Celery Documentation

Release 1.0.6 (stable)

Ask Solem

February 04, 2014

Contents:

CHAPTER 1

Getting Started

Release 1.0

Date February 04, 2014

1.1 Introduction

Version 1.0.6

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, django, python, webhooks, queue, distributed

–

Celery is a 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 was originally created for use with Django, but is now usable from any Python project. It can also operate with other languages via webhooks.

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

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

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 are in *Executing Tasks*.

1.3.2 Configuration

Celery is configured by using a configuration module. By convention, this module is called `celeryconfig.py`. This module must be in the Python path so it can be imported.

You can set a custom name for the configuration module with the `CELERY_CONFIG_MODULE` variable. 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"
   ```

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:

```
$ PYTHONPATH="." 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:

```
$  PYTHONPATH="." 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. See *Executing Tasks* for more information.

```
>>> 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 celery 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 celery 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 and get its return value (or exception if the task failed).

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

```
>>> 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 First steps with Django

1.4.1 Configuring your Django project to use Celery

You only need three simple steps to use celery with your Django project.

1. Add `celery` to `INSTALLED_APPS`.
2. Create the celery database tables:

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

3. **Configure celery to use the AMQP user and virtual host we created** before, by adding the following to your `settings.py`:

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

That's it.

There are more options available, like how many processes you want to work in parallel (the `CELERY_CONCURRENCY` setting). You can also configure the backend used for storing task statuses. For now though, this should do. For all of the options available, please see the *configuration directive reference*.

Note: If you're using SQLite as the Django database back-end, `celeryd` will only be able to process one task at a time, this is because SQLite doesn't allow concurrent writes.

1.4.2 Running the celery worker server

To test this we'll be running the worker server in the foreground, so we can see what's going on without consulting the logfile:

```
$ python manage.py celeryd
```

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. See *daemon mode reference*.

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

```
$ python manage.py help celeryd
```

1.4.3 Defining and executing tasks

Please note: All the tasks have to be stored in a real module, they can't be defined in the python shell or ipython/bpython. This is because the celery worker server needs access to the task function to be able to run it. Put them in the `tasks` module of your Django application. The worker server will automatically load any `tasks.py` file for all of the applications listed in `settings.INSTALLED_APPS`. Executing tasks using `delay` and `apply_async`

can be done from the python shell, but keep in mind that since arguments are pickled, you can't use custom classes defined in the shell session.

This is a task that adds two numbers:

```
from celery.decorators import task

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

To execute this 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. See *Executing Tasks* for more information.

```
>>> from myapp.tasks import MyTask
>>> MyTask.delay(some_arg="foo")
```

At this point, the task has been sent to the message broker. The message broker will hold on to the task until a celery 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 celery 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 and get its return value (or exception if the task failed).

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

```
>>> 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.

1.5 Periodic Tasks

You can schedule tasks to run at intervals like `cron`. Here's an example of a periodic task:

```
from celery.task import PeriodicTask
from celery.registry import tasks
from datetime import timedelta

class MyPeriodicTask(PeriodicTask):
    run_every = timedelta(seconds=30)

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

```
        logger.info("Running periodic task!")
>>> tasks.register(MyPeriodicTask)
```

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 `crontab` to set the `run_every` property:

```
from celery.task import PeriodicTask
from celery.task.schedules import crontab

class EveryMondayMorningTask(PeriodicTask):
    run_every = crontab(hour=7, minute=30, day_of_week=1)

    def run(self, **kwargs):
        logger = self.get_logger(**kwargs)
        logger.info("Execute every Monday at 7:30AM.")
```

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
```

or if using Django:

```
$ python manage.py celerybeat
```

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

```
$ celeryd -B
```

or if using Django:

```
$ python manage.py celeryd  -B
```

1.6 Resources

1.6.1 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.

1.6.2 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/

1.6.3 Wiki

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

1.6.4 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.

1.6.5 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 1.0

Date February 04, 2014

2.1 Tasks

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 the `Task` class does:

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

An alternative way to use the decorator is to give the function as an argument instead, but if you do this be sure to set the resulting tasks `__name__` attribute, so pickle is able to find it in reverse:

```
create_user_task = task()(create_user)
create_user_task.__name__ = "create_user_task"
```

2.1.1 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 `self.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.

2.1.2 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.

2.1.3 Retrying a task if something fails

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

2.1.4 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.execptions.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

- **routing_key** Override the global default `routing_key` for this task.
- **exchange** Override the global default `exchange` 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.

2.1.5 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
```

2.1.6 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':
  <celery.task.builtins.DeleteExpiredTaskMetaTask object at 0x101d1f510>,
'celery.execute_remote':
  <celery.task.base.ExecuteRemoteTask object at 0x101d17890>,
```

```
'celery.task.rest.RESTProxyTask':
  <celery.task.rest.RESTProxyTask object at 0x101d1f410>,
'celery.task.rest.Task': <celery.task.rest.Task object at 0x101d1f4d0>,
'celery.map_async':
  <celery.task.base.AsynchronousMapTask object at 0x101d17910>,
'celery.ping': <celery.task.builtins.PingTask object at 0x101d1f550>}
```

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. If using Django it loads all `tasks.py` modules for the applications listed in `INSTALLED_APPS`. If you want to do something special you can create your own loader to do what you want.

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.

2.1.7 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:

```
from functools import curry

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

@task(ignore_result=True)
def fetch_page(url, callback=None):
    page = myparser.parse_document(page)
    if callback:
        callback(page)

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

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

2.1.8 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.

2.2 Executing Tasks

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"})
```

But `delay` 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
```

2.2.1 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)
```

2.2.2 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")
```

2.2.3 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`:

```
from celery.messaging import establish_connection

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

results = []
connection = establish_connection()
try:
    for args in numbers:
        res = add.apply_async(args=args, connection=connection)
        results.append(res)
finally:
    connection.close()

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

In Python 2.5 and above, you can use the `with` statement:

```
from __future__ import with_statement
from celery.messaging import establish_connection

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

results = []
with establish_connection() as connection:
    for args in numbers:
        res = add.apply_async(args=args, connection=connection)
        results.append(res)

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:

```
connection = establish_connection(connect_timeout=3)
```

2.2.4 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"`.

2.2.5 AMQP options

- 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 HTTP Callback Tasks (Webhooks)

2.3.1 Executing tasks on a web server

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."}
```

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")
```

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.

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.

2.4 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.

2.4.1 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":
 "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.

TODO Mindblowing one-line simple explanation here. TODO

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 (unless the queue's `auto_declare` setting is set)

Here's an example queue configuration with three queues; One for video, one for images and 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 a 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.
```

CHAPTER 3

Configuration and defaults

This document describes the configuration options available.

If you're using celery in a Django project these settings should be defined in the project's `settings.py` file.

In a regular Python environment, that is 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.

```
CELERY_RESULT_BACKEND = "database"
DATABASE_ENGINE = "sqlite3"
DATABASE_NAME = "mydatabase.db"

BROKER_HOST = "localhost"
BROKER_PORT = 5672
BROKER_VHOST = "/"
BROKER_USER = "guest"
BROKER_PASSWORD = "guest"

## 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 = 8

# 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 the Django ORM.
  - **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 the Django ORM database settings documentation: http://docs.djangoproject.com/en/dev/ref/settings/#database-engine

If you use this backend, make sure to initialize the database tables after configuration. When using celery with a Django project this means executing:

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

When using celery in a regular Python environment you have to execute:

```
$ celeryinit
```

3.4.1 Example configuration

```
CELERY_RESULT_BACKEND = "database"
DATABASE_ENGINE = "mysql"
DATABASE_USER = "myusername"
DATABASE_PASSWORD = "mypassword"
DATABASE_NAME = "mydatabase"
DATABASE_HOST = "localhost"
```

3.5 AMQP backend settings

The AMQP backend does not have any settings yet.

3.5.1 Example configuration

CELERY_RESULT_BACKEND = “amqp”

3.6 Cache backend settings

Please see the documentation for the Django cache framework settings: http://docs.djangoproject.com/en/dev/topics/cache/#memcached

To use a custom cache backend for Celery, while using another for Django, you should use the `CELERY_CACHE_BACKEND` setting instead of the regular django `CACHE_BACKEND` setting.

3.6.1 Example configuration

Using a single memcached server:

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

Using multiple memcached servers:

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

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

  Name of the database to use. Default is `celery_results`.

- 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_DATABASE = "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`.

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_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 are deleted.

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

3.12 Worker: celeryd

- **CELERY_IMPORTS** A sequence of modules to import when the celery daemon starts. This is useful to add tasks if you are not using django or cannot use task auto-discovery.
- **CELERY_SEND_EVENTS** Send events so the worker can be monitored by tools like `celerymon`.
- **CELERY_SEND_TASK_ERROR_EMAILS** If set to `True`, errors in tasks will be sent to admins by e-mail. If unset, it will send the e-mails if `settings.DEBUG` is False.
- **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 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. Can be overridden using the `--loglevel` option to `celeryd`.

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

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

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

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

- contrib/debian/init.d/

4.2.2 supervisord

- contrib/supervisord/

4.2.3 launchd (OS X)

- contrib/mac/

4.3 Unit Testing

4.3.1 Testing with Django

The problem that you'll first run in to when trying to write a test that runs a task is that Django's test runner doesn't use the same database that your celery daemon is using. If you're using the database backend, this means that your tombstones won't show up in your test database and you won't be able to check on your tasks to get the return value or check the status.

There are two ways to get around this. You can either take advantage of `CELERY_ALWAYS_EAGER = True` to skip the daemon, or you can avoid testing anything that needs to check the status or result of a task.

4.3.2 Using a custom test runner to test with celery

If you're going the `CELERY_ALWAYS_EAGER` route, which is probably better than just never testing some parts of your app, a custom Django test runner does the trick. Celery provides a simple test runner, but it's easy enough to roll your own if you have other things that need to be done. http://docs.djangoproject.com/en/dev/topics/testing/#defining-a-test-runner

For this example, we'll use the `celery.contrib.test_runner` to test the `add` task from the *User Guide: Tasks* examples.

To enable the test runner, set the following settings:

```
TEST_RUNNER = 'celery.contrib.test_runner.run_tests'
```

Then we can write our actually test in a `tests.py` somewhere:

```
from django.test import TestCase
from myapp.tasks import add

class AddTestCase(TestCase):

    def testNoError(self):
        """Test that the ``add`` task runs with no errors,
        and returns the correct result."""
        result = add.delay(8, 8)

        self.assertEquals(result.get(), 16)
        self.assertTrue(result.successful())
```

This test assumes that you put your example `add` task in `maypp.tasks` so of course adjust the import for wherever you actually put the class.

This page contains common recipes and techniques. Whenever a setting is mentioned, you should use `celeryconf.py` if using regular Python, or `settings.py` if running under Django.

CHAPTER 5

Tutorials

Release 1.0

Date February 04, 2014

5.1 External tutorials and resources

5.1.1 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

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/

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/

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

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

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/

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

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
   ```

   Or if you're not using django, but the default loader instead run `celeryinit`:

   ```
   $ celeryinit
   ```

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 carrot.connection import DjangoBrokerConnection
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 = DjangoBrokerConnection()
    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 = DjangoBrokerConnection()
    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.

You can use all of the features without using Django.

6.2.3 Why is Django a dependency?

Celery uses the Django ORM for database access when using the database result backend, the Django cache framework when using the cache result backend, and the Django signal dispatch mechanisms for signaling.

This doesn't mean you need to have a Django project to use celery, it just means that sometimes we use internal Django components.

The long term plan is to replace these with other solutions, (e.g. SQLAlchemy as the ORM, and louie, for signaling). The celery distribution will be split into two:

- celery

  The core. Using SQLAlchemy for the database backend.

- django-celery

  Celery integration for Django, using the Django ORM for the database backend.

We're currently seeking people with SQLAlchemy experience, so please contact the project if you want this done sooner.

The reason for the split is for purity only. It shouldn't affect you much as a user, so please don't worry about the Django dependency, just have a good time using celery.

6.2.4 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.

6.2.5 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: Did you register the task in the applications `tasks.py` module? (or in some other module Django loads by default, like `models.py`?). Also 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`.

6.3.11 Windows: The `-B` / `--beat` option to celeryd doesn't work?

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

6.4 Tasks

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

Answer: See *Executing Tasks*.

6.4.2 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>
```

6.5 Results

6.5.1 How dow 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()
```

6.6 Brokers

6.6.1 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.

6.6.2 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 like RabbitMQ has. If you have any problems with using STOMP and celery, please report the bugs to the issue tracker:

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

First you have to use the `master` branch of `celery`:

```
$ git clone git://github.com/ask/celery.git
$ cd celery
$ sudo python setup.py install
$ cd ..
```

Then you need to install the `stompbackend` branch of `carrot`:

```
$ git clone git://github.com/ask/carrot.git
$ cd carrot
$ git checkout stompbackend
$ sudo python setup.py install
$ cd ..
```

And my fork of `python-stomp` which adds non-blocking support:

```
$ hg clone http://bitbucket.org/asksol/python-stomp/
$ 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: For ActiveMQ the queue name has to have `"/queue/"` prepended to it. i.e. the queue `celery` becomes `/queue/celery`.

Since a STOMP queue is a single named entity and it doesn't have the routing capabilities of AMQP you need to set both the `queue`, and `exchange` settings to your queue name. This is a minor inconvenience since carrot needs to maintain the same interface for both AMQP and STOMP (obviously the one with the most capabilities won).

Use the following specific settings in your `settings.py`:

```
# Makes python-stomp the default backend for carrot.
CARROT_BACKEND = "stomp"

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

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

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

Now you can go on reading the tutorial in the README, ignoring any AMQP specific options.

6.6.3 What features are not supported when using 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

6.7 Features

6.7.1 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:
        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, ))


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

6.7.2 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)
```

6.7.3 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.

6.7.4 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.

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:

- Servers `x` and `y`: settings.py:

```
CELERY_DEFAULT_QUEUE = "regular_tasks"
CELERY_QUEUES = {
    "regular_tasks": {
        "binding_key": "task.#",
    },
}
CELERY_DEFAULT_EXCHANGE = "tasks"
CELERY_DEFAULT_EXCHANGE_TYPE = "topic"
CELERY_DEFAULT_ROUTING_KEY = "task.regular"
```

- Server `z`: settings.py:

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

`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.

Now to make a Task run on the `z` server you need to set its `routing_key` attribute so it starts with the words `"task.feed."`:

```
from feedaggregator.models import Feed
from celery.decorators import task

@task(routing_key="feed.importer")
def import_feed(feed_url):
    Feed.objects.import_feed(feed_url)
```

or if subclassing the `Task` class directly:

```
class FeedImportTask(Task):
    routing_key = "feed.importer"
```

```
    def run(self, feed_url):
        Feed.objects.import_feed(feed_url)
```

You can also override this using the `routing_key` argument to `celery.task.apply_async()`:

```
>>> from myapp.tasks import RefreshFeedTask
>>> RefreshFeedTask.apply_async(args=["http://cnn.com/rss"],
...                             routing_key="feed.importer")
```

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. Just add a new queue to server `z`'s `CELERY_QUEUES`:

```
CELERY_QUEUES = {
    "feed_tasks": {
        "binding_key": "feed.#",
    },
    "regular_tasks": {
        "binding_key": "task.#",
    },
}
```

Since the default exchange is `tasks`, they will both use the same exchange.

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 and RabbitMQ. Rabbits and Warrens is 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 also be useful as a source of information.

6.7.5 Can I use celery without Django?

Answer: Yes.

Celery uses something called loaders to read/setup configuration, import modules that register tasks and to decide what happens when a task is executed. Currently there are two loaders, the default loader and the Django loader. If you want to use celery without a Django project, you either have to use the default loader, or write a loader of your own.

The rest of this answer describes how to use the default loader.

While it is possible to use Celery from outside of Django, we still need Django itself to run, this is to use the ORM and cache-framework. Duplicating these features would be time consuming and mostly pointless, so while me might rewrite these in the future, this is a good solution in the mean time. Install Django using your favorite install tool, `easy_install`, `pip`, or whatever:

```
# easy_install django # as root
```

You need a configuration file named `celeryconfig.py`, either in the directory you run `celeryd` in, or in a Python library path where it is able to find it. The configuration file can contain any of the settings described in `celery.conf`. In addition; if you're using the database backend you have to configure the database. Here is an example configuration using the database backend with MySQL:

```
# Broker configuration
BROKER_HOST = "localhost"
BROKER_PORT = "5672"
BROKER_VHOST = "celery"
BROKER_USER = "celery"
BROKER_PASSWORD = "celerysecret"
CARROT_BACKEND="amqp"

# Using the database backend.
CELERY_RESULT_BACKEND = "database"
DATABASE_ENGINE = "mysql" # see Django docs for a description of these.
DATABASE_NAME = "mydb"
DATABASE_HOST = "mydb.example.org"
DATABASE_USER = "myuser"
DATABASE_PASSWORD = "mysecret"

# Number of processes that processes tasks simultaneously.
CELERYD_CONCURRENCY = 8

# Modules to import when celeryd starts.
# This must import every module where you register tasks so celeryd
# is able to find and run them.
CELERY_IMPORTS = ("mytaskmodule1", "mytaskmodule2")
```

With this configuration file in the current directory you have to run `celeryinit` to create the database tables:

```
$ celeryinit
```

At this point you should be able to successfully run `celeryd`:

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

and send a task from a python shell (note that it must be able to import `celeryconfig.py`):

```
>>> from celery.task.builtins import PingTask
>>> result = PingTask.apply_async()
>>> result.get()
'pong'
```

6.7.6 The celery test-suite is failing

Answer: If you're running tests from your Django project, and the celery test suite is failing in that context, then follow the steps below. If the celery tests are failing in another context, please report an issue to our issue tracker at GitHub:

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

That Django is running tests for all applications in `INSTALLED_APPS` by default is a pet peeve for many. You should use a test runner that either

1. Explicitly lists the apps you want to run tests for, or

2. Make a test runner that skips tests for apps you don't want to run.

For example the test runner that celery is using:

http://bit.ly/NVKep

To use this test runner, add the following to your `settings.py`:

```
TEST_RUNNER = "celery.tests.runners.run_tests"
TEST_APPS = (
    "app1",
    "app2",
    "app3",
    "app4",
)
```

Or, if you just want to skip the celery tests:

```
INSTALLED_APPS = (.....)
TEST_RUNNER = "celery.tests.runners.run_tests"
TEST_APPS = filter(lambda k: k != "celery", INSTALLED_APPS)
```

6.7.7 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(...)
```

6.7.8 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.

6.7.9 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.

6.7.10 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")
```

6.7.11 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`).

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

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

6.8 Django

6.8.1 Generating a template in a task doesn't seem to respect my i18n settings?

Answer: To enable the Django translation machinery you need to activate it with a language. **Note**: Be sure to reset to the previous language when done.

```
>>> from django.utils import translation
```

```
>>> prev_language = translation.get_language()
>>> translation.activate(language)
>>> try:
...     render_template()
... finally:
        translation.activate(prev_language)
```

The common pattern here would be for the task to take a `language` argument:

```
from celery.decorators import task

from django.utils import translation
from django.template.loader import render_to_string

@task()
def generate_report(template="report.html", language=None):
    prev_language = translation.get_language()
    language and translation.activate(language)
    try:
        report = render_to_string(template)
    finally:
        translation.activate(prev_language)
    save_report_somewhere(report)
```

CHAPTER 7

API Reference

Release 1.0

Date February 04, 2014

7.1 Task Decorators - celery.decorators

7.2 Defining Tasks - celery.task.base

7.3 Executing Tasks - celery.execute

7.4 Task Result - celery.result

7.5 Task Information and Utilities - celery.task

7.6 Configuration - celery.conf

QUEUES
Queue name/options mapping.

DEFAULT_QUEUE
Name of the default queue.

DEFAULT_EXCHANGE
Default exchange.

DEFAULT_EXCHANGE_TYPE
Default exchange type.

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

DEFAULT_ROUTING_KEY
Default routing key used when sending tasks.

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

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".

BROADCAST_EXCHANGE
Name of the exchange used for broadcast messages.

Default is "celeryctl".

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

EVENT_QUEUE
Name of queue used to listen for event messages. Default is "celeryevent".

EVENT_EXCHANGE
Exchange used to send event messages. Default is "celeryevent".

EVENT_EXCHANGE_TYPE
Exchange type used for the event exchange. Default is "topic".

EVENT_ROUTING_KEY
Routing key used for events. Default is "celeryevent".

EVENT_SERIALIZER
Type of serialization method used to serialize events. Default is "json".

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

CELERY_SEND_TASK_ERROR_EMAILS
If set to True, errors in tasks will be sent to admins by e-mail. If unset, it will send the e-mails if settings.DEBUG is True.

ALWAYS_EAGER
Always execute tasks locally, don't send to the queue.

TASK_RESULT_EXPIRES
Task tombstone expire time in seconds.

IGNORE_RESULT
If enabled, the default behavior will be to not store task results.

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

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

STORE_ERRORS_EVEN_IF_IGNORED
If enabled, task errors will be stored even though Task.ignore_result is enabled.

MAX_CACHED_RESULTS
Total number of results to store before results are evicted from the result cache.

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

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.

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`.

RESULT_BACKEND
The backend used to store task results (tombstones).

CELERY_CACHE_BACKEND
Use a custom cache backend for celery. If not set the django-global cache backend in `CACHE_BACKEND` will be used.

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

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

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

CELERYBEAT_LOG_LEVEL
Default log level for celerybeat. Default is: `INFO`.

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

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

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).

CELERYMON_LOG_LEVEL
Default log level for celerymon. Default is: `INFO`.

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

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

CELERYD_LOG_FORMAT
The format to use for log messages.

CELERYD_TASK_LOG_FORMAT
The format to use for task log messages.

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

CELERYD_LOG_LEVEL
Default log level for daemons. (`WARN`)

CELERYD_CONCURRENCY
The number of concurrent worker processes. If set to `0`, the total number of available CPUs/cores will be used.

CELERYD_PREFETCH_MULTIPLIER
The number of concurrent workers is multipled by this number to yield the wanted AMQP QoS message prefetch count.

CELERYD_POOL
Name of the task pool class used by the worker. Default is `"celery.worker.pool.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"`.

7.7 Remote Management of Workers - celery.task.control

7.8 HTTP Callback Tasks - celery.task.http

7.9 Periodic Task Schedule Behaviors - celery.task.schedules

7.10 Signals - celery.signals

task_sent
Triggered 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.

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).

task_prerun
Triggered before a task is executed.

Provides arguments:

- **task_id** Id of the task to be executed.
- **task** The task being executed.
- **args** the tasks positional arguments.
- **kwargs** The tasks keyword arguments.

task_postrun
Triggered after a task has been 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_init
Triggered before the worker is started.

worker_ready
Triggered when the worker is ready to accept work.

worker_shutdown
Triggered when the worker is about to shut down.

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

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

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

7.13 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.

7.14 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.

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.

7.15 Default Loader - celery.loaders.default

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

See the FAQ for example usage.

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.

7.16 Django Loader - celery.loaders.djangoapp

class celery.loaders.djangoapp.**Loader**
The Django loader.

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

Does everything necessary for Django to work in a long-living, multiprocessing environment.

on_worker_init()
Called when the worker starts.

Automatically discovers any tasks.py files in the applications listed in INSTALLED_APPS.

read_configuration()
Load configuration from Django settings.

celery.loaders.djangoapp.**autodiscover**()
Include tasks for all applications in INSTALLED_APPS.

celery.loaders.djangoapp.**find_related_module**(*app*, *related_name*)
Given an application name and a module name, tries to find that module in the application.

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.**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

Task 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.**RETRY = 'RETRY'**

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

7.19 Messaging - celery.messaging

7.20 Contrib: Test runner - celery.contrib.test_runner

celery.contrib.test_runner.**run_tests**(*test_labels*, **args*, ***kwargs*)
Django test runner allowing testing of celery delayed tasks.

All tasks are run locally, not in a worker.

To use this runner set `settings.TEST_RUNNER`:

```
TEST_RUNNER = "celery.contrib.test_runner.run_tests"
```

7.21 Contrib: Abortable tasks - celery.contrib.abortable

7.22 Django Views - celery.views

7.23 Events - celery.events

7.24 Celery Worker Daemon - celery.bin.celeryd

7.25 Celery Periodic Task Server - celery.bin.celerybeat

7.26 Celery Initialize - celery.bin.celeryinit

7.27 caqmadm: AMQP API Command-line Shell - celery.bin.camqadm

CHAPTER 8

Internals

Release 1.0

Date February 04, 2014

8.1 Celery Deprecation Timeline

- 1.2
  - 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

NOTE This describes the internals of the development version, not the current release.

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.

8.2.1 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.

8.2.2 Components

CarrotListener

Receives messages from the broker using `carrot`.

When a message is received it's converted into a `celery.worker.job.TaskWrapper` 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.TaskWrapper.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

- **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.

8.3.1 Example

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"}
```

8.3.2 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

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-accepted(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-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

8.5.1 Worker

celery.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.

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

- celery.worker.registry
- celery.worker.builtins

8.5.2 Tasks

celery.decorators

celery.registry

celery.task

celery.task.base

celery.task.http

celery.task.rest

Backward compatible interface to `celery.task.http`. Will be deprecated in future versions.

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 Django-specific

celery.models

celery.managers

celery.views

celery.urls

celery.management

8.5.6 Result backends

celery.backends

celery.backends.base

celery.backends.amqp

celery.backends.database

8.5.7 Loaders

celery.loaders

Loader autodetection, and working with the currently selected loader.

celery.loaders.base - Loader base classes

celery.loaders.default - The default loader

celery.loaders.djangoapp - The Django loader

8.5.8 CeleryBeat

celery.beat

8.5.9 Events

celery.events

8.5.10 Logging

celery.log

celery.utils.patch

8.5.11 Configuration

celery.conf

8.5.12 Miscellaneous

celery.datastructures

celery.exceptions

celery.platform

celery.utils

celery.utils.info

celery.utils.compat

8.6 Internal Module Reference

Release 1.0

Date February 04, 2014

8.6.1 Multiprocessing Worker - celery.worker

8.6.2 Worker Message Listener - celery.worker.listener

8.6.3 Executable Jobs - celery.worker.job

8.6.4 Worker Controller Threads - celery.worker.controllers

8.6.5 Token Bucket (rate limiting) - celery.worker.buckets

8.6.6 Worker Scheduler - celery.worker.scheduler

8.6.7 Task Pool - celery.worker.pool

8.6.8 Worker Heartbeats - celery.worker.heartbeat

8.6.9 Worker Control - celery.worker.control

8.6.10 Built-in Remote Control Commands - celery.worker.control.builtins

8.6.11 Remote Control Command Registry - celery.worker.control.registry

8.6.12 Worker Revoked Tasks - celery.worker.revoke

revoked

A `celery.datastructures.LimitedSet` containing revoked task ids.

Items expire after one hour, and the structure can only hold 10000 expired items at a time (about 300kb).

8.6.13 Clock Service - celery.beat

8.6.14 Backends - celery.backends

8.6.15 Backend: Base - celery.backends.base

8.6.16 Backend: AMQP - celery.backends.amqp

8.6.17 Backend: Database - celery.backends.database

8.6.18 Backend: Cache - celery.backends.cache

8.6.19 Backend: MongoDB - celery.backends.mongodb

8.6.20 Backend: Redis - celery.backends.pyredis

8.6.21 Backend: Tokyo Tyrant - celery.backends.tyrant

8.6.22 Tracing Execution - celery.execute.trace

8.6.23 Datastructures - celery.datastructures

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.

`first`
Get the oldest member.

`pop_value`(*value*)
Remove membership by finding value.

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))
[]
```

8.6.24 Logging - celery.log

8.6.25 Multiprocessing Worker - celery.worker

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.**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.worker.pool.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.worker.pool.TaskPool")
<class 'celery.worker.pool.TaskPool'>

>>> get_cls_by_name("default", {
...     "default": "celery.worker.pool.TaskPool"})
<class 'celery.worker.pool.TaskPool'>
```

Does not try to look up non-string names. >>> from celery.worker.pool 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.**mattrgetter**(**attrs*)
Like `operator.itemgetter()` but returns `None` on missing attributes instead of raising `AttributeError`.

celery.utils.**mitemgetter**(**items*)
Like `operator.itemgetter()` but returns `None` on missing items instead of raising `KeyError`.

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", "Constanza", "NYC"], 3)
("George", "Constanza", "NYC")
>>> first, last, city = padlist(["George", "Constanza"], 3)
("George", "Constanza", None)
>>> first, last, city, planet = padlist(["George", "Constanza",
                                         "NYC"], 4, default="Earth")
("George", "Constanza", "NYC", "Earth")
```

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 0x41a5a28>*, *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.

8.6.26 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.

Examples:

```
>>> now = datetime.now()
>>> now
datetime.datetime(2010, 3, 30, 11, 50, 58, 41065)
>>> delta_resolution(now, timedelta(days=2))
datetime.datetime(2010, 3, 30, 0, 0)
>>> delta_resolution(now, timedelta(hours=2))
datetime.datetime(2010, 3, 30, 11, 0)
>>> delta_resolution(now, timedelta(minutes=2))
datetime.datetime(2010, 3, 30, 11, 50)
>>> delta_resolution(now, timedelta(seconds=2))
datetime.datetime(2010, 3, 30, 11, 50, 58, 41065)
```

`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()`.

Examples:

```
>>> remaining(datetime.now(), ends_in=timedelta(seconds=30))
'0:0:29.999948'

>>> str(remaining(datetime.now() - timedelta(minutes=29),
        ends_in=timedelta(hours=2)))
```

```
'1:30:59.999938'

>>> str(remaining(datetime.now() - timedelta(minutes=29),
        ends_in=timedelta(hours=2),
        relative=False))
'1:11:18.458437'
```

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).

```
>>> weekday("sunday")
0
>>> weekday("sun")
0
>>> weekday("mon")
1
```

8.6.27 Debugging Info - celery.utils.info

8.6.28 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).

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.

8.6.29 Compatibility Patches - celery.utils.patch

8.6.30 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.

8.6.31 Django Model Managers - celery.managers

class celery.managers.**TaskManager**
Manager for `celery.models.Task` models.

get_task(**args*, ***kwargs*)
Get task meta for task by `task_id`.

Parameters **exception_retry_count** – How many times to retry by transaction rollback on exception. This could theoretically happen in a race condition if another worker is trying to create the same task. The default is to retry once.

store_result(**args*, ***kwargs*)
Store the result and status of a task.

Parameters

- **task_id** – task id
- **result** – The return value of the task, or an exception instance raised by the task.
- **status** – Task status. See `celery.result.AsyncResult.get_status()` for a list of possible status values.
- **traceback** – The traceback at the point of exception (if the task failed).
- **exception_retry_count** – How many times to retry by transaction rollback on exception. This could theoretically happen in a race condition if another worker is trying to create the same task. The default is to retry twice.

class celery.managers.**TaskSetManager**
Manager for `celery.models.TaskSet` models.

restore_taskset(**args*, ***kwargs*)
Get taskset meta for task by `taskset_id`.

store_result(**args*, ***kwargs*)
Store the result of a taskset.

Parameters

- **taskset_id** – task set id
- **result** – The return value of the taskset

celery.managers.**transaction_retry**(*max_retries=1*)
Decorator for methods doing database operations.

If the database operation fails, it will retry the operation at most `max_retries` times.

8.6.32 Django Models - celery.models

TASK_STATUS_PENDING
The string status of a pending task.

TASK_STATUS_RETRY
The string status of a task which is to be retried.

TASK_STATUS_FAILURE
The string status of a failed task.

TASK_STATUS_DONE
The string status of a task that was successfully executed.

TASK_STATUSES
List of possible task statuses.

TASK_STATUSES_CHOICES
Django choice tuple of possible task statuses, for usage in model/form fields `choices` argument.

class TaskMeta
Model for storing the result and status of a task.

Note Only used if you're running the `database` backend.

task_id
The unique task id.

status
The current status for this task.

result
The result after successful/failed execution. If the task failed, this contains the execption it raised.

date_done
The date this task changed status.

class PeriodicTaskMeta
Metadata model for periodic tasks.

name
The name of this task, as registered in the task registry.

last_run_at
The date this periodic task was last run. Used to find out when it should be run next.

total_run_count
The number of times this periodic task has been run.

task
The class/function for this task.

delay()
Delay the execution of a periodic task, and increment its total run count.

CHAPTER 9

Change history

9.1 1.0.6 [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:

  ```
  $ PYTHONPATH=. camqadm exchange.delete celeryresults
  ```

  or:

  ```
  $ python manage.py camqadm exchange.delete celeryresults
  ```

9.2 1.0.5 [2010-06-01 02:36 P.M CEST]

9.2.1 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.

9.2.2 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?

9.3 1.0.4 [2010-05-31 09:54 A.M CEST]

- Changlog merged with 1.0.5 as the release was never announced.

9.4 1.0.3 [2010-05-15 03:00 P.M CEST]

9.4.1 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
  ```
- 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

9.4.2 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!

9.4.3 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': [<TaskWrapper ....>]}]
  ```

- 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 <TaskWrapper:
              {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 <TaskWrapper:
              {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 <TaskWrapper:
              {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}"}>']}]
  ```

9.4.4 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 `TaskWrapper.on_failure`.
- Database backend: `TaskMeta.result`: default value should be `None` not empty string.

9.5 1.0.2 [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.worker.pool.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.

9.6 1.0.1 [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 `1.2.0` 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.

9.7 1.0.0 [2010-02-10 04:00 P.M CET]

9.7.1 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()
```

9.7.2 DEPRECATIONS

- The following configuration variables has been renamed and will be deprecated in v1.2:
  - 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 v1.2.
- `TaskSet.run` has been renamed to `TaskSet.apply_async`.

  `TaskSet.run` has now been deprecated, and is scheduled for removal in v1.2.

9.7.3 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.

9.7.4 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.

9.7.5 BUGS

- Fixed a race condition that could happen while storing task results in the database.

9.7.6 DOCUMENTATION

- Reference now split into two sections; API reference and internal module reference.

9.7.7 0.8.4 [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.

9.7.8 0.8.3 [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.

9.7.9 0.8.2 [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.

9.8 0.8.1 [2009-11-16 05:21 P.M CEST]

9.8.1 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!

9.8.2 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

9.8.3 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.

9.9 0.8.0 [2009-09-22 03:06 P.M CEST]

9.9.1 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

9.9.2 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`.

9.9.3 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

9.10 0.6.0 [2009-08-07 06:54 A.M CET]

9.10.1 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.

9.10.2 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!

9.11 0.4.1 [2009-07-02 01:42 P.M CET]

- Fixed a bug with parsing the message options (`mandatory`, `routing_key`, `priority`, `immediate`)

9.12 0.4.0 [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.

9.13 0.3.20 [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`.

9.13.1 0.3.7 [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.

9.14 0.3.3 [2009-06-08 01:07 P.M CET]

- The `PeriodicWorkController` now sleeps for 1 second between checking for periodic tasks to execute.

9.15 0.3.2 [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.

9.16 0.3.1 [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`)

9.17 0.3.0 [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.

9.18 0.2.0 [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.

9.19 0.2.0-pre3 [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.

9.20 0.2.0-pre2 [2009-05-20 01:56 P.M CET]

- Now handles unpickleable exceptions (like the dynimically generated subclasses of `django.core.exception.MultipleObjectsReturned`).

9.21 0.2.0-pre1 [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.

9.22 0.1.15 [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!

9.23 0.1.14 [2009-05-19 01:08 P.M CET]

- Fixed a syntax error in the `TaskSet` class. (No such variable `TimeOutError`).

9.24 0.1.13 [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`.

9.25 0.1.12 [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.
  ```

9.26 0.1.11 [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)

9.27 0.1.10 [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.

9.28 0.1.8 [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.

9.29 0.1.7 [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`.

9.30 0.1.6 [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`.

9.31 0.1.0 [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