Previous topic

Signals

Next topic

Extensions and Bootsteps

This Page

Show Source

Quick search

Go

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

Testing with Celery

Tasks and unit tests

To test task behavior in unit tests the preferred method is mocking.

Eager mode:

The eager mode enabled by the `task_always_eager` setting is by definition not suitable for unit tests.

When testing with eager mode you are only testing an emulation of what happens in a worker, and there are many discrepancies between the emulation and what happens in reality.

A Celery task is much like a web view, in that it should only define how to perform the action in the context of being called as a task.

This means optimally tasks only handle things like serialization, message headers, retries, and so on, with the actual logic implemented elsewhere.

Say we had a task like this:

```
from .models import Product


@app.task(bind=True)
def send_order(self, product_pk, quantity, price):
    price = Decimal(price)  # json serializes this to string.

    # models are passed by id, not serialized.
    product = Product.objects.get(product_pk)

    try:
        product.order(quantity, price)
    except OperationalError as exc:
        raise self.retry(exc=exc)
```

You could write unit tests for this task, using mocking like in this example:

```python
from pytest import raises

from celery.exceptions import Retry

# for python 2: use mock.patch from `pip install mock`.
from unittest.mock import patch

from proj.models import Product
from proj.tasks import send_order

class test_send_order:

    @patch('proj.tasks.Product.order')  # < patching Product in module a
    def test_success(self, product_order):
        product = Product.objects.create(
            name='Foo',
        )
        send_order(product.pk, 3, Decimal(30.3))
        product_order.assert_called_with(3, Decimal(30.3))

    @patch('proj.tasks.Product.order')
    @patch('proj.tasks.send_order.retry')
    def test_failure(self, send_order_retry, product_order):
        product = Product.objects.create(
            name='Foo',
        )

        # Set a side effect on the patched methods
        # so that they raise the errors we want.
        send_order_retry.side_effect = Retry()
        product_order.side_effect = OperationalError()

        with raises(Retry):
            send_order(product.pk, 3, Decimal(30.6))
```

Py.test

New in version 4.0.

Celery is also a pytest plugin that adds fixtures that you can use in your integration (or unit) test suites.

Marks

`celery` - Set test app configuration.

The `celery` mark enables you to override the configuration used for a single test case:

```python
@pytest.mark.celery(result_backend='redis://')
def test_something():
    ...
```

or for all the test cases in a class:

```python
@pytest.mark.celery(result_backend='redis://')
class test_something:

    def test_one(self):
        ...

    def test_two(self):
        ...
```

Fixtures

Function scope

`celery_app` - Celery app used for testing.

This fixture returns a Celery app you can use for testing.

Example:

```
def test_create_task(celery_app, celery_worker):
    @celery_app.task
    def mul(x, y):
        return x * y

    assert mul.delay(4, 4).get(timeout=10) == 16
```

`celery_worker` - Embed live worker.

This fixture starts a Celery worker instance that you can use for integration tests. The worker will be started in a *separate thread* and will be shutdown as soon as the test returns.

Example:

```
# Put this in your conftest.py
@pytest.fixture(scope='session')
def celery_config():
    return {
        'broker_url': 'amqp://',
        'result_backend': 'redis://'
    }

def test_add(celery_worker):
    mytask.delay()


# If you wish to override some setting in one test cases
# only - you can use the ``celery`` mark:
@pytest.mark.celery(result_backend='rpc')
def test_other(celery_worker):
    ...
```

Session scope

`celery_config` - Override to setup Celery test app configuration.

You can redefine this fixture to configure the test Celery app.

The config returned by your fixture will then be used to configure the **`celery_app()`**, and **`celery_session_app()`** fixtures.

Example:

```
@pytest.fixture(scope='session')
def celery_config():
    return {
        'broker_url': 'amqp://',
        'result_backend': 'rpc',
    }
```

`celery_parameters` - Override to setup Celery test app parameters.

You can redefine this fixture to change the `__init__` parameters of test Celery app. In contrast to **`celery_config()`**, these are directly passed to when instantiating **`Celery`**.

The config returned by your fixture will then be used to configure the **`celery_app()`**, and **`celery_session_app()`** fixtures.

Example:

```
@pytest.fixture(scope='session')
def celery_parameters():
    return {
        'task_cls':  my.package.MyCustomTaskClass,
        'strict_typing': False,
    }
```

`celery_worker_parameters` - Override to setup Celery worker parameters.

You can redefine this fixture to change the `__init__` parameters of test Celery workers. These are directly passed to **`WorkController`** when it is instantiated.

The config returned by your fixture will then be used to configure the **`celery_worker()`**, and **`celery_session_worker()`** fixtures.

Example:

```
@pytest.fixture(scope='session')
def celery_worker_parameters():
    return {
        'queues':  ('high-prio', 'low-prio'),
        'exclude_queues': ('celery'),
    }
```

`celery_enable_logging` - Override to enable logging in embedded workers.

This is a fixture you can override to enable logging in embedded workers.

Example:

```
@pytest.fixture(scope='session')
def celery_enable_logging():
    return True
```

`celery_includes` - Add additional imports for embedded workers.

You can override fixture to include modules when an embedded worker starts.

You can have this return a list of module names to import, which can be task modules, modules registering signals, and so on.

Example:

```
@pytest.fixture(scope='session')
def celery_includes():
    return [
        'proj.tests.tasks',
        'proj.tests.celery_signal_handlers',
    ]
```

`celery_worker_pool` - Override the pool used for embedded workers.

You can override fixture to configure the execution pool used for embedded workers.

Example:

```
@pytest.fixture(scope='session')
def celery_worker_pool():
    return 'prefork'
```

Warning:

You cannot use the gevent/eventlet pools, that is unless your whole test suite is running with the monkeypatches enabled.

`celery_session_worker` - Embedded worker that lives throughout the session.

This fixture starts a worker that lives throughout the testing session (it won't be started/stopped for every test).

Example:

```
# Add this to your conftest.py
@pytest.fixture(scope='session')
def celery_config():
    return {
        'broker_url': 'amqp://',
        'result_backend': 'rpc',
    }

# Do this in your tests.
def test_add_task(celery_session_worker):
    assert add.delay(2, 2) == 4
```

Warning:

It's probably a bad idea to mix session and ephemeral workers…

`celery_session_app` - Celery app used for testing (session scope).

This can be used by other session scoped fixtures when they need to refer to a Celery app instance.

`use_celery_app_trap` - Raise exception on falling back to default app.

This is a fixture you can override in your `conftest.py`, to enable the "app trap": if something tries to access the default or current_app, an exception is raised.

Example:

```
@pytest.fixture(scope='session')
def use_celery_app_trap():
    return True
```

If a test wants to access the default app, you would have to mark it using the `depends_on_current_app` fixture:

```
@pytest.mark.usefixtures('depends_on_current_app')
def test_something():
    something()
```