Celery Documentation

Release 2.0.3 (stable)

Ask Solem

February 04, 2014

Contents:

CHAPTER 1

Getting Started

Release 2.0

Date February 04, 2014

1.1 Introduction

Version 2.0.3

Web http://celeryproject.org/

Download http://pypi.python.org/pypi/celery/

Source http://github.com/ask/celery/

Keywords task queue, job queue, asynchronous, rabbitmq, amqp, redis, python, webhooks, queue, distributed

–

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

The execution units, called tasks, are executed concurrently on a single or more worker servers. Tasks can execute asynchronously (in the background) or synchronously (wait until ready).

Celery is already used in production to process millions of tasks a day.

Celery is written in Python, but the protocol can be implemented in any language. It can also operate with other languages using webhooks.

The recommended message broker is RabbitMQ, but support for Redis and databases (SQLAlchemy) is also available.

You may also be pleased to know that full Django integration exists, delivered by the django-celery package.

1.1.1 Overview

This is a high level overview of the architecture.

The broker pushes tasks to the worker servers. A worker server is a networked machine running `celeryd`. This can be one or more machines depending on the workload.

The result of the task can be stored for later retrieval (called its "tombstone").

1.1.2 Example

You probably want to see some code by now, so here's an example task adding two numbers:

```
from celery.decorators import task

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

You can execute the task in the background, or wait for it to finish:

```
>>> result = add.delay(4, 4)
>>> result.wait() # wait for and return the result
8
```

Simple!

1.1.3 Features

Messaging	Supported brokers include RabbitMQ, Stomp, Redis, and most common SQL databases.
Robust	Using *RabbitMQ*, celery survives most error scenarios, and your tasks will never be lost.
Distributed	Runs on one or more machines. Supports clustering when used in combination with RabbitMQ. You can set up new workers without central configuration (e.g. use your dads laptop while the queue is temporarily overloaded).
Concurrency	Tasks are executed in parallel using the `multiprocessing` module.
Scheduling	Supports recurring tasks like cron, or specifying an exact date or countdown for when after the task should be executed.
Performance	Able to execute tasks while the user waits.
Return Values	Task return values can be saved to the selected result store backend. You can wait for the result, retrieve it later, or ignore it.
Result Stores	Database, MongoDB, Redis, *Tokyo Tyrant*, AMQP (high performance).
Webhooks	Your tasks can also be HTTP callbacks, enabling cross-language communication.
Rate limiting	Supports rate limiting by using the token bucket algorithm, which accounts for bursts of traffic. Rate limits can be set for each task type, or globally for all.
Routing	Using AMQP you can route tasks arbitrarily to different workers.
Remote-control	You can rate limit and delete (revoke) tasks remotely.
Monitoring	You can capture everything happening with the workers in real-time by subscribing to events. A real-time web monitor is in development.
Serialization	Supports Pickle, JSON, YAML, or easily defined custom schemes. One task invocation can have a different scheme than another.
Tracebacks	Errors and tracebacks are stored and can be investigated after the fact.
UUID	Every task has an UUID (Universally Unique Identifier), which is the task id used to query task status and return value.
Retries	Tasks can be retried if they fail, with configurable maximum number of retries, and delays between each retry.
Task Sets	A Task set is a task consisting of several sub-tasks. You can find out how many, or if all of the sub-tasks has been executed, and even retrieve the results in order. Progress bars, anyone?
Made for Web	You can query status and results via URLs, enabling the ability to poll task status using Ajax.
Error e-mails	Can be configured to send e-mails to the administrators when tasks fails.
Supervised	Pool workers are supervised and automatically replaced if they crash.

1.1.4 Documentation

The latest documentation with user guides, tutorials and API reference is hosted at Github.

1.1.5 Installation

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

To install using `pip`,:

```
$ pip install celery
```

To install using `easy_install`,:

```
$ easy_install celery
```

Downloading and installing from source

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

You can install it by doing the following,:

```
$ tar xvfz celery-0.0.0.tar.gz
$ cd celery-0.0.0
$ python setup.py build
# python setup.py install # as root
```

Using the development version

You can clone the repository by doing the following:

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

1.2 Broker Installation

1.2.1 Installing RabbitMQ

See Installing RabbitMQ over at RabbitMQ's website. For Mac OS X see Installing RabbitMQ on OS X.

1.2.2 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:

```
$ rabbitmqctl add_user myuser mypassword

$ rabbitmqctl add_vhost myvhost

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

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

1.2.3 Installing RabbitMQ on OS X

The easiest way to install RabbitMQ on Snow Leopard is using Homebrew; the new and shiny package management system for OS X.

In this example we'll install homebrew into `/lol`, but you can choose whichever destination, even in your home directory if you want, as one of the strengths of homebrew is that it's relocateable.

Homebrew is actually a git repository, so to install homebrew, you first need to install git. Download and install from the disk image at http://code.google.com/p/git-osx-installer/downloads/list?can=3

When git is installed you can finally clone the repo, storing it at the `/lol` location:

```
$ git clone git://github.com/mxcl/homebrew /lol
```

Brew comes with a simple utility called `brew`, used to install, remove and query packages. To use it you first have to add it to `PATH`, by adding the following line to the end of your `~/.profile`:

```
export PATH="/lol/bin:/lol/sbin:$PATH"
```

Save your profile and reload it:

```
$ source ~/.profile
```

Finally, we can install rabbitmq using `brew`:

```
$ brew install rabbitmq
```

Configuring the system hostname

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

Use the `scutil` command to permanently set your hostname:

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

Then add that hostname 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 hostname starting with an IP address, (e.g. `23.10.112.31.comcast.net`), because then RabbitMQ will try to use `rabbit@23`, which is an illegal hostname.

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` 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.

1.3 First steps with Celery

- Creating a simple task
- Configuration
- Running the celery worker server
- Executing the task

1.3.1 Creating a simple task

In this example we are creating a simple task that adds two numbers. Tasks are defined in a normal python module. The module can be named whatever you like, but the convention is to call it `tasks.py`.

Our addition task looks like this:

`tasks.py`:

```
from celery.decorators import task

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

All celery tasks are classes that inherit from the `Task` class. In this case we're using a decorator that wraps the add function in an appropriate class for us automatically. The full documentation on how to create tasks and task classes is in the *Tasks* part of the user guide.

1.3.2 Configuration

Celery is configured by using a configuration module. By default this module is called `celeryconfig.py`.

> **Note** This configuration module must be on the Python path so it can be imported.

You can set a custom name for the configuration module with the `CELERY_CONFIG_MODULE` variable, but in these examples we use the default name.

Let's create our `celeryconfig.py`.

1. Configure how we communicate with the broker:

```
BROKER_HOST = "localhost"
BROKER_PORT = 5672
BROKER_USER = "myuser"
BROKER_PASSWORD = "mypassword"
BROKER_VHOST = "myvhost"
```

2. In this example we don't want to store the results of the tasks, so we'll use the simplest backend available; the AMQP backend:

   ```
   CELERY_RESULT_BACKEND = "amqp"
   ```

   The AMQP backend is non-persistent by default, and you can only fetch the result of a task once (as it's sent as a message).

3. Finally, we list the modules to import, that is, all the modules that contain tasks. This is so Celery knows about what tasks it can be asked to perform.

   We only have a single task module, `tasks.py`, which we added earlier:

   ```
   CELERY_IMPORTS = ("tasks", )
   ```

That's it.

There are more options available, like how many processes you want to process work in parallel (the `CELERY_CONCURRENCY` setting), and we could use a persistent result store backend, but for now, this should do. For all of the options available, see the *configuration directive reference*.

1.3.3 Running the celery worker server

To test we will run the worker server in the foreground, so we can see what's going on in the terminal:

```
$ celeryd --loglevel=INFO
```

However, in production you probably want to run the worker in the background as a daemon. To do this you need to use to tools provided by your platform, or something like supervisord.

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

```
$  celeryd --help
```

For info on how to run celery as standalone daemon, see *daemon mode reference*

1.3.4 Executing the task

Whenever we want to execute our task, we can use the `delay()` method of the task class.

This is a handy shortcut to the `apply_async()` method which gives greater control of the task execution. Read the *Executing Tasks* part of the user guide for more information about executing tasks.

```
>>> from tasks import add
>>> add.delay(4, 4)
<AsyncResult: 889143a6-39a2-4e52-837b-d80d33efb22d>
```

At this point, the task has been sent to the message broker. The message broker will hold on to the task until a worker server has successfully picked it up.

Note: If everything is just hanging when you execute `delay`, please check that RabbitMQ is running, and that the user/password has access to the virtual host you configured earlier.

Right now we have to check the worker log files to know what happened with the task. This is because we didn't keep the `AsyncResult` object returned by `delay()`.

The `AsyncResult` lets us find the state of the task, wait for the task to finish, get its return value (or exception if the task failed), and more.

So, let's execute the task again, but this time we'll keep track of the task by keeping the `AsyncResult`:

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

>>> result.ready() # returns True if the task has finished processing.
False

>>> result.result # task is not ready, so no return value yet.
None

>>> result.get()   # Waits until the task is done and returns the retval.
8

>>> result.result # direct access to result, doesn't re-raise errors.
8

>>> result.successful() # returns True if the task didn't end in failure.
True
```

If the task raises an exception, the return value of `result.successful()` will be `False`, and `result.result` will contain the exception instance raised by the task.

That's all for now! After this you should probably read the *User Guide*.

1.4 Periodic Tasks

1.4.1 Introduction

The `celerybeat` service enables you to schedule tasks to run at intervals.

Periodic tasks are defined as special task classes. Here's an example of a periodic task:

```
from celery.decorators import periodic_task
from datetime import timedelta

@periodic_task(run_every=timedelta(seconds=30))
def every_30_seconds():
    print("Running periodic task!")
```

1.4.2 Crontab-like schedules

If you want a little 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:

```
from celery.task.schedules import crontab
from celery.decorators import periodic_task

@periodic_task(run_every=crontab(hour=7, minute=30, day_of_week=1))
def every_monday_morning():
    print("Execute every Monday at 7:30AM.")
```

The syntax of these crontab expressions is very flexible. Some examples:

Example	Meaning
crontab()	Execute every minute.
crontab(minute=0, hour=0)	Execute daily at midnight.
crontab(minute=0, **crontab(minute=0,** hour=[0,3,6,9,12,15,18,21])	Execute every three hours—at midnight, 3am, 6am, 9am, noon, 3pm, 6pm, 9pm. Same as previous.
crontab(minute="*/15")	Execute every 15 minutes.
crontab(day_of_week="sunday") **crontab(minute="*",** hour="*", day_of_week="sun")	Execute every minute (!) at sundays. 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 divisable by three. This means: at every hour *except*: 1am, 5am, 7am, 11am, 1pm, 5pm, 7pm, 11pm
crontab(minute=0, hour="*/5")	Execute hour divisable by 5. This means that it is triggered at 3pm, not 5pm (since 3pm equals the 24-hour clock value of "15", which is divisable by 5).
crontab(minute=0, hour="*/3,8-17")	Execute every hour divisable by 3, and every hour during office hours (8am-5pm).

1.4.3 Starting celerybeat

If you want to use periodic tasks you need to start the `celerybeat` service. You have to make sure only one instance of this server is running at any time, or else you will end up with multiple executions of the same task.

To start the `celerybeat` service:

```
$ celerybeat
```

You can also start `celerybeat` with `celeryd` by using the `-B` option, this is convenient if you only have one server:

```
$ celeryd -B
```

1.5 Resources

1.5.1 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.

1.5.2 Bug tracker

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

1.5.3 Wiki

http://wiki.github.com/ask/celery/

1.5.4 Contributing

Development of `celery` happens at Github: http://github.com/ask/celery

You are 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.

1.5.5 License

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

CHAPTER 2

User Guide

Release 2.0

Date February 04, 2014

2.1 Tasks

2.1.1 Basics

A task is a class that encapsulates a function and its execution options. Given a function `create_user`, that takes two arguments: `username` and `password`, you can create a task like this:

```
from celery.task import Task

class CreateUserTask(Task):
```

```
    def run(self, username, password):
        create_user(username, password)
```

For convenience there is a shortcut decorator that turns any function into a task, `celery.decorators.task()`:

```
from celery.decorators import task
from django.contrib.auth import User

@task
def create_user(username, password):
    User.objects.create(username=username, password=password)
```

The task decorator takes the same execution options as the `Task` class does:

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

2.1.2 Default keyword arguments

Celery supports a set of default arguments that can be forwarded to any task. Tasks can choose not to take these, or list the ones they want. The worker will do the right thing.

The current default keyword arguments are:

- logfile

  The log file, can be passed on to `get_logger()` to gain access to the workers log file. See Logging.

- loglevel

  The loglevel used.

- task_id

  The unique id of the executing task.

- task_name

  Name of the executing task.

- task_retries

  How many times the current task has been retried. An integer starting at `0`.

- task_is_eager

  Set to `True` if the task is executed locally in the client, and not by a worker.

- delivery_info

  Additional message delivery information. This is a mapping containing the exchange and routing key used to deliver this task. It's used by e.g. `retry()` to resend the task to the same destination queue.

  NOTE As some messaging backends doesn't have advanced routing capabilities, you can't trust the availability of keys in this mapping.

2.1.3 Logging

You can use the workers logger to add diagnostic output to the worker log:

```
class AddTask(Task):

    def run(self, x, y, **kwargs):
        logger = self.get_logger(**kwargs)
        logger.info("Adding %s + %s" % (x, y))
        return x + y
```

or using the decorator syntax:

```
@task()
def add(x, y, **kwargs):
    logger = add.get_logger(**kwargs)
    logger.info("Adding %s + %s" % (x, y))
    return x + y
```

There are several logging levels available, and the workers `loglevel` setting decides whether or not they will be written to the log file.

Of course, you can also simply use `print` as anything written to standard out/-err will be written to the logfile as well.

2.1.4 Retrying a task if something fails

Simply use `retry()` to re-send the task. It will do the right thing, and respect the `max_retries` attribute:

```
@task()
def send_twitter_status(oauth, tweet, **kwargs):
    try:
        twitter = Twitter(oauth)
        twitter.update_status(tweet)
    except (Twitter.FailWhaleError, Twitter.LoginError), exc:
        send_twitter_status.retry(args=[oauth, tweet], kwargs=kwargs, exc=exc)
```

Here we used the `exc` argument to pass the current exception to `Task.retry()`. At each step of the retry this exception is available as the tombstone (result) of the task. When `Task.max_retries` has been exceeded this is the exception raised. However, if an `exc` argument is not provided the `RetryTaskError` exception is raised instead.

Important note: The task has to take the magic keyword arguments in order for max retries to work properly, this is because it keeps track of the current number of retries using the `task_retries` keyword argument passed on to the task. In addition, it also uses the `task_id` keyword argument to use the same task id, and `delivery_info` to route the retried task to the same destination.

Using a custom retry delay

When a task is to be retried, it will wait for a given amount of time before doing so. The default delay is in the `Task.default_retry_delay` attribute on the task. 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.

```
class MyTask(Task):
    default_retry_delay = 30 * 60 # retry in 30 minutes

    def run(self, x, y, **kwargs):
        try:
            ...
```

```
    except Exception, exc:
        self.retry([x, y], kwargs, exc=exc,
                   countdown=60) # override the default and
                                 # - retry in 1 minute
```

2.1.5 Task options

- name

  The name the task is registered as. You can set this name manually, or just use the default which is automatically generated using the module and class name.

- abstract

  Abstract classes are not registered, but are used as the superclass when making new task types by subclassing.

- max_retries

  The maximum number of attempted retries before giving up. If this is exceeded the :exc‘~celery.exceptions.MaxRetriesExceeded‘ exception will be raised. Note that you have to retry manually, it’s not something that happens automatically.

- default_retry_delay

  Default time in seconds before a retry of the task should be executed. Can be either an `int` or a `float`. Default is a 1 minute delay (`60 seconds`).

- rate_limit

  Set the rate limit for this task type, that is, how many times in a given period of time is the task allowed to run.

  If this is `None` no rate limit is in effect. If it is an integer, 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. Example: `"100/m" (hundred tasks a minute). Default is the ``CELERY_DEFAULT_RATE_LIMIT` setting, which if not specified means rate limiting for tasks is turned off by default.

- 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.

- disable_error_emails

  Disable error e-mails for this task. Default is `False`. *Note:* You can also turn off error e-mails globally using the `CELERY_SEND_TASK_ERROR_EMAILS` setting.

- serializer

  A string identifying the default serialization method to use. Defaults to the `CELERY_TASK_SERIALIZER` setting. Can be `pickle json`, `yaml`, or any custom serialization methods that have been registered with `carrot.serialization.registry`.

  Please see *Executing Tasks* for more information.

Message and routing options

- queue

  Use the routing settings from a queue defined in `CELERY_QUEUES`. If defined the `exchange` and `routing_key` options will be ignored.

- exchange

  Override the global default `exchange` for this task.

- routing_key

  Override the global default `routing_key` for this task.

- **mandatory** If set, the task message has mandatory routing. By default the task is silently dropped by the broker if it can't be routed to a queue. However - If the task is mandatory, an exception will be raised instead.

- **immediate** Request immediate delivery. If the task cannot be routed to a task worker immediately, an exception will be raised. This is instead of the default behavior, where the broker will accept and queue the task, but with no guarantee that the task will ever be executed.

- **priority** The message priority. A number from `0` to `9`, where `0` is the highest. **Note:** RabbitMQ does not support priorities yet.

See *Executing Tasks* for more information about the messaging options available, also *Routing Tasks*.

2.1.6 Example

Let's take a real wold example; A blog where comments posted needs 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.

We have a Django blog application allowing comments on blog posts. We'll describe parts of the models/views and tasks for this application.

blog/models.py

The comment model looks like this:

```
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(_("e-mail 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, we first write the comment to the database, then we launch the spam filter task in the background.

blog/views.py

```
from django import forms
frmo django.http import HttpResponseRedirect
from django.template.context import RequestContext
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 we use Akismet, the service used to filter spam in comments posted to the free weblog 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 we use the akismet.py library written by Michael Foord.

blog/tasks.py

```
from akismet import Akismet
from celery.decorators import task

from django.core.exceptions import ImproperlyConfigured
from django.contrib.sites.models import Site

from blog.models import Comment


@task
```

```
def spam_filter(comment_id, remote_addr=None, **kwargs):
        logger = spam_filter.get_logger(**kwargs)
        logger.info("Running spam filter for comment %s" % comment_id)

        comment = Comment.objects.get(pk=comment_id)
        current_domain = Site.objects.get_current().domain
        akismet = Akismet(settings.AKISMET_KEY, "http://%s" % 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
```

2.1.7 How it works

Here comes 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 celery import registry
>>> from celery import task
>>> registry.tasks
{'celery.delete_expired_task_meta':
    <PeriodicTask: celery.delete_expired_task_meta (periodic)>,
 'celery.task.http.HttpDispatchTask':
    <Task: celery.task.http.HttpDispatchTask (regular)>,
 'celery.execute_remote':
    <Task: celery.execute_remote (regular)>,
 'celery.map_async':
    <Task: celery.map_async (regular)>,
 'celery.ping':
    <Task: celery.ping (regular)>}
```

This is the list of tasks built-in to celery. Note that we had to import `celery.task` first for these to show up. This is because the tasks will only be registered when the module they are defined in is imported.

The default loader imports any modules listed in the `CELERY_IMPORTS` setting.

The entity responsible for registering your task in the registry is a meta class, `TaskType`. This is the default meta class for `Task`. If you want to register your task manually you can set the `abstract` attribute:

```
class MyTask(Task):
    abstract = True
```

This way the task won't be registered, but any task subclassing it will.

When tasks are sent, we don't send the function code, just the name of the task. When the worker receives the message it can just look it up in the 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 has yet to be solved.

2.1.8 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.

```
@task(ignore_result=True)
def mytask(...)
    something()
```

Results can even be disabled globally using the `CELERY_IGNORE_RESULT` setting.

Disable rate limits if they're not used

Disabling rate limits altogether is recommended if you don't have any tasks using them. This is because the rate limit subsystem introduces quite a lot of complexity.

Set the `CELERY_DISABLE_RATE_LIMITS` setting to globally disable rate limits:

```
CELERY_DISABLE_RATE_LIMITS = True
```

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:

```
@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)

@task()
def fetch_page(url):
    return myhttplib.get(url)

@task()
def parse_page(url, page):
    return myparser.parse_document(page)

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

Good:

```
@task(ignore_result=True)
def update_page_info(url):
    # fetch_page -> parse_page -> store_page
    fetch_page.delay(url, callback=subtask(parse_page,
                                           callback=subtask(store_page_info)))

@task(ignore_result=True)
def fetch_page(url, callback=None):
    page = myhttplib.get(url)
    if callback:
        # The callback may have been serialized with JSON,
        # so best practice is to convert the subtask dict back
        # into a subtask object.
        subtask(callback).delay(url, page)

@task(ignore_result=True)
def parse_page(url, page, callback=None):
    info = myparser.parse_document(page)
    if callback:
        subtask(callback).delay(url, info)

@task(ignore_result=True)
def store_page_info(url, info):
    PageInfo.objects.create(url, info)
```

We use `subtask` here to safely pass around the callback task. `subtask` is a subclass of dict used to wrap the arguments and execution options for a single task invocation. See *Sets of tasks, Subtasks and Callbacks* for more information about subtasks.

2.1.9 Performance and Strategies

Granularity

The task's granularity is the degree of parallelization your task have. It's better to have many small tasks, than a few long running ones.

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, there's a limit. Sending messages takes processing power and bandwidth. If your tasks are so short the overhead of passing them around is worse than just executing them in-line, you should reconsider your strategy. There is no universal answer here.

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 being 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 caching system, like memcached.

For more information about data-locality, please read http://research.microsoft.com/pubs/70001/tr-2003-24.pdf

State

Since celery is a distributed system, you can't know in which process, or even on what machine the task will run. Indeed you can't even know if the task will run in a timely manner, so please be wary of the state you pass on to tasks.

One gotcha is Django model objects. They shouldn't be passed on as arguments to task classes, it's almost always better to re-fetch the object from the database instead, as there are possible race conditions involved.

Imagine the following scenario where you have an article and a task that automatically expands some abbreviations in it.

```
class Article(models.Model):
    title = models.CharField()
    body = models.TextField()

@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.

```
>>> article = Article.objects.get(id=102)
>>> expand_abbreviations.delay(model_object)
```

Now, the queue is very busy, so the task won't be run for another 2 minutes, in the meantime another author makes some changes to the article, 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:

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

>>> expand_abbreviations(article_id)
```

There might even be performance benefits to this approach, as sending large messages may be expensive.

Database transactions

Let's look at another example:

```
from django.db import transaction

@transaction.commit_on_success
def create_article(request):
    article = Article.objects.create(....)
    expand_abbreviations.delay(article.pk)
```

This is a Django view creating an article object in the database, then passing its primary key to a task. It uses the *commit_on_success* decorator, which will commit the transaction when the view returns, or roll back if the view raises an exception.

There is a race condition if the task starts executing before the transaction has been committed: the database object does not exist yet!

The solution is to **always commit transactions before applying tasks that depends on state from the current transaction**:

```
@transaction.commit_manually
def create_article(request):
    try:
        article = Article.objects.create(...)
    except:
        transaction.rollback()
        raise
    else:
        transaction.commit()
        expand_abbreviations.delay(article.pk)
```

2.2 Executing Tasks

- Basics
- ETA and countdown
- Serializers
- Connections and connection timeouts.
- Routing options
- AMQP options

2.2.1 Basics

Executing tasks is done with `apply_async()`, and its shortcut: `delay()`.

`delay` is simple and convenient, as it looks like calling a regular function:

```
Task.delay(arg1, arg2, kwarg1="x", kwarg2="y")
```

The same thing using `apply_async` is written like this:

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

You can also execute a task by name using `send_task()`, if you don't have access to the task's class:

```
>>> from celery.execute import send_task
>>> result = send_task("tasks.add", [2, 2])
>>> result.get()
4
```

While `delay` is convenient, it doesn't give you as much control as using `apply_async`. With `apply_async` you can override the execution options available as attributes on the `Task` class: `routing_key`, `exchange`, `immediate`, `mandatory`, `priority`, and `serializer`. In addition you can set a countdown/eta, or provide a custom broker connection.

Let's go over these in more detail. The following examples use this simple task, which adds together two numbers:

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

2.2.2 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 execute. `countdown` is a shortcut to set this by seconds in the future.

```
>>> result = add.apply_async(args=[10, 10], countdown=3)
>>> result.get()    # this takes at least 3 seconds to return
20
```

Note that your task is guaranteed to be executed at some time *after* the specified date and time has passed, but not necessarily at that exact time.

While `countdown` is an integer, `eta` must be a `datetime` object, specifying an exact date and time in the future. This is good if you already have a `datetime` object and need to modify it with a `timedelta`, or when using time in seconds is not very readable.

```
from datetime import datetime, timedelta

def quickban(username):
    """Ban user for 24 hours."""
    ban(username)
    tomorrow = datetime.now() + timedelta(days=1)
    UnbanTask.apply_async(args=[username], eta=tomorrow)
```

2.2.3 Serializers

Data passed between celery and workers has to be serialized to be transferred. The default serializer is `pickle`, but you can change this for each task. There is built-in support for using `pickle`, `JSON` and `YAML`, and you can add your own custom serializers by registering them into the carrot serializer registry.

The default serializer (pickle) supports Python objects, like `datetime` and any custom datatypes you define yourself. But since pickle has poor support outside of the Python language, you need to choose another serializer if you need to communicate with other languages. In that case, `JSON` is a very popular choice.

The serialization method is sent with the message, so the worker knows how to deserialize any task. Of course, if you use a custom serializer, this must also be registered in the worker.

When sending a task the serialization method is taken from the following places in order: The `serializer` argument to `apply_async`, the Task's `serializer` attribute, and finally the global default `CELERY_SERIALIZER` configuration directive.

```
>>> add.apply_async(args=[10, 10], serializer="json")
```

2.2.4 Connections and connection timeouts.

Currently there is no support for broker connection pools in celery, so this is something you need to be aware of when sending more than one task at a time, as `apply_async/delay` establishes and closes a connection every time.

If you need to send more than one task at the same time, it's a good idea to establish the connection yourself and pass it to `apply_async`:

```
numbers = [(2, 2), (4, 4), (8, 8), (16, 16)]

results = []
publisher = add.get_publisher()
try:
    for args in numbers:
```

```
        res = add.apply_async(args=args, publisher=publisher)
        results.append(res)
finally:
    publisher.close()
    publisher.connection.close()

print([res.get() for res in results])
```

The connection timeout is the number of seconds to wait before we give up establishing the connection. You can set this with the `connect_timeout` argument to `apply_async`:

```
add.apply_async([10, 10], connect_timeout=3)
```

Or if you handle the connection manually:

```
publisher = add.get_publisher(connect_timeout=3)
```

2.2.5 Routing options

Celery uses the AMQP routing mechanisms to route tasks to different workers. You can route tasks using the following entities: exchange, queue and routing key.

Messages (tasks) are sent to exchanges, a queue binds to an exchange with a routing key. Let's look at an example:

Our application has a lot of tasks, some process video, others process images, and some gather collective intelligence about users. Some of these have higher priority than others so we want to make sure the high priority tasks get sent to powerful machines, while low priority tasks are sent to dedicated machines that can handle these at their own pace.

For the sake of example we have only one exchange called `tasks`. There are different types of exchanges that matches the routing key in different ways, the exchange types are:

- direct

  Matches the routing key exactly.

- topic

  In the topic exchange the routing key is made up of words separated by dots (`.`). Words can be matched by the wild cards `*` and `#`, where `*` matches one exact word, and `#` matches one or many.

  For example, `*.stock.#` matches the routing keys `usd.stock` and `euro.stock.db` but not `stock.nasdaq`.

(there are also other exchange types, but these are not used by celery)

So, we create three queues, `video`, `image` and `lowpri` that bind to our `tasks` exchange. For the queues we use the following binding keys:

```
video: video.#
image: image.#
lowpri: misc.#
```

Now we can send our tasks to different worker machines, by making the workers listen to different queues:

```
>>> CompressVideoTask.apply_async(args=[filename],
...                               routing_key="video.compress")

>>> ImageRotateTask.apply_async(args=[filename, 360],
...                             routing_key="image.rotate")

>>> ImageCropTask.apply_async(args=[filename, selection],
```

```
...                               routing_key="image.crop")
>>> UpdateReccomendationsTask.apply_async(routing_key="misc.recommend")
```

Later, if the crop task is consuming a lot of resources, we can bind some new workers to handle just the "image.crop" task, by creating a new queue that binds to "image.crop".

2.2.6 AMQP options

NOTE The mandatory and immediate flags are not supported by amqplib at this point.

- mandatory

This sets the delivery to be mandatory. An exception will be raised if there are no running workers able to take on the task.

- immediate

Request immediate delivery. Will raise an exception if the task cannot be routed to a worker immediately.

- priority

A number between 0 and 9, where 0 is the highest priority. Note that RabbitMQ does not implement AMQP priorities, and maybe your broker does not either, consult your broker's documentation for more information.

2.3 Workers Guide

2.3.1 Starting the worker

You can start celeryd to run in the foreground by executing the command:

```
$ celeryd --loglevel=INFO
```

You probably want to use a daemonization tool to start `celeryd` in the background. See *Running celeryd as a daemon* for help starting celeryd with some of the most popular daemonization tools.

For a full list of available command line options see `celeryd`, or simply execute the command:

```
$ celeryd --help
```

You can also start multiple celeryd's on the same machine. If you do so be sure to give a unique name to each individual worker by specifying a hostname with the `--hostname|-n` argument:

```
$ celeryd --loglevel=INFO --concurrency=10 -n worker1.example.com
$ celeryd --loglevel=INFO --concurrency=10 -n worker2.example.com
$ celeryd --loglevel=INFO --concurrency=10 -n worker3.example.com
```

2.3.2 Stopping the worker

Shutdown should be accomplished using the `TERM` signal.

When shutdown is initiated the worker will finish any tasks it's currently executing before it terminates, so 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 example because of tasks stuck in an infinite-loop, you can use the `KILL` signal to force terminate the worker, but be aware that currently executing tasks will be lost (unless the tasks have the `acks_late` option set).

Also, since the `KILL` signal can't be catched by processes the worker will not be able to reap its children so make sure you do it manually. This command usually does the trick:

```
$ ps auxww | grep celeryd | awk '{print $2}' | xargs kill -9
```

2.3.3 Restarting the worker

Other than stopping then starting the worker to restart, you can also restart the worker using the `HUP` signal:

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

The worker will then replace itself with a new instance using the same arguments as it was started with.

2.3.4 Concurrency

Multiprocessing is used to perform concurrent execution of tasks. The number of worker processes can be changed using the `--concurrency` argument and defaults to the number of CPUs available.

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

2.3.5 Time limits

A single task can potentially run forever, if you have lots of tasks waiting for some event that will never happen you will 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:

```
from celery.decorators import task
from celery.exceptions import SoftTimeLimitExceeded

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

Time limits can also be set using the `CELERYD_TASK_TIME_LIMIT` / `CELERYD_SOFT_TASK_TIME_LIMIT` settings.

NOTE Time limits does not currently work on Windows.

2.3.6 Max tasks per child setting

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 `--maxtasksperchild` argument to `celeryd` or using the `CELERYD_MAX_TASKS_PER_CHILD` setting.

2.3.7 Remote control

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, but since there's no central authority to know how many workers are available in the cluster, there is also no way to estimate how many workers may send a reply. Therefore 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.

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:

```
>>> from celery.task.control import broadcast
>>> 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:

```
>>> 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:

```
>>> broadcast
>>> 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()`.

Rate limits

Example changing the rate limit for the `myapp.mytask` task to accept 200 tasks a minute on all servers:

```
>>> from celery.task.control import rate_limit
>>> rate_limit("myapp.mytask", "200/m")
```

Example changing the rate limit on a single host by specifying the destination hostname:

```
>>> rate_limit("myapp.mytask", "200/m",
...            destination=["worker1.example.com"])
```

NOTE This won't affect workers with the `CELERY_DISABLE_RATE_LIMITS` setting on. To re-enable rate limits then you have to restart the worker.

Remote shutdown

This command will gracefully shut down the worker remotely:

```
>>> broadcast("shutdown") # shutdown all workers
>>> 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:

```
>>> from celery.task.control import ping
>>> 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 which 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 celeryev/celerymon.

```
>>> broadcast("enable_events")
>>> broadcast("disable_events")
```

Writing your own remote control commands

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 `celery.worker.listener.CarrotListener` if needed.

Here's an example control command that restarts the broker connection:

```
from celery.worker.control import Panel

@Panel.register
def reset_connection(panel):
    panel.logger.critical("Connection reset by remote control.")
    panel.listener.reset_connection()
    return {"ok": "connection reset"}
```

These can be added to task modules, or you can keep them in their own module then import them using the `CELERY_IMPORTS` setting:

```
CELERY_IMPORTS = ("myapp.worker.control", )
```

2.3.8 Inspecting workers

`celery.task.control.inspect` lets you inspect running workers. It uses remote control commands under the hood.

```
>>> from celery.task.control import inspect

# Inspect all nodes.
>>> i = inspect()

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

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

Dump of registered tasks

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

```
>>> i.registered_tasks()
[{'worker1.example.com': ['celery.delete_expired_task_meta',
                          'celery.execute_remote',
                          'celery.map_async',
                          'celery.ping',
                          'celery.task.http.HttpDispatchTask',
                          '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 that these are tasks with an eta/countdown argument, not periodic tasks.

Dump of reserved tasks

Reserved tasks are tasks that has been received, but is 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": "{}"}]}]
```

2.4 Sets of tasks, Subtasks and Callbacks

2.4.1 Subtasks

The `subtask` class is used to wrap the arguments and execution options for a single task invocation:

```
subtask(task_name_or_cls, args, kwargs, options)
```

For convenience every task also has a shortcut to create subtask instances:

```
task.subtask(args, kwargs, options)
```

`subtask` is actually a subclass of `dict`, which means it can be serialized with JSON or other encodings that doesn't support complex Python objects.

Also it can be regarded as a type, as the following usage works:

```
>>> s = subtask("tasks.add", args=(2, 2), kwargs={})

>>> subtask(dict(s))  # coerce dict into subtask
```

This makes it excellent as a means to pass callbacks around to tasks.

Callbacks

Let's improve our `add` task so it can accept a callback that takes the result as an argument:

```
from celery.decorators import task
from celery.task.sets import subtask

@task
def add(x, y, callback=None):
    result = x + y
    if callback is not None:
        subtask(callback).delay(result)
    return result
```

See? `subtask` also knows how it should be applied, asynchronously by `delay()`, and eagerly by `apply()`.

The best thing is that any arguments you add to `subtask.delay`, will be prepended to the arguments specified by the subtask itself!

So if you have the subtask:

```
>>> add.subtask(args=(10, ))
```

`subtask.delay(result)` becomes:

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

Now let's execute our new `add` task with a callback:

```
>>> add.delay(2, 2, callback=add.subtask((8, )))
```

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

2.4.2 Task Sets

The `TaskSet` enables easy invocation of several tasks at once, and is then able to join the results in the same order as the tasks were invoked.

A task set takes a list of `subtask`'s:

```
>>> from celery.task.sets import TaskSet
>>> from tasks import add

>>> job = TaskSet(tasks=[
...             add.subtask((4, 4)),
...             add.subtask((8, 8)),
...             add.subtask((16, 16)),
...             add.subtask((32, 32)),
... ])

>>> result = job.apply_async()

>>> result.ready()  # has all subtasks completed?
True
>>> result.successful() # was all subtasks successful?

>>> result.join()
[4, 8, 16, 32, 64]
```

Results

When a `TaskSet` is applied it returns a `TaskSetResult` object.

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

It supports the following operations:

- `successful()`

  Returns `True` if all of the subtasks finished successfully (e.g. did not raise an exception).

- `failed()`

  Returns `True` if any of the subtasks failed.

- `waiting()`

  Returns `True` if any of the subtasks is not ready yet.

- `ready()`

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

- `completed_count()`

  Returns the number of completed subtasks.

- `revoke()`

  Revokes all of the subtasks.

- `iterate()`

  Iterates over the return values of the subtasks as they finish, one by one.

- `join()`

  Gather the results for all of the subtasks and return a list with them ordered by the order of which they were called.

2.5 HTTP Callback Tasks (Webhooks)

2.5.1 Basics

If you need to call into another language, framework or similar, you can do so by using HTTP callback tasks.

The HTTP callback tasks use GET/POST arguments and a simple JSON response to return results. The scheme to call a task is:

```
GET http://example.com/mytask/?arg1=a&arg2=b&arg3=c
```

or using POST:

```
POST http://example.com/mytask
```

Note: POST data has to be form encoded. Whether to use GET or POST is up to you and your requirements.

The web page should then return a response in the following format if the execution was successful:

```
{"status": "success", "retval": ....}
```

or if there was an error:

```
{"status": "failure": "reason": "Invalid moon alignment."}
```

2.5.2 Django webhook example

With this information you could define a simple task in Django:

```
from django.http import HttpResponse
from anyjson import serialize


def multiply(request):
    x = int(request.GET["x"])
    y = int(request.GET["y"])
    result = x * y
    response = {"status": "success", "retval": result}
    return HttpResponse(serialize(response), mimetype="application/json")
```

2.5.3 Ruby on Rails webhook example

or in Ruby on Rails:

```
def multiply
    @x = params[:x].to_i
    @y = params[:y].to_i

    @status = {:status => "success", :retval => @x * @y}

    render :json => @status
end
```

You can easily port this scheme to any language/framework; new examples and libraries are very welcome.

2.5.4 Executing webhook tasks

To execute the task you use the `URL` class:

```
>>> from celery.task.http import URL
>>> res = URL("http://example.com/multiply").get_async(x=10, y=10)
```

`URL` is a shortcut to the `HttpDispatchTask`. You can subclass this to extend the functionality.

```
>>> from celery.task.http import HttpDispatchTask
>>> res = HttpDispatchTask.delay(url="http://example.com/multiply", method="GET", x=10, y=10)
>>> res.get()
100
```

The output of celeryd (or the logfile if you've enabled it) should show the task being processed:

```
[INFO/MainProcess] Task celery.task.http.HttpDispatchTask
        [f2cc8efc-2a14-40cd-85ad-f1c77c94beeb] processed: 100
```

Since applying tasks can be done via HTTP using the `celery.views.apply` view, executing tasks from other languages is easy. For an example service exposing tasks via HTTP you should have a look at `examples/celery_http_gateway`.

2.6 Routing Tasks

NOTE This document refers to functionality only available in brokers using AMQP. Other brokers may implement some functionality, see their respective documenation for more information, or contact the mailinglist.

2.6.1 Basics

Automatic routing

The simplest way to do routing is to use the `CELERY_CREATE_MISSING_QUEUES` setting (on by default).

With this setting on, a named queue that is not already defined in `CELERY_QUEUES` will be created automatically. This makes it easy to perform simple routing tasks.

Say you have two servers, `x`, and `y` that handles regular tasks, and one server `z`, that only handles feed related tasks. You can use this configuration:

```
CELERY_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 historic reasons).

Now you can start server `z` to only process the feeds queue like this:

```
(z)$ celeryd -Q feeds
```

You can specify as many queues as you want, so you can make this server process the default queue as well:

```
(z)$ celeryd -Q feeds,celery
```

Changing the name of the default queue

You can change the name of the default queue by using the following configuration:

```
CELERY_QUEUES = {"default": {"exchange": "default",
                             "binding_key": "default"}}
CELERY_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 defined.

A queue named `"video"` will be created with the following settings:

```
{"exchange": "video",
 "exchange_type": "direct",
 "routing_key": "video"}
```

The non-AMQP backends like `ghettoq` does not 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 handles regular tasks, and one server `z`, that only handles feed related tasks, you can use this configuration:

```
CELERY_DEFAULT_QUEUE = "default"
CELERY_QUEUES = {
    "default": {
        "binding_key": "task.#",
    },
    "feed_tasks": {
        "binding_key": "feed.#",
    },
}
CELERY_DEFAULT_EXCHANGE = "tasks"
CELERY_DEFAULT_EXCHANGE_TYPE = "topic"
CELERY_DEFAULT_ROUTING_KEY = "task.default"
```

`CELERY_QUEUES` is a map of queue names and their exchange/type/binding_key, if you don't set exchange or exchange type, they will be taken from the `CELERY_DEFAULT_EXCHANGE`/`CELERY_DEFAULT_EXCHANGE_TYPE` settings.

To route a task to the `feed_tasks` queue, you can add an entry in the `CELERY_ROUTES` setting:

```
CELERY_ROUTES = {
        "feeds.tasks.import_feed": {
            "queue": "feed_tasks",
            "routing_key": "feed.import",
        },
}
```

You can also override this using the `routing_key` argument to `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 `-Q` option:

```
(z)$ celeryd -Q feed_tasks --hostname=z.example.com
```

Servers `x` and `y` must be configured to consume from the default queue:

```
(x)$ celeryd -Q default --hostname=x.example.com
(y)$ celeryd -Q default --hostname=y.example.com
```

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:

```
(z)$ celeryd -Q feed_tasks,default --hostname=z.example.com
```

If you have another queue but on another exchange you want to add, just specify a custom exchange and exchange type:

```
CELERY_QUEUES = {
        "feed_tasks": {
            "binding_key": "feed.#",
        },
        "regular_tasks": {
            "binding_key": "task.#",
        },
        "image_tasks": {
            "binding_key": "image.compress",
            "exchange": "mediatasks",
            "exchange_type": "direct",
        },
    }
```

If you're confused about these terms, you should read up on AMQP concepts.

In addition to the *AMQP Primer* below, there's Rabbits and Warrens, an excellent blog post describing queues and exchanges. There's also AMQP in 10 minutes*: Flexible Routing Model, and Standard Exchange Types. For users of RabbitMQ the RabbitMQ FAQ could be useful as a source of information.

2.6.2 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. In Celery the content type is usually the serialization format used to serialize the message, and the body contains the name of the task to execute, the task id (UUID), the arguments to execute it with and some additional metadata - like the number of retries and its ETA (if any).

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 are 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.
3. The message waits in the queue until someone consumes from 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 `CELERY_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 finally, one default queue for everything else:

```
CELERY_QUEUES = {
    "default": {
        "exchange": "default",
        "binding_key": "default"},
    "videos": {
        "exchange": "media",
        "binding_key": "media.video",
    },
    "images": {
        "exchange": "media",
        "binding_key": "media.image",
    }
}
CELERY_DEFAULT_QUEUE = "default"
CELERY_DEFAULT_EXCHANGE_TYPE = "direct"
CELERY_DEFAULT_ROUTING_KEY = "default"
```

NOTE: In Celery the `routing_key` is the key used to send the message, while `binding_key` is the key the queue is bound with. In the AMQP API they are both referred to as the routing key.

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 plugins 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 with the routing key `video` only receives messages with the same routing key.

Topic exchanges

Topic exchanges matches routing keys using dot-separated words, and can include wildcard characters: `*` matches a single word, # 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.

- `passive` means the exchange won't be created, but you can use this to check if the exchange already exists.
- Durable exchanges are persistent. That is - they survive a broker restart.
- `auto_delete` means the queue 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.

- exclusive queues can only be consumed from by the current connection. implies `auto_delete`.

queue.bind(queue_name, exchange_name, routing_key)

Binds a queue to an exchange with a routing key. Unbound queues will not receive messages, so this is necessary.

queue.delete(name, if_unused, if_empty)

Deletes a queue and its binding.

exchange.delete(name, if_unused)

Deletes an exchange.

NOTE: Declaring does not necessarily mean "create". When you declare you *assert* that the entity exists and that it's operable. There is 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 `camqadm` (short for celery AMQP admin). It's used for simple admnistration tasks like creating/deleting queues and exchanges, purging queues and sending messages. In short it's for simple command-line access to the AMQP API.

You can write commands directly in the arguments to `camqadm`, or just start with no arguments to start it in shell-mode:

```
$ camqadm
-> connecting to amqp://guest@localhost:5672/.
-> connected.
1>
```

Here `1>` is the prompt. The number is counting the number of commands you have executed. Type `help` for a list of commands. It also has autocompletion, so you can start typing a command and then hit the `tab` key to show a list of possible matches.

Now let's create a queue we can send messages to:

```
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. We 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 we can retrieve it again. We use the `basic.get` command here, which pops a single message off the queue, this command is not recommended for production as it implies polling, any real application would declare consumers 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. The message is sent to the next receiver if it has not been acknowledged before the client connection is closed.

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. Note that delivery tags are not 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 we received using `basic.ack`:

```
6> basic.ack 1
ok.
```

To clean up after our test session we should delete the entities we created:

```
7> queue.delete testqueue
ok. 0 messages deleted.
8> exchange.delete testexchange
ok.
```

2.6.3 Routing Tasks

Defining queues

In Celery the queues are defined by the `CELERY_QUEUES` setting.

Here's an example queue configuration with three queues; One for video, one for images and finally, one default queue for everything else:

```
CELERY_QUEUES = {
    "default": {
        "exchange": "default",
        "binding_key": "default"},
    "videos": {
        "exchange": "media",
        "exchange_type": "topic",
        "binding_key": "media.video",
    },
    "images": {
        "exchange": "media",
        "exchange_type": "topic",
        "binding_key": "media.image",
    }
}
CELERY_DEFAULT_QUEUE = "default"
CELERY_DEFAULT_EXCHANGE = "default"
CELERY_DEFAULT_EXCHANGE_TYPE = "direct"
CELERY_DEFAULT_ROUTING_KEY = "default"
```

Here, the `CELERY_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 `CELERY_QUEUES`.

Specifying task destination

The destination for a task is decided by the following (in order):

1. The *Routers* defined in `CELERY_ROUTES`.
2. The routing arguments to `apply_async()`.
3. Routing related attributes defined on the `Task` itself.

It is 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 class that decides the routing options for a task.

All you need to define a new router is to create a class with a `route_for_task` method:

```
class MyRouter(object):

    def route_for_task(self, task, args=None, kwargs=None):
        if task == "myapp.tasks.compress_video":
            return {"exchange": "video",
                    "exchange_type": "topic",
```

```
                    "routing_key": "video.compress"}
        return None
```

If you return the `queue` key, it will expand with the defined settings of that queue in `CELERY_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 it to the `CELERY_ROUTES` setting:

```
CELERY_ROUTES = (MyRouter, )
```

Router classes can also be added by name:

```
CELERY_ROUTES = ("myapp.routers.MyRouter", )
```

For simple task name -> route mappings like the router example above, you can simply drop a dict into `CELERY_ROUTES` to get the same result:

```
CELERY_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 value and use that as the final route for the task.

CHAPTER 3

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 is available on the Python path.

3.1 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.

```
# List of modules to import when celery starts.
CELERY_IMPORTS = ("myapp.tasks", )

## Result store settings.
CELERY_RESULT_BACKEND = "database"
CELERY_RESULT_DBURI = "sqlite:///mydatabase.db"

## Broker settings.
BROKER_HOST = "localhost"
BROKER_PORT = 5672
BROKER_VHOST = "/"
BROKER_USER = "guest"
BROKER_PASSWORD = "guest"

## Worker settings
## If you're doing mostly I/O you can have more processes,
## but if mostly spending CPU, try to keep it close to the
## number of CPUs on your machine. If not set, the number of CPUs/cores
## available will be used.
CELERYD_CONCURRENCY = 10
# CELERYD_LOG_FILE = "celeryd.log"
# CELERYD_LOG_LEVEL = "INFO"
```

3.2 Concurrency settings

- CELERYD_CONCURRENCY

  The number of concurrent worker processes, executing tasks simultaneously.

  Defaults to the number of CPUs/cores available.

- CELERYD_PREFETCH_MULTIPLIER

  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 seems pretty good here. 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 balanced among the workers.

3.3 Task result backend settings

- CELERY_RESULT_BACKEND

  The backend used to store task results (tombstones). Can be one of the following:

  - **database (default)** Use a relational database supported by SQLAlchemy.
  - **cache** Use memcached to store the results.
  - **mongodb** Use MongoDB to store the results.
  - **redis** Use Redis to store the results.
  - **tyrant** Use Tokyo Tyrant to store the results.
  - **amqp** Send results back as AMQP messages (**WARNING** While very fast, you must make sure you only receive the result once. See *Executing Tasks*).

3.4 Database backend settings

Please see Supported Databases for a table of supported databases. To use this backend you need to configure it with an 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 Connection String 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}
```

3.4.1 Example configuration

```
CELERY_RESULT_BACKEND = "database"
CELERY_RESULT_DBURI = "mysql://user:password@host/dbname"
```

3.5 AMQP backend settings

- CELERY_RESULT_EXCHANGE

  Name of the exchange to publish results in. Default is `"celeryresults"`.

- CELERY_RESULT_EXCHANGE_TYPE

  The exchange type of the result exchange. Default is to use a `direct` exchange.

- CELERY_RESULT_SERIALIZER

  Result message serialization format. Default is `"pickle"`.

- CELERY_RESULTS_PERSISTENT

  If set to `True`, result messages will be persistent. This means the messages will not be lost after a broker restart. The default is for the results to be transient.

3.5.1 Example configuration

CELERY_RESULT_BACKEND = “amqp”

3.6 Cache backend settings

The cache backend supports the pylibmc and *python-memcached* libraries. The latter is used only if pylibmc is not installed.

3.6.1 Example configuration

Using a single memcached server:

```
CELERY_CACHE_BACKEND = 'memcached://127.0.0.1:11211/'
```

Using multiple memcached servers:

```
CELERY_RESULT_BACKEND = "cache"
CELERY_CACHE_BACKEND = 'memcached://172.19.26.240:11211;172.19.26.242:11211/'
```

You can set pylibmc options using the `CELERY_CACHE_BACKEND_OPTIONS` setting:

```
CELERY_CACHE_BACKEND_OPTIONS = {"binary": True,
                                "behaviors": {"tcp_nodelay": True}}
```

3.7 Tokyo Tyrant backend settings

NOTE The Tokyo Tyrant backend requires the `pytyrant` library: http://pypi.python.org/pypi/pytyrant/

This backend requires the following configuration directives to be set:

- **TT_HOST** Hostname of the Tokyo Tyrant server.
- **TT_PORT** The port the Tokyo Tyrant server is listening to.

3.7.1 Example configuration

```
CELERY_RESULT_BACKEND = "tyrant"
TT_HOST = "localhost"
TT_PORT = 1978
```

3.8 Redis backend settings

NOTE The Redis backend requires the `redis` library: http://pypi.python.org/pypi/redis/0.5.5

To install the redis package use `pip` or `easy_install`:

```
$ pip install redis
```

This backend requires the following configuration directives to be set:

- REDIS_HOST

  Hostname of the Redis database server. e.g. `"localhost"`.

- REDIS_PORT

  Port to the Redis database server. e.g. `6379`.

Also, the following optional configuration directives are available:

- REDIS_DB

  Database number to use. Default is 0

- REDIS_PASSWORD

  Password used to connect to the database.

3.8.1 Example configuration

```
CELERY_RESULT_BACKEND = "redis"
REDIS_HOST = "localhost"
REDIS_PORT = 6379
REDIS_DB = "celery_results"
REDIS_CONNECT_RETRY=True
```

3.9 MongoDB backend settings

NOTE The MongoDB backend requires the `pymongo` library: http://github.com/mongodb/mongo-python-driver/tree/master

- CELERY_MONGODB_BACKEND_SETTINGS

  This is a dict supporting the following keys:

  - **host** Hostname of the MongoDB server. Defaults to "localhost".
  - **port** The port the MongoDB server is listening to. Defaults to 27017.
  - **user** User name to authenticate to the MongoDB server as (optional).
  - **password** Password to authenticate to the MongoDB server (optional).
  - **database** The database name to connect to. Defaults to "celery".
  - **taskmeta_collection** The collection name to store task meta data. Defaults to "celery_taskmeta".

3.9.1 Example configuration

```
CELERY_RESULT_BACKEND = "mongodb"
CELERY_MONGODB_BACKEND_SETTINGS = {
    "host": "192.168.1.100",
    "port": 30000,
    "database": "mydb",
    "taskmeta_collection": "my_taskmeta_collection",
}
```

3.10 Messaging settings

3.10.1 Routing

- **CELERY_QUEUES** The mapping of queues the worker consumes from. This is a dictionary of queue name/options. See *Routing Tasks* for more information.

  The default is a queue/exchange/binding key of `"celery"`, with exchange type `direct`.

  You don't have to care about this unless you want custom routing facilities.

- **CELERY_DEFAULT_QUEUE** The queue used by default, if no custom queue is specified. This queue must be listed in `CELERY_QUEUES`. The default is: `celery`.

- **CELERY_DEFAULT_EXCHANGE** Name of the default exchange to use when no custom exchange is specified. The default is: `celery`.

- **CELERY_DEFAULT_EXCHANGE_TYPE** Default exchange type used when no custom exchange is specified. The default is: `direct`.

- **CELERY_DEFAULT_ROUTING_KEY** The default routing key used when sending tasks. The default is: `celery`.

- CELERY_DEFAULT_DELIVERY_MODE

  Can be `transient` or `persistent`. Default is to send persistent messages.

3.10.2 Connection

- **CELERY_BROKER_CONNECTION_TIMEOUT** The timeout in seconds before we give up establishing a connection to the AMQP server. Default is 4 seconds.

- **CELERY_BROKER_CONNECTION_RETRY** Automatically try to re-establish the connection to the AMQP broker if it's lost.

  The time between retries is increased for each retry, and is not exhausted before `CELERY_BROKER_CONNECTION_MAX_RETRIES` is exceeded.

  This behavior is on by default.

- **CELERY_BROKER_CONNECTION_MAX_RETRIES** 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 will retry forever.

  Default is 100 retries.

3.11 Task execution settings

- **CELERY_ALWAYS_EAGER** If this is `True`, all tasks will be executed locally by blocking until it is finished. `apply_async` and `Task.delay` will return a `celery.result.EagerResult` which emulates the behavior of `celery.result.AsyncResult`, except the result has already been evaluated.

  Tasks will never be sent to the queue, but executed locally instead.

- CELERY_EAGER_PROPAGATES_EXCEPTIONS

If this is `True`, eagerly executed tasks (using `.apply`, or with `CELERY_ALWAYS_EAGER` on), will raise exceptions.

It's the same as always running `apply` with `throw=True`.

- CELERY_IGNORE_RESULT

  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 `CELERY_STORE_ERRORS_EVEN_IF_IGNORED`.

- **CELERY_TASK_RESULT_EXPIRES** Time (in seconds, or a `datetime.timedelta` object) for when after stored task tombstones will be deleted.

  A built-in periodic task will delete the results after this time (`celery.task.builtins.DeleteExpiredTaskMetaTask`).

  NOTE: For the moment this only works with the database, cache and MongoDB backends.

  NOTE: `celerybeat` must be running for the results to be expired.

- CELERY_MAX_CACHED_RESULTS

  Total number of results to store before results are evicted from the result cache. The default is `5000`.

- CELERY_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 behaviour 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 is a need to report which task is currently running. backends.

- **CELERY_TASK_SERIALIZER** A string identifying the default serialization method to use. Can be `pickle` (default), `json`, `yaml`, or any custom serialization methods that have been registered with `carrot.serialization.registry`.

  Default is `pickle`.

- CELERY_DEFAULT_RATE_LIMIT

  The global default rate limit for tasks.

  This value is used for tasks that does not have a custom rate limit The default is no rate limit.

- CELERY_DISABLE_RATE_LIMITS

  Disable all rate limits, even if tasks has explicit rate limits set.

- 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.

  See http://ask.github.com/celery/faq.html#should-i-use-retry-or-acks-late

3.12 Worker: celeryd

- CELERY_IMPORTS

  A sequence of modules to import when the celery daemon starts.

  This is used to specify the task modules to import, but also to import signal handlers and additional remote control commands, etc.

- CELERYD_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.

- CELERYD_TASK_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.

- CELERYD_SOFT_TASK_TIME_LIMIT

  Task soft time limit in seconds. The `celery.exceptions.SoftTimeLimitExceeded` exception will be raised when this is exceeded. The task can catch this to e.g. clean up before the hard time limit comes.

  ```
  from celery.decorators import task
  from celery.exceptions import SoftTimeLimitExceeded

  @task()
  def mytask():
      try:
          return do_work()
      except SoftTimeLimitExceeded:
          cleanup_in_a_hurry()
  ```

- CELERY_STORE_ERRORS_EVEN_IF_IGNORED

  If set, the worker stores all task errors in the result store even if `Task.ignore_result` is on.

3.12.1 Error E-Mails

- CELERY_SEND_TASK_ERROR_EMAILS

  If set to `True`, errors in tasks will be sent to admins by e-mail.

- ADMINS

  List of `(name, email_address)` tuples for the admins that should receive error e-mails.

- SERVER_EMAIL

  The e-mail address this worker sends e-mails from. Default is `"celery@localhost"`.

- MAIL_HOST

  The mail server to use. Default is `"localhost"`.

- MAIL_HOST_USER

  Username (if required) to log on to the mail server with.

- MAIL_HOST_PASSWORD

  Password (if required) to log on to the mail server with.

- MAIL_PORT

  The port the mail server is listening on. Default is `25`.

Example E-Mail configuration

This configuration enables the sending of error e-mails to `george@vandelay.com` and `kramer@vandelay.com`:

```
# Enables error e-mails.
CELERY_SEND_TASK_ERROR_EMAILS = True

# Name and e-mail addresses of recipients
ADMINS = (
    ("George Costanza", "george@vandelay.com"),
    ("Cosmo Kramer", "kosmo@vandelay.com"),
)

# E-mail address used as sender (From field).
SERVER_EMAIL = "no-reply@vandelay.com"

# Mailserver configuration
EMAIL_HOST = "mail.vandelay.com"
EMAIL_PORT = 25
# EMAIL_HOST_USER = "servers"
# EMAIL_HOST_PASSWORD = "s3cr3t"
```

3.12.2 Events

- CELERY_SEND_EVENTS

  Send events so the worker can be monitored by tools like `celerymon`.

- CELERY_EVENT_EXCHANGE

  Name of the exchange to send event messages to. Default is `"celeryevent"`.

- CELERY_EVENT_EXCHANGE_TYPE

  The exchange type of the event exchange. Default is to use a `direct` exchange.

- CELERY_EVENT_ROUTING_KEY

  Routing key used when sending event messages. Default is `"celeryevent"`.

- CELERY_EVENT_SERIALIZER

  Message serialization format used when sending event messages. Default is `"json"`.

3.12.3 Broadcast Commands

- CELERY_BROADCAST_QUEUE

  Name prefix for the queue used when listening for broadcast messages. The workers hostname will be appended to the prefix to create the final queue name.

  Default is `"celeryctl"`.

- CELERY_BROADCAST_EXCHANGE

  Name of the exchange used for broadcast messages.

  Default is `"celeryctl"`.

- CELERY_BROADCAST_EXCHANGE_TYPE

  Exchange type used for broadcast messages. Default is `"fanout"`.

3.12.4 Logging

- **CELERYD_LOG_FILE** The default file name the worker daemon logs messages to, can be overridden using the *–logfile‘* option to `celeryd`.

  The default is `None` (`stderr`) Can also be set via the `--logfile` argument.

- CELERYD_LOG_LEVEL

  Worker log level, can be any of `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`.

  Can also be set via the `--loglevel` argument.

  See the `logging` module for more information.

- CELERYD_LOG_FORMAT

  The format to use for log messages.

  Default is `[%(asctime)s: %(levelname)s/%(processName)s] %(message)s`

  See the Python `logging` module for more information about log formats.

- CELERYD_TASK_LOG_FORMAT

  The format to use for log messages logged in tasks. Can be overridden using the `--loglevel` option to `celeryd`.

  Default is:

```
[%(asctime)s: %(levelname)s/%(processName)s]
    [%(task_name)s(%(task_id)s)] %(message)s
```

  See the Python `logging` module for more information about log formats.

3.12.5 Custom Component Classes (advanced)

- CELERYD_POOL

  Name of the task pool class used by the worker. Default is `"celery.concurrency.processes.TaskPool"`.

- CELERYD_LISTENER

  Name of the listener class used by the worker. Default is `"celery.worker.listener.CarrotListener"`.

- CELERYD_MEDIATOR

  Name of the mediator class used by the worker. Default is `"celery.worker.controllers.Mediator"`.

- CELERYD_ETA_SCHEDULER

  Name of the ETA scheduler class used by the worker. Default is `"celery.worker.controllers.ScheduleController"`.

3.13 Periodic Task Server: celerybeat

- CELERYBEAT_SCHEDULE_FILENAME

Name of the file celerybeat stores the current schedule in. Can be a relative or absolute path, but be aware that the suffix `.db` will be appended to the file name.

Can also be set via the `--schedule` argument.

- CELERYBEAT_MAX_LOOP_INTERVAL

  The maximum number of seconds celerybeat can sleep between checking the schedule. Default is 300 seconds (5 minutes).

- **CELERYBEAT_LOG_FILE** The default file name to log messages to, can be overridden using the *–logfile‘* option.

  The default is `None` (`stderr`). Can also be set via the `--logfile` argument.

- **CELERYBEAT_LOG_LEVEL** Logging level. Can be any of `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `CRITICAL`.

  Can also be set via the `--loglevel` argument.

  See the `logging` module for more information.

3.14 Monitor Server: celerymon

- **CELERYMON_LOG_FILE** The default file name to log messages to, can be overridden using the *–logfile‘* option.

  The default is `None` (`stderr`) Can also be set via the `--logfile` argument.

- **CELERYMON_LOG_LEVEL** Logging level. Can be any of `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `CRITICAL`.

  See the `logging` module for more information.

CHAPTER 4

Cookbook

4.1 Creating Tasks

- Ensuring a task is only executed one at a time

4.1.1 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 is 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 md5sum of the feed URL.

The cache key expires after some time in case something unexpected happens (you never know, right?)

```
from celery.task import Task
from django.core.cache import cache
from django.utils.hashcompat import md5_constructor as md5
from djangofeeds.models import Feed

LOCK_EXPIRE = 60 * 5 # Lock expires in 5 minutes

class FeedImporter(Task):
    name = "feed.import"

    def run(self, feed_url, **kwargs):
        logger = self.get_logger(**kwargs)

        # The cache key consists of the task name and the MD5 digest
        # of the feed URL.
        feed_url_digest = md5(feed_url).hexdigest()
        lock_id = "%s-lock-%s" % (self.name, feed_url_hexdigest)

        # cache.add fails if if the key already exists
        acquire_lock = lambda: cache.add(lock_id, "true", LOCK_EXPIRE)
        # memcache delete is very slow, but we have to use it to take
        # advantage of using add() for atomic locking
        release_lock = lambda: cache.delete(lock_id)
```

```
        logger.debug("Importing feed: %s" % feed_url)
        if aquire_lock():
            try:
                feed = Feed.objects.import_feed(feed_url)
            finally:
                release_lock()
            return feed.url

        logger.debug(
            "Feed %s is already being imported by another worker" % (
                feed_url))
        return
```

4.2 Running celeryd as a daemon

Celery does not daemonize itself, please use one of the following daemonization tools.

4.2.1 start-stop-daemon (Debian/Ubuntu/++)

See the contrib/debian/init.d/ directory in the celery distribution, this directory contains init scripts for celeryd and celerybeat.

These scripts are configured in `/etc/default/celeryd`.

Init script: celeryd

Usage `/etc/init.d/celeryd {start|stop|force-reload|restart|try-restart|status}`

Configuration file /etc/default/celeryd

To configure celeryd you probably need to at least tell it where to chdir when it starts (to find your celeryconfig).

Example configuration

This is an example configuration for a Python project.

`/etc/default/celeryd`:

```
# Where to chdir at start.
CELERYD_CHDIR="/opt/Myproject/"

# Extra arguments to celeryd
CELERYD_OPTS="--time-limit 300"

# Name of the celery config module.#
CELERY_CONFIG_MODULE="celeryconfig"
```

Example Django configuration

This is an example configuration for those using `django-celery`:

```
# Where the Django project is.
CELERYD_CHDIR="/opt/Project/"

# Path to celeryd
CELERYD="/opt/Project/manage.py celeryd"

# Name of the projects settings module.
export DJANGO_SETTINGS_MODULE="settings"
```

Available options

- **CELERYD_OPTS** Additional arguments to celeryd, see `celeryd --help` for a list.
- **CELERYD_CHDIR** Path to chdir at start. Default is to stay in the current directory.
- **CELERYD_PID_FILE** Full path to the pidfile. Default is /var/run/celeryd.pid.
- **CELERYD_LOG_FILE** Full path to the celeryd logfile. Default is /var/log/celeryd.log
- **CELERYD_LOG_LEVEL** Log level to use for celeryd. Default is INFO.
- **CELERYD** Path to the celeryd program. Default is `celeryd`. You can point this to an virtualenv, or even use manage.py for django.
- **CELERYD_USER** User to run celeryd as. Default is current user.
- **CELERYD_GROUP** Group to run celeryd as. Default is current user.

Init script: celerybeat

Usage `/etc/init.d/celerybeat {start|stop|force-reload|restart|try-restart|status}`

Configuration file /etc/default/celerybeat or /etc/default/celeryd

Example configuration

This is an example configuration for a Python project:

`/etc/default/celeryd`:

```
# Where to chdir at start.
CELERYD_CHDIR="/opt/Myproject/"

# Extra arguments to celeryd
CELERYD_OPTS="--time-limit 300"

# Extra arguments to celerybeat
CELERYBEAT_OPTS="--schedule=/var/run/celerybeat-schedule"

# Name of the celery config module.#
CELERY_CONFIG_MODULE="celeryconfig"
```

Example Django configuration

This is an example configuration for those using `django-celery`:

```
# Where the Django project is.
CELERYD_CHDIR="/opt/Project/"

# Name of the projects settings module.
DJANGO_SETTINGS_MODULE="settings"

# Path to celeryd
CELERYD="/opt/Project/manage.py celeryd"

# Path to celerybeat
CELERYBEAT="/opt/Project/manage.py celerybeat"

# Extra arguments to celerybeat
CELERYBEAT_OPTS="--schedule=/var/run/celerybeat-schedule"
```

Available options

- **CELERYBEAT_OPTS** Additional arguments to celerybeat, see `celerybeat --help` for a list.
- **CELERYBEAT_PIDFILE** Full path to the pidfile. Default is /var/run/celeryd.pid.
- **CELERYBEAT_LOGFILE** Full path to the celeryd logfile. Default is /var/log/celeryd.log
- **CELERYBEAT_LOG_LEVEL** Log level to use for celeryd. Default is INFO.
- **CELERYBEAT** Path to the celeryd program. Default is `celeryd`. You can point this to an virtualenv, or even use manage.py for django.
- **CELERYBEAT_USER** User to run celeryd as. Default is current user.
- **CELERYBEAT_GROUP** Group to run celeryd as. Default is current user.

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.

Also you will see the commands generated, so you can try to run the celeryd command manually to read the resulting error output.

For example my `sh -x` output does this:

```
++ start-stop-daemon --start --chdir /opt/Opal/release/opal --quiet \
    --oknodo --background --make-pidfile --pidfile /var/run/celeryd.pid \
    --exec /opt/Opal/release/opal/manage.py celeryd -- --time-limit=300 \
    -f /var/log/celeryd.log -l INFO
```

Run the celeryd command after `--exec` (without the `--`) to show the actual resulting output:

```
$ /opt/Opal/release/opal/manage.py celeryd --time-limit=300 \
    -f /var/log/celeryd.log -l INFO
```

4.2.2 supervisord

- contrib/supervisord/

4.2.3 launchd (OS X)

- contrib/mac/

This page contains common recipes and techniques.

CHAPTER 5

Tutorials

Release 2.0

Date February 04, 2014

5.1 Tutorials and resources from the community

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.

- Django/Celery Quickstart (or, how I learned to stop using cron and love celery)
- How Celery, Carrot, and your messaging stack work
- Large Problems in Django, Mostly Solved: Delayed Execution
- Introduction to Celery
- RabbitMQ, Celery and Django
- Message Queues, Django and Celery Quick Start
- Background task processing and deferred execution in Django
- Build a processing queue [...] in less than a day using RabbitMQ and Celery
- How to get celeryd to work on FreeBSD
- Web-based 3D animation software
- Queued Storage Backend for Django
- RabbitMQ with Python and Ruby

5.1.1 Django/Celery Quickstart (or, how I learned to stop using cron and love celery)

http://bitkickers.blogspot.com/2010/07/djangocelery-quickstart-or-how-i.html

5.1.2 How Celery, Carrot, and your messaging stack work

http://jasonmbaker.com/how-celery-carrot-and-your-messaging-stack-wo

5.1.3 Large Problems in Django, Mostly Solved: Delayed Execution

http://ericholscher.com/blog/2010/jun/23/large-problems-django-mostly-solved-delayed-execut/

5.1.4 Introduction to Celery

Awesome slides from when Idan Gazit had a talk about Celery at PyWeb-IL: http://www.slideshare.net/idangazit/an-introduction-to-celery

5.1.5 RabbitMQ, Celery and Django

Great Celery tutorial by Robert Pogorzelski at his blog "Happy Stream of Thoughts": http://robertpogorzelski.com/blog/2009/09/10/rabbitmq-celery-and-django/

5.1.6 Message Queues, Django and Celery Quick Start

Celery tutorial by Rich Leland, the installation section is Mac OS X specific: http://mathematism.com/2010/feb/16/message-queues-django-and-celery-quick-start/

5.1.7 Background task processing and deferred execution in Django

Alon Swartz writes about celery and RabbitMQ on his blog: http://www.turnkeylinux.org/blog/django-celery-rabbitmq

5.1.8 Build a processing queue [...] in less than a day using RabbitMQ and Celery

Tutorial in 2 parts written by Tim Bull: http://timbull.com/build-a-processing-queue-with-multi-threading

5.1.9 How to get celeryd to work on FreeBSD

Installing multiprocessing on FreeBSD isn't that easy, but thanks to Viktor Petersson we now have a step-to-step guide: http://www.playingwithwire.com/2009/10/how-to-get-celeryd-to-work-on-freebsd/

5.1.10 Web-based 3D animation software

Indy Chang Liu at ThinkingCactus uses Celery to render animations asynchronously (PDF): http://ojs.pythonpapers.org/index.php/tppm/article/viewFile/105/122

5.1.11 Queued Storage Backend for Django

http://stepsandnumbers.com/archive/2010/01/04/queued-storage-backend-for-django/

5.1.12 RabbitMQ with Python and Ruby

http://www.slideshare.net/hungryblank/rabbitmq-with-python-and-ruby-rupy-2009

5.2 Using Celery with Redis/Database as the messaging queue.

There's a plug-in for celery that enables the use of Redis or an SQL database as the messaging queue. This is not part of celery itself, but exists as an extension to carrot.

5.2.1 Installation

You need to install the ghettoq library:

```
$ pip install -U ghettoq
```

5.2.2 Redis

For the Redis support you have to install the Python redis client:

```
$ pip install -U redis
```

Configuration

Configuration is easy, set the carrot backend, and configure the location of your Redis database:

```
CARROT_BACKEND = "ghettoq.taproot.Redis"

BROKER_HOST = "localhost"  # Maps to redis host.
BROKER_PORT = 6379         # Maps to redis port.
BROKER_VHOST = "celery"    # Maps to database name.
```

5.2.3 Database

Configuration

The database backend uses the Django `DATABASE_*` settings for database configuration values.

1. Set your carrot backend:

   ```
   CARROT_BACKEND = "ghettoq.taproot.Database"
   ```

2. Add `ghettoq` to `INSTALLED_APPS`:

   ```
   INSTALLED_APPS = ("ghettoq", )
   ```

3. Verify you database settings:

```
DATABASE_ENGINE = "mysql"
DATABASE_NAME = "mydb"
DATABASE_USER = "myuser"
DATABASE_PASSWORD = "secret"
```

The above is just an example, if you haven't configured your database before you should read the Django database settings reference: http://docs.djangoproject.com/en/1.1/ref/settings/#database-engine

1. Sync your database schema.

   When using Django:

   ```
   $ python manage.py syncdb
   ```

Important notes

These message queues does not have the concept of exchanges and routing keys, there's only the queue entity. As a result of this you need to set the name of the exchange to be the same as the queue:

```
CELERY_DEFAULT_EXCHANGE = "tasks"
```

or in a custom queue-mapping:

```
CELERY_QUEUES = {
    "tasks": {"exchange": "tasks"},
    "feeds": {"exchange": "feeds"},
}
```

This isn't a problem if you use the default queue setting, as the default is already using the same name for queue/exchange.

5.3 Tutorial: Creating a click counter using carrot and celery

5.3.1 Introduction

A click counter should be easy, right? Just a simple view that increments a click in the DB and forwards you to the real destination.

This would work well for most sites, but when traffic starts to increase, you are likely to bump into problems. One database write for every click is not good if you have millions of clicks a day.

So what can you do? In this tutorial we will send the individual clicks as messages using `carrot`, and then process them later with a `celery` periodic task.

Celery and carrot is excellent in tandem, and while this might not be the perfect example, you'll at least see one example how of they can be used to solve a task.

5.3.2 The model

The model is simple, `Click` has the URL as primary key and a number of clicks for that URL. Its manager, `ClickManager` implements the `increment_clicks` method, which takes a URL and by how much to increment its count by.

clickmuncher/models.py:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _


class ClickManager(models.Manager):

    def increment_clicks(self, for_url, increment_by=1):
        """Increment the click count for an URL.

            >>> Click.objects.increment_clicks("http://google.com", 10)

        """
        click, created = self.get_or_create(url=for_url,
                                    defaults={"click_count": increment_by})
        if not created:
            click.click_count += increment_by
            click.save()

        return click.click_count


class Click(models.Model):
    url = models.URLField(_(u"URL"), verify_exists=False, unique=True)
    click_count = models.PositiveIntegerField(_(u"click_count"),
                                              default=0)

    objects = ClickManager()

    class Meta:
        verbose_name = _(u"URL clicks")
        verbose_name_plural = _(u"URL clicks")
```

5.3.3 Using carrot to send clicks as messages

The model is normal django stuff, nothing new there. But now we get on to the messaging. It has been a tradition for me to put the projects messaging related code in its own `messaging.py` module, and I will continue to do so here so maybe you can adopt this practice. In this module we have two functions:

- `send_increment_clicks`

  This function sends a simple message to the broker. The message body only contains the URL we want to increment as plain-text, so the exchange and routing key play a role here. We use an exchange called `clicks`, with a routing key of `increment_click`, so any consumer binding a queue to this exchange using this routing key will receive these messages.

- `process_clicks`

  This function processes all currently gathered clicks sent using `send_increment_clicks`. Instead of issuing one database query for every click it processes all of the messages first, calculates the new click count and issues one update per URL. A message that has been received will not be deleted from the broker until it has

been acknowledged by the receiver, so if the receiver dies in the middle of processing the message, it will be re-sent at a later point in time. This guarantees delivery and we respect this feature here by not acknowledging the message until the clicks has actually been written to disk.

Note: This could probably be optimized further with some hand-written SQL, but it will do for now. Let's say it's an exercise left for the picky reader, albeit a discouraged one if you can survive without doing it.

On to the code...

clickmuncher/messaging.py:

```
from celery.messaging import establish_connection
from carrot.messaging import Publisher, Consumer
from clickmuncher.models import Click


def send_increment_clicks(for_url):
    """Send a message for incrementing the click count for an URL."""
    connection = establish_connection()
    publisher = Publisher(connection=connection,
                          exchange="clicks",
                          routing_key="increment_click",
                          exchange_type="direct")

    publisher.send(for_url)

    publisher.close()
    connection.close()


def process_clicks():
    """Process all currently gathered clicks by saving them to the
    database."""
    connection = establish_connection()
    consumer = Consumer(connection=connection,
                        queue="clicks",
                        exchange="clicks",
                        routing_key="increment_click",
                        exchange_type="direct")

    # First process the messages: save the number of clicks
    # for every URL.
    clicks_for_url = {}
    messages_for_url = {}
    for message in consumer.iterqueue():
        url = message.body
        clicks_for_url[url] = clicks_for_url.get(url, 0) + 1
        # We also need to keep the message objects so we can ack the
        # messages as processed when we are finished with them.
        if url in messages_for_url:
            messages_for_url[url].append(message)
        else:
            messages_for_url[url] = [message]

    # Then increment the clicks in the database so we only need
    # one UPDATE/INSERT for each URL.
    for url, click_count in clicks_for_urls.items():
        Click.objects.increment_clicks(url, click_count)
        # Now that the clicks has been registered for this URL we can
        # acknowledge the messages
```

```
        [message.ack() for message in messages_for_url[url]]

    consumer.close()
    connection.close()
```

5.3.4 View and URLs

This is also simple stuff, don't think I have to explain this code to you. The interface is as follows, if you have a link to http://google.com you would want to count the clicks for, you replace the URL with:

    http://mysite/clickmuncher/count/?u=http://google.com

and the `count` view will send off an increment message and forward you to that site.

clickmuncher/views.py:

```
from django.http import HttpResponseRedirect
from clickmuncher.messaging import send_increment_clicks


def count(request):
    url = request.GET["u"]
    send_increment_clicks(url)
    return HttpResponseRedirect(url)
```

clickmuncher/urls.py:

```
from django.conf.urls.defaults import patterns, url
from clickmuncher import views

urlpatterns = patterns("",
    url(r'^$', views.count, name="clickmuncher-count"),
)
```

5.3.5 Creating the periodic task

Processing the clicks every 30 minutes is easy using celery periodic tasks.

clickmuncher/tasks.py:

```
from celery.task import PeriodicTask
from clickmuncher.messaging import process_clicks
from datetime import timedelta


class ProcessClicksTask(PeriodicTask):
    run_every = timedelta(minutes=30)

    def run(self, **kwargs):
        process_clicks()
```

We subclass from `celery.task.base.PeriodicTask`, set the `run_every` attribute and in the body of the task just call the `process_clicks` function we wrote earlier.

5.3.6 Finishing

There are still ways to improve this application. The URLs could be cleaned so the URL http://google.com and http://google.com/ is the same. Maybe it's even possible to update the click count using a single UPDATE query?

If you have any questions regarding this tutorial, please send a mail to the mailing-list or come join us in the #celery IRC channel at Freenode: http://celeryq.org/introduction.html#getting-help

CHAPTER 6

Frequently Asked Questions

6.1 General

6.1.1 What kinds of things should I use celery for?

Answer: Queue everything and delight everyone is a good article describing why you would 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 performane 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.

6.2 Misconceptions

6.2.1 Is celery dependent on pickle?

Answer: No.

Celery can support any serialization scheme and has support for JSON/YAML and Pickle by default. You can even send one task using pickle, and another one with JSON seamlessly, this is because every task is associated with a content-type. The default serialization scheme is pickle because it's the most used, and it has support for sending complex objects as task arguments.

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.

6.2.2 Is celery for Django only?

Answer: No.

Celery does not depend on Django anymore. To use Celery with Django you have to use the django-celery package.

6.2.3 Do I have to use AMQP/RabbitMQ?

Answer: No.

You can also use Redis or an SQL database, see Using other queues.

Redis or a database won't perform as well as an AMQP broker. If you have strict reliability requirements you are encouraged to use RabbitMQ or another AMQP broker. Redis/database also use polling, so they are likely to consume more resources. However, if you for some reason are not 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.

6.2.4 Is celery multi-lingual?

Answer: Yes.

celeryd 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 consume messages. There's no other communication involved.

Also, there's another way to be language indepedent, and that is 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. See: User Guide: Remote Tasks.

6.3 Troubleshooting

6.3.1 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 InnoDBs 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)

6.3.2 celeryd is not doing anything, just hanging

Answer: See MySQL is throwing deadlock errors, what can I do?. or *Why is Task.delay/apply* just hanging?*.

6.3.3 Why is Task.delay/apply*/celeryd just hanging?

Answer: There is a bug in some AMQP clients that will make it hang if it's not able to authenticate the current user, the password doesn't match or the user does not have access to the virtual host specified. Be sure to check your broker logs (for RabbitMQ that is `/var/log/rabbitmq/rabbit.log` on most systems), it usually contains a message describing the reason.

6.3.4 Why won't celeryd run on FreeBSD?

Answer: multiprocessing.Pool requires a working POSIX semaphore implementation which isn't enabled in FreeBSD by default. You have to enable POSIX semaphores in the kernel and manually recompile multiprocessing.

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/

6.3.5 I'm having `IntegrityError: Duplicate Key` errors. Why?

Answer: See MySQL is throwing deadlock errors, what can I do?. Thanks to howsthedotcom.

6.3.6 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 celery process taking the messages hostage. This could happen if celeryd wasn't properly shut down.

When a message is recieved by a worker the broker waits for it to be acknowledged before marking the message as processed. The broker will not 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:

```
ps auxww | grep celeryd | awk '{print $2}' | xargs kill
```

You might have to wait a while until all workers have finished the work they're doing. If it's still hanging after a long time you can kill them by force with:

```
ps auxww | grep celeryd | awk '{print $2}' | xargs kill -9
```

6.3.7 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 celeryds logfile to see if it's able to find the task, or if some other error is happening.

6.3.8 Why won't my Periodic Task run?

Answer: See Why won't my Task run?.

6.3.9 How do I discard all waiting tasks?

Answer: Use `celery.task.discard_all()`, like this:

```
>>> from celery.task import discard_all
>>> discard_all()
1753
```

The number `1753` is the number of messages deleted.

You can also start celeryd with the `--discard` argument which will accomplish the same thing.

6.3.10 I've discarded messages, but there are still messages left in the queue?

Answer: Tasks are acknowledged (removed from the queue) as soon as they are actually executed. After the worker has received a task, it will take some time until it is actually executed, especially if there are a lot of tasks already waiting for execution. Messages that are not acknowledged are hold 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 discard the tasks using `discard_all`.

6.4 Results

6.4.1 How do I get the result of a task if I have the ID that points there?

Answer: Use `Task.AsyncResult`:

```
>>> result = MyTask.AsyncResult(task_id)
>>> result.get()
```

This will give you a `celery.result.BaseAsyncResult` instance using the tasks current result backend.

If you need to specify a custom result backend you should use `celery.result.BaseAsyncResult` directly:

```
>>> from celery.result import BaseAsyncResult
>>> result = BaseAsyncResult(task_id, backend=...)
>>> result.get()
```

6.5 Brokers

6.5.1 Why is RabbitMQ crashing?

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: http://www.rabbitmq.com/faq.html#node-runs-out-of-memory

Some common Celery misconfigurations can crash RabbitMQ:

- Events.

Running `celeryd` with the `-E`/`--events` option will send messages for events happening inside of the worker. If these event messages are not consumed, you will eventually run out of memory.

Events should only be enabled if you have an active monitor consuming them.

- 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.

If you don't use the results for a task, make sure you set the `ignore_result` option:

Results can also be disabled globally using the `CELERY_IGNORE_RESULT` setting.

6.5.2 Can I use celery with ActiveMQ/STOMP?

Answer: Yes, but this is somewhat experimental for now. It is working ok in a test configuration, but it has not been tested in production. If you have any problems using STOMP with celery, please report an issue here:

`http://github.com/ask/celery/issues/`

The STOMP carrot backend requires the stompy library:

```
$ pip install stompy
$ cd python-stomp
$ sudo python setup.py install
$ cd ..
```

In this example we will use a queue called `celery` which we created in the ActiveMQ web admin interface.

Note: When using ActiveMQ the queue name needs to have `"/queue/"` prepended to it. i.e. the queue `celery` becomes `/queue/celery`.

Since STOMP doesn't have exchanges and the routing capabilities of AMQP, you need to set `exchange` name to the same as the queue name. This is a minor inconvenience since carrot needs to maintain the same interface for both AMQP and STOMP.

Use the following settings in your `celeryconfig.py`/django `settings.py`:

```
# Use the stomp carrot backend.
CARROT_BACKEND = "stomp"

# STOMP hostname and port settings.
BROKER_HOST = "localhost"
BROKER_PORT = 61613

# The queue name to use (the exchange *must* be set to the
# same as the queue name when using STOMP)
CELERY_DEFAULT_QUEUE = "/queue/celery"
CELERY_DEFAULT_EXCHANGE = "/queue/celery"

CELERY_QUEUES = {
    "/queue/celery": {"exchange": "/queue/celery"}
}
```

6.5.3 What features are not supported when using ghettoq/STOMP?

This is a (possible incomplete) list of features not available when using the STOMP backend:

- routing keys
- exchange types (direct, topic, headers, etc)
- immediate
- mandatory

6.6 Tasks

6.6.1 How can I reuse the same connection when applying tasks?

Answer: See *Executing Tasks*.

6.6.2 Can I execute a task by name?

Answer: Yes. Use `celery.execute.send_task()`. You can also execute a task by name from any language that has an AMQP client.

```
>>> from celery.execute import send_task
>>> send_task("tasks.add", args=[2, 2], kwargs={})
<AsyncResult: 373550e8-b9a0-4666-bc61-ace01fa4f91d>
```

6.6.3 How can I get the task id of the current task?

Answer: Celery does set some default keyword arguments if the task accepts them (you can accept them by either using `**kwargs`, or list them specifically):

```
@task
def mytask(task_id=None):
    cache.set(task_id, "Running")
```

The default keyword arguments are documented here: http://celeryq.org/docs/userguide/tasks.html#default-keyword-arguments

6.6.4 Can I specify a custom task_id?

Answer: Yes. Use the `task_id` argument to `apply_async()`:

```
>>> task.apply_async(args, kwargs, task_id="...")
```

6.6.5 Can I use natural task ids?

Answer: Yes, but make sure it is unique, as the behavior for two tasks existing with the same id is undefined.

The world will probably not explode, but at the worst they can overwrite each others results.

6.6.6 How can I run a task once another task has finished?

Answer: You can safely launch a task inside a task. Also, a common pattern is to use callback tasks:

```
@task()
def add(x, y, callback=None):
    result = x + y
    if callback:
        subtask(callback).delay(result)
    return result


@task(ignore_result=True)
def log_result(result, **kwargs):
    logger = log_result.get_logger(**kwargs)
    logger.info("log_result got: %s" % (result, ))
```

Invocation:

```
>>> add.delay(2, 2, callback=log_result.subtask())
```

See *Sets of tasks, Subtasks and Callbacks* for more information.

6.6.7 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 celery.task.control import revoke
>>> revoke(task_id)
```

6.6.8 Why aren't my remote control commands received by all workers?

Answer: To receive broadcast remote control commands, every `celeryd` uses its hostname to create a unique queue name to listen to, so if you have more than one worker with the same hostname, the control commands will be recieved in round-robin between them.

To work around this you can explicitly set the hostname for every worker using the `--hostname` argument to `celeryd`:

```
$ celeryd --hostname=$(hostname).1
$ celeryd --hostname=$(hostname).2
```

etc, etc.

6.6.9 Can I send some tasks to only some servers?

Answer: Yes. You can route tasks to an arbitrary server using AMQP, and a worker can bind to as many queues as it wants.

See *Routing Tasks* for more information.

6.6.10 Can I change the interval of a periodic task at runtime?

Answer: Yes. You can override `PeriodicTask.is_due` or turn `PeriodicTask.run_every` into a property:

```
class MyPeriodic(PeriodicTask):

    def run(self):
        # ...

    @property
    def run_every(self):
        return get_interval_from_database(...)
```

6.6.11 Does celery support task priorities?

Answer: No. In theory, yes, as AMQP supports priorities. However RabbitMQ doesn't implement them yet.

The usual way to prioritize work in celery, is to route high priority tasks to different servers. In the real world this may actually work better than per message priorities. You can use this in combination with rate limiting to achieve a highly performant system.

6.6.12 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 catchable with the `try:` block. The AMQP transaction is not used for these errors: **if the task raises an exception it is still acked!**.

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 is not known to crash, and if it does it is 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 has failed, but this is rarely the case. Imagine the following task:

```
@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 is not 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 acknowledgement)

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 reentrant combine that with `acks_late` if that level of reliability is required.

6.6.13 Can I schedule tasks to execute at a specific time?

Answer: Yes. You can use the `eta` argument of `Task.apply_async()`.

Or to schedule a periodic task at a specific time, use the `celery.task.schedules.crontab` schedule behavior:

```
from celery.task.schedules import crontab
from celery.decorators import periodic_task

@periodic_task(run_every=crontab(hours=7, minute=30, day_of_week="mon"))
def every_monday_morning():
    print("This is run every monday morning at 7:30")
```

6.6.14 How do I shut down `celeryd` safely?

Answer: Use the `TERM` signal, and celery will finish all currently executing jobs and shut down as soon as possible. No tasks should be lost.

You should never stop `celeryd` with the `KILL` signal (`-9`), unless you've tried `TERM` a few times and waited a few minutes to let it get a chance to shut down. As if you do tasks may be terminated mid-execution, and they will not be re-run unless you have the `acks_late` option set. (`Task.acks_late` / `CELERY_ACKS_LATE`).

6.6.15 How do I run celeryd in the background on [platform]?

Answer: Please see *Running celeryd as a daemon*.

6.7 Windows

6.7.1 celeryd keeps spawning processes at startup

Answer: This is a known issue on Windows. You have to start celeryd with the command:

```
$ python -m celeryd.bin.celeryd
```

Any additional arguments can be appended to this command.

See http://bit.ly/bo9RSw

6.7.2 The `-B` / `--beat` option to celeryd doesn't work?

Answer: That's right. Run `celerybeat` and `celeryd` as separate services instead.

6.7.3 `django-celery` can't find settings?

Answer: You need to specify the `--settings` argument to `manage.py`:

```
$ python manage.py celeryd start --settings=settings
```

See http://bit.ly/bo9RSw

CHAPTER 7

API Reference

Release 2.0

Date February 04, 2014

7.1 Task Decorators - celery.decorators

7.2 Defining Tasks - celery.task.base

7.3 Task Sets, Subtasks and Callbacks - celery.task.sets

7.4 Executing Tasks - celery.execute

7.5 Task Result - celery.result

7.6 Task Information and Utilities - celery.task

7.7 Configuration - celery.conf

7.7.1 Queues

`celery.conf.`**`QUEUES`**
Queue name/options mapping.

celery.conf.**DEFAULT_QUEUE**
Name of the default queue.

celery.conf.**DEFAULT_EXCHANGE**
Default exchange.

celery.conf.**DEFAULT_EXCHANGE_TYPE**
Default exchange type.

celery.conf.**DEFAULT_DELIVERY_MODE**
Default delivery mode (`"persistent"` or `"non-persistent"`). Default is `"persistent"`.

celery.conf.**DEFAULT_ROUTING_KEY**
Default routing key used when sending tasks.

celery.conf.**BROKER_CONNECTION_TIMEOUT**
The timeout in seconds before we give up establishing a connection to the AMQP server.

celery.conf.**BROADCAST_QUEUE**
Name prefix for the queue used when listening for broadcast messages. The workers hostname will be appended to the prefix to create the final queue name.

Default is `"celeryctl"`.

celery.conf.**BROADCAST_EXCHANGE**
Name of the exchange used for broadcast messages.

Default is `"celeryctl"`.

celery.conf.**BROADCAST_EXCHANGE_TYPE**
Exchange type used for broadcast messages. Default is `"fanout"`.

celery.conf.**EVENT_QUEUE**
Name of queue used to listen for event messages. Default is `"celeryevent"`.

celery.conf.**EVENT_EXCHANGE**
Exchange used to send event messages. Default is `"celeryevent"`.

celery.conf.**EVENT_EXCHANGE_TYPE**
Exchange type used for the event exchange. Default is `"topic"`.

celery.conf.**EVENT_ROUTING_KEY**
Routing key used for events. Default is `"celeryevent"`.

celery.conf.**EVENT_SERIALIZER**
Type of serialization method used to serialize events. Default is `"json"`.

celery.conf.**RESULT_EXCHANGE**
Exchange used by the AMQP result backend to publish task results. Default is `"celeryresult"`.

7.7.2 Sending E-Mails

celery.conf.**CELERY_SEND_TASK_ERROR_EMAILS**
If set to `True`, errors in tasks will be sent to `ADMINS` by e-mail.

celery.conf.**ADMINS**
List of `(name, email_address)` tuples for the admins that should receive error e-mails.

celery.conf.**SERVER_EMAIL**
The e-mail address this worker sends e-mails from. Default is `"celery@localhost"`.

celery.conf.**MAIL_HOST**
The mail server to use. Default is `"localhost"`.

`celery.conf.`**`MAIL_HOST_USER`**
Username (if required) to log on to the mail server with.

`celery.conf.`**`MAIL_HOST_PASSWORD`**
Password (if required) to log on to the mail server with.

`celery.conf.`**`MAIL_PORT`**
The port the mail server is listening on. Default is `25`.

7.7.3 Execution

`celery.conf.`**`ALWAYS_EAGER`**
Always execute tasks locally, don't send to the queue.

`celery.conf.`**`EAGER_PROPAGATES_EXCEPTIONS`**
If set to `True`, `celery.execute.apply()` will re-raise task exceptions. It's the same as always running apply with `throw=True`.

`celery.conf.`**`TASK_RESULT_EXPIRES`**
Task tombstone expire time in seconds.

`celery.conf.`**`IGNORE_RESULT`**
If enabled, the default behavior will be to not store task results.

`celery.conf.`**`TRACK_STARTED`**
If enabled, the default behavior will be to track when tasks starts by storing the `STARTED` state.

`celery.conf.`**`ACKS_LATE`**
If enabled, the default behavior will be to acknowledge task messages after the task is executed.

`celery.conf.`**`STORE_ERRORS_EVEN_IF_IGNORED`**
If enabled, task errors will be stored even though `Task.ignore_result` is enabled.

`celery.conf.`**`MAX_CACHED_RESULTS`**
Total number of results to store before results are evicted from the result cache.

`celery.conf.`**`TASK_SERIALIZER`**
A string identifying the default serialization method to use. Can be `pickle` (default), `json`, `yaml`, or any custom serialization methods that have been registered with `carrot.serialization.registry`.

Default is `pickle`.

`celery.conf.`**`RESULT_BACKEND`**
The backend used to store task results (tombstones).

`celery.conf.`**`CELERY_CACHE_BACKEND`**
Celery cache backend.

`celery.conf.`**`SEND_EVENTS`**
If set, celery will send events that can be captured by monitors like `celerymon`. Default is: `False`.

`celery.conf.`**`DEFAULT_RATE_LIMIT`**
The default rate limit applied to all tasks which doesn't have a custom rate limit defined. (Default: None)

`celery.conf.`**`DISABLE_RATE_LIMITS`**
If `True` all rate limits will be disabled and all tasks will be executed as soon as possible.

7.7.4 Broker

celery.conf.**BROKER_CONNECTION_RETRY**
Automatically try to re-establish the connection to the AMQP broker if it's lost.

celery.conf.**BROKER_CONNECTION_MAX_RETRIES**
Maximum number of retries before we give up re-establishing a connection to the broker.

If this is set to `0` or `None`, we will retry forever.

Default is `100` retries.

7.7.5 Celerybeat

celery.conf.**CELERYBEAT_LOG_LEVEL**
Default log level for celerybeat. Default is: `INFO`.

celery.conf.**CELERYBEAT_LOG_FILE**
Default log file for celerybeat. Default is: `None` (stderr)

celery.conf.**CELERYBEAT_SCHEDULE_FILENAME**
Name of the persistent schedule database file. Default is: `celerybeat-schedule`.

celery.conf.**CELERYBEAT_MAX_LOOP_INTERVAL**
The maximum number of seconds celerybeat is allowed to sleep between checking the schedule. The default is 5 minutes, which means celerybeat can only sleep a maximum of 5 minutes after checking the schedule run-times for a periodic task to apply. If you change the run_times of periodic tasks at run-time, you may consider lowering this value for changes to take effect faster (A value of 5 minutes, means the changes will take effect in 5 minutes at maximum).

7.7.6 Celerymon

celery.conf.**CELERYMON_LOG_LEVEL**
Default log level for celerymon. Default is: `INFO`.

celery.conf.**CELERYMON_LOG_FILE**
Default log file for celerymon. Default is: `None` (stderr)

7.7.7 Celeryd

celery.conf.**LOG_LEVELS**
Mapping of log level names to `logging` module constants.

celery.conf.**CELERYD_LOG_FORMAT**
The format to use for log messages.

celery.conf.**CELERYD_TASK_LOG_FORMAT**
The format to use for task log messages.

celery.conf.**CELERYD_LOG_FILE**
Filename of the daemon log file. Default is: `None` (stderr)

celery.conf.**CELERYD_LOG_LEVEL**
Default log level for daemons. (`WARN`)

`celery.conf.`**CELERYD_CONCURRENCY**
The number of concurrent worker processes. If set to `0` (the default), the total number of available CPUs/cores will be used.

`celery.conf.`**CELERYD_PREFETCH_MULTIPLIER**
The number of concurrent workers is multipled by this number to yield the wanted AMQP QoS message prefetch count. Default is: `4`

`celery.conf.`**CELERYD_POOL**
Name of the task pool class used by the worker. Default is `"celery.concurrency.processes.TaskPool"`.

`celery.conf.`**CELERYD_LISTENER**
Name of the listener class used by the worker. Default is `"celery.worker.listener.CarrotListener"`.

`celery.conf.`**CELERYD_MEDIATOR**
Name of the mediator class used by the worker. Default is `"celery.worker.controllers.Mediator"`.

`celery.conf.`**CELERYD_ETA_SCHEDULER**
Name of the ETA scheduler class used by the worker. Default is `"celery.worker.controllers.ScheduleController"`.

7.8 Remote Management of Workers - celery.task.control

7.9 HTTP Callback Tasks - celery.task.http

7.10 Periodic Task Schedule Behaviors - celery.task.schedules

7.11 Signals - celery.signals

7.11.1 Basics

Several kinds of events trigger signals, you can connect to these signals to perform actions as they trigger.

Example connecting to the `task_sent` signal:

```
from celery.signals import task_sent

def task_sent_handler(sender=None, task_id=None, task=None, args=None,
                      kwargs=None, **kwds):
    print("Got signal task_sent for task id %s" % (task_id, ))

task_sent.connect(task_sent_handler)
```

Some signals also have a sender which you can filter by. For example the `task_sent` signal uses the task name as a sender, so you can connect your handler to be called only when tasks with name `"tasks.add"` has been sent by providing the `sender` argument to `connect`:

```
task_sent.connect(task_sent_handler, sender="tasks.add")
```

7.11.2 Signals

Task Signals

`celery.signals.`**`task_sent`**

Dispatched when a task has been sent to the broker. Note that this is executed in the client process, the one sending the task, not in the worker.

Sender is the name of the task being sent.

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.
- **eta** The time to execute the task.
- **taskset** Id of the taskset this task is part of (if any).

`celery.signals.`**`task_prerun`**

Dispatched before a task is executed.

Sender is the task class 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.

`celery.signals.`**`task_postrun`**

Dispatched after a task has been executed.

Sender is the task class 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.

Worker Signals

celery.signals.**worker_init**
Dispatched before the worker is started.

celery.signals.**worker_ready**
Dispatched when the worker is ready to accept work.

celery.signals.**worker_process_init**
Dispatched by each new pool worker process when it starts.

celery.signals.**worker_shutdown**
Dispatched when the worker is about to shut down.

7.12 Exceptions - celery.exceptions

Common Exceptions

exception celery.exceptions.**AlreadyRegistered**
The task is already registered.

exception celery.exceptions.**ImproperlyConfigured**
Celery is somehow improperly configured.

exception celery.exceptions.**MaxRetriesExceededError**
The tasks max restart limit has been exceeded.

exception celery.exceptions.**NotRegistered**(*message*, **args*, ***kwargs*)
The task is not registered.

exception celery.exceptions.**QueueNotFound**
Task routed to a queue not in CELERY_QUEUES.

exception celery.exceptions.**RetryTaskError**(*message*, *exc*, **args*, ***kwargs*)
The task is to be retried later.

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.**TaskRevokedError**
The task has been revoked, so no result available.

exception celery.exceptions.**TimeLimitExceeded**
The time limit has been exceeded and the job has been terminated.

exception celery.exceptions.**TimeoutError**
The operation timed out.

exception celery.exceptions.**WorkerLostError**
The worker processing a task has exited prematurely.

7.13 Built-in Task Classes - celery.task.builtins

7.14 Loaders - celery.loaders

celery.loaders.**current_loader**()
Detect and return the current loader.

celery.loaders.**get_loader_cls**(*loader*)
Get loader class by name/alias

celery.loaders.**load_settings**()
Load the global settings object.

celery.loaders.**setup_loader**()

7.15 Loader Base Classes - celery.loaders.base

class celery.loaders.base.**BaseLoader**
The base class for loaders.

Loaders handles to following things:

- 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 modules are imported to find tasks?

conf
Loader configuration.

configured = **False**

import_default_modules()

import_task_module(*module*)

init_worker()

on_process_cleanup()
This method is called after a task is executed.

on_task_init(*task_id*, *task*)
This method is called before a task is executed.

on_worker_init()
This method is called when the worker (celeryd) starts.

override_backends = **{}**

worker_initialized = **False**

7.16 Default Loader - celery.loaders.default

class celery.loaders.default.**Loader**
The default loader.

See the FAQ for example usage.

import_from_cwd(*module*, *imp=<function import_module at 0x432a578>*)
Import module, but make sure it finds modules located in the current directory.

Modules located in the current directory has precedence over modules located in sys.path.

on_worker_init()
Imports modules at worker init so tasks can be registered and used by the worked.

The list of modules to import is taken from the CELERY_IMPORTS setting in celeryconf.py.

read_configuration()
Read configuration from celeryconfig.py and configure celery and Django so it can be used by regular Python.

setup_settings(*settingsdict*)

exception celery.loaders.default.**NotConfigured**
Celery has not been configured, as no config module has been found.

class celery.loaders.default.**Settings**

celery.loaders.default.**wanted_module_item**(*item*)

7.17 Task Registry - celery.registry

celery.registry

class celery.registry.**TaskRegistry**
Site registry for tasks.

exception NotRegistered(*message*, **args*, ***kwargs*)
The task is not registered.

TaskRegistry.**filter_types**(*type*)
Return all tasks of a specific type.

TaskRegistry.**periodic**()
Get all periodic task types.

TaskRegistry.**pop**(*key*, **args*)

TaskRegistry.**register**(*task*)
Register a task in the task registry.

The task will be automatically instantiated if not already an instance.

TaskRegistry.**regular**()
Get all regular task types.

TaskRegistry.**unregister**(*name*)
Unregister task by name.

Parameters name – name of the task to unregister, or a celery.task.base.Task with a valid name attribute.

Raises celery.exceptions.NotRegistered if the task has not been registered.

7.18 Task States - celery.states

- States
- Sets

7.18.1 States

celery.states.**PENDING**
Task is waiting for execution or unknown.

celery.states.**STARTED**
Task has been started.

celery.states.**SUCCESS**
Task has been successfully executed.

celery.states.**FAILURE**
Task execution resulted in failure.

celery.states.**RETRY**
Task is being retried.

celery.states.**REVOKED**
Task has been revoked.

7.18.2 Sets

celery.states.**READY_STATES**
Set of states meaning the task result is ready (has been executed).

celery.states.**UNREADY_STATES**
Set of states meaning the task result is not ready (has not been executed).

celery.states.**EXCEPTION_STATES**
Set of states meaning the task returned an exception.

celery.states.**PROPAGATE_STATES**
Set of exception states that should propagate exceptions to the user.

celery.states.**ALL_STATES**
Set of all possible states.

7.19 Messaging - celery.messaging

7.20 Contrib: Abortable tasks - celery.contrib.abortable

7.21 Events - celery.events

7.22 In-memory Representation of Cluster State - celery.events.state

7.23 Celery Worker Daemon - celery.bin.celeryd

7.24 Celery Periodic Task Server - celery.bin.celerybeat

7.25 celeryev: Curses Event Viewer - celery.bin.celeryev

7.26 caqmadm: AMQP API Command-line Shell - celery.bin.camqadm

7.27 Celeryd Multi Tool - celery.bin.celeryd_multi

class celery.bin.celeryd_multi.**MultiTool**

expand(*argv*, *cmd=None*)

get(*argv*, *cmd*)

help(*argv*, *cmd=None*)

names(*argv*, *cmd*)

start(*argv*, *cmd*)

usage()

class celery.bin.celeryd_multi.**NamespacedOptionParser**(*args*)

add_option(*name*, *value*, *short=False*, *ns=None*)

optmerge(*ns*, *defaults=None*)

parse()

process_long_opt(*arg*, *value=None*)

process_short_opt(*arg*, *value=None*)

celery.bin.celeryd_multi.**abbreviations**(*map*)

celery.bin.celeryd_multi.**format_opt**(*opt*, *value*)

celery.bin.celeryd_multi.**main**()

celery.bin.celeryd_multi.**multi_args**(*p*, *cmd='celeryd'*, *append=''*, *prefix=''*, *suffix=''*)

celery.bin.celeryd_multi.**parse_ns_range**(*ns*, *ranges=False*)

celery.bin.celeryd_multi.**quote**(*v*)

celery.bin.celeryd_multi.**say**(*m*)

CHAPTER 8

Internals

Release 2.0

Date February 04, 2014

8.1 Celery Deprecation Timeline

8.1.1 Removals for version 2.0

- The following settings will be removed:

Setting name	Replace with
CELERY_AMQP_CONSUMER_QUEUES	CELERY_QUEUES
CELERY_AMQP_CONSUMER_QUEUES	CELERY_QUEUES
CELERY_AMQP_EXCHANGE	CELERY_DEFAULT_EXCHANGE
CELERY_AMQP_EXCHANGE_TYPE	CELERY_DEFAULT_AMQP_EXCHANGE_TYPE
CELERY_AMQP_CONSUMER_ROUTING_KEY	CELERY_QUEUES
CELERY_AMQP_PUBLISHER_ROUTING_KEY	CELERY_DEFAULT_ROUTING_KEY

- `CELERY_LOADER` definitions without class name.

  E.g. `celery.loaders.default`, needs to include the class name: `celery.loaders.default.Loader`.

- **`TaskSet.run()`. Use `celery.task.base.TaskSet.apply_async()`** instead.
- The module `celery.task.rest`; use `celery.task.http` instead.

8.2 Internals: The worker

8.2.1 Introduction

The worker consists of 4 main components: the broker listener, 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.

8.2.2 Data structures

ready_queue

The ready queue is either an instance of `Queue.Queue`, or *celery.buckets.TaskBucket*. The latter if rate limiting is enabled.

eta_schedule

The ETA schedule is a heap queue sorted by time.

8.2.3 Components

CarrotListener

Receives messages from the broker using `carrot`.

When a message is received it's converted into a `celery.worker.job.TaskRequest` object.

Tasks with an ETA are entered into the `eta_schedule`, messages that can be immediately processed are moved directly to the `ready_queue`.

ScheduleController

The schedule controller is running the `eta_schedule`. If the scheduled tasks eta has passed it is moved to the `ready_queue`, otherwise the thread sleeps until the eta is met (remember that the schedule is sorted by time).

Mediator

The mediator simply moves tasks in the `ready_queue` over to the task pool for execution using `celery.worker.job.TaskRequest.execute_using_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.

8.3 Task Message Protocol

8.3.1 Message format

- **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 is not scheduled, but will be executed asap.

8.3.2 Example message

This is an example invocation of the `celery.task.PingTask` task in JSON format:

```
{"task": "celery.task.PingTask",
 "args": [],
 "kwargs": {},
 "retries": 0,
 "eta": "2009-11-17T12:30:56.527191"}
```

8.3.3 Serialization

The protocol supports several serialization formats 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

8.4 List of Worker Events

This is the list of events sent by the worker. The monitor uses these to visualize the state of the cluster.

8.4.1 Task Events

- task-received(uuid, name, args, kwargs, retries, eta, hostname, timestamp)

  Sent when the worker receives a task.

- task-started(uuid, hostname, timestamp)

  Sent just before the worker executes the task.

- task-succeeded(uuid, result, runtime, hostname, timestamp)

  Sent if the task executed successfully. Runtime is the time it took to execute the task using the pool. (Time starting from the task is sent to the pool, and ending when the pool result handlers callback is called).

- task-failed(uuid, exception, traceback, hostname, timestamp)

  Sent if the execution of the task failed.

- task-revoked(uuid)

  Sent if the task has been revoked (Note that this is likely to be sent by more than one worker)

- task-retried(uuid, exception, traceback, hostname, delay, timestamp)

  Sent if the task failed, but will be retried in the future. (**NOT IMPLEMENTED**)

8.4.2 Worker Events

- worker-online(hostname, timestamp)

  The worker has connected to the broker and is online.

- worker-heartbeat(hostname, timestamp)

  Sent every minute, if the worker has not sent a heartbeat in 2 minutes, it's considered to be offline.

- worker-offline(hostname, timestamp)

  The worker has disconnected from the broker.

8.5 Module Index

- Worker
  - celery.worker
  - celery.worker.job
  - celery.worker.pool
  - celery.worker.listener
  - celery.worker.controllers
  - celery.worker.scheduler
  - celery.worker.buckets
  - celery.worker.heartbeat
  - celery.worker.revoke
  - celery.worker.control
- Tasks
  - celery.decorators
  - celery.registry
  - celery.task
  - celery.task.base
  - celery.task.http
  - celery.task.control
  - celery.task.builtins
- Execution
  - celery.execute
  - celery.execute.trace
  - celery.result
  - celery.states
- Messaging
  - celery.messaging
- Result backends
  - celery.backends
  - celery.backends.base
  - celery.backends.amqp
  - celery.backends.database
- Loaders
  - celery.loaders
  - celery.loaders.base - Loader base classes
  - celery.loaders.default - The default loader
- CeleryBeat
  - celery.beat
- Events
  - celery.events
- Logging
  - celery.log
  - celery.utils.patch
- Configuration
  - celery.conf
- Miscellaneous
  - celery.datastructures
  - celery.exceptions
  - celery.platform
  - celery.utils
  - celery.utils.info
  - celery.utils.compat

8.5.1 Worker

celery.worker

- `WorkController`

This is the worker's main process. It starts and stops all the components required by the worker: Pool, Mediator, Scheduler, ClockService, and Listener.

- `process_initializer()`

This is the function used to initialize pool processes. It sets up loggers and imports required task modules, etc.

celery.worker.job

- `TaskRequest`

A request to execute a task. Contains the task name, id, args and kwargs. Handles acknowledgement, execution, writing results to backends and error handling (including error e-mails)

celery.worker.pool

celery.worker.listener

celery.worker.controllers

celery.worker.scheduler

celery.worker.buckets

celery.worker.heartbeat

celery.worker.revoke

celery.worker.control

- celery.worker.registry
- celery.worker.builtins

8.5.2 Tasks

celery.decorators

celery.registry

celery.task

celery.task.base

celery.task.http

celery.task.control

celery.task.builtins

8.5.3 Execution

celery.execute

celery.execute.trace

celery.result

celery.states

celery.signals

8.5.4 Messaging

celery.messaging

8.5.5 Result backends

celery.backends

celery.backends.base

celery.backends.amqp

celery.backends.database

8.5.6 Loaders

celery.loaders

Loader autodetection, and working with the currently selected loader.

celery.loaders.base - Loader base classes

celery.loaders.default - The default loader

8.5.7 CeleryBeat

celery.beat

8.5.8 Events

celery.events

8.5.9 Logging

celery.log

celery.utils.patch

8.5.10 Configuration

celery.conf

8.5.11 Miscellaneous

celery.datastructures

celery.exceptions

celery.platform

celery.utils

celery.utils.info

celery.utils.compat

8.6 Internal Module Reference

Release 2.0

Date February 04, 2014

exception celery.serialization.**UnpickleableExceptionWrapper**(*exc_module*, *exc_cls_name*, *exc_args*)

Parameters

- **exc_module** – see `exc_module`.
- **exc_cls_name** – see `exc_cls_name`.
- **exc_args** – see `exc_args`

exc_module
The module of the original exception.

exc_cls_name
The name of the original exception class.

exc_args
The arguments for the original exception.

Example

```
>>> try:
...     something_raising_unpickleable_exc()
>>> except Exception, e:
...     exc = UnpickleableException(e.__class__.__module__,
...                                 e.__class__.__name__,
...                                 e.args)
...     pickle.dumps(exc) # Works fine.
```

classmethod from_exception(*exc*)

restore()

celery.serialization.**create_exception_cls**(*name*, *module*, *parent=None*)
Dynamically create an exception class.

celery.serialization.**find_nearest_pickleable_exception**(*exc*)
With an exception instance, iterate over its super classes (by mro) and find the first super exception that is 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** – An exception instance.

Returns the nearest exception if it's not `Exception` or below, if it is it returns `None`.

:rtype `Exception`:

celery.serialization.**get_pickleable_exception**(*exc*)
Make sure exception is pickleable.

celery.serialization.**get_pickled_exception**(*exc*)
Get original exception from exception pickled using `get_pickleable_exception()`.

celery.serialization.**subclass_exception**(*name*, *parent*, *module*)

8.6.27 Datastructures - celery.datastructures

class celery.datastructures.**AttributeDict**
Dict subclass with attribute access.

class celery.datastructures.**ExceptionInfo**(*exc_info*)
Exception wrapping an exception and its traceback.

Parameters **exc_info** – The exception tuple info as returned by `traceback.format_exception()`.

exception
The original exception.

traceback
A traceback from the point when `exception` was raised.

class `celery.datastructures.`**LimitedSet**(*maxlen=None*, *expires=None*)
Kind-of Set with limitations.

Good for when you need to test for membership (`a in set`), but the list might become to big, so you want to limit it so it doesn't consume too much resources.

Parameters

- **maxlen** – Maximum number of members before we start deleting expired members.
- **expires** – Time in seconds, before a membership expires.

add(*value*)
Add a new member.

as_dict()

chronologically

first
Get the oldest member.

pop_value(*value*)
Remove membership by finding value.

update(*other*)

class `celery.datastructures.`**LocalCache**(*limit=None*)
Dictionary with a finite number of keys.

Older items expires first.

class `celery.datastructures.`**PositionQueue**(*length*)
A positional queue of a specific length, with slots that are either filled or unfilled. When all of the positions are filled, the queue is considered `full()`.

Parameters **length** – see `length`.

length
The number of items required for the queue to be considered full.

class UnfilledPosition(*position*)
Describes an unfilled slot.

`PositionQueue.`**filled**
Returns the filled slots as a list.

`PositionQueue.`**full**()
Returns `True` if all of the slots has been filled.

class `celery.datastructures.`**SharedCounter**(*initial_value*)
Thread-safe counter.

Please note that the final value is not synchronized, this means that you should not update the value by using a previous value, the only reliable operations are increment and decrement.

Example

```
>>> max_clients = SharedCounter(initial_value=10)
```

Thread one >>> max_clients += 1 # OK (safe)

Thread two >>> max_clients -= 3 # OK (safe)

Main thread >>> if client >= int(max_clients): # Max clients now at 8 ... wait()

```
>>> max_client = max_clients + 10 # NOT OK (unsafe)
```

decrement(*n=1*)
Decrement value.

increment(*n=1*)
Increment value.

celery.datastructures.**consume_queue**(*queue*)
Iterator yielding all immediately available items in a `Queue.Queue`.

The iterator stops as soon as the queue raises `Queue.Empty`.

Example

```
>>> q = Queue()
>>> map(q.put, range(4))
>>> list(consume_queue(q))
[0, 1, 2, 3]
>>> list(consume_queue(q))
[]
```

8.6.28 Message Routers - celery.routes

class celery.routes.**MapRoute**(*map*)
Makes a router out of a `dict`.

route_for_task(*task*, **args*, ***kwargs*)

class celery.routes.**Router**(*routes=None*, *queues=None*, *create_missing=False*)

add_queue(*queue*)

expand_destination(*route*)

lookup_route(*task*, *args=None*, *kwargs=None*)

route(*options*, *task*, *args=()*, *kwargs={}*)

celery.routes.**prepare**(*routes*)
Expand ROUTES setting.

8.6.29 Logging - celery.log

8.6.30 SQLAlchemy Models - celery.db.models

8.6.31 SQLAlchemy Session - celery.db.session

8.6.32 Utilities - celery.utils

`celery.utils.`**`chunks`**(*it*, *n*)

Split an iterator into chunks with `n` elements each.

Examples

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.`**`first`**(*predicate*, *iterable*)

Returns the first element in `iterable` that `predicate` returns a `True` value for.

`celery.utils.`**`firstmethod`**(*method*)

Returns a functions that with a list of instances, finds the first instance that returns a value for the given method.

The list can also contain promises (`promise`.)

`celery.utils.`**`fun_takes_kwargs`**(*fun*, *kwlist=*[])

With a function, and a list of keyword arguments, returns arguments in the list which the function takes.

If the object has an `argspec` attribute that is used instead of using the `inspect.getargspec`() introspection.

Parameters

- **fun** – The function to inspect arguments of.
- **kwlist** – The list of keyword arguments.

Examples

```
>>> def foo(self, x, y, logfile=None, loglevel=None):
...     return x * y
>>> fun_takes_kwargs(foo, ["logfile", "loglevel", "task_id"])
["logfile", "loglevel"]

>>> def foo(self, x, y, **kwargs):
>>> fun_takes_kwargs(foo, ["logfile", "loglevel", "task_id"])
["logfile", "loglevel", "task_id"]
```

`celery.utils.`**`gen_unique_id`**()

Generate a unique id, having - hopefully - a very small chance of collission.

For now this is provided by `uuid.uuid4`().

`celery.utils.`**`get_cls_by_name`**(*name*, *aliases={}*)

Get class by name.

The name should be the full dot-separated path to the class:

```
modulename.ClassName
```

Example:

```
celery.concurrency.processes.TaskPool
                            ^- class name
```

If `aliases` is provided, a dict containing short name/long name mappings, the name is looked up in the aliases first.

Examples:

```
>>> get_cls_by_name("celery.concurrency.processes.TaskPool")
<class 'celery.concurrency.processes.TaskPool'>

>>> get_cls_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 >>> get_cls_by_name(TaskPool) is TaskPool True

`celery.utils.`**`get_full_cls_name`**(*cls*)
With a class, get its full module and class name.

`celery.utils.`**`instantiate`**(*name*, **args*, ***kwargs*)
Instantiate class by name.

See `get_cls_by_name()`.

`celery.utils.`**`is_iterable`**(*obj*)

`celery.utils.`**`kwdict`**(*kwargs*)
Make sure keyword arguments are not in unicode.

This should be fixed in newer Python versions, see: http://bugs.python.org/issue4978.

`celery.utils.`**`mattrgetter`**(**attrs*)
Like `operator.itemgetter()` but returns `None` on missing attributes instead of raising `AttributeError`.

`celery.utils.`**`maybe_promise`**(*value*)
Evaluates if the value is a promise.

`celery.utils.`**`mitemgetter`**(**items*)
Like `operator.itemgetter()` but returns `None` on missing items instead of raising `KeyError`.

class `celery.utils.`**`mpromise`**(*fun*, **args*, ***kwargs*)
Memoized promise.

The function is only evaluated once, every subsequent access will return the same value.

`evaluated`
Set to to `True` after the promise has been evaluated.

`evaluate`()

`evaluated` = **False**

`celery.utils.`**`noop`**(**args*, ***kwargs*)
No operation.

Takes any arguments/keyword arguments and does nothing.

`celery.utils.`**`padlist`**(*container*, *size*, *default=None*)
Pad list with default elements.

Examples:

```
>>> 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")
```

class celery.utils.**promise**(*fun*, **args*, ***kwargs*)

A promise.

Evaluated when called or if the `evaluate()` method is called. The function is evaluated on every access, so the value is not memoized (see `mpromise`).

Overloaded operations that will evaluate the promise: `__str__()`, `__repr__()`, `__cmp__()`.

evaluate()

celery.utils.**repeatlast**(*it*)

Iterate over all elements in the iterator, and when its exhausted yield the last value infinitely.

celery.utils.**retry_over_time**(*fun*, *catch*, *args=[]*, *kwargs={}*, *errback=<function noop at 0x4a54f50>*, *max_retries=None*, *interval_start=2*, *interval_step=2*, *interval_max=30*)

Retry the function over and over until max retries is exceeded.

For each retry we sleep a for a while before we try again, this interval is increased for every retry until the max seconds is reached.

Parameters

- **fun** – The function to try
- **catch** – Exceptions to catch, can be either tuple or a single exception class.
- **args** – Positional arguments passed on to the function.
- **kwargs** – Keyword arguments passed on to the function.
- **errback** – Callback for when an exception in `catch` is raised. The callback must take two arguments: `exc` and `interval`, where `exc` is the exception instance, and `interval` is the time in seconds to sleep next..
- **max_retries** – Maximum number of retries before we give up. If this is not set, we will retry forever.
- **interval_start** – How long (in seconds) we start sleeping between retries.
- **interval_step** – By how much the interval is increased for each retry.
- **interval_max** – Maximum number of seconds to sleep between retries.

8.6.33 Time and Date Utilities - celery.utils.timeutils

celery.utils.timeutils.**delta_resolution**(*dt*, *delta*)

Round a datetime to the resolution of a 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.timeutils.**rate**(*rate*)
Parses rate strings, such as `"100/m"` or `"2/h"` and converts them to seconds.

celery.utils.timeutils.**remaining**(*start*, *ends_in*, *now=None*, *relative=True*)
Calculate the remaining time for a start date and a timedelta.

E.g. "how many seconds left for 30 seconds after `start`?"

Parameters

- **start** – Start `datetime.datetime`.
- **ends_in** – The end delta as a `datetime.timedelta`.
- **relative** – If set to `False`, the end time will be calculated using `delta_resolution()` (i.e. rounded to the resolution

  of `ends_in`).
- **now** – The current time, defaults to `datetime.now()`.

celery.utils.timeutils.**timedelta_seconds**(*delta*)
Convert `datetime.timedelta` to seconds.

Doesn't account for negative values.

celery.utils.timeutils.**weekday**(*name*)
Return the position of a weekday (0 - 7, where 0 is Sunday).

Example:

```
>>> weekday("sunday"), weekday("sun"), weekday("mon")
(0, 0, 1)
```

8.6.34 Debugging Info - celery.utils.info

8.6.35 Python Compatibility - celery.utils.compat

class celery.utils.compat.**OrderedDict**(**args*, ***kwds*)
Dictionary that remembers insertion order

clear() → None. Remove all items from od.

copy() → a shallow copy of od

classmethod fromkeys(*S*[, *v*]) → New ordered dictionary with keys from S
and values equal to v (which defaults to None).

items()

iteritems()

iterkeys()

itervalues()

keys()

pop(*key*, *default=<object object at 0x3e52760>*)

popitem() -> (*k*, *v*)
Return and remove a (key, value) pair. Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault(*key*, *default=None*)

update(*other=()*, ***kwds*)

values()

celery.utils.compat.**chain_from_iterable**()
chain.from_iterable(iterable) –> chain object

Alternate chain() contructor taking a single iterable argument that evaluates lazily.

8.6.36 Sending E-mail - celery.utils.mail

class celery.utils.mail.**Mailer**(*host='localhost'*, *port=0*, *user=None*, *password=None*)

send(*message*)

class celery.utils.mail.**Message**(*to=None*, *sender=None*, *subject=None*, *body=None*, *charset='us-ascii'*)

exception celery.utils.mail.**SendmailWarning**
Problem happened while sending the e-mail message.

celery.utils.mail.**mail_admins**(*subject*, *message*, *fail_silently=False*)
Send a message to the admins in conf.ADMINS.

8.6.37 Compatibility Patches - celery.utils.patch

celery.utils.patch.**ensure_process_aware_logger**()

8.6.38 functools compat - celery.utils.functional

Functional utilities for Python 2.4 compatibility.

8.6.39 Signal Dispatch - celery.utils.dispatch

8.6.40 Signals: Dispatcher - celery.utils.dispatch.signal

Signal class.

class celery.utils.dispatch.signal.**Signal**(*providing_args=None*)
Base class for all signals

receivers
Internal attribute, holds a dictionary of
``{receriverkey (id): weakref(receiver)}`` mappings.

connect(*receiver*, *sender=None*, *weak=True*, *dispatch_uid=None*)
Connect receiver to sender for signal.

Parameters

- **receiver** – 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-referencable (more precisely `saferef.safe_ref()` must be able to create a reference to the receiver).

  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** – The sender to which the receiver should respond. Must either be of type `Signal`, or `None` to receive events from any sender.
- **weak** – 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** – An identifier used to uniquely identify a particular instance of a receiver. This will usually be a string, though it may be anything hashable.

disconnect(*receiver=None*, *sender=None*, *weak=True*, *dispatch_uid=None*)

Disconnect receiver from sender for signal.

If weak references are used, disconnect need not be called. The receiver will be removed from dispatch automatically.

Parameters

- **receiver** – The registered receiver to disconnect. May be none if `dispatch_uid` is specified.
- **sender** – The registered sender to disconnect.
- **weak** – The weakref state to disconnect.
- **dispatch_uid** – the unique identifier of the receiver to disconnect

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** – The sender of the signal. Either a specific object or `None`.
- ****named** – Named arguments which will be passed to receivers.

Returns a list of tuple pairs: `[(receiver, response), ... ]`.

send_robust(*sender*, ***named*)

Send signal from sender to all connected receivers catching errors.

Parameters

- **sender** – The sender of the signal. Can be any python object (normally one registered with a connect if you actually want something to occur).
- ****named** – Named arguments which will be passed to receivers. These arguments must be a subset of the argument names defined in `providing_args`.

Returns a list of tuple pairs: `[(receiver, response), ... ]`.

Raises DispatcherKeyError

if any receiver raises an error (specifically any subclass of `Exception`), the error instance is returned as the result for that receiver.

8.6.41 Signals: Safe References - celery.utils.dispatch.saferef

"Safe weakrefs", originally from pyDispatcher.

Provides a way to safely weakref any function, including bound methods (which aren't handled by the core weakref module).

class `celery.utils.dispatch.saferef.`**BoundMethodWeakref**(*target*, *on_delete=None*)

'Safe' and reusable weak references to instance methods.

BoundMethodWeakref objects provide a mechanism for referencing a bound method without requiring that the method object itself (which is normally a transient object) is kept alive. Instead, the BoundMethodWeakref object keeps weak references to both the object and the function which together define the instance method.

key

the identity key for the reference, calculated by the class's `calculate_key()` method applied to the target instance method

deletion_methods

sequence of callable objects taking single argument, a reference to this object which will be called when *either* the target object or target function is garbage collected (i.e. when this object becomes invalid). These are specified as the on_delete parameters of `safe_ref()` calls.

weak_self

weak reference to the target object

weak_func

weak reference to the target function

_all_instances

class attribute pointing to all live BoundMethodWeakref objects indexed by the class's `calculate_key(target)` method applied to the target objects. This weak value dictionary is used to short-circuit creation so that multiple references to the same (object, function) pair produce the same BoundMethodWeakref instance.

classmethod calculate_key(*target*)

Calculate the reference key for this reference

Currently this is a two-tuple of the `id()`'s of the target object and the target function respectively.

class `celery.utils.dispatch.saferef.`**BoundNonDescriptorMethodWeakref**(*target*, *on_delete=None*)

A specialized `BoundMethodWeakref`, for platforms where instance methods are not descriptors.

It assumes that the function name and the target attribute name are the same, instead of assuming that the function is a descriptor. This approach is equally fast, but not 100% reliable because functions can be stored on an attribute named differenty than the function's name such as in:

```
>>> class A(object):
...     pass

>>> def foo(self):
...     return "foo"
>>> A.bar = foo
```

But this shouldn't be a common use case. So, on platforms where methods aren't descriptors (such as Jython) this implementation has the advantage of working in the most cases.

`celery.utils.dispatch.saferef.`**get_bound_method_weakref**(*target*, *on_delete*)

Instantiates the appropiate `BoundMethodWeakRef`, depending on the details of the underlying class method implementation.

celery.utils.dispatch.saferef.**safe_ref**(*target*, *on_delete=None*)
Return a *safe* weak reference to a callable target

Parameters

- **target** – the object to be weakly referenced, if it's a bound method reference, will create a BoundMethodWeakref, otherwise creates a simple weakref.ref.
- **on_delete** – if provided, will have a hard reference stored to the callable to be called after the safe reference goes out of scope with the reference object, (either a weakref.ref or a BoundMethodWeakref) as argument.

8.6.42 Platform Specific - celery.platform

celery.platform.**ignore_signal**(*signal_name*)
Ignore signal using SIG_IGN.

Does nothing if the platform doesn't support signals, or the specified signal in particular.

celery.platform.**install_signal_handler**(*signal_name*, *handler*)
Install a handler.

Does nothing if the current platform doesn't support signals, or the specified signal in particular.

celery.platform.**reset_signal**(*signal_name*)
Reset signal to the default signal handler.

Does nothing if the platform doesn't support signals, or the specified signal in particular.

celery.platform.**set_mp_process_title**(*progname*, *info=None*)
Set the ps name using the multiprocessing process name.

Only works if setproctitle is installed.

celery.platform.**set_process_title**(*progname*, *info=None*)
Set the ps name for the currently running process.

Only works if :mod‘setproctitle‘ is installed.

CHAPTER 9

Change history

9.1 2.0.3

release-date 2010-08-27 12:00 P.M CEST

9.1.1 Fixes

- celeryd: Properly handle connection errors happening while closing consumers.
- celeryd: 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 namespace 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 http://github.com/ask/celery/issues/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 was not 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 did not work correctly if no destination was specified.
- Can now store result/metadata for custom states.
- celeryd: A warning is now emitted if the sending of task error e-mails fails.
- celeryev: Curses monitor no longer crashes if the terminal window is resized.

  See http://github.com/ask/celery/issues/issue/160
- celeryd: On OS X it is not possible to run `os.exec*` in a process that is threaded.

  This breaks the SIGHUP restart handler, and is now disabled on OS X, emitting a warning instead.

  See http://github.com/ask/celery/issues/issue/152

- `celery.execute.trace`: Properly handle `raise(str)`, which is still allowed in Python 2.4.

  See http://github.com/ask/celery/issues/issue/175

- Using urllib2 in a periodic task on OS X crashed because of the proxy autodetection used in OS X.

  This is now fixed by using a workaround. See http://github.com/ask/celery/issues/issue/143

- Debian init scripts: Commands should not run in a subshell

  See http://github.com/ask/celery/issues/issue/163

- Debian init scripts: Use abspath for celeryd to allow stat

  See http://github.com/ask/celery/issues/issue/162

9.1.2 Documentation

- getting-started/broker-installation: Fixed typo

  `set_permissions ""` -> `set_permissions ".*"`.

- Tasks Userguide: Added section on database transactions.

  See http://github.com/ask/celery/issues/issue/169

- Routing Userguide: Fixed typo `"feed": -> {"queue": "feeds"}`.

  See http://github.com/ask/celery/issues/issue/169

- Documented the default values for the `CELERYD_CONCURRENCY` and `CELERYD_PREFETCH_MULTIPLIER` settings.

- Tasks Userguide: 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.

9.2 2.0.2

release-date 2010-07-22 11:31 A.M CEST

- Routes: When using the dict route syntax, the exchange for a task could dissapear making the task unroutable.

  See http://github.com/ask/celery/issues/issue/158

- Test suite now passing on Python 2.4

- No longer have to type PYTHONPATH=. to use celeryconfig in current dir.

  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 cwd 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 behvavior will only apply to the modules imported in the config module, which I think is a good compromise (certainly better than just explictly setting PYTHONPATH=. anyway)

- Experimental Cassandra backend added.
- celeryd: SIGHUP handler accidentally propagated to worker pool processes.

  In combination with 7a7c44e39344789f11b5346e9cc8340f5fe4846c this would make each child process start a new celeryd when the terminal window was closed :/
- celeryd: Do not install SIGHUP handler if running from a terminal.

  This fixes the problem where celeryd is launched in the background when closing the terminal.
- celeryd: Now joins threads at shutdown.

  See http://github.com/ask/celery/issues/issue/152
- Test teardown: Don't use atexit but nose's `teardown()` functionality instead.

  See http://github.com/ask/celery/issues/issue/154
- Debian init script for celeryd: Stop now works correctly.
- Task logger: `warn` method added (synonym for `warning`)
- Can now define a whitelist of errors to send error e-mails for.

  Example:

  ```
  CELERY_TASK_ERROR_WHITELIST = ('myapp.MalformedInputError')
  ```

  See http://github.com/ask/celery/issues/issue/153
- celeryd: 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 loglevel 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 celeryd process, for use in functional testing.

9.3 2.0.1

release-date 2010-07-09 03:02 P.M CEST

- multiprocessing.pool: Now handles encoding errors, so that pickling errors doesn't crash the worker processes.
- The remote control command replies was not 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 did not 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://celeryproject.org/docs
- 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` was not honored by apply_async.
- New remote control command: `stats`

  Dumps information about the worker, like pool process pids, 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 is 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 celeryd 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.

9.4 2.0.0

release-date 2010-07-02 02:30 P.M CEST

9.4.1 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!

9.4.2 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:

  ```
  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.

9.4.3 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.

9.4.4 Backward incompatible changes

- Default (python) loader now prints warning on missing `celeryconfig.py` instead of raising `ImportError`.

  celeryd raises `ImproperlyConfigured` if the configuration is not 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 deprecation timeline):

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 deprecation timeline).
- It's no longer allowed to skip the class name in loader names. (as scheduled by the deprecation timeline):

  Assuming the implicit `Loader` class name is no longer supported, if you use e.g.:

  ```
  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 declaration:

  ```
  $ camqadm exchange.delete celeryresults
  ```

- Now uses pickle instead of cPickle on Python versions <= 2.5

  cPikle is broken in Python <= 2.5.

  It unsafely and incorrectly uses relative instead of absolute imports, so e.g:

  ```
  exceptions.KeyError
  ```

  becomes:

  ```
  celery.exceptions.KeyError
  ```

  Your best choice is to upgrade to Python 2.6, as while the pure pickle version has worse performance, it is the only safe option for older Python versions.

9.4.5 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:

  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*
  - *Sets of tasks, Subtasks and Callbacks*
  - *Routing Tasks*
- celeryd: Standard out/error is now being redirected to the logfile.
- `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`
- celeryd: Added `--purge` as an alias to `--discard`.
- celeryd: Ctrl+C (SIGINT) once does warm shutdown, hitting Ctrl+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*.

- celeryd: 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 http://github.com/ask/celery/issues/closed#issue/122

- New built-in way to do task callbacks using `subtask`.

  See *Sets of tasks, Subtasks and Callbacks* for more information.

- TaskSets can now contain several types of tasks.

  `TaskSet` has been refactored to use a new syntax, please see *Sets of tasks, Subtasks and Callbacks* for more information.

  The previous syntax is still supported, but will be deprecated in version 1.4.

- TaskSet failed() result was incorrect.

  See http://github.com/ask/celery/issues/closed#issue/132

- Now creates different loggers per task class.

  See http://github.com/ask/celery/issues/closed#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 `celeryd`:

  ```
  $ celeryd -Q video, image
  ```

  See the new routing section of the userguide 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 timelimits.

  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_SOFT_TASK_TIME_LIMIT

    Soft time limit. The celery.exceptions.SoftTimeLimitExceeded exception will be raised when this is exceeded. The task can catch this to e.g. 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 alltogether 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 multidb routers)
  - New setting: CELERY_ROUTES

This is a single, or a list of routers to traverse when sending tasks. Dicts 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:

```
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.
- celeryd: Added `CELERYD_MAX_TASKS_PER_CHILD` / *`--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: `worker_process_init`: Sent inside the pool worker process at init.
- celeryd *`-Q`* option: Ability to specifiy 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 celeryd only consume from the `image` and `video` queues:

  ```
  $ celeryd -Q image,video
  ```

- celeryd: New return value for the `revoke` control command:

  Now returns:

  ```
  {"ok": "task $id revoked"}
  ```

  instead of `True`.
- celeryd: 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 unittests doesn't require any special setup. `celery/tests/__init__` now configures the `CELERY_CONFIG_MODULE` and `CELERY_LOADER`, 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 contrib/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`.

- celeryd: 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

# Additionl 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 works 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.

9.5 1.0.6

release-date 2010-06-30 09:57 A.M CEST

- 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
  ```

9.6 1.0.5

release-date 2010-06-01 02:36 P.M CEST

9.6.1 Critical

- SIGINT/Ctrl+C killed the pool, abrubtly terminating the currently executing tasks.

  Fixed by making the pool worker processes ignore `SIGINT`.

- Should not close the consumers before the pool is terminated, just cancel the consumers.

  Issue #122. http://github.com/ask/celery/issues/issue/122

- Now depends on `billiard` >= 0.3.1
- celeryd: Previously exceptions raised by worker components could stall startup, now it correctly logs the exceptions and shuts down.
- celeryd: Prefetch counts was set too late. QoS is now set as early as possible, so celeryd can't slurp in all the messages at start-up.

9.6.2 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 unittests: `NOSE_VERBOSE` environment var now enables verbose output from Nose.
- `celery.execute.apply()`: Pass logfile/loglevel arguments as task kwargs.

  Issue #110 http://github.com/ask/celery/issues/issue/110

- celery.execute.apply: Should return exception, not `ExceptionInfo` on error.

  Issue #111 http://github.com/ask/celery/issues/issue/111

- Added new entries to the *FAQs*:
  - Should I use retry or acks_late?
  - Can I execute a task by name?

9.7 1.0.4

release-date 2010-05-31 09:54 A.M CEST

- Changlog merged with 1.0.5 as the release was never announced.

9.8 1.0.3

release-date 2010-05-15 03:00 P.M CEST

9.8.1 Important notes

- Messages are now acked *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 multiprocessing module. The previous behavior was not 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 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 was not really useful and made it harder to refactor some parts.
- Now depends on carrot >= 0.10.4
- Now depends on billiard >= 0.3.0

9.8.2 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 that this means the tasks may be executed twice if the worker crashes in the middle of their execution. Not acceptable for most applications, but desirable for others.

- Added crontab-like scheduling to periodic tasks.

  Like a cron job, you can specify units of time of when you would 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:

```
from celery.task.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 that this a late addition. While we have unittests, 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 is not a list, or kwargs is not a tuple (http://github.com/ask/celery/issues/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 behaviour 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 is 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!

9.8.3 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(panel, **kwargs):
    panel.listener.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}"}>']}]
  ```

9.8.4 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 was not properly applied (http://github.com/ask/celery/issues/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.

9.9 1.0.2

release-date 2010-03-31 12:50 P.M CET

- 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()
```

- Unittests: Don't disable the django test database teardown, instead fixed the underlying issue which was caused by modifications to the `DATABASE_NAME` setting (http://github.com/ask/celery/issues/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 would very much like to reuse the connection, but a safe number of reuses is not 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

- celeryd: The worker components are now configurable: `CELERYD_POOL`, `CELERYD_LISTENER`, `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_LISTENER = "celery.worker.listener.CarrotListener"
```

  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 `&&` (http://github.com/ask/celery/issues/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 was not properly flushed to disk if the schedule had not been properly initialized.
- celerybeat: Now syncs the schedule to disk when receiving the `SIGTERM` and `SIGINT` signals.
- Control commands: Make sure keywords arguments are not 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` (http://github.com/ask/celery/issues/83).
- PeriodicTask.delta_resolution: Was not working for days and hours, now fixed by rounding to the nearest day/hour.
- Fixed a potential infinite loop in `BaseAsyncResult.__eq__`, although there is no evidence that it has ever been triggered.
- celeryd: Now handles messages with encoding problems by acking them and emitting an error message.

9.10 1.0.1

release-date 2010-02-24 07:05 P.M CET

- Tasks are now acknowledged early instead of late.

  This is done because messages can only be acked within the same connection channel, so if the connection is lost we would have to refetch 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 has 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.

  Please note that 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.

- celeryd now shutdowns cleanly when receving the `TERM` signal.
- celeryd now does a cold shutdown if the `INT` signal is received (Ctrl+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 refetch already retrieved results using `backend.reload_task_result` + `backend.reload_taskset_result` (that's for those who want to send results incrementally).

- `celeryd` now works on Windows again.

Note that if running 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 does not 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, celeryd 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 are not yet persistent, this is done to make sure the task is revoked even though it's currently being hold because its eta is e.g. 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 was not passed by `.send()` (fixes the uuid keyerrors in celerymon)
- Added `--schedule`/`-s` option to celeryd, 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 `contrib/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 are 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 executeable. Does not modify `CELERYD` when using django with virtualenv.

9.11 1.0.0

release-date 2010-02-10 04:00 P.M CET

9.11.1 Backward incompatible changes

- Celery does not support detaching anymore, so you have to use the tools available on your platform, or something like supervisord to make celeryd/celerybeat/celerymon into background processes.

  We've had too many problems with celeryd daemonizing itself, so it was decided it has to be removed. Example startup scripts has been added to `contrib/`:

  - Debian, Ubuntu, (start-stop-daemon)

    `contrib/debian/init.d/celeryd` `contrib/debian/init.d/celerybeat`

  - Mac OS X launchd

    ```
    contrib/mac/org.celeryq.celeryd.plist
    contrib/mac/org.celeryq.celerybeat.plist
    contrib/mac/org.celeryq.celerymon.plist
    ```

  - Supervisord (http://supervisord.org)

    `contrib/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 celeryd loglevel is now `WARN`, to enable the previous log level start celeryd 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:

  ```
  from celery.decorators import task

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

  See the User Guide: *Tasks* for more information.

- The periodic task system has been rewritten to a centralized solution.

  This means `celeryd` 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 celeryd 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 behaviour set `CELERY_STORE_ERRORS_EVEN_IF_IGNORED` to `True`.
- The staticstics 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.
- `CELERY_LOADER` now needs loader class name in addition to module name,

  E.g. 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 is not 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()
  ```

9.11.2 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: http://bit.ly/aiWoH.

  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.

9.11.3 News

- Rate limiting support (per task type, or globally).
- New periodic task system.
- Automatic registration.
- New cool task decorator syntax.

- celeryd now sends events if enabled with the `-E` argument.

  Excellent for monitoring tools, one is already in the making (http://github.com/ask/celerymon).

  Current events include: worker-heartbeat, task-[received/succeeded/failed/retried], worker-online, worker-offline.

- You can now delete (revoke) tasks that has already been applied.
- You can now set the hostname celeryd identifies as using the `--hostname` argument.
- Cache backend now respects `CELERY_TASK_RESULT_EXPIRES`.
- Message format has been standardized and now uses ISO-8601 format for dates instead of datetime.
- `celeryd` now responds to the `HUP` signal by restarting itself.
- Periodic tasks are now scheduled on the clock.

  I.e. `timedelta(hours=1)` means every hour at :00 minutes, not every hour from the server starts. To revert to the previous behaviour you can set `PeriodicTask.relative = True`.

- Now supports passing execute options to a TaskSets list of args, e.g.:

```
>>> 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 tasksets are now cached by storing it in the result backend.

9.11.4 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).
- Loglevel for stdout/stderr changed from INFO to ERROR
- ImportErrors are now properly propogated when autodiscovering 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.
- `celery.exceptions.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).
- celeryd now correctly handles malformed messages by throwing away and acknowledging the message, instead of crashing.

9.11.5 Bugs

- Fixed a race condition that could happen while storing task results in the database.

9.11.6 Documentation

- Reference now split into two sections; API reference and internal module reference.

9.12 0.8.4

release-date 2010-02-05 01:52 P.M CEST

- Now emits a warning if the –detach argument is used. –detach should not be used anymore, as it has several not easily fixed bugs related to it. Instead, use something like start-stop-daemon, supervisord or launchd (os x).
- Make sure logger class is process aware, even if running Python >= 2.6.
- Error e-mails are not sent anymore when the task is retried.

9.13 0.8.3

release-date 2009-12-22 09:43 A.M CEST

- Fixed a possible race condition that could happen when storing/querying task results using the the database backend.
- Now has console script entry points in the setup.py file, so tools like buildout will correctly install the programs celerybin and celeryinit.

9.14 0.8.2

release-date 2009-11-20 03:40 P.M CEST

- QOS Prefetch count was not 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 cahnged.

9.15 0.8.1

release-date 2009-11-16 05:21 P.M CEST

9.15.1 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!

9.15.2 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

9.15.3 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 (curry) 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 db backend is used.

9.16 0.8.0

release-date 2009-09-22 03:06 P.M CEST

9.16.1 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 mailinglist or IRC channel listed in README 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 the case when using Django 1.1 or higher)
- Now depends on `carrot` version 0.6.0.
- Now depends on python-daemon 1.4.8

9.16.2 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 Can I use celery without Django?.

- Celery now supports task retries.

  See Cookbook: Retrying Tasks 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 http://github.com/ask/celery/issues/closed#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 celeryd logfile 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.

- celeryd 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`.

9.16.3 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 memcached 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 information.

- **`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 mimetype to "application/json"
- views.task_status now returns exception if status is RETRY
- **views.task_status now returns traceback if status 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 mikedizon

9.17 0.6.0

release-date 2009-08-07 06:54 A.M CET

9.17.1 Important changes

- **Fixed a bug where tasks raising unpickleable exceptions crashed pool** workers. So if you've had pool workers mysteriously dissapearing, or problems with celeryd 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, e.g.** "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.

9.17.2 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 celeryd** startup 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.
- **celeryd 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` loglevel (`--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 loglevel `ERROR`
- **Log message `"Got task from broker"` is now emitted for all tasks, even if** the task has an ETA (estimated time of arrival). Also the message now includes the ETA for the task (if any).
- **Acknowledgement 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 overrided pool behaviour (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!

9.18 0.4.1

release-date 2009-07-02 01:42 P.M CET

- Fixed a bug with parsing the message options (`mandatory`, `routing_key`, `priority`, `immediate`)

9.19 0.4.0

release-date 2009-07-01 07:29 P.M CET

- Adds eager execution. `celery.execute.apply``|``Task.apply` executes the function blocking until the task is done, for API compatiblity it returns an `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.

9.20 0.3.20

release-date 2009-06-25 08:42 P.M CET

- 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 the pidfile if it's stale.
- 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.task_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 `celeryd` 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 celeryd with the `--supervised` option (or alternatively `-S`).
- views.apply: View applying a task. Example:

```
http://e.com/celery/apply/task_name/arg1/arg2//?kwarg1=a&kwarg2=b
```

  NOTE Use with caution, preferably not make this publicly accessible without ensuring 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`.

9.21 0.3.7

release-date 2008-06-16 11:41 P.M CET

- **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 pidlock is stale.
- Now compatible with carrot 0.4.5
- Default AMQP connnection 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 e-mails sent to admins 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.
- `celeryd`: 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 `Ctrl-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 memcached/libmemcached.
- Tyrant Backend: Now re-establishes the connection for every task executed.

9.22 0.3.3

release-date 2009-06-08 01:07 P.M CET

- The `PeriodicWorkController` now sleeps for 1 second between checking for periodic tasks to execute.

9.23 0.3.2

release-date 2009-06-08 01:07 P.M CET

- celeryd: Added option `--discard`: Discard (delete!) all waiting messages in the queue.
- celeryd: The `--wakeup-after` option was not handled as a float.

9.24 0.3.1

release-date 2009-06-08 01:07 P.M CET

- 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`)

9.25 0.3.0

release-date 2009-06-08 12:41 P.M CET

NOTE 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 e-mails. 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 have 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 loglevel `ERROR` instead of `INFO`, and backtraces 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.
- **IMOPORTANT** 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.

9.26 0.2.0

release-date 2009-05-20 05:14 P.M CET

- 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.

9.27 0.2.0-pre3

release-date 2009-05-20 05:14 P.M CET

- *Internal release*. Improved handling of unpickled exceptions, `get_result` now tries to recreate something looking like the original exception.

9.28 0.2.0-pre2

release-date 2009-05-20 01:56 P.M CET

- Now handles unpickleable exceptions (like the dynimically generated subclasses of `django.core.exception.MultipleObjectsReturned`).

9.29 0.2.0-pre1

release-date 2009-05-20 12:33 P.M CET

- 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.

9.30 0.1.15

release-date 2009-05-19 04:13 P.M CET

- 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!

9.31 0.1.14

release-date 2009-05-19 01:08 P.M CET

- Fixed a syntax error in the `TaskSet` class. (No such variable `TimeOutError`).

9.32 0.1.13

release-date 2009-05-19 12:36 P.M CET

- 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`.

9.33 0.1.12

release-date 2009-05-18 04:38 P.M CET

- `delay_task()` etc. now returns `celery.task.AsyncResult` object, which lets you check the result and any failure that might have 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 are tasks distributed to the celery server. Example:

```
>>> 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 metadata cache and database backends, and added a new backend for Tokyo Tyrant. You can set the backend in your django settings file. e.g:

```
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.
```

9.34 0.1.11

release-date 2009-05-12 02:08 P.M CET

- The logging system was leaking file descriptors, resulting in servers stopping with the EMFILES (too many open files) error. (fixed)

9.35 0.1.10

release-date 2009-05-11 12:46 P.M CET

- 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.

9.36 0.1.8

release-date 2009-05-07 12:27 P.M CET

- Better test coverage
- More documentation
- celeryd doesn't emit `Queue is empty` message if `settings.CELERYD_EMPTY_MSG_EMIT_EVERY` is 0.

9.37 0.1.7

release-date 2009-04-30 1:50 P.M CET

- Added some unittests
- Can now use the database for task metadata (like if the task has been executed or not). Set `settings.CELERY_TASK_META`
- Can now run `python setup.py test` to run the unittests 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`.

9.38 0.1.6

release-date 2009-04-28 2:13 P.M CET

- 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 does't happen for pure functions yet, only `Task` classes.
- `autodiscover()` now works with zipped eggs.
- celeryd: Now adds curernt working directory to `sys.path` for convenience.
- The `run_every` attribute of `PeriodicTask` classes can now be a `datetime.timedelta()` object.
- celeryd: You can now set the `DJANGO_PROJECT_DIR` variable for `celeryd` 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 like e.g:

```
>>> {"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`.

9.39 0.1.0

release-date 2009-04-24 11:28 A.M CET

- Initial release

CHAPTER 10

Interesting Links

10.1 celery

- **IRC logs from `#celery` (Freenode):** http://botland.oebfare.com/logger/celery/

10.2 AMQP

- *RabbitMQ-shovel*: Message Relocation Equipment (as a plug-in to RabbitMQ)
- Shovel: An AMQP Relay (generic AMQP shovel)

10.3 RabbitMQ

- **Trixx: Administration and Monitoring tool for RabbitMQ (in** development).
- **Cony: HTTP based service for providing insight into running** RabbitMQ processes.
- **RabbitMQ Munin Plug-ins: Use Munin to monitor RabbitMQ, and alert** on critical events.

CHAPTER 11

Indices and tables

- *genindex*
- *modindex*
- *search*

C