Dockerize Your FastAPI and Celery Application

May 9, 2024 • 5 min read
Tags: FastAPI, Celery, Docker, Compose

A while ago I wrote a tutorial about **how to use Celery with FastAPI** to run asynchronous tasks. In this post, I will explain how to dockerize the application and simplify the deployment with Docker Compose.

Make sure you already have Docker installed on your system.

The source code to deploy

To recap, here is the source code from my previous FastAPI and Celery tutorial. We have three files:

- `requirements.txt` - The file that specify the required dependencies.
- `main.py` - Contains the FastAPI application
- `tasks.py` - Contains the Celery tasks

The contents of requirements.txt:

```
fastapi==0.111.0
celery==5.4.0
redis==5.0.4
```

The contents of `main.py`:

```
# main.py
from fastapi import FastAPI
from .tasks import celery, square_root

app = FastAPI()

@app.post('/square_root')
def process(num: float):
  task = square_root.delay(num)
  return {'taskId': task.id}

@app.get('/status/{task_id}')
def status(task_id: str):
    task = celery.AsyncResult(task_id)
    if task.ready():
        return {'status': 'DONE', 'result': task.get()}
    else:
        return {'status': 'IN_PROGRESS'}
```

The contents of `tasks.py`:

```
# tasks.py
import math
import time
import os

from celery import Celery

REDIS_HOST = os.getenv('REDIS_HOST', 'localhost')
REDIS_PORT = os.getenv('REDIS_PORT', 6379)

celery = Celery(
    'tasks',
    broker=f'redis://{REDIS_HOST}:{REDIS_PORT}/0',
    backend=f'redis://{REDIS_HOST}:{REDIS_PORT}/0'
)

@celery.task
def square_root(num: float):
    time.sleep(10)
    return math.sqrt(num)
```

In short, this simple application provides two endpoints:

- `/square_root`: start the task
- `/status/<task_id>`: check the status of the task.

When you make POST request to `/square_root`, it will return a task ID and run the background task to do the calculation. The background task simply use delay to simulate long-running code.

To check the status of the task, make a GET request to `/status/<task_id>`.

Writing the Dockerfile

In this scenario, we only need one Dockerfile for both the API and the worker code. Create a new file named `Dockerfile` and put the following content:

```
FROM python:3.12-slim

WORKDIR /app

COPY . .
```

```
RUN pip install --no-cache-dir -r requirements.txt
```

Let’s break down the contents of the Dockerfile step by step:

- **`FROM python:3.12-slim`**
  This line specifies the base image for the Docker image. In this case, it uses the Python 3.12-slim image as the base, which is a lightweight version of Python.

- **`WORKDIR /app`**
  This line sets the working directory inside the container to `/app`. It means that all subsequent commands will be executed in this directory.

- **`COPY . .`**
  This line copies the entire contents of the current directory (where the Dockerfile resides) to the `/app` directory inside the container. It includes your application code and any other necessary files.

- **`RUN pip install --no-cache-dir -r requirements.txt`**
  This line installs the dependencies specified in the `requirements.txt` file. The `--no-cache-dir` flag ensures that pip doesn’t cache the downloaded packages, reducing the size of the final Docker image.

Writing the Docker Compose file

The Docker Compose file allows you to define and manage the services required for your application in a single configuration file. By using Docker Compose commands, you can easily build and run the entire application stack with just one command.

Create a new file named `docker-compose.yml` and put the following contents:

```
services:
  api:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - 8000:8000
    depends_on:
      - worker
    environment:
      - REDIS_HOST=redis
      - REDIS_PORT=6379
    command: uvicorn main:app --host 0.0.0.0 --port 8000

  worker:
    build:
      context: .
      dockerfile: Dockerfile
    depends_on:
      - redis
    environment:
      - REDIS_HOST=redis
      - REDIS_PORT=6379
    command: celery -A tasks worker --loglevel=info

  redis:
    image: redis:latest
```

Let’s briefly summarize the file.

The **`api`** service is responsible for running the FastAPI application. It depends on the `worker` service, which means it will only start after `worker` is running. The environment variables `REDIS_HOST` and `REDIS_PORT` are set to connect to the Redis service.

The **`worker`** service is responsible for running the Celery worker. It depends on the `redis` service and sets the `REDIS_HOST` and `REDIS_PORT` environment variables to connect to Redis.

The **`redis`** service uses the `redis:latest` image to run a Redis server. It provides the message broker for the Celery worker and the API.

Run the application

Now that everything is in place, you can run the application by using this command:

```
docker compose up
```