Let's check the coverage report for `routes/events.py`. Click on it to display it.

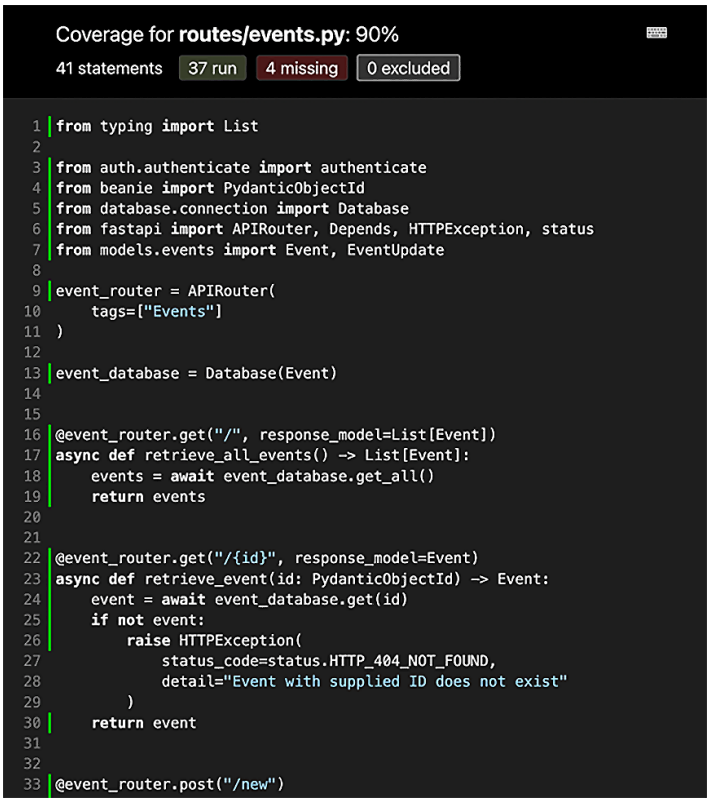

Figure 8.19 – Coverage report showing executed code in green and untouched code in red

Summary

In this chapter, you have successfully tested the API by writing tests for the authentication routes and the CRUD route. You have learned what testing is and how to write tests with `pytest`, a fast testing library built for Python applications. You also learned what `pytest` fixtures are and used them in creating reusable access tokens and database objects, as well as preserving the application instance throughout the testing session. You were able to assert the responses of your API HTTP requests and verify the behavior of your API. Finally, you learned how to generate a coverage report for your tests and distinguish the blocks of code run during the testing session.

Now that you have been equipped with the knowledge of testing web APIs, you are ready to publish your application to the World Wide Web through a deployment channel. In the next and final chapter, you'll learn how to containerize your application and deploy your locally using Docker and docker-compose.

9
Deploying FastAPI Applications

In the last chapter, you learned how to write tests for API endpoints created in a FastAPI application. We started by learning what testing means and walked through the basics of unit testing using the `pytest` library. We also looked at how to eliminate repetition and reuse test components with fixtures and then proceeded to set up our test environment. We wrapped up the last chapter by writing tests for each endpoint and then testing them alongside checking the test coverage reports after testing.

In this chapter, you'll learn how to deploy your FastAPI application locally using **Docker** and **docker-compose**. A brief section is also added with external resources to deploy your application on serverless platforms of your choice.

In this chapter, we'll be covering the following topics:

- Preparing for deployment
- Deploying with Docker
- Deploying Docker images

Technical requirements

The code used in this chapter can be found at `https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI/tree/main/ch09/planner`.

Preparing for deployment

Deployment usually marks the end of an application's life cycle. Before deploying our applications, we must make sure the right settings required for a smooth deployment are put in place. These settings include ensuring the application dependencies are up to date in the `requirements.txt` file, configuring environment variables, and so on.

Managing dependencies

In a few earlier chapters, we installed packages such as `beanie` and `pytest`. These packages are absent from the `requirements.txt` file, which serves as the dependency manager for our application. It is important that the `requirements.txt` file is kept up to date.

In Python, the list of packages used in a development environment can be retrieved using the `pip freeze` command. The `pip freeze` command returns a list of all packages installed directly and the sub-dependencies for each package installed. Luckily, the `requirements.txt` file can be maintained manually, enabling us to list only the main packages, thereby making dependency management easier.

Let's list the dependencies used by the application before overwriting the `requirements.txt` file:

```
(venv)$ pip freeze
anyio==3.5.0
asgi-lifespan==1.0.1
asgiref==3.5.0
attrs==21.4.0
bcrypt==3.2.2
cffi==1.15.0
python-multipart==0.0.5
...
```

The command returns several dependencies, some of which we do not use directly in the application. Let's manually fill the `requirements.txt` file with the packages we will be using:

requirements.txt

```
fastapi==0.78.0
bcrypt==3.2.2
beanie==1.11.1
email-validator==1.2.1
httpx==0.22.0
Jinja2==3.0.3
motor==2.5.1
passlib==1.7.4
pytest==7.1.2
python-multipart==.0.0.5
python-dotenv==0.20.0
python-jose==3.3.0
sqlmodel==0.0.6
uvicorn==0.17.6
```

In this code block, we have populated the `requirements.txt` file with the dependencies used directly in our application.

Configuring environment variables

We used environment variables in *Chapter 6*, *Connecting to a Database*. Environment variables can be injected during deployment, as we'll see in the next section.

> **Note**
>
> It is important to note that environment variables are to be properly handled and kept out of version control systems such as GitHub.

Now that we have covered the necessary steps in preparation for our deployments, let's proceed to deploying our application locally with Docker in the next section.

Deploying with Docker

In *Chapter 1*, *Getting Started with FastAPI*, you were introduced to the basics of Docker and the Dockerfile. In this section, you'll be writing a Dockerfile for the event planner API.

Docker is the most popular technology used for containerization. Containers are self-contained systems consisting of packages, code, and dependencies that enable them to run in different environments with little to no dependence on their running environment. Docker uses Dockerfiles for the containerization process.

Docker can be used for local development as well as for deploying applications to production. We'll only be looking at local deployment in this chapter, and links to official guides on deploying to cloud services will be included as well.

For managing applications with multiple containers, such as an application container and a database container, the compose tool is used. Compose is a tool used to manage multi-container Docker applications defined in the configuration file, usually `docker-compose.yaml`. The compose tool, docker-compose, comes installed with the Docker engine.

Writing the Dockerfile

A Dockerfile contains a set of instructions employed to build a Docker image. The Docker image built can then be distributed to registries (private and public), deployed to cloud servers such as AWS and Google Cloud, and used on different operating systems by creating a container.

Now that we know what a Dockerfile does, let's create a Dockerfile to build the application's image. In the project directory, create the `Dockerfile` file:

```
(venv)$ touch Dockerfile
```

Dockerfile

```
FROM python:3.10

WORKDIR /app

COPY requirements.txt /app

RUN pip install --upgrade pip && pip install -r /app/
requirements.txt
```