Last updated May 10th, 2021

Asynchronous Tasks with FastAPI and Celery

Michael Herman

If a long-running process is part of your application's workflow, rather 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 FastAPI 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.

Objectives

By the end of this tutorial, you will be able to:

1. Integrate Celery into a FastAPI app and create tasks.
2. Containerize FastAPI, 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. 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.

It's worth noting that you can leverage FastAPI's BackgroundTasks class, which comes directly from Starlette, to run tasks in the background.

docker fastapi flask front-end heroku kubernetes machine learning python react task queue testing vue web scraping

TABLE OF CONTENTS

For example:

```
from fastapi import BackgroundTasks


def send_email(email, message):
    pass


@app.get("/")
async def ping(background_tasks: BackgroundTasks):
    background_tasks.add_task(send_email, "email@address.com", "Hi!")
    return {"message": "pong!"}
```

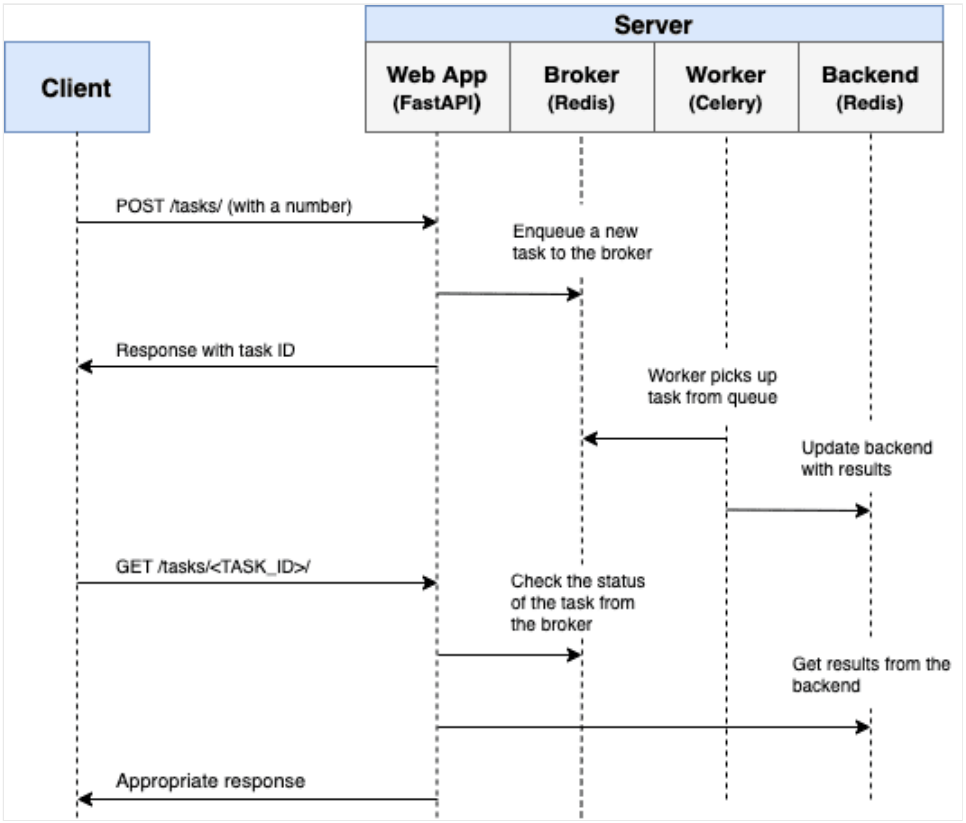
So, when should you use Celery instead of `BackgroundTasks`?

1. **CPU intensive tasks**: Celery should be used for tasks that perform heavy background computations since `BackgroundTasks` runs in the same event loop that serves your app's requests.
2. **Task queue:** If you require a task queue to manage the tasks and workers, you should use Celery. Often you'll want to retrieve the status of a job and then perform some action based on the status -- i.e., send an error email, kick off a different background task, or retry the task. Celery manages all this for you.

Workflow

Our goal is to develop a FastAPI 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 route handler, 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.



Project Setup

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

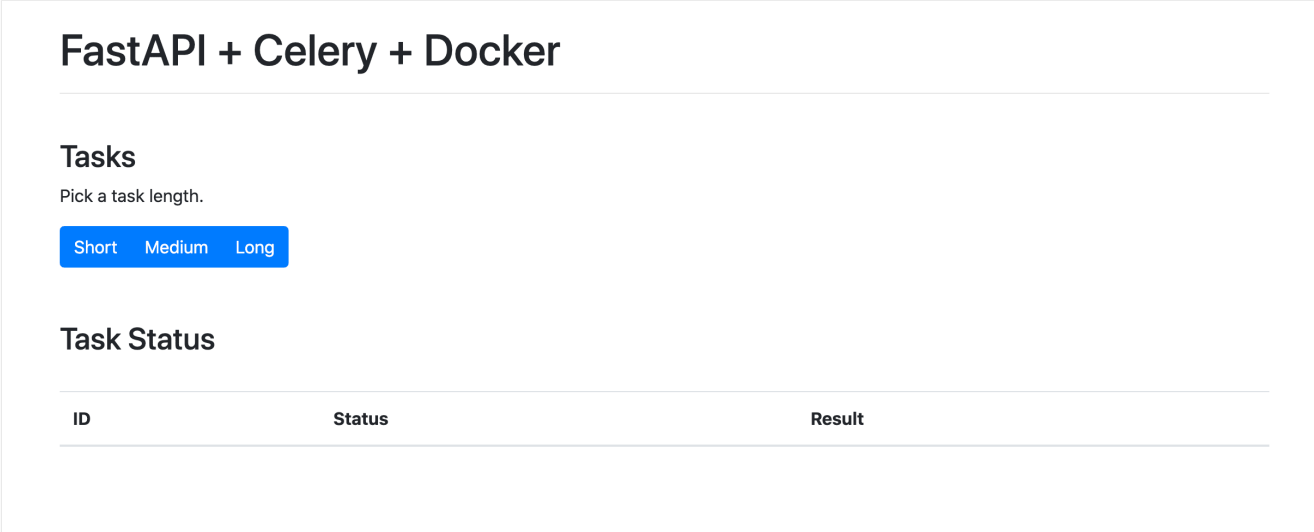
```
$ git clone https://github.com/testdrivenio/fastapi-celery --branch v1 --single-branch
$ cd fastapi-celery
$ git checkout v1 -b master
```

Since we'll need to manage three processes in total (FastAPI, Redis, Celery 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:8004:



Make sure the tests pass as well:

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

================================== test session starts ===================================
platform linux -- Python 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /usr/src/app
collected 1 item

tests/test_tasks.py .                                                              [100%]

=================================== 1 passed in 0.06s ====================================
```

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

```
├── .gitignore
├── LICENSE
├── README.md
├── docker-compose.yml
└── project
    ├── Dockerfile
    ├── main.py
    ├── requirements.txt
    ├── static
    │   ├── main.css
    │   └── main.js
    ├── templates
    │   ├── _base.html
    │   ├── footer.html
    │   └── home.html
    └── tests
        ├── __init__.py
        ├── conftest.py
        └── test_tasks.py
```

Trigger a Task

An `onclick` event handler in *project/templates/home.html* is set up that listens for a button click:

```
<div class="btn-group" role="group" aria-label="Basic example">
  <button type="button" class="btn btn-primary" onclick="handleClick(1)">Short</a>
  <button type="button" class="btn btn-primary" onclick="handleClick(2)">Medium</a>
  <button type="button" class="btn btn-primary" onclick="handleClick(3)">Long</a>
</div>
```

`onclick` calls `handleClick` found in *project/static/main.js*, which sends an AJAX POST request to the server with the appropriate task type: `1`, `2`, or `3`.

```
function handleClick(type) {
  fetch('/tasks', {
    method: 'POST',
    headers: {
```

```
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({ type: type }),
  })
  .then(response => response.json())
  .then(res => getStatus(res.data.task_id));
}
```

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

```
@app.post("/tasks", status_code=201)
def run_task(payload = Body(...)):
    task_type = payload["type"]
    return JSONResponse(task_type)
```

Now comes the fun part -- wiring up Celery!

Celery Setup

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

```
aiofiles==0.6.0
celery==4.4.7
fastapi==0.64.0
Jinja2==2.11.3
pytest==6.2.4
redis==3.5.3
requests==2.25.1
uvicorn==0.13.4
```

> This tutorial uses Celery v4.4.7 since Flower does not support Celery 5.

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
    ports:
      - 8004:8000
    command: uvicorn main:app --host 0.0.0.0 --reload
    volumes:
      - ./project:/usr/src/app
    environment:
      - CELERY_BROKER_URL=redis://redis:6379/0
      - CELERY_RESULT_BACKEND=redis://redis:6379/0
    depends_on:
      - redis

  worker:
    build: ./project
    command: celery worker --app=worker.celery --loglevel=info
    volumes:
      - ./project:/usr/src/app
    environment:
      - CELERY_BROKER_URL=redis://redis:6379/0
      - CELERY_RESULT_BACKEND=redis://redis:6379/0
    depends_on:
      - web
      - redis

  redis:
    image: redis:6-alpine
```

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

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

Next, create a new file called *worker.py* in "project":

```
import os
import time

from celery import Celery


celery = Celery(__name__)
celery.conf.broker_url = os.environ.get("CELERY_BROKER_URL", "redis://localhost:6379")
celery.conf.result_backend = os.environ.get("CELERY_RESULT_BACKEND", "redis://localhost:6379


@celery.task(name="create_task")
def create_task(task_type):
    time.sleep(int(task_type) * 10)
    return True
```

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

> Keep in mind that the task itself will be executed by the Celery worker.

Trigger a Task

Update the route handler to kick off the task and respond with the task ID:

```
@app.post("/tasks", status_code=201)
def run_task(payload = Body(...)):
    task_type = payload["type"]
```

```
    task = create_task.delay(int(task_type))
    return JSONResponse({"task_id": task.id})
```

Don't forget to import the task:

```
from worker import create_task
```

Build the images and spin up the new containers:

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

To trigger a new task, run:

```
$ curl http://localhost:8004/tasks -H "Content-Type: application/json" --data '{"type": 0}'
```

You should see something like:

```
{
  "task_id": "14049663-6257-4a1f-81e5-563c714e90af"
}
```

Task Status

Turn back to the `handleClick` function on the client-side:

```
function handleClick(type) {
  fetch('/tasks', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({ type: type }),
```

```
  })
  .then(response => response.json())
  .then(res => getStatus(res.data.task_id));
}
```

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) {
  fetch(`/tasks/${taskID}`, {
    method: 'GET',
    headers: {
      'Content-Type': 'application/json'
    },
  })
  .then(response => response.json())
  .then(res => {
    const html = `
      <tr>
        <td>${taskID}</td>
        <td>${res.data.task_status}</td>
        <td>${res.data.task_result}</td>
      </tr>`;
    document.getElementById('tasks').prepend(html);
    const newRow = document.getElementById('table').insertRow();
    newRow.innerHTML = html;
    const taskStatus = res.data.task_status;
    if (taskStatus === 'finished' || taskStatus === 'failed') return false;
    setTimeout(function() {
      getStatus(res.data.task_id);
    }, 1000);
  })
  .catch(err => console.log(err));
}
```

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

Update the `get_status` route handler to return the status:

```
@app.get("/tasks/{task_id}")
def get_status(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)
```

Import AsyncResult:

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

Update the containers:

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

Trigger a new task:

```
$ curl http://localhost:8004/tasks -H "Content-Type: application/json" --data '{"type": 1}'
```

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

```
$ curl http://localhost:8004/tasks/f3ae36f1-58b8-4c2b-bf5b-739c80e9d7ff

{
  "task_id": "455234e0-f0ea-4a39-bbe9-e3947e248503",
  "task_result": true,
  "task_status": "SUCCESS"
}
```

Test it out in the browser as well:

Celery Logs

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

```
worker:
  build: ./project
  command: celery worker --app=worker.celery --loglevel=info --logfile=logs/celery.log
  volumes:
    - ./project:/usr/src/app
  environment:
    - CELERY_BROKER_URL=redis://redis:6379/0
    - CELERY_RESULT_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:

```
[2021-05-08 15:32:24,407: INFO/MainProcess] Connected to redis://redis:6379/0
[2021-05-08 15:32:24,415: INFO/MainProcess] mingle: searching for neighbors
[2021-05-08 15:32:25,434: INFO/MainProcess] mingle: all alone
[2021-05-08 15:32:25,448: INFO/MainProcess] celery@365a3b836a91 ready.
[2021-05-08 15:32:29,834: INFO/MainProcess]
    Received task: create_task[013df48c-4548-4a2b-9b22-7267da215361]
[2021-05-08 15:32:39,825: INFO/ForkPoolWorker-7]
    Task create_task[013df48c-4548-4a2b-9b22-7267da215361]
    succeeded in 10.02114040000015s: 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*:

```
aiofiles==0.6.0
celery==4.4.7
fastapi==0.64.0
flower==0.9.7
Jinja2==2.11.3
pytest==6.2.4
redis==3.5.3
requests==2.25.1
uvicorn==0.13.4
```

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

```
dashboard:
  build: ./project
  command:  flower --app=worker.celery --port=5555 --broker=redis://redis:6379/0
  ports:
    - 5556:5555
  environment:
    - CELERY_BROKER_URL=redis://redis:6379/0
    - CELERY_RESULT_BACKEND=redis://redis:6379/0
  depends_on:
    - web
    - redis
    - worker
```

Test it out:

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

Navigate to http://localhost:5556 to view the dashboard. You should see one worker ready to go:

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

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

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

Tests

Let's start with the most basic test:

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

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

```
from worker import create_task
```

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 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /usr/src/app
plugins: celery-4.4.7
collected 2 items / 1 deselected / 1 selected

tests/test_tasks.py .                                                            [100%]

====================== 1 passed, 1 deselected in 60.05s (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("worker.create_task.run")
def test_mock_task(mock_run):
    assert create_task.run(1)
    create_task.run.assert_called_once_with(1)

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

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

Import:

```
from unittest.mock import patch, call
```

Test:

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

================================== test session starts ==================================
platform linux -- Python 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /usr/src/app
```

```
plugins: celery-4.4.7
collected 3 items / 2 deselected / 1 selected

tests/test_tasks.py .                                                    [100%]

============================== 1 passed, 2 deselected in 0.13s ===============================
```

Much quicker!

How about a full integration test?

```
def test_task_status(test_app):
    response = test_app.post(
        "/tasks",
        data=json.dumps({"type": 1})
    )
    content = response.json()
    task_id = content["task_id"]
    assert task_id

    response = test_app.get(f"tasks/{task_id}")
    content = response.json()
    assert content == {"task_id": task_id, "task_status": "PENDING", "task_result": None}
    assert response.status_code == 200

    while content["task_status"] == "PENDING":
        response = test_app.get(f"tasks/{task_id}")
        content = response.json()
    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.

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 FastAPI 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.

Grab the code from the repo.

docker fastapi task 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.

RECOMMENDED TUTORIALS

Stay Sharp with Course Updates

Join our mailing list to be notified about updates and new releases.

Enter your email | Subscribe

TestDriven.io is a proud supporter of open source

10% of profits from our FastAPI and Flask Web Development courses will be donated to the FastAPI and Flask teams, respectively.

Follow our contributions ›

testdriven.io

Follow @testdrivenio

Feedback