Asynchronous Tasks with FastAPI and Celery

Jan 28, 2024 • 5 min read

Tags: Python, FastAPI, Celery

Overview

When you have a long running Python function that you want to expose via an API endpoint, you might want to run the function asynchronously to avoid timeouts. Running a function asynchronously means it won't block the rest of your code. Instead of waiting the function to finish the task, your program can continue executing other tasks, making it more responsive.

In this post, I will show you how to use **Celery** to execute tasks asynchronously in your FastAPI application. We will use the function below to simulate long-running code:

```
import math
import time

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

In the real world applications, this might be sending emails to users, processing long duration videos, or training ML models.

A quick intro to Celery

Celery is a task queuing in Python. You use Celery to manage and distribute tasks across worker processes or machines. You define tasks in your Python code, and Celery takes care of running them asynchronously. Its a perfect tool for handling things such as time-consuming operations or external API calls.

Celery requires a message broker to transmit messages between the client (where tasks are initiated) and the workers (where tasks are executed). There are many options for messages brokers to be used with Celery. In this post, we will use **Redis**.

Write the API endpoints

We will use FastAPI to build the API endpoints. We will write two API endpoints:

- `/square_root` to initiate the execution of the function.
- `/status/<task_id>` to get the status of the task and retrieve the result.

Let's get started. Install the required packages with pip:

```
pip install fastapi uvicorn celery redis
```

Create a new file named `main.py` and write dummy API endpoints with FastAPI:

```
# main.py
from fastapi import FastAPI

app = FastAPI()

@app.post('/square_root')
def process(num: float):
    return {'taskId': None}

@app.get(f'/status/{task_id}')
def status(task_id: str):
    return {'status': 'IN_PROGRESS'}
```

Run the development server:

```
uvicorn main:app --reload
```

The server will run on localhost and listening on port 8000. You can test the two endpoints we created earlier with **Curl** or **HTTPie** and see that the endpoints return the dummy responses.

Write the Celery worker

We will wrap the long-running function in a Celery worker file. Create a new file named `tasks.py` and modify the contents to the following:

```
# 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 this file, we configure Celery with Redis as the message broker. The `@celery.task` decorator transforms the `square_root` function into a Celery task, allowing it to be scheduled and executed asynchronously.

Run the following command from another terminal:

```
celery -A tasks worker --loglevel=info
```

The command will start a Celery worker that will process tasks defined in the `tasks` module. It also sets the logging level to “info” to provide more detailed information about the tasks being processed.

Execute the Celery tasks from API endpoints

Now that we have basic API endpoints and the Celery task ready, we need to wire up the two components. Open `main.py` and update its content:

```
# main.py

...

from .tasks import celery, square_root


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


@app.get(f'/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'}
```

In the `/square_root` endpoint, the code executes the `square_root` function asynchronously and return the ID of the task. The client then need to check the status of the task using that ID.

In the `/status/<task_id>` endpoint, the code accepts a task ID and returns its status. If the task result is already available, the endpoint will also return it.

Next steps

After you build your FastAPI and Celery application, you might want to **dockerize the code and use Docker Compose** to simplify the deployment.