5. Lastly, let's delete the event:

```
(venv)$ curl -X 'DELETE' \
  'http://0.0.0.0:8080/event/624daab1585059e8a3fa77ac'
  \
  -H 'accept: application/json'
```

Here is the response received to the request:

```
{
  "message": "Event deleted successfully."
}
```

6. Now that we have tested the routes for the events, let's create a new user and then sign in:

```
(venv)$ curl -X 'POST' \
  'http://0.0.0.0:8080/user/signup' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "email": "fastapi@packt.com",
  "password": "strong!!!",
  "events": []
}'
```

The request returns a response:

```
{
  "message": "User created successfully"
}
```

Running the request again returns an `HTTP 409` error, indicating a conflict:

```
{
  "detail": "User with email provided exists already."
}
```

We originally designed the route to check for existing users to avoid duplicates.

7. Now, let's send a `POST` request to sign in the user we just created:

```
(venv)$ curl -X 'POST' \
  'http://0.0.0.0:8080/user/signin' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "email": "fastapi@packt.com",
  "password": "strong!!!"
}'
```

The request returns an `HTTP 200` success message:

```
{
  "message": "User signed in successfully."
}
```

We have successfully implemented CRUD operations using the Beanie library.

Summary

In this chapter, we learned how to add SQL and NoSQL databases using SQLModel and Beanie respectively. We made use of all our knowledge from the previous chapters. We also tested the routes to ensure that they are working as planned.

In the next chapter, you will be introduced to securing your application. You will first be taught the basics of authentication as well as the various authentication methods available to FastAPI developers. You will then implement an authentication system that relies on **JSON Web Token** (**JWT**) and secure the routes to create, update, and delete events. Lastly, you will modify the route, to create events to allow the linking of events to a user.

7
Securing FastAPI Applications

In the last chapter, we looked at how to connect a FastAPI application to a SQL and NoSQL database. We successfully implemented database methods and updated the existing routes to enable interactions between the application and the database. However, the planner application continues to allow anybody to add an event as opposed to only authenticated users. In this chapter, we will secure the application using **JSON Web Token (JWT)** and restrict some event operations to only authenticated users.

Securing an application involves the addition of security measures to restrict access to application functionalities from unauthorized entities to prevent hacks or illegal modifications of the application. Authentication is the process of verifying the credentials passed by an entity and authorization simply means giving an entity permission to perform designated actions. When credentials have been verified, the entity is then authorized to carry out various actions.

By the end of this chapter, you will be able to add an authentication layer to a FastAPI application. This chapter will explain the processes for securing passwords by hashing them, adding an authentication layer, and securing routes from unauthenticated users. In this chapter, we'll be covering the following topics:

- Authentication methods in FastAPI
- Securing the application with OAuth2 and JWT
- Protecting routes using dependency injection
- Configuring CORS

Technical requirements

To follow along, the MongoDB database component is required. The installation procedures for your operating system can be found in their official documentation. The code used in this chapter can be found at `https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI/tree/main/ch07/planner`.

Authentication methods in FastAPI

There are several authentication methods available in FastAPI. FastAPI supports the common authentication methods of basic HTTP authentication, cookies, and bearer token authentication. Let's briefly look at what each method entails:

- **Basic HTTP authentication**: In this authentication method, the user credentials, which is usually a username and password, are sent via an `Authorization` HTTP header. The request in turn returns a `WWW-Authenticate` header containing a `Basic` value and an optional realm parameter, which indicates the resource the authentication request is made to.
- **Cookies**: Cookies are employed when data is to be stored on the client side, such as in web browsers. FastAPI applications can also employ cookies to store user data, which can be retrieved by the server for authentication purposes.
- **Bearer token authentication**: This method of authentication involves the use of security tokens called bearer tokens. These tokens are sent alongside the `Bearer` keyword in an `Authorization` header request. The most used token is JWT, which is usually a dictionary comprising the user ID and the token's expiry time.

Every authentication method listed here has its specific use cases as well as its pros and cons. However, in this chapter, we'll be making use of bearer token authentication.

Authentication methods are injected into FastAPI applications as dependencies that are called at runtime. This simply means when authentication methods are defined, they are dormant until injected into their place of use. This activity is called **Dependency Injection**.

Dependency injection

Dependency injection is a pattern where an object – in this case, a function – receives an instance variable needed for the further execution of the function.

In FastAPI, dependencies are injected by declaring them in the path operation function arguments. We have been using dependency injection in previous chapters. Here's an example from the previous chapter where we retrieve the email field from the user model passed to the function:

```
@user_router.post("/signup")
async def sign_user_up(user: User) -> dict:
    user_exist = await User.find_one(User.email == user.email)
```

In this code block, the dependency defined is the `User` model class, which is injected into the `sign_user_up()` function. By injecting the `User` model into the user function argument, we can easily retrieve the attributes of the object.

Creating and using a dependency

In FastAPI, a dependency can be defined as either a function or a class. The dependency created gives us access to its underlying values or methods, eliminating the need to create these objects in the functions inheriting them. Dependency injection helps in reducing code repetition in some cases, such as in enforcing authentication and authorization.

An example dependency is defined as follows:

```
async def get_user(token: str):
    user = decode_token(token)
    return user
```

This dependency is a function that takes `token` as the argument and returns a `user` parameter from an external function, `decode_token`. To use this dependency, the dependent function argument declared is set to have a `Depends` parameter, for example:

```
from fastapi import Depends

@router.get("/user/me")
```