collective.celery Documentation

Release 1.0a1

Plone Intranet

May 14, 2015

Contents

`collective.celery` provides the necessary bits to use Celery within Plone.

Much of the code here is based off of David Glick's gists, Asko's work and pyramid_celery.

CHAPTER 1

Configuration

Add the python package to your buildout eggs section:

```
eggs =
    ...
# Change this to celery[redis] or celery[librabbitmq] if you want to use Redis or RabbitMQ respective
    celery[sqlalchemy]
    collective.celery
    ...
```

You'll also need to configure buildout to include the celery script in your bin directory:

```
parts =
    ...
    scripts
    ...

[scripts]
recipe = zc.recipe.egg
eggs = ${buildout:eggs}
scripts = pcelery
```

Note: If you already have a `scripts` section, just make sure it also generates pcelery and that the eggs are correct.

Finally, configure celery by setting `environment-vars` on your client configuration. All variables defined there are passed on to celery configuration:

```
environment-vars =
    ...
# CELERY_IMPORTS is required to load your tasks correctly for your project
    CELERY_IMPORTS ('my.package.tasks',)
# basic example just using sqlalchemy
    BROKER_URL sqla+sqlite:///${buildout:directory}/celerydb.sqlite?timeout=30
    CELERY_RESULT_BACKEND db+sqlite:///${buildout:directory}/celeryresults.sqlite?timeout=30
    ...
```

CHAPTER 2

Creating tasks

This package comes with two decorators to use for creating tasks.

default run the task as the user who created the task

as_admin run the task as an admin

Example:

```
from collective.celery import task

@task()
def do_something(context, arg1, foo='bar'):
    pass

@task.as_admin()
def do_something_as_admin(context, arg1, foo='bar'):
    pass
```

And to schedule the taks:

```
my_content_object = self.context
do_something.delay(my_content_object, 'something', foo='bar')
```

Or alternatively:

```
my_content_object = self.context
do_something.apply_async((my_content_object, 'something'), {'foo': 'bar'})
```

Check out calliung tasks in the celery documentation for more details.

Note: You do not need to specify a context object if you don't use it for anything meaningful in the task: the system will already set up the correct site and if you just need that you can obtain it easily (maybe via `plone.api`).

CHAPTER 3

Starting the task runner

The package simply provides a wrapper around the default task runner script which takes an additional zope config parameter:

```
$ bin/pcelery worker parts/instance/etc/zope.conf
```

Note: In order for the worker to start correctly, you should have a ZEO server setup. Else the worker will fail stating it cannot obtain a lock on the database.

CHAPTER 4

Developing and testing

If you are developing, and do not want the hassle of setting up a ZEO server and run the worker, you can set the following in your instance `environment-vars`:

```
environment-vars =
    ...
    CELERY_ALWAYS_EAGER True
# CELERY_IMPORTS is required to load your tasks correctly for your project
    CELERY_IMPORTS ('my.package.tasks',)
# basic example just using sqlalchemy
    BROKER_URL sqla+sqlite:///${buildout:directory}/celerydb.sqlite?timeout=30
    CELERY_RESULT_BACKEND db+sqlite:///${buildout:directory}/celeryresults.sqlite?timeout=30
    ...
```

In this way, thanks to the CELERY_ALWAYS_EAGER setting, celery will not send the task to the worker at all but execute immediately when `delay` or `apply_async` are called.

Similarly, in tests, we provide a layer that does the following:

1. Set `CELERY_ALWAYS_EAGER` for you, so any function you are testing that calls an asyncroinous function will have that function executed after commit (see The execution model)
2. Use a simple, in-memory SQLite database to store results

To use it, your package should depend, in its `test` extra requirement, from `collective.celery[test]`:

```
# setup.py
...
setup(name='my.package',
    ...
    extras_require={
        ...
        'test': [
            'collective.celery[test]',
        ],
        ...
    },
...
```

And then, in your `testing.py`:

```
...
from collective.celery.testing import CELERY
...

class MyLayer(PloneSandboxLayer):
```

```
    defaultBases = (PLONE_FIXTURE, CELERY, ...)

...
```

4.1 Complete API and advanced usage

4.1.1 The execution model

`collective.celery` doesn't queue tasks immediately when you call `delay` or `apply_async`, but rather waits after the current transaction is committed (via an after commit hook) to do so.

This is to avoid a problem where the task depends on some objects that are being added in the current transaction: should we send the task immediately, it might be executed before the transaction is committed and therefore might not find some objects [1].

This is implemented by adding an "after-commit hook" to the transaction that will queue the task after the current transaction has safely committed, but has a number of implications:

1. The task id has to be pre-generated in order to return a `AsyncResult`. That is done by using the same udnerlying function that Celery uses, `kombu.utils.uuid()`
2. In the case an eager result was requested (see *Usage of CELERY_ALWAYS_EAGER*), a special wrapped result is constructed that will mimick the Celery API, while assuring consistent usage (unlike with standard Celery, we insert the eventual result into the result backend even if we are in eager mode)

Authorization

When the task is run, unless you use the `as_admin()` method, it will be run with the user security context (i.e. the same user) as the one who queued the task. So if our user Alice queues the task `foo` (by .e.g. invoking a browser view) that task will be run as if Alice has invoked it directly.

Exception handling

Tasks are run wrapped into their own transaction: a transaction is begun before the task is run in the worker, and committed if the task runs without failure, or else it will be rollbacked.

If the exception raised is a `ConflictError`, the task will be retried using the Celery retry mechanism (substantially requeued and executed again as soon as possible).

Any other exception will be reraised after rollbacking the transaction and therefore catched by the celery worker logging.

Argument serialization

Each argument to the task will be serialized before being sent to the task. The serialization folows the standard Celery mechanism (i.e. nothing special is done) with the exception of content objects (those implementing `OFS.interfaces.IItem`).

These objects are serialized to a string with the format `object://<path-to-object>` where `path-to-object` is the object path (obtained via `getPhysicalPath`).

[1] See http://celery.readthedocs.org/en/latest/userguide/tasks.html#database-transactions

Warning: It is therefore quite insecure to pass as arguments any objects residing in the ZODB which are not "content objects", such as for example single field or object attribute. In general, you should only pass safely pickleable objects (pickleable by the tandrad pickler) and "content objects" as arguments

The custom worker

In order to render our tasks capable of executing properly, `collective.celery` comes with a custom worker: this worker basically just wraps the standard worker by doing the initial Zope setup (reading configuration, connecting to the ZODB, pulling up ZCML).

This is why you have to pass a zope configuration file to the worker, and why you have to use ZEO or an equivalent architecture (the worker does connect to the database).

4.1.2 Using class based tasks

If you need to do advanced things with tasks and you think you need a class-based task (see Custom task classes), you can do it, but you have to keep in mind two things:

1. Always inherit from `collective.celery.base_task.AfterCommitTask`
2. If you're doing weird stuff during registration, remember that the default celery app is obtained via `collective.celery.utils.getCelery()`

Example

Here's an example on how to create a custom base task class that vfails quite loudly:

```
from collective.celery.base_task import AfterCommitTask
from collective.celery import task

class LoudBaseTask(AfterCommitTask):

    abstract = True

    def on_failure(self, exc, task_id, args, kwargs, einfo):
        # Send emails to people, ring bells, call 911
        yell_quite_loudly(exc, task_id, args, kwargs, einfo)

@task(base=LoudBaseTask)
def fails_loudly(*args):
    return sum(args)
```

4.1.3 Tips and tricks

Usage of `CELERY_ALWAYS_EAGER`

The `CELERY_ALWAYS_EAGER` setting is very useful when developing, and also the only available option to test your tasks without going mad [#]_.

In a nutshell, it works by completely skipping the whole "enqueing the task and letting the worker run it" part of Celery and directly executing the task when you "queue" it.

But since we do always delay the actual execution after the transaction has committed (see *The execution model*) this doesn't go as simply as stated.

While the testing layer sets up evrything for you without you having to worry about these nasty details (see *Developing and testing*) when you are developing with `CELERY_ALWAYS_EAGER` enabled you **must** provide a result backend for celery to use (else retrieving tasks result will break horribly).

There are two backends that you can use:

1. The safest option is the SQLite backend used in *Developing and testing*
2. The in-memory result backend (the test default), which involves also setting the `CELERY_CACHE_BACKEND` to `memory://`. Note that this backend, while thread safe, absolutely does not work across different processes. Therefore it isn't recommended.

Tests

When you are testing a product that relies on `collective.celery`, yopu have two options available: call the task directly, or call it indirectly.

In the first, you get an `EagerResult` back, and therefore you can immediately see its return value:

```
import unittest
import transaction
from collective.celery import task
# Make sure has collective.celery.testing.CELERY as base
from ..testing import MY_LAYER


@task()
def sum_all(*args):
    return sum(args)


class TestSum(unittest.TestCase):

    layer = MY_LAYER

    def test_sum_all(self):
        result = sum_all.delay(1,2,3)
        transaction.commit() # Up until here, it is not executed
        self.assertEqual(result, 6)
```

In more complex cases, like a robot test, where you might have a browserview that polls the result backend, everything should work smoothly as long as you have `collective.celery.testing.CELERY` within your layer's bases.

4.1.4 `collective.celery` Package

4.2 Changelog

4.2.1 Changelog

1.0a5 (2015-03-11)

- Add options argument to pass more options to apply_async (e.a: countdown eta etc.)

1.0a4 (2015-03-04)

- use unrestrictedTraverse when deserializing parameters
- Fix documentation

1.0a2 (2015-03-03)

- Initial release

CHAPTER 5

Indices and tables

- genindex
- modindex
- search