https://testdriven.io/blog/django-and-celery/

If a long-running process is part of your application's workflow, rather than blocking the response, you should handle it in the background, outside the normal request/response flow.

Perhaps your web application requires users to submit a thumbnail (which will probably need to be re-sized) and confirm their email when they register. If your application processed the image and sent a confirmation email directly in the request handler, then the end user would have to wait unnecessarily for them both to finish processing before the page loads or updates. Instead, you'll want to pass these processes off to a task queue and let a separate worker process deal with it, so you can immediately send a response back to the client. The end user can then do other things on the client-side while the processing takes place. Your application is also free to respond to requests from other users and clients.

To achieve this, we'll walk you through the process of setting up and configuring Celery and Redis for handling long-running processes in a Django app. We'll also use Docker and Docker Compose to tie everything together. Finally, we'll look at how to test the Celery tasks with unit and integration tests.

Django + Celery Series:

1. Asynchronous Tasks with Django and Celery (this article!)
2. Handling Periodic Tasks in Django with Celery and Docker
3. Automatically Retrying Failed Celery Tasks
4. Working with Celery and Database Transactions

Objectives

By the end of this article you will be able to:

1. Integrate Celery into a Django app and create tasks
2. Containerize Django, Celery, and Redis with Docker
3. Run processes in the background with a separate worker process
4. Save Celery logs to a file
5. Set up Flower to monitor and administer Celery jobs and workers
6. Test a Celery task with both unit and integration tests

Background Tasks

Again, to improve user experience, long-running processes should be run outside the normal HTTP request/response flow, in a background process.

Examples:

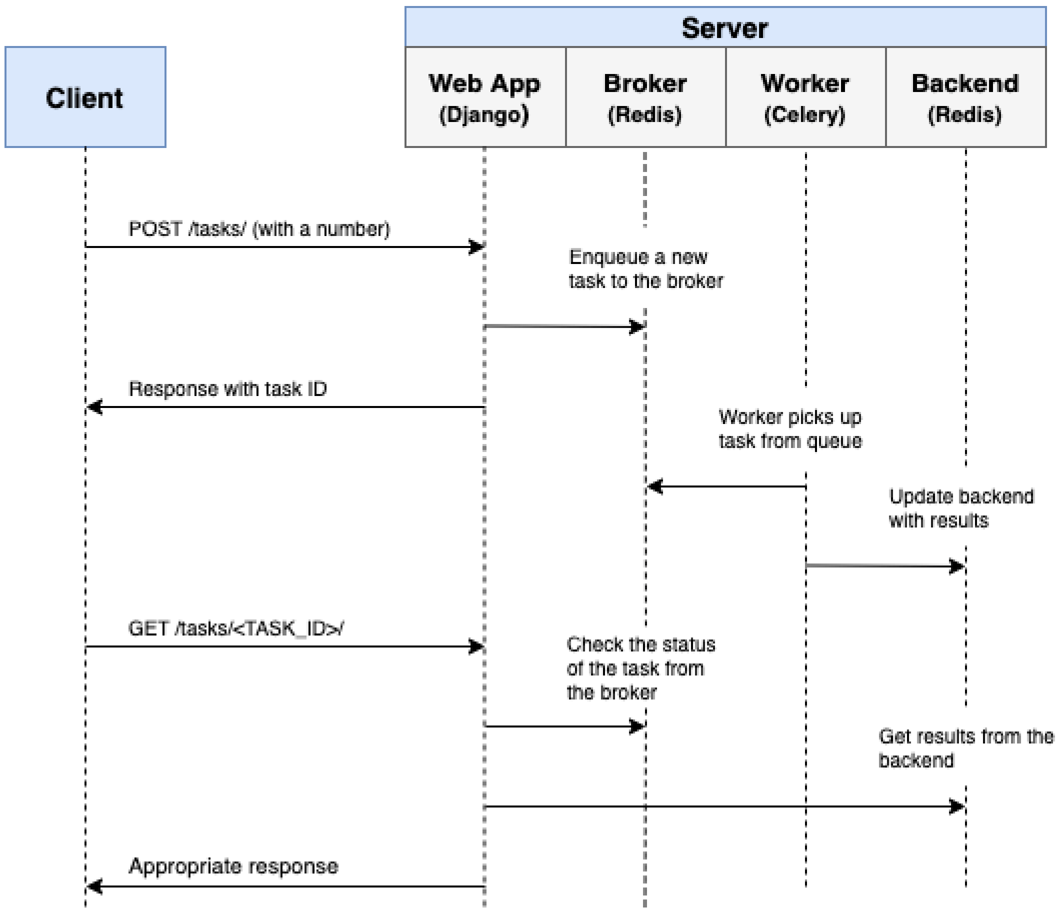
1. Running machine learning models
2. Sending confirmation emails
3. Web scraping and crawling
4. Analyzing data
5. Processing images
6. Generating reports

As you're building out an app, try to distinguish tasks that should run during the request/response lifecycle, like CRUD operations, from those that should run in the background.

Workflow

Our goal is to develop a Django application that works in conjunction with Celery to handle long-running processes outside the normal request/response cycle.

1. The end user kicks off a new task via a POST request to the server-side.
2. Within the view, a task is added to the queue and the task id is sent back to the client-side.
3. Using AJAX, the client continues to poll the server to check the status of the task while the task itself is running in the background.


Server
Client
Web App (Django)
Broker (Redis)
Worker (Celery)
Backend (Redis)
POST /tasks/ (with a number)
Enqueue a new task to the broker
Response with task ID
Worker picks up task from queue
Update backend with results
GET /tasks/<TASK_ID>/
Check the status of the task from the broker
Get results from the backend
Appropriate response

Project Setup

Clone down the base project from the django-celery repo, and then check out the v1 tag to the master branch:

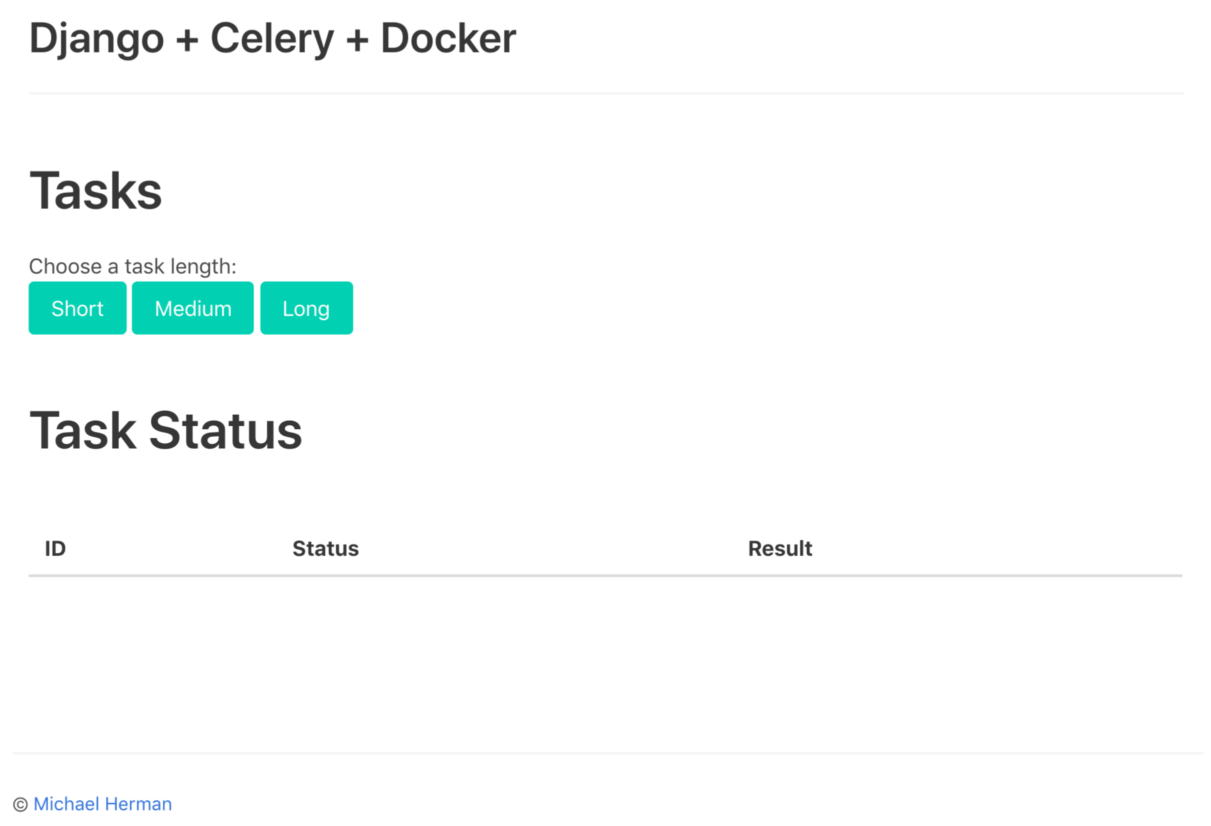
```
$ git clone https://github.com/testdrivenio/django-celery --branch v1 --single-
branch
$ cd django-celery
$ git checkout v1 -b master
```

Since we'll need to manage three processes in total (Django, Redis, worker), we'll use Docker to simplify our workflow by wiring them up so that they can all be run from one terminal window with a single command.

From the project root, create the images and spin up the Docker containers:

```
$ docker-compose up -d --build
```

Once the build is complete, navigate to http://localhost:1337:


Django + Celery + Docker

Tasks

Choose a task length:

Task Status

ID	Status	Result

© Michael Herman

Make sure the tests pass as well:

```
$ docker-compose exec web python -m pytest

=============================== test session starts
================================
platform linux -- Python 3.11.1, pytest-7.2.0, pluggy-1.0.0
django: settings: core.settings (from ini)
rootdir: /usr/src/app, configfile: pytest.ini
plugins: django-4.5.2
collected 1 item

tests/test_tasks.py .                                                     [100%]

================================ 1 passed in 0.17s
================================
```

Take a quick look at the project structure before moving on:

```
├── .gitignore
├── LICENSE
├── README.md
├── docker-compose.yml
└── project
    ├── Dockerfile
    ├── core
    │   ├── __init__.py
    │   ├── asgi.py
    │   ├── settings.py
    │   ├── urls.py
    │   └── wsgi.py
    ├── entrypoint.sh
    ├── manage.py
    ├── pytest.ini
    ├── requirements.txt
    ├── static
    │   ├── bulma.min.css
    │   ├── jquery-3.4.1.min.js
    │   ├── main.css
    │   └── main.js
    ├── tasks
    │   ├── __init__.py
    │   ├── apps.py
    │   ├── migrations
    │   │   └── __init__.py
    │   ├── templates
    │   │   └── home.html
    │   └── views.py
    └── tests
        ├── __init__.py
        └── test_tasks.py
```

Want to learn how to build this project? Check out the Dockerizing Django with Postgres, Gunicorn, and Nginx article.

Trigger a Task

An event handler in *project/static/main.js* is set up that listens for a button click. On click, an AJAX POST request is sent to the server with the appropriate task type: `1`, `2`, or `3`.

```
$('.button').on('click', function() {
  $.ajax({
    url: '/tasks/',
    data: { type: $(this).data('type') },
    method: 'POST',
  })
  .done((res) => {
    getStatus(res.task_id);
  })
  .fail((err) => {
    console.log(err);
  });
});
```

On the server-side, a view is already configured to handle the request in *project/tasks/views.py*:

```
@csrf_exempt
def run_task(request):
    if request.POST:
        task_type = request.POST.get("type")
        return JsonResponse({"task_type": task_type}, status=202)
```

Now comes the fun part: wiring up Celery!

Celery Setup

Start by adding both Celery and Redis to the *project/requirements.txt* file:

```
Django==4.1.4
pytest==7.2.0
pytest-django==4.5.2

celery==5.2.7
redis==4.4.0
```

Celery uses a message broker -- RabbitMQ, Redis, or AWS Simple Queue Service (SQS) -- to facilitate communication between the Celery worker and the web application. Messages are added to the broker, which are then processed by the worker(s). Once done, the results are added to the backend.

Redis will be used as both the broker and backend. Add both Redis and a Celery worker to the *docker-compose.yml* file like so:

```
version: '3.8'

services:
  web:
    build: ./project
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - ./project:/usr/src/app/
    ports:
      - 1337:8000
    environment:
      - DEBUG=1
      - SECRET_KEY=dbaa1_i7%*3r9-=z-+_mz4r-!qeed@(-a_r(g@k8jo8y3r27%m
      - DJANGO_ALLOWED_HOSTS=localhost 127.0.0.1 [::1]
      - CELERY_BROKER=redis://redis:6379/0
      - CELERY_BACKEND=redis://redis:6379/0
    depends_on:
      - redis

  celery:
    build: ./project
    command: celery --app=core worker --loglevel=info
    volumes:
      - ./project:/usr/src/app
    environment:
      - DEBUG=1
      - SECRET_KEY=dbaa1_i7%*3r9-=z-+_mz4r-!qeed@(-a_r(g@k8jo8y3r27%m
      - DJANGO_ALLOWED_HOSTS=localhost 127.0.0.1 [::1]
      - CELERY_BROKER=redis://redis:6379/0
      - CELERY_BACKEND=redis://redis:6379/0
    depends_on:
      - web
      - redis

  redis:
    image: redis:7-alpine
```

Take note of `celery --app=core worker --loglevel=info`:

1. `celery worker` is used to start a Celery worker
2. `--app=core` runs the `core` Celery Application (which we'll define shortly)
3. `--loglevel=info` sets the logging level to info

Within the project's settings module, add the following at the bottom to tell Celery to use Redis as the broker and backend:

```
CELERY_BROKER_URL = os.environ.get("CELERY_BROKER", "redis://redis:6379/0")
CELERY_RESULT_BACKEND = os.environ.get("CELERY_BROKER", "redis://redis:6379/0")
```

Next, create a new file called *sample_tasks.py* in "project/tasks":

```
# project/tasks/sample_tasks.py

import time

from celery import shared_task


@shared_task
def create_task(task_type):
    time.sleep(int(task_type) * 10)
    return True
```

Here, using the shared_task decorator, we defined a new Celery task function called `create_task`.

Keep in mind that the task itself will *not* be executed from the Django process; it will be executed by the Celery worker.

Now, add a *celery.py* file to "project/core":

```
import os

from celery import Celery


os.environ.setdefault("DJANGO_SETTINGS_MODULE", "core.settings")
app = Celery("core")
app.config_from_object("django.conf:settings", namespace="CELERY")
app.autodiscover_tasks()
```

What's happening here?

1. First, we set a default value for the `DJANGO_SETTINGS_MODULE` environment variable so that Celery will know how to find the Django project.
2. Next, we created a new Celery instance, with the name `core`, and assigned the value to a variable called `app`.
3. We then loaded the celery configuration values from the settings object from `django.conf`. We used `namespace="CELERY"` to prevent clashes with other Django settings. All config settings for Celery must be prefixed with `CELERY_`, in other words.
4. Finally, `app.autodiscover_tasks()` tells Celery to look for Celery tasks from applications defined in `settings.INSTALLED_APPS`.

Update *project/core/__init__.py* so that the Celery app is automatically imported when Django starts:

```
from .celery import app as celery_app


__all__ = ("celery_app",)
```

Trigger a Task

Update the view to kick off the task and respond with the id:

```
# project/tasks/views.py

@csrf_exempt
def run_task(request):
    if request.POST:
        task_type = request.POST.get("type")
        task = create_task.delay(int(task_type))
        return JsonResponse({"task_id": task.id}, status=202)
```

Don't forget to import the task:

```
from tasks.sample_tasks import create_task
```

Build the images and spin up the new containers:

```
$ docker-compose up -d --build
```

To trigger a new task, run:

```
$ curl -F type=0 http://localhost:1337/tasks/
```

You should see something like:

```
{
  "task_id": "6f025ed9-09be-4cbb-be10-1dce919797de"
}
```

Task Status

Turn back to the event handler on the client-side:

```
// project/static/main.js

$('.button').on('click', function() {
  $.ajax({
    url: '/tasks/',
    data: { type: $(this).data('type') },
    method: 'POST',
  })
  .done((res) => {
    getStatus(res.task_id);
  })
  .fail((err) => {
    console.log(err);
  });
});
```

When the response comes back from the original AJAX request, we then continue to call `getStatus()` with the task id every second:

```
function getStatus(taskID) {
  $.ajax({
    url: `/tasks/${taskID}/`,
    method: 'GET'
  })
  .done((res) => {
    const html = `
      <tr>
        <td>${res.task_id}</td>
        <td>${res.task_status}</td>
        <td>${res.task_result}</td>
      </tr>`
    $('#tasks').prepend(html);

    const taskStatus = res.task_status;

    if (taskStatus === 'SUCCESS' || taskStatus === 'FAILURE') return false;
    setTimeout(function() {
      getStatus(res.task_id);
    }, 1000);
  })
  .fail((err) => {
    console.log(err)
  });
}
```

If the response is successful, a new row is added to the table on the DOM.

Update the `get_status` view to return the status:

```
# project/tasks/views.py

@csrf_exempt
def get_status(request, task_id):
    task_result = AsyncResult(task_id)
    result = {
        "task_id": task_id,
        "task_status": task_result.status,
        "task_result": task_result.result
    }
    return JsonResponse(result, status=200)
```

Import AsyncResult:

```
from celery.result import AsyncResult
```

Update the containers:

```
$ docker-compose up -d --build
```

Trigger a new task:

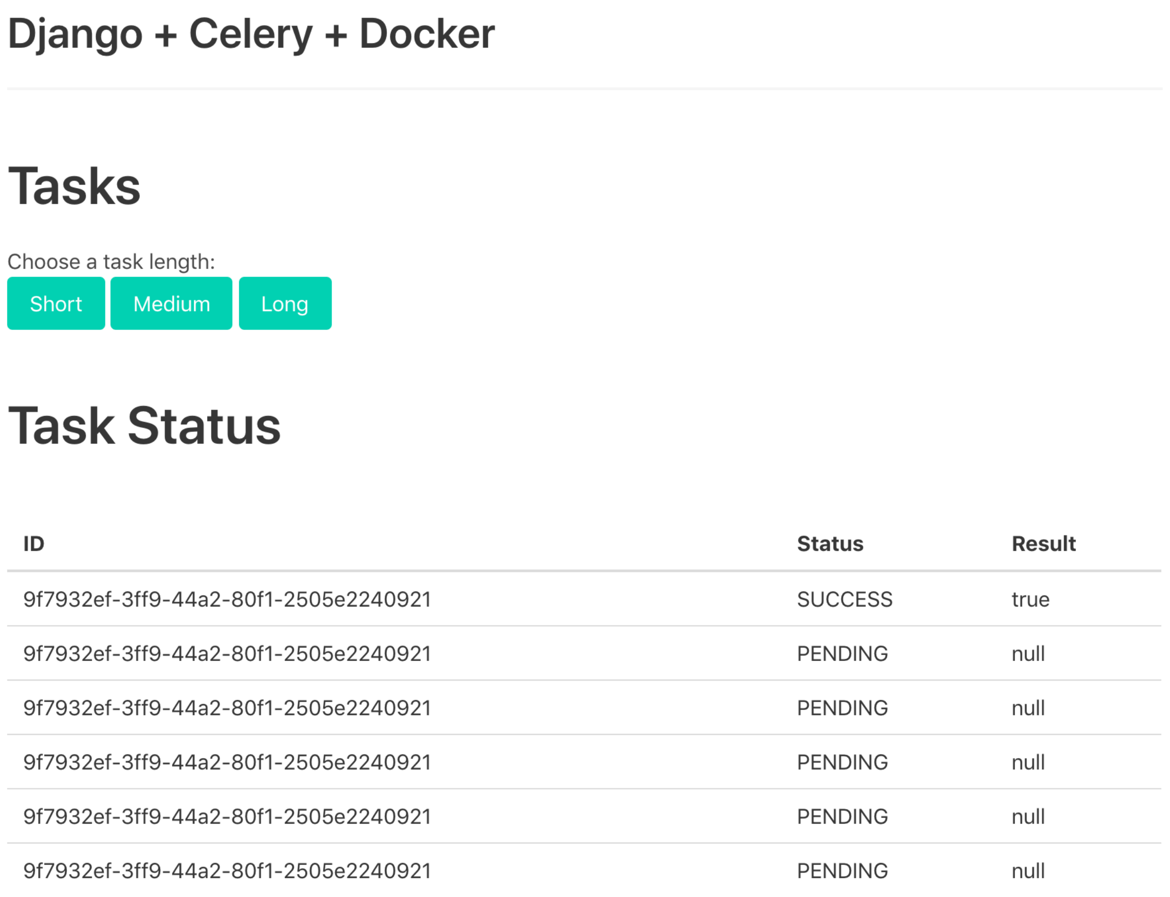
```
$ curl -F type=1 http://localhost:1337/tasks/
```

Then, grab the `task_id` from the response and call the updated endpoint to view the status:

```
$ curl http://localhost:1337/tasks/25278457-0957-4b0b-b1da-2600525f812f/

{
    "task_id": "25278457-0957-4b0b-b1da-2600525f812f",
    "task_status": "SUCCESS",
    "task_result": true
}
```

Test it out in the browser as well:


Django + Celery + Docker

Tasks

Choose a task length:

Task Status

ID	Status	Result
9f7932ef-3ff9-44a2-80f1-2505e2240921	SUCCESS	true
9f7932ef-3ff9-44a2-80f1-2505e2240921	PENDING	null
9f7932ef-3ff9-44a2-80f1-2505e2240921	PENDING	null
9f7932ef-3ff9-44a2-80f1-2505e2240921	PENDING	null
9f7932ef-3ff9-44a2-80f1-2505e2240921	PENDING	null
9f7932ef-3ff9-44a2-80f1-2505e2240921	PENDING	null

Celery Logs

Update the `celery` service, in *docker-compose.yml*, so that Celery logs are dumped to a log file:

```yaml
celery:
  build: ./project
  command: celery --app=core worker --loglevel=info --logfile=logs/celery.log
  volumes:
    - ./project:/usr/src/app
  environment:
    - DEBUG=1
    - SECRET_KEY=dbaa1_i7%*3r9-=z-+_mz4r-!qeed@(-a_r(g@k8jo8y3r27%m
    - DJANGO_ALLOWED_HOSTS=localhost 127.0.0.1 [::1]
    - CELERY_BROKER=redis://redis:6379/0
    - CELERY_BACKEND=redis://redis:6379/0
  depends_on:
    - web
    - redis
```

Add a new directory to "project" called "logs. Then, add a new file called *celery.log* to that newly created directory.

Update:

```
$ docker-compose up -d --build
```

You should see the log file fill up locally since we set up a volume:

```
[2022-12-15 18:15:20,338: INFO/MainProcess] Connected to redis://redis:6379/0
[2022-12-15 18:15:21,328: INFO/MainProcess] mingle: searching for neighbors
[2022-12-15 18:15:23,342: INFO/MainProcess] mingle: all alone
[2022-12-15 18:15:24,214: WARNING/MainProcess] /usr/local/lib/python3.11/site-
packages/celery/fixups/django.py:203: UserWarning: Using settings.DEBUG leads to a
memory
            leak, never use this setting in production environments!
  warnings.warn('''Using settings.DEBUG leads to a memory

[2022-12-15 18:15:25,080: INFO/MainProcess] celery@de308b1b8017 ready.
[2022-12-15 18:15:40,132: INFO/MainProcess] Task
tasks.sample_tasks.create_task[c4589a0d-f718-4d75-a673-fe3626827385] received
[2022-12-15 18:15:40,153: INFO/ForkPoolWorker-1] Task
tasks.sample_tasks.create_task[c4589a0d-f718-4d75-a673-fe3626827385] succeeded in
0.016174572000181797s: True
```

Flower Dashboard

Flower is a lightweight, real-time, web-based monitoring tool for Celery. You can monitor currently running tasks, increase or decrease the worker pool, view graphs and a number of statistics, to name a few.

Add it to *requirements.txt*:

```
Django==4.1.4
pytest==7.2.0
pytest-django==4.5.2

celery==5.2.7
redis==4.4.0
flower==1.2.0
```

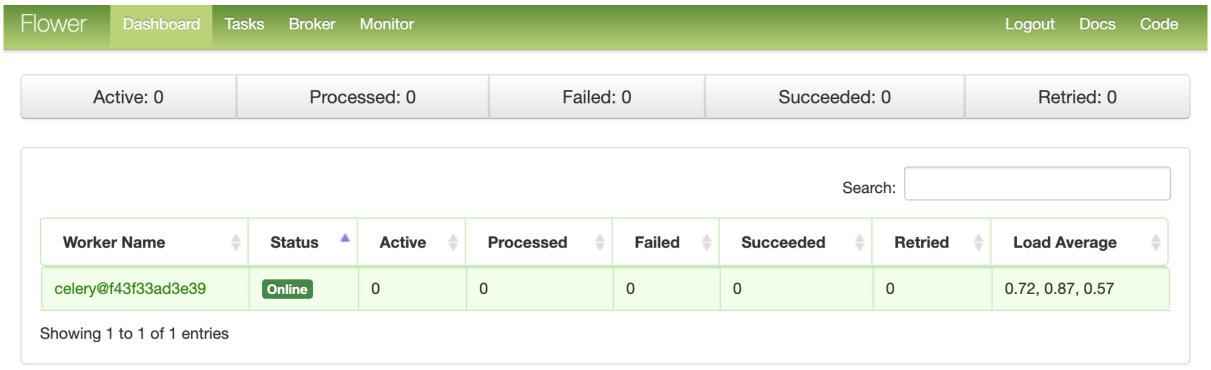
Then, add a new service to *docker-compose.yml*:

```
dashboard:
  build: ./project
  command: celery flower -A core --port=5555 --broker=redis://redis:6379/0
  ports:
    - 5555:5555
  environment:
    - DEBUG=1
    - SECRET_KEY=dbaa1_i7%*3r9-=z-+_mz4r-!qeed@(-a_r(g@k8jo8y3r27%m
    - DJANGO_ALLOWED_HOSTS=localhost 127.0.0.1 [::1]
    - CELERY_BROKER=redis://redis:6379/0
    - CELERY_BACKEND=redis://redis:6379/0
  depends_on:
    - web
    - redis
    - celery
```

Test it out:

```
$ docker-compose up -d --build
```

Navigate to http://localhost:5555 to view the dashboard. You should see one worker ready to go:

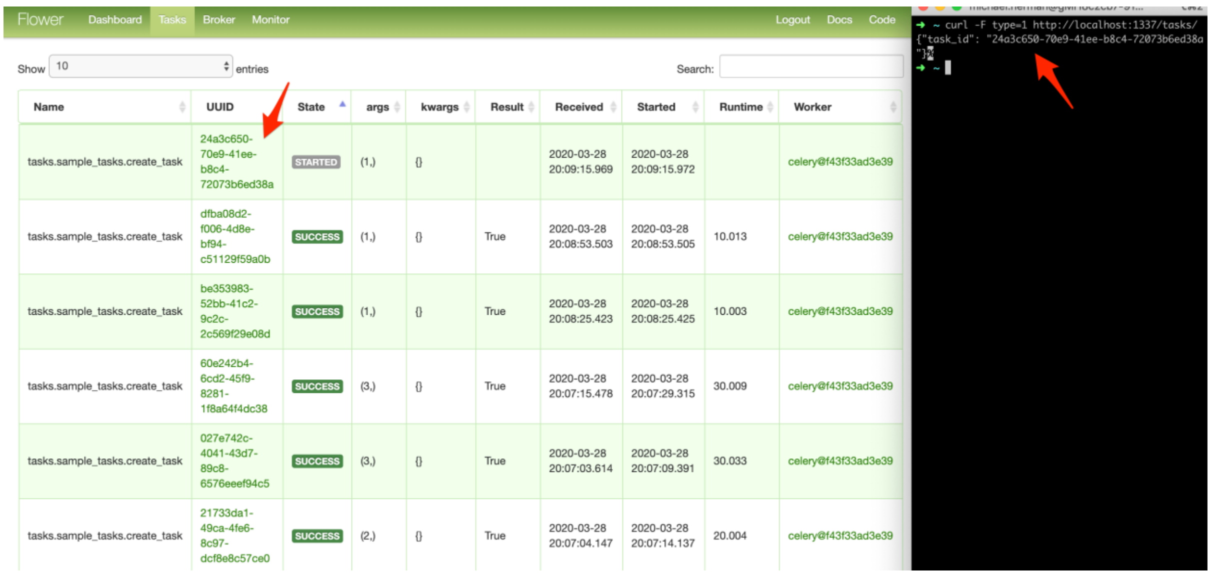


Active: 0	Processed: 0	Failed: 0	Succeeded: 0	Retried: 0

Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@f43f33ad3e39	Online	0	0	0	0	0	0.72, 0.87, 0.57

Showing 1 to 1 of 1 entries

Kick off a few more tasks to fully test the dashboard:



Name	UUID	State	args	kwargs	Result	Received	Started	Runtime	Worker
tasks.sample_tasks.create_task	24a3c650-70e9-41ee-b8c4-72073b6ed38a	STARTED	(1,)	{}		2020-03-28 20:09:15.969	2020-03-28 20:09:15.972		celery@f43f33ad3e39
tasks.sample_tasks.create_task	dfba08d2-f006-4d8e-bf94-c51129f59a0b	SUCCESS	(1,)	{}	True	2020-03-28 20:08:53.503	2020-03-28 20:08:53.505	10.013	celery@f43f33ad3e39
tasks.sample_tasks.create_task	be353983-52bb-41c2-9c2c-2c569f29e08d	SUCCESS	(1,)	{}	True	2020-03-28 20:08:25.423	2020-03-28 20:08:25.425	10.003	celery@f43f33ad3e39
tasks.sample_tasks.create_task	60e242b4-6cd2-45f9-8281-1f8a64f4dc38	SUCCESS	(3,)	{}	True	2020-03-28 20:07:15.478	2020-03-28 20:07:29.315	30.009	celery@f43f33ad3e39
tasks.sample_tasks.create_task	027e742c-4041-43d7-89c8-6576eeef94c5	SUCCESS	(3,)	{}	True	2020-03-28 20:07:03.614	2020-03-28 20:07:09.391	30.033	celery@f43f33ad3e39
tasks.sample_tasks.create_task	21733da1-49ca-4fe6-8c97-dcf8e8c57ce0	SUCCESS	(2,)	{}	True	2020-03-28 20:07:04.147	2020-03-28 20:07:14.137	20.004	celery@f43f33ad3e39

```
➜ ~ curl -F type=1 http://localhost:1337/tasks/
{"task_id": "24a3c650-70e9-41ee-b8c4-72073b6ed38a
"}%
➜ ~
```

Try adding a few more workers to see how that affects things:

```
$ docker-compose up -d --build --scale celery=3
```

Tests

Let's start with the most basic test:

```
def test_task():
    assert sample_tasks.create_task.run(1)
    assert sample_tasks.create_task.run(2)
    assert sample_tasks.create_task.run(3)
```

Add the above test case to *project/tests/test_tasks.py*, and then add the following import:

```
from tasks import sample_tasks
```

Run that test individually:

```
$ docker-compose exec web python -m pytest -k "test_task and not test_home"
```

It should take about one minute to run:

```
=============================== test session starts
===============================
platform linux -- Python 3.11.1, pytest-7.2.0, pluggy-1.0.0
django: settings: core.settings (from ini)
rootdir: /usr/src/app, configfile: pytest.ini
plugins: django-4.5.2
collected 2 items / 1 deselected / 1 selected

tests/test_tasks.py .                                                      [100%]

=================== 1 passed, 1 deselected in 60.69s (0:01:00)
====================
```

It's worth noting that in the above asserts, we used the `.run` method (rather than `.delay`) to run the task directly without a Celery worker.

Want to mock the `.run` method to speed things up?

```
@patch("tasks.sample_tasks.create_task.run")
def test_mock_task(mock_run):
    assert sample_tasks.create_task.run(1)
    sample_tasks.create_task.run.assert_called_once_with(1)

    assert sample_tasks.create_task.run(2)
    assert sample_tasks.create_task.run.call_count == 2

    assert sample_tasks.create_task.run(3)
    assert sample_tasks.create_task.run.call_count == 3
```

Import:

```
from unittest.mock import patch
```

Test:

```
$ docker-compose exec web python -m pytest -k "test_mock_task"
```

```
=============================== test session starts
================================
platform linux -- Python 3.11.1, pytest-7.2.0, pluggy-1.0.0
django: settings: core.settings (from ini)
rootdir: /usr/src/app, configfile: pytest.ini
plugins: django-4.5.2
collected 3 items / 2 deselected / 1 selected

tests/test_tasks.py .                                                      [100%]

========================= 1 passed, 2 deselected in 0.64s
=========================
```

Much quicker!

How about a full integration test?

```
def test_task_status(client):
    response = client.post(reverse("run_task"), {"type": 0})
    content = json.loads(response.content)
    task_id = content["task_id"]
    assert response.status_code == 202
    assert task_id

    response = client.get(reverse("get_status", args=[task_id]))
    content = json.loads(response.content)
    assert content == {"task_id": task_id, "task_status": "PENDING",
"task_result": None}
    assert response.status_code == 200

    while content["task_status"] == "PENDING":
        response = client.get(reverse("get_status", args=[task_id]))
        content = json.loads(response.content)
    assert content == {"task_id": task_id, "task_status": "SUCCESS",
"task_result": True}
```

Keep in mind that this test uses the same broker and backend used in development. You may want to instantiate a new Celery app for testing:

```
app = celery.Celery('tests', broker=CELERY_TEST_BROKER,
backend=CELERY_TEST_BACKEND)
```

Add the import:

```
import json
```

Ensure the test passes.

Conclusion

This has been a basic guide on how to configure Celery to run long-running tasks in a Django app. You should let the queue handle any processes that could block or slow down the user-facing code.

Celery can also be used to execute repeatable tasks and break up complex, resource-intensive tasks so that the computational workload can be distributed across a number of machines to reduce (1) the time to completion and (2) the load on the machine handling client requests.

Finally, if you're curious about how to use WebSockets (via Django Channels) to check the status of a Celery task, instead of using AJAX polling, check out the The Definitive Guide to Celery and Django course.

Grab the code from the repo.

Django + Celery Series:

1. Asynchronous Tasks with Django and Celery (this article!)
2. Handling Periodic Tasks in Django with Celery and Docker
3. Automatically Retrying Failed Celery Tasks
4. Working with Celery and Database Transactions

djangodockertask queue

Michael Herman

Michael is a software engineer and educator who lives and works in the Denver/Boulder area. He is the co-founder/author of Real Python. Besides development, he enjoys building financial models, tech writing, content marketing, and teaching.