that there'd be no official support for the transports.

As this subtle hint for the need of funding failed we've removed them completely, breaking backwards compatibility.

- Using the Django ORM as a broker is no longer supported.

  You can still use the Django ORM as a result backend: see *django-celery-results - Using the Django ORM/Cache as a result backend* section for more information.

- Using SQLAlchemy as a broker is no longer supported.

  You can still use SQLAlchemy as a result backend.

- Using CouchDB as a broker is no longer supported.

  You can still use CouchDB as a result backend.

- Using IronMQ as a broker is no longer supported.

- Using Beanstalk as a broker is no longer supported.

In addition some features have been removed completely so that attempting to use them will raise an exception:

- The `--autoreload` feature has been removed.

  This was an experimental feature, and not covered by our deprecation timeline guarantee. The flag is removed completely so the worker will crash at startup when present. Luckily this flag isn't used in production systems.

- The experimental `threads` pool is no longer supported and has been removed.

- The `force_execv` feature is no longer supported.

  The `celery worker` command now ignores the `--no-execv`, `--force-execv`, and the `CELERYD_FORCE_EXECV` setting.

  This flag will be removed completely in 5.0 and the worker will raise an error.

- The old legacy "amqp" result backend has been deprecated, and will be removed in Celery 5.0.

  Please use the `rpc` result backend for RPC-style calls, and a persistent result backend for multi-consumer results.

We think most of these can be fixed without considerable effort, so if you're interested in getting any of these features back, please get in touch.

Now to the good news. . .

New Task Message Protocol

This version introduces a brand new task message protocol, the first major change to the protocol since the beginning of the project.

The new protocol is enabled by default in this version and since the new version isn't backwards compatible you have to be careful when upgrading.

The 3.1.25 version was released to add compatibility with the new protocol so the easiest way to upgrade is to upgrade to that version first, then upgrade to 4.0 in a second deployment.

If you wish to keep using the old protocol you may also configure the protocol version number used:

```
app = Celery()
app.conf.task_protocol = 1
```

Read more about the features available in the new protocol in the news section found later in this document.

Lowercase setting names

In the pursuit of beauty all settings are now renamed to be in all lowercase and some setting names have been renamed for consistency.

This change is fully backwards compatible so you can still use the uppercase setting names, but we would like you to upgrade as soon as possible and you can do this automatically using the **celery upgrade settings** command:

```
$ celery upgrade settings proj/settings.py
```

This command will modify your module in-place to use the new lower-case names (if you want uppercase with a "CELERY" prefix see block below), and save a backup in `proj/settings.py.orig`.

For Django users and others who want to keep uppercase names

If you're loading Celery configuration from the Django settings module then you'll want to keep using the uppercase names.

You also want to use a CELERY_ prefix so that no Celery settings collide with Django settings used by other apps.

To do this, you'll first need to convert your settings file to use the new consistent naming scheme, and add the prefix to all Celery related settings:

```
$ celery upgrade settings proj/settings.py --django
```

After upgrading the settings file, you need to set the prefix explicitly in your `proj/celery.py` module:

```
app.config_from_object('django.conf:settings', namespace='CELERY')
```

You can find the most up to date Django Celery integration example here: *First steps with Django*.

Note: This will also add a prefix to settings that didn't previously have one, for example BROKER_URL should be written CELERY_BROKER_URL with a namespace of CELERY CELERY_BROKER_URL.

Luckily you don't have to manually change the files, as the **celery upgrade settings --django** program should do the right thing.

The loader will try to detect if your configuration is using the new format, and act accordingly, but this also means you're not allowed to mix and match new and old setting names, that's unless you provide a value for both alternatives.

The major difference between previous versions, apart from the lower case names, are the renaming of some prefixes, like `celerybeat_` to `beat_`, `celeryd_` to `worker_`.

The `celery_` prefix has also been removed, and task related settings from this name-space is now prefixed by `task_`, worker related settings with `worker_`.

Apart from this most of the settings will be the same in lowercase, apart from a few special ones:

Setting name	Replace with
CELERY_MAX_CACHED_RESULTS	*result_cache_max*
CELERY_MESSAGE_COMPRESSION	*result_compression*/*task_compression*.
CELERY_TASK_RESULT_EXPIRES	*result_expires*
CELERY_RESULT_DBURI	*result_backend*
CELERY_RESULT_ENGINE_OPTIONS	*database_engine_options*
-*-_DB_SHORT_LIVED_SESSIONS	*database_short_lived_sessions*
CELERY_RESULT_DB_TABLE_NAMES	database_db_names
CELERY_ACKS_LATE	*task_acks_late*
CELERY_ALWAYS_EAGER	*task_always_eager*
CELERY_ANNOTATIONS	*task_annotations*
CELERY_MESSAGE_COMPRESSION	*task_compression*
CELERY_CREATE_MISSING_QUEUES	*task_create_missing_queues*
CELERY_DEFAULT_DELIVERY_MODE	*task_default_delivery_mode*
CELERY_DEFAULT_EXCHANGE	*task_default_exchange*
CELERY_DEFAULT_EXCHANGE_TYPE	*task_default_exchange_type*
CELERY_DEFAULT_QUEUE	*task_default_queue*
CELERY_DEFAULT_RATE_LIMIT	*task_default_rate_limit*
CELERY_DEFAULT_ROUTING_KEY	*task_default_routing_key*
-"-_EAGER_PROPAGATES_EXCEPTIONS	*task_eager_propagates*
CELERY_IGNORE_RESULT	*task_ignore_result*
CELERY_TASK_PUBLISH_RETRY	*task_publish_retry*
CELERY_TASK_PUBLISH_RETRY_POLICY	*task_publish_retry_policy*
CELERY_QUEUES	*task_queues*
CELERY_ROUTES	*task_routes*
CELERY_SEND_TASK_SENT_EVENT	*task_send_sent_event*
CELERY_TASK_SERIALIZER	*task_serializer*
CELERYD_TASK_SOFT_TIME_LIMIT	*task_soft_time_limit*
CELERYD_TASK_TIME_LIMIT	*task_time_limit*
CELERY_TRACK_STARTED	*task_track_started*
CELERY_DISABLE_RATE_LIMITS	*worker_disable_rate_limits*
CELERY_ENABLE_REMOTE_CONTROL	*worker_enable_remote_control*
CELERYD_SEND_EVENTS	*worker_send_task_events*

You can see a full table of the changes in *New lowercase settings*.

Json is now the default serializer

The time has finally come to end the reign of `pickle` as the default serialization mechanism, and json is the default serializer starting from this version.

This change was *announced with the release of Celery 3.1*.

If you're still depending on `pickle` being the default serializer, then you have to configure your app before upgrading to 4.0:

```
task_serializer = 'pickle'
result_serializer = 'pickle'
accept_content = {'pickle'}
```

The Json serializer now also supports some additional types:

- `datetime`, `time`, `date`

Converted to json text, in ISO-8601 format.

- Decimal

    Converted to json text.

- django.utils.functional.Promise

    Django only: Lazy strings used for translation etc., are evaluated and conversion to a json type is attempted.

- uuid.UUID

    Converted to json text.

You can also define a __json__ method on your custom classes to support JSON serialization (must return a json compatible type):

```python
class Person:
    first_name = None
    last_name = None
    address = None

    def __json__(self):
        return {
            'first_name': self.first_name,
            'last_name': self.last_name,
            'address': self.address,
        }
```

The Task base class no longer automatically register tasks

The *Task* class is no longer using a special meta-class that automatically registers the task in the task registry.

Instead this is now handled by the *app.task* decorators.

If you're still using class based tasks, then you need to register these manually:

```python
class CustomTask(Task):
    def run(self):
        print('running')
CustomTask = app.register_task(CustomTask())
```

The best practice is to use custom task classes only for overriding general behavior, and then using the task decorator to realize the task:

```python
@app.task(bind=True, base=CustomTask)
def custom(self):
    print('running')
```

This change also means that the abstract attribute of the task no longer has any effect.

Task argument checking

The arguments of the task are now verified when calling the task, even asynchronously:

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

>>> add.delay(8, 8)
<AsyncResult: f59d71ca-1549-43e0-be41-4e8821a83c0c>

>>> 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
```

Or if you would like to disable this completely for all tasks you can pass `strict_typing=False` when creating the app:

```
app = Celery(..., strict_typing=False)
```

Redis Events not backward compatible

The Redis `fanout_patterns` and `fanout_prefix` transport options are now enabled by default.

Workers/monitors without these flags enabled won't be able to see workers with this flag disabled. They can still execute tasks, but they cannot receive each others monitoring messages.

You can upgrade in a backward compatible manner by first configuring your 3.1 workers and monitors to enable the settings, before the final upgrade to 4.0:

```
BROKER_TRANSPORT_OPTIONS = {
    'fanout_patterns': True,
    'fanout_prefix': True,
}
```

Redis Priorities Reversed

Priority 0 is now lowest, 9 is highest.

This change was made to make priority support consistent with how it works in AMQP.

Contributed by **Alex Koshelev**.

Django: Auto-discover now supports Django app configurations

The `autodiscover_tasks()` function can now be called without arguments, and the Django handler will automatically find your installed apps:

```
app.autodiscover_tasks()
```

The Django integration *example in the documentation* has been updated to use the argument-less call.

This also ensures compatibility with the new, ehm, `AppConfig` stuff introduced in recent Django versions.

Worker direct queues no longer use auto-delete

Workers/clients running 4.0 will no longer be able to send worker direct messages to workers running older versions, and vice versa.

If you're relying on worker direct messages you should upgrade your 3.x workers and clients to use the new routing settings first, by replacing *celery.utils.worker_direct()* with this implementation:

```python
from kombu import Exchange, Queue

worker_direct_exchange = Exchange('C.dq2')

def worker_direct(hostname):
    return Queue(
        '{hostname}.dq2'.format(hostname),
        exchange=worker_direct_exchange,
        routing_key=hostname,
    )
```

This feature closed Issue #2492.

Old command-line programs removed

Installing Celery will no longer install the `celeryd`, `celerybeat` and `celeryd-multi` programs.

This was announced with the release of Celery 3.1, but you may still have scripts pointing to the old names, so make sure you update these to use the new umbrella command:

Program	New Status	Replacement
celeryd	**REMOVED**	**celery worker**
celerybeat	**REMOVED**	**celery beat**
celeryd-multi	**REMOVED**	**celery multi**

News

New protocol highlights

The new protocol fixes many problems with the old one, and enables some long-requested features:

- Most of the data are now sent as message headers, instead of being serialized with the message body.

    In version 1 of the protocol the worker always had to deserialize the message to be able to read task meta-data like the task id, name, etc. This also meant that the worker was forced to double-decode the data, first deserializing the message on receipt, serializing the message again to send to child process, then finally the child process deserializes the message again.

> Keeping the meta-data fields in the message headers means the worker doesn't actually have to decode the payload before delivering the task to the child process, and also that it's now possible for the worker to reroute a task written in a language different from Python to a different worker.

- A new `lang` message header can be used to specify the programming language the task is written in.

- Worker stores results for internal errors like `ContentDisallowed`, and other deserialization errors.

- Worker stores results and sends monitoring events for unregistered task errors.

- Worker calls callbacks/errbacks even when the result is sent by the parent process (e.g., `WorkerLostError` when a child process terminates, deserialization errors, unregistered tasks).

- A new `origin` header contains information about the process sending the task (worker node-name, or PID and host-name information).

- A new `shadow` header allows you to modify the task name used in logs.

  > This is useful for dispatch like patterns, like a task that calls any function using pickle (don't do this at home):

  ```python
  from celery import Task
  from celery.utils.imports import qualname

  class call_as_task(Task):

      def shadow_name(self, args, kwargs, options):
          return 'call_as_task:{0}'.format(qualname(args[0]))

      def run(self, fun, *args, **kwargs):
          return fun(*args, **kwargs)
  call_as_task = app.register_task(call_as_task())
  ```

- New `argsrepr` and `kwargsrepr` fields contain textual representations of the task arguments (possibly truncated) for use in logs, monitors, etc.

  > This means the worker doesn't have to deserialize the message payload to display the task arguments for informational purposes.

- Chains now use a dedicated `chain` field enabling support for chains of thousands and more tasks.

- New `parent_id` and `root_id` headers adds information about a tasks relationship with other tasks.

  - `parent_id` is the task id of the task that called this task

  - `root_id` is the first task in the work-flow.

  > These fields can be used to improve monitors like flower to group related messages together (like chains, groups, chords, complete work-flows, etc).

- `app.TaskProducer` replaced by *app.amqp.create_task_message()* and *app.amqp.send_task_message()*.

  > Dividing the responsibilities into creating and sending means that people who want to send messages using a Python AMQP client directly, don't have to implement the protocol.

  > The *app.amqp.create_task_message()* method calls either `app.amqp.as_task_v2()`, or `app.amqp.as_task_v1()` depending on the configured task protocol, and returns a special `task_message` tuple containing the headers, properties and body of the task message.

See also:

The new task protocol is documented in full here: *Version 2*.

Prefork Pool Improvements

Tasks now log from the child process

Logging of task success/failure now happens from the child process executing the task. As a result logging utilities, like Sentry can get full information about tasks, including variables in the traceback stack.

`-Ofair` is now the default scheduling strategy

To re-enable the default behavior in 3.1 use the `-Ofast` command-line option.

There's been lots of confusion about what the `-Ofair` command-line option does, and using the term "prefetch" in explanations have probably not helped given how confusing this terminology is in AMQP.

When a Celery worker using the prefork pool receives a task, it needs to delegate that task to a child process for execution.

The prefork pool has a configurable number of child processes (`--concurrency`) that can be used to execute tasks, and each child process uses pipes/sockets to communicate with the parent process:

- inqueue (pipe/socket): parent sends task to the child process
- outqueue (pipe/socket): child sends result/return value to the parent.

In Celery 3.1 the default scheduling mechanism was simply to send the task to the first `inqueue` that was writable, with some heuristics to make sure we round-robin between them to ensure each child process would receive the same amount of tasks.

This means that in the default scheduling strategy, a worker may send tasks to the same child process that is already executing a task. If that task is long running, it may block the waiting task for a long time. Even worse, hundreds of short-running tasks may be stuck behind a long running task even when there are child processes free to do work.

The `-Ofair` scheduling strategy was added to avoid this situation, and when enabled it adds the rule that no task should be sent to the a child process that is already executing a task.

The fair scheduling strategy may perform slightly worse if you have only short running tasks.

Limit child process resident memory size

You can now limit the maximum amount of memory allocated per prefork pool child process by setting the worker *--max-memory-per-child* option, or the *worker_max_memory_per_child* setting.

The limit is for RSS/resident memory size and is specified in kilobytes.

A child process having exceeded the limit will be terminated and replaced with a new process after the currently executing task returns.

See *Max memory per child setting* for more information.

Contributed by **Dave Smith**.

One log-file per child process

Init-scripts and **celery multi** now uses the *%I* log file format option (e.g., `/var/log/celery/%n%I.log`).

This change was necessary to ensure each child process has a separate log file after moving task logging to the child process, as multiple processes writing to the same log file can cause corruption.

You're encouraged to upgrade your init-scripts and **celery multi** arguments to use this new option.

Transports

RabbitMQ priority queue support

See *RabbitMQ Message Priorities* for more information.

Contributed by **Gerald Manipon**.

Configure broker URL for read/write separately

New *broker_read_url* and *broker_write_url* settings have been added so that separate broker URLs can be provided for connections used for consuming/publishing.

In addition to the configuration options, two new methods have been added the app API:

- `app.connection_for_read()`

- `app.connection_for_write()`

These should now be used in place of `app.connection()` to specify the intent of the required connection.

Note: Two connection pools are available: `app.pool` (read), and `app.producer_pool` (write). The latter doesn't actually give connections but full `kombu.Producer` instances.

```python
def publish_some_message(app, producer=None):
    with app.producer_or_acquire(producer) as producer:
        ...

def consume_messages(app, connection=None):
    with app.connection_or_acquire(connection) as connection:
        ...
```

RabbitMQ queue extensions support

Queue declarations can now set a message TTL and queue expiry time directly, by using the `message_ttl` and `expires` arguments

New arguments have been added to `Queue` that lets you directly and conveniently configure RabbitMQ queue extensions in queue declarations:

- `Queue(expires=20.0)`

    Set queue expiry time in float seconds.

    See `kombu.Queue.expires`.

- `Queue(message_ttl=30.0)`

    Set queue message time-to-live float seconds.

    See `kombu.Queue.message_ttl`.

- `Queue(max_length=1000)`

> > Set queue max length (number of messages) as int.
> >
> > See `kombu.Queue.max_length`.
>
> - `Queue(max_length_bytes=1000)`
>
>   > Set queue max length (message size total in bytes) as int.
>   >
>   > See `kombu.Queue.max_length_bytes`.
>
> - `Queue(max_priority=10)`
>
>   > Declare queue to be a priority queue that routes messages based on the `priority` field of the message.
>   >
>   > See `kombu.Queue.max_priority`.

Amazon SQS transport now officially supported

The SQS broker transport has been rewritten to use async I/O and as such joins RabbitMQ, Redis and QPid as officially supported transports.

The new implementation also takes advantage of long polling, and closes several issues related to using SQS as a broker.

This work was sponsored by Nextdoor.

Apache QPid transport now officially supported

Contributed by **Brian Bouterse**.

Redis: Support for Sentinel

You can point the connection to a list of sentinel URLs like:

```
sentinel://0.0.0.0:26379;sentinel://0.0.0.0:26380/...
```

where each sentinel is separated by a `;`. Multiple sentinels are handled by `kombu.Connection` constructor, and placed in the alternative list of servers to connect to in case of connection failure.

Contributed by **Sergey Azovskov**, and **Lorenzo Mancini**.

Tasks

Task Auto-retry Decorator

Writing custom retry handling for exception events is so common that we now have built-in support for it.

For this a new `autoretry_for` argument is now supported by the task decorators, where you can specify a tuple of exceptions to automatically retry for:

```python
from twitter.exceptions import FailWhaleError


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

See *Automatic retry for known exceptions* for more information.

Contributed by **Dmitry Malinovsky**.

`Task.replace` Improvements

- `self.replace(signature)` can now replace any task, chord or group, and the signature to replace with can be a chord, group or any other type of signature.

- No longer inherits the callbacks and errbacks of the existing task.

  If you replace a node in a tree, then you wouldn't expect the new node to inherit the children of the old node.

- `Task.replace_in_chord` has been removed, use `.replace` instead.

- If the replacement is a group, that group will be automatically converted to a chord, where the callback "accumulates" the results of the group tasks.

  A new built-in task (*celery.accumulate* was added for this purpose)

Contributed by **Steeve Morin**, and **Ask Solem**.

Remote Task Tracebacks

The new `task_remote_tracebacks` will make task tracebacks more useful by injecting the stack of the remote worker.

This feature requires the additional tblib library.

Contributed by **Ionel Cristian Mărieș**.

Handling task connection errors

Connection related errors occurring while sending a task is now re-raised as a `kombu.exceptions.OperationalError` error:

```
>>> try:
...     add.delay(2, 2)
... except add.OperationalError as exc:
...     print('Could not send task %r: %r' % (add, exc))
```

See *Connection Error Handling* for more information.

Gevent/Eventlet: Dedicated thread for consuming results

When using gevent, or eventlet there is now a single thread responsible for consuming events.

This means that if you have many calls retrieving results, there will be a dedicated thread for consuming them:

```
result = add.delay(2, 2)

# this call will delegate to the result consumer thread:
#   once the consumer thread has received the result this greenlet can
# continue.
value = result.get(timeout=3)
```

This makes performing RPC calls when using gevent/eventlet perform much better.

AsyncResult.then(on_success, on_error)

The AsyncResult API has been extended to support the `promise` protocol.

This currently only works with the RPC (amqp) and Redis result backends, but lets you attach callbacks to when tasks finish:

```python
import gevent.monkey
monkey.patch_all()

import time
from celery import Celery

app = Celery(broker='amqp://', backend='rpc')

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

def on_result_ready(result):
    print('Received result for id %r: %r' % (result.id, result.result,))

add.delay(2, 2).then(on_result_ready)

time.sleep(3)  # run gevent event loop for a while.
```

Demonstrated using gevent here, but really this is an API that's more useful in callback-based event loops like twisted, or tornado.

New Task Router API

The `task_routes` setting can now hold functions, and map routes now support glob patterns and regexes.

Instead of using router classes you can now simply define a function:

```python
def route_for_task(name, args, kwargs, options, task=None, **kwargs):
    from proj import tasks

    if name == tasks.add.name:
        return {'queue': 'hipri'}
```

If you don't need the arguments you can use start arguments, just make sure you always also accept star arguments so that we have the ability to add more features in the future:

```python
def route_for_task(name, *args, **kwargs):
    from proj import tasks
    if name == tasks.add.name:
        return {'queue': 'hipri', 'priority': 9}
```

Both the `options` argument and the new `task` keyword argument are new to the function-style routers, and will make it easier to write routers based on execution options, or properties of the task.

The optional `task` keyword argument won't be set if a task is called by name using `app.send_task()`.

For more examples, including using glob/regexes in routers please see `task_routes` and *Automatic routing*.

Canvas Refactor

The canvas/work-flow implementation have been heavily refactored to fix some long outstanding issues.

- Error callbacks can now take real exception and traceback instances (Issue #2538).

```python
>>> add.s(2, 2).on_error(log_error.s()).delay()
```

  Where `log_error` could be defined as:

```python
@app.task
def log_error(request, exc, traceback):
    with open(os.path.join('/var/errors', request.id), 'a') as fh:
        print('--\n\n{0} {1} {2}'.format(
            task_id, exc, traceback), file=fh)
```

  See *Canvas: Designing Work-flows* for more examples.

- `chain(a, b, c)` now works the same as `a | b | c`.

  This means chain may no longer return an instance of `chain`, instead it may optimize the workflow so that e.g. two groups chained together becomes one group.

- Now unrolls groups within groups into a single group (Issue #1509).

- chunks/map/starmap tasks now routes based on the target task

- chords and chains can now be immutable.

- Fixed bug where serialized signatures weren't converted back into signatures (Issue #2078)

  Fix contributed by **Ross Deane**.

- Fixed problem where chains and groups didn't work when using JSON serialization (Issue #2076).

  Fix contributed by **Ross Deane**.

- Creating a chord no longer results in multiple values for keyword argument 'task_id' (Issue #2225).

  Fix contributed by **Aneil Mallavarapu**.

- Fixed issue where the wrong result is returned when a chain contains a chord as the penultimate task.

  Fix contributed by **Aneil Mallavarapu**.

- Special case of `group(A.s() | group(B.s() | C.s()))` now works.

- Chain: Fixed bug with incorrect id set when a subtask is also a chain.

- `group | group` is now flattened into a single group (Issue #2573).

- Fixed issue where `group | task` wasn't upgrading correctly to chord (Issue #2922).

- Chords now properly sets `result.parent` links.

- `chunks`/`map`/`starmap` are now routed based on the target task.

- **Signature.link now works when argument is scalar (not a list)** (Issue #2019).

- `group()` now properly forwards keyword arguments (Issue #3426).

  Fix contributed by **Samuel Giffard**.

- A `chord` where the header group only consists of a single task is now turned into a simple chain.

- Passing a `link` argument to `group.apply_async()` now raises an error (Issue #3508).

- `chord | sig` now attaches to the chord callback (Issue #3356).

Periodic Tasks

New API for configuring periodic tasks

This new API enables you to use signatures when defining periodic tasks, removing the chance of mistyping task names.

An example of the new API is *here*.

Optimized Beat implementation

The **celery beat** implementation has been optimized for millions of periodic tasks by using a heap to schedule entries.

Contributed by **Ask Solem** and **Alexander Koshelev**.

Schedule tasks based on sunrise, sunset, dawn and dusk

See *Solar schedules* for more information.

Contributed by **Mark Parncutt**.

Result Backends

RPC Result Backend matured

Lots of bugs in the previously experimental RPC result backend have been fixed and can now be considered to production use.

Contributed by **Ask Solem**, **Morris Tweed**.

Redis: Result backend optimizations

`result.get()` is now using pub/sub for streaming task results

Calling `result.get()` when using the Redis result backend used to be extremely expensive as it was using polling to wait for the result to become available. A default polling interval of 0.5 seconds didn't help performance, but was necessary to avoid a spin loop.

The new implementation is using Redis Pub/Sub mechanisms to publish and retrieve results immediately, greatly improving task round-trip times.

Contributed by **Yaroslav Zhavoronkov** and **Ask Solem**.

New optimized chord join implementation

This was an experimental feature introduced in Celery 3.1, that could only be enabled by adding `?new_join=1` to the result backend URL configuration.

We feel that the implementation has been tested thoroughly enough to be considered stable and enabled by default.

The new implementation greatly reduces the overhead of chords, and especially with larger chords the performance benefit can be massive.

New Riak result backend introduced

See *Riak backend settings* for more information.

Contributed by **Gilles Dartiguelongue**, **Alman One** and **NoKriK**.

New CouchDB result backend introduced

See *CouchDB backend settings* for more information.

Contributed by **Nathan Van Gheem**.

New Consul result backend introduced

Add support for Consul as a backend using the Key/Value store of Consul.

Consul has an HTTP API where through you can store keys with their values.

The backend extends KeyValueStoreBackend and implements most of the methods.

Mainly to set, get and remove objects.

This allows Celery to store Task results in the K/V store of Consul.

Consul also allows to set a TTL on keys using the Sessions from Consul. This way the backend supports auto expiry of Task results.

For more information on Consul visit https://consul.io/

The backend uses python-consul for talking to the HTTP API. This package is fully Python 3 compliant just as this backend is:

```
$ pip install python-consul
```

That installs the required package to talk to Consul's HTTP API from Python.

You can also specify consul as an extension in your dependency on Celery:

```
$ pip install celery[consul]
```

See *Bundles* for more information.

Contributed by **Wido den Hollander**.

Brand new Cassandra result backend

A brand new Cassandra backend utilizing the new cassandra-driver library is replacing the old result backend using the older pycassa library.

See *Cassandra backend settings* for more information.

To depend on Celery with Cassandra as the result backend use:

```
$ pip install celery[cassandra]
```

You can also combine multiple extension requirements, please see *Bundles* for more information.

New Elasticsearch result backend introduced

See *Elasticsearch backend settings* for more information.

To depend on Celery with Elasticsearch as the result bakend use:

```
$ pip install celery[elasticsearch]
```

You can also combine multiple extension requirements, please see *Bundles* for more information.

Contributed by **Ahmet Demir**.

New File-system result backend introduced

See *File-system backend settings* for more information.

Contributed by **Môshe van der Sterre**.

Event Batching

Events are now buffered in the worker and sent as a list, reducing the overhead required to send monitoring events.

For authors of custom event monitors there will be no action required as long as you're using the Python Celery helpers (`Receiver`) to implement your monitor.

However, if you're parsing raw event messages you must now account for batched event messages, as they differ from normal event messages in the following way:

- The routing key for a batch of event messages will be set to `<event-group>.multi` where the only batched event group is currently `task` (giving a routing key of `task.multi`).

- The message body will be a serialized list-of-dictionaries instead of a dictionary. Each item in the list can be regarded as a normal event message body.

In Other News. . .

Requirements

- Now depends on Kombu 4.0.

- Now depends on billiard version 3.5.

- No longer depends on anyjson. Good-bye old friend :(

Tasks

- The "anon-exchange" is now used for simple name-name direct routing.

  This increases performance as it completely bypasses the routing table, in addition it also improves reliability for the Redis broker transport.

- An empty ResultSet now evaluates to True.

  > Fix contributed by **Colin McIntosh**.

- The default routing key (`task_default_routing_key`) and exchange name (`task_default_exchange`) is now taken from the `task_default_queue` setting.

  > This means that to change the name of the default queue, you now only have to set a single setting.

- New `task_reject_on_worker_lost` setting, and `reject_on_worker_lost` task attribute decides what happens when the child worker process executing a late ack task is terminated.

  > Contributed by **Michael Permana**.

- `Task.subtask` renamed to `Task.signature` with alias.

- `Task.subtask_from_request` renamed to `Task.signature_from_request` with alias.

- The `delivery_mode` attribute for `kombu.Queue` is now respected (Issue #1953).

- Routes in `task-routes` can now specify a `Queue` instance directly.

  > Example:

  ```
  task_routes = {'proj.tasks.add': {'queue': Queue('add')}}
  ```

- `AsyncResult` now raises `ValueError` if task_id is None. (Issue #1996).

- Retried tasks didn't forward expires setting (Issue #3297).

- `result.get()` now supports an `on_message` argument to set a callback to be called for every message received.

- New abstract classes added:

  - *CallableTask*

    > Looks like a task.

  - *CallableSignature*

    > Looks like a task signature.

- `Task.replace` now properly forwards callbacks (Issue #2722).

  > Fix contributed by **Nicolas Unravel**.

- `Task.replace`: Append to chain/chord (Closes #3232)

  > Fixed issue #3232, adding the signature to the chain (if there's any). Fixed the chord suppress if the given signature contains one.

  > Fix contributed by @honux.

- Task retry now also throws in eager mode.

  > Fix contributed by **Feanil Patel**.

Beat

- Fixed crontab infinite loop with invalid date.

    When occurrence can never be reached (example, April, 31th), trying to reach the next occurrence would trigger an infinite loop.

    Try fixing that by raising a `RuntimeError` after 2,000 iterations

    (Also added a test for crontab leap years in the process)

    Fix contributed by **Romuald Brunet**.

- Now ensures the program exits with a non-zero exit code when an exception terminates the service.

    Fix contributed by **Simon Peeters**.

App

- Dates are now always timezone aware even if `enable_utc` is disabled (Issue #943).

    Fix contributed by **Omer Katz**.

- **Config**: App preconfiguration is now also pickled with the configuration.

    Fix contributed by **Jeremy Zafran**.

- **The application can now change how task names are generated using** the `gen_task_name()` method.

    Contributed by **Dmitry Malinovsky**.

- App has new `app.current_worker_task` property that returns the task that's currently being worked on (or `None`). (Issue #2100).

Logging

- `get_task_logger()` now raises an exception if trying to use the name "celery" or "celery.task" (Issue #3475).

Execution Pools

- **Eventlet/Gevent**: now enables AMQP heartbeat (Issue #3338).
- **Eventlet/Gevent**: Fixed race condition leading to "simultaneous read" errors (Issue #2755).
- **Prefork**: Prefork pool now uses `poll` instead of `select` where available (Issue #2373).
- **Prefork**: Fixed bug where the pool would refuse to shut down the worker (Issue #2606).
- **Eventlet**: Now returns pool size in **celery inspect stats** command.

    Contributed by **Alexander Oblovatniy**.

Testing

- Celery is now a pytest plugin, including fixtures useful for unit and integration testing.

    See the *testing user guide* for more information.

Transports

- `amqps://` can now be specified to require SSL.

- **Redis Transport**: The Redis transport now supports the *broker_use_ssl* option.

  Contributed by **Robert Kolba**.

- JSON serializer now calls `obj.__json__` for unsupported types.

  This means you can now define a __json__ method for custom types that can be reduced down to a built-in json type.

  Example:

```python
class Person:
    first_name = None
    last_name = None
    address = None

    def __json__(self):
        return {
            'first_name': self.first_name,
            'last_name': self.last_name,
            'address': self.address,
        }
```

- JSON serializer now handles datetime's, Django promise, UUID and Decimal.

- New `Queue.consumer_arguments` can be used for the ability to set consumer priority via `x-priority`.

  See https://www.rabbitmq.com/consumer-priority.html

  Example:

```python
consumer = Consumer(channel, consumer_arguments={'x-priority': 3})
```

- Queue/Exchange: `no_declare` option added (also enabled for internal amq. exchanges).

Programs

- Celery is now using `argparse`, instead of `optparse`.

- All programs now disable colors if the controlling terminal is not a TTY.

- **celery worker**: The `-q` argument now disables the startup banner.

- **celery worker**: The "worker ready" message is now logged using severity info, instead of warn.

- **celery multi**: `%n` format for is now synonym with `%N` to be consistent with **celery worker**.

- **celery inspect**/**celery control**: now supports a new *--json* option to give output in json format.

- **celery inspect registered**: now ignores built-in tasks.

- **celery purge** now takes `-Q` and `-X` options used to specify what queues to include and exclude from the purge.

- New **celery logtool**: Utility for filtering and parsing celery worker log-files

- **celery multi**: now passes through *%i* and *%I* log file formats.

- General: `%p` can now be used to expand to the full worker node-name in log-file/pid-file arguments.

- **A new command line option** `--executable` is now available for daemonizing programs (**celery worker** and **celery beat**).

    Contributed by **Bert Vanderbauwhede**.

- **celery worker**: supports new `--prefetch-multiplier` option.

    Contributed by **Mickaël Penhard**.

- The `--loader` argument is now always effective even if an app argument is set (Issue #3405).

- inspect/control now takes commands from registry

    This means user remote-control commands can also be used from the command-line.

    Note that you need to specify the arguments/and type of arguments for the arguments to be correctly passed on the command-line.

    There are now two decorators, which use depends on the type of command: *@inspect_command* + *@control_command*:

    ```python
    from celery.worker.control import control_command

    @control_command(
        args=[('n', int)]
        signature='[N=1]',
    )
    def something(state, n=1, **kwargs):
        ...
    ```

    Here `args` is a list of args supported by the command. The list must contain tuples of `(argument_name, type)`.

    `signature` is just the command-line help used in e.g. `celery -A proj control --help`.

    Commands also support *variadic* arguments, which means that any arguments left over will be added to a single variable. Here demonstrated by the `terminate` command which takes a signal argument and a variable number of task_ids:

    ```python
    from celery.worker.control import control_command

    @control_command(
        args=[('signal', str)],
        signature='<signal> [id1, [id2, [..., [idN]]]]',
        variadic='ids',
    )
    def terminate(state, signal, ids, **kwargs):
        ...
    ```

    This command can now be called using:

    ```
    $ celery -A proj control terminate SIGKILL id1 id2 id3`
    ```

    See *Writing your own remote control commands* for more information.

Worker

- Improvements and fixes for *LimitedSet*.

Getting rid of leaking memory + adding `minlen` size of the set: the minimal residual size of the set after operating for some time. `minlen` items are kept, even if they should've been expired.

Problems with older and even more old code:

1. Heap would tend to grow in some scenarios (like adding an item multiple times).

2. Adding many items fast wouldn't clean them soon enough (if ever).

3. When talking to other workers, revoked._data was sent, but it was processed on the other side as iterable. That means giving those keys new (current) time-stamp. By doing this workers could recycle items forever. Combined with 1) and 2), this means that in large set of workers, you're getting out of memory soon.

All those problems should be fixed now.

This should fix issues #3095, #3086.

Contributed by **David Pravec**.

- New settings to control remote control command queues.

    - *control_queue_expires*

        Set queue expiry time for both remote control command queues, and remote control reply queues.

    - *control_queue_ttl*

        Set message time-to-live for both remote control command queues, and remote control reply queues.

    Contributed by **Alan Justino**.

- The *worker_shutdown* signal is now always called during shutdown.

    Previously it would not be called if the worker instance was collected by gc first.

- Worker now only starts the remote control command consumer if the broker transport used actually supports them.

- Gossip now sets `x-message-ttl` for event queue to heartbeat_interval s. (Issue #2005).

- Now preserves exit code (Issue #2024).

- Now rejects messages with an invalid ETA value (instead of ack, which means they will be sent to the dead-letter exchange if one is configured).

- Fixed crash when the `-purge` argument was used.

- Log–level for unrecoverable errors changed from `error` to `critical`.

- Improved rate limiting accuracy.

- Account for missing timezone information in task expires field.

    Fix contributed by **Albert Wang**.

- **The worker no longer has a `Queues` bootsteps, as it is now** superfluous.

- Now emits the "Received task" line even for revoked tasks. (Issue #3155).

- Now respects *broker_connection_retry* setting.

    Fix contributed by **Nat Williams**.

- New *control_queue_ttl* and *control_queue_expires* settings now enables you to configure remote control command message TTLs, and queue expiry time.

Contributed by **Alan Justino**.

- New `celery.worker.state.requests` enables O(1) loookup of active/reserved tasks by id.

- Auto-scale didn't always update keep-alive when scaling down.

   Fix contributed by **Philip Garnero**.

- Fixed typo `options_list` -> `option_list`.

   Fix contributed by **Greg Wilbur**.

- Some worker command-line arguments and `Worker()` class arguments have been renamed for consistency.

   All of these have aliases for backward compatibility.

   - `--send-events` -> `--task-events`

   - `--schedule` -> `--schedule-filename`

   - `--maxtasksperchild` -> `--max-tasks-per-child`

   - `Beat(scheduler_cls=)` -> `Beat(scheduler=)`

   - `Worker(send_events=True)` -> `Worker(task_events=True)`

   - `Worker(task_time_limit=)` -> `Worker(time_limit=)`

   - `Worker(task_soft_time_limit=)` -> `Worker(soft_time_limit=)`

   - `Worker(state_db=)` -> `Worker(statedb=)`

   - `Worker(working_directory=)` -> `Worker(workdir=)`

Debugging Utilities

- *`celery.contrib.rdb`*: Changed remote debugger banner so that you can copy and paste the address easily (no longer has a period in the address).

   Contributed by **Jonathan Vanasco**.

- Fixed compatibility with recent psutil versions (Issue #3262).

Signals

- **App**: New signals for app configuration/finalization:
   - *app.on_configure*
   - *app.on_after_configure*
   - *app.on_after_finalize*

- **Task**: New task signals for rejected task messages:
   - `celery.signals.task_rejected`.
   - `celery.signals.task_unknown`.

- **Worker**: New signal for when a heartbeat event is sent.
   - `celery.signals.heartbeat_sent`

      Contributed by **Kevin Richardson**.

Events

- Event messages now uses the RabbitMQ `x-message-ttl` option to ensure older event messages are discarded.

  The default is 5 seconds, but can be changed using the *event_queue_ttl* setting.

- `Task.send_event` now automatically retries sending the event on connection failure, according to the task publish retry settings.

- Event monitors now sets the *event_queue_expires* setting by default.

  The queues will now expire after 60 seconds after the monitor stops consuming from it.

- Fixed a bug where a None value wasn't handled properly.

  Fix contributed by **Dongweiming**.

- New *event_queue_prefix* setting can now be used to change the default `celeryev` queue prefix for event receiver queues.

  Contributed by **Takeshi Kanemoto**.

- `State.tasks_by_type` and `State.tasks_by_worker` can now be used as a mapping for fast access to this information.

Deployment

- Generic init-scripts now support `CELERY_SU` and `CELERYD_SU_ARGS` environment variables to set the path and arguments for **su** (*su(1)*).

- Generic init-scripts now better support FreeBSD and other BSD systems by searching `/usr/local/etc/` for the configuration file.

  Contributed by **Taha Jahangir**.

- Generic init-script: Fixed strange bug for `celerybeat` where restart didn't always work (Issue #3018).

- The systemd init script now uses a shell when executing services.

  Contributed by **Tomas Machalek**.

Result Backends

- Redis: Now has a default socket timeout of 120 seconds.

  The default can be changed using the new *redis_socket_timeout* setting.

  Contributed by **Raghuram Srinivasan**.

- RPC Backend result queues are now auto delete by default (Issue #2001).

- RPC Backend: Fixed problem where exception wasn't deserialized properly with the json serializer (Issue #2518).

  Fix contributed by **Allard Hoeve**.

- CouchDB: The backend used to double-json encode results.

  Fix contributed by **Andrew Stewart**.

- CouchDB: Fixed typo causing the backend to not be found (Issue #3287).

Fix contributed by **Andrew Stewart**.

- MongoDB: Now supports setting the `result_serialzier` setting to `bson` to use the MongoDB libraries own serializer.

  Contributed by **Davide Quarta**.

- **MongoDB: URI handling has been improved to use** database name, user and password from the URI if provided.

  Contributed by **Samuel Jaillet**.

- SQLAlchemy result backend: Now ignores all result engine options when using NullPool (Issue #1930).

- SQLAlchemy result backend: Now sets max char size to 155 to deal with brain damaged MySQL Unicode implementation (Issue #1748).

- **General**: All Celery exceptions/warnings now inherit from common *CeleryError*/*CeleryWarning*. (Issue #2643).

Documentation Improvements

Contributed by:

- Adam Chainz
- Amir Rustamzadeh
- Arthur Vuillard
- Batiste Bieler
- Berker Peksag
- Bryce Groff
- Daniel Devine
- Edward Betts
- Jason Veatch
- Jeff Widman
- Maciej Obuchowski
- Manuel Kaufmann
- Maxime Beauchemin
- Mitchel Humpherys
- Pavlo Kapyshin
- Pierre Fersing
- Rik
- Steven Sklar
- Tayfun Sen
- Wieland Hoffmann

Reorganization, Deprecations, and Removals

Incompatible changes

- Prefork: Calling `result.get()` or joining any result from within a task now raises `RuntimeError`.

  In previous versions this would emit a warning.

- `celery.worker.consumer` is now a package, not a module.

- Module `celery.worker.job` renamed to `celery.worker.request`.

- Beat: `Scheduler.Publisher`/`.publisher` renamed to `.Producer`/`.producer`.

- Result: The task_name argument/attribute of `app.AsyncResult` was removed.

  This was historically a field used for `pickle` compatibility, but is no longer needed.

- Backends: Arguments named `status` renamed to `state`.

- Backends: `backend.get_status()` renamed to `backend.get_state()`.

- Backends: `backend.maybe_reraise()` renamed to `.maybe_throw()`

  The promise API uses .throw(), so this change was made to make it more consistent.

  There's an alias available, so you can still use maybe_reraise until Celery 5.0.

Unscheduled Removals

- The experimental `celery.contrib.methods` feature has been removed, as there were far many bugs in the implementation to be useful.

- The CentOS init-scripts have been removed.

  These didn't really add any features over the generic init-scripts, so you're encouraged to use them instead, or something like supervisor.

Reorganization Deprecations

These symbols have been renamed, and while there's an alias available in this version for backward compatibility, they will be removed in Celery 5.0, so make sure you rename these ASAP to make sure it won't break for that release.

Chances are that you'll only use the first in this list, but you never know:

- `celery.utils.worker_direct` -> `celery.utils.nodenames.worker_direct()`.

- `celery.utils.nodename` -> `celery.utils.nodenames.nodename()`.

- `celery.utils.anon_nodename` -> `celery.utils.nodenames.anon_nodename()`.

- `celery.utils.nodesplit` -> `celery.utils.nodenames.nodesplit()`.

- `celery.utils.default_nodename` -> `celery.utils.nodenames.default_nodename()`.

- `celery.utils.node_format` -> `celery.utils.nodenames.node_format()`.

- `celery.utils.host_format` -> `celery.utils.nodenames.host_format()`.

Scheduled Removals

Modules

- Module `celery.worker.job` has been renamed to `celery.worker.request`.

  This was an internal module so shouldn't have any effect. It's now part of the public API so must not change again.

- Module `celery.task.trace` has been renamed to `celery.app.trace` as the `celery.task` package is being phased out. The module will be removed in version 5.0 so please change any import from:

```python
from celery.task.trace import X
```

  to:

```python
from celery.app.trace import X
```

- Old compatibility aliases in the `celery.loaders` module has been removed.

  - Removed `celery.loaders.current_loader()`, use: `current_app.loader`

  - Removed `celery.loaders.load_settings()`, use: `current_app.conf`

Result

- **AsyncResult.serializable() and celery.result.from_serializable** has been removed:

  Use instead:

```python
>>> tup = result.as_tuple()
>>> from celery.result import result_from_tuple
>>> result = result_from_tuple(tup)
```

- Removed `BaseAsyncResult`, use `AsyncResult` for instance checks instead.

- Removed `TaskSetResult`, use `GroupResult` instead.

  - `TaskSetResult.total` -> `len(GroupResult)`

  - `TaskSetResult.taskset_id` -> `GroupResult.id`

- Removed `ResultSet.subtasks`, use `ResultSet.results` instead.

TaskSet

TaskSet has been removed, as it was replaced by the `group` construct in Celery 3.0.

If you have code like this:

```python
>>> from celery.task import TaskSet

>>> TaskSet(add.subtask((i, i)) for i in xrange(10)).apply_async()
```

You need to replace that with:

```python
>>> from celery import group
>>> group(add.s(i, i) for i in xrange(10))()
```

Events

- Removals for class *celery.events.state.Worker*:

    - `Worker._defaults` attribute.

        Use `{k: getattr(worker, k) for k in worker._fields}`.

    - `Worker.update_heartbeat`

        Use `Worker.event(None, timestamp, received)`

    - `Worker.on_online`

        Use `Worker.event('online', timestamp, received, fields)`

    - `Worker.on_offline`

        Use `Worker.event('offline', timestamp, received, fields)`

    - `Worker.on_heartbeat`

        Use `Worker.event('heartbeat', timestamp, received, fields)`

- Removals for class *celery.events.state.Task*:

    - `Task._defaults` attribute.

        Use `{k: getattr(task, k) for k in task._fields}`.

    - `Task.on_sent`

        Use `Worker.event('sent', timestamp, received, fields)`

    - `Task.on_received`

        Use `Task.event('received', timestamp, received, fields)`

    - `Task.on_started`

        Use `Task.event('started', timestamp, received, fields)`

    - `Task.on_failed`

        Use `Task.event('failed', timestamp, received, fields)`

    - `Task.on_retried`

        Use `Task.event('retried', timestamp, received, fields)`

    - `Task.on_succeeded`

        Use `Task.event('succeeded', timestamp, received, fields)`

    - `Task.on_revoked`

        Use `Task.event('revoked', timestamp, received, fields)`

    - `Task.on_unknown_event`

        Use `Task.event(short_type, timestamp, received, fields)`

    - `Task.update`

        Use `Task.event(short_type, timestamp, received, fields)`

    - `Task.merge`

        Contact us if you need this.

Magic keyword arguments

Support for the very old magic keyword arguments accepted by tasks is finally removed in this version.

If you're still using these you have to rewrite any task still using the old `celery.decorators` module and depending on keyword arguments being passed to the task, for example:

```python
from celery.decorators import task


@task()
def add(x, y, task_id=None):
    print('My task id is %r' % (task_id,))
```

should be rewritten into:

```python
from celery import task


@task(bind=True)
def add(self, x, y):
    print('My task id is {0.request.id}'.format(self))
```

Removed Settings

The following settings have been removed, and is no longer supported:

Logging Settings

Setting name	Replace with
CELERYD_LOG_LEVEL	*celery worker --loglevel*
CELERYD_LOG_FILE	*celery worker --logfile*
CELERYBEAT_LOG_LEVEL	*celery beat --loglevel*
CELERYBEAT_LOG_FILE	*celery beat --logfile*
CELERYMON_LOG_LEVEL	celerymon is deprecated, use flower
CELERYMON_LOG_FILE	celerymon is deprecated, use flower
CELERYMON_LOG_FORMAT	celerymon is deprecated, use flower

Task Settings

Setting name	Replace with
CELERY_CHORD_PROPAGATES	N/A

Changes to internal API

- Module `celery.datastructures` renamed to *celery.utils.collections*.

- Module `celery.utils.timeutils` renamed to *celery.utils.time*.

- `celery.utils.datastructures.DependencyGraph` moved to *celery.utils.graph*.

- `celery.utils.jsonify` is now `celery.utils.serialization.jsonify()`.

- celery.utils.strtobool is now *celery.utils.serialization.strtobool()*.

- celery.utils.is_iterable has been removed.

   Instead use:

   ```
   isinstance(x, collections.Iterable)
   ```

- celery.utils.lpmerge is now *celery.utils.collections.lpmerge()*.

- celery.utils.cry is now celery.utils.debug.cry().

- celery.utils.isatty is now *celery.platforms.isatty()*.

- celery.utils.gen_task_name is now *celery.utils.imports.gen_task_name()*.

- celery.utils.deprecated is now *celery.utils.deprecated.Callable()*

- celery.utils.deprecated_property is now *celery.utils.deprecated.Property()*.

- celery.utils.warn_deprecated is now *celery.utils.deprecated.warn()*

Deprecation Time-line Changes

See the *Celery Deprecation Time-line*.

3.13.6 Change history

This document contains change notes for bugfix releases in the 4.0.x series (latentcall), please see *What's new in Celery 4.0 (latentcall)* for an overview of what's new in Celery 4.0.

4.0.2

release-date 2016-12-15 03:40 PM PST

release-by Ask Solem

- **Requirements**

   – Now depends on Kombu 4.0.2.

- **Tasks**: Fixed problem with JSON serialization of *group* (keys must be string error, Issue #3688).

- **Worker**: Fixed JSON serialization issue when using inspect active and friends (Issue #3667).

- **App**: Fixed saferef errors when using signals (Issue #3670).

- **Prefork**: Fixed bug with pack requiring bytes argument on Python 2.7.5 and earlier (Issue #3674).

- **Tasks**: Saferepr did not handle unicode in bytestrings on Python 2 (Issue #3676).

- **Testing**: Added new celery_worker_paremeters fixture.

   Contributed by **Michael Howitz**.

- **Tasks**: Added new app argument to GroupResult.restore (Issue #3669).

   This makes the restore method behave the same way as the GroupResult constructor.

   Contributed by **Andreas Pelme**.

- **Tasks**: Fixed type checking crash when task takes *args on Python 3 (Issue #3678).

- Documentation and examples improvements by:

    - **BLAGA Razvan-Paul**

    - **Michael Howitz**

    - @paradox41

4.0.1

release-date 2016-12-08 05:22 PM PST

release-by Ask Solem

- [Security: CELERYSA-0003] Insecure default configuration

    The default *accept_content* setting was set to allow deserialization of pickled messages in
    Celery 4.0.0.

    The insecure default has been fixed in 4.0.1, and you can also configure the 4.0.0 version to explic-
    itly only allow json serialized messages:

    ```
    app.conf.accept_content = ['json']
    ```

- **Tasks**: Added new method to register class-based tasks (Issue #3615).

    To register a class based task you should now call `app.register_task`:

    ```
    from celery import Celery, Task

    app = Celery()

    class CustomTask(Task):

        def run(self):
            return 'hello'

    app.register_task(CustomTask())
    ```

- **Tasks**: Argument checking now supports keyword-only arguments on Python3 (Issue #3658).

    Contributed by @sww.

- **Tasks**: The `task-sent` event was not being sent even if configured to do so (Issue #3646).

- **Worker**: Fixed AMQP heartbeat support for eventlet/gevent pools (Issue #3649).

- **App**: `app.conf.humanize()` would not work if configuration not finalized (Issue #3652).

- **Utils**: `saferepr` attempted to show iterables as lists and mappings as dicts.

- **Utils**: `saferepr` did not handle unicode-errors when attempting to format `bytes` on Python 3 (Issue #3610).

- **Utils**: `saferepr` should now properly represent byte strings with non-ascii characters (Issue #3600).

- **Results**: Fixed bug in elasticsearch where _index method missed the body argument (Issue #3606).

    Fix contributed by (Sean Ho).

- **Canvas**: Fixed `ValueError` in chord with single task header (Issue #3608).

    Fix contributed by **Viktor Holmqvist**.

- **Task**: Ensure class-based task has name prior to registration (Issue #3616).

Fix contributed by **Rick Wargo**.

- **Beat**: Fixed problem with strings in shelve (Issue #3644).

    Fix contributed by **Alli**.

- **Worker**: Fixed `KeyError` in `inspect stats` when `-O` argument set to something other than `fast` or `fair` (Issue #3621).

- **Task**: Retried tasks were no longer sent to the original queue (Issue #3622).

- **Worker**: Python 3: Fixed None/int type comparison in `apps/worker.py` (Issue #3631).

- **Results**: Redis has a new `redis_socket_connect_timeout` setting.

- **Results**: Redis result backend passed the `socket_connect_timeout` argument to UNIX socket based connections by mistake, causing a crash.

- **Worker**: Fixed missing logo in worker splash screen when running on Python 3.x (Issue #3627).

    Fix contributed by **Brian Luan**.

- **Deps**: Fixed `celery[redis]` bundle installation (Issue #3643).

    Fix contributed by **Rémi Marenco**.

- **Deps**: Bundle `celery[sqs]` now also requires [pycurl](Issue #3619).

- **Worker**: Hard time limits were no longer being respected (Issue #3618).

- **Worker**: Soft time limit log showed `Trues` instead of the number of seconds.

- **App**: `registry_cls` argument no longer had any effect (Issue #3613).

- **Worker**: Event producer now uses `connection_for_Write` (Issue #3525).

- **Results**: Redis/memcache backends now uses `result_expires` to expire chord counter (Issue #3573).

    Contributed by **Tayfun Sen**.

- **Django**: Fixed command for upgrading settings with Django (Issue #3563).

    Fix contributed by **François Voron**.

- **Testing**: Added a `celery_parameters` test fixture to be able to use customized `Celery` init parameters. (#3626)

    Contributed by **Steffen Allner**.

- Documentation improvements contributed by

    - @csfeathers
    - **Moussa Taifi**
    - **Yuhannaa**
    - **Laurent Peuch**
    - **Christian**
    - **Bruno Alla**
    - **Steven Johns**
    - @tnir
    - **GDR!**

4.0.0

release-date 2016-11-04 02:00 P.M PDT

release-by Ask Solem

See *What's new in Celery 4.0 (latentcall)* (in `docs/whatsnew-4.0.rst`).

4.0.0rc7

release-date 2016-11-02 01:30 P.M PDT

Important notes

- Database result backend related setting names changed from `sqlalchemy_*` -> `database_*`.

  The `sqlalchemy_` named settings won't work at all in this version so you need to rename them. This is a last minute change, and as they were not supported in 3.1 we will not be providing aliases.

- `chain(A, B, C)` now works the same way as `A | B | C`.

  This means calling `chain()` might not actually return a chain, it can return a group or any other type depending on how the workflow can be optimized.

3.13.7 What's new in Celery 3.1 (Cipater)

Author Ask Solem (`ask at celeryproject.org`)

Change history

What's new documents describe the changes in major versions, we also have a *Change history* that lists the changes in bugfix releases (0.0.x), while older series are archived under the *History* section.

Celery is a simple, flexible, and reliable distributed system to process vast amounts of messages, while providing operations with the tools required to maintain such a system.

It's a task queue with focus on real-time processing, while also supporting task scheduling.

Celery has a large and diverse community of users and contributors, you should come join us *on IRC* or *our mailing-list*.

To read more about Celery you should go read the *introduction*.

While this version is backward compatible with previous versions it's important that you read the following section.

This version is officially supported on CPython 2.6, 2.7, and 3.3, and also supported on PyPy.

Table of Contents

Make sure you read the important notes before upgrading to this version.

Preface

Deadlocks have long plagued our workers, and while uncommon they're not acceptable. They're also infamous for being extremely hard to diagnose and reproduce, so to make this job easier I wrote a stress test suite that bombards the worker with different tasks in an attempt to break it.

What happens if thousands of worker child processes are killed every second? what if we also kill the broker connection every 10 seconds? These are examples of what the stress test suite will do to the worker, and it reruns these tests using different configuration combinations to find edge case bugs.

The end result was that I had to rewrite the prefork pool to avoid the use of the POSIX semaphore. This was extremely challenging, but after months of hard work the worker now finally passes the stress test suite.

There's probably more bugs to find, but the good news is that we now have a tool to reproduce them, so should you be so unlucky to experience a bug then we'll write a test for it and squash it!

Note that I've also moved many broker transports into experimental status: the only transports recommended for production use today is RabbitMQ and Redis.

I don't have the resources to maintain all of them, so bugs are left unresolved. I wish that someone will step up and take responsibility for these transports or donate resources to improve them, but as the situation is now I don't think the quality is up to date with the rest of the code-base so I cannot recommend them for production use.

The next version of Celery 4.0 will focus on performance and removing rarely used parts of the library. Work has also started on a new message protocol, supporting multiple languages and more. The initial draft can be found *here*.

This has probably been the hardest release I've worked on, so no introduction to this changelog would be complete without a massive thank you to everyone who contributed and helped me test it!

Thank you for your support!

— *Ask Solem*

Important Notes

Dropped support for Python 2.5

Celery now requires Python 2.6 or later.

The new dual code base runs on both Python 2 and 3, without requiring the `2to3` porting tool.

Note: This is also the last version to support Python 2.6! From Celery 4.0 and on-wards Python 2.7 or later will be required.

Last version to enable Pickle by default

Starting from Celery 4.0 the default serializer will be json.

If you depend on pickle being accepted you should be prepared for this change by explicitly allowing your worker to consume pickled messages using the `CELERY_ACCEPT_CONTENT` setting:

```
CELERY_ACCEPT_CONTENT = ['pickle', 'json', 'msgpack', 'yaml']
```

Make sure you only select the serialization formats you'll actually be using, and make sure you've properly secured your broker from unwanted access (see the *Security Guide*).

The worker will emit a deprecation warning if you don't define this setting.

for Kombu users

Kombu 3.0 no longer accepts pickled messages by default, so if you use Kombu directly then you have to configure your consumers: see the Kombu 3.0 Changelog for more information.

Old command-line programs removed and deprecated

Everyone should move to the new **celery** umbrella command, so we're incrementally deprecating the old command names.

In this version we've removed all commands that aren't used in init-scripts. The rest will be removed in 4.0.

Program	New Status	Replacement
`celeryd`	*DEPRECATED*	`celery worker`
`celerybeat`	*DEPRECATED*	`celery beat`
`celeryd-multi`	*DEPRECATED*	`celery multi`
`celeryctl`	**REMOVED**	`celery inspect|control`
`celeryev`	**REMOVED**	`celery events`
`camqadm`	**REMOVED**	`celery amqp`

If this isn't a new installation then you may want to remove the old commands:

```
$ pip uninstall celery
$ # repeat until it fails
# ...
$ pip uninstall celery
$ pip install celery
```

Please run `celery --help` for help using the umbrella command.

News

Prefork Pool Improvements

These improvements are only active if you use an async capable transport. This means only RabbitMQ (AMQP) and Redis are supported at this point and other transports will still use the thread-based fallback implementation.

- Pool is now using one IPC queue per child process.

   Previously the pool shared one queue between all child processes, using a POSIX semaphore as a mutex to achieve exclusive read and write access.

   The POSIX semaphore has now been removed and each child process gets a dedicated queue. This means that the worker will require more file descriptors (two descriptors per process), but it also means that performance is improved and we can send work to individual child processes.

   POSIX semaphores aren't released when a process is killed, so killing processes could lead to a deadlock if it happened while the semaphore was acquired. There's no good solution to fix this, so the best option was to remove the semaphore.

- Asynchronous write operations

   The pool now uses async I/O to send work to the child processes.

- Lost process detection is now immediate.

   If a child process is killed or exits mysteriously the pool previously had to wait for 30 seconds before marking the task with a *WorkerLostError*. It had to do this because the out-queue was shared between all processes, and the pool couldn't be certain whether the process completed the task or not. So an arbitrary timeout of 30 seconds was chosen, as it was believed that the out-queue would've been drained by this point.

   This timeout is no longer necessary, and so the task can be marked as failed as soon as the pool gets the notification that the process exited.

- Rare race conditions fixed

   Most of these bugs were never reported to us, but were discovered while running the new stress test suite.

Caveats

> **Long running tasks**
>
> The new pool will send tasks to a child process as long as the process in-queue is writable, and since the socket is buffered this means that the processes are, in effect, prefetching tasks.
>
> This benefits performance but it also means that other tasks may be stuck waiting for a long running task to complete:
>
> ```
> -> send T1 to Process A
> # A executes T1
> -> send T2 to Process B
> # B executes T2
> <- T2 complete
>
> -> send T3 to Process A
> # A still executing T1, T3 stuck in local buffer and
> # won't start until T1 returns
> ```
>
> The buffer size varies based on the operating system: some may have a buffer as small as 64KB but on recent Linux versions the buffer size is 1MB (can only be changed system wide).
>
> You can disable this prefetching behavior by enabling the `-Ofair` worker option:
>
> ```
> $ celery -A proj worker -l info -Ofair
> ```
>
> With this option enabled the worker will only write to workers that are available for work, disabling the prefetch behavior.

> **Max tasks per child**
>
> If a process exits and pool prefetch is enabled the worker may have already written many tasks to the process in-queue, and these tasks must then be moved back and rewritten to a new process.
>
> This is very expensive if you have the `--max-tasks-per-child` option set to a low value (e.g., less than 10), you should not be using the `-Ofast` scheduler option.

Django supported out of the box

Celery 3.0 introduced a shiny new API, but unfortunately didn't have a solution for Django users.

The situation changes with this version as Django is now supported in core and new Django users coming to Celery are now expected to use the new API directly.

The Django community has a convention where there's a separate `django-x` package for every library, acting like a bridge between Django and the library.

Having a separate project for Django users has been a pain for Celery, with multiple issue trackers and multiple documentation sources, and then lastly since 3.0 we even had different APIs.

With this version we challenge that convention and Django users will use the same library, the same API and the same documentation as everyone else.

There's no rush to port your existing code to use the new API, but if you'd like to experiment with it you should know that:

- You need to use a Celery application instance.

    The new Celery API introduced in 3.0 requires users to instantiate the library by creating an application:

    ```
    from celery import Celery

    app = Celery()
    ```

- You need to explicitly integrate Celery with Django

    Celery won't automatically use the Django settings, so you can either configure Celery separately or you can tell it to use the Django settings with:

    ```
    app.config_from_object('django.conf:settings')
    ```

    Neither will it automatically traverse your installed apps to find task modules. If you want this behavior, you must explicitly pass a list of Django instances to the Celery app:

    ```
    from django.conf import settings
    app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
    ```

- You no longer use `manage.py`

    Instead you use the **celery** command directly:

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

    For this to work your app module must store the `DJANGO_SETTINGS_MODULE` environment variable, see the example in the *Django guide*.

To get started with the new API you should first read the *First Steps with Celery* tutorial, and then you should read the Django-specific instructions in *First steps with Django*.

The fixes and improvements applied by the django-celery library are now automatically applied by core Celery when it detects that the `DJANGO_SETTINGS_MODULE` environment variable is set.

The distribution ships with a new example project using Django in `examples/django`:

https://github.com/celery/celery/tree/3.1/examples/django

Some features still require the django-celery library:

- Celery doesn't implement the Django database or cache result backends.

- **Celery doesn't ship with the database-based periodic task** scheduler.

Note: If you're still using the old API when you upgrade to Celery 3.1 then you must make sure that your settings module contains the `djcelery.setup_loader()` line, since this will no longer happen as a side-effect of importing the django-celery module.

New users (or if you've ported to the new API) don't need the `setup_loader` line anymore, and must make sure to remove it.

Events are now ordered using logical time

Keeping physical clocks in perfect sync is impossible, so using time-stamps to order events in a distributed system isn't reliable.

Celery event messages have included a logical clock value for some time, but starting with this version that field is also used to order them.

Also, events now record timezone information by including a new `utcoffset` field in the event message. This is a signed integer telling the difference from UTC time in hours, so for example, an event sent from the Europe/London timezone in daylight savings time will have an offset of 1.

`app.events.Receiver` will automatically convert the time-stamps to the local timezone.

Note: The logical clock is synchronized with other nodes in the same cluster (neighbors), so this means that the logical epoch will start at the point when the first worker in the cluster starts.

If all of the workers are shutdown the clock value will be lost and reset to 0. To protect against this, you should specify the `celery worker --statedb` option such that the worker can persist the clock value at shutdown.

You may notice that the logical clock is an integer value and increases very rapidly. Don't worry about the value overflowing though, as even in the most busy clusters it may take several millennium before the clock exceeds a 64 bits value.

New worker node name format (`name@host`)

Node names are now constructed by two elements: name and host-name separated by '@'.

This change was made to more easily identify multiple instances running on the same machine.

If a custom name isn't specified then the worker will use the name 'celery' by default, resulting in a fully qualified node name of 'celery@hostname':

```
$ celery worker -n example.com
celery@example.com
```

To also set the name you must include the @:

```
$ celery worker -n worker1@example.com
worker1@example.com
```

The worker will identify itself using the fully qualified node name in events and broadcast messages, so where before a worker would identify itself as 'worker1.example.com', it'll now use 'celery@worker1.example.com'.

Remember that the `-n` argument also supports simple variable substitutions, so if the current host-name is *george.example.com* then the `%h` macro will expand into that:

```
$ celery worker -n worker1@%h
worker1@george.example.com
```

The available substitutions are as follows:

Variable	Substitution
%h	Full host-name (including domain name)
%d	Domain name only
%n	Host-name only (without domain name)
%%	The character %

Bound tasks

The task decorator can now create "bound tasks", which means that the task will receive the `self` argument.

```python
@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)
```

Using *bound tasks* is now the recommended approach whenever you need access to the task instance or request context. Previously one would've to refer to the name of the task instead (`send_twitter_status.retry`), but this could lead to problems in some configurations.

Mingle: Worker synchronization

The worker will now attempt to synchronize with other workers in the same cluster.

Synchronized data currently includes revoked tasks and logical clock.

This only happens at start-up and causes a one second start-up delay to collect broadcast responses from other workers.

You can disable this bootstep using the `celery worker --without-mingle` option.

Gossip: Worker <-> Worker communication

Workers are now passively subscribing to worker related events like heartbeats.

This means that a worker knows what other workers are doing and can detect if they go offline. Currently this is only used for clock synchronization, but there are many possibilities for future additions and you can write extensions that take advantage of this already.

Some ideas include consensus protocols, reroute task to best worker (based on resource usage or data locality) or restarting workers when they crash.

We believe that although this is a small addition, it opens amazing possibilities.

You can disable this bootstep using the `celery worker --without-gossip` option.

Bootsteps: Extending the worker

By writing bootsteps you can now easily extend the consumer part of the worker to add additional features, like custom message consumers.

The worker has been using bootsteps for some time, but these were never documented. In this version the consumer part of the worker has also been rewritten to use bootsteps and the new *Extensions and Bootsteps* guide documents examples extending the worker, including adding custom message consumers.

See the *Extensions and Bootsteps* guide for more information.

Note: Bootsteps written for older versions won't be compatible with this version, as the API has changed significantly.

The old API was experimental and internal but should you be so unlucky to use it then please contact the mailing-list and we'll help you port the bootstep to the new API.

New RPC result backend

This new experimental version of the `amqp` result backend is a good alternative to use in classical RPC scenarios, where the process that initiates the task is always the process to retrieve the result.

It uses Kombu to send and retrieve results, and each client uses a unique queue for replies to be sent to. This avoids the significant overhead of the original amqp result backend which creates one queue per task.

By default results sent using this backend won't persist, so they won't survive a broker restart. You can enable the `CELERY_RESULT_PERSISTENT` setting to change that.

```
CELERY_RESULT_BACKEND = 'rpc'
CELERY_RESULT_PERSISTENT = True
```

Note that chords are currently not supported by the RPC backend.

Time limits can now be set by the client

Two new options have been added to the Calling API: `time_limit` and `soft_time_limit`:

```
>>> res = add.apply_async((2, 2), time_limit=10, soft_time_limit=8)

>>> res = add.subtask((2, 2), time_limit=10, soft_time_limit=8).delay()

>>> res = add.s(2, 2).set(time_limit=10, soft_time_limit=8).delay()
```

Contributed by Mher Movsisyan.

Redis: Broadcast messages and virtual hosts

Broadcast messages are currently seen by all virtual hosts when using the Redis transport. You can now fix this by enabling a prefix to all channels so that the messages are separated:

```
BROKER_TRANSPORT_OPTIONS = {'fanout_prefix': True}
```

Note that you'll not be able to communicate with workers running older versions or workers that doesn't have this setting enabled.

This setting will be the default in a future version.

Related to Issue #1490.

pytz replaces python-dateutil dependency

Celery no longer depends on the python-dateutil library, but instead a new dependency on the pytz library was added.

The pytz library was already recommended for accurate timezone support.

This also means that dependencies are the same for both Python 2 and Python 3, and that the `requirements/default-py3k.txt` file has been removed.

Support for setuptools extra requirements

Pip now supports the setuptools extra requirements format, so we've removed the old bundles concept, and instead specify setuptools extras.

You install extras by specifying them inside brackets:

```
$ pip install celery[redis,mongodb]
```

The above will install the dependencies for Redis and MongoDB. You can list as many extras as you want.

> **Warning:** You can't use the `celery-with-*` packages anymore, as these won't be updated to use Celery 3.1.

Extension	Requirement entry	Type
Redis	`celery[redis]`	transport, result backend
MongoDB	`celery[mongodb]`	transport, result backend
CouchDB	`celery[couchdb]`	transport
Beanstalk	`celery[beanstalk]`	transport
ZeroMQ	`celery[zeromq]`	transport
Zookeeper	`celery[zookeeper]`	transport
SQLAlchemy	`celery[sqlalchemy]`	transport, result backend
librabbitmq	`celery[librabbitmq]`	transport (C amqp client)

The complete list with examples is found in the *Bundles* section.

subtask.__call__() now executes the task directly

A misunderstanding led to `Signature.__call__` being an alias of `.delay` but this doesn't conform to the calling API of `Task` which calls the underlying task method.

This means that:

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

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

now does the same as calling the task directly:

```python
>>> add(2, 2)
```

In Other News

- Now depends on Kombu 3.0.

- Now depends on billiard version 3.3.

- Worker will now crash if running as the root user with pickle enabled.

- Canvas: `group.apply_async` and `chain.apply_async` no longer starts separate task.

That the group and chord primitives supported the "calling API" like other subtasks was a nice idea, but it was useless in practice and often confused users. If you still want this behavior you can define a task to do it for you.

- New method `Signature.freeze()` can be used to "finalize" signatures/subtask.

    Regular signature:

    ```
    >>> s = add.s(2, 2)
    >>> result = s.freeze()
    >>> result
    <AsyncResult: ffacf44b-f8a1-44e9-80a3-703150151ef2>
    >>> s.delay()
    <AsyncResult: ffacf44b-f8a1-44e9-80a3-703150151ef2>
    ```

    Group:

    ```
    >>> g = group(add.s(2, 2), add.s(4, 4))
    >>> result = g.freeze()
    <GroupResult: e1094b1d-08fc-4e14-838e-6d601b99da6d [
        70c0fb3d-b60e-4b22-8df7-aa25b9abc86d,
        58fcd260-2e32-4308-a2ea-f5be4a24f7f4]>
    >>> g()
    <GroupResult: e1094b1d-08fc-4e14-838e-6d601b99da6d [70c0fb3d-b60e-4b22-
    ↪8df7-aa25b9abc86d, 58fcd260-2e32-4308-a2ea-f5be4a24f7f4]>
    ```

- Chord exception behavior defined (Issue #1172).

    From this version the chord callback will change state to FAILURE when a task part of a chord raises an exception.

    See more at *Error handling*.

- New ability to specify additional command line options to the worker and beat programs.

    The *app.user_options* attribute can be used to add additional command-line arguments, and expects `optparse`-style options:

    ```python
    from celery import Celery
    from celery.bin import Option

    app = Celery()
    app.user_options['worker'].add(
        Option('--my-argument'),
    )
    ```

    See the *Extensions and Bootsteps* guide for more information.

- All events now include a `pid` field, which is the process id of the process that sent the event.

- Event heartbeats are now calculated based on the time when the event was received by the monitor, and not the time reported by the worker.

    This means that a worker with an out-of-sync clock will no longer show as 'Offline' in monitors.

    A warning is now emitted if the difference between the senders time and the internal time is greater than 15 seconds, suggesting that the clocks are out of sync.

- Monotonic clock support.

    A monotonic clock is now used for timeouts and scheduling.

The monotonic clock function is built-in starting from Python 3.4, but we also have fallback implementations for Linux and macOS.

- **celery worker** now supports a new *--detach* argument to start the worker as a daemon in the background.

- app.events.Receiver now sets a local_received field for incoming events, which is set to the time of when the event was received.

- app.events.Dispatcher now accepts a groups argument which decides a white-list of event groups that'll be sent.

    The type of an event is a string separated by '-', where the part before the first '-' is the group. Currently there are only two groups: worker and task.

    A dispatcher instantiated as follows:

    ```
    >>> app.events.Dispatcher(connection, groups=['worker'])
    ```

    will only send worker related events and silently drop any attempts to send events related to any other group.

- New BROKER_FAILOVER_STRATEGY setting.

    This setting can be used to change the transport fail-over strategy, can either be a callable returning an iterable or the name of a Kombu built-in failover strategy. Default is "round-robin".

    Contributed by Matt Wise.

- Result.revoke will no longer wait for replies.

    You can add the reply=True argument if you really want to wait for responses from the workers.

- Better support for link and link_error tasks for chords.

    Contributed by Steeve Morin.

- Worker: Now emits warning if the CELERYD_POOL setting is set to enable the eventlet/gevent pools.

    The *-P* option should always be used to select the eventlet/gevent pool to ensure that the patches are applied as early as possible.

    If you start the worker in a wrapper (like Django's manage.py) then you must apply the patches manually, for example by creating an alternative wrapper that monkey patches at the start of the program before importing any other modules.

- There's a now an 'inspect clock' command which will collect the current logical clock value from workers.

- *celery inspect stats* now contains the process id of the worker's main process.

    Contributed by Mher Movsisyan.

- New remote control command to dump a workers configuration.

    Example:

    ```
    $ celery inspect conf
    ```

    Configuration values will be converted to values supported by JSON where possible.

    Contributed by Mher Movsisyan.

- New settings CELERY_EVENT_QUEUE_TTL and CELERY_EVENT_QUEUE_EXPIRES.

    These control when a monitors event queue is deleted, and for how long events published to that queue will be visible. Only supported on RabbitMQ.

- New Couchbase result backend.

  This result backend enables you to store and retrieve task results using Couchbase.

  See *Couchbase backend settings* for more information about configuring this result backend.

  Contributed by Alain Masiero.

- CentOS init-script now supports starting multiple worker instances.

  See the script header for details.

  Contributed by Jonathan Jordan.

- `AsyncResult.iter_native` now sets default interval parameter to 0.5

  Fix contributed by Idan Kamara

- New setting `BROKER_LOGIN_METHOD`.

  This setting can be used to specify an alternate login method for the AMQP transports.

  Contributed by Adrien Guinet

- The `dump_conf` remote control command will now give the string representation for types that aren't JSON compatible.

- Function *celery.security.setup_security* is now `app.setup_security()`.

- Task retry now propagates the message expiry value (Issue #980).

  The value is forwarded at is, so the expiry time won't change. To update the expiry time you'd've to pass a new expires argument to `retry()`.

- Worker now crashes if a channel error occurs.

  Channel errors are transport specific and is the list of exceptions returned by `Connection.channel_errors`. For RabbitMQ this means that Celery will crash if the equivalence checks for one of the queues in `CELERY_QUEUES` mismatches, which makes sense since this is a scenario where manual intervention is required.

- Calling `AsyncResult.get()` on a chain now propagates errors for previous tasks (Issue #1014).

- The parent attribute of `AsyncResult` is now reconstructed when using JSON serialization (Issue #1014).

- Worker disconnection logs are now logged with severity warning instead of error.

  Contributed by Chris Adams.

- `events.State` no longer crashes when it receives unknown event types.

- SQLAlchemy Result Backend: New `CELERY_RESULT_DB_TABLENAMES` setting can be used to change the name of the database tables used.

  Contributed by Ryan Petrello.

- **SQLAlchemy Result Backend: Now calls `enginge.dispose` after fork** (Issue #1564).

  If you create your own SQLAlchemy engines then you must also make sure that these are closed after fork in the worker:

  ```
  from multiprocessing.util import register_after_fork

  engine = create_engine(*engine_args)
  register_after_fork(engine, engine.dispose)
  ```

- A stress test suite for the Celery worker has been written.

This is located in the `funtests/stress` directory in the git repository. There's a README file there to get you started.

- The logger named `celery.concurrency` has been renamed to `celery.pool`.

- New command line utility `celery graph`.

  This utility creates graphs in GraphViz dot format.

  You can create graphs from the currently installed bootsteps:

```
# Create graph of currently installed bootsteps in both the worker
# and consumer name-spaces.
$ celery graph bootsteps | dot -T png -o steps.png

# Graph of the consumer name-space only.
$ celery graph bootsteps consumer | dot -T png -o consumer_only.png

# Graph of the worker name-space only.
$ celery graph bootsteps worker | dot -T png -o worker_only.png
```

  Or graphs of workers in a cluster:

```
# Create graph from the current cluster
$ celery graph workers | dot -T png -o workers.png

# Create graph from a specified list of workers
$ celery graph workers nodes:w1,w2,w3 | dot -T png workers.png

# also specify the number of threads in each worker
$ celery graph workers nodes:w1,w2,w3 threads:2,4,6

# ...also specify the broker and backend URLs shown in the graph
$ celery graph workers broker:amqp:// backend:redis://

# ...also specify the max number of workers/threads shown (wmax/tmax),
# enumerating anything that exceeds that number.
$ celery graph workers wmax:10 tmax:3
```

- Changed the way that app instances are pickled.

  Apps can now define a `__reduce_keys__` method that's used instead of the old `AppPickler` attribute. For example, if your app defines a custom 'foo' attribute that needs to be preserved when pickling you can define a `__reduce_keys__` as such:

```python
import celery

class Celery(celery.Celery):

    def __init__(self, *args, **kwargs):
        super(Celery, self).__init__(*args, **kwargs)
        self.foo = kwargs.get('foo')

    def __reduce_keys__(self):
        return super(Celery, self).__reduce_keys__().update(
            foo=self.foo,
        )
```

  This is a much more convenient way to add support for pickling custom attributes. The old `AppPickler` is still supported but its use is discouraged and we would like to remove it in a

> future version.

- Ability to trace imports for debugging purposes.

  > The C_IMPDEBUG can be set to trace imports as they occur:

  ```
  $ C_IMPDEBUG=1 celery worker -l info
  ```

  ```
  $ C_IMPDEBUG=1 celery shell
  ```

- Message headers now available as part of the task request.

  > Example adding and retrieving a header value:

  ```python
  @app.task(bind=True)
  def t(self):
      return self.request.headers.get('sender')

  >>> t.apply_async(headers={'sender': 'George Costanza'})
  ```

- New *before_task_publish* signal dispatched before a task message is sent and can be used to modify the final message fields (Issue #1281).

- New *after_task_publish* signal replaces the old *task_sent* signal.

  > The *task_sent* signal is now deprecated and shouldn't be used.

- New *worker_process_shutdown* signal is dispatched in the prefork pool child processes as they exit.

  > Contributed by Daniel M Taub.

- celery.platforms.PIDFile renamed to *celery.platforms.Pidfile*.

- MongoDB Backend: Can now be configured using a URL:

- MongoDB Backend: No longer using deprecated pymongo.Connection.

- MongoDB Backend: Now disables auto_start_request.

- MongoDB Backend: Now enables use_greenlets when eventlet/gevent is used.

- subtask() / maybe_subtask() renamed to signature()/maybe_signature().

  > Aliases still available for backwards compatibility.

- The correlation_id message property is now automatically set to the id of the task.

- The task message eta and expires fields now includes timezone information.

- All result backends store_result/mark_as_* methods must now accept a request keyword argument.

- Events now emit warning if the broken yajl library is used.

- The *celeryd_init* signal now takes an extra keyword argument: option.

  > This is the mapping of parsed command line arguments, and can be used to prepare new preload arguments (app.user_options['preload']).

- New callback: *app.on_configure()*.

  > This callback is called when an app is about to be configured (a configuration key is required).

- Worker: No longer forks on HUP.

This means that the worker will reuse the same pid for better support with external process supervisors.

Contributed by Jameel Al-Aziz.

- Worker: The log message `Got task from broker ...` was changed to `Received task ...`.

- Worker: The log message `Skipping revoked task ...` was changed to `Discarding revoked task ...`.

- Optimization: Improved performance of `ResultSet.join_native()`.

    Contributed by Stas Rudakou.

- The `task_revoked` signal now accepts new `request` argument (Issue #1555).

    The revoked signal is dispatched after the task request is removed from the stack, so it must instead use the `Request` object to get information about the task.

- Worker: New `-X` command line argument to exclude queues (Issue #1399).

    The `-X` argument is the inverse of the `-Q` argument and accepts a list of queues to exclude (not consume from):

    ```
    # Consume from all queues in CELERY_QUEUES, but not the 'foo' queue.
    $ celery worker -A proj -l info -X foo
    ```

- Adds `C_FAKEFORK` environment variable for simple init-script/**celery multi** debugging.

    This means that you can now do:

    ```
    $ C_FAKEFORK=1 celery multi start 10
    ```

    or:

    ```
    $ C_FAKEFORK=1 /etc/init.d/celeryd start
    ```

    to avoid the daemonization step to see errors that aren't visible due to missing stdout/stderr.

    A `dryrun` command has been added to the generic init-script that enables this option.

- New public API to push and pop from the current task stack:

    `celery.app.push_current_task()` and `celery.app.pop_current_task`().

- `RetryTaskError` has been renamed to `Retry`.

    The old name is still available for backwards compatibility.

- New semi-predicate exception `Reject`.

    This exception can be raised to `reject`/requeue the task message, see *Reject* for examples.

- *Semipredicates* documented: (Retry/Ignore/Reject).

Scheduled Removals

- The `BROKER_INSIST` setting and the `insist` argument to `~@connection` is no longer supported.

- The `CELERY_AMQP_TASK_RESULT_CONNECTION_MAX` setting is no longer supported.

    Use `BROKER_POOL_LIMIT` instead.

- The `CELERY_TASK_ERROR_WHITELIST` setting is no longer supported.

You should set the `ErrorMail` attribute of the task class instead. You can also do this using `CELERY_ANNOTATIONS`:

```python
from celery import Celery
from celery.utils.mail import ErrorMail

class MyErrorMail(ErrorMail):
    whitelist = (KeyError, ImportError)

    def should_send(self, context, exc):
        return isinstance(exc, self.whitelist)

app = Celery()
app.conf.CELERY_ANNOTATIONS = {
    '*': {
        'ErrorMail': MyErrorMails,
    }
}
```

- Functions that creates a broker connections no longer supports the `connect_timeout` argument.

    This can now only be set using the `BROKER_CONNECTION_TIMEOUT` setting. This is because functions no longer create connections directly, but instead get them from the connection pool.

- The `CELERY_AMQP_TASK_RESULT_EXPIRES` setting is no longer supported.

    Use `CELERY_TASK_RESULT_EXPIRES` instead.

Deprecation Time-line Changes

See the *Celery Deprecation Time-line*.

Fixes

- AMQP Backend: join didn't convert exceptions when using the json serializer.

- Non-abstract task classes are now shared between apps (Issue #1150).

    Note that non-abstract task classes shouldn't be used in the new API. You should only create custom task classes when you use them as a base class in the `@task` decorator.

    This fix ensure backwards compatibility with older Celery versions so that non-abstract task classes works even if a module is imported multiple times so that the app is also instantiated multiple times.

- Worker: Workaround for Unicode errors in logs (Issue #427).

- Task methods: `.apply_async` now works properly if args list is None (Issue #1459).

- Eventlet/gevent/solo/threads pools now properly handles `BaseException` errors raised by tasks.

- `autoscale` and `pool_grow`/`pool_shrink` remote control commands will now also automatically increase and decrease the consumer prefetch count.

    Fix contributed by Daniel M. Taub.

- `celery control pool_` commands didn't coerce string arguments to int.

- Redis/Cache chords: Callback result is now set to failure if the group disappeared from the database (Issue #1094).

- Worker: Now makes sure that the shutdown process isn't initiated more than once.

- Programs: **celery multi** now properly handles both `-f` and `--logfile` options (Issue #1541).

Internal changes

- Module `celery.task.trace` has been renamed to *celery.app.trace*.
- Module `celery.concurrency.processes` has been renamed to *celery.concurrency.prefork*.
- Classes that no longer fall back to using the default app:

  - Result backends (*celery.backends.base.BaseBackend*)
  - *celery.worker.WorkController*
  - `celery.worker.Consumer`
  - *celery.worker.request.Request*

  This means that you have to pass a specific app when instantiating these classes.

- `EventDispatcher.copy_buffer` renamed to `app.events.Dispatcher.extend_buffer()`.
- Removed unused and never documented global instance `celery.events.state.state`.
- `app.events.Receiver` is now a `kombu.mixins.ConsumerMixin` subclass.
- *celery.apps.worker.Worker* has been refactored as a subclass of *celery.worker.WorkController*.

  This removes a lot of duplicate functionality.

- The `Celery.with_default_connection` method has been removed in favor of `with app.connection_or_acquire` (*app.connection_or_acquire()*)
- The `celery.results.BaseDictBackend` class has been removed and is replaced by `celery.results.BaseBackend`.

3.13.8 Change history

This document contains change notes for bugfix releases in the 3.1.x series (Cipater), please see *What's new in Celery 3.1 (Cipater)* for an overview of what's new in Celery 3.1.

3.1.26

release-date 2018-23-03 16:00 PM IST

release-by Omer Katz

- Fixed a crash caused by tasks cycling between Celery 3 and Celery 4 workers.

3.1.25

release-date 2016-10-10 12:00 PM PDT

release-by Ask Solem

- **Requirements**

  - Now depends on Kombu 3.0.37

- Fixed problem with chords in group introduced in 3.1.24 (Issue #3504).

3.1.24

release-date 2016-09-30 04:21 PM PDT

release-by Ask Solem

- **Requirements**

  – Now depends on [Kombu 3.0.36](#).

- Now supports Task protocol 2 from the future 4.0 release.

    Workers running 3.1.24 are now able to process messages sent using the [new task message protocol](#) to be introduced in Celery 4.0.

    Users upgrading to Celery 4.0 when this is released are encouraged to upgrade to this version as an intermediate step, as this means workers not yet upgraded will be able to process messages from clients/workers running 4.0.

- `Task.send_events` can now be set to disable sending of events for that task only.

    Example when defining the task:

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

- **Utils**: Fixed compatibility with recent [psutil](#) versions (Issue #3262).

- **Canvas**: Chord now forwards partial arguments to its subtasks.

    Fix contributed by Tayfun Sen.

- **App**: Arguments to app such as `backend`, `broker`, etc are now pickled and sent to the child processes on Windows.

    Fix contributed by Jeremy Zafran.

- **Deployment**: Generic init scripts now supports being symlinked in runlevel directories (Issue #3208).

- **Deployment**: Updated CentOS scripts to work with CentOS 7.

    Contributed by Joe Sanford.

- **Events**: The curses monitor no longer crashes when the result of a task is empty.

    Fix contributed by Dongweiming.

- **Worker**: `repr(worker)` would crash when called early in the startup process (Issue #2514).

- **Tasks**: GroupResult now defines __bool__ and __nonzero__.

    This is to fix an issue where a ResultSet or GroupResult with an empty result list are not properly tupled with the as_tuple() method when it is a parent result. This is due to the as_tuple() method performing a logical and operation on the ResultSet.

    Fix contributed by Colin McIntosh.

- **Worker**: Fixed wrong values in autoscale related logging message.

    Fix contributed by `@raducc`.

- Documentation improvements by

  – Alexandru Chirila

  – Michael Aquilina

- Mikko Ekström

- Mitchel Humpherys

- Thomas A. Neil

- Tiago Moreira Vieira

- Yuriy Syrovetskiy

- `@dessant`

3.1.23

release-date 2016-03-09 06:00 P.M PST

release-by Ask Solem

- **Programs**: Last release broke support for the `--hostnmame` argument to **celery multi** and **celery worker --detach** (Issue #3103).

- **Results**: MongoDB result backend could crash the worker at startup if not configured using an URL.

3.1.22

release-date 2016-03-07 01:30 P.M PST

release-by Ask Solem

- **Programs**: The worker would crash immediately on startup on `backend.as_uri()` when using some result backends (Issue #3094).

- **Programs**: **celery multi/celery worker --detach** would create an extraneous logfile including literal formats (e.g. `%I`) in the filename (Issue #3096).

3.1.21

release-date 2016-03-04 11:16 a.m. PST

release-by Ask Solem

- **Requirements**

    - Now depends on Kombu 3.0.34.

    - Now depends on `billiard` 3.3.0.23.

- **Prefork pool**: Fixes 100% CPU loop on Linux *epoll* (Issue #1845).

    Also potential fix for: Issue #2142, Issue #2606

- **Prefork pool**: Fixes memory leak related to processes exiting (Issue #2927).

- **Worker**: Fixes crash at start-up when trying to censor passwords in MongoDB and Cache result backend URLs (Issue #3079, Issue #3045, Issue #3049, Issue #3068, Issue #3073).

    Fix contributed by Maxime Verger.

- **Task**: An exception is now raised if countdown/expires is less than -2147483648 (Issue #3078).

- **Programs**: **celery shell --ipython** now compatible with newer IPython versions.

- **Programs**: The DuplicateNodeName warning emitted by inspect/control now includes a list of the node names returned.

    Contributed by Sebastian Kalinowski.

- **Utils**: The `.discard(item)` method of *LimitedSet* didn't actually remove the item (Issue #3087).

    Fix contributed by Dave Smith.

- **Worker**: Node name formatting now emits less confusing error message for unmatched format keys (Issue #3016).

- **Results**: RPC/AMQP backends: Fixed deserialization of JSON exceptions (Issue #2518).

    Fix contributed by Allard Hoeve.

- **Prefork pool**: The *process inqueue damaged* error message now includes the original exception raised.

- **Documentation**: Includes improvements by:

    – Jeff Widman.

3.1.20

release-date 2016-01-22 06:50 p.m. UTC

release-by Ask Solem

- **Requirements**

    – Now depends on Kombu 3.0.33.

    – Now depends on `billiard` 3.3.0.22.

        Includes binary wheels for Microsoft Windows x86 and x86_64!

- **Task**: Error emails now uses `utf-8` character set by default (Issue #2737).

- **Task**: Retry now forwards original message headers (Issue #3017).

- **Worker**: Bootsteps can now hook into `on_node_join`/`leave`/`lost`.

    See *Attributes* for an example.

- **Events**: Fixed handling of DST timezones (Issue #2983).

- **Results**: Redis backend stopped respecting certain settings.

    Contributed by Jeremy Llewellyn.

- **Results**: Database backend now properly supports JSON exceptions (Issue #2441).

- **Results**: Redis `new_join` didn't properly call task errbacks on chord error (Issue #2796).

- **Results**: Restores Redis compatibility with Python redis < 2.10.0 (Issue #2903).

- **Results**: Fixed rare issue with chord error handling (Issue #2409).

- **Tasks**: Using queue-name values in `CELERY_ROUTES` now works again (Issue #2987).

- **General**: Result backend password now sanitized in report output (Issue #2812, Issue #2004).

- **Configuration**: Now gives helpful error message when the result backend configuration points to a module, and not a class (Issue #2945).

- **Results**: Exceptions sent by JSON serialized workers are now properly handled by pickle configured workers.

- **Programs**: `celery control autoscale` now works (Issue #2950).

- **Programs**: `celery beat --detached` now runs after fork callbacks.
- **General**: Fix for LRU cache implementation on Python 3.5 (Issue #2897).

  Contributed by Dennis Brakhane.

  Python 3.5's `OrderedDict` doesn't allow mutation while it is being iterated over. This breaks "update" if it is called with a dict larger than the maximum size.

  This commit changes the code to a version that doesn't iterate over the dict, and should also be a little bit faster.
- **Init-scripts**: The beat init-script now properly reports service as down when no pid file can be found.

  Eric Zarowny
- **Beat**: Added cleaning of corrupted scheduler files for some storage backend errors (Issue #2985).

  Fix contributed by Aleksandr Kuznetsov.
- **Beat**: Now syncs the schedule even if the schedule is empty.

  Fix contributed by Colin McIntosh.
- **Supervisord: Set higher process priority in the** supervisord **example.**

  Contributed by George Tantiras.
- **Documentation**: Includes improvements by:

  @Bryson Caleb Mingle Christopher Martin Dieter Adriaenssens Jason Veatch Jeremy Cline Juan Rossi Kevin Harvey Kevin McCarthy Kirill Pavlov Marco Buttu @Mayflower Mher Movsisyan Michael Floering @michael-k Nathaniel Varona Rudy Attias Ryan Luckie Steven Parker @squfrans Tadej Janež TakesxiSximada Tom S

3.1.19

release-date 2015-10-26 01:00 p.m. UTC

release-by Ask Solem

- **Requirements**

  - Now depends on Kombu 3.0.29.
  - Now depends on `billiard` 3.3.0.21.
- **Results**: Fixed MongoDB result backend URL parsing problem (Issue celery/kombu#375).
- **Worker**: Task request now properly sets `priority` in delivery_info.

  Fix contributed by Gerald Manipon.
- **Beat**: PyPy shelve may raise `KeyError` when setting keys (Issue #2862).
- **Programs**: `celery beat --deatched` now working on PyPy.

  Fix contributed by Krzysztof Bujniewicz.
- **Results**: Redis result backend now ensures all pipelines are cleaned up.

  Contributed by Justin Patrin.
- **Results**: Redis result backend now allows for timeout to be set in the query portion of the result backend URL.

  For example `CELERY_RESULT_BACKEND = 'redis://?timeout=10'`

  Contributed by Justin Patrin.

- **Results**: `result.get` now properly handles failures where the exception value is set to `None` (Issue #2560).

- **Prefork pool**: Fixed attribute error `proc.dead`.

- **Worker**: Fixed worker hanging when gossip/heartbeat disabled (Issue #1847).

    Fix contributed by Aaron Webber and Bryan Helmig.

- **Results**: MongoDB result backend now supports pymongo 3.x (Issue #2744).

    Fix contributed by Sukrit Khera.

- **Results**: RPC/AMQP backends didn't deserialize exceptions properly (Issue #2691).

    Fix contributed by Sukrit Khera.

- **Programs**: Fixed problem with **celery amqp**'s `basic_publish` (Issue #2013).

- **Worker**: Embedded beat now properly sets app for thread/process (Issue #2594).

- **Documentation**: Many improvements and typos fixed.

    Contributions by:

    > Carlos Garcia-Dubus D. Yu @jerry Jocelyn Delalande Josh Kupershmidt Juan Rossi
    > @kanemra Paul Pearce Pavel Savchenko Sean Wang Seungha Kim Zhaorong Ma

3.1.18

release-date 2015-04-22 05:30 p.m. UTC

release-by Ask Solem

- **Requirements**

    - Now depends on Kombu 3.0.25.

    - Now depends on `billiard` 3.3.0.20.

- **Django**: Now supports Django 1.8 (Issue #2536).

    Fix contributed by Bence Tamas and Mickaël Penhard.

- **Results**: MongoDB result backend now compatible with pymongo 3.0.

    Fix contributed by Fatih Sucu.

- **Tasks**: Fixed bug only happening when a task has multiple callbacks (Issue #2515).

    Fix contributed by NotSqrt.

- **Commands**: Preload options now support `--arg value` syntax.

    Fix contributed by John Anderson.

- **Compat**: A typo caused `celery.log.setup_logging_subsystem` to be undefined.

    Fix contributed by Gunnlaugur Thor Briem.

- **init-scripts**: The beat generic init-script now uses `/bin/sh` instead of **bash** (Issue #2496).

    Fix contributed by Jelle Verstraaten.

- **Django**: Fixed a `TypeError` sometimes occurring in logging when validating models.

    Fix contributed by Alexander.

- **Commands**: Worker now supports new `--executable` argument that can be used with `celery worker --detach`.

Contributed by Bert Vanderbauwhede.

- **Canvas**: Fixed crash in chord unlock fallback task (Issue #2404).

- **Worker**: Fixed rare crash occurring with `--autoscale` enabled (Issue #2411).

- **Django**: Properly recycle worker Django database connections when the Django `CONN_MAX_AGE` setting is enabled (Issue #2453).

  Fix contributed by Luke Burden.

3.1.17

release-date 2014-11-19 03:30 p.m. UTC

release-by Ask Solem

Don't enable the *CELERYD_FORCE_EXECV* setting!

Please review your configuration and disable this option if you're using the RabbitMQ or Redis transport.

Keeping this option enabled after 3.1 means the async based prefork pool will be disabled, which can easily cause instability.

- **Requirements**

  - Now depends on Kombu 3.0.24.

    Includes the new Qpid transport coming in Celery 3.2, backported to support those who may still require Python 2.6 compatibility.

  - Now depends on `billiard` 3.3.0.19.

  - `celery[librabbitmq]` now depends on librabbitmq 1.6.1.

- **Task**: The timing of ETA/countdown tasks were off after the example `LocalTimezone` implementation in the Python documentation no longer works in Python 3.4. (Issue #2306).

- **Task**: Raising *Ignore* no longer sends `task-failed` event (Issue #2365).

- **Redis result backend**: Fixed unbound local errors.

  Fix contributed by Thomas French.

- **Task**: Callbacks wasn't called properly if `link` was a list of signatures (Issue #2350).

- **Canvas**: chain and group now handles json serialized signatures (Issue #2076).

- **Results**: `.join_native()` would accidentally treat the `STARTED` state as being ready (Issue #2326).

  This could lead to the chord callback being called with invalid arguments when using chords with the `CELERY_TRACK_STARTED` setting enabled.

- **Canvas**: The `chord_size` attribute is now set for all canvas primitives, making sure more combinations will work with the `new_join` optimization for Redis (Issue #2339).

- **Task**: Fixed problem with app not being properly propagated to `trace_task` in all cases.

  Fix contributed by @kristaps.

- **Worker**: Expires from task message now associated with a timezone.

  Fix contributed by Albert Wang.

- **Cassandra result backend**: Fixed problems when using detailed mode.

When using the Cassandra backend in detailed mode, a regression caused errors when attempting to retrieve results.

Fix contributed by Gino Ledesma.

- **Mongodb Result backend**: Pickling the backend instance will now include the original URL (Issue #2347).

    Fix contributed by Sukrit Khera.

- **Task**: Exception info wasn't properly set for tasks raising *Reject* (Issue #2043).

- **Worker**: Duplicates are now removed when loading the set of revoked tasks from the worker state database (Issue #2336).

- **celery.contrib.rdb**: Fixed problems with `rdb.set_trace` calling stop from the wrong frame.

    Fix contributed by @llllllllll.

- **Canvas**: `chain` and `chord` can now be immutable.

- **Canvas**: `chord.apply_async` will now keep partial args set in `self.args` (Issue #2299).

- **Results**: Small refactoring so that results are decoded the same way in all result backends.

- **Logging**: The `processName` format was introduced in Python 2.6.2 so for compatibility this format is now excluded when using earlier versions (Issue #1644).

3.1.16

release-date 2014-10-03 06:00 p.m. UTC

release-by Ask Solem

- **Worker**: 3.1.15 broke *-Ofair* behavior (Issue #2286).

    This regression could result in all tasks executing in a single child process if `-Ofair` was enabled.

- **Canvas**: `celery.signature` now properly forwards app argument in all cases.

- **Task**: `.retry()` didn't raise the exception correctly when called without a current exception.

    Fix contributed by Andrea Rabbaglietti.

- **Worker**: The `enable_events` remote control command disabled worker-related events by mistake (Issue #2272).

    Fix contributed by Konstantinos Koukopoulos.

- **Django**: Adds support for Django 1.7 class names in INSTALLED_APPS when using `app.autodiscover_tasks()` (Issue #2248).

- **Sphinx**: `celery.contrib.sphinx` now uses `getfullargspec` on Python 3 (Issue #2302).

- **Redis/Cache Backends**: Chords will now run at most once if one or more tasks in the chord are executed multiple times for some reason.

3.1.15

release-date 2014-09-14 11:00 p.m. UTC

release-by Ask Solem

- **Django**: Now makes sure `django.setup()` is called before importing any task modules (Django 1.7 compatibility, Issue #2227)

- **Results**: `result.get()` was misbehaving by calling `backend.get_task_meta` in a `finally` call leading to AMQP result backend queues not being properly cleaned up (Issue #2245).

3.1.14

release-date 2014-09-08 03:00 p.m. UTC

release-by Ask Solem

- **Requirements**

    - Now depends on Kombu 3.0.22.

- **Init-scripts**: The generic worker init-scripts `status` command now gets an accurate pidfile list (Issue #1942).

- **Init-scripts: The generic beat script now implements the `status` command.**

    Contributed by John Whitlock.

- **Commands**: Multi now writes informational output to stdout instead of stderr.

- **Worker**: Now ignores not implemented error for `pool.restart` (Issue #2153).

- **Task**: Retry no longer raises retry exception when executed in eager mode (Issue #2164).

- **AMQP Result backend**: Now ensured `on_interval` is called at least every second for blocking calls to properly propagate parent errors.

- **Django**: Compatibility with Django 1.7 on Windows (Issue #2126).

- **Programs**: `--umask` argument can now be specified in both octal (if starting with 0) or decimal.

3.1.13

Security Fixes

- [Security: CELERYSA-0002] Insecure default umask.

    The built-in utility used to daemonize the Celery worker service sets an insecure umask by default (umask 0).

    This means that any files or directories created by the worker will end up having world-writable permissions.

    Special thanks to Red Hat for originally discovering and reporting the issue!

    This version will no longer set a default umask by default, so if unset the umask of the parent process will be used.

News

- **Requirements**

    - Now depends on Kombu 3.0.21.

    - Now depends on `billiard` 3.3.0.18.

- **App**: `backend` argument now also sets the `CELERY_RESULT_BACKEND` setting.

- **Task**: `signature_from_request` now propagates `reply_to` so that the RPC backend works with retried tasks (Issue #2113).

- **Task**: `retry` will no longer attempt to re-queue the task if sending the retry message fails.

    Unrelated exceptions being raised could cause a message loop, so it was better to remove this behavior.

- **Beat**: Accounts for standard 1ms drift by always waking up 0.010s earlier.

    This will adjust the latency so that the periodic tasks won't move 1ms after every invocation.

- Documentation fixes

    Contributed by Yuval Greenfield, Lucas Wiman, @nicholsonjf.

- **Worker**: Removed an outdated assert statement that could lead to errors being masked (Issue #2086).

3.1.12

release-date 2014-06-09 10:12 p.m. UTC

release-by Ask Solem

- **Requirements**

    Now depends on Kombu 3.0.19.

- **App**: Connections weren't being closed after fork due to an error in the after fork handler (Issue #2055).

    This could manifest itself by causing framing errors when using RabbitMQ. (`Unexpected frame`).

- **Django**: `django.setup()` was being called too late when using Django 1.7 (Issue #1802).

- **Django**: Fixed problems with event timezones when using Django (`Substantial drift`).

    Celery didn't take into account that Django modifies the `time.timeone` attributes and friends.

- **Canvas**: `Signature.link` now works when the link option is a scalar value (Issue #2019).

- **Prefork pool**: Fixed race conditions for when file descriptors are removed from the event loop.

    Fix contributed by Roger Hu.

- **Prefork pool**: Improved solution for dividing tasks between child processes.

    This change should improve performance when there are many child processes, and also decrease the chance that two subsequent tasks are written to the same child process.

- **Worker**: Now ignores unknown event types, instead of crashing.

    Fix contributed by Illes Solt.

- **Programs**: **celery worker --detach** no longer closes open file descriptors when `C_FAKEFORK` is used so that the workers output can be seen.

- **Programs**: The default working directory for **celery worker --detach** is now the current working directory, not `/`.

- **Canvas**: `signature(s, app=app)` didn't upgrade serialized signatures to their original class (`subtask_type`) when the `app` keyword argument was used.

- **Control**: The `duplicate nodename` warning emitted by control commands now shows the duplicate node name.

- **Tasks**: Can now call `ResultSet.get()` on a result set without members.

    Fix contributed by Alexey Kotlyarov.

- **App**: Fixed strange traceback mangling issue for `app.connection_or_acquire`.

- **Programs**: The **`celery multi stopwait`** command is now documented in usage.

- **Other**: Fixed cleanup problem with `PromiseProxy` when an error is raised while trying to evaluate the promise.

- **Other**: The utility used to censor configuration values now handles non-string keys.

    Fix contributed by Luke Pomfrey.

- **Other**: The `inspect conf` command didn't handle non-string keys well.

    Fix contributed by Jay Farrimond.

- **Programs**: Fixed argument handling problem in **`celery worker --detach`**.

    Fix contributed by Dmitry Malinovsky.

- **Programs**: **`celery worker --detach`** didn't forward working directory option (Issue #2003).

- **Programs**: **`celery inspect registered`** no longer includes the list of built-in tasks.

- **Worker**: The `requires` attribute for boot steps weren't being handled correctly (Issue #2002).

- **Eventlet**: The eventlet pool now supports the `pool_grow` and `pool_shrink` remote control commands.

    Contributed by Mher Movsisyan.

- **Eventlet**: The eventlet pool now implements statistics for :program:`celery inspect stats`.

    Contributed by Mher Movsisyan.

- **Documentation**: Clarified `Task.rate_limit` behavior.

    Contributed by Jonas Haag.

- **Documentation**: `AbortableTask` examples now updated to use the new API (Issue #1993).

- **Documentation**: The security documentation examples used an out of date import.

    Fix contributed by Ian Dees.

- **Init-scripts**: The CentOS init-scripts didn't quote `CELERY_CHDIR`.

    Fix contributed by [@ffeast](#).

3.1.11

release-date 2014-04-16 11:00 p.m. UTC

release-by Ask Solem

- **Now compatible with RabbitMQ 3.3.0**

    You need to run Celery 3.1.11 or later when using RabbitMQ 3.3, and if you use the `librabbitmq` module you also have to upgrade to librabbitmq 1.5.0:

    ```
    $ pip install -U librabbitmq
    ```

- **Requirements**:

    – Now depends on [Kombu 3.0.15](#).

    – Now depends on [billiard 3.3.0.17](#).

    – Bundle `celery[librabbitmq]` now depends on `librabbitmq` 1.5.0.

- **Tasks**: The `CELERY_DEFAULT_DELIVERY_MODE` setting was being ignored (Issue #1953).

- **Worker**: New *`celery worker --heartbeat-interval`* can be used to change the time (in seconds) between sending event heartbeats.

  Contributed by Matthew Duggan and Craig Northway.

- **App**: Fixed memory leaks occurring when creating lots of temporary app instances (Issue #1949).

- **MongoDB**: SSL configuration with non-MongoDB transport breaks MongoDB results backend (Issue #1973).

  Fix contributed by Brian Bouterse.

- **Logging**: The color formatter accidentally modified `record.msg` (Issue #1939).

- **Results**: Fixed problem with task trails being stored multiple times, causing `result.collect()` to hang (Issue #1936, Issue #1943).

- **Results**: `ResultSet` now implements a `.backend` attribute for compatibility with `AsyncResult`.

- **Results**: `.forget()` now also clears the local cache.

- **Results**: Fixed problem with multiple calls to `result._set_cache` (Issue #1940).

- **Results**: `join_native` populated result cache even if disabled.

- **Results**: The YAML result serializer should now be able to handle storing exceptions.

- **Worker**: No longer sends task error emails for expected errors (in `@task(throws=(..., ))`).

- **Canvas**: Fixed problem with exception deserialization when using the JSON serializer (Issue #1987).

- **Eventlet**: Fixes crash when `celery.contrib.batches` attempted to cancel a non-existing timer (Issue #1984).

- Can now import `celery.version_info_t`, and `celery.five` (Issue #1968).

3.1.10

release-date 2014-03-22 09:40 p.m. UTC

release-by Ask Solem

- **Requirements**:

  - Now depends on Kombu 3.0.14.

- **Results**:

  Reliability improvements to the SQLAlchemy database backend. Previously the connection from the MainProcess was improperly shared with the workers. (Issue #1786)

- **Redis:** Important note about events (Issue #1882).

  There's a new transport option for Redis that enables monitors to filter out unwanted events. Enabling this option in the workers will increase performance considerably:

  ```
  BROKER_TRANSPORT_OPTIONS = {'fanout_patterns': True}
  ```

  Enabling this option means that your workers won't be able to see workers with the option disabled (or is running an older version of Celery), so if you do enable it then make sure you do so on all nodes.

  See *Caveats*.

  This will be the default in Celery 3.2.

- **Results**: The *app.AsyncResult* object now keeps a local cache of the final state of the task.

    This means that the global result cache can finally be disabled, and you can do so by setting CELERY_MAX_CACHED_RESULTS to -1. The lifetime of the cache will then be bound to the lifetime of the result object, which will be the default behavior in Celery 3.2.

- **Events**: The "Substantial drift" warning message is now logged once per node name only (Issue #1802).

- **Worker**: Ability to use one log file per child process when using the prefork pool.

    This can be enabled by using the new %i and %I format specifiers for the log file name. See *Prefork pool process index*.

- **Redis**: New experimental chord join implementation.

    This is an optimization for chords when using the Redis result backend, where the join operation is now considerably faster and using less resources than the previous strategy.

    The new option can be set in the result backend URL:

    ```
    CELERY_RESULT_BACKEND = 'redis://localhost?new_join=1'
    ```

    This must be enabled manually as it's incompatible with workers and clients not using it, so be sure to enable the option in all clients and workers if you decide to use it.

- **Multi**: With -opt:index (e.g., -c:1) the index now always refers to the position of a node in the argument list.

    This means that referring to a number will work when specifying a list of node names and not just for a number range:

    ```
    celery multi start A B C D -c:1 4 -c:2-4 8
    ```

    In this example 1 refers to node A (as it's the first node in the list).

- **Signals**: The sender argument to Signal.connect can now be a proxy object, which means that it can be used with the task decorator (Issue #1873).

- **Task**: A regression caused the queue argument to Task.retry to be ignored (Issue #1892).

- **App**: Fixed error message for *config_from_envvar()*.

    Fix contributed by Dmitry Malinovsky.

- **Canvas**: Chords can now contain a group of other chords (Issue #1921).

- **Canvas**: Chords can now be combined when using the amqp result backend (a chord where the callback is also a chord).

- **Canvas**: Calling result.get() for a chain task will now complete even if one of the tasks in the chain is ignore_result=True (Issue #1905).

- **Canvas**: Worker now also logs chord errors.

- **Canvas**: A chord task raising an exception will now result in any errbacks (link_error) to the chord callback to also be called.

- **Results**: Reliability improvements to the SQLAlchemy database backend (Issue #1786).

    Previously the connection from the MainProcess was improperly inherited by child processes.

    Fix contributed by Ionel Cristian Măries.

- **Task**: Task callbacks and errbacks are now called using the group primitive.

- **Task**: Task.apply now properly sets request.headers (Issue #1874).

- **Worker**: Fixed `UnicodeEncodeError` occurring when worker is started by supervisor.

    Fix contributed by Codeb Fan.

- **Beat**: No longer attempts to upgrade a newly created database file (Issue #1923).

- **Beat**: New setting :setting:`CELERYBEAT_SYNC_EVERY` can be be used to control file sync by specifying the number of tasks to send between each sync.

    Contributed by Chris Clark.

- **Commands**: **`celery inspect memdump`** no longer crashes if the `psutil` module isn't installed (Issue #1914).

- **Worker**: Remote control commands now always accepts json serialized messages (Issue #1870).

- **Worker**: Gossip will now drop any task related events it receives by mistake (Issue #1882).

3.1.9

release-date 2014-02-10 06:43 p.m. UTC

release-by Ask Solem

- **Requirements**:

    - Now depends on Kombu 3.0.12.

- **Prefork pool**: Better handling of exiting child processes.

    Fix contributed by Ionel Cristian Mărieș.

- **Prefork pool**: Now makes sure all file descriptors are removed from the hub when a process is cleaned up.

    Fix contributed by Ionel Cristian Mărieș.

- **New Sphinx extension**: for autodoc documentation of tasks: `celery.contrib.spinx` (Issue #1833).

- **Django**: Now works with Django 1.7a1.

- **Task**: Task.backend is now a property that forwards to `app.backend` if no custom backend has been specified for the task (Issue #1821).

- **Generic init-scripts**: Fixed bug in stop command.

    Fix contributed by Rinat Shigapov.

- **Generic init-scripts**: Fixed compatibility with GNU *stat*.

    Fix contributed by Paul Kilgo.

- **Generic init-scripts**: Fixed compatibility with the minimal **dash** shell (Issue #1815).

- **Commands**: The **`celery amqp basic.publish`** command wasn't working properly.

    Fix contributed by Andrey Voronov.

- **Commands**: Did no longer emit an error message if the pidfile exists and the process is still alive (Issue #1855).

- **Commands**: Better error message for missing arguments to preload options (Issue #1860).

- **Commands**: **`celery -h`** didn't work because of a bug in the argument parser (Issue #1849).

- **Worker**: Improved error message for message decoding errors.

- **Time**: Now properly parses the *Z* timezone specifier in ISO 8601 date strings.

    Fix contributed by Martin Davidsson.

- **Worker**: Now uses the *negotiated* heartbeat value to calculate how often to run the heartbeat checks.

- **Beat**: Fixed problem with beat hanging after the first schedule iteration (Issue #1822).

    Fix contributed by Roger Hu.

- **Signals**: The header argument to *before_task_publish* is now always a dictionary instance so that signal handlers can add headers.

- **Worker**: A list of message headers is now included in message related errors.

3.1.8

release-date 2014-01-17 10:45 p.m. UTC

release-by Ask Solem

- **Requirements**:

    - Now depends on Kombu 3.0.10.

    - Now depends on billiard 3.3.0.14.

- **Worker**: The event loop wasn't properly reinitialized at consumer restart which would force the worker to continue with a closed `epoll` instance on Linux, resulting in a crash.

- **Events:** Fixed issue with both heartbeats and task events that could result in the data not being kept in sorted order.

    As a result this would force the worker to log "heartbeat missed" events even though the remote node was sending heartbeats in a timely manner.

- **Results:** The pickle serializer no longer converts group results to tuples, and will keep the original type (*Issue #1750*).

- **Results:** `ResultSet.iterate` is now pending deprecation.

    The method will be deprecated in version 3.2 and removed in version 3.3.

    Use `result.get(callback=)` (or `result.iter_native()` where available) instead.

- **Worker**|eventlet/gevent: A regression caused `Control-c` to be ineffective for shutdown.

- **Redis result backend:** Now using a pipeline to store state changes for improved performance.

    Contributed by Pepijn de Vos.

- **Redis result backend:** Will now retry storing the result if disconnected.

- **Worker**|gossip: Fixed attribute error occurring when another node leaves.

    Fix contributed by Brodie Rao.

- **Generic init-scripts:** Now runs a check at start-up to verify that any configuration scripts are owned by root and that they aren't world/group writable.

    The init-script configuration is a shell script executed by root, so this is a preventive measure to ensure that users don't leave this file vulnerable to changes by unprivileged users.

    Note: Note that upgrading Celery won't update the init-scripts, instead you need to manually copy the improved versions from the source distribution: https://github.com/celery/celery/tree/3.1/extra/generic-init.d

    ---

- **Commands**: The **celery purge** command now warns that the operation will delete all tasks and prompts the user for confirmation.

  A new `-f` was added that can be used to disable interactive mode.

- **Task**: `.retry()` didn't raise the value provided in the `exc` argument when called outside of an error context (*Issue #1755*).

- **Commands:** The **celery multi** command didn't forward command line configuration to the target workers.

  The change means that multi will forward the special `--` argument and configuration content at the end of the arguments line to the specified workers.

  Example using command-line configuration to set a broker heartbeat from **celery multi**:

  ```
  $ celery multi start 1 -c3 -- broker.heartbeat=30
  ```

  Fix contributed by Antoine Legrand.

- **Canvas:** `chain.apply_async()` now properly forwards execution options.

  Fix contributed by Konstantin Podshumok.

- **Redis result backend:** Now takes `connection_pool` argument that can be used to change the connection pool class/constructor.

- **Worker:** Now truncates very long arguments and keyword arguments logged by the pool at debug severity.

- **Worker:** The worker now closes all open files on `SIGHUP` (regression) (*Issue #1768*).

  Fix contributed by Brodie Rao

- **Worker:** Will no longer accept remote control commands while the worker start-up phase is incomplete (*Issue #1741*).

- **Commands:** The output of the event dump utility (**celery events -d**) can now be piped into other commands.

- **Documentation:** The RabbitMQ installation instructions for macOS was updated to use modern Homebrew practices.

  Contributed by Jon Chen.

- **Commands:** The **celery inspect conf** utility now works.

- **Commands:** The `--no-color` argument was not respected by all commands (*Issue #1799*).

- **App:** Fixed rare bug with `autodiscover_tasks()` (*Issue #1797*).

- **Distribution:** The sphinx docs will now always add the parent directory to path so that the current Celery source code is used as a basis for API documentation (*Issue #1782*).

- **Documentation:** supervisor examples contained an extraneous '-' in a `--logfile` argument example.

  Fix contributed by Mohammad Almeer.

3.1.7

release-date 2013-12-17 06:00 p.m. UTC

release-by Ask Solem

Important Notes

Init-script security improvements

Where the generic init-scripts (for `celeryd`, and `celerybeat`) before delegated the responsibility of dropping privileges to the target application, it will now use `su` instead, so that the Python program isn't trusted with superuser privileges.

This isn't in reaction to any known exploit, but it will limit the possibility of a privilege escalation bug being abused in the future.

You have to upgrade the init-scripts manually from this directory: https://github.com/celery/celery/tree/3.1/extra/generic-init.d

AMQP result backend

The 3.1 release accidentally left the amqp backend configured to be non-persistent by default.

Upgrading from 3.0 would give a "not equivalent" error when attempting to set or retrieve results for a task. That's unless you manually set the persistence setting:

```
CELERY_RESULT_PERSISTENT = True
```

This version restores the previous value so if you already forced the upgrade by removing the existing exchange you must either keep the configuration by setting CELERY_RESULT_PERSISTENT = False or delete the `celeryresults` exchange again.

Synchronous subtasks

Tasks waiting for the result of a subtask will now emit a `RuntimeWarning` warning when using the prefork pool, and in 3.2 this will result in an exception being raised.

It's not legal for tasks to block by waiting for subtasks as this is likely to lead to resource starvation and eventually deadlock when using the prefork pool (see also *Avoid launching synchronous subtasks*).

If you really know what you're doing you can avoid the warning (and the future exception being raised) by moving the operation in a white-list block:

```python
from celery.result import allow_join_result

@app.task
def misbehaving():
    result = other_task.delay()
    with allow_join_result():
        result.get()
```

Note also that if you wait for the result of a subtask in any form when using the prefork pool you must also disable the pool prefetching behavior with the worker *-Ofair option*.

Fixes

- Now depends on Kombu 3.0.8.
- Now depends on `billiard` 3.3.0.13

- Events: Fixed compatibility with non-standard json libraries that sends float as `decimal.Decimal` (Issue #1731)

- Events: State worker objects now always defines attributes: `active`, `processed`, `loadavg`, `sw_ident`, `sw_ver` and `sw_sys`.

- Worker: Now keeps count of the total number of tasks processed, not just by type (`all_active_count`).

- Init-scripts: Fixed problem with reading configuration file when the init-script is symlinked to a runlevel (e.g., `S02celeryd`). (Issue #1740).

    This also removed a rarely used feature where you can symlink the script to provide alternative configurations. You instead copy the script and give it a new name, but perhaps a better solution is to provide arguments to CELERYD_OPTS to separate them:

    ```
    CELERYD_NODES="X1 X2 Y1 Y2"
    CELERYD_OPTS="-A:X1 x -A:X2 x -A:Y1 y -A:Y2 y"
    ```

- Fallback chord unlock task is now always called after the chord header (Issue #1700).

    This means that the unlock task won't be started if there's an error sending the header.

- Celery command: Fixed problem with arguments for some control commands.

    Fix contributed by Konstantin Podshumok.

- Fixed bug in `utcoffset` where the offset when in DST would be completely wrong (Issue #1743).

- Worker: Errors occurring while attempting to serialize the result of a task will now cause the task to be marked with failure and a `kombu.exceptions.EncodingError` error.

    Fix contributed by Ionel Cristian Mărieș.

- Worker with `-B` argument didn't properly shut down the beat instance.

- Worker: The `%n` and `%h` formats are now also supported by the `--logfile`, `--pidfile` and `--statedb` arguments.

    Example:

    ```
    $ celery -A proj worker -n foo@%h --logfile=%n.log --statedb=%n.db
    ```

- Redis/Cache result backends: Will now timeout if keys evicted while trying to join a chord.

- The fallback unlock chord task now raises `Retry` so that the retry even is properly logged by the worker.

- Multi: Will no longer apply Eventlet/gevent monkey patches (Issue #1717).

- Redis result backend: Now supports UNIX sockets.

    Like the Redis broker transport the result backend now also supports using `redis+socket://` `/tmp/redis.sock` URLs.

    Contributed by Alcides Viamontes Esquivel.

- Events: Events sent by clients was mistaken for worker related events (Issue #1714).

    For `events.State` the tasks now have a `Task.client` attribute that's set when a `task-sent` event is being received.

    Also, a clients logical clock isn't in sync with the cluster so they live in a "time bubble." So for this reason monitors will no longer attempt to merge with the clock of an event sent by a client, instead it will fake the value by using the current clock with a skew of -1.

- Prefork pool: The method used to find terminated processes was flawed in that it didn't also take into account missing `popen` objects.

- Canvas: group and chord now works with anon signatures as long as the group/chord object is associated with an app instance (Issue #1744).

    You can pass the app by using group(..., app=app).

3.1.6

release-date 2013-12-02 06:00 p.m. UTC

release-by Ask Solem

- Now depends on billiard 3.3.0.10.

- Now depends on Kombu 3.0.7.

- Fixed problem where Mingle caused the worker to hang at start-up (Issue #1686).

- Beat: Would attempt to drop privileges twice (Issue #1708).

- Windows: Fixed error with geteuid not being available (Issue #1676).

- Tasks can now provide a list of expected error classes (Issue #1682).

    The list should only include errors that the task is expected to raise during normal operation:

    ```
    @task(throws=(KeyError, HttpNotFound))
    ```

    What happens when an exceptions is raised depends on the type of error:

    - Expected errors (included in Task.throws)

        Will be logged using severity INFO, and traceback is excluded.

    - Unexpected errors

        Will be logged using severity ERROR, with traceback included.

- Cache result backend now compatible with Python 3 (Issue #1697).

- CentOS init-script: Now compatible with SysV style init symlinks.

    Fix contributed by Jonathan Jordan.

- Events: Fixed problem when task name isn't defined (Issue #1710).

    Fix contributed by Mher Movsisyan.

- Task: Fixed unbound local errors (Issue #1684).

    Fix contributed by Markus Ullmann.

- Canvas: Now unrolls groups with only one task (optimization) (Issue #1656).

- Task: Fixed problem with ETA and timezones.

    Fix contributed by Alexander Koval.

- Django: Worker now performs model validation (Issue #1681).

- Task decorator now emits less confusing errors when used with incorrect arguments (Issue #1692).

- Task: New method Task.send_event can be used to send custom events to Flower and other monitors.

- Fixed a compatibility issue with non-abstract task classes

- Events from clients now uses new node name format (gen<pid>@<hostname>).

- Fixed rare bug with Callable not being defined at interpreter shutdown (Issue #1678).

> Fix contributed by Nick Johnson.

- Fixed Python 2.6 compatibility (Issue #1679).

3.1.5

> **release-date** 2013-11-21 06:20 p.m. UTC
>
> **release-by** Ask Solem

- Now depends on Kombu 3.0.6.

- Now depends on `billiard` 3.3.0.8

- App: `config_from_object` is now lazy (Issue #1665).

- App: `autodiscover_tasks` is now lazy.

  > Django users should now wrap access to the settings object in a lambda:

  ```
  app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
  ```

  > this ensures that the settings object isn't prepared prematurely.

- Fixed regression for `--app` argument experienced by some users (Issue #1653).

- Worker: Now respects the `--uid` and `--gid` arguments even if `--detach` isn't enabled.

- Beat: Now respects the `--uid` and `--gid` arguments even if `--detach` isn't enabled.

- Python 3: Fixed unorderable error occurring with the worker `-B` argument enabled.

- `celery.VERSION` is now a named tuple.

- `maybe_signature(list)` is now applied recursively (Issue #1645).

- `celery shell` command: Fixed `IPython.frontend` deprecation warning.

- The default app no longer includes the built-in fix-ups.

  > This fixes a bug where `celery multi` would attempt to load the Django settings module before entering the target working directory.

- The Django daemonization tutorial was changed.

  > Users no longer have to explicitly export `DJANGO_SETTINGS_MODULE` in `/etc/default/celeryd` when the new project layout is used.

- Redis result backend: expiry value can now be 0 (Issue #1661).

- Censoring settings now accounts for non-string keys (Issue #1663).

- App: New `autofinalize` option.

  > Apps are automatically finalized when the task registry is accessed. You can now disable this behavior so that an exception is raised instead.

  > Example:

  ```
  app = Celery(autofinalize=False)

  # raises RuntimeError
  tasks = app.tasks

  @app.task
  ```

  (continues on next page)

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

# raises RuntimeError
add.delay(2, 2)

app.finalize()
# no longer raises:
tasks = app.tasks
add.delay(2, 2)
```

- The worker didn't send monitoring events during shutdown.

- Worker: Mingle and gossip is now automatically disabled when used with an unsupported transport (Issue #1664).

- `celery` command: Preload options now supports the rare `--opt value` format (Issue #1668).

- `celery` command: Accidentally removed options appearing before the sub-command, these are now moved to the end instead.

- Worker now properly responds to `inspect stats` commands even if received before start-up is complete (Issue #1659).

- `task_postrun` is now sent within a `finally` block, to make sure the signal is always sent.

- Beat: Fixed syntax error in string formatting.

    Contributed by @nadad.

- Fixed typos in the documentation.

    Fixes contributed by Loic Bistuer, @sunfinite.

- Nested chains now works properly when constructed using the `chain` type instead of the `|` operator (Issue #1656).

3.1.4

release-date 2013-11-15 11:40 p.m. UTC

release-by Ask Solem

- Now depends on Kombu 3.0.5.

- Now depends on `billiard` 3.3.0.7

- Worker accidentally set a default socket timeout of 5 seconds.

- Django: Fix-up now sets the default app so that threads will use the same app instance (e.g., for **manage.py runserver**).

- Worker: Fixed Unicode error crash at start-up experienced by some users.

- Calling `.apply_async` on an empty chain now works again (Issue #1650).

- The `celery multi show` command now generates the same arguments as the start command does.

- The `--app` argument could end up using a module object instead of an app instance (with a resulting crash).

- Fixed a syntax error problem in the beat init-script.

    Fix contributed by Vsevolod.

- Tests now passing on PyPy 2.1 and 2.2.

3.1.3

release-date 2013-11-13 00:55 a.m. UTC

release-by Ask Solem

- Fixed compatibility problem with Python 2.7.0 - 2.7.5 (Issue #1637)

    `unpack_from` started supporting `memoryview` arguments in Python 2.7.6.

- Worker: `-B` argument accidentally closed files used for logging.

- Task decorated tasks now keep their docstring (Issue #1636)

3.1.2

release-date 2013-11-12 08:00 p.m. UTC

release-by Ask Solem

- Now depends on `billiard` 3.3.0.6

- No longer needs the billiard C extension to be installed.

- The worker silently ignored task errors.

- Django: Fixed `ImproperlyConfigured` error raised when no database backend specified.

    Fix contributed by [@j0hnsmith](#).

- Prefork pool: Now using `_multiprocessing.read` with `memoryview` if available.

- `close_open_fds` now uses `os.closerange` if available.

- `get_fdmax` now takes value from `sysconfig` if possible.

3.1.1

release-date 2013-11-11 06:30 p.m. UTC

release-by Ask Solem

- Now depends on `billiard` 3.3.0.4.

- Python 3: Fixed compatibility issues.

- Windows: Accidentally showed warning that the billiard C extension wasn't installed (Issue #1630).

- Django: Tutorial updated with a solution that sets a default `DJANGO_SETTINGS_MODULE` so that it doesn't have to be typed in with the **celery** command.

    Also fixed typos in the tutorial, and added the settings required to use the Django database backend.

    Thanks to Chris Ward, [@orarbel](#).

- Django: Fixed a problem when using the Django settings in Django 1.6.

- Django: Fix-up shouldn't be applied if the django loader is active.

- Worker: Fixed attribute error for `human_write_stats` when using the compatibility prefork pool implementation.

- Worker: Fixed compatibility with billiard without C extension.

- Inspect.conf: Now supports a `with_defaults` argument.

- Group.restore: The backend argument wasn't respected.

3.1.0

> **release-date** 2013-11-09 11:00 p.m. UTC
>
> **release-by** Ask Solem

See *What's new in Celery 3.1 (Cipater)*.

3.13.9 What's new in Celery 3.0 (Chiastic Slide)

Celery is a simple, flexible, and reliable distributed system to process vast amounts of messages, while providing operations with the tools required to maintain such a system.

It's a task queue with focus on real-time processing, while also supporting task scheduling.

Celery has a large and diverse community of users and contributors, you should come join us *on IRC* or *our mailing-list*.

To read more about Celery you should go read the *introduction*.

While this version is backward compatible with previous versions it's important that you read the following section.

If you use Celery in combination with Django you must also read the django-celery changelog and upgrade to django-celery 3.0.

This version is officially supported on CPython 2.5, 2.6, 2.7, 3.2 and 3.3, as well as PyPy and Jython.

Highlights

> **Overview**
>
> - A new and improved API, that's both simpler and more powerful.
>
>   Everyone must read the new *First Steps with Celery* tutorial, and the new *Next Steps* tutorial. Oh, and why not reread the user guide while you're at it :)
>
>   There are no current plans to deprecate the old API, so you don't have to be in a hurry to port your applications.
>
> - The worker is now thread-less, giving great performance improvements.
>
> - The new "Canvas" makes it easy to define complex work-flows.
>
>   Ever wanted to chain tasks together? This is possible, but not just that, now you can even chain together groups and chords, or even combine multiple chains.
>
>   Read more in the *Canvas* user guide.
>
> - All of Celery's command-line programs are now available from a single **celery** umbrella command.
>
> - This is the last version to support Python 2.5.
>
>   Starting with Celery 3.1, Python 2.6 or later is required.

> • Support for the new librabbitmq C client.
>
>> Celery will automatically use the librabbitmq module if installed, which is a very fast and memory-optimized replacement for the amqp module.
>
> • Redis support is more reliable with improved ack emulation.
>
> • Celery now always uses UTC
>
> • Over 600 commits, 30k additions/36k deletions.
>
>> In comparison 1.0 2.0 had 18k additions/8k deletions.

Important Notes

Broadcast exchanges renamed

The workers remote control command exchanges has been renamed (a new *pidbox* name), this is because the `auto_delete` flag on the exchanges has been removed, and that makes it incompatible with earlier versions.

You can manually delete the old exchanges if you want, using the **celery amqp** command (previously called `camqadm`):

```
$ celery amqp exchange.delete celeryd.pidbox
$ celery amqp exchange.delete reply.celeryd.pidbox
```

Event-loop

The worker is now running *without threads* when used with RabbitMQ (AMQP), or Redis as a broker, resulting in:

- Much better overall performance.
- Fixes several edge case race conditions.
- Sub-millisecond timer precision.
- Faster shutdown times.

The transports supported are: `py-amqp librabbitmq`, `redis`, and `amqplib`. Hopefully this can be extended to include additional broker transports in the future.

For increased reliability the `CELERY_FORCE_EXECV` setting is enabled by default if the event-loop isn't used.

New **celery** umbrella command

All Celery's command-line programs are now available from a single **celery** umbrella command.

You can see a list of sub-commands and options by running:

```
$ celery help
```

Commands include:

- `celery worker` (previously `celeryd`).
- `celery beat` (previously `celerybeat`).
- `celery amqp` (previously `camqadm`).

The old programs are still available (`celeryd`, `celerybeat`, etc), but you're discouraged from using them.

Now depends on billiard

Billiard is a fork of the multiprocessing containing the no-execv patch by `sbt` (http://bugs.python.org/issue8713), and also contains the pool improvements previously located in Celery.

This fork was necessary as changes to the C extension code was required for the no-execv patch to work.

- Issue #625
- Issue #627
- Issue #640
- *django-celery #122 <https://github.com/celery/django-celery/issues/122*
- *django-celery #124 <https://github.com/celery/django-celery/issues/122*

`celery.app.task` no longer a package

The *`celery.app.task`* module is now a module instead of a package.

The `setup.py` install script will try to remove the old package, but if that doesn't work for some reason you have to remove it manually. This command helps:

```
$ rm -r $(dirname $(python -c 'import celery;print(celery.__file__)'))/app/task/
```

If you experience an error like `ImportError:  cannot import name _unpickle_task`, you just have to remove the old package and everything is fine.

Last version to support Python 2.5

The 3.0 series will be last version to support Python 2.5, and starting from 3.1 Python 2.6 and later will be required.

With several other distributions taking the step to discontinue Python 2.5 support, we feel that it is time too.

Python 2.6 should be widely available at this point, and we urge you to upgrade, but if that's not possible you still have the option to continue using the Celery 3.0, and important bug fixes introduced in Celery 3.1 will be back-ported to Celery 3.0 upon request.

UTC timezone is now used

This means that ETA/countdown in messages aren't compatible with Celery versions prior to 2.5.

You can disable UTC and revert back to old local time by setting the `CELERY_ENABLE_UTC` setting.

Redis: Ack emulation improvements

Reducing the possibility of data loss.

Acks are now implemented by storing a copy of the message when the message is consumed. The copy isn't removed until the consumer acknowledges or rejects it.

This means that unacknowledged messages will be redelivered either when the connection is closed, or when the visibility timeout is exceeded.

- Visibility timeout

    This is a timeout for acks, so that if the consumer doesn't ack the message within this time limit, the message is redelivered to another consumer.

    The timeout is set to one hour by default, but can be changed by configuring a transport option:

    ```
    BROKER_TRANSPORT_OPTIONS = {'visibility_timeout': 18000}  # 5
    ↪hours
    ```

Note: Messages that haven't been acked will be redelivered if the visibility timeout is exceeded, for Celery users this means that ETA/countdown tasks that are scheduled to execute with a time that exceeds the visibility timeout will be executed twice (or more). If you plan on using long ETA/countdowns you should tweak the visibility timeout accordingly.

Setting a long timeout means that it'll take a long time for messages to be redelivered in the event of a power failure, but if so happens you could temporarily set the visibility timeout lower to flush out messages when you start up the systems again.

News

Chaining Tasks

Tasks can now have callbacks and errbacks, and dependencies are recorded

- The task message format have been updated with two new extension keys

    Both keys can be empty/undefined or a list of subtasks.

    - `callbacks`

        Applied if the task exits successfully, with the result of the task as an argument.

    - `errbacks`

        Applied if an error occurred while executing the task, with the uuid of the task as an argument. Since it may not be possible to serialize the exception instance, it passes the uuid of the task instead. The uuid can then be used to retrieve the exception and traceback of the task from the result backend.

    - `link` and `link_error` keyword arguments has been added to `apply_async`.

        These add callbacks and errbacks to the task, and you can read more about them at *Linking (callbacks/errbacks)*.

    - We now track what subtasks a task sends, and some result backends supports retrieving this information.

        * task.request.children

            Contains the result instances of the subtasks the currently executing task has applied.

        * AsyncResult.children

            Returns the tasks dependencies, as a list of `AsyncResult`/`ResultSet` instances.

        * AsyncResult.iterdeps

> > Recursively iterates over the tasks dependencies, yielding *(parent, node)* tuples.
> >
> > Raises IncompleteStream if any of the dependencies hasn't returned yet.
>
> * AsyncResult.graph
>
> > A [`DependencyGraph`](#) of the tasks dependencies. With this you can also convert to dot format:
> >
> > ```python
> > with open('graph.dot') as fh:
> >     result.graph.to_dot(fh)
> > ```
> >
> > then produce an image of the graph:
> >
> > ```
> > $ dot -Tpng graph.dot -o graph.png
> > ```

- A new special subtask called `chain` is also included:

  ```python
  >>> from celery import chain

  # (2 + 2) * 8 / 2
  >>> res = chain(add.subtask((2, 2)),
                  mul.subtask((8,)),
                  div.subtask((2,))).apply_async()
  >>> res.get() == 16

  >>> res.parent.get() == 32

  >>> res.parent.parent.get() == 4
  ```

- Adds `AsyncResult.get_leaf()`

  Waits and returns the result of the leaf subtask. That's the last node found when traversing the graph, but this means that the graph can be 1-dimensional only (in effect a list).

- Adds `subtask.link(subtask)` + `subtask.link_error(subtask)`

  Shortcut to `s.options.setdefault('link', []).append(subtask)`

- Adds `subtask.flatten_links()`

  Returns a flattened list of all dependencies (recursively)

Redis: Priority support

The message's `priority` field is now respected by the Redis transport by having multiple lists for each named queue. The queues are then consumed by in order of priority.

The priority field is a number in the range of 0 - 9, where 0 is the default and highest priority.

The priority range is collapsed into four steps by default, since it is unlikely that nine steps will yield more benefit than using four steps. The number of steps can be configured by setting the `priority_steps` transport option, which must be a list of numbers in **sorted order**:

```python
>>> BROKER_TRANSPORT_OPTIONS = {
...     'priority_steps': [0, 2, 4, 6, 8, 9],
... }
```

Priorities implemented in this way isn't as reliable as priorities on the server side, which is why the feature is nick-named "quasi-priorities"; **Using routing is still the suggested way of ensuring quality of service**, as client im-plemented priorities fall short in a number of ways, for example if the worker is busy with long running tasks, has prefetched many messages, or the queues are congested.

Still, it is possible that using priorities in combination with routing can be more beneficial than using routing or priorities alone. Experimentation and monitoring should be used to prove this.

Contributed by Germán M. Bravo.

Redis: Now cycles queues so that consuming is fair

This ensures that a very busy queue won't block messages from other queues, and ensures that all queues have an equal chance of being consumed from.

This used to be the case before, but the behavior was accidentally changed while switching to using blocking pop.

group/*chord*/*chain* are now subtasks

- group is no longer an alias to `TaskSet`, but new all together, since it was very difficult to migrate the `TaskSet` class to become a subtask.

- A new shortcut has been added to tasks:

  ```
  >>> task.s(arg1, arg2, kw=1)
  ```

  as a shortcut to:

  ```
  >>> task.subtask((arg1, arg2), {'kw': 1})
  ```

- Tasks can be chained by using the | operator:

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

- Subtasks can be "evaluated" using the ~ operator:

  ```
  >>> ~add.s(2, 2)
  4

  >>> ~(add.s(2, 2) | pow.s(2))
  ```

  is the same as:

  ```
  >>> chain(add.s(2, 2), pow.s(2)).apply_async().get()
  ```

- A new subtask_type key has been added to the subtask dictionary.

  This can be the string `"chord"`, `"group"`, `"chain"`, `"chunks"`, `"xmap"`, or `"xstarmap"`.

- maybe_subtask now uses subtask_type to reconstruct the object, to be used when using non-pickle serializers.

- The logic for these operations have been moved to dedicated tasks celery.chord, celery.chain and celery.group.

- subtask no longer inherits from AttributeDict.

  It's now a pure dict subclass with properties for attribute access to the relevant keys.

- The repr's now outputs how the sequence would like imperatively:

```
>>> from celery import chord

>>> (chord([add.s(i, i) for i in xrange(10)], xsum.s())
        | pow.s(2))
tasks.xsum([tasks.add(0, 0),
            tasks.add(1, 1),
            tasks.add(2, 2),
            tasks.add(3, 3),
            tasks.add(4, 4),
            tasks.add(5, 5),
            tasks.add(6, 6),
            tasks.add(7, 7),
            tasks.add(8, 8),
            tasks.add(9, 9)]) | tasks.pow(2)
```

New remote control commands

These commands were previously experimental, but they've proven stable and is now documented as part of the official API.

- *add_consumer*/*cancel_consumer*

  Tells workers to consume from a new queue, or cancel consuming from a queue. This command has also been changed so that the worker remembers the queues added, so that the change will persist even if the connection is re-connected.

  These commands are available programmatically as *app.control.add_consumer()* / *app. control.cancel_consumer()*:

  ```
  >>> celery.control.add_consumer(queue_name,
  ...     destination=['w1.example.com'])
  >>> celery.control.cancel_consumer(queue_name,
  ...     destination=['w1.example.com'])
  ```

  or using the **celery control** command:

  ```
  $ celery control -d w1.example.com add_consumer queue
  $ celery control -d w1.example.com cancel_consumer queue
  ```

  Note: Remember that a control command without *destination* will be sent to **all workers**.

- autoscale

  Tells workers with --autoscale enabled to change autoscale max/min concurrency settings.

  This command is available programmatically as *app.control.autoscale()*:

  ```
  >>> celery.control.autoscale(max=10, min=5,
  ...     destination=['w1.example.com'])
  ```

  or using the **celery control** command:

  ```
  $ celery control -d w1.example.com autoscale 10 5
  ```

- pool_grow/pool_shrink

Tells workers to add or remove pool processes.

These commands are available programmatically as *app.control.pool_grow()* / *app.control.pool_shrink()*:

```
>>> celery.control.pool_grow(2, destination=['w1.example.com'])
>>> celery.contorl.pool_shrink(2, destination=['w1.example.com'])
```

or using the **celery control** command:

```
$ celery control -d w1.example.com pool_grow 2
$ celery control -d w1.example.com pool_shrink 2
```

- **celery control** now supports *rate_limit* and time_limit commands.

    See celery control --help for details.

Crontab now supports Day of Month, and Month of Year arguments

See the updated list of examples at *Crontab schedules*.

Immutable subtasks

subtask's can now be immutable, which means that the arguments won't be modified when calling callbacks:

```
>>> chain(add.s(2, 2), clear_static_electricity.si())
```

means it'll not receive the argument of the parent task, and .si() is a shortcut to:

```
>>> clear_static_electricity.subtask(immutable=True)
```

Logging Improvements

Logging support now conforms better with best practices.

- Classes used by the worker no longer uses app.get_default_logger, but uses *celery.utils.log.get_logger* which simply gets the logger not setting the level, and adds a NullHandler.
- Loggers are no longer passed around, instead every module using logging defines a module global logger that's used throughout.
- All loggers inherit from a common logger called "celery".
- Before task.get_logger would setup a new logger for every task, and even set the log level. This is no longer the case.

    – Instead all task loggers now inherit from a common "celery.task" logger that's set up when programs call *setup_logging_subsystem*.

    – Instead of using LoggerAdapter to augment the formatter with the task_id and task_name field, the task base logger now use a special formatter adding these values at run-time from the currently executing task.

- In fact, task.get_logger is no longer recommended, it is better to add a module-level logger to your tasks module.

    For example, like this:

```
from celery.utils.log import get_task_logger

logger = get_task_logger(__name__)

@celery.task
def add(x, y):
    logger.debug('Adding %r + %r' % (x, y))
    return x + y
```

The resulting logger will then inherit from the `"celery.task"` logger so that the current task name and id is included in logging output.

- Redirected output from stdout/stderr is now logged to a "celery.redirected" logger.

- In addition a few warnings.warn have been replaced with logger.warn.

- Now avoids the 'no handlers for logger multiprocessing' warning

Task registry no longer global

Every Celery instance now has its own task registry.

You can make apps share registries by specifying it:

```
>>> app1 = Celery()
>>> app2 = Celery(tasks=app1.tasks)
```

Note that tasks are shared between registries by default, so that tasks will be added to every subsequently created task registry. As an alternative tasks can be private to specific task registries by setting the `shared` argument to the `@task` decorator:

```
@celery.task(shared=False)
def add(x, y):
    return x + y
```

Abstract tasks are now lazily bound

The `Task` class is no longer bound to an app by default, it will first be bound (and configured) when a concrete subclass is created.

This means that you can safely import and make task base classes, without also initializing the app environment:

```
from celery.task import Task

class DebugTask(Task):
    abstract = True

    def __call__(self, *args, **kwargs):
        print('CALLING %r' % (self,))
        return self.run(*args, **kwargs)
```

```
>>> DebugTask
<unbound DebugTask>

>>> @celery1.task(base=DebugTask)
```

(continues on next page)

```
...  def add(x, y):
...      return x + y
>>> add.__class__
<class add of <Celery default:0x101510d10>>
```

Lazy task decorators

The `@task` decorator is now lazy when used with custom apps.

That is, if `accept_magic_kwargs` is enabled (her by called "compat mode"), the task decorator executes inline like before, however for custom apps the @task decorator now returns a special PromiseProxy object that's only evaluated on access.

All promises will be evaluated when *app.finalize()* is called, or implicitly when the task registry is first used.

Smart *–app* option

The *--app* option now 'auto-detects'

- If the provided path is a module it tries to get an attribute named 'celery'.

- If the provided path is a package it tries to import a sub module named celery', and get the celery attribute from that module.

For example, if you have a project named `proj` where the celery app is located in `from proj.celery import app`, then the following will be equivalent:

```
$ celery worker --app=proj
$ celery worker --app=proj.celery:
$ celery worker --app=proj.celery:app
```

In Other News

- New `CELERYD_WORKER_LOST_WAIT` to control the timeout in seconds before `billiard.WorkerLostError` is raised when a worker can't be signaled (Issue #595).

  Contributed by Brendon Crawford.

- Redis event monitor queues are now automatically deleted (Issue #436).

- App instance factory methods have been converted to be cached descriptors that creates a new subclass on access.

  For example, this means that `app.Worker` is an actual class and will work as expected when:

  ```
  class Worker(app.Worker):
      ...
  ```

- New signal: *task_success*.

- Multiprocessing logs are now only emitted if the `MP_LOG` environment variable is set.

- The Celery instance can now be created with a broker URL

```
app = Celery(broker='redis://')
```

- Result backends can now be set using a URL

    Currently only supported by redis. Example use:

    ```
    CELERY_RESULT_BACKEND = 'redis://localhost/1'
    ```

- Heartbeat frequency now every 5s, and frequency sent with event

    The heartbeat frequency is now available in the worker event messages, so that clients can decide when to consider workers offline based on this value.

- Module celery.actors has been removed, and will be part of cl instead.

- Introduces new `celery` command, which is an entry-point for all other commands.

    The main for this command can be run by calling `celery.start()`.

- Annotations now supports decorators if the key starts with '@'.

    For example:

    ```python
    def debug_args(fun):

        @wraps(fun)
        def _inner(*args, **kwargs):
            print('ARGS: %r' % (args,))
        return _inner

    CELERY_ANNOTATIONS = {
        'tasks.add': {'@__call__': debug_args},
    }
    ```

    Also tasks are now always bound by class so that annotated methods end up being bound.

- Bug-report now available as a command and broadcast command

    - Get it from a Python REPL:

        ```python
        >>> import celery
        >>> print(celery.bugreport())
        ```

    - Using the `celery` command line program:

        ```
        $ celery report
        ```

    - Get it from remote workers:

        ```
        $ celery inspect report
        ```

- Module `celery.log` moved to *celery.app.log*.

- Module `celery.task.control` moved to *celery.app.control*.

- New signal: *task_revoked*

    Sent in the main process when the task is revoked or terminated.

- `AsyncResult.task_id` renamed to `AsyncResult.id`

- `TasksetResult.taskset_id` renamed to `.id`

- xmap(task, sequence) and xstarmap(task, sequence)

    Returns a list of the results applying the task function to every item in the sequence.

    Example:

    ```
    >>> from celery import xstarmap

    >>> xstarmap(add, zip(range(10), range(10))).apply_async()
    [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
    ```

- chunks(task, sequence, chunksize)

- group.skew(start=, stop=, step=)

    Skew will skew the countdown for the individual tasks in a group – for example with this group:

    ```
    >>> g = group(add.s(i, i) for i in xrange(10))
    ```

    Skewing the tasks from 0 seconds to 10 seconds:

    ```
    >>> g.skew(stop=10)
    ```

    Will have the first task execute in 0 seconds, the second in 1 second, the third in 2 seconds and so on.

- 99% test Coverage

- CELERY_QUEUES can now be a list/tuple of `Queue` instances.

    Internally `app.amqp.queues` is now a mapping of name/Queue instances, instead of converting on the fly.

- Can now specify connection for `app.control.inspect`.

    ```
    from kombu import Connection

    i = celery.control.inspect(connection=Connection('redis://'))
    i.active_queues()
    ```

- CELERY_FORCE_EXECV is now enabled by default.

    If the old behavior is wanted the setting can be set to False, or the new *–no-execv* option to **celery worker**.

- Deprecated module celery.conf has been removed.

- The CELERY_TIMEZONE now always require the pytz library to be installed (except if the timezone is set to *UTC*).

- The Tokyo Tyrant backend has been removed and is no longer supported.

- Now uses `maybe_declare()` to cache queue declarations.

- There's no longer a global default for the CELERYBEAT_MAX_LOOP_INTERVAL setting, it is instead set by individual schedulers.

- Worker: now truncates very long message bodies in error reports.

- No longer deep-copies exceptions when trying to serialize errors.

- CELERY_BENCH environment variable, will now also list memory usage statistics at worker shutdown.

- Worker: now only ever use a single timer for all timing needs, and instead set different priorities.

- An exceptions arguments are now safely pickled

Contributed by Matt Long.

- Worker/Beat no longer logs the start-up banner.

    Previously it would be logged with severity warning, now it's only written to stdout.

- The `contrib/` directory in the distribution has been renamed to `extra/`.

- New signal: *task_revoked*

- *celery.contrib.migrate*: Many improvements, including; filtering, queue migration, and support for acking messages on the broker migrating from.

    Contributed by John Watson.

- Worker: Prefetch count increments are now optimized and grouped together.

- Worker: No longer calls `consume` on the remote control command queue twice.

    Probably didn't cause any problems, but was unnecessary.

Internals

- `app.broker_connection` is now `app.connection`

    Both names still work.

- Compatibility modules are now generated dynamically upon use.

    These modules are `celery.messaging`, `celery.log`, `celery.decorators` and `celery.registry`.

- *celery.utils* refactored into multiple modules:

    celery.utils.text celery.utils.imports celery.utils.functional

- Now using *kombu.utils.encoding* instead of `celery.utils.encoding`.

- Renamed module `celery.routes` -> *celery.app.routes*.

- Renamed package `celery.db` -> *celery.backends.database*.

- Renamed module `celery.abstract` -> `celery.worker.bootsteps`.

- Command line docs are now parsed from the module docstrings.

- Test suite directory has been reorganized.

- **setup.py** now reads docs from the `requirements/` directory.

- Celery commands no longer wraps output (Issue #700).

    Contributed by Thomas Johansson.

Experimental

`celery.contrib.methods`: Task decorator for methods

This is an experimental module containing a task decorator, and a task decorator filter, that can be used to create tasks out of methods:

```
from celery.contrib.methods import task_method

class Counter(object):

    def __init__(self):
        self.value = 1

    @celery.task(name='Counter.increment', filter=task_method)
    def increment(self, n=1):
        self.value += 1
        return self.value
```

See `celery.contrib.methods` for more information.

Unscheduled Removals

Usually we don't make backward incompatible removals, but these removals should have no major effect.

- The following settings have been renamed:

    - CELERYD_ETA_SCHEDULER -> CELERYD_TIMER

    - CELERYD_ETA_SCHEDULER_PRECISION -> CELERYD_TIMER_PRECISION

Deprecation Time-line Changes

See the *Celery Deprecation Time-line*.

- The `celery.backends.pyredis` compat module has been removed.

    Use *`celery.backends.redis`* instead!

- The following undocumented API's has been moved:

    - control.inspect.add_consumer -> *app.control.add_consumer()*.

    - control.inspect.cancel_consumer -> *app.control.cancel_consumer()*.

    - control.inspect.enable_events -> *app.control.enable_events()*.

    - control.inspect.disable_events -> *app.control.disable_events()*.

    This way `inspect()` is only used for commands that don't modify anything, while idempotent control commands that make changes are on the control objects.

Fixes

- Retry SQLAlchemy backend operations on DatabaseError/OperationalError (Issue #634)

- Tasks that called `retry` wasn't acknowledged if acks late was enabled

    Fix contributed by David Markey.

- The message priority argument wasn't properly propagated to Kombu (Issue #708).

    Fix contributed by Eran Rundstein

3.13.10 Change history for Celery 3.0

If you're looking for versions prior to 3.0.x you should go to *History*.

3.0.24

release-date 2013-10-11 04:40 p.m. BST

release-by Ask Solem

- Now depends on Kombu 2.5.15.
- Now depends on billiard version 2.7.3.34.
- AMQP Result backend: No longer caches queue declarations.

> The queues created by the AMQP result backend are always unique, so caching the declarations caused a slow memory leak.

- Worker: Fixed crash when hostname contained Unicode characters.

  Contributed by Daodao.

- The worker would no longer start if the *-P solo* pool was selected (Issue #1548).

- Redis/Cache result backends wouldn't complete chords if any of the tasks were retried (Issue #1401).

- Task decorator is no longer lazy if app is finalized.

- AsyncResult: Fixed bug with `copy(AsyncResult)` when no `current_app` available.

- ResultSet: Now properly propagates app when passed string id's.

- Loader now ignores `CELERY_CONFIG_MODULE` if value is empty string.

- Fixed race condition in Proxy object where it tried to delete an attribute twice, resulting in `AttributeError`.

- Task methods now works with the `CELERY_ALWAYS_EAGER` setting (Issue #1478).

- `Broadcast` queues were accidentally declared when publishing tasks (Issue #1540).

- New `C_FAKEFORK` environment variable can be used to debug the init-scripts.

  Setting this will skip the daemonization step so that errors printed to stderr after standard outs are closed can be seen:

  ```
  $ C_FAKEFORK /etc/init.d/celeryd start
  ```

  This works with the *celery multi* command in general.

- `get_pickleable_etype` didn't always return a value (Issue #1556).

- Fixed bug where `app.GroupResult.restore` would fall back to the default app.

- Fixed rare bug where built-in tasks would use the current_app.

- `maybe_fileno()` now handles `ValueError`.

3.0.23

release-date 2013-09-02 01:00 p.m. BST

release-by Ask Solem

- Now depends on Kombu 2.5.14.

- `send_task` didn't honor `link` and `link_error` arguments.

  This had the side effect of chains not calling unregistered tasks, silently discarding them.

  Fix contributed by Taylor Nelson.

- `celery.state`: Optimized precedence lookup.

  Contributed by Matt Robenolt.

- POSIX: Daemonization didn't redirect `sys.stdin` to `/dev/null`.

  Fix contributed by Alexander Smirnov.

- Canvas: group bug caused fallback to default app when `.apply_async` used (Issue #1516)

- Canvas: generator arguments wasn't always pickleable.

3.0.22

release-date 2013-08-16 04:30 p.m. BST

release-by Ask Solem

- Now depends on Kombu 2.5.13.

- Now depends on billiard 2.7.3.32

- Fixed bug with monthly and yearly Crontabs (Issue #1465).

    Fix contributed by Guillaume Gauvrit.

- Fixed memory leak caused by time limits (Issue #1129, Issue #1427)

- Worker will now sleep if being restarted more than 5 times in one second to avoid spamming with `worker-online` events.

- Includes documentation fixes

    Contributed by: Ken Fromm, Andreas Savvides, Alex Kiriukha, Michael Fladischer.

3.0.21

release-date 2013-07-05 04:30 p.m. BST

release-by Ask Solem

- Now depends on billiard 2.7.3.31.

    This version fixed a bug when running without the billiard C extension.

- 3.0.20 broke eventlet/gevent support (worker not starting).

- Fixed memory leak problem when MongoDB result backend was used with the gevent pool.

    Fix contributed by Ross Lawley.

3.0.20

release-date 2013-06-28 04:00 p.m. BST

release-by Ask Solem

- Contains workaround for deadlock problems.

    A better solution will be part of Celery 3.1.

- Now depends on Kombu 2.5.12.

- Now depends on billiard 2.7.3.30.

- `--loader` argument no longer supported importing loaders from the current directory.

- [Worker] Fixed memory leak when restarting after connection lost (Issue #1325).

- [Worker] Fixed UnicodeDecodeError at start-up (Issue #1373).

    Fix contributed by Jessica Tallon.

- [Worker] Now properly rewrites unpickleable exceptions again.

- Fixed possible race condition when evicting items from the revoked task set.

- [generic-init.d] Fixed compatibility with Ubuntu's minimal Dash shell (Issue #1387).

> Fix contributed by [@monkut](#).

- `Task.apply`/ALLWAYS_EAGER now also executes callbacks and errbacks (Issue #1336).
- [Worker] The `worker-shutdown` signal was no longer being dispatched (Issue #1339)j
- [Python 3] Fixed problem with threading.Event.

  > Fix contributed by Xavier Ordoquy.

- [Python 3] Now handles `io.UnsupportedOperation` that may be raised by `file.fileno()` in Python 3.
- [Python 3] Fixed problem with `qualname`.
- [events.State] Now ignores unknown event-groups.
- [MongoDB backend] No longer uses deprecated `safe` parameter.

  > Fix contributed by [@rfkrocktk](#).

- The eventlet pool now imports on Windows.
- [Canvas] Fixed regression where immutable chord members may receive arguments (Issue #1340).

  > Fix contributed by Peter Brook.

- [Canvas] chain now accepts generator argument again (Issue #1319).
- `celery.migrate` command now consumes from all queues if no queues specified.

  > Fix contributed by John Watson.

3.0.19

release-date 2013-04-17 04:30:00 p.m. BST

release-by Ask Solem

- Now depends on [billiard](#) 2.7.3.28
- A Python 3 related fix managed to disable the deadlock fix announced in 3.0.18.

  > Tests have been added to make sure this doesn't happen again.

- Task retry policy: Default max_retries is now 3.

  > This ensures clients won't be hanging while the broker is down.

  > ---
  >
  > **Note:** You can set a longer retry for the worker by using the *celeryd_after_setup* signal:
  >
  > ```python
  > from celery.signals import celeryd_after_setup
  >
  > @celeryd_after_setup.connect
  > def configure_worker(instance, conf, **kwargs):
  >     conf.CELERY_TASK_PUBLISH_RETRY_POLICY = {
  >         'max_retries': 100,
  >         'interval_start': 0,
  >         'interval_max': 1,
  >         'interval_step': 0.2,
  >     }
  > ```
  >
  > ---

- Worker: Will now properly display message body in error messages even if the body is a buffer instance.

- 3.0.18 broke the MongoDB result backend (Issue #1303).

3.0.18

release-date 2013-04-12 05:00:00 p.m. BST

release-by Ask Solem

- Now depends on kombu 2.5.10.

    See the kombu changelog.

- Now depends on billiard 2.7.3.27.

- Can now specify a white-list of accepted serializers using the new CELERY_ACCEPT_CONTENT setting.

    This means that you can force the worker to discard messages serialized with pickle and other untrusted serializers. For example to only allow JSON serialized messages use:

    ```
    CELERY_ACCEPT_CONTENT = ['json']
    ```

    you can also specify MIME types in the white-list:

    ```
    CELERY_ACCEPT_CONTENT = ['application/json']
    ```

- Fixed deadlock in multiprocessing's pool caused by the semaphore not being released when terminated by signal.

- Processes Pool: It's now possible to debug pool processes using GDB.

- celery report now censors possibly secret settings, like passwords and secret tokens.

    You should still check the output before pasting anything on the internet.

- Connection URLs now ignore multiple '+' tokens.

- Worker/statedb: Now uses pickle protocol 2 (Python 2.5+)

- Fixed Python 3 compatibility issues.

- Worker: A warning is now given if a worker is started with the same node name as an existing worker.

- Worker: Fixed a deadlock that could occur while revoking tasks (Issue #1297).

- Worker: The HUP handler now closes all open file descriptors before restarting to ensure file descriptors doesn't leak (Issue #1270).

- Worker: Optimized storing/loading the revoked tasks list (Issue #1289).

    After this change the *celery worker --statedb* file will take up more disk space, but loading from and storing the revoked tasks will be considerably faster (what before took 5 minutes will now take less than a second).

- Celery will now suggest alternatives if there's a typo in the broker transport name (e.g., ampq -> amqp).

- Worker: The auto-reloader would cause a crash if a monitored file was unlinked.

    Fix contributed by Agris Ameriks.

- Fixed AsyncResult pickling error.

    Fix contributed by Thomas Minor.

- Fixed handling of Unicode in logging output when using log colors (Issue #427).

- ConfigurationView is now a MutableMapping.

Contributed by Aaron Harnly.

- Fixed memory leak in LRU cache implementation.

    Fix contributed by Romuald Brunet.

- `celery.contrib.rdb`: Now works when sockets are in non-blocking mode.

    Fix contributed by Theo Spears.

- The *inspect reserved* remote control command included active (started) tasks with the reserved tasks (Issue #1030).

- The [`task_failure`](#) signal received a modified traceback object meant for pickling purposes, this has been fixed so that it now receives the real traceback instead.

- The `@task` decorator silently ignored positional arguments, it now raises the expected `TypeError` instead (Issue #1125).

- The worker will now properly handle messages with invalid ETA/expires fields (Issue #1232).

- The `pool_restart` remote control command now reports an error if the `CELERYD_POOL_RESTARTS` setting isn't set.

- `add_defaults`() can now be used with non-dict objects.

- Fixed compatibility problems in the Proxy class (Issue #1087).

    The class attributes `__module__`, `__name__` and `__doc__` are now meaningful string objects.

    Thanks to Marius Gedminas.

- MongoDB Backend: The `MONGODB_BACKEND_SETTINGS` setting now accepts a `option` key that lets you forward arbitrary kwargs to the underlying `pymongo.Connection` object (Issue #1015).

- Beat: The daily backend cleanup task is no longer enabled for result backends that support automatic result expiration (Issue #1031).

- Canvas list operations now takes application instance from the first task in the list, instead of depending on the `current_app` (Issue #1249).

- Worker: Message decoding error log message now includes traceback information.

- Worker: The start-up banner now includes system platform.

- `celery inspect|status|control` now gives an error if used with a SQL based broker transport.

3.0.17

release-date 2013-03-22 04:00:00 p.m. UTC

release-by Ask Solem

- Now depends on kombu 2.5.8

- Now depends on billiard 2.7.3.23

- RabbitMQ/Redis: thread-less and lock-free rate-limit implementation.

    This means that rate limits pose minimal overhead when used with RabbitMQ/Redis or future transports using the event-loop, and that the rate-limit implementation is now thread-less and lock-free.

    The thread-based transports will still use the old implementation for now, but the plan is to use the timer also for other broker transports in Celery 3.1.

- Rate limits now works with eventlet/gevent if using RabbitMQ/Redis as the broker.

- A regression caused `task.retry` to ignore additional keyword arguments.

    Extra keyword arguments are now used as execution options again. Fix contributed by Simon Engledew.

- Windows: Fixed problem with the worker trying to pickle the Django settings module at worker start-up.

- generic-init.d: No longer double quotes `$CELERYD_CHDIR` (Issue #1235).

- generic-init.d: Removes bash-specific syntax.

    Fix contributed by Pär Wieslander.

- Cassandra Result Backend: Now handles the `AllServersUnavailable` error (Issue #1010).

    Fix contributed by Jared Biel.

- Result: Now properly forwards apps to GroupResults when deserializing (Issue #1249).

    Fix contributed by Charles-Axel Dein.

- `GroupResult.revoke` now supports the `terminate` and `signal` keyword arguments.

- Worker: Multiprocessing pool workers now import task modules/configuration before setting up the logging system so that logging signals can be connected before they're dispatched.

- chord: The `AsyncResult` instance returned now has its `parent` attribute set to the header `GroupResult`.

    This is consistent with how `chain` works.

3.0.16

release-date 2013-03-07 04:00:00 p.m. UTC

release-by Ask Solem

- Happy International Women's Day!

    We have a long way to go, so this is a chance for you to get involved in one of the organizations working for making our communities more diverse.

    - PyLadies — http://pyladies.com

    - Girls Who Code — http://www.girlswhocode.com

    - Women Who Code — http://www.meetup.com/Women-Who-Code-SF/

- Now depends on kombu version 2.5.7

- Now depends on billiard version 2.7.3.22

- AMQP heartbeats are now disabled by default.

    Some users experiences issues with heartbeats enabled, and it's not strictly necessary to use them.

    If you're experiencing problems detecting connection failures, you can re-enable heartbeats by configuring the `BROKER_HEARTBEAT` setting.

- Worker: Now propagates connection errors occurring in multiprocessing callbacks, so that the connection can be reset (Issue #1226).

- Worker: Now propagates connection errors occurring in timer callbacks, so that the connection can be reset.

- The modules in `CELERY_IMPORTS` and `CELERY_INCLUDE` are now imported in the original order (Issue #1161).

> > The modules in `CELERY_IMPORTS` will be imported first, then continued by `CELERY_INCLUDE`.
> >
> > Thanks to Joey Wilhelm.

- New bash completion for `celery` available in the git repository:

    https://github.com/celery/celery/tree/3.0/extra/bash-completion

    You can source this file or put it in `bash_completion.d` to get auto-completion for the `celery` command-line utility.

- The node name of a worker can now include unicode characters (Issue #1186).

- The repr of a `crontab` object now displays correctly (Issue #972).

- `events.State` no longer modifies the original event dictionary.

- No longer uses `Logger.warn` deprecated in Python 3.

- Cache Backend: Now works with chords again (Issue #1094).

- Chord unlock now handles errors occurring while calling the callback.

- Generic worker init.d script: Status check is now performed by querying the pid of the instance instead of sending messages.

    Contributed by Milen Pavlov.

- Improved init-scripts for CentOS.

    - Updated to support Celery 3.x conventions.

    - Now uses CentOS built-in `status` and `killproc`

    - Support for multi-node / multi-pid worker services.

    - Standard color-coded CentOS service-init output.

    - A test suite.

    Contributed by Milen Pavlov.

- `ResultSet.join` now always works with empty result set (Issue #1219).

- A `group` consisting of a single task is now supported (Issue #1219).

- Now supports the `pycallgraph` program (Issue #1051).

- Fixed Jython compatibility problems.

- Django tutorial: Now mentions that the example app must be added to `INSTALLED_APPS` (Issue #1192).

3.0.15

release-date 2013-02-11 04:30:00 p.m. UTC

release-by Ask Solem

- Now depends on billiard 2.7.3.21 which fixed a syntax error crash.

- Fixed bug with `CELERY_SEND_TASK_SENT_EVENT`.

3.0.14

release-date 2013-02-08 05:00:00 p.m. UTC

release-by Ask Solem

- Now depends on Kombu 2.5.6

- Now depends on billiard 2.7.3.20

- `execv` is now disabled by default.

    It was causing too many problems for users, you can still enable it using the *CEL-ERYD_FORCE_EXECV* setting.

    execv was only enabled when transports other than AMQP/Redis was used, and it's there to prevent deadlocks caused by mutexes not being released before the process forks. Unfortunately it also changes the environment introducing many corner case bugs that're hard to fix without adding horrible hacks. Deadlock issues are reported far less often than the bugs that execv are causing, so we now disable it by default.

    Work is in motion to create non-blocking versions of these transports so that execv isn't necessary (which is the situation with the amqp and redis broker transports)

- Chord exception behavior defined (Issue #1172).

    From Celery 3.1 the chord callback will change state to FAILURE when a task part of a chord raises an exception.

    It was never documented what happens in this case, and the actual behavior was very unsatisfactory, indeed it will just forward the exception value to the chord callback.

    For backward compatibility reasons we don't change to the new behavior in a bugfix release, even if the current behavior was never documented. Instead you can enable the `CELERY_CHORD_PROPAGATES` setting to get the new behavior that'll be default from Celery 3.1.

    See more at *Error handling*.

- worker: Fixes bug with ignored and retried tasks.

    The `on_chord_part_return` and `Task.after_return` callbacks, nor the `task_postrun` signal should be called when the task was retried/ignored.

    Fix contributed by Vlad.

- `GroupResult.join_native` now respects the `propagate` argument.

- `subtask.id` added as an alias to `subtask['options'].id`

    ```
    >>> s = add.s(2, 2)
    >>> s.id = 'my-id'
    >>> s['options']
    {'task_id': 'my-id'}

    >>> s.id
    'my-id'
    ```

- worker: Fixed error *Could not start worker processes* occurring when restarting after connection failure (Issue #1118).

- Adds new signal `task-retried` (Issue #1169).

- *celery events –dumper* now handles connection loss.

- Will now retry sending the task-sent event in case of connection failure.

- amqp backend: Now uses `Message.requeue` instead of republishing the message after poll.

- New `BROKER_HEARTBEAT_CHECKRATE` setting introduced to modify the rate at which broker connection heartbeats are monitored.

    The default value was also changed from 3.0 to 2.0.

- *celery.events.state.State* is now pickleable.

    Fix contributed by Mher Movsisyan.

- *celery.utils.functional.LRUCache* is now pickleable.

    Fix contributed by Mher Movsisyan.

- The stats broadcast command now includes the workers pid.

    Contributed by Mher Movsisyan.

- New `conf` remote control command to get a workers current configuration.

    Contributed by Mher Movsisyan.

- Adds the ability to modify the chord unlock task's countdown argument (Issue #1146).

    Contributed by Jun Sakai

- beat: The scheduler now uses the *now()`* method of the schedule, so that schedules can provide a custom way to get the current date and time.

    Contributed by Raphaël Slinckx

- Fixed pickling of configuration modules on Windows or when execv is used (Issue #1126).

- Multiprocessing logger is now configured with loglevel `ERROR` by default.

    Since 3.0 the multiprocessing loggers were disabled by default (only configured when the `MP_LOG` environment variable was set).

3.0.13

release-date 2013-01-07 04:00:00 p.m. UTC

release-by Ask Solem

- Now depends on Kombu 2.5

    – amqp has replaced amqplib as the default transport, gaining support for AMQP 0.9, and the RabbitMQ extensions, including Consumer Cancel Notifications and heartbeats.

    – support for multiple connection URLs for failover.

    – Read more in the Kombu 2.5 changelog.

- Now depends on billiard 2.7.3.19

- Fixed a deadlock issue that could occur when the producer pool inherited the connection pool instance of the parent process.

- The `--loader` option now works again (Issue #1066).

- **celery** umbrella command: All sub-commands now supports the `--workdir` option (Issue #1063).

- Groups included in chains now give GroupResults (Issue #1057)

    Previously it would incorrectly add a regular result instead of a group result, but now this works:

```
>>> # [4 + 4, 4 + 8, 16 + 8]
>>> res = (add.s(2, 2) | group(add.s(4), add.s(8), add.s(16)))()
>>> res
<GroupResult: a0acf905-c704-499e-b03a-8d445e6398f7 [
    4346501c-cb99-4ad8-8577-12256c7a22b1,
    b12ead10-a622-4d44-86e9-3193a778f345,
    26c7a420-11f3-4b33-8fac-66cd3b62abfd]>
```

- Chains can now chain other chains and use partial arguments (Issue #1057).

    Example:

    ```
    >>> c1 = (add.s(2) | add.s(4))
    >>> c2 = (add.s(8) | add.s(16))

    >>> c3 = (c1 | c2)

    >>> # 8 + 2 + 4 + 8 + 16
    >>> assert c3(8).get() == 38
    ```

- Subtasks can now be used with unregistered tasks.

    You can specify subtasks even if you just have the name:

    ```
    >>> s = subtask(task_name, args=(), kwargs=())
    >>> s.delay()
    ```

- The **celery shell** command now always adds the current directory to the module path.

- The worker will now properly handle the pytz.AmbiguousTimeError exception raised when an ETA/countdown is prepared while being in DST transition (Issue #1061).

- force_execv: Now makes sure that task symbols in the original task modules will always use the correct app instance (Issue #1072).

- AMQP Backend: Now republishes result messages that have been polled (using result.ready() and friends, result.get() won't do this in this version).

- Crontab schedule values can now "wrap around"

    This means that values like 11-1 translates to [11, 12, 1].

    Contributed by Loren Abrams.

- multi stopwait command now shows the pid of processes.

    Contributed by Loren Abrams.

- **Handling of ETA/countdown fixed when the CELERY_ENABLE_UTC** setting is disabled (Issue #1065).

- A number of unneeded properties were included in messages, caused by accidentally passing Queue.as_dict as message properties.

- Rate limit values can now be float

    This also extends the string format so that values like "0.5/s" works.

    Contributed by Christoph Krybus

- Fixed a typo in the broadcast routing documentation (Issue #1026).

- Rewrote confusing section about idempotence in the task user guide.

- Fixed typo in the daemonization tutorial (Issue #1055).

- Fixed several typos in the documentation.

    Contributed by Marius Gedminas.

- Batches: Now works when using the eventlet pool.

    Fix contributed by Thomas Grainger.

- Batches: Added example sending results to `celery.contrib.batches`.

    Contributed by Thomas Grainger.

- MongoDB   backend: Connection   `max_pool_size`   can   now   be   set   in
  `CELERY_MONGODB_BACKEND_SETTINGS`.

    Contributed by Craig Younkins.

- Fixed problem when using earlier versions of pytz.

    Fix contributed by Vlad.

- Docs updated to include the default value for the `CELERY_TASK_RESULT_EXPIRES` setting.

- Improvements to the django-celery tutorial.

    Contributed by Locker537.

- The `add_consumer` control command didn't properly persist the addition of new queues so that they survived
  connection failure (Issue #1079).

3.0.12

release-date 2012-11-06 02:00 p.m. UTC

release-by Ask Solem

- Now depends on kombu 2.4.8

    - [Redis] New and improved fair queue cycle algorithm (Kevin McCarthy).

    - [Redis] Now uses a Redis-based mutex when restoring messages.

    - **[Redis] Number of messages that can be restored in one interval is no** longer limited (but can be set
      using the `unacked_restore_limit transport option`).

    - Heartbeat value can be specified in broker URLs (Mher Movsisyan).

    - Fixed problem with msgpack on Python 3 (Jasper Bryant-Greene).

- Now depends on billiard 2.7.3.18

- Celery can now be used with static analysis tools like PyDev/PyCharm/pylint etc.

- Development documentation has moved to Read The Docs.

    The new URL is: http://docs.celeryproject.org/en/master

- New `CELERY_QUEUE_HA_POLICY` setting used to set the default HA policy for queues when using Rab-
  bitMQ.

- New method `Task.subtask_from_request` returns a subtask using the current request.

- Results get_many method didn't respect timeout argument.

    Fix contributed by Remigiusz Modrzejewski

- generic_init.d scripts now support setting `CELERY_CREATE_DIRS` to always create log and pid directories
  (Issue #1045).

This can be set in your `/etc/default/celeryd`.

- Fixed strange kombu import problem on Python 3.2 (Issue #1034).

- Worker: ETA scheduler now uses millisecond precision (Issue #1040).

- The `--config` argument to programs is now supported by all loaders.

- The `CASSANDRA_OPTIONS` setting has now been documented.

    Contributed by Jared Biel.

- Task methods (`celery.contrib.methods`) cannot be used with the old task base class, the task decorator in that module now inherits from the new.

- An optimization was too eager and caused some logging messages to never emit.

- `celery.contrib.batches` now works again.

- Fixed missing white-space in `bdist_rpm` requirements (Issue #1046).

- Event state's `tasks_by_name` applied limit before filtering by name.

    Fix contributed by Alexander A. Sosnovskiy.

3.0.11

release-date 2012-09-26 04:00 p.m. UTC

release-by Ask Solem

- [security:low] generic-init.d scripts changed permissions of /var/log & /var/run

    In the daemonization tutorial the recommended directories were as follows:

    ```
    CELERYD_LOG_FILE="/var/log/celery/%n.log"
    CELERYD_PID_FILE="/var/run/celery/%n.pid"
    ```

    But in the scripts themselves the default files were `/var/log/celery%n.log` and `/var/run/celery%n.pid`, so if the user didn't change the location by configuration, the directories `/var/log` and `/var/run` would be created - and worse have their permissions and owners changed.

    This change means that:

    - Default pid file is `/var/run/celery/%n.pid`

    - Default log file is `/var/log/celery/%n.log`

    - The directories are only created and have their permissions changed if *no custom locations are set*.

    Users can force paths to be created by calling the `create-paths` sub-command:

    ```
    $ sudo /etc/init.d/celeryd create-paths
    ```

    Upgrading Celery won't update init-scripts

    To update the init-scripts you have to re-download the files from source control and update them manually. You can find the init-scripts for version 3.0.x at:

    https://github.com/celery/celery/tree/3.0/extra/generic-init.d

    ---

- Now depends on billiard 2.7.3.17

- Fixes request stack protection when app is initialized more than once (Issue #1003).

- ETA tasks now properly works when system timezone isn't same as the configured timezone (Issue #1004).

- Terminating a task now works if the task has been sent to the pool but not yet acknowledged by a pool process (Issue #1007).

    Fix contributed by Alexey Zatelepin

- Terminating a task now properly updates the state of the task to revoked, and sends a `task-revoked` event.

- Generic worker init-script now waits for workers to shutdown by default.

- Multi: No longer parses –app option (Issue #1008).

- Multi: `stop_verify` command renamed to `stopwait`.

- Daemonization: Now delays trying to create pidfile/logfile until after the working directory has been changed into.

- **celery worker** and **celery beat** commands now respects the *--no-color* option (Issue #999).

- Fixed typos in eventlet examples (Issue #1000)

    Fix contributed by Bryan Bishop. Congratulations on opening bug #1000!

- Tasks that raise *Ignore* are now acknowledged.

- Beat: Now shows the name of the entry in `sending due task` logs.

3.0.10

release-date 2012-09-20 05:30 p.m. BST

release-by Ask Solem

- Now depends on kombu 2.4.7

- Now depends on billiard 2.7.3.14

    - Fixes crash at start-up when using Django and pre-1.4 projects (`setup_environ`).

    - Hard time limits now sends the KILL signal shortly after TERM, to terminate processes that have signal handlers blocked by C extensions.

    - Billiard now installs even if the C extension cannot be built.

        It's still recommended to build the C extension if you're using a transport other than RabbitMQ/Redis (or use forced execv for some other reason).

    - Pool now sets a `current_process().index` attribute that can be used to create as many log files as there are processes in the pool.

- Canvas: chord/group/chain no longer modifies the state when called

    Previously calling a chord/group/chain would modify the ids of subtasks so that:

    ```
    >>> c = chord([add.s(2, 2), add.s(4, 4)], xsum.s())
    >>> c()
    >>> c() <-- call again
    ```

    at the second time the ids for the tasks would be the same as in the previous invocation. This is now fixed, so that calling a subtask won't mutate any options.

- Canvas: Chaining a chord to another task now works (Issue #965).

- Worker: Fixed a bug where the request stack could be corrupted if relative imports are used.

    Problem usually manifested itself as an exception while trying to send a failed task result (`NoneType does not have id attribute`).

    Fix contributed by Sam Cooke.

- Tasks can now raise *Ignore* to skip updating states or events after return.

    Example:

    ```python
    from celery.exceptions import Ignore

    @task
    def custom_revokes():
        if redis.sismember('tasks.revoked', custom_revokes.request.id):
            raise Ignore()
    ```

- The worker now makes sure the request/task stacks aren't modified by the initial `Task.__call__`.

    This would previously be a problem if a custom task class defined `__call__` and also called `super()`.

- Because of problems the fast local optimization has been disabled, and can only be enabled by setting the `USE_FAST_LOCALS` attribute.

- Worker: Now sets a default socket timeout of 5 seconds at shutdown so that broken socket reads don't hinder proper shutdown (Issue #975).

- More fixes related to late eventlet/gevent patching.

- Documentation for settings out of sync with reality:

    - `CELERY_TASK_PUBLISH_RETRY`

        Documented as disabled by default, but it was enabled by default since 2.5 as stated by the 2.5 changelog.

    - `CELERY_TASK_PUBLISH_RETRY_POLICY`

        The default max_retries had been set to 100, but documented as being 3, and the interval_max was set to 1 but documented as 0.2. The default setting are now set to 3 and 0.2 as it was originally documented.

    Fix contributed by Matt Long.

- Worker: Log messages when connection established and lost have been improved.

- The repr of a Crontab schedule value of '0' should be '*' (Issue #972).

- Revoked tasks are now removed from reserved/active state in the worker (Issue #969)

    Fix contributed by Alexey Zatelepin.

- gevent: Now supports hard time limits using `gevent.Timeout`.

- Documentation: Links to init-scripts now point to the 3.0 branch instead of the development branch (master).

- Documentation: Fixed typo in signals user guide (Issue #986).

    `instance.app.queues` -> `instance.app.amqp.queues`.

- Eventlet/gevent: The worker didn't properly set the custom app for new greenlets.

- Eventlet/gevent: Fixed a bug where the worker could not recover from connection loss (Issue #959).

Also, because of a suspected bug in gevent the `BROKER_CONNECTION_TIMEOUT` setting has been disabled when using gevent

3.0.9

release-date 2012-08-31 06:00 p.m. BST

release-by Ask Solem

- Important note for users of Django and the database scheduler!

  Recently a timezone issue has been fixed for periodic tasks, but erroneous timezones could have already been stored in the database, so for the fix to work you need to reset the `last_run_at` fields.

  You can do this by executing the following command:

  ```
  $ python manage.py shell
  >>> from djcelery.models import PeriodicTask
  >>> PeriodicTask.objects.update(last_run_at=None)
  ```

  You also have to do this if you change the timezone or `CELERY_ENABLE_UTC` setting.

- Note about the `CELERY_ENABLE_UTC` setting.

  If you previously disabled this just to force periodic tasks to work with your timezone, then you're now *encouraged to re-enable it*.

- Now depends on Kombu 2.4.5 which fixes PyPy + Jython installation.

- Fixed bug with timezones when `CELERY_ENABLE_UTC` is disabled (Issue #952).

- Fixed a typo in the `celerybeat` upgrade mechanism (Issue #951).

- Make sure the *exc_info* argument to logging is resolved (Issue #899).

- Fixed problem with Python 3.2 and thread join timeout overflow (Issue #796).

- A test case was occasionally broken for Python 2.5.

- Unit test suite now passes for PyPy 1.9.

- App instances now supports the `with` statement.

  This calls the new *app.close()* method at exit, which cleans up after the app like closing pool connections.

  Note that this is only necessary when dynamically creating apps, for example "temporary" apps.

- Support for piping a subtask to a chain.

  For example:

  ```
  pipe = sometask.s() | othertask.s()
  new_pipe = mytask.s() | pipe
  ```

  Contributed by Steve Morin.

- Fixed problem with group results on non-pickle serializers.

  Fix contributed by Steeve Morin.

3.0.8

release-date 2012-08-29 05:00 p.m. BST

release-by Ask Solem

- Now depends on Kombu 2.4.4

- Fixed problem with amqplib and receiving larger message payloads (Issue #922).

    The problem would manifest itself as either the worker hanging, or occasionally a `Framing error` exception appearing.

    Users of the new `pyamqp://` transport must upgrade to amqp 0.9.3.

- Beat: Fixed another timezone bug with interval and Crontab schedules (Issue #943).

- Beat: The schedule file is now automatically cleared if the timezone is changed.

    The schedule is also cleared when you upgrade to 3.0.8 from an earlier version, this to register the initial timezone info.

- Events: The *worker-heartbeat* event now include processed and active count fields.

    Contributed by Mher Movsisyan.

- Fixed error with error email and new task classes (Issue #931).

- `BaseTask.__call__` is no longer optimized away if it has been monkey patched.

- Fixed shutdown issue when using gevent (Issue #911 & Issue #936).

    Fix contributed by Thomas Meson.

3.0.7

release-date 2012-08-24 05:00 p.m. BST

release-by Ask Solem

- Fixes several problems with periodic tasks and timezones (Issue #937).

- Now depends on kombu 2.4.2

    - Redis: Fixes a race condition crash

    - Fixes an infinite loop that could happen when retrying establishing the broker connection.

- Daemons now redirect standard file descriptors to `/dev/null`

    Though by default the standard outs are also redirected to the logger instead, but you can disable this by changing the `CELERY_REDIRECT_STDOUTS` setting.

- Fixes possible problems when eventlet/gevent is patched too late.

- `LoggingProxy` no longer defines `fileno()` (Issue #928).

- Results are now ignored for the chord unlock task.

    Fix contributed by Steeve Morin.

- Cassandra backend now works if result expiry is disabled.

    Fix contributed by Steeve Morin.

- The traceback object is now passed to signal handlers instead of the string representation.

    Fix contributed by Adam DePue.

- Celery command: Extensions are now sorted by name.

- A regression caused the `task-failed` event to be sent with the exception object instead of its string representation.

- The worker daemon would try to create the pid file before daemonizing to catch errors, but this file wasn't immediately released (Issue #923).

- Fixes Jython compatibility.

- `billiard.forking_enable` was called by all pools not just the processes pool, which would result in a useless warning if the billiard C extensions weren't installed.

3.0.6

release-date 2012-08-17 11:00 p.mp.m. Ask Solem

- Now depends on kombu 2.4.0

- Now depends on billiard 2.7.3.12

- Redis: Celery now tries to restore messages whenever there are no messages in the queue.

- Crontab schedules now properly respects `CELERY_TIMEZONE` setting.

    It's important to note that Crontab schedules uses UTC time by default unless this setting is set.

    Issue #904 and [django-celery](#) #150.

- `billiard.enable_forking` is now only set by the processes pool.

- The transport is now properly shown by **celery report** (Issue #913).

- The *–app* argument now works if the last part is a module name (Issue #921).

- Fixed problem with unpickleable exceptions (billiard #12).

- Adds `task_name` attribute to `EagerResult` which is always `None` (Issue #907).

- Old Task class in `celery.task` no longer accepts magic kwargs by default (Issue #918).

    A regression long ago disabled magic kwargs for these, and since no one has complained about it we don't have any incentive to fix it now.

- The `inspect reserved` control command didn't work properly.

- Should now play better with tools for static analysis by explicitly specifying dynamically created attributes in the `celery` and `celery.task` modules.

- Terminating a task now results in `RevokedTaskError` instead of a `WorkerLostError`.

- `AsyncResult.revoke` now accepts `terminate` and `signal` arguments.

- The `task-revoked` event now includes new fields: `terminated`, `signum`, and `expired`.

- The argument to `TaskRevokedError` is now one of the reasons `revoked`, `expired` or `terminated`.

- Old Task class does no longer use `classmethod` for `push_request` and `pop_request` (Issue #912).

- `GroupResult` now supports the `children` attribute (Issue #916).

- `AsyncResult.collect` now respects the `intermediate` argument (Issue #917).

- Fixes example task in documentation (Issue #902).

- Eventlet fixed so that the environment is patched as soon as possible.

- eventlet: Now warns if Celery related modules that depends on threads are imported before eventlet is patched.

- Improved event and camera examples in the monitoring guide.

- Disables celery command setuptools entry-points if the command can't be loaded.

- Fixed broken `dump_request` example in the tasks guide.

3.0.5

release-date 2012-08-01 04:00 p.m. BST

release-by Ask Solem

- Now depends on kombu 2.3.1 + billiard 2.7.3.11

- Fixed a bug with the -B option (`cannot pickle thread.lock objects`) (Issue #894 + Issue #892, + [django-celery](#) #154).

- The `restart_pool` control command now requires the CELERYD_POOL_RESTARTS setting to be enabled

  This change was necessary as the multiprocessing event that the restart command depends on is responsible for creating many semaphores/file descriptors, resulting in problems in some environments.

- `chain.apply` now passes args to the first task (Issue #889).

- Documented previously secret options to the [django-celery](#) monitor in the monitoring user guide (Issue #396).

- Old changelog are now organized in separate documents for each series, see *History*.

3.0.4

release-date 2012-07-26 07:00 p.m. BST

release-by Ask Solem

- Now depends on Kombu 2.3

- New experimental standalone Celery monitor: Flower

  See *Flower: Real-time Celery web-monitor* to read more about it!

  Contributed by Mher Movsisyan.

- Now supports AMQP heartbeats if using the new `pyamqp://` transport.

  – The [amqp](#) transport requires the [amqp](#) library to be installed:

    ```
    $ pip install amqp
    ```

  – Then you need to set the transport URL prefix to `pyamqp://`.

  – The default heartbeat value is 10 seconds, but this can be changed using the BROKER_HEARTBEAT setting:

    ```
    BROKER_HEARTBEAT = 5.0
    ```

  – If the broker heartbeat is set to 10 seconds, the heartbeats will be monitored every 5 seconds (double the heartbeat rate).

  See the [Kombu 2.3 changelog](#) for more information.

- Now supports RabbitMQ Consumer Cancel Notifications, using the `pyamqp://` transport.

This is essential when running RabbitMQ in a cluster.

See the Kombu 2.3 changelog for more information.

- Delivery info is no longer passed directly through.

    It was discovered that the SQS transport adds objects that can't be pickled to the delivery info mapping, so we had to go back to using the white-list again.

    Fixing this bug also means that the SQS transport is now working again.

- The semaphore wasn't properly released when a task was revoked (Issue #877).

    This could lead to tasks being swallowed and not released until a worker restart.

    Thanks to Hynek Schlawack for debugging the issue.

- Retrying a task now also forwards any linked tasks.

    This means that if a task is part of a chain (or linked in some other way) and that even if the task is retried, then the next task in the chain will be executed when the retry succeeds.

- Chords: Now supports setting the interval and other keyword arguments to the chord unlock task.

    - The interval can now be set as part of the chord subtasks kwargs:

    ```
    chord(header)(body, interval=10.0)
    ```

    - In addition the chord unlock task now honors the Task.default_retry_delay option, used when none is specified, which also means that the default interval can also be changed using annotations:

    ```
    CELERY_ANNOTATIONS = {
        'celery.chord_unlock': {
            'default_retry_delay': 10.0,
        }
    }
    ```

- New *app.add_defaults()* method can add new default configuration dictionaries to the applications configuration.

    For example:

    ```
    config = {'FOO': 10}

    app.add_defaults(config)
    ```

    is the same as `app.conf.update(config)` except that data won't be copied, and that it won't be pickled when the worker spawns child processes.

    In addition the method accepts a callable:

    ```
    def initialize_config():
        # insert heavy stuff that can't be done at import time here.

    app.add_defaults(initialize_config)
    ```

    which means the same as the above except that it won't happen until the Celery configuration is actually used.

    As an example, Celery can lazily use the configuration of a Flask app:

```
flask_app = Flask()
app = Celery()
app.add_defaults(lambda: flask_app.config)
```

- Revoked tasks weren't marked as revoked in the result backend (Issue #871).

  Fix contributed by Hynek Schlawack.

- Event-loop now properly handles the case when the `epoll` poller object has been closed (Issue #882).

- Fixed syntax error in `funtests/test_leak.py`

  Fix contributed by Catalin Iacob.

- group/chunks: Now accepts empty task list (Issue #873).

- New method names:

  - `Celery.default_connection()` *connection_or_acquire()*.

  - `Celery.default_producer()` *producer_or_acquire()*.

  The old names still work for backward compatibility.

3.0.3

release-date 2012-07-20 09:17 p.m. BST

release-by Ask Solem

- amqplib passes the channel object as part of the delivery_info and it's not pickleable, so we now remove it.

3.0.2

release-date 2012-07-20 04:00 p.m. BST

release-by Ask Solem

- **A bug caused the following task options to not take defaults from the** configuration (Issue #867 + Issue #858)

  The following settings were affected:

  - `CELERY_IGNORE_RESULT`

  - `CELERYD_SEND_TASK_ERROR_EMAILS`

  - `CELERY_TRACK_STARTED`

  - `CElERY_STORE_ERRORS_EVEN_IF_IGNORED`

  Fix contributed by John Watson.

- Task Request: `delivery_info` is now passed through as-is (Issue #807).

- The ETA argument now supports datetime's with a timezone set (Issue #855).

- The worker's banner displayed the autoscale settings in the wrong order (Issue #859).

- Extension commands are now loaded after concurrency is set up so that they don't interfere with things like eventlet patching.

- Fixed bug in the threaded pool (Issue #863)

- The task failure handler mixed up the fields in `sys.exc_info()`.

> Fix contributed by Rinat Shigapov.

- Fixed typos and wording in the docs.

  > Fix contributed by Paul McMillan

- New setting: `CELERY_WORKER_DIRECT`

  > If enabled each worker will consume from their own dedicated queue which can be used to route tasks to specific workers.

- Fixed several edge case bugs in the add consumer remote control command.

- *migrate*: Can now filter and move tasks to specific workers if `CELERY_WORKER_DIRECT` is enabled.

  > Among other improvements, the following functions have been added:
  >
  > - `move_direct(filterfun, **opts)`
  >
  > - `move_direct_by_id(task_id, worker_hostname, **opts)`
  >
  > - `move_direct_by_idmap({task_id:  worker_hostname, ...}, **opts)`
  >
  > - `move_direct_by_taskmap({task_name:  worker_hostname, ...}, **opts)`

- `default_connection()` now accepts a pool argument that if set to false causes a new connection to be created instead of acquiring one from the pool.

- New signal: *celeryd_after_setup*.

- Default loader now keeps lowercase attributes from the configuration module.

3.0.1

release-date 2012-07-10 06:00 p.m. BST

release-by Ask Solem

- Now depends on kombu 2.2.5

- inspect now supports limit argument:

```
myapp.control.inspect(limit=1).ping()
```

- Beat: now works with timezone aware datetime's.

- Task classes inheriting `from celery import Task` mistakenly enabled `accept_magic_kwargs`.

- Fixed bug in `inspect scheduled` (Issue #829).

- Beat: Now resets the schedule to upgrade to UTC.

- The **celery worker** command now works with eventlet/gevent.

  > Previously it wouldn't patch the environment early enough.

- The **celery** command now supports extension commands using setuptools entry-points.

  > Libraries can add additional commands to the **celery** command by adding an entry-point like:

```
setup(
    entry_points=[
        'celery.commands': [
            'foo = my.module:Command',
        ],
],
...)
```

The command must then support the interface of `celery.bin.base.Command`.

- contrib.migrate: New utilities to move tasks from one queue to another.

    - `move_tasks()`

    - *`move_task_by_id()`*

- The *`task-sent`* event now contains `exchange` and `routing_key` fields.

- Fixes bug with installing on Python 3.

    Fix contributed by Jed Smith.

3.0.0 (Chiastic Slide)

release-date 2012-07-07 01:30 p.m. BST

release-by Ask Solem

See *What's new in Celery 3.0 (Chiastic Slide)*.

3.13.11 What's new in Celery 2.5

Celery aims to be a flexible and reliable, best-of-breed solution to process vast amounts of messages in a distributed fashion, while providing operations with the tools to maintain such a system.

Celery has a large and diverse community of users and contributors, you should come join us *on IRC* or *our mailing-list*.

To read more about Celery you should visit our website.

While this version is backward compatible with previous versions it's important that you read the following section.

If you use Celery in combination with Django you must also read the *django-celery changelog <djcelery:version-2.5.0>* and upgrade to django-celery 2.5.

This version is officially supported on CPython 2.5, 2.6, 2.7, 3.2 and 3.3, as well as PyPy and Jython.

- *Important Notes*

    - *Broker connection pool now enabled by default*

    - *Rabbit Result Backend: Exchange is no longer* auto delete

    - *Solution for hanging workers (but must be manually enabled)*

- *Optimization*

- *Deprecation Time-line Changes*

    - *Removals*

Important Notes

Broker connection pool now enabled by default

The default limit is 10 connections, if you have many threads/green-threads using connections at the same time you may want to tweak this limit to avoid contention.

See the `BROKER_POOL_LIMIT` setting for more information.

Also note that publishing tasks will be retried by default, to change this default or the default retry policy see `CELERY_TASK_PUBLISH_RETRY` and `CELERY_TASK_PUBLISH_RETRY_POLICY`.

Rabbit Result Backend: Exchange is no longer *auto delete*

The exchange used for results in the Rabbit (AMQP) result backend used to have the *auto_delete* flag set, which could result in a race condition leading to an annoying warning.

For RabbitMQ users

Old exchanges created with the *auto_delete* flag enabled has to be removed.

The **camqadm** command can be used to delete the previous exchange:

```
$ camqadm exchange.delete celeryresults
```

As an alternative to deleting the old exchange you can configure a new name for the exchange:

```
CELERY_RESULT_EXCHANGE = 'celeryresults2'
```

But you have to make sure that all clients and workers use this new setting, so they're updated to use the same exchange name.

Solution for hanging workers (but must be manually enabled)

The *CELERYD_FORCE_EXECV* setting has been added to solve a problem with deadlocks that originate when threads and fork is mixed together:

```
CELERYD_FORCE_EXECV = True
```

This setting is recommended for all users using the prefork pool, but especially users also using time limits or a max tasks per child setting.

- See Python Issue 6721 to read more about this issue, and why resorting to `execv`() is the only safe solution.

Enabling this option will result in a slight performance penalty when new child worker processes are started, and it will also increase memory usage (but many platforms are optimized, so the impact may be minimal). Considering that it ensures reliability when replacing lost worker processes, it should be worth it.

- It's already the default behavior on Windows.

- It will be the default behavior for all platforms in a future version.

Optimization

- The code path used when the worker executes a task has been heavily optimized, meaning the worker is able to process a great deal more tasks/second compared to previous versions. As an example the solo pool can now process up to 15000 tasks/second on a 4 core MacBook Pro when using the pylibrabbitmq transport, where it previously could only do 5000 tasks/second.

- The task error tracebacks are now much shorter.

- Fixed a noticeable delay in task processing when rate limits are enabled.

Deprecation Time-line Changes

Removals

- The old `TaskSet` signature of `(task_name, list_of_tasks)` can no longer be used (originally scheduled for removal in 2.4). The deprecated `.task_name` and `.task` attributes has also been removed.

- The functions `celery.execute.delay_task`, `celery.execute.apply`, and `celery.execute.apply_async` has been removed (originally) scheduled for removal in 2.3).

- The built-in `ping` task has been removed (originally scheduled for removal in 2.3). Please use the ping broadcast command instead.

- It's no longer possible to import `subtask` and `TaskSet` from `celery.task.base`, please import them from `celery.task` instead (originally scheduled for removal in 2.4).

Deprecated modules

- The `celery.decorators` module has changed status from pending deprecation to deprecated, and is scheduled for removal in version 4.0. The `celery.task` module must be used instead.

News

Timezone support

Celery can now be configured to treat all incoming and outgoing dates as UTC, and the local timezone can be configured.

This isn't yet enabled by default, since enabling time zone support means workers running versions pre-2.5 will be out of sync with upgraded workers.

To enable UTC you have to set `CELERY_ENABLE_UTC`:

```
CELERY_ENABLE_UTC = True
```

When UTC is enabled, dates and times in task messages will be converted to UTC, and then converted back to the local timezone when received by a worker.

You can change the local timezone using the CELERY_TIMEZONE setting. Installing the pytz library is recommended when using a custom timezone, to keep timezone definition up-to-date, but it will fallback to a system definition of the timezone if available.

UTC will enabled by default in version 3.0.

Note: django-celery will use the local timezone as specified by the TIME_ZONE setting, it will also honor the new USE_TZ setting introduced in Django 1.4.

New security serializer using cryptographic signing

A new serializer has been added that signs and verifies the signature of messages.

The name of the new serializer is auth, and needs additional configuration to work (see *Security*).

See also:

Security

Contributed by Mher Movsisyan.

New `CELERY_ANNOTATIONS` setting

This new setting enables the configuration to modify task classes and their attributes.

The setting can be a dict, or a list of annotation objects that filter for tasks and return a map of attributes to change.

As an example, this is an annotation to change the rate_limit attribute for the tasks.add task:

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

or change the same for all tasks:

```
CELERY_ANNOTATIONS = {'*': {'rate_limit': '10/s'}}
```

You can change methods too, for example the on_failure handler:

```
def my_on_failure(self, exc, task_id, args, kwargs, einfo):
    print('Oh no! Task failed: %r' % (exc,))

CELERY_ANNOTATIONS = {'*': {'on_failure': my_on_failure}}
```

If you need more flexibility then you can also create objects that filter for tasks to annotate:

```
class MyAnnotate(object):

    def annotate(self, task):
        if task.name.startswith('tasks.'):
            return {'rate_limit': '10/s'}

CELERY_ANNOTATIONS = (MyAnnotate(), {other_annotations,})
```

`current` provides the currently executing task

The new `celery.task.current` proxy will always give the currently executing task.

Example:

```python
from celery.task import current, task

@task
def update_twitter_status(auth, message):
    twitter = Twitter(auth)
    try:
        twitter.update_status(message)
    except twitter.FailWhale, exc:
        # retry in 10 seconds.
        current.retry(countdown=10, exc=exc)
```

Previously you'd've to type `update_twitter_status.retry(...)` here, which can be annoying for long task names.

Note: This won't work if the task function is called directly (i.e., `update_twitter_status(a, b)`). For that to work `apply` must be used: `update_twitter_status.apply((a, b))`.

In Other News

- Now depends on Kombu 2.1.0.
- Efficient Chord support for the Memcached backend (Issue #533)

  This means Memcached joins Redis in the ability to do non-polling chords.

  Contributed by Dan McGee.

- Adds Chord support for the Rabbit result backend (amqp)

  The Rabbit result backend can now use the fallback chord solution.

- Sending `QUIT` to `celeryd` will now cause it cold terminate.

  That is, it won't finish executing the tasks it's currently working on.

  Contributed by Alec Clowes.

- New "detailed" mode for the Cassandra backend.

  Allows to have a "detailed" mode for the Cassandra backend. Basically the idea is to keep all states using Cassandra wide columns. New states are then appended to the row as new columns, the last state being the last column.

  See the `CASSANDRA_DETAILED_MODE` setting.

  Contributed by Steeve Morin.

- The Crontab parser now matches Vixie Cron behavior when parsing ranges with steps (e.g., 1-59/2).

  Contributed by Daniel Hepper.

- `celerybeat` can now be configured on the command-line like `celeryd`.

  Additional configuration must be added at the end of the argument list followed by `--`, for example:

```
$ celerybeat -l info -- celerybeat.max_loop_interval=10.0
```

- Now limits the number of frames in a traceback so that `celeryd` doesn't crash on maximum recursion limit exceeded exceptions (Issue #615).

    The limit is set to the current recursion limit divided by 8 (which is 125 by default).

    To get or set the current recursion limit use `sys.getrecursionlimit()` and `sys.setrecursionlimit()`.

- More information is now preserved in the pickleable traceback.

    This has been added so that Sentry can show more details.

    Contributed by Sean O'Connor.

- CentOS init-script has been updated and should be more flexible.

    Contributed by Andrew McFague.

- MongoDB result backend now supports `forget()`.

    Contributed by Andrew McFague

- `task.retry()` now re-raises the original exception keeping the original stack trace.

    Suggested by @ojii.

- The *–uid* argument to daemons now uses `initgroups()` to set groups to all the groups the user is a member of.

    Contributed by Łukasz Oleś.

- `celeryctl`: Added `shell` command.

    The shell will have the current_app (`celery`) and all tasks automatically added to locals.

- `celeryctl`: Added `migrate` command.

    The migrate command moves all tasks from one broker to another. Note that this is experimental and you should have a backup of the data before proceeding.

    Examples:

    ```
    $ celeryctl migrate redis://localhost amqp://localhost
    $ celeryctl migrate amqp://localhost//v1 amqp://localhost//v2
    $ python manage.py celeryctl migrate django:// redis://
    ```

- Routers can now override the `exchange` and `routing_key` used to create missing queues (Issue #577).

    By default this will always use the name of the queue, but you can now have a router return exchange and routing_key keys to set them.

    This is useful when using routing classes which decides a destination at run-time.

    Contributed by Akira Matsuzaki.

- Redis result backend: Adds support for a `max_connections` parameter.

    It's now possible to configure the maximum number of simultaneous connections in the Redis connection pool used for results.

    The default max connections setting can be configured using the `CELERY_REDIS_MAX_CONNECTIONS` setting, or it can be changed individually by `RedisBackend(max_connections=int)`.

Contributed by Steeve Morin.

- Redis result backend: Adds the ability to wait for results without polling.

    Contributed by Steeve Morin.

- MongoDB result backend: Now supports save and restore `taskset`.

    Contributed by Julien Poissonnier.

- There's a new *Security* guide in the documentation.

- The init-scripts have been updated, and many bugs fixed.

    Contributed by Chris Streeter.

- User (tilde) is now expanded in command-line arguments.

- Can now configure `CELERYCTL` environment variable in `/etc/default/celeryd`.

    While not necessary for operation, **celeryctl** is used for the `celeryd status` command, and the path to **celeryctl** must be configured for that to work.

    The daemonization cookbook contains examples.

    Contributed by Jude Nagurney.

- The MongoDB result backend can now use Replica Sets.

    Contributed by Ivan Metzlar.

- gevent: Now supports autoscaling (Issue #599).

    Contributed by Mark Lavin.

- multiprocessing: Mediator thread is now always enabled, even though rate limits are disabled, as the pool semaphore is known to block the main thread, causing broadcast commands and shutdown to depend on the semaphore being released.

Fixes

- Exceptions that are re-raised with a new exception object now keeps the original stack trace.

- Windows: Fixed the `no handlers found for multiprocessing` warning.

- Windows: The `celeryd` program can now be used.

    Previously Windows users had to launch `celeryd` using `python -m celery.bin.celeryd`.

- Redis result backend: Now uses `SETEX` command to set result key, and expiry atomically.

    Suggested by @yaniv-aknin.

- `celeryd`: Fixed a problem where shutdown hanged when `Control-c` was used to terminate.

- `celeryd`: No longer crashes when channel errors occur.

    Fix contributed by Roger Hu.

- Fixed memory leak in the eventlet pool, caused by the use of `greenlet.getcurrent`.

    Fix contributed by Ignas Mikalajūnas.

- Cassandra backend: No longer uses `pycassa.connect()` which is deprecated since pycassa 1.4.

    Fix contributed by Jeff Terrace.

- Fixed unicode decode errors that could occur while sending error emails.

    Fix contributed by Seong Wun Mun.

- `celery.bin` programs now always defines `__package__` as recommended by PEP-366.

- `send_task` now emits a warning when used in combination with `CELERY_ALWAYS_EAGER` (Issue #581).

    Contributed by Mher Movsisyan.

- `apply_async` now forwards the original keyword arguments to `apply` when `CELERY_ALWAYS_EAGER` is enabled.

- `celeryev` now tries to re-establish the connection if the connection to the broker is lost (Issue #574).

- `celeryev`: Fixed a crash occurring if a task has no associated worker information.

    Fix contributed by Matt Williamson.

- The current date and time is now consistently taken from the current loaders `now` method.

- Now shows helpful error message when given a configuration module ending in `.py` that can't be imported.

- `celeryctl`: The `--expires` and `--eta` arguments to the apply command can now be an ISO-8601 formatted string.

- `celeryctl` now exits with exit status `EX_UNAVAILABLE` (69) if no replies have been received.

3.13.12 Change history for Celery 2.5

This document contains change notes for bugfix releases in the 2.5.x series, please see *What's new in Celery 2.5* for an overview of what's new in Celery 2.5.

If you're looking for versions prior to 2.5 you should visit our *History* of releases.

- *2.5.5*

- *2.5.3*

- *2.5.2*

    - *News*

    - *Fixes*

- *2.5.1*

    - *Fixes*

- *2.5.0*

2.5.5

release-date 2012-06-06 04:00 p.m. BST

release-by Ask Solem

This is a dummy release performed for the following goals:

- Protect against force upgrading to Kombu 2.2.0

- Version parity with django-celery

2.5.3

release-date 2012-04-16 07:00 p.m. BST

release-by Ask Solem

- A bug causes messages to be sent with UTC time-stamps even though `CELERY_ENABLE_UTC` wasn't enabled (Issue #636).

- `celerybeat`: No longer crashes if an entry's args is set to None (Issue #657).

- Auto-reload didn't work if a module's `__file__` attribute was set to the modules `.pyc` file. (Issue #647).

- Fixes early 2.5 compatibility where `__package__` doesn't exist (Issue #638).

2.5.2

release-date 2012-04-13 04:30 p.m. GMT

release-by Ask Solem

News

- Now depends on Kombu 2.1.5.

- Django documentation has been moved to the main Celery docs.

    See *Django*.

- New *celeryd_init* signal can be used to configure workers by hostname.

- Signal.connect can now be used as a decorator.

    Example:

    ```python
    from celery.signals import task_sent

    @task_sent.connect
    def on_task_sent(**kwargs):
        print('sent task: %r' % (kwargs,))
    ```

- Invalid task messages are now rejected instead of acked.

    This means that they will be moved to the dead-letter queue introduced in the latest RabbitMQ version (but must be enabled manually, consult the RabbitMQ documentation).

- Internal logging calls has been cleaned up to work better with tools like Sentry.

    Contributed by David Cramer.

- New method `subtask.clone()` can be used to clone an existing subtask with augmented arguments/options.

    Example:

    ```python
    >>> s = add.subtask((5,))
    >>> new = s.clone(args=(10,), countdown=5})
    >>> new.args
    (10, 5)

    >>> new.options
    {'countdown': 5}
    ```

• Chord callbacks are now triggered in eager mode.

Fixes

• Programs now verifies that the pidfile is actually written correctly (Issue #641).

> Hopefully this will crash the worker immediately if the system is out of space to store the complete pidfile.
>
> In addition, we now verify that existing pidfiles contain a new line so that a partially written pidfile is detected as broken, as before doing:

```
$ echo -n "1" > celeryd.pid
```

> would cause the worker to think that an existing instance was already running (init has pid 1 after all).

• Fixed 2.5 compatibility issue with use of print_exception.

> Fix contributed by Martin Melin.

• Fixed 2.5 compatibility issue with imports.

> Fix contributed by Iurii Kriachko.

• All programs now fix up __package__ when called as main.

> This fixes compatibility with Python 2.5.
>
> Fix contributed by Martin Melin.

• [celery control|inspect] can now be configured on the command-line.

> Like with the worker it is now possible to configure Celery settings on the command-line for celery control|inspect

```
$ celery inspect -- broker.pool_limit=30
```

• Version dependency for python-dateutil fixed to be strict.

> Fix contributed by Thomas Meson.

• `Task.__call__` is now optimized away in the task tracer rather than when the task class is created.

> This fixes a bug where a custom __call__ may mysteriously disappear.

• Auto-reload's `inotify` support has been improved.

> Contributed by Mher Movsisyan.

• The Django broker documentation has been improved.

• Removed confusing warning at top of routing user guide.

2.5.1

release-date 2012-03-01 01:00 p.m. GMT

release-by Ask Solem

Fixes

- Eventlet/Gevent: A small typo caused the worker to hang when eventlet/gevent was used, this was because the environment wasn't monkey patched early enough.

- Eventlet/Gevent: Another small typo caused the mediator to be started with eventlet/gevent, which would make the worker sometimes hang at shutdown.

- `multiprocessing`: Fixed an error occurring if the pool was stopped before it was properly started.

- Proxy objects now redirects `__doc__` and `__name__` so `help(obj)` works.

- Internal timer (timer2) now logs exceptions instead of swallowing them (Issue #626).

- celery shell: can now be started with `--eventlet` or `--gevent` options to apply their monkey patches.

2.5.0

release-date 2012-02-24 04:00 p.m. GMT

release-by Ask Solem

See *What's new in Celery 2.5*.

Since the changelog has gained considerable size, we decided to do things differently this time: by having separate "what's new" documents for major version changes.

Bugfix releases will still be found in the changelog.

3.13.13 Change history for Celery 2.4

2.4.5

release-date 2011-12-02 05:00 p.m. GMT

release-by Ask Solem

- Periodic task interval schedules were accidentally rounded down, resulting in some periodic tasks being executed early.

- Logging of humanized times in the beat log is now more detailed.

- New *Brokers* section in the Getting Started part of the Documentation

    This replaces the old "Other queues" tutorial, and adds documentation for MongoDB, Beanstalk and CouchDB.

2.4.4

release-date 2011-11-25 04:00 p.m. GMT

release-by Ask Solem

Security Fixes

- [Security: CELERYSA-0001] Daemons would set effective id's rather than real id's when the `--uid`/ `--gid` arguments to `celery multi`, `celeryd_detach`, `celery beat` and `celery events` were used.

    This means privileges weren't properly dropped, and that it would be possible to regain supervisor privileges later.

Fixes

- Processes pool: Fixed rare deadlock at shutdown (Issue #523).

    Fix contributed by Ionel Maries Christian.

- Webhook tasks issued the wrong HTTP POST headers (Issue #515).

    The *Content-Type* header has been changed from `application/json application/x-www-form-urlencoded`, and adds a proper *Content-Length* header.

    Fix contributed by Mitar.

- Daemonization tutorial: Adds a configuration example using Django and virtualenv together (Issue #505).

    Contributed by Juan Ignacio Catalano.

- generic init-scripts now automatically creates log and pid file directories (Issue #545).

    Contributed by Chris Streeter.

2.4.3

release-date 2011-11-22 06:00 p.m. GMT

release-by Ask Solem

- Fixes module import typo in *celeryctl* (Issue #538).

    Fix contributed by Chris Streeter.

2.4.2

release-date 2011-11-14 12:00 p.m. GMT

release-by Ask Solem

- Program module no longer uses relative imports so that it's possible to do `python -m celery.bin.name`.

2.4.1

release-date 2011-11-07 06:00 p.m. GMT

release-by Ask Solem

- `celeryctl inspect` commands was missing output.
- processes pool: Decrease polling interval for less idle CPU usage.
- processes pool: MaybeEncodingError wasn't wrapped in ExceptionInfo (Issue #524).
- worker: would silence errors occurring after task consumer started.
- logging: Fixed a bug where unicode in stdout redirected log messages couldn't be written (Issue #522).

2.4.0

release-date 2011-11-04 04:00 p.m. GMT

release-by Ask Solem

Important Notes

- Now supports Python 3.
- Fixed deadlock in worker process handling (Issue #496).

    A deadlock could occur after spawning new child processes because the logging library's mutex wasn't properly reset after fork.

    The symptoms of this bug affecting would be that the worker simply stops processing tasks, as none of the workers child processes are functioning. There was a greater chance of this bug occurring with `maxtasksperchild` or a time-limit enabled.

    This is a workaround for http://bugs.python.org/issue6721#msg140215.

    Be aware that while this fixes the logging library lock, there could still be other locks initialized in the parent process, introduced by custom code.

    Fix contributed by Harm Verhagen.

- AMQP Result backend: Now expires results by default.

    The default expiration value is now taken from the `CELERY_TASK_RESULT_EXPIRES` setting.

    The old `CELERY_AMQP_TASK_RESULT_EXPIRES` setting has been deprecated and will be removed in version 4.0.

    Note that this means that the result backend requires RabbitMQ 2.1.0 or higher, and that you have to disable expiration if you're running with an older version. You can do so by disabling the `CELERY_TASK_RESULT_EXPIRES` setting:

```
CELERY_TASK_RESULT_EXPIRES = None
```

- Eventlet: Fixed problem with shutdown (Issue #457).

- Broker transports can be now be specified using URLs

    The broker can now be specified as a URL instead. This URL must have the format:

    ```
    transport://user:password@hostname:port/virtual_host
    ```

    for example the default broker is written as:

    ```
    amqp://guest:guest@localhost:5672//
    ```

    The scheme is required, so that the host is identified as a URL and not just a host name. User, password, port and virtual_host are optional and defaults to the particular transports default value.

    Note: Note that the path component (virtual_host) always starts with a forward-slash. This is necessary to distinguish between the virtual host `''` (empty) and `'/'`, which are both acceptable virtual host names.

    A virtual host of `'/'` becomes:

    ```
    amqp://guest:guest@localhost:5672//
    ```

    and a virtual host of `''` (empty) becomes:

    ```
    amqp://guest:guest@localhost:5672/
    ```

    So the leading slash in the path component is **always required**.

    In addition the `BROKER_URL` setting has been added as an alias to `BROKER_HOST`. Any broker setting specified in both the URL and in the configuration will be ignored, if a setting isn't provided in the URL then the value from the configuration will be used as default.

    Also, programs now support the `--broker` option to specify a broker URL on the command-line:

    ```
    $ celery worker -b redis://localhost

    $ celery inspect -b amqp://guest:guest@localhost//e
    ```

    The environment variable `CELERY_BROKER_URL` can also be used to easily override the default broker used.

- The deprecated `celery.loaders.setup_loader()` function has been removed.

- The `CELERY_TASK_ERROR_WHITELIST` setting has been replaced by a more flexible approach (Issue #447).

    The error mail sending logic is now available as `Task.ErrorMail`, with the implementation (for reference) in `celery.utils.mail`.

    The error mail class can be sub-classed to gain complete control of when error messages are sent, thus removing the need for a separate white-list setting.

    The `CELERY_TASK_ERROR_WHITELIST` setting has been deprecated, and will be removed completely in version 4.0.

- Additional Deprecations

The following functions has been deprecated and is scheduled for removal in version 4.0:

Old function	Alternative
celery.loaders.current_loader	*celery.current_app.loader*
celery.loaders.load_settings	*celery.current_app.conf*
celery.execute.apply	*Task.apply*
celery.execute.apply_async	*Task.apply_async*
celery.execute.delay_task	*celery.execute.send_task*

The following settings has been deprecated and is scheduled for removal in version 4.0:

Old setting	Alternative
CELERYD_LOG_LEVEL	`celery worker --loglevel=`
CELERYD_LOG_FILE	`celery worker --logfile=`
CELERYBEAT_LOG_LEVEL	`celery beat --loglevel=`
CELERYBEAT_LOG_FILE	`celery beat --logfile=`
CELERYMON_LOG_LEVEL	`celerymon --loglevel=`
CELERYMON_LOG_FILE	`celerymon --logfile=`

News

- No longer depends on pyparsing.

- Now depends on Kombu 1.4.3.

- CELERY_IMPORTS can now be a scalar value (Issue #485).

  It's too easy to forget to add the comma after the sole element of a tuple, and this is something that often affects newcomers.

  The docs should probably use a list in examples, as using a tuple for this doesn't even make sense. Nonetheless, there are many tutorials out there using a tuple, and this change should be a help to new users.

  Suggested by @jsaxon-cars.

- Fixed a memory leak when using the thread pool (Issue #486).

  Contributed by Kornelijus Survila.

- The `statedb` wasn't saved at exit.

  This has now been fixed and it should again remember previously revoked tasks when a `--statedb` is enabled.

- Adds `EMAIL_USE_TLS` to enable secure SMTP connections (Issue #418).

  Contributed by Stefan Kjartansson.

- Now handles missing fields in task messages as documented in the message format documentation.

  - Missing required field throws *InvalidTaskError*

  - Missing args/kwargs is assumed empty.

  Contributed by Chris Chamberlin.

- Fixed race condition in *celery.events.state* (celerymon/celeryev) where task info would be removed while iterating over it (Issue #501).

- The Cache, Cassandra, MongoDB, Redis and Tyrant backends now respects the `CELERY_RESULT_SERIALIZER` setting (Issue #435).

  This means that only the database (Django/SQLAlchemy) backends currently doesn't support using custom serializers.

  Contributed by Steeve Morin

- Logging calls no longer manually formats messages, but delegates that to the logging system, so tools like Sentry can easier work with the messages (Issue #445).

  Contributed by Chris Adams.

- `multi` now supports a `stop_verify` command to wait for processes to shutdown.

- Cache backend didn't work if the cache key was unicode (Issue #504).

  Fix contributed by Neil Chintomby.

- New setting `CELERY_RESULT_DB_SHORT_LIVED_SESSIONS` added, which if enabled will disable the caching of SQLAlchemy sessions (Issue #449).

  Contributed by Leo Dirac.

- All result backends now implements `__reduce__` so that they can be pickled (Issue #441).

  Fix contributed by Remy Noel

- multi didn't work on Windows (Issue #472).

- New-style `CELERY_REDIS_*` settings now takes precedence over the old `REDIS_*` configuration keys (Issue #508).

  Fix contributed by Joshua Ginsberg

- Generic beat init-script no longer sets *bash -e* (Issue #510).

  Fix contributed by Roger Hu.

- Documented that Chords don't work well with **redis-server** versions before 2.2.

  Contributed by Dan McGee.

- The `CELERYBEAT_MAX_LOOP_INTERVAL` setting wasn't respected.

- `inspect.registered_tasks` renamed to `inspect.registered` for naming consistency.

  The previous name is still available as an alias.

  Contributed by Mher Movsisyan

- Worker logged the string representation of args and kwargs without safe guards (Issue #480).

- RHEL init-script: Changed worker start-up priority.

  The default start / stop priorities for MySQL on RHEL are:

  ```
  # chkconfig: - 64 36
  ```

  Therefore, if Celery is using a database as a broker / message store, it should be started after the database is up and running, otherwise errors will ensue. This commit changes the priority in the init-script to:

  ```
  # chkconfig: - 85 15
  ```

which are the default recommended settings for 3-rd party applications and assure that Celery will be started after the database service & shut down before it terminates.

Contributed by Yury V. Zaytsev.

- KeyValueStoreBackend.get_many didn't respect the `timeout` argument (Issue #512).

- beat/events's `--workdir` option didn't `chdir(2)` before after configuration was attempted (Issue #506).

- After deprecating 2.4 support we can now name modules correctly, since we can take use of absolute imports.

  Therefore the following internal modules have been renamed:

  ```
  celery.concurrency.evlet  ->  celery.concurrency.eventlet
  celery.concurrency.evg -> celery.concurrency.gevent
  ```

- `AUTHORS` file is now sorted alphabetically.

  Also, as you may have noticed the contributors of new features/fixes are now mentioned in the Changelog.

3.13.14 Change history for Celery 2.3

2.3.4

release-date 2011-11-25 04:00 p.m. GMT

release-by Ask Solem

Security Fixes

- [Security: CELERYSA-0001] Daemons would set effective id's rather than real id's when the `--uid`/ `--gid` arguments to **celery multi**, **celeryd_detach**, **celery beat** and **celery events** were used.

This means privileges weren't properly dropped, and that it would be possible to regain supervisor privileges later.

Fixes

- Backported fix for #455 from 2.4 to 2.3.

- StateDB wasn't saved at shutdown.

- Fixes worker sometimes hanging when hard time limit exceeded.

2.3.3

release-date 2011-16-09 05:00 p.m. BST

release-by Mher Movsisyan

- Monkey patching `sys.stdout` could result in the worker crashing if the replacing object didn't define `isatty()` (Issue #477).

- `CELERYD` option in `/etc/default/celeryd` shouldn't be used with generic init-scripts.

2.3.2

release-date 2011-10-07 05:00 p.m. BST

release-by Ask Solem

News

- Improved Contributing guide.

    If you'd like to contribute to Celery you should read the *Contributing Gudie*.

    We're looking for contributors at all skill levels, so don't hesitate!

- Now depends on Kombu 1.3.1

- `Task.request` now contains the current worker host name (Issue #460).

    Available as `task.request.hostname`.

- **It's now easier for app subclasses to extend how they're pickled.** (see `celery.app.AppPickler`).

Fixes

- *purge/discard_all* wasn't working correctly (Issue #455).

- The coloring of log messages didn't handle non-ASCII data well (Issue #427).

- [Windows] the multiprocessing pool tried to import `os.kill` even though this isn't available there (Issue #450).

- Fixes case where the worker could become unresponsive because of tasks exceeding the hard time limit.

- The `task-sent` event was missing from the event reference.

- `ResultSet.iterate` now returns results as they finish (Issue #459).

>   This wasn't the case previously, even though the documentation states this was the expected behavior.

- Retries will no longer be performed when tasks are called directly (using `__call__`).

  Instead the exception passed to `retry` will be re-raised.

- Eventlet no longer crashes if autoscale is enabled.

  growing and shrinking eventlet pools is still not supported.

- `py24` target removed from `tox.ini`.

2.3.1

release-date 2011-08-07 08:00 p.m. BST

release-by Ask Solem

Fixes

- The `CELERY_AMQP_TASK_RESULT_EXPIRES` setting didn't work, resulting in an AMQP related error about not being able to serialize floats while trying to publish task states (Issue #446).

2.3.0

release-date 2011-08-05 12:00 p.m. BST

tested CPython: 2.5, 2.6, 2.7; PyPy: 1.5; Jython: 2.5.2

release-by Ask Solem

Important Notes

- Now requires Kombu 1.2.1

- Results are now disabled by default.

  The AMQP backend wasn't a good default because often the users were not consuming the results, resulting in thousands of queues.

  While the queues can be configured to expire if left unused, it wasn't possible to enable this by default because this was only available in recent RabbitMQ versions (2.1.1+)

  With this change enabling a result backend will be a conscious choice, which will hopefully lead the user to read the documentation and be aware of any common pitfalls with the particular backend.

  The default backend is now a dummy backend (`celery.backends.base.DisabledBackend`). Saving state is simply an no-op, and AsyncResult.wait(), .result, .state, etc. will raise a `NotImplementedError` telling the user to configure the result backend.

  For help choosing a backend please see *Result Backends*.

  If you depend on the previous default which was the AMQP backend, then you have to set this explicitly before upgrading:

  ```
  CELERY_RESULT_BACKEND = 'amqp'
  ```

Note: For django-celery users the default backend is still `database`, and results are not disabled by default.

- The Debian init-scripts have been deprecated in favor of the generic-init.d init-scripts.

  In addition generic init-scripts for `celerybeat` and `celeryev` has been added.

News

- Automatic connection pool support.

  The pool is used by everything that requires a broker connection, for example calling tasks, sending broadcast commands, retrieving results with the AMQP result backend, and so on.

  The pool is disabled by default, but you can enable it by configuring the `BROKER_POOL_LIMIT` setting:

  ```
  BROKER_POOL_LIMIT = 10
  ```

  A limit of 10 means a maximum of 10 simultaneous connections can co-exist. Only a single connection will ever be used in a single-thread environment, but in a concurrent environment (threads, greenlets, etc., but not processes) when the limit has been exceeded, any try to acquire a connection will block the thread and wait for a connection to be released. This is something to take into consideration when choosing a limit.

  A limit of `None` or 0 means no limit, and connections will be established and closed every time.

- Introducing Chords (taskset callbacks).

  A chord is a task that only executes after all of the tasks in a taskset has finished executing. It's a fancy term for "taskset callbacks" adopted from Cω).

  It works with all result backends, but the best implementation is currently provided by the Redis result backend.

  Here's an example chord:

  ```
  >>> chord(add.subtask((i, i))
  ...         for i in xrange(100))(tsum.subtask()).get()
  9900
  ```

  Please read the *Chords section in the user guide*, if you want to know more.

- Time limits can now be set for individual tasks.

  To set the soft and hard time limits for a task use the `time_limit` and `soft_time_limit` attributes:

  ```python
  import time

  @task(time_limit=60, soft_time_limit=30)
  def sleeptask(seconds):
      time.sleep(seconds)
  ```

  If the attributes are not set, then the workers default time limits will be used.

  New in this version you can also change the time limits for a task at runtime using the `time_limit()` remote control command:

```
>>> from celery.task import control
>>> control.time_limit('tasks.sleeptask',
...                     soft=60, hard=120, reply=True)
[{'worker1.example.com': {'ok': 'time limits set successfully'}}]
```

Only tasks that starts executing after the time limit change will be affected.

Note: Soft time limits will still not work on Windows or other platforms that don't have the SIGUSR1 signal.

- **Redis backend configuration directive names changed to include the** CELERY_ prefix.

Old setting name	Replace with
REDIS_HOST	*CELERY_REDIS_HOST*
REDIS_PORT	*CELERY_REDIS_PORT*
REDIS_DB	*CELERY_REDIS_DB*
REDIS_PASSWORD	*CELERY_REDIS_PASSWORD*

  The old names are still supported but pending deprecation.

- PyPy: The default pool implementation used is now multiprocessing if running on PyPy 1.5.

- multi: now supports "pass through" options.

  Pass through options makes it easier to use Celery without a configuration file, or just add last-minute options on the command line.

  Example use:

```
$ celery multi start 4 -c 2 -- broker.host=amqp.example.com \
                              broker.vhost=/              \
                              celery.disable_rate_limits=yes
```

- celerybeat: Now retries establishing the connection (Issue #419).

- celeryctl: New list bindings command.

  Lists the current or all available bindings, depending on the broker transport used.

- Heartbeat is now sent every 30 seconds (previously every 2 minutes).

- ResultSet.join_native() and iter_native() is now supported by the Redis and Cache result backends.

  This is an optimized version of join() using the underlying backends ability to fetch multiple results at once.

- Can now use SSL when sending error e-mails by enabling the EMAIL_USE_SSL setting.

- events.default_dispatcher(): Context manager to easily obtain an event dispatcher instance using the connection pool.

- Import errors in the configuration module won't be silenced anymore.

- ResultSet.iterate: Now supports the timeout, propagate and interval arguments.

- with_default_connection -> with default_connection

- TaskPool.apply_async: Keyword arguments callbacks and errbacks has been renamed to callback and errback and take a single scalar value instead of a list.

- No longer propagates errors occurring during process cleanup (Issue #365)

- Added `TaskSetResult.delete()`, which will delete a previously saved taskset result.

- `celerybeat` now syncs every 3 minutes instead of only at shutdown (Issue #382).

- Monitors now properly handles unknown events, so user-defined events are displayed.

- Terminating a task on Windows now also terminates all of the tasks child processes (Issue #384).

- worker: `-I|--include` option now always searches the current directory to import the specified modules.

- Cassandra backend: Now expires results by using TTLs.

- Functional test suite in `funtests` is now actually working properly, and passing tests.

Fixes

- `celeryev` was trying to create the pidfile twice.

- celery.contrib.batches: Fixed problem where tasks failed silently (Issue #393).

- Fixed an issue where logging objects would give "<Unrepresentable", even though the objects were.

- `CELERY_TASK_ERROR_WHITE_LIST` is now properly initialized in all loaders.

- `celeryd_detach` now passes through command line configuration.

- Remote control command `add_consumer` now does nothing if the queue is already being consumed from.

3.13.15 Change history for Celery 2.2

2.2.8

release-date 2011-11-25 04:00 p.m. GMT

release-by Ask Solem

Security Fixes

- [Security: CELERYSA-0001] Daemons would set effective id's rather than real id's when the `--uid`/ `--gid` arguments to **celery multi**, **celeryd_detach**, **celery beat** and **celery events** were used.

  This means privileges weren't properly dropped, and that it would be possible to regain supervisor privileges later.

2.2.7

release-date 2011-06-13 04:00 p.m. BST

release-by Ask Solem

- New signals: *after_setup_logger* and *after_setup_task_logger*

    These signals can be used to augment logging configuration after Celery has set up logging.

- Redis result backend now works with Redis 2.4.4.

- multi: The `--gid` option now works correctly.

- worker: Retry wrongfully used the repr of the traceback instead of the string representation.

- App.config_from_object: Now loads module, not attribute of module.

- Fixed issue where logging of objects would give "<Unrepresentable: . . .>"

2.2.6

release-date 2011-04-15 04:00 p.m. CEST

release-by Ask Solem

Important Notes

- Now depends on Kombu 1.1.2.
- Dependency lists now explicitly specifies that we don't want python-dateutil 2.x, as this version only supports Python 3.

  If you have installed dateutil 2.0 by accident you should downgrade to the 1.5.0 version:

  ```
  $ pip install -U python-dateutil==1.5.0
  ```

  or by `easy_install`:

  ```
  $ easy_install -U python-dateutil==1.5.0
  ```

Fixes

- The new `WatchedFileHandler` broke Python 2.5 support (Issue #367).
- Task: Don't use `app.main` if the task name is set explicitly.
- Sending emails didn't work on Python 2.5, due to a bug in the version detection code (Issue #378).
- Beat: Adds method `ScheduleEntry._default_now`

  This method can be overridden to change the default value of `last_run_at`.
- An error occurring in process cleanup could mask task errors.

  We no longer propagate errors happening at process cleanup, but log them instead. This way they won't interfere with publishing the task result (Issue #365).
- Defining tasks didn't work properly when using the Django `shell_plus` utility (Issue #366).
- **AsyncResult.get didn't accept the `interval` and `propagate` arguments.**
- **worker: Fixed a bug where the worker wouldn't shutdown if a** `socket.error` **was raised.**

2.2.5

release-date 2011-03-28 06:00 p.m. CEST

release-by Ask Solem

Important Notes

- Now depends on Kombu 1.0.7

News

- Our documentation is now hosted by Read The Docs (http://docs.celeryproject.org), and all links have been changed to point to the new URL.
- Logging: Now supports log rotation using external tools like logrotate.d (Issue #321)

  This is accomplished by using the `WatchedFileHandler`, which re-opens the file if it's re-named or deleted.

- **`otherqueues` tutorial now documents how to configure Redis/Database result** backends.

- gevent: Now supports ETA tasks.

  But gevent still needs `CELERY_DISABLE_RATE_LIMITS=True` to work.

- TaskSet User Guide: now contains TaskSet callback recipes.

- Eventlet: New signals:

  - `eventlet_pool_started`

  - `eventlet_pool_preshutdown`

  - `eventlet_pool_postshutdown`

  - `eventlet_pool_apply`

  See `celery.signals` for more information.

- New `BROKER_TRANSPORT_OPTIONS` setting can be used to pass additional arguments to a particular broker transport.

- worker: `worker_pid` is now part of the request info as returned by broadcast commands.

- TaskSet.apply/Taskset.apply_async now accepts an optional `taskset_id` argument.

- The taskset_id (if any) is now available in the Task request context.

- SQLAlchemy result backend: taskset_id and taskset_id columns now have a unique constraint (tables need to recreated for this to take affect).

- Task user guide: Added section about choosing a result backend.

- Removed unused attribute `AsyncResult.uuid`.

Fixes

- multiprocessing.Pool: Fixes race condition when marking job with `WorkerLostError` (Issue #268).

  The process may have published a result before it was terminated, but we have no reliable way to detect that this is the case.

  So we have to wait for 10 seconds before marking the result with WorkerLostError. This gives the result handler a chance to retrieve the result.

- multiprocessing.Pool: Shutdown could hang if rate limits disabled.

  There was a race condition when the MainThread was waiting for the pool semaphore to be released. The ResultHandler now terminates after 5 seconds if there are unacked jobs, but no worker processes left to start them (it needs to timeout because there could still be an ack+result that we haven't consumed from the result queue. It is unlikely we'll receive any after 5 seconds with no worker processes).

- `celerybeat`: Now creates pidfile even if the `--detach` option isn't set.

- eventlet/gevent: The broadcast command consumer is now running in a separate green-thread.

  This ensures broadcast commands will take priority even if there are many active tasks.

- Internal module `celery.worker.controllers` renamed to `celery.worker.mediator`.

- worker: Threads now terminates the program by calling `os._exit`, as it is the only way to ensure exit in the case of syntax errors, or other unrecoverable errors.

- Fixed typo in `maybe_timedelta` (Issue #352).

- worker: Broadcast commands now logs with loglevel debug instead of warning.

- AMQP Result Backend: Now resets cached channel if the connection is lost.

- Polling results with the AMQP result backend wasn't working properly.

- Rate limits: No longer sleeps if there are no tasks, but rather waits for the task received condition (Performance improvement).

- ConfigurationView: `iter(dict)` should return keys, not items (Issue #362).

- `celerybeat`: PersistentScheduler now automatically removes a corrupted schedule file (Issue #346).

- Programs that doesn't support positional command-line arguments now provides a user friendly error message.

- Programs no longer tries to load the configuration file when showing `--version` (Issue #347).

- Autoscaler: The "all processes busy" log message is now severity debug instead of error.

- worker: If the message body can't be decoded, it's now passed through `safe_str` when logging.

    This to ensure we don't get additional decoding errors when trying to log the failure.

- `app.config_from_object`/`app.config_from_envvar` now works for all loaders.

- Now emits a user-friendly error message if the result backend name is unknown (Issue #349).

- `celery.contrib.batches`: Now sets loglevel and logfile in the task request so `task.get_logger` works with batch tasks (Issue #357).

- worker: An exception was raised if using the amqp transport and the prefetch count value exceeded 65535 (Issue #359).

    The prefetch count is incremented for every received task with an ETA/countdown defined. The prefetch count is a short, so can only support a maximum value of 65535. If the value exceeds the maximum value we now disable the prefetch count, it's re-enabled as soon as the value is below the limit again.

- `cursesmon`: Fixed unbound local error (Issue #303).

- eventlet/gevent is now imported on demand so autodoc can import the modules without having eventlet/gevent installed.

- worker: Ack callback now properly handles `AttributeError`.

- `Task.after_return` is now always called *after* the result has been written.

- Cassandra Result Backend: Should now work with the latest `pycassa` version.

- multiprocessing.Pool: No longer cares if the `putlock` semaphore is released too many times (this can happen if one or more worker processes are killed).

- SQLAlchemy Result Backend: Now returns accidentally removed `date_done` again (Issue #325).

- Task.request context is now always initialized to ensure calling the task function directly works even if it actively uses the request context.

- Exception occurring when iterating over the result from `TaskSet.apply` fixed.

- eventlet: Now properly schedules tasks with an ETA in the past.

2.2.4

release-date 2011-02-19 00:00 AM CET

release-by Ask Solem

Fixes

- worker: 2.2.3 broke error logging, resulting in tracebacks not being logged.

- AMQP result backend: Polling task states didn't work properly if there were more than one result message in the queue.

- `TaskSet.apply_async()` and `TaskSet.apply()` now supports an optional `taskset_id` keyword argument (Issue #331).

- The current taskset id (if any) is now available in the task context as `request.taskset` (Issue #329).

- SQLAlchemy result backend: *date_done* was no longer part of the results as it had been accidentally removed. It's now available again (Issue #325).

- SQLAlchemy result backend: Added unique constraint on *Task.id* and *TaskSet.taskset_id*. Tables needs to be recreated for this to take effect.

- Fixed exception raised when iterating on the result of `TaskSet.apply()`.

- Tasks user guide: Added section on choosing a result backend.

2.2.3

release-date 2011-02-12 04:00 p.m. CET

release-by Ask Solem

Fixes

- Now depends on Kombu 1.0.3

- Task.retry now supports a `max_retries` argument, used to change the default value.

- *multiprocessing.cpu_count* may raise `NotImplementedError` on platforms where this isn't supported (Issue #320).

- Coloring of log messages broke if the logged object wasn't a string.

- Fixed several typos in the init-script documentation.

- A regression caused *Task.exchange* and *Task.routing_key* to no longer have any effect. This is now fixed.

- Routing user guide: Fixes typo, routers in `CELERY_ROUTES` must be instances, not classes.

- **celeryev** didn't create pidfile even though the `--pidfile` argument was set.

- Task logger format was no longer used (Issue #317).

    The id and name of the task is now part of the log message again.

- A safe version of `repr()` is now used in strategic places to ensure objects with a broken `__repr__` doesn't crash the worker, or otherwise make errors hard to understand (Issue #298).

- Remote control command `active_queues`: didn't account for queues added at runtime.

    In addition the dictionary replied by this command now has a different structure: the exchange key is now a dictionary containing the exchange declaration in full.

- The `celery worker -Q` option removed unused queue declarations, so routing of tasks could fail.

> Queues are no longer removed, but rather *app.amqp.queues.consume_from()* is used as the list of queues to consume from.
>
> This ensures all queues are available for routing purposes.

- `celeryctl`: Now supports the *inspect active_queues* command.

2.2.2

release-date 2011-02-03 04:00 p.m. CET

release-by Ask Solem

Fixes

- `celerybeat` couldn't read the schedule properly, so entries in `CELERYBEAT_SCHEDULE` wouldn't be scheduled.

- Task error log message now includes *exc_info* again.

- The *eta* argument can now be used with *task.retry*.

    > Previously it was overwritten by the countdown argument.

- `celery multi`/`celeryd_detach`: Now logs errors occurring when executing the *celery worker* command.

- daemonizing tutorial: Fixed typo `--time-limit 300` -> `--time-limit=300`

- Colors in logging broke non-string objects in log messages.

- `setup_task_logger` no longer makes assumptions about magic task kwargs.

2.2.1

release-date 2011-02-02 04:00 p.m. CET

release-by Ask Solem

Fixes

- Eventlet pool was leaking memory (Issue #308).

- Deprecated function `celery.execute.delay_task` was accidentally removed, now available again.

- `BasePool.on_terminate` stub didn't exist

- `celeryd_detach`: Adds readable error messages if user/group name doesn't exist.

- Smarter handling of unicode decode errors when logging errors.

2.2.0

release-date 2011-02-01 10:00 AM CET

release-by Ask Solem

Important Notes

- Carrot has been replaced with Kombu

    Kombu is the next generation messaging library for Python, fixing several flaws present in Carrot that was hard to fix without breaking backwards compatibility.

    Also it adds:

    - First-class support for virtual transports; Redis, Django ORM, SQLAlchemy, Beanstalk, MongoDB, CouchDB and in-memory.

    - Consistent error handling with introspection,

    - The ability to ensure that an operation is performed by gracefully handling connection and channel errors,

    - Message compression (`zlib`, `bz2`, or custom compression schemes).

    This means that *ghettoq* is no longer needed as the functionality it provided is already available in Celery by default. The virtual transports are also more feature complete with support for exchanges (direct and topic). The Redis transport even supports fanout exchanges so it's able to perform worker remote control commands.

- Magic keyword arguments pending deprecation.

    The magic keyword arguments were responsible for many problems and quirks: notably issues with tasks and decorators, and name collisions in keyword arguments for the unaware.

    It wasn't easy to find a way to deprecate the magic keyword arguments, but we think this is a solution that makes sense and it won't have any adverse effects for existing code.

    The path to a magic keyword argument free world is:

    - the *celery.decorators* module is deprecated and the decorators can now be found in *celery.task*.

    - The decorators in *celery.task* disables keyword arguments by default

    - All examples in the documentation have been changed to use *celery.task*.

    This means that the following will have magic keyword arguments enabled (old style):

    ```python
    from celery.decorators import task

    @task()
    def add(x, y, **kwargs):
        print('In task %s' % kwargs['task_id'])
        return x + y
    ```

    And this won't use magic keyword arguments (new style):

    ```python
    from celery.task import task

    @task()
    def add(x, y):
        print('In task %s' % add.request.id)
        return x + y
    ```

    In addition, tasks can choose not to accept magic keyword arguments by setting the *task.accept_magic_kwargs* attribute.

Deprecation

Using the decorators in `celery.decorators` emits a `PendingDeprecationWarning` with a helpful message urging you to change your code, in version 2.4 this will be replaced with a `DeprecationWarning`, and in version 4.0 the `celery.decorators` module will be removed and no longer exist.

Similarly, the *task.accept_magic_kwargs* attribute will no longer have any effect starting from version 4.0.

• The magic keyword arguments are now available as *task.request*

This is called *the context*. Using thread-local storage the context contains state that's related to the current request.

It's mutable and you can add custom attributes that'll only be seen by the current task request.

The following context attributes are always available:

Magic Keyword Argument	Replace with
kwargs['task_id']	*self.request.id*
kwargs['delivery_info']	*self.request.delivery_info*
kwargs['task_retries']	*self.request.retries*
kwargs['logfile']	*self.request.logfile*
kwargs['loglevel']	*self.request.loglevel*
kwargs['task_is_eager']	*self.request.is_eager*
NEW	*self.request.args*
NEW	*self.request.kwargs*

In addition, the following methods now automatically uses the current context, so you don't have to pass *kwargs* manually anymore:

– *task.retry*

– *task.get_logger*

– *task.update_state*

• [Eventlet](#) support.

This is great news for I/O-bound tasks!

To change pool implementations you use the `celery worker --pool` argument, or globally using the `CELERYD_POOL` setting. This can be the full name of a class, or one of the following aliases: *processes*, *eventlet*, *gevent*.

For more information please see the *Concurrency with Eventlet* section in the User Guide.

Why not gevent?

For our first alternative concurrency implementation we've focused on [Eventlet](#), but there's also an experimental [gevent](#) pool available. This is missing some features, notably the ability to schedule ETA tasks.

Hopefully the [gevent](#) support will be feature complete by version 2.3, but this depends on user demand (and contributions).

• Python 2.4 support deprecated!

We're happy^H^H^H^H^Hsad to announce that this is the last version to support Python 2.4.

You're urged to make some noise if you're currently stuck with Python 2.4. Complain to your package maintainers, sysadmins and bosses: tell them it's time to move on!

Apart from wanting to take advantage of `with` statements, coroutines, conditional expressions and enhanced `try` blocks, the code base now contains so many 2.4 related hacks and workarounds it's no longer just a compromise, but a sacrifice.

If it really isn't your choice, and you don't have the option to upgrade to a newer version of Python, you can just continue to use Celery 2.2. Important fixes can be back ported for as long as there's interest.

- worker: Now supports Autoscaling of child worker processes.

    The `--autoscale` option can be used to configure the minimum and maximum number of child worker processes:

    ```
    --autoscale=AUTOSCALE
          Enable autoscaling by providing
          max_concurrency,min_concurrency. Example:
            --autoscale=10,3 (always keep 3 processes, but grow to
           10 if necessary).
    ```

- Remote Debugging of Tasks

    `celery.contrib.rdb` is an extended version of `pdb` that enables remote debugging of processes that doesn't have terminal access.

    Example usage:

    ```
        from celery.contrib import rdb
        from celery.task import task


        @task()
        def add(x, y):
            result = x + y
            # set breakpoint
            rdb.set_trace()
            return result

    :func:`~celery.contrib.rdb.set_trace` sets a breakpoint at the current
    location and creates a socket you can telnet into to remotely debug
    your task.

    The debugger may be started by multiple processes at the same time,
    so rather than using a fixed port the debugger will search for an
    available port, starting from the base port (6900 by default).
    The base port can be changed using the environment variable
    :envvar:`CELERY_RDB_PORT`.

    By default the debugger will only be available from the local host,
    to enable access from the outside you have to set the environment
    variable :envvar:`CELERY_RDB_HOST`.

    When the worker encounters your breakpoint it will log the following
    information::

        [INFO/MainProcess] Received task:
            tasks.add[d7261c71-4962-47e5-b342-2448bedd20e8]
    ```

(continues on next page)

```
    [WARNING/PoolWorker-1] Remote Debugger:6900:
        Please telnet 127.0.0.1 6900.  Type `exit` in session to
→continue.
    [2011-01-18 14:25:44,119: WARNING/PoolWorker-1] Remote Debugger:6900:
        Waiting for client...

If you telnet the port specified you'll be presented
with a ``pdb`` shell:

.. code-block:: console

    $ telnet localhost 6900
    Connected to localhost.
    Escape character is '^]'.
    > /opt/devel/demoapp/tasks.py(128)add()
    -> return result
    (Pdb)

Enter ``help`` to get a list of available commands,
It may be a good idea to read the `Python Debugger Manual`_ if
you have never used `pdb` before.
```

- Events are now transient and is using a topic exchange (instead of direct).

  The *CELERYD_EVENT_EXCHANGE*, *CELERYD_EVENT_ROUTING_KEY*, *CELERYD_EVENT_EXCHANGE_TYPE* settings are no longer in use.

  This means events won't be stored until there's a consumer, and the events will be gone as soon as the consumer stops. Also it means there can be multiple monitors running at the same time.

  The routing key of an event is the type of event (e.g., *worker.started*, *worker.heartbeat*, *task.succeeded*, etc. This means a consumer can filter on specific types, to only be alerted of the events it cares about.

  Each consumer will create a unique queue, meaning it's in effect a broadcast exchange.

  This opens up a lot of possibilities, for example the workers could listen for worker events to know what workers are in the neighborhood, and even restart workers when they go down (or use this information to optimize tasks/autoscaling).

  Note: The event exchange has been renamed from `"celeryevent"` to `"celeryev"` so it doesn't collide with older versions.

  If you'd like to remove the old exchange you can do so by executing the following command:

  ```
  $ camqadm exchange.delete celeryevent
  ```

- The worker now starts without configuration, and configuration can be specified directly on the command-line.

  Configuration options must appear after the last argument, separated by two dashes:

  ```
  $ celery worker -l info -I tasks -- broker.host=localhost broker.vhost=/app
  ```

- Configuration is now an alias to the original configuration, so changes to the original will reflect Celery at runtime.

- *celery.conf* has been deprecated, and modifying *celery.conf.ALWAYS_EAGER* will no longer have any effect.

> The default configuration is now available in the `celery.app.defaults` module. The available configuration options and their types can now be introspected.

- Remote control commands are now provided by *kombu.pidbox*, the generic process mailbox.

- Internal module *celery.worker.listener* has been renamed to *celery.worker.consumer*, and *.CarrotListener* is now *.Consumer*.

- Previously deprecated modules *celery.models* and *celery.management.commands* have now been removed as per the deprecation time-line.

- **[Security: Low severity] Removed *celery.task.RemoteExecuteTask* and** accompanying functions: *dmap*, *dmap_async*, and *execute_remote*.

  > Executing arbitrary code using pickle is a potential security issue if someone gains unrestricted access to the message broker.

  > If you really need this functionality, then you'd've to add this to your own project.

- [Security: Low severity] The *stats* command no longer transmits the broker password.

  > One would've needed an authenticated broker connection to receive this password in the first place, but sniffing the password at the wire level would've been possible if using unencrypted communication.

News

- The internal module *celery.task.builtins* has been removed.

- The module *celery.task.schedules* is deprecated, and *celery.schedules* should be used instead.

  > For example if you have:

  ```
  from celery.task.schedules import crontab
  ```

  > You should replace that with:

  ```
  from celery.schedules import crontab
  ```

  > The module needs to be renamed because it must be possible to import schedules without importing the *celery.task* module.

- The following functions have been deprecated and is scheduled for removal in version 2.3:

  - *celery.execute.apply_async*

    > Use *task.apply_async()* instead.

  - *celery.execute.apply*

    > Use *task.apply()* instead.

  - *celery.execute.delay_task*

    > Use *registry.tasks[name].delay()* instead.

- Importing *TaskSet* from *celery.task.base* is now deprecated.

  > You should use:

  ```
  >>> from celery.task import TaskSet
  ```

  > instead.

- New remote control commands:

    - *active_queues*

        Returns the queue declarations a worker is currently consuming from.

- Added the ability to retry publishing the task message in the event of connection loss or failure.

    This is disabled by default but can be enabled using the `CELERY_TASK_PUBLISH_RETRY` setting, and tweaked by the `CELERY_TASK_PUBLISH_RETRY_POLICY` setting.

    In addition *retry*, and *retry_policy* keyword arguments have been added to *Task.apply_async*.

    Note: Using the *retry* argument to *apply_async* requires you to handle the publisher/connection manually.

- Periodic Task classes (*@periodic_task*/*PeriodicTask*) will *not* be deprecated as previously indicated in the source code.

    But you're encouraged to use the more flexible `CELERYBEAT_SCHEDULE` setting.

- Built-in daemonization support of the worker using *celery multi* is no longer experimental and is considered production quality.

    See *Generic init-scripts* if you want to use the new generic init scripts.

- Added support for message compression using the `CELERY_MESSAGE_COMPRESSION` setting, or the *compression* argument to *apply_async*. This can also be set using routers.

- **worker: Now logs stack-trace of all threads when receiving the** *SIGUSR1* signal (doesn't work on CPython 2.4, Windows or Jython).

    Inspired by https://gist.github.com/737056

- Can now remotely terminate/kill the worker process currently processing a task.

    The *revoke* remote control command now supports a *terminate* argument Default signal is *TERM*, but can be specified using the *signal* argument. Signal can be the uppercase name of any signal defined in the `signal` module in the Python Standard Library.

    Terminating a task also revokes it.

    Example:

    ```
    >>> from celery.task.control import revoke

    >>> revoke(task_id, terminate=True)
    >>> revoke(task_id, terminate=True, signal='KILL')
    >>> revoke(task_id, terminate=True, signal='SIGKILL')
    ```

- *TaskSetResult.join_native*: Backend-optimized version of *join()*.

    If available, this version uses the backends ability to retrieve multiple results at once, unlike *join()* which fetches the results one by one.

    So far only supported by the AMQP result backend. Support for Memcached and Redis may be added later.

- Improved implementations of *TaskSetResult.join* and *AsyncResult.wait*.

    An *interval* keyword argument have been added to both so the polling interval can be specified (default interval is 0.5 seconds).

> A *propagate* keyword argument have been added to *result.wait()*, errors will be returned instead of raised if this is set to False.

> **Warning:** You should decrease the polling interval when using the database result backend, as frequent polling can result in high database load.

- The PID of the child worker process accepting a task is now sent as a field with the `task-started` event.

- The following fields have been added to all events in the worker class:

  - *sw_ident*: Name of worker software (e.g., `"py-celery"`).

  - *sw_ver*: Software version (e.g., 2.2.0).

  - *sw_sys*: Operating System (e.g., Linux, Windows, Darwin).

- For better accuracy the start time reported by the multiprocessing worker process is used when calculating task duration.

  > Previously the time reported by the accept callback was used.

- *celerybeat*: **New built-in daemonization support using the –detach** option.

- *celeryev*: **New built-in daemonization support using the –detach** option.

- *TaskSet.apply_async*: Now supports custom publishers by using the *publisher* argument.

- Added `CELERY_SEND_TASK_SENT_EVENT` setting.

  > If enabled an event will be sent with every task, so monitors can track tasks before the workers receive them.

- *celerybeat*: **Now reuses the broker connection when calling** scheduled tasks.

- The configuration module and loader to use can now be specified on the command-line.

  > For example:

  ```
  $ celery worker --config=celeryconfig.py --loader=myloader.Loader
  ```

- Added signals: *beat_init* and *beat_embedded_init*

  - `celery.signals.beat_init`

    > Dispatched when **celerybeat** starts (either standalone or embedded). Sender is the `celery.beat.Service` instance.

  - `celery.signals.beat_embedded_init`

    > Dispatched in addition to the `beat_init` signal when **celerybeat** is started as an embedded process. Sender is the `celery.beat.Service` instance.

- Redis result backend: Removed deprecated settings *REDIS_TIMEOUT* and *REDIS_CONNECT_RETRY*.

- CentOS init-script for **celery worker** now available in *extra/centos*.

- Now depends on pyparsing version 1.5.0 or higher.

  > There have been reported issues using Celery with pyparsing 1.4.x, so please upgrade to the latest version.

- Lots of new unit tests written, now with a total coverage of 95%.

Fixes

- *celeryev* Curses Monitor: Improved resize handling and UI layout (Issue #274 + Issue #276)

- AMQP Backend: Exceptions occurring while sending task results are now propagated instead of silenced.

  the worker will then show the full traceback of these errors in the log.

- AMQP Backend: No longer deletes the result queue after successful poll, as this should be handled by the `CELERY_AMQP_TASK_RESULT_EXPIRES` setting instead.

- AMQP Backend: Now ensures queues are declared before polling results.

- Windows: worker: Show error if running with *-B* option.

  Running `celerybeat` embedded is known not to work on Windows, so users are encouraged to run `celerybeat` as a separate service instead.

- Windows: Utilities no longer output ANSI color codes on Windows

- `camqadm`: Now properly handles `Control-c` by simply exiting instead of showing confusing traceback.

- Windows: All tests are now passing on Windows.

- Remove bin/ directory, and *scripts* section from `setup.py`.

  This means we now rely completely on setuptools entry-points.

Experimental

- Jython: worker now runs on Jython using the threaded pool.

  All tests pass, but there may still be bugs lurking around the corners.

- PyPy: worker now runs on PyPy.

  It runs without any pool, so to get parallel execution you must start multiple instances (e.g., using **multi**).

  Sadly an initial benchmark seems to show a 30% performance decrease on `pypy-1.4.1` + JIT. We would like to find out why this is, so stay tuned.

- `PublisherPool`: Experimental pool of task publishers and connections to be used with the *retry* argument to *apply_async*.

The example code below will re-use connections and channels, and retry sending of the task message if the connection is lost.

```python
from celery import current_app

# Global pool
pool = current_app().amqp.PublisherPool(limit=10)

def my_view(request):
    with pool.acquire() as publisher:
        add.apply_async((2, 2), publisher=publisher, retry=True)
```

3.13.16 Change history for Celery 2.1

2.1.4

release-date 2010-12-03 12:00 p.m. CEST

release-by Ask Solem

Fixes

- Execution options to *apply_async* now takes precedence over options returned by active routers. This was a regression introduced recently (Issue #244).

- curses monitor: Long arguments are now truncated so curses doesn't crash with out of bounds errors (Issue #235).

- multi: Channel errors occurring while handling control commands no longer crash the worker but are instead logged with severity error.

- SQLAlchemy database backend: Fixed a race condition occurring when the client wrote the pending state. Just like the Django database backend, it does no longer save the pending state (Issue #261 + Issue #262).

- Error email body now uses *repr(exception)* instead of *str(exception)*, as the latter could result in Unicode decode errors (Issue #245).

- Error email timeout value is now configurable by using the `EMAIL_TIMEOUT` setting.

- *celeryev*: Now works on Windows (but the curses monitor won't work without having curses).

- Unit test output no longer emits non-standard characters.

- worker: The broadcast consumer is now closed if the connection is reset.

- worker: Now properly handles errors occurring while trying to acknowledge the message.

- *TaskRequest.on_failure* **now encodes traceback using the current file-system** encoding (Issue #286).

- *EagerResult* can now be pickled (Issue #288).

Documentation

- Adding *Contributing*.

- Added *Optimizing*.

- Added *Security* section to the FAQ.

2.1.3

release-date 2010-11-09 05:00 p.m. CEST

release-by Ask Solem

- Fixed deadlocks in *timer2* which could lead to *djcelerymon/celeryev -c* hanging.

- *EventReceiver*: now sends heartbeat request to find workers.

  This means **celeryev** and friends finds workers immediately at start-up.

- `celeryev` curses monitor: Set screen_delay to 10ms, so the screen refreshes more often.

- Fixed pickling errors when pickling `AsyncResult` on older Python versions.

- worker: prefetch count was decremented by ETA tasks even if there were no active prefetch limits.

2.1.2

release-data TBA

Fixes

- worker: Now sends the `task-retried` event for retried tasks.

- worker: Now honors ignore result for `WorkerLostError` and timeout errors.

- celerybeat: Fixed `UnboundLocalError` in `celerybeat` logging when using logging setup signals.

- worker: All log messages now includes *exc_info*.

2.1.1

release-date 2010-10-14 02:00 p.m. CEST

release-by Ask Solem

Fixes

- Now working on Windows again.

    Removed dependency on the `pwd`/`grp` modules.

- snapshots: Fixed race condition leading to loss of events.

- worker: Reject tasks with an ETA that cannot be converted to a time stamp.

    See issue #209

- concurrency.processes.pool: The semaphore was released twice for each task (both at ACK and result ready).

    This has been fixed, and it is now released only once per task.

- docs/configuration: Fixed typo *CELERYD_TASK_SOFT_TIME_LIMIT* -> `CELERYD_TASK_SOFT_TIME_LIMIT`.

    See issue #214

- control command *dump_scheduled*: was using old .info attribute

- **multi: Fixed** *set changed size during iteration* **bug** occurring in the restart command.

- worker: Accidentally tried to use additional command-line arguments.

    This would lead to an error like:

    got multiple values for keyword argument 'concurrency'.

    Additional command-line arguments are now ignored, and doesn't produce this error. However – we do reserve the right to use positional arguments in the future, so please don't depend on this behavior.

- `celerybeat`: Now respects routers and task execution options again.

- `celerybeat`: Now reuses the publisher instead of the connection.

- Cache result backend: Using `float` as the expires argument to *cache.set* is deprecated by the Memcached libraries, so we now automatically cast to `int`.

- unit tests: No longer emits logging and warnings in test output.

News

- Now depends on carrot version 0.10.7.

- Added `CELERY_REDIRECT_STDOUTS`, and `CELERYD_REDIRECT_STDOUTS_LEVEL` settings.

    `CELERY_REDIRECT_STDOUTS` is used by the worker and beat. All output to *stdout* and *stderr* will be redirected to the current logger if enabled.

    `CELERY_REDIRECT_STDOUTS_LEVEL` decides the log level used and is `WARNING` by default.

- Added `CELERYBEAT_SCHEDULER` setting.

    This setting is used to define the default for the -S option to **celerybeat**.

    Example:

    ```
    CELERYBEAT_SCHEDULER = 'djcelery.schedulers.DatabaseScheduler'
    ```

- Added Task.expires: Used to set default expiry time for tasks.

- New remote control commands: *add_consumer* and *cancel_consumer*.

    add_consumer(queue, exchange, exchange_type, routing_key, **options)
    : Tells the worker to declare and consume from the specified declaration.

    cancel_consumer(*queue_name*)
    : Tells the worker to stop consuming from queue (by queue name).

    Commands also added to **celeryctl** and inspect.

    Example using celeryctl to start consuming from queue "queue", in exchange "exchange", of type "direct" using binding key "key":

    ```
    $ celeryctl inspect add_consumer queue exchange direct key
    $ celeryctl inspect cancel_consumer queue
    ```

    See *Management Command-line Utilities (inspect/control)* for more information about the **celeryctl** program.

    Another example using inspect:

    ```
    >>> from celery.task.control import inspect
    >>> inspect.add_consumer(queue='queue', exchange='exchange',
    ...                      exchange_type='direct',
    ...                      routing_key='key',
    ...                      durable=False,
    ...                      auto_delete=True)

    >>> inspect.cancel_consumer('queue')
    ```

- celerybeat: Now logs the traceback if a message can't be sent.

- celerybeat: Now enables a default socket timeout of 30 seconds.

- README/introduction/homepage: Added link to Flask-Celery.

2.1.0

release-date 2010-10-08 12:00 p.m. CEST

release-by Ask Solem

Important Notes

- Celery is now following the versioning semantics defined by semver.

    This means we're no longer allowed to use odd/even versioning semantics By our previous versioning scheme this stable release should've been version 2.2.

- Now depends on Carrot 0.10.7.

- No longer depends on SQLAlchemy, this needs to be installed separately if the database result backend is used.

- django-celery now comes with a monitor for the Django Admin interface. This can also be used if you're not a Django user. (Update: Django-Admin monitor has been replaced with Flower, see the Monitoring guide).

- If you get an error after upgrading saying: *AttributeError: 'module' object has no attribute 'system'*,

Then this is because the *celery.platform* module has been renamed to *celery.platforms* to not collide with the built-in `platform` module.

You have to remove the old `platform.py` (and maybe `platform.pyc`) file from your previous Celery installation.

To do this use **python** to find the location of this module:

```
$ python
>>> import celery.platform
>>> celery.platform
<module 'celery.platform' from '/opt/devel/celery/celery/platform.pyc'>
```

Here the compiled module is in `/opt/devel/celery/celery/`, to remove the offending files do:

```
$ rm -f /opt/devel/celery/celery/platform.py*
```

News

- Added support for expiration of AMQP results (requires RabbitMQ 2.1.0)

  The new configuration option `CELERY_AMQP_TASK_RESULT_EXPIRES` sets the expiry time in seconds (can be int or float):

  ```
  CELERY_AMQP_TASK_RESULT_EXPIRES = 30 * 60  # 30 minutes.
  CELERY_AMQP_TASK_RESULT_EXPIRES = 0.80     # 800 ms.
  ```

- `celeryev`: Event Snapshots

  If enabled, the worker sends messages about what the worker is doing. These messages are called "events". The events are used by real-time monitors to show what the cluster is doing, but they're not very useful for monitoring over a longer period of time. Snapshots lets you take "pictures" of the clusters state at regular intervals. This can then be stored in a database to generate statistics with, or even monitoring over longer time periods.

  django-celery now comes with a Celery monitor for the Django Admin interface. To use this you need to run the django-celery snapshot camera, which stores snapshots to the database at configurable intervals.

  To use the Django admin monitor you need to do the following:

  1. Create the new database tables:

     ```
     $ python manage.py syncdb
     ```

  2. Start the django-celery snapshot camera:

     ```
     $ python manage.py celerycam
     ```

  3. Open up the django admin to monitor your cluster.

  The admin interface shows tasks, worker nodes, and even lets you perform some actions, like revoking and rate limiting tasks, and shutting down worker nodes.

  There's also a Debian init.d script for `events` available, see *Daemonization* for more information.

  New command-line arguments to `celeryev`:

  - `celery events --camera`: Snapshot camera class to use.

- *celery events --logfile*: Log file
- *celery events --loglevel*: Log level
- *celery events --maxrate*: Shutter rate limit.
- *celery events --freq*: Shutter frequency

The *--camera* argument is the name of a class used to take snapshots with. It must support the interface defined by *celery.events.snapshot.Polaroid*.

Shutter frequency controls how often the camera thread wakes up, while the rate limit controls how often it will actually take a snapshot. The rate limit can be an integer (snapshots/s), or a rate limit string which has the same syntax as the task rate limit strings (*"200/m"*, *"10/s"*, *"1/h"*, etc).

For the Django camera case, this rate limit can be used to control how often the snapshots are written to the database, and the frequency used to control how often the thread wakes up to check if there's anything new.

The rate limit is off by default, which means it will take a snapshot for every *--frequency* seconds.

- `broadcast()`: Added callback argument, this can be used to process replies immediately as they arrive.
- `celeryctl`: New command line utility to manage and inspect worker nodes, apply tasks and inspect the results of tasks.

  See also:

  The *Management Command-line Utilities (inspect/control)* section in the *User Guide*.

  Some examples:

  ```
  $ celeryctl apply tasks.add -a '[2, 2]' --countdown=10

  $ celeryctl inspect active
  $ celeryctl inspect registered_tasks
  $ celeryctl inspect scheduled
  $ celeryctl inspect --help
  $ celeryctl apply --help
  ```

- Added the ability to set an expiry date and time for tasks.

  Example:

  ```
  >>> # Task expires after one minute from now.
  >>> task.apply_async(args, kwargs, expires=60)
  >>> # Also supports datetime
  >>> task.apply_async(args, kwargs,
  ...                   expires=datetime.now() + timedelta(days=1)
  ```

  When a worker receives a task that's been expired it will be marked as revoked (*TaskRevokedError*).

- Changed the way logging is configured.

  We now configure the root logger instead of only configuring our custom logger. In addition we don't hijack the multiprocessing logger anymore, but instead use a custom logger name for different applications:

Application	Logger Name
celeryd	"celery"
celerybeat	"celery.beat"
celeryev	"celery.ev"

This means that the *loglevel* and *logfile* arguments will affect all registered loggers (even those from third-party libraries). Unless you configure the loggers manually as shown below, that is.

Users can choose to configure logging by subscribing to the :signal:'~celery.signals.setup_logging'
signal:

```python
from logging.config import fileConfig
from celery import signals

@signals.setup_logging.connect
def setup_logging(**kwargs):
    fileConfig('logging.conf')
```

If there are no receivers for this signal, the logging subsystem will be configured using the `--loglevel`/ `--logfile` arguments, this will be used for *all defined loggers*.

Remember that the worker also redirects stdout and stderr to the Celery logger, if manually configure logging you also need to redirect the standard outs manually:

```python
from logging.config import fileConfig
from celery import log

def setup_logging(**kwargs):
    import logging
    fileConfig('logging.conf')
    stdouts = logging.getLogger('mystdoutslogger')
    log.redirect_stdouts_to_logger(stdouts, loglevel=logging.WARNING)
```

- worker Added command line option `--include`:

  A comma separated list of (task) modules to be imported.

  Example:

  ```
  $ celeryd -I app1.tasks,app2.tasks
  ```

- worker: now emits a warning if running as the root user (euid is 0).

- `celery.messaging.establish_connection()`: Ability to override defaults used using keyword argument "defaults".

- worker: Now uses *multiprocessing.freeze_support()* so that it should work with **py2exe**, **PyInstaller**, **cx_Freeze**, etc.

- worker: Now includes more meta-data for the `STARTED` state: PID and host name of the worker that started the task.

  See issue #181

- subtask: Merge additional keyword arguments to *subtask()* into task keyword arguments.

  For example:

```
>>> s = subtask((1, 2), {'foo': 'bar'}, baz=1)
>>> s.args
(1, 2)
>>> s.kwargs
{'foo': 'bar', 'baz': 1}
```

See issue #182.

- worker: Now emits a warning if there's already a worker node using the same name running on the same virtual host.

- AMQP result backend: Sending of results are now retried if the connection is down.

- **AMQP result backend:** *result.get()*: **Wait for next state if state isn't** in READY_STATES.

- TaskSetResult now supports subscription.

```
>>> res = TaskSet(tasks).apply_async()
>>> res[0].get()
```

- Added *Task.send_error_emails* + *Task.error_whitelist*, so these can be configured per task instead of just by the global setting.

- Added *Task.store_errors_even_if_ignored*, so it can be changed per Task, not just by the global setting.

- The Crontab scheduler no longer wakes up every second, but implements *remaining_estimate* (*Optimization*).

- **worker: Store** ***FAILURE*** **result if the** *WorkerLostError* exception occurs (worker process disappeared).

- worker: Store *FAILURE* result if one of the **TimeLimitExceeded* exceptions occurs.

- Refactored the periodic task responsible for cleaning up results.

    - **The backend cleanup task is now only added to the schedule if**
      CELERY_TASK_RESULT_EXPIRES is set.

    - If the schedule already contains a periodic task named "celery.backend_cleanup" it won't change it, so the behavior of the backend cleanup task can be easily changed.

    - The task is now run every day at 4:00 AM, rather than every day since the first time it was run (using Crontab schedule instead of *run_every*)

    - **Renamed** ***celery.task.builtins.DeleteExpiredTaskMetaTask*** -> celery.task.builtins.backend_cleanup

    - The task itself has been renamed from "celery.delete_expired_task_meta" to "celery.backend_cleanup"

    See issue #134.

- Implemented *AsyncResult.forget* for SQLAlchemy/Memcached/Redis/Tokyo Tyrant backends (forget and remove task result).

    See issue #184.

- TaskSetResult.join: Added 'propagate=True' argument.

    When set to False exceptions occurring in subtasks will not be re-raised.

- Added *Task.update_state(task_id, state, meta)* as a shortcut to *task.backend.store_result(task_id, meta, state)*.

    The backend interface is "private" and the terminology outdated, so better to move this to Task so it can be used.

- timer2: Set *self.running=False* in *stop()* so it won't try to join again on subsequent calls to *stop()*.

- Log colors are now disabled by default on Windows.

- *celery.platform* renamed to `celery.platforms`, so it doesn't collide with the built-in `platform` module.

- Exceptions occurring in Mediator+Pool callbacks are now caught and logged instead of taking down the worker.

- Redis result backend: Now supports result expiration using the Redis *EXPIRE* command.

- unit tests: Don't leave threads running at tear down.

- worker: Task results shown in logs are now truncated to 46 chars.

- **Task.__name__ is now an alias to self.__class__.__name__.** This way tasks introspects more like regular functions.

- *Task.retry*: Now raises `TypeError` if kwargs argument is empty.

    See issue #164.

- `timedelta_seconds`: Use `timedelta.total_seconds` if running on Python 2.7

- `TokenBucket`: Generic Token Bucket algorithm

- `celery.events.state`: Recording of cluster state can now be paused and resumed, including support for buffering.

    State.**freeze**(*buffer=True*)
        Pauses recording of the stream.

        If *buffer* is true, events received while being frozen will be buffered, and may be replayed later.

    State.**thaw**(*replay=True*)
        Resumes recording of the stream.

        If *replay* is true, then the recorded buffer will be applied.

    State.**freeze_while**(*fun*)
        With a function to apply, freezes the stream before, and replays the buffer after the function returns.

- `EventReceiver.capture` Now supports a timeout keyword argument.

- worker: The mediator thread is now disabled if CELERY_RATE_LIMITS is enabled, and tasks are directly sent to the pool without going through the ready queue (*Optimization*).

Fixes

- Pool: Process timed out by *TimeoutHandler* must be joined by the Supervisor, so don't remove it from the internal process list.

    See issue #192.

- *TaskPublisher.delay_task* now supports exchange argument, so exchange can be overridden when sending tasks in bulk using the same publisher

    See issue #187.

- the worker no longer marks tasks as revoked if CELERY_IGNORE_RESULT is enabled.

    See issue #207.

- AMQP Result backend: Fixed bug with *result.get()* if CELERY_TRACK_STARTED enabled.

    result.get() would stop consuming after receiving the *STARTED* state.

- Fixed bug where new processes created by the pool supervisor becomes stuck while reading from the task Queue.

    See http://bugs.python.org/issue10037

- Fixed timing issue when declaring the remote control command reply queue

    This issue could result in replies being lost, but have now been fixed.

- Backward compatible *LoggerAdapter* implementation: Now works for Python 2.4.

    Also added support for several new methods: *fatal*, *makeRecord*, *_log*, *log*, *isEnabledFor*, *addHandler*, *removeHandler*.

Experimental

- multi: Added daemonization support.

    multi can now be used to start, stop and restart worker nodes:

    ```
    $ celeryd-multi start jerry elaine george kramer
    ```

    This also creates PID files and log files (`celeryd@jerry.pid`, ..., `celeryd@jerry.log`. To specify a location for these files use the *–pidfile* and *–logfile* arguments with the *%n* format:

    ```
    $ celeryd-multi start jerry elaine george kramer \
                    --logfile=/var/log/celeryd@%n.log \
                    --pidfile=/var/run/celeryd@%n.pid
    ```

    Stopping:

    ```
    $ celeryd-multi stop jerry elaine george kramer
    ```

    Restarting. The nodes will be restarted one by one as the old ones are shutdown:

    ```
    $ celeryd-multi restart jerry elaine george kramer
    ```

    Killing the nodes (**WARNING**: Will discard currently executing tasks):

    ```
    $ celeryd-multi kill jerry elaine george kramer
    ```

    See *celeryd-multi help* for help.

- multi: *start* command renamed to *show*.

    celeryd-multi start will now actually start and detach worker nodes. To just generate the commands you have to use *celeryd-multi show*.

- worker: Added *–pidfile* argument.

    The worker will write its pid when it starts. The worker will not be started if this file exists and the pid contained is still alive.

- Added generic init.d script using *celeryd-multi*

    https://github.com/celery/celery/tree/master/extra/generic-init.d/celeryd

Documentation

- Added User guide section: Monitoring
- Added user guide section: Periodic Tasks

    Moved from *getting-started/periodic-tasks* and updated.

- tutorials/external moved to new section: "community".
- References has been added to all sections in the documentation.

    This makes it easier to link between documents.

3.13.17 Change history for Celery 2.0

- *2.0.3*
    - *Fixes*
    - *Documentation*
- *2.0.2*
- *2.0.1*
- *2.0.0*
    - *Foreword*
    - *Upgrading for Django-users*
    - *Upgrading for others*
        * *Database result backend*
        * *Cache result backend*
    - *Backward incompatible changes*
    - *News*

2.0.3

release-date 2010-08-27 12:00 p.m. CEST

release-by Ask Solem

Fixes

- Worker: Properly handle connection errors happening while closing consumers.
- Worker: Events are now buffered if the connection is down, then sent when the connection is re-established.
- No longer depends on the mailer package.

    This package had a name space collision with *django-mailer*, so its functionality was replaced.

- Redis result backend: Documentation typos: Redis doesn't have database names, but database numbers. The default database is now 0.

- `inspect`: *registered_tasks* was requesting an invalid command because of a typo.

    See issue #170.

- `CELERY_ROUTES`: Values defined in the route should now have precedence over values defined in `CELERY_QUEUES` when merging the two.

    With the follow settings:

    ```
    CELERY_QUEUES = {'cpubound': {'exchange': 'cpubound',
                                  'routing_key': 'cpubound'}}

    CELERY_ROUTES = {'tasks.add': {'queue': 'cpubound',
                                   'routing_key': 'tasks.add',
                                   'serializer': 'json'}}
    ```

    The final routing options for *tasks.add* will become:

    ```
    {'exchange': 'cpubound',
     'routing_key': 'tasks.add',
     'serializer': 'json'}
    ```

    This wasn't the case before: the values in `CELERY_QUEUES` would take precedence.

- Worker crashed if the value of `CELERY_TASK_ERROR_WHITELIST` was not an iterable

- `apply()`: Make sure *kwargs['task_id']* is always set.

- *AsyncResult.traceback*: Now returns `None`, instead of raising `KeyError` if traceback is missing.

- `inspect`: Replies didn't work correctly if no destination was specified.

- Can now store result/meta-data for custom states.

- Worker: A warning is now emitted if the sending of task error emails fails.

- `celeryev`: Curses monitor no longer crashes if the terminal window is resized.

    See issue #160.

- Worker: On macOS it isn't possible to run *os.exec** in a process that's threaded.

    This breaks the SIGHUP restart handler, and is now disabled on macOS, emitting a warning instead.

    See issue #152.

- `celery.execute.trace`: Properly handle *raise(str)*, which is still allowed in Python 2.4.

    See issue #175.

- Using urllib2 in a periodic task on macOS crashed because of the proxy auto detection used in macOS.

    This is now fixed by using a workaround. See issue #143.

- Debian init-scripts: Commands shouldn't run in a sub shell

    See issue #163.

- Debian init-scripts: Use the absolute path of `celeryd` program to allow stat

    See issue #162.

Documentation

- getting-started/broker-installation: Fixed typo

  set_permissions "" -> set_permissions "."*.

- Tasks User Guide: Added section on database transactions.

  See issue #169.

- Routing User Guide: Fixed typo *"feed": -> {"queue": "feeds"}*.

  See issue #169.

- Documented the default values for the `CELERYD_CONCURRENCY` and `CELERYD_PREFETCH_MULTIPLIER` settings.

- Tasks User Guide: Fixed typos in the subtask example

- celery.signals: Documented worker_process_init.

- Daemonization cookbook: Need to export DJANGO_SETTINGS_MODULE in */etc/default/celeryd*.

- Added some more FAQs from stack overflow

- Daemonization cookbook: Fixed typo *CELERYD_LOGFILE/CELERYD_PIDFILE*

  to *CELERYD_LOG_FILE / CELERYD_PID_FILE*

  Also added troubleshooting section for the init-scripts.

2.0.2

release-date 2010-07-22 11:31 a.m. CEST

release-by Ask Solem

- Routes: When using the dict route syntax, the exchange for a task could disappear making the task unroutable.

  See issue #158.

- Test suite now passing on Python 2.4

- No longer have to type *PYTHONPATH=.* to use `celeryconfig` in the current directory.

  This is accomplished by the default loader ensuring that the current directory is in *sys.path* when loading the config module. *sys.path* is reset to its original state after loading.

  Adding the current working directory to *sys.path* without the user knowing may be a security issue, as this means someone can drop a Python module in the users directory that executes arbitrary commands. This was the original reason not to do this, but if done *only when loading the config module*, this means that the behavior will only apply to the modules imported in the config module, which I think is a good compromise (certainly better than just explicitly setting *PYTHONPATH=.* anyway)

- Experimental Cassandra backend added.

- Worker: SIGHUP handler accidentally propagated to worker pool processes.

  In combination with GitHub SHA@7a7c44e39344789f11b5346e9cc8340f5fe4846c this would make each child process start a new worker instance when the terminal window was closed :/

- Worker: Don't install SIGHUP handler if running from a terminal.

  This fixes the problem where the worker is launched in the background when closing the terminal.

- Worker: Now joins threads at shutdown.

  See issue #152.

- Test tear down: Don't use *atexit* but nose's *teardown()* functionality instead.

  See issue #154.

- Debian worker init-script: Stop now works correctly.

- Task logger: *warn* method added (synonym for *warning*)

- Can now define a white list of errors to send error emails for.

  Example:

  ```
  CELERY_TASK_ERROR_WHITELIST = ('myapp.MalformedInputError',)
  ```

  See issue #153.

- Worker: Now handles overflow exceptions in *time.mktime* while parsing the ETA field.

- LoggerWrapper: Try to detect loggers logging back to stderr/stdout making an infinite loop.

- Added `celery.task.control.inspect`: Inspects a running worker.

  Examples:

  ```
  # Inspect a single worker
  >>> i = inspect('myworker.example.com')

  # Inspect several workers
  >>> i = inspect(['myworker.example.com', 'myworker2.example.com'])

  # Inspect all workers consuming on this vhost.
  >>> i = inspect()

  ### Methods

  # Get currently executing tasks
  >>> i.active()

  # Get currently reserved tasks
  >>> i.reserved()

  # Get the current ETA schedule
  >>> i.scheduled()

  # Worker statistics and info
  >>> i.stats()

  # List of currently revoked tasks
  >>> i.revoked()

  # List of registered tasks
  >>> i.registered_tasks()
  ```

- Remote control commands *dump_active/dump_reserved/dump_schedule* now replies with detailed task requests.

  Containing the original arguments and fields of the task requested.

  In addition the remote control command *set_loglevel* has been added, this only changes the log level for the main process.

- Worker control command execution now catches errors and returns their string representation in the reply.

- Functional test suite added

  `celery.tests.functional.case` contains utilities to start and stop an embedded worker process, for use in functional testing.

2.0.1

release-date 2010-07-09 03:02 p.m. CEST

release-by Ask Solem

- multiprocessing.pool: Now handles encoding errors, so that pickling errors doesn't crash the worker processes.

- The remote control command replies wasn't working with RabbitMQ 1.8.0's stricter equivalence checks.

  If you've already hit this problem you may have to delete the declaration:

  ```
  $ camqadm exchange.delete celerycrq
  ```

  or:

  ```
  $ python manage.py camqadm exchange.delete celerycrq
  ```

- A bug sneaked in the ETA scheduler that made it only able to execute one task per second(!)

  The scheduler sleeps between iterations so it doesn't consume too much CPU. It keeps a list of the scheduled items sorted by time, at each iteration it sleeps for the remaining time of the item with the nearest deadline. If there are no ETA tasks it will sleep for a minimum amount of time, one second by default.

  A bug sneaked in here, making it sleep for one second for every task that was scheduled. This has been fixed, so now it should move tasks like hot knife through butter.

  In addition a new setting has been added to control the minimum sleep interval; `CELERYD_ETA_SCHEDULER_PRECISION`. A good value for this would be a float between 0 and 1, depending on the needed precision. A value of 0.8 means that when the ETA of a task is met, it will take at most 0.8 seconds for the task to be moved to the ready queue.

- Pool: Supervisor didn't release the semaphore.

  This would lead to a deadlock if all workers terminated prematurely.

- Added Python version trove classifiers: 2.4, 2.5, 2.6 and 2.7

- Tests now passing on Python 2.7.

- Task.__reduce__: Tasks created using the task decorator can now be pickled.

- `setup.py`: nose added to *tests_require*.

- Pickle should now work with SQLAlchemy 0.5.x

- New homepage design by Jan Henrik Helmers: http://celeryproject.org

- New Sphinx theme by Armin Ronacher: http://docs.celeryproject.org/

- Fixed "pending_xref" errors shown in the HTML rendering of the documentation. Apparently this was caused by new changes in Sphinx 1.0b2.

- Router classes in `CELERY_ROUTES` are now imported lazily.

> Importing a router class in a module that also loads the Celery environment would cause a circular dependency. This is solved by importing it when needed after the environment is set up.

- `CELERY_ROUTES` was broken if set to a single dict.

  This example in the docs should now work again:

  ```
  CELERY_ROUTES = {'feed.tasks.import_feed': 'feeds'}
  ```

- *CREATE_MISSING_QUEUES* wasn't honored by apply_async.

- New remote control command: *stats*

  Dumps information about the worker, like pool process ids, and total number of tasks executed by type.

  Example reply:

  ```
  [{'worker.local':
      'total': {'tasks.sleeptask': 6},
      'pool': {'timeouts': [None, None],
               'processes': [60376, 60377],
               'max-concurrency': 2,
               'max-tasks-per-child': None,
               'put-guarded-by-semaphore': True}}]
  ```

- New remote control command: *dump_active*

  Gives a list of tasks currently being executed by the worker. By default arguments are passed through repr in case there are arguments that's not JSON encodable. If you know the arguments are JSON safe, you can pass the argument *safe=True*.

  Example reply:

  ```
  >>> broadcast('dump_active', arguments={'safe': False}, reply=True)
  [{'worker.local': [
      {'args': '(1,)',
       'time_start': 1278580542.6300001,
       'name': 'tasks.sleeptask',
       'delivery_info': {
           'consumer_tag': '30',
           'routing_key': 'celery',
           'exchange': 'celery'},
       'hostname': 'casper.local',
       'acknowledged': True,
       'kwargs': '{}',
       'id': '802e93e9-e470-47ed-b913-06de8510aca2',
      }
  ]}]
  ```

- Added experimental support for persistent revokes.

  Use the *-S*|*–statedb* argument to the worker to enable it:

  ```
  $ celeryd --statedb=/var/run/celeryd
  ```

  This will use the file: */var/run/celeryd.db*, as the *shelve* module automatically adds the *.db* suffix.

2.0.0

release-date 2010-07-02 02:30 p.m. CEST

release-by Ask Solem

Foreword

Celery 2.0 contains backward incompatible changes, the most important being that the Django dependency has been removed so Celery no longer supports Django out of the box, but instead as an add-on package called django-celery.

We're very sorry for breaking backwards compatibility, but there's also many new and exciting features to make up for the time you lose upgrading, so be sure to read the *News* section.

Quite a lot of potential users have been upset about the Django dependency, so maybe this is a chance to get wider adoption by the Python community as well.

Big thanks to all contributors, testers and users!

Upgrading for Django-users

Django integration has been moved to a separate package: django-celery.

- To upgrade you need to install the django-celery module and change:

```
INSTALLED_APPS = 'celery'
```

to:

```
INSTALLED_APPS = 'djcelery'
```

- If you use *mod_wsgi* you need to add the following line to your *.wsgi* file:

```python
import os
os.environ['CELERY_LOADER'] = 'django'
```

- The following modules has been moved to django-celery:

Module name	Replace with
celery.models	*djcelery.models*
celery.managers	*djcelery.managers*
celery.views	*djcelery.views*
celery.urls	*djcelery.urls*
celery.management	*djcelery.management*
celery.loaders.djangoapp	*djcelery.loaders*
celery.backends.database	*djcelery.backends.database*
celery.backends.cache	*djcelery.backends.cache*

Importing `djcelery` will automatically setup Celery to use Django loader. loader. It does this by setting the `CELERY_LOADER` environment variable to *"django"* (it won't change it if a loader is already set).

When the Django loader is used, the "database" and "cache" result backend aliases will point to the `djcelery` backends instead of the built-in backends, and configuration will be read from the Django settings.

Upgrading for others

Database result backend

The database result backend is now using SQLAlchemy instead of the Django ORM, see Supported Databases for a table of supported databases.

The *DATABASE_** settings has been replaced by a single setting: `CELERY_RESULT_DBURI`. The value here should be an SQLAlchemy Connection String, some examples include:

```
# sqlite (filename)
CELERY_RESULT_DBURI = 'sqlite:///celerydb.sqlite'

# mysql
CELERY_RESULT_DBURI = 'mysql://scott:tiger@localhost/foo'

# postgresql
CELERY_RESULT_DBURI = 'postgresql://scott:tiger@localhost/mydatabase'

# oracle
CELERY_RESULT_DBURI = 'oracle://scott:tiger@127.0.0.1:1521/sidname'
```

See SQLAlchemy Connection Strings for more information about connection strings.

To specify additional SQLAlchemy database engine options you can use the `CELERY_RESULT_ENGINE_OPTIONS` setting:

```
# echo enables verbose logging from SQLAlchemy.
CELERY_RESULT_ENGINE_OPTIONS = {'echo': True}
```

Cache result backend

The cache result backend is no longer using the Django cache framework, but it supports mostly the same configuration syntax:

```
CELERY_CACHE_BACKEND = 'memcached://A.example.com:11211;B.example.com'
```

To use the cache backend you must either have the pylibmc or python-memcached library installed, of which the former is regarded as the best choice.

The support backend types are *memcached://* and *memory://*, we haven't felt the need to support any of the other backends provided by Django.

Backward incompatible changes

- Default (python) loader now prints warning on missing *celeryconfig.py* instead of raising `ImportError`.

    The worker raises *ImproperlyConfigured* if the configuration isn't set up. This makes it possible to use *–help* etc., without having a working configuration.

    Also this makes it possible to use the client side of Celery without being configured:

    ```
    >>> from carrot.connection import BrokerConnection
    >>> conn = BrokerConnection('localhost', 'guest', 'guest', '/')
    >>> from celery.execute import send_task
    ```

```
>>> r = send_task('celery.ping', args=(), kwargs={}, connection=conn)
>>> from celery.backends.amqp import AMQPBackend
>>> r.backend = AMQPBackend(connection=conn)
>>> r.get()
'pong'
```

- The following deprecated settings has been removed (as scheduled by the *Celery Deprecation Time-line*):

Setting name	Replace with
CELERY_AMQP_CONSUMER_QUEUES	*CELERY_QUEUES*
CELERY_AMQP_EXCHANGE	*CELERY_DEFAULT_EXCHANGE*
CELERY_AMQP_EXCHANGE_TYPE	*CELERY_DEFAULT_EXCHANGE_TYPE*
CELERY_AMQP_CONSUMER_ROUTING_KEY	*CELERY_QUEUES*
CELERY_AMQP_PUBLISHER_ROUTING_KEY	*CELERY_DEFAULT_ROUTING_KEY*

- The *celery.task.rest* module has been removed, use *celery.task.http* instead (as scheduled by the *Celery Depre-cation Time-line*).

- It's no longer allowed to skip the class name in loader names. (as scheduled by the *Celery Deprecation Time-line*):

  Assuming the implicit *Loader* class name is no longer supported, for example, if you use:

  ```
  CELERY_LOADER = 'myapp.loaders'
  ```

  You need to include the loader class name, like this:

  ```
  CELERY_LOADER = 'myapp.loaders.Loader'
  ```

- CELERY_TASK_RESULT_EXPIRES now defaults to 1 day.

  Previous default setting was to expire in 5 days.

- AMQP backend: Don't use different values for *auto_delete*.

  This bug became visible with RabbitMQ 1.8.0, which no longer allows conflicting declarations for the auto_delete and durable settings.

  If you've already used Celery with this backend chances are you have to delete the previous decla-ration:

  ```
  $ camqadm exchange.delete celeryresults
  ```

- Now uses pickle instead of cPickle on Python versions <= 2.5

  cPickle is broken in Python <= 2.5.

  It unsafely and incorrectly uses relative instead of absolute imports, so for example:

  ```
  exceptions.KeyError
  ```

  becomes:

  ```
  celery.exceptions.KeyError
  ```
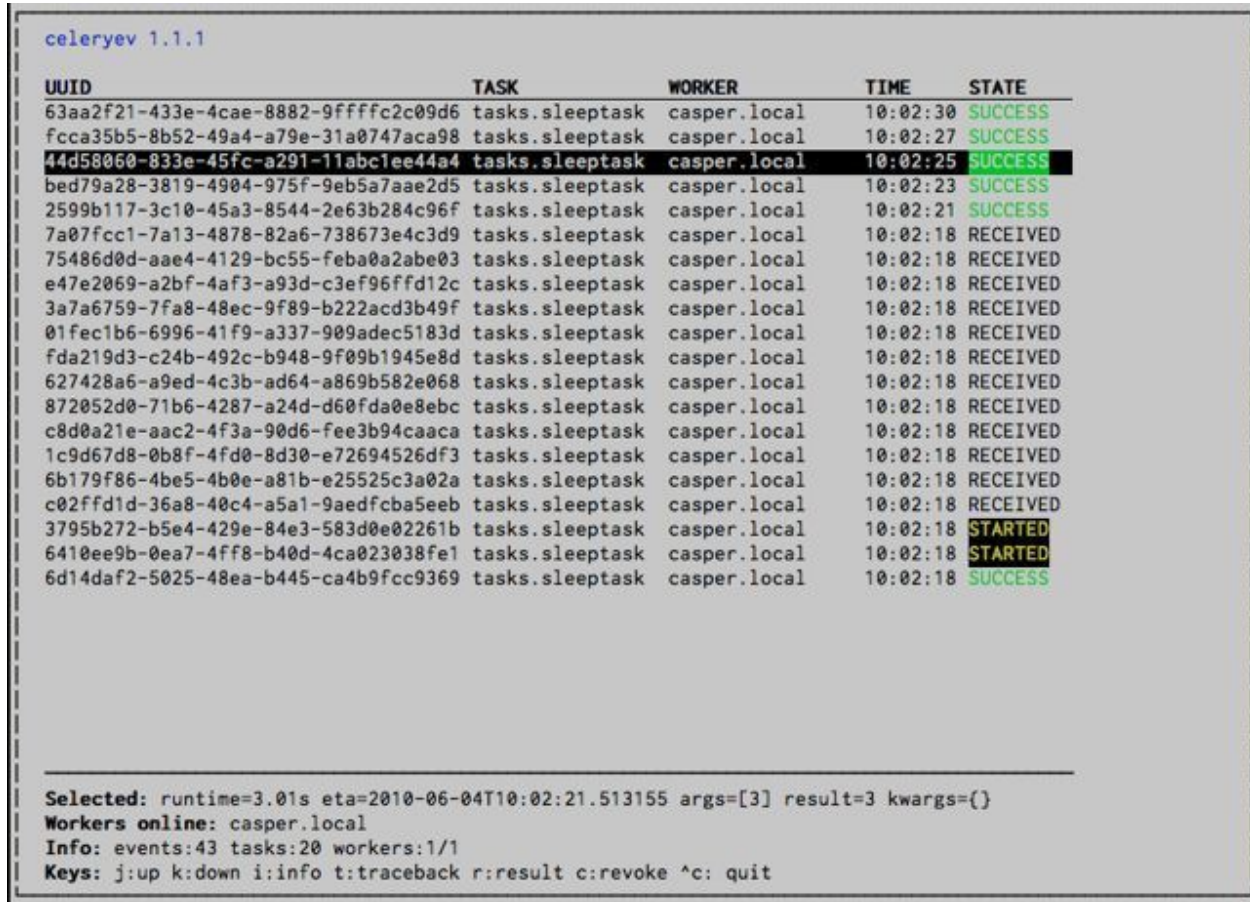
  Your best choice is to upgrade to Python 2.6, as while the pure pickle version has worse perfor-mance, it is the only safe option for older Python versions.

News

- **celeryev**: Curses Celery Monitor and Event Viewer.

  This is a simple monitor allowing you to see what tasks are executing in real-time and investigate tracebacks and results of ready tasks. It also enables you to set new rate limits and revoke tasks.

  Screenshot:

```
celeryev 1.1.1

UUID                                   TASK            WORKER        TIME       STATE
63aa2f21-433e-4cae-8882-9ffffc2c09d6 tasks.sleeptask casper.local  10:02:30 SUCCESS
fcca35b5-8b52-49a4-a79e-31a0747aca98 tasks.sleeptask casper.local  10:02:27 SUCCESS
44d58060-833e-45fc-a291-11abc1ee44a4 tasks.sleeptask casper.local  10:02:25 SUCCESS
bed79a28-3819-4904-975f-9eb5a7aae2d5 tasks.sleeptask casper.local  10:02:23 SUCCESS
2599b117-3c10-45a3-8544-2e63b284c96f tasks.sleeptask casper.local  10:02:21 SUCCESS
7a07fcc1-7a13-4878-82a6-738673e4c3d9 tasks.sleeptask casper.local  10:02:18 RECEIVED
75486d0d-aae4-4129-bc55-feba0a2abe03 tasks.sleeptask casper.local  10:02:18 RECEIVED
e47e2069-a2bf-4af3-a93d-c3ef96ffd12c tasks.sleeptask casper.local  10:02:18 RECEIVED
3a7a6759-7fa8-48ec-9f89-b222acd3b49f tasks.sleeptask casper.local  10:02:18 RECEIVED
01fec1b6-6996-41f9-a337-909adec5183d tasks.sleeptask casper.local  10:02:18 RECEIVED
fda219d3-c24b-492c-b948-9f09b1945e8d tasks.sleeptask casper.local  10:02:18 RECEIVED
627428a6-a9ed-4c3b-ad64-a869b582e068 tasks.sleeptask casper.local  10:02:18 RECEIVED
872052d0-71b6-4287-a24d-d60fda0e8ebc tasks.sleeptask casper.local  10:02:18 RECEIVED
c8d0a21e-aac2-4f3a-90d6-fee3b94caaca tasks.sleeptask casper.local  10:02:18 RECEIVED
1c9d67d8-0b8f-4fd0-8d30-e72694526df3 tasks.sleeptask casper.local  10:02:18 RECEIVED
6b179f86-4be5-4b0e-a81b-e25525c3a02a tasks.sleeptask casper.local  10:02:18 RECEIVED
c02ffd1d-36a8-40c4-a5a1-9aedfcba5eeb tasks.sleeptask casper.local  10:02:18 RECEIVED
3795b272-b5e4-429e-84e3-583d0e02261b tasks.sleeptask casper.local  10:02:18 STARTED
6410ee9b-0ea7-4ff8-b40d-4ca023038fe1 tasks.sleeptask casper.local  10:02:18 STARTED
6d14daf2-5025-48ea-b445-ca4b9fcc9369 tasks.sleeptask casper.local  10:02:18 SUCCESS




Selected: runtime=3.01s eta=2010-06-04T10:02:21.513155 args=[3] result=3 kwargs={}
Workers online: casper.local
Info: events:43 tasks:20 workers:1/1
Keys: j:up k:down i:info t:traceback r:result c:revoke ^c: quit
```

  If you run *celeryev* with the *-d* switch it will act as an event dumper, simply dumping the events it receives to standard out:

```
$ celeryev -d
-> celeryev: starting capture...
casper.local [2010-06-04 10:42:07.020000] heartbeat
casper.local [2010-06-04 10:42:14.750000] task received:
    tasks.add(61a68756-27f4-4879-b816-3cf815672b0e) args=[2, 2] kwargs={}
    eta=2010-06-04T10:42:16.669290, retries=0
casper.local [2010-06-04 10:42:17.230000] task started
    tasks.add(61a68756-27f4-4879-b816-3cf815672b0e) args=[2, 2] kwargs={}
casper.local [2010-06-04 10:42:17.960000] task succeeded:
    tasks.add(61a68756-27f4-4879-b816-3cf815672b0e)
    args=[2, 2] kwargs={} result=4, runtime=0.782663106918

The fields here are, in order: *sender hostname*, *timestamp*, *event␣
↪type* and
```

```
*additional event fields*.
```

- AMQP result backend: Now supports *.ready()*, *.successful()*, *.result*, *.status*, and even responds to changes in task state

- New user guides:

    - *Workers Guide*

    - *Canvas: Designing Work-flows*

    - *Routing Tasks*

- Worker: Standard out/error is now being redirected to the log file.

- billiard has been moved back to the Celery repository.

Module name	celery equivalent
billiard.pool	*celery.concurrency.processes.pool*
billiard.serialization	*celery.serialization*
billiard.utils.functional	*celery.utils.functional*

   The billiard distribution may be maintained, depending on interest.

- now depends on carrot >= 0.10.5

- now depends on pyparsing

- Worker: Added *–purge* as an alias to *–discard*.

- Worker: `Control-c` (SIGINT) once does warm shutdown, hitting `Control-c` twice forces termination.

- Added support for using complex Crontab-expressions in periodic tasks. For example, you can now use:

```
>>> crontab(minute='*/15')
```

   or even:

```
>>> crontab(minute='*/30', hour='8-17,1-2', day_of_week='thu-fri')
```

   See *Periodic Tasks*.

- Worker: Now waits for available pool processes before applying new tasks to the pool.

   This means it doesn't have to wait for dozens of tasks to finish at shutdown because it has applied prefetched tasks without having any pool processes available to immediately accept them.

   See issue #122.

- New built-in way to do task callbacks using `subtask`.

   See *Canvas: Designing Work-flows* for more information.

- TaskSets can now contain several types of tasks.

   `TaskSet` has been refactored to use a new syntax, please see *Canvas: Designing Work-flows* for more information.

   The previous syntax is still supported, but will be deprecated in version 1.4.

- TaskSet failed() result was incorrect.

   See issue #132.

- Now creates different loggers per task class.

    See issue #129.

- Missing queue definitions are now created automatically.

    You can disable this using the `CELERY_CREATE_MISSING_QUEUES` setting.

    The missing queues are created with the following options:

    ```
    CELERY_QUEUES[name] = {'exchange': name,
                           'exchange_type': 'direct',
                           'routing_key': 'name}
    ```

    This feature is added for easily setting up routing using the *-Q* option to the worker:

    ```
    $ celeryd -Q video, image
    ```

    See the new routing section of the User Guide for more information: *Routing Tasks*.

- New Task option: *Task.queue*

    If set, message options will be taken from the corresponding entry in `CELERY_QUEUES`. *exchange*, *exchange_type* and *routing_key* will be ignored

- Added support for task soft and hard time limits.

    New settings added:

    - `CELERYD_TASK_TIME_LIMIT`

        Hard time limit. The worker processing the task will be killed and replaced with a new one when this is exceeded.

    - `CELERYD_TASK_SOFT_TIME_LIMIT`

        Soft time limit. The *SoftTimeLimitExceeded* exception will be raised when this is exceeded. The task can catch this to, for example, clean up before the hard time limit comes.

    New command-line arguments to `celeryd` added: *–time-limit* and *–soft-time-limit*.

    What's left?

    This won't work on platforms not supporting signals (and specifically the *SIGUSR1* signal) yet. So an alternative the ability to disable the feature all together on nonconforming platforms must be implemented.

    Also when the hard time limit is exceeded, the task result should be a *TimeLimitExceeded* exception.

- Test suite is now passing without a running broker, using the carrot in-memory backend.

- Log output is now available in colors.

Log level	Color
DEBUG	Blue
WARNING	Yellow
CRITICAL	Magenta
ERROR	Red

    This is only enabled when the log output is a tty. You can explicitly enable/disable this feature using the `CELERYD_LOG_COLOR` setting.

- Added support for task router classes (like the django multi-db routers)

> – New setting: CELERY_ROUTES

This is a single, or a list of routers to traverse when sending tasks. Dictionaries in this list converts to a `celery.routes.MapRoute` instance.

Examples:

```
>>> CELERY_ROUTES = {'celery.ping': 'default',
                     'mytasks.add': 'cpu-bound',
                     'video.encode': {
                         'queue': 'video',
                         'exchange': 'media'
                         'routing_key': 'media.video.encode'}}
```

```
>>> CELERY_ROUTES = ('myapp.tasks.Router',
                     {'celery.ping': 'default'})
```

Where *myapp.tasks.Router* could be:

```
class Router(object):

    def route_for_task(self, task, args=None, kwargs=None):
        if task == 'celery.ping':
            return 'default'
```

route_for_task may return a string or a dict. A string then means it's a queue name in CELERY_QUEUES, a dict means it's a custom route.

When sending tasks, the routers are consulted in order. The first router that doesn't return *None* is the route to use. The message options is then merged with the found route settings, where the routers settings have priority.

Example if `apply_async()` has these arguments:

```
>>> Task.apply_async(immediate=False, exchange='video',
...                  routing_key='video.compress')
```

and a router returns:

```
{'immediate': True,
 'exchange': 'urgent'}
```

the final message options will be:

```
>>> task.apply_async(
...     immediate=True,
...     exchange='urgent',
...     routing_key='video.compress',
... )
```

(and any default message options defined in the `Task` class)

- New Task handler called after the task returns: `after_return()`.

- **ExceptionInfo now passed to** `on_retry()`/ `on_failure()` as `einfo` keyword argument.

- Worker: Added CELERYD_MAX_TASKS_PER_CHILD / celery worker --maxtasksperchild.

  Defines the maximum number of tasks a pool worker can process before the process is terminated and replaced by a new one.

- Revoked tasks now marked with state *REVOKED*, and *result.get()* will now raise *TaskRevokedError*.

- `celery.task.control.ping()` now works as expected.

- *apply(throw=True)* / `CELERY_EAGER_PROPAGATES_EXCEPTIONS`: Makes eager execution re-raise task errors.

- New signal: `~celery.signals.worker_process_init`: Sent inside the pool worker process at init.

- Worker: *celery worker -Q* option: Ability to specify list of queues to use, disabling other configured queues.

  For example, if `CELERY_QUEUES` defines four queues: *image*, *video*, *data* and *default*, the following command would make the worker only consume from the *image* and *video* queues:

  ```
  $ celeryd -Q image,video
  ```

- Worker: New return value for the *revoke* control command:

  Now returns:

  ```
  {'ok': 'task $id revoked'}
  ```

  instead of `True`.

- Worker: Can now enable/disable events using remote control

  Example usage:

  ```
  >>> from celery.task.control import broadcast
  >>> broadcast('enable_events')
  >>> broadcast('disable_events')
  ```

- Removed top-level tests directory. Test config now in celery.tests.config

  This means running the unit tests doesn't require any special setup. *celery/tests/__init__* now configures the `CELERY_CONFIG_MODULE` and `CELERY_LOADER` environment variables, so when *nosetests* imports that, the unit test environment is all set up.

  Before you run the tests you need to install the test requirements:

  ```
  $ pip install -r requirements/test.txt
  ```

  Running all tests:

  ```
  $ nosetests
  ```

  Specifying the tests to run:

  ```
  $ nosetests celery.tests.test_task
  ```

  Producing HTML coverage:

  ```
  $ nosetests --with-coverage3
  ```

  The coverage output is then located in *celery/tests/cover/index.html*.

- Worker: New option *–version*: Dump version info and exit.

- `celeryd-multi`: Tool for shell scripts to start multiple workers.

  Some examples:

– Advanced example with 10 workers:

  * Three of the workers processes the images and video queue

  * Two of the workers processes the data queue with loglevel DEBUG

  * the rest processes the default' queue.

```
$ celeryd-multi start 10 -l INFO -Q:1-3 images,video -Q:4,
↪5:data -Q default -L:4,5 DEBUG
```

– Get commands to start 10 workers, with 3 processes each

```
$ celeryd-multi start 3 -c 3
celeryd -n celeryd1.myhost -c 3
celeryd -n celeryd2.myhost -c 3
celeryd -n celeryd3.myhost -c 3
```

– Start 3 named workers

```
$ celeryd-multi start image video data -c 3
celeryd -n image.myhost -c 3
celeryd -n video.myhost -c 3
celeryd -n data.myhost -c 3
```

– Specify custom hostname

```
$ celeryd-multi start 2 -n worker.example.com -c 3
celeryd -n celeryd1.worker.example.com -c 3
celeryd -n celeryd2.worker.example.com -c 3
```

  Additional options are added to each celeryd, but you can also modify the
  options for ranges of or single workers

– 3 workers: Two with 3 processes, and one with 10 processes.

```
$ celeryd-multi start 3 -c 3 -c:1 10
celeryd -n celeryd1.myhost -c 10
celeryd -n celeryd2.myhost -c 3
celeryd -n celeryd3.myhost -c 3
```

– Can also specify options for named workers

```
$ celeryd-multi start image video data -c 3 -c:image 10
celeryd -n image.myhost -c 10
celeryd -n video.myhost -c 3
celeryd -n data.myhost -c 3
```

– Ranges and lists of workers in options is also allowed: (-c:1-3 can also be written as
  -c:1,2,3)

```
$ celeryd-multi start 5 -c 3  -c:1-3 10
celeryd-multi -n celeryd1.myhost -c 10
celeryd-multi -n celeryd2.myhost -c 10
celeryd-multi -n celeryd3.myhost -c 10
celeryd-multi -n celeryd4.myhost -c 3
celeryd-multi -n celeryd5.myhost -c 3
```

– Lists also work with named workers:

```
$ celeryd-multi start foo bar baz xuzzy -c 3 -c:foo,bar,
↪baz 10
celeryd-multi -n foo.myhost -c 10
celeryd-multi -n bar.myhost -c 10
celeryd-multi -n baz.myhost -c 10
celeryd-multi -n xuzzy.myhost -c 3
```

- The worker now calls the result backends *process_cleanup* method *after* task execution instead of before.

- AMQP result backend now supports Pika.

3.13.18 Change history for Celery 1.0

1.0.6

release-date 2010-06-30 09:57 a.m. CEST

release-by Ask Solem

- RabbitMQ 1.8.0 has extended their exchange equivalence tests to include *auto_delete* and *durable*. This broke the AMQP backend.

  If you've already used the AMQP backend this means you have to delete the previous definitions:

```
$ camqadm exchange.delete celeryresults
```

or:

```
$ python manage.py camqadm exchange.delete celeryresults
```

1.0.5

release-date 2010-06-01 02:36 p.m. CEST

release-by Ask Solem

Critical

- `INT`/`Control-c` killed the pool, abruptly terminating the currently executing tasks.

    Fixed by making the pool worker processes ignore `SIGINT`.

- Shouldn't close the consumers before the pool is terminated, just cancel the consumers.

    See issue #122.

- Now depends on billiard >= 0.3.1

- worker: Previously exceptions raised by worker components could stall start-up, now it correctly logs the exceptions and shuts down.

- worker: Prefetch counts was set too late. QoS is now set as early as possible, so the worker: can't slurp in all the messages at start-up.

Changes

- *celery.contrib.abortable*: Abortable tasks.

    Tasks that defines steps of execution, the task can then be aborted after each step has completed.

- *EventDispatcher*: No longer creates AMQP channel if events are disabled

- Added required RPM package names under *[bdist_rpm]* section, to support building RPMs from the sources using `setup.py`.

- Running unit tests: `NOSE_VERBOSE` environment var now enables verbose output from Nose.

- `celery.execute.apply()`: Pass log file/log level arguments as task kwargs.

    See issue #110.

- celery.execute.apply: Should return exception, not `ExceptionInfo` on error.

    See issue #111.

- Added new entries to the *FAQs*:

    – Should I use retry or acks_late?

    – Can I call a task by name?

1.0.4

release-date 2010-05-31 09:54 a.m. CEST

release-by Ask Solem

• Changelog merged with 1.0.5 as the release was never announced.

1.0.3

release-date 2010-05-15 03:00 p.m. CEST

release-by Ask Solem

Important notes

• Messages are now acknowledged *just before* the task function is executed.

   This is the behavior we've wanted all along, but couldn't have because of limitations in the multi-processing module. The previous behavior wasn't good, and the situation worsened with the release of 1.0.1, so this change will definitely improve reliability, performance and operations in general.

   For more information please see http://bit.ly/9hom6T

• Database result backend: result now explicitly sets *null=True* as *django-picklefield* version 0.1.5 changed the default behavior right under our noses :(

   See: http://bit.ly/d5OwMr

   This means those who created their Celery tables (via `syncdb` or `celeryinit`) with django-picklefield` versions >= 0.1.5 has to alter their tables to allow the result field to be *NULL* manually.

   MySQL:

```
ALTER TABLE celery_taskmeta MODIFY result TEXT NULL
```

   PostgreSQL:

```
ALTER TABLE celery_taskmeta ALTER COLUMN result DROP NOT NULL
```

• Removed *Task.rate_limit_queue_type*, as it wasn't really useful and made it harder to refactor some parts.

• Now depends on carrot >= 0.10.4

• Now depends on billiard >= 0.3.0

News

• AMQP backend: Added timeout support for *result.get()* / *result.wait()*.

• New task option: *Task.acks_late* (default: `CELERY_ACKS_LATE`)

   Late ack means the task messages will be acknowledged **after** the task has been executed, not *just before*, which is the default behavior.

   Note: This means the tasks may be executed twice if the worker crashes in mid-execution. Not acceptable for most applications, but desirable for others.

- Added Crontab-like scheduling to periodic tasks.

    Like a cronjob, you can specify units of time of when you'd like the task to execute. While not a full implementation of **cron**'s features, it should provide a fair degree of common scheduling needs.

    You can specify a minute (0-59), an hour (0-23), and/or a day of the week (0-6 where 0 is Sunday, or by names: sun, mon, tue, wed, thu, fri, sat).

    Examples:

    ```python
    from celery.schedules import crontab
    from celery.decorators import periodic_task

    @periodic_task(run_every=crontab(hour=7, minute=30))
    def every_morning():
        print('Runs every morning at 7:30a.m')

    @periodic_task(run_every=crontab(hour=7, minute=30, day_of_week='mon'))
    def every_monday_morning():
        print('Run every monday morning at 7:30a.m')

    @periodic_task(run_every=crontab(minutes=30))
    def every_hour():
        print('Runs every hour on the clock (e.g., 1:30, 2:30, 3:30 etc.).')
    ```

    Note: This a late addition. While we have unit tests, due to the nature of this feature we haven't been able to completely test this in practice, so consider this experimental.

- *TaskPool.apply_async*: Now supports the *accept_callback* argument.
- *apply_async*: Now raises ValueError if task args isn't a list, or kwargs isn't a tuple (Issue #95).
- *Task.max_retries* can now be *None*, which means it will retry forever.
- celerybeat: Now reuses the same connection when publishing large sets of tasks.
- Modified the task locking example in the documentation to use *cache.add* for atomic locking.
- Added experimental support for a *started* status on tasks.

    If *Task.track_started* is enabled 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 which task is currently running.

    The global default can be overridden by the CELERY_TRACK_STARTED setting.

- User Guide: New section *Tips and Best Practices*.

    Contributions welcome!

Remote control commands

- Remote control commands can now send replies back to the caller.

    Existing commands has been improved to send replies, and the client interface in *celery.task.control* has new keyword arguments: *reply*, *timeout* and *limit*. Where reply means it will wait for replies,

timeout is the time in seconds to stop waiting for replies, and limit is the maximum number of replies to get.

By default, it will wait for as many replies as possible for one second.

– rate_limit(task_name, destination=all, reply=False, timeout=1, limit=0)

Worker returns *{'ok': message}* on success, or *{'failure': message}* on failure.

```
>>> from celery.task.control import rate_limit
>>> rate_limit('tasks.add', '10/s', reply=True)
[{'worker1': {'ok': 'new rate limit set successfully'}},
 {'worker2': {'ok': 'new rate limit set successfully'}}]
```

– ping(destination=all, reply=False, timeout=1, limit=0)

Worker returns the simple message *"pong"*.

```
>>> from celery.task.control import ping
>>> ping(reply=True)
[{'worker1': 'pong'},
 {'worker2': 'pong'},
```

– revoke(destination=all, reply=False, timeout=1, limit=0)

Worker simply returns *True*.

```
>>> from celery.task.control import revoke
>>> revoke('419e46eb-cf6a-4271-86a8-442b7124132c',
→reply=True)
[{'worker1': True},
 {'worker2'; True}]
```

- You can now add your own remote control commands!

    Remote control commands are functions registered in the command registry. Registering a command is done using *celery.worker.control.Panel.register()*:

```
from celery.task.control import Panel

@Panel.register
def reset_broker_connection(state, **kwargs):
    state.consumer.reset_connection()
    return {'ok': 'connection re-established'}
```

    With this module imported in the worker, you can launch the command using *celery.task.control.broadcast*:

```
>>> from celery.task.control import broadcast
>>> broadcast('reset_broker_connection', reply=True)
[{'worker1': {'ok': 'connection re-established'},
 {'worker2': {'ok': 'connection re-established'}}]
```

    TIP You can choose the worker(s) to receive the command by using the *destination* argument:

```
>>> broadcast('reset_broker_connection', destination=['worker1'])
[{'worker1': {'ok': 'connection re-established'}]
```

- New remote control command: *dump_reserved*

    Dumps tasks reserved by the worker, waiting to be executed:

```
>>> from celery.task.control import broadcast
>>> broadcast('dump_reserved', reply=True)
[{'myworker1': [<TaskRequest ....>]}]
```

- New remote control command: *dump_schedule*

    Dumps the workers currently registered ETA schedule. These are tasks with an *eta* (or *countdown*)
    argument waiting to be executed by the worker.

```
>>> from celery.task.control import broadcast
>>> broadcast('dump_schedule', reply=True)
[{'w1': []},
 {'w3': []},
 {'w2': ['0. 2010-05-12 11:06:00 pri0 <TaskRequest
            {name:'opalfeeds.tasks.refresh_feed_slice',
             id:'95b45760-4e73-4ce8-8eac-f100aa80273a',
             args:'(<Feeds freq_max:3600 freq_min:60
                        start:2184.0 stop:3276.0>,)',
             kwargs:'{'page': 2}'}>']},
 {'w4': ['0. 2010-05-12 11:00:00 pri0 <TaskRequest
            {name:'opalfeeds.tasks.refresh_feed_slice',
             id:'c053480b-58fb-422f-ae68-8d30a464edfe',
             args:'(<Feeds freq_max:3600 freq_min:60
                        start:1092.0 stop:2184.0>,)',
             kwargs:'{\'page\': 1}'}>',
        '1. 2010-05-12 11:12:00 pri0 <TaskRequest
            {name:'opalfeeds.tasks.refresh_feed_slice',
             id:'ab8bc59e-6cf8-44b8-88d0-f1af57789758',
             args:'(<Feeds freq_max:3600 freq_min:60
                        start:3276.0 stop:4365>,)',
             kwargs:'{\'page\': 3}'}>']}]
```

Fixes

- Mediator thread no longer blocks for more than 1 second.

    With rate limits enabled and when there was a lot of remaining time, the mediator thread could
    block shutdown (and potentially block other jobs from coming in).

- Remote rate limits wasn't properly applied (Issue #98).

- Now handles exceptions with Unicode messages correctly in *TaskRequest.on_failure*.

- Database backend: *TaskMeta.result*: default value should be *None* not empty string.

1.0.2

release-date 2010-03-31 12:50 p.m. CET

release-by Ask Solem

- Deprecated: `CELERY_BACKEND`, please use `CELERY_RESULT_BACKEND` instead.

- We now use a custom logger in tasks. This logger supports task magic keyword arguments in formats.

    The default format for tasks (`CELERYD_TASK_LOG_FORMAT`) now includes the id and the name
    of tasks so the origin of task log messages can easily be traced.

> **Example output::**
>
> > **[2010-03-25 13:11:20,317: INFO/PoolWorker-1]** [tasks.add(a6e1c5ad-60d9-42a0-8b24-
> > 9e39363125a4)] Hello from add
>
> To revert to the previous behavior you can set:

```
CELERYD_TASK_LOG_FORMAT = """
    [%(asctime)s: %(levelname)s/%(processName)s] %(message)s
""".strip()
```

- Unit tests: Don't disable the django test database tear down, instead fixed the underlying issue which was caused by modifications to the *DATABASE_NAME* setting (Issue #82).

- Django Loader: New config CELERY_DB_REUSE_MAX (max number of tasks to reuse the same database connection)

  The default is to use a new connection for every task. We'd very much like to reuse the connection, but a safe number of reuses isn't known, and we don't have any way to handle the errors that might happen, which may even be database dependent.

  See: http://bit.ly/94fwdd

- worker: The worker components are now configurable: CELERYD_POOL, CELERYD_CONSUMER, CELERYD_MEDIATOR, and CELERYD_ETA_SCHEDULER.

  The default configuration is as follows:

  ```
  CELERYD_POOL = 'celery.concurrency.processes.TaskPool'
  CELERYD_MEDIATOR = 'celery.worker.controllers.Mediator'
  CELERYD_ETA_SCHEDULER = 'celery.worker.controllers.ScheduleController'
  CELERYD_CONSUMER = 'celery.worker.consumer.Consumer'
  ```

  The CELERYD_POOL setting makes it easy to swap out the multiprocessing pool with a threaded pool, or how about a twisted/eventlet pool?

  Consider the competition for the first pool plug-in started!

- Debian init-scripts: Use *-a* not *&&* (Issue #82).

- Debian init-scripts: Now always preserves *$CELERYD_OPTS* from the */etc/default/celeryd* and */etc/default/celerybeat*.

- celery.beat.Scheduler: Fixed a bug where the schedule wasn't properly flushed to disk if the schedule hadn't been properly initialized.

- celerybeat: Now syncs the schedule to disk when receiving the SIGTERM and SIGINT signals.

- Control commands: Make sure keywords arguments aren't in Unicode.

- ETA scheduler: Was missing a logger object, so the scheduler crashed when trying to log that a task had been revoked.

- management.commands.camqadm: Fixed typo *camqpadm -> camqadm* (Issue #83).

- PeriodicTask.delta_resolution: wasn't working for days and hours, now fixed by rounding to the nearest day/hour.

- Fixed a potential infinite loop in *BaseAsyncResult.__eq__*, although there's no evidence that it has ever been triggered.

- worker: Now handles messages with encoding problems by acking them and emitting an error message.

1.0.1

release-date 2010-02-24 07:05 p.m. CET

release-by Ask Solem

- Tasks are now acknowledged early instead of late.

    This is done because messages can only be acknowledged within the same connection channel, so if the connection is lost we'd've to re-fetch the message again to acknowledge it.

    This might or might not affect you, but mostly those running tasks with a really long execution time are affected, as all tasks that's made it all the way into the pool needs to be executed before the worker can safely terminate (this is at most the number of pool workers, multiplied by the `CELERYD_PREFETCH_MULTIPLIER` setting).

    We multiply the prefetch count by default to increase the performance at times with bursts of tasks with a short execution time. If this doesn't apply to your use case, you should be able to set the prefetch multiplier to zero, without sacrificing performance.

    Note: A patch to `multiprocessing` is currently being worked on, this patch would enable us to use a better solution, and is scheduled for inclusion in the *2.0.0* release.

- The worker now shutdowns cleanly when receiving the `SIGTERM` signal.

- The worker now does a cold shutdown if the `SIGINT` signal is received (`Control-c`), this means it tries to terminate as soon as possible.

- Caching of results now moved to the base backend classes, so no need to implement this functionality in the base classes.

- Caches are now also limited in size, so their memory usage doesn't grow out of control.

    You can set the maximum number of results the cache can hold using the `CELERY_MAX_CACHED_RESULTS` setting (the default is five thousand results). In addition, you can re-fetch already retrieved results using *backend.reload_task_result* + *backend.reload_taskset_result* (that's for those who want to send results incrementally).

- The worker now works on Windows again.

    > **Warning:** If you're using Celery with Django, you can't use *project.settings* as the settings module name, but the following should work:
    >
    > ```
    > $ python manage.py celeryd --settings=settings
    > ```

- Execution: *.messaging.TaskPublisher.send_task* now incorporates all the functionality apply_async previously did.

    Like converting countdowns to ETA, so `celery.execute.apply_async()` is now simply a convenient front-end to `celery.messaging.TaskPublisher.send_task()`, using the task classes default options.

    Also `celery.execute.send_task()` has been introduced, which can apply tasks using just the task name (useful if the client doesn't have the destination task in its task registry).

    Example:

```
>>> from celery.execute import send_task
>>> result = send_task('celery.ping', args=[], kwargs={})
>>> result.get()
'pong'
```

- *camqadm*: This is a new utility for command-line access to the AMQP API.

    Excellent for deleting queues/bindings/exchanges, experimentation and testing:

    ```
    $ camqadm
    1> help
    ```

    Gives an interactive shell, type *help* for a list of commands.

    When using Django, use the management command instead:

    ```
    $ python manage.py camqadm
    1> help
    ```

- Redis result backend: To conform to recent Redis API changes, the following settings has been deprecated:

    – *REDIS_TIMEOUT*

    – *REDIS_CONNECT_RETRY*

    These will emit a *DeprecationWarning* if used.

    A *REDIS_PASSWORD* setting has been added, so you can use the new simple authentication mechanism in Redis.

- The redis result backend no longer calls *SAVE* when disconnecting, as this is apparently better handled by Redis itself.

- If *settings.DEBUG* is on, the worker now warns about the possible memory leak it can result in.

- The ETA scheduler now sleeps at most two seconds between iterations.

- The ETA scheduler now deletes any revoked tasks it might encounter.

    As revokes aren't yet persistent, this is done to make sure the task is revoked even though, for example, it's currently being hold because its ETA is a week into the future.

- The *task_id* argument is now respected even if the task is executed eagerly (either using apply, or `CELERY_ALWAYS_EAGER`).

- The internal queues are now cleared if the connection is reset.

- New magic keyword argument: *delivery_info*.

    Used by retry() to resend the task to its original destination using the same exchange/routing_key.

- Events: Fields wasn't passed by *.send()* (fixes the UUID key errors in celerymon)

- Added *–schedule/-s* option to the worker, so it is possible to specify a custom schedule filename when using an embedded `celerybeat` server (the *-B/–beat*) option.

- Better Python 2.4 compatibility. The test suite now passes.

- task decorators: Now preserve docstring as *cls.__doc__*, (was previously copied to *cls.run.__doc__*)

- The *testproj* directory has been renamed to *tests* and we're now using *nose + django-nose* for test discovery, and *unittest2* for test cases.

- New pip requirements files available in `requirements`.

- TaskPublisher: Declarations are now done once (per process).

- Added *Task.delivery_mode* and the `CELERY_DEFAULT_DELIVERY_MODE` setting.

    These can be used to mark messages non-persistent (i.e., so they're lost if the broker is restarted).

- Now have our own *ImproperlyConfigured* exception, instead of using the Django one.

- Improvements to the Debian init-scripts: Shows an error if the program is not executable. Does not modify *CELERYD* when using django with virtualenv.

1.0.0

release-date 2010-02-10 04:00 p.m. CET

release-by Ask Solem

Backward incompatible changes

- Celery doesn't support detaching anymore, so you have to use the tools available on your platform, or something like supervisor to make `celeryd`/`celerybeat`/`celerymon` into background processes.

    We've had too many problems with the worker daemonizing itself, so it was decided it has to be removed. Example start-up scripts has been added to the *extra/* directory:

    - Debian, Ubuntu, (**start-stop-daemon**)

        extra/debian/init.d/celeryd extra/debian/init.d/celerybeat

    - macOS **launchd**

        extra/mac/org.celeryq.celeryd.plist   extra/mac/org.celeryq.celerybeat.plist   extra/mac/org.celeryq.celerymon.plist

    - Supervisor (http://supervisord.org)

        extra/supervisord/supervisord.conf

    In addition to *–detach*, the following program arguments has been removed: *–uid*, *–gid*, *–workdir*, *–chroot*, *–pidfile*, *–umask*. All good daemonization tools should support equivalent functionality, so don't worry.

    Also the following configuration keys has been removed: *CELERYD_PID_FILE*, *CELERYBEAT_PID_FILE*, *CELERYMON_PID_FILE*.

- Default worker loglevel is now *WARN*, to enable the previous log level start the worker with *–loglevel=INFO*.

- Tasks are automatically registered.

    This means you no longer have to register your tasks manually. You don't have to change your old code right away, as it doesn't matter if a task is registered twice.

    If you don't want your task to be automatically registered you can set the *abstract* attribute

    ```
    class MyTask(Task):
        abstract = True
    ```

    By using *abstract* only tasks subclassing this task will be automatically registered (this works like the Django ORM).

    If you don't want subclasses to be registered either, you can set the *autoregister* attribute to *False*.

Incidentally, this change also fixes the problems with automatic name assignment and relative imports. So you also don't have to specify a task name anymore if you use relative imports.

- You can no longer use regular functions as tasks.

    This change was added because it makes the internals a lot more clean and simple. However, you can now turn functions into tasks by using the *@task* decorator:

    ```python
    from celery.decorators import task

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

    See also:

    Tasks for more information about the task decorators.

- The periodic task system has been rewritten to a centralized solution.

    This means the worker no longer schedules periodic tasks by default, but a new daemon has been introduced: *celerybeat*.

    To launch the periodic task scheduler you have to run celerybeat:

    ```
    $ celerybeat
    ```

    Make sure this is running on one server only, if you run it twice, all periodic tasks will also be executed twice.

    If you only have one worker server you can embed it into the worker like this:

    ```
    $ celeryd --beat # Embed celerybeat in celeryd.
    ```

- The supervisor has been removed.

    This means the *-S* and *–supervised* options to *celeryd* is no longer supported. Please use something like http://supervisord.org instead.

- *TaskSet.join* has been removed, use *TaskSetResult.join* instead.

- The task status *"DONE"* has been renamed to *"SUCCESS"*.

- *AsyncResult.is_done* has been removed, use *AsyncResult.successful* instead.

- The worker no longer stores errors if *Task.ignore_result* is set, to revert to the previous behavior set `CELERY_STORE_ERRORS_EVEN_IF_IGNORED` to *True*.

- The statistics functionality has been removed in favor of events, so the *-S* and *–statistics'* switches has been removed.

- The module *celery.task.strategy* has been removed.

- *celery.discovery* has been removed, and it's `autodiscover` function is now in *celery.loaders.djangoapp*. Reason: Internal API.

- The `CELERY_LOADER` environment variable now needs loader class name in addition to module name,

    For example, where you previously had: *"celery.loaders.default"*, you now need *"celery.loaders.default.Loader"*, using the previous syntax will result in a *DeprecationWarning*.

- Detecting the loader is now lazy, and so isn't done when importing *celery.loaders*.

To make this happen *celery.loaders.settings* has been renamed to *load_settings* and is now a function returning the settings object. *celery.loaders.current_loader* is now also a function, returning the current loader.

So:

```
loader = current_loader
```

needs to be changed to:

```
loader = current_loader()
```

Deprecations

- The following configuration variables has been renamed and will be deprecated in v2.0:

    - `CELERYD_DAEMON_LOG_FORMAT` -> `CELERYD_LOG_FORMAT`

    - `CELERYD_DAEMON_LOG_LEVEL` -> `CELERYD_LOG_LEVEL`

    - `CELERY_AMQP_CONNECTION_TIMEOUT` -> `CELERY_BROKER_CONNECTION_TIMEOUT`

    - `CELERY_AMQP_CONNECTION_RETRY` -> `CELERY_BROKER_CONNECTION_RETRY`

    - `CELERY_AMQP_CONNECTION_MAX_RETRIES` -> `CELERY_BROKER_CONNECTION_MAX_RETRIES`

    - `SEND_CELERY_TASK_ERROR_EMAILS` -> `CELERY_SEND_TASK_ERROR_EMAILS`

- The public API names in celery.conf has also changed to a consistent naming scheme.

- We now support consuming from an arbitrary number of queues.

    To do this we had to rename the configuration syntax. If you use any of the custom AMQP routing options (queue/exchange/routing_key, etc.), you should read the new FAQ entry: *Can I send some tasks to only some servers?*.

    The previous syntax is deprecated and scheduled for removal in v2.0.

- *TaskSet.run* has been renamed to *TaskSet.apply_async*.

    TaskSet.run has now been deprecated, and is scheduled for removal in v2.0.

News

- Rate limiting support (per task type, or globally).

- New periodic task system.

- Automatic registration.

- New cool task decorator syntax.

- worker: now sends events if enabled with the *-E* argument.

    Excellent for monitoring tools, one is already in the making (https://github.com/celery/celerymon).

    Current events include: `worker-heartbeat`, task-[received/succeeded/failed/retried], `worker-online`, `worker-offline`.

- You can now delete (revoke) tasks that's already been applied.

- You can now set the hostname the worker identifies as using the *–hostname* argument.

- Cache backend now respects the `CELERY_TASK_RESULT_EXPIRES` setting.

- Message format has been standardized and now uses ISO-8601 format for dates instead of datetime.

- worker now responds to the `SIGHUP` signal by restarting itself.

- Periodic tasks are now scheduled on the clock.

   That is, *timedelta(hours=1)* means every hour at :00 minutes, not every hour from the server starts. To revert to the previous behavior you can set *PeriodicTask.relative = True*.

- Now supports passing execute options to a TaskSets list of args.

   Example:

   ```
   >>> ts = TaskSet(add, [([2, 2], {}, {'countdown': 1}),
   ...                     ([4, 4], {}, {'countdown': 2}),
   ...                     ([8, 8], {}, {'countdown': 3})])
   >>> ts.run()
   ```

- Got a 3x performance gain by setting the prefetch count to four times the concurrency, (from an average task round-trip of 0.1s to 0.03s!).

   A new setting has been added: `CELERYD_PREFETCH_MULTIPLIER`, which is set to *4* by default.

- Improved support for webhook tasks.

   celery.task.rest is now deprecated, replaced with the new and shiny *celery.task.http*. With more reflective names, sensible interface, and it's possible to override the methods used to perform HTTP requests.

- The results of task sets are now cached by storing it in the result backend.

Changes

- Now depends on carrot >= 0.8.1

- New dependencies: billiard, python-dateutil, django-picklefield.

- No longer depends on python-daemon

- The *uuid* distribution is added as a dependency when running Python 2.4.

- Now remembers the previously detected loader by keeping it in the `CELERY_LOADER` environment variable.

   This may help on windows where fork emulation is used.

- ETA no longer sends datetime objects, but uses ISO 8601 date format in a string for better compatibility with other platforms.

- No longer sends error mails for retried tasks.

- Task can now override the backend used to store results.

- Refactored the ExecuteWrapper, *apply* and `CELERY_ALWAYS_EAGER` now also executes the task callbacks and signals.

- Now using a proper scheduler for the tasks with an ETA.

   This means waiting ETA tasks are sorted by time, so we don't have to poll the whole list all the time.

- Now also imports modules listed in `CELERY_IMPORTS` when running with django (as documented).

- Log level for stdout/stderr changed from INFO to ERROR

- ImportErrors are now properly propagated when auto-discovering tasks.

- You can now use *celery.messaging.establish_connection* to establish a connection to the broker.

- When running as a separate service the periodic task scheduler does some smart moves to not poll too regularly.

    If you need faster poll times you can lower the value of `CELERYBEAT_MAX_LOOP_INTERVAL`.

- You can now change periodic task intervals at runtime, by making *run_every* a property, or subclassing *PeriodicTask.is_due*.

- The worker now supports control commands enabled through the use of a broadcast queue, you can remotely revoke tasks or set the rate limit for a task type. See `celery.task.control`.

- The services now sets informative process names (as shown in *ps* listings) if the setproctitle module is installed.

- *NotRegistered* now inherits from `KeyError`, and *TaskRegistry.__getitem__*+'pop* raises *NotRegistered* instead

- You can set the loader via the `CELERY_LOADER` environment variable.

- You can now set `CELERY_IGNORE_RESULT` to ignore task results by default (if enabled, tasks doesn't save results or errors to the backend used).

- The worker now correctly handles malformed messages by throwing away and acknowledging the message, instead of crashing.

Bugs

- Fixed a race condition that could happen while storing task results in the database.

Documentation

- Reference now split into two sections; API reference and internal module reference.

0.8.4

release-date 2010-02-05 01:52 p.m. CEST

release-by Ask Solem

- Now emits a warning if the –detach argument is used. –detach shouldn't be used anymore, as it has several not easily fixed bugs related to it. Instead, use something like start-stop-daemon, supervisor or **launchd** (macOS).

- Make sure logger class is process aware, even if running Python >= 2.6.

- Error emails are not sent anymore when the task is retried.

0.8.3

release-date 2009-12-22 09:43 a.m. CEST

release-by Ask Solem

- Fixed a possible race condition that could happen when storing/querying task results using the database backend.

- Now has console script entry points in the `setup.py` file, so tools like zc.buildout will correctly install the programs `celeryd` and `celeryinit`.

0.8.2

release-date 2009-11-20 03:40 p.m. CEST

release-by Ask Solem

- QOS Prefetch count wasn't applied properly, as it was set for every message received (which apparently behaves like, "receive one more"), instead of only set when our wanted value changed.

0.8.1

release-date 2009-11-16 05:21 p.m. CEST

release-by Ask Solem

Very important note

This release (with carrot 0.8.0) enables AMQP QoS (quality of service), which means the workers will only receive as many messages as it can handle at a time. As with any release, you should test this version upgrade on your development servers before rolling it out to production!

Important changes

- If you're using Python < 2.6 and you use the multiprocessing backport, then multiprocessing version 2.6.2.1 is required.

- All AMQP_* settings has been renamed to BROKER_*, and in addition AMQP_SERVER has been renamed to BROKER_HOST, so before where you had:

```
AMQP_SERVER = 'localhost'
AMQP_PORT = 5678
AMQP_USER = 'myuser'
AMQP_PASSWORD = 'mypassword'
AMQP_VHOST = 'celery'
```

You need to change that to:

```
BROKER_HOST = 'localhost'
BROKER_PORT = 5678
BROKER_USER = 'myuser'
BROKER_PASSWORD = 'mypassword'
BROKER_VHOST = 'celery'
```

- Custom carrot backends now need to include the backend class name, so before where you had:

```
CARROT_BACKEND = 'mycustom.backend.module'
```

you need to change it to:

```
CARROT_BACKEND = 'mycustom.backend.module.Backend'
```

where *Backend* is the class name. This is probably *"Backend"*, as that was the previously implied name.

- New version requirement for carrot: 0.8.0

Changes

- Incorporated the multiprocessing backport patch that fixes the *processName* error.
- Ignore the result of PeriodicTask's by default.
- Added a Redis result store backend
- Allow `/etc/default/celeryd` to define additional options for the `celeryd` init-script.
- MongoDB periodic tasks issue when using different time than UTC fixed.
- Windows specific: Negate test for available `os.fork` (thanks @miracle2k).
- Now tried to handle broken PID files.
- Added a Django test runner to contrib that sets *CELERY_ALWAYS_EAGER = True* for testing with the database backend.
- Added a `CELERY_CACHE_BACKEND` setting for using something other than the Django-global cache backend.
- Use custom implementation of `functools.partial` for Python 2.4 support (Probably still problems with running on 2.4, but it will eventually be supported)
- Prepare exception to pickle when saving *RETRY* status for all backends.
- SQLite no concurrency limit should only be effective if the database backend is used.

0.8.0

release-date 2009-09-22 03:06 p.m. CEST

release-by Ask Solem

Backward incompatible changes

- Add traceback to result value on failure.

  > **Note:** If you use the database backend you have to re-create the database table *celery_taskmeta*.
  >
  > Contact the *Mailing list* or *IRC* channel for help doing this.

- Database tables are now only created if the database backend is used, so if you change back to the database backend at some point, be sure to initialize tables (django: *syncdb*, python: *celeryinit*).

  Note: This is only applies if using Django version 1.1 or higher.

- Now depends on *carrot* version 0.6.0.
- Now depends on python-daemon 1.4.8

Important changes

- Celery can now be used in pure Python (outside of a Django project).

> This means Celery is no longer Django specific.
>
> For more information see the FAQ entry *Is Celery for Django only?*.

- Celery now supports task retries.

  > See *Retrying* for more information.

- We now have an AMQP result store backend.

  > It uses messages to publish task return value and status. And it's incredibly fast!
  >
  > See issue #6 for more info!

- AMQP QoS (prefetch count) implemented:

  > This to not receive more messages than we can handle.

- Now redirects stdout/stderr to the workers log file when detached

- **Now uses *inspect.getargspec* to only pass default arguments** the task supports.

- **Add Task.on_success, .on_retry, .on_failure handlers**

  > See `celery.task.base.Task.on_success(),` celery.task.base.Task.
  > on_retry(), celery.task.base.Task.on_failure(),

- *celery.utils.gen_unique_id*: **Workaround for** http://bugs.python.org/issue4607

- **You can now customize what happens at worker start, at process init, etc.,** by creating your own loaders
  (see `celery.loaders.default`, celery.loaders.djangoapp, `celery.loaders`).

- Support for multiple AMQP exchanges and queues.

  > This feature misses documentation and tests, so anyone interested is encouraged to improve this
  > situation.

- The worker now survives a restart of the AMQP server!

  Automatically re-establish AMQP broker connection if it's lost.

  New settings:

  - **AMQP_CONNECTION_RETRY** Set to *True* to enable connection retries.
  - **AMQP_CONNECTION_MAX_RETRIES.** Maximum number of restarts before we give up. Default:
    100.

News

- **Fix an incompatibility between python-daemon and multiprocessing,** which resulted in the *[Errno 10] No
  child processes* problem when detaching.

- **Fixed a possible DjangoUnicodeDecodeError being raised when saving pickled** data to Django's Mem-
  cached cache backend.

- Better Windows compatibility.

- **New version of the pickled field (taken from** http://www.djangosnippets.org/snippets/513/)

- **New signals introduced:** *task_sent*, *task_prerun* and *task_postrun*, see `celery.signals` for more infor-
  mation.

- *TaskSetResult.join* **caused** *TypeError* **when** *timeout=None*. Thanks Jerzy Kozera. Closes #31

- *views.apply* **should return** *HttpResponse* **instance.** Thanks to Jerzy Kozera. Closes #32

- *PeriodicTask*: **Save conversion of** *run_every* **from** *int* to *timedelta* to the class attribute instead of on the instance.

- **Exceptions has been moved to** *celery.exceptions*, **but are still** available in the previous module.

- **Try to rollback transaction and retry saving result if an error happens** while setting task status with the database backend.

- jail() refactored into `celery.execute.ExecuteWrapper`.

- *views.apply* now correctly sets mime-type to "application/json"

- *views.task_status* now returns exception if state is `RETRY`

- *views.task_status* **now returns traceback if state is** `FAILURE` or `RETRY`

- Documented default task arguments.

- Add a sensible __repr__ to ExceptionInfo for easier debugging

- **Fix documentation typo .. *import map -> .. import dmap*.** Thanks to @mikedizon.

0.6.0

release-date 2009-08-07 06:54 a.m. CET

release-by Ask Solem

Important changes

- **Fixed a bug where tasks raising unpickleable exceptions crashed pool** workers. So if you've had pool workers mysteriously disappearing, or problems with the worker stopping working, this has been fixed in this version.

- Fixed a race condition with periodic tasks.

- **The task pool is now supervised, so if a pool worker crashes,** goes away or stops responding, it is automatically replaced with a new one.

- Task.name is now automatically generated out of class module+name, for example *"djangotwitter.tasks.UpdateStatusesTask"*. Very convenient. No idea why we didn't do this before. Some documentation is updated to not manually specify a task name.

News

- Tested with Django 1.1

- New Tutorial: Creating a click counter using Carrot and Celery

- **Database entries for periodic tasks are now created at the workers** start-up instead of for each check (which has been a forgotten TODO/XXX in the code for a long time)

- **New settings variable:** `CELERY_TASK_RESULT_EXPIRES` Time (in seconds, or a *datetime.timedelta* object) for when after stored task results are deleted. For the moment this only works for the database backend.

- **The worker now emits a debug log message for which periodic tasks** has been launched.

- **The periodic task table is now locked for reading while getting** periodic task status (MySQL only so far, seeking patches for other engines)

- **A lot more debugging information is now available by turning on the** *DEBUG* log level (*–loglevel=DEBUG*).

- Functions/methods with a timeout argument now works correctly.

- **New:** *celery.strategy.even_time_distribution*: With an iterator yielding task args, kwargs tuples, evenly distribute the processing of its tasks throughout the time window available.

- Log message *Unknown task ignored. . .* now has log level *ERROR*

- **Log message when task is received is now emitted for all tasks, even if** the task has an ETA (estimated time of arrival). Also the log message now includes the ETA for the task (if any).

- **Acknowledgment now happens in the pool callback. Can't do ack in the job** target, as it's not pickleable (can't share AMQP connection, etc.).

- Added note about .delay hanging in README

- Tests now passing in Django 1.1

- Fixed discovery to make sure app is in INSTALLED_APPS

- **Previously overridden pool behavior (process reap, wait until pool worker** available, etc.) is now handled by *multiprocessing.Pool* itself.

- Convert statistics data to Unicode for use as kwargs. Thanks Lucy!

0.4.1

release-date 2009-07-02 01:42 p.m. CET

release-by Ask Solem

- Fixed a bug with parsing the message options (*mandatory*, *routing_key*, *priority*, *immediate*)

0.4.0

release-date 2009-07-01 07:29 p.m. CET

release-by Ask Solem

- Adds eager execution. *celery.execute.apply'|'Task.apply* executes the function blocking until the task is done, for API compatibility it returns a *celery.result.EagerResult* instance. You can configure Celery to always run tasks locally by setting the CELERY_ALWAYS_EAGER setting to *True*.

- Now depends on *anyjson*.

- 99% coverage using Python *coverage* 3.0.

0.3.20

release-date 2009-06-25 08:42 p.m. CET

release-by Ask Solem

- New arguments to *apply_async* (the advanced version of *delay_task*), *countdown* and *eta*;

```
>>> # Run 10 seconds into the future.
>>> res = apply_async(MyTask, countdown=10);
```

```
>>> # Run 1 day from now
>>> res = apply_async(MyTask,
...                    eta=datetime.now() + timedelta(days=1))
```

- Now unlinks stale PID files

- Lots of more tests.

- Now compatible with carrot >= 0.5.0.

- **IMPORTANT** The *subtask_ids* attribute on the *TaskSetResult* instance has been removed. To get this information instead use:

```
>>> subtask_ids = [subtask.id for subtask in ts_res.subtasks]
```

- *Taskset.run()* now respects extra message options from the task class.

- Task: Add attribute *ignore_result*: Don't store the status and return value. This means you can't use the *celery.result.AsyncResult* to check if the task is done, or get its return value. Only use if you need the performance and is able live without these features. Any exceptions raised will store the return value/status as usual.

- Task: Add attribute *disable_error_emails* to disable sending error emails for that task.

- Should now work on Windows (although running in the background won't work, so using the *–detach* argument results in an exception being raised).

- Added support for statistics for profiling and monitoring. To start sending statistics start the worker with the *–statistics option. Then after a while you can dump the results by running 'python manage.py celerystats*. See *celery.monitoring* for more information.

- The Celery daemon can now be supervised (i.e., it is automatically restarted if it crashes). To use this start the worker with the *–supervised'* option (or alternatively *-S*).

- views.apply: View calling a task.

    Example:

    ```
    http://e.com/celery/apply/task_name/arg1/arg2//?kwarg1=a&kwarg2=b
    ```

    > **Warning:** Use with caution! Don't expose this URL to the public without first ensuring that your code is safe!

- Refactored *celery.task*. It's now split into three modules:

    - `celery.task`

        Contains *apply_async*, *delay_task*, *discard_all*, and task shortcuts, plus imports objects from *celery.task.base* and *celery.task.builtins*

    - `celery.task.base`

        Contains task base classes: *Task*, *PeriodicTask*, *TaskSet*, *AsynchronousMapTask*, *ExecuteRemoteTask*.

    - `celery.task.builtins`

        Built-in tasks: *PingTask*, *DeleteExpiredTaskMetaTask*.

0.3.7

release-date 2008-06-16 11:41 p.m. CET

release-by Ask Solem

- **IMPORTANT** Now uses AMQP's *basic.consume* instead of *basic.get*. This means we're no longer polling the broker for new messages.

- **IMPORTANT** Default concurrency limit is now set to the number of CPUs available on the system.

- **IMPORTANT** *tasks.register*: Renamed *task_name* argument to *name*, so:

```
>>> tasks.register(func, task_name='mytask')
```

has to be replaced with:

```
>>> tasks.register(func, name='mytask')
```

- The daemon now correctly runs if the pidfile is stale.

- Now compatible with carrot 0.4.5

- Default AMQP connection timeout is now 4 seconds.

- *AsyncResult.read()* was always returning *True*.

- Only use README as long_description if the file exists so easy_install doesn't break.

- *celery.view*: JSON responses now properly set its mime-type.

- *apply_async* now has a *connection* keyword argument so you can re-use the same AMQP connection if you want to execute more than one task.

- Handle failures in task_status view such that it won't throw 500s.

- Fixed typo *AMQP_SERVER* in documentation to *AMQP_HOST*.

- Worker exception emails sent to administrators now works properly.

- No longer depends on *django*, so installing *celery* won't affect the preferred Django version installed.

- Now works with PostgreSQL (psycopg2) again by registering the *PickledObject* field.

- Worker: Added *–detach* option as an alias to *–daemon*, and it's the term used in the documentation from now on.

- Make sure the pool and periodic task worker thread is terminated properly at exit (so `Control-c` works again).

- Now depends on *python-daemon*.

- Removed dependency to *simplejson*

- Cache Backend: Re-establishes connection for every task process if the Django cache backend is python-memcached/libmemcached.

- Tyrant Backend: Now re-establishes the connection for every task executed.

0.3.3

release-date 2009-06-08 01:07 p.m. CET

release-by Ask Solem

- The *PeriodicWorkController* now sleeps for 1 second between checking for periodic tasks to execute.

0.3.2

release-date 2009-06-08 01:07 p.m. CET

release-by Ask Solem

- worker: Added option *–discard*: Discard (delete!) all waiting messages in the queue.

- Worker: The *–wakeup-after* option wasn't handled as a float.

0.3.1

release-date 2009-06-08 01:07 p.m. CET

release-by Ask Solem

- The *PeriodicTask* worker is now running in its own thread instead of blocking the *TaskController* loop.

- Default *QUEUE_WAKEUP_AFTER* has been lowered to *0.1* (was *0.3*)

0.3.0

release-date 2009-06-08 12:41 p.m. CET

release-by Ask Solem

> **Warning:** This is a development version, for the stable release, please see versions 0.2.x.

VERY IMPORTANT: Pickle is now the encoder used for serializing task arguments, so be sure to flush your task queue before you upgrade.

- **IMPORTANT** TaskSet.run() now returns a `celery.result.TaskSetResult` instance, which lets you inspect the status and return values of a taskset as it was a single entity.

- **IMPORTANT** Celery now depends on carrot >= 0.4.1.

- The Celery daemon now sends task errors to the registered admin emails. To turn off this feature, set *SEND_CELERY_TASK_ERROR_EMAILS* to *False* in your *settings.py*. Thanks to Grégoire Cachet.

- You can now run the Celery daemon by using *manage.py*:

```
$ python manage.py celeryd
```

Thanks to Grégoire Cachet.

- Added support for message priorities, topic exchanges, custom routing keys for tasks. This means we've introduced *celery.task.apply_async*, a new way of executing tasks.

  You can use *celery.task.delay* and *celery.Task.delay* like usual, but if you want greater control over the message sent, you want *celery.task.apply_async* and *celery.Task.apply_async*.

  This also means the AMQP configuration has changed. Some settings has been renamed, while others are new:

  - CELERY_AMQP_EXCHANGE
  - CELERY_AMQP_PUBLISHER_ROUTING_KEY
  - CELERY_AMQP_CONSUMER_ROUTING_KEY
  - CELERY_AMQP_CONSUMER_QUEUE

– `CELERY_AMQP_EXCHANGE_TYPE`

See the entry *Can I send some tasks to only some servers?* in the *FAQ* for more information.

- Task errors are now logged using log level *ERROR* instead of *INFO*, and stack-traces are dumped. Thanks to Grégoire Cachet.

- Make every new worker process re-establish it's Django DB connection, this solving the "MySQL connection died?" exceptions. Thanks to Vitaly Babiy and Jirka Vejrazka.

- **IMPORTANT** Now using pickle to encode task arguments. This means you now can pass complex Python objects to tasks as arguments.

- Removed dependency to *yadayada*.

- Added a FAQ, see *docs/faq.rst*.

- Now converts any Unicode keys in task *kwargs* to regular strings. Thanks Vitaly Babiy.

- Renamed the *TaskDaemon* to *WorkController*.

- *celery.datastructures.TaskProcessQueue* is now renamed to *celery.pool.TaskPool*.

- The pool algorithm has been refactored for greater performance and stability.

0.2.0

release-date 2009-05-20 05:14 p.m. CET

release-by Ask Solem

- Final release of 0.2.0

- Compatible with carrot version 0.4.0.

- Fixes some syntax errors related to fetching results from the database backend.

0.2.0-pre3

release-date 2009-05-20 05:14 p.m. CET

release-by Ask Solem

- *Internal release*. Improved handling of unpickleable exceptions, *get_result* now tries to recreate something looking like the original exception.

0.2.0-pre2

release-date 2009-05-20 01:56 p.m. CET

release-by Ask Solem

- Now handles unpickleable exceptions (like the dynamically generated subclasses of *django.core.exception.MultipleObjectsReturned*).

0.2.0-pre1

release-date 2009-05-20 12:33 p.m. CET

release-by Ask Solem

- It's getting quite stable, with a lot of new features, so bump version to 0.2. This is a pre-release.

- *celery.task.mark_as_read()* and *celery.task.mark_as_failure()* has been removed. Use *celery.backends.default_backend.mark_as_read()*, and *celery.backends.default_backend.mark_as_failure()* instead.

0.1.15

release-date 2009-05-19 04:13 p.m. CET

release-by Ask Solem

- The Celery daemon was leaking AMQP connections, this should be fixed, if you have any problems with too many files open (like *emfile* errors in *rabbit.log*, please contact us!

0.1.14

release-date 2009-05-19 01:08 p.m. CET

release-by Ask Solem

- Fixed a syntax error in the *TaskSet* class (no such variable *TimeOutError*).

0.1.13

release-date 2009-05-19 12:36 p.m. CET

release-by Ask Solem

- Forgot to add *yadayada* to install requirements.

- Now deletes all expired task results, not just those marked as done.

- Able to load the Tokyo Tyrant backend class without django configuration, can specify tyrant settings directly in the class constructor.

- Improved API documentation

- Now using the Sphinx documentation system, you can build the html documentation by doing:

```
$ cd docs
$ make html
```

and the result will be in *docs/_build/html*.

0.1.12

release-date 2009-05-18 04:38 p.m. CET

release-by Ask Solem

- *delay_task()* etc. now returns *celery.task.AsyncResult* object, which lets you check the result and any failure that might've happened. It kind of works like the *multiprocessing.AsyncResult* class returned by *multiprocessing.Pool.map_async*.

- Added `dmap()` and `dmap_async()`. This works like the *multiprocessing.Pool* versions except they're tasks distributed to the Celery server. Example:

```python
>>> from celery.task import dmap
>>> import operator
>>> dmap(operator.add, [[2, 2], [4, 4], [8, 8]])
>>> [4, 8, 16]

>>> from celery.task import dmap_async
>>> import operator
>>> result = dmap_async(operator.add, [[2, 2], [4, 4], [8, 8]])
>>> result.ready()
False
>>> time.sleep(1)
>>> result.ready()
True
>>> result.result
[4, 8, 16]
```

- Refactored the task meta-data cache and database backends, and added a new backend for Tokyo Tyrant. You can set the backend in your django settings file.

    Example:

```python
CELERY_RESULT_BACKEND = 'database'; # Uses the database
CELERY_RESULT_BACKEND = 'cache'; # Uses the django cache framework
CELERY_RESULT_BACKEND = 'tyrant'; # Uses Tokyo Tyrant
TT_HOST = 'localhost'; # Hostname for the Tokyo Tyrant server.
TT_PORT = 6657; # Port of the Tokyo Tyrant server.
```

0.1.11

release-date 2009-05-12 02:08 p.m. CET

release-by Ask Solem

- The logging system was leaking file descriptors, resulting in servers stopping with the EMFILES (too many open files) error (fixed).

0.1.10

release-date 2009-05-11 12:46 p.m. CET

release-by Ask Solem

- Tasks now supports both positional arguments and keyword arguments.

- Requires carrot 0.3.8.

- The daemon now tries to reconnect if the connection is lost.

0.1.8

release-date 2009-05-07 12:27 p.m. CET

release-by Ask Solem

- Better test coverage

- More documentation

- The worker doesn't emit *Queue is empty* message if *settings.CELERYD_EMPTY_MSG_EMIT_EVERY* is 0.

0.1.7

release-date 2009-04-30 01:50 p.m. CET

release-by Ask Solem

- Added some unit tests

- Can now use the database for task meta-data (like if the task has been executed or not). Set *settings.CELERY_TASK_META*

- Can now run *python setup.py test* to run the unit tests from within the *tests* project.

- Can set the AMQP exchange/routing key/queue using *settings.CELERY_AMQP_EXCHANGE*, *settings.CELERY_AMQP_ROUTING_KEY*, and *settings.CELERY_AMQP_CONSUMER_QUEUE*.

0.1.6

release-date 2009-04-28 02:13 p.m. CET

release-by Ask Solem

- Introducing *TaskSet*. A set of subtasks is executed and you can find out how many, or if all them, are done (excellent for progress bars and such)

- Now catches all exceptions when running *Task.__call__*, so the daemon doesn't die. This doesn't happen for pure functions yet, only *Task* classes.

- *autodiscover()* now works with zipped eggs.

- Worker: Now adds current working directory to *sys.path* for convenience.

- The *run_every* attribute of *PeriodicTask* classes can now be a *datetime.timedelta()* object.

- Worker: You can now set the *DJANGO_PROJECT_DIR* variable for the worker and it will add that to *sys.path* for easy launching.

- Can now check if a task has been executed or not via HTTP.

- You can do this by including the Celery *urls.py* into your project,

```
>>> url(r'^celery/$', include('celery.urls'))
```

then visiting the following URL:

```
http://mysite/celery/$task_id/done/
```

this will return a JSON dictionary, for example:

```
{"task": {"id": "TASK_ID", "executed": true}}
```

- *delay_task* now returns string id, not *uuid.UUID* instance.

- Now has *PeriodicTasks*, to have *cron* like functionality.

- Project changed name from *crunchy* to *celery*. The details of the name change request is in *docs/name_change_request.txt*.

0.1.0

release-date 2009-04-24 11:28 a.m. CET

release-by Ask Solem

- Initial release

Sphinx started sucking by removing images from _static, so we need to add them here into actual content to ensure they are included :-(

3.14 Glossary

ack Short for *acknowledged*.

acknowledged Workers acknowledge messages to signify that a message has been handled. Failing to acknowledge a message will cause the message to be redelivered. Exactly when a transaction is considered a failure varies by transport. In AMQP the transaction fails when the connection/channel is closed (or lost), but in Redis/SQS the transaction times out after a configurable amount of time (the `visibility_timeout`).

apply Originally a synonym to *call* but used to signify that a function is executed by the current process.

billiard Fork of the Python multiprocessing library containing improvements required by Celery.

calling Sends a task message so that the task function is *executed* by a worker.

cipater Celery release 3.1 named after song by Autechre (http://www.youtube.com/watch?v=OHsaqUr_33Y)

context The context of a task contains information like the id of the task, it's arguments and what queue it was delivered to. It can be accessed as the tasks `request` attribute. See *Task Request*

early ack Short for *early acknowledgment*

early acknowledgment Task is *acknowledged* just-in-time before being executed, meaning the task won't be redelivered to another worker if the machine loses power, or the worker instance is abruptly killed, mid-execution.

Configured using `task_acks_late`.

ETA "Estimated Time of Arrival", in Celery and Google Task Queue, etc., used as the term for a delayed message that should not be processed until the specified ETA time. See *ETA and Countdown*.

executing Workers *execute* task *requests*.

idempotent Idempotence is a mathematical property that describes a function that can be called multiple times without changing the result. Practically it means that a function can be repeated many times without unintended effects, but not necessarily side-effect free in the pure sense (compare to *nullipotent*).

Further reading: https://en.wikipedia.org/wiki/Idempotent

kombu Python messaging library used by Celery to send and receive messages.

late ack Short for *late acknowledgment*

late acknowledgment Task is *acknowledged* after execution (both if successful, or if the task is raising an error), which means the task will be redelivered to another worker in the event of the machine losing power, or the worker instance being killed mid-execution.

> Configured using `task_acks_late`.

nullipotent describes a function that'll have the same effect, and give the same result, even if called zero or multiple times (side-effect free). A stronger version of *idempotent*.

pidbox A process mailbox, used to implement remote control commands.

prefetch count Maximum number of unacknowledged messages a consumer can hold and if exceeded the transport shouldn't deliver any more messages to that consumer. See *Prefetch Limits*.

prefetch multiplier The *prefetch count* is configured by using the `worker_prefetch_multiplier` setting, which is multiplied by the number of pool slots (threads/processes/greenthreads).

reentrant describes a function that can be interrupted in the middle of execution (e.g., by hardware interrupt or signal), and then safely called again later. Reentrancy isn't the same as *idempotence* as the return value doesn't have to be the same given the same inputs, and a reentrant function may have side effects as long as it can be interrupted; An idempotent function is always reentrant, but the reverse may not be true.

request Task messages are converted to *requests* within the worker. The request information is also available as the task's *context* (the `task.request` attribute).

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Bibliography

[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.

Symbols

B

H

heartbeats (*celery.events.state.Worker attribute*), 364
hello() (*celery.app.control.Inspect method*), 314
help (*celery.events.cursesmon.CursesMonitor attribute*), 471
help() (*celery.bin.multi.MultiTool method*), 407
help_title (*celery.events.cursesmon.CursesMonitor attribute*), 471
host (*celery.backends.arangodb.ArangoDbBackend attribute*), 465
host (*celery.backends.couchbase.CouchbaseBackend attribute*), 464
host (*celery.backends.couchdb.CouchBackend attribute*), 460
host (*celery.backends.elasticsearch.ElasticsearchBackend attribute*), 462
host (*celery.backends.mongodb.MongoBackend attribute*), 461
host (*celery.backends.redis.RedisBackend attribute*), 463
host (*celery.backends.riak.RiakBackend attribute*), 464
host_format() (*celery.bin.base.Command method*), 390
host_format() (*in module celery.utils.nodenames*), 481
hostname, 184, 186
hostname (*celery.events.state.State.Worker attribute*), 367
hostname (*celery.events.state.Worker attribute*), 364
hostname (*celery.worker.request.Request attribute*), 375
hour (*celery.schedules.crontab attribute*), 335
http_protocol (*celery.backends.arangodb.ArangoDbBackend attribute*), 465
hub, 184, 186
Hub (*class in celery.worker.components*), 433
human_seconds (*celery.schedules.schedule attribute*), 334
human_state() (*celery.bootsteps.Blueprint method*), 322
humaninfo() (*celery.worker.request.Request method*), 375
humanize() (*celery.app.utils.Settings method*), 321
humanize() (*celery.exceptions.Retry method*), 341
humanize_seconds() (*in module celery.contrib.testing.manager*), 355
humanize_seconds() (*in module celery.utils.time*), 489

I

iblue() (*celery.utils.term.colored method*), 487
icyan() (*celery.utils.term.colored method*), 487
id (*celery.backends.database.models.Task attribute*), 473

id (*celery.backends.database.models.TaskExtended attribute*), 473
id (*celery.backends.database.models.TaskSet attribute*), 473
id (*celery.events.state.State.Task attribute*), 366
id (*celery.events.state.State.Worker attribute*), 367
id (*celery.events.state.Task attribute*), 364
id (*celery.events.state.Worker attribute*), 364
id (*celery.result.AsyncResult attribute*), 326
id (*celery.result.GroupResult attribute*), 332
id (*celery.utils.abstract.CallableSignature attribute*), 476
id (*celery.worker.request.Request attribute*), 375
idempotent, **730**
identchars (*celery.bin.amqp.AMQShell attribute*), 403
Ignore, 341
ignore_errno() (*in module celery.platforms*), 503
ignore_result (*celery.app.task.Task attribute*), 306
ignore_result (*Task attribute*), 60
ignored (*celery.result.AsyncResult attribute*), 326
igreen() (*celery.utils.term.colored method*), 487
imagenta() (*celery.utils.term.colored method*), 487
immutable (*celery.utils.abstract.CallableSignature attribute*), 476
implements_incr (*celery.backends.cache.CacheBackend attribute*), 459
import_default_modules() (*celery.loaders.base.BaseLoader method*), 343
import_from_cwd() (*celery.loaders.base.BaseLoader method*), 343
import_from_cwd() (*in module celery.utils*), 475
import_from_cwd() (*in module celery.utils.imports*), 495
import_module() (*celery.loaders.base.BaseLoader method*), 343
import_modules
    signal, 170
import_task_module() (*celery.loaders.base.BaseLoader method*), 343
imports
    setting, 226
ImproperlyConfigured, 341
in_sighandler() (*in module celery.utils.log*), 496
inc_counter (*celery.bin.amqp.AMQShell attribute*), 403
include
    setting, 226
include() (*celery.bootsteps.StartStopStep method*), 323
include() (*celery.bootsteps.Step method*), 323
include_if() (*celery.bootsteps.Step method*), 323

M