Celery Background Tasks

If your application has a long running task, such as processing some uploaded data or sending email, you don't want to wait for it to finish during a request. Instead, use a task queue to send the necessary data to another process that will run the task in the background while the request returns immediately.

Celery is a powerful task queue that can be used for simple background tasks as well as complex multi-stage programs and schedules. This guide will show you how to configure Celery using Flask, but assumes you've already read the First Steps with Celery guide in the Celery documentation.

Install

Celery is a separate Python package. Install it from PyPI using pip:

```
$ pip install celery
```

Configure

The first thing you need is a Celery instance, this is called the celery application. It serves the same purpose as the **`Flask`** object in Flask, just for Celery. Since this instance is used as the entry-point for everything you want to do in Celery, like creating tasks and managing workers, it must be possible for other modules to import it.

For instance you can place this in a `tasks` module. While you can use Celery without any reconfiguration with Flask, it becomes a bit nicer by subclassing tasks and adding support for Flask's application contexts and hooking it up with the Flask configuration.

This is all that is necessary to properly integrate Celery with Flask:

```
from celery import Celery

def make_celery(app):
    celery = Celery(
        app.import_name,
        backend=app.config['CELERY_RESULT_BACKEND'],
        broker=app.config['CELERY_BROKER_URL']
    )
    celery.conf.update(app.config)

    class ContextTask(celery.Task):
```

```
        def __call__(self, *args, **kwargs):
            with app.app_context():
                return self.run(*args, **kwargs)

    celery.Task = ContextTask
    return celery
```

The function creates a new Celery object, configures it with the broker from the application config, updates the rest of the Celery config from the Flask config and then creates a subclass of the task that wraps the task execution in an application context.

An example task

Let's write a task that adds two numbers together and returns the result. We configure Celery's broker and backend to use Redis, create a `celery` application using the factor from above, and then use it to define the task.

```
from flask import Flask

flask_app = Flask(__name__)
flask_app.config.update(
    CELERY_BROKER_URL='redis://localhost:6379',
    CELERY_RESULT_BACKEND='redis://localhost:6379'
)
celery = make_celery(flask_app)

@celery.task()
def add_together(a, b):
    return a + b
```

This task can now be called in the background:

```
result = add_together.delay(23, 42)
result.wait()  # 65
```

Run a worker

If you jumped in and already executed the above code you will be disappointed to learn that `.wait()` will never actually return. That's because you also need to run a Celery worker to receive and execute the task.

```
$ celery -A your_application.celery worker
```

The `your_application` string has to point to your application's package or module that creates the `celery` object.

Now that the worker is running, `wait` will return the result once the task is finished.