6
Connecting to a Database

In the last chapter, we looked at how to structure a FastAPI application. We successfully implemented some routes and models for our application and tested the endpoints. However, the application still uses an in-app database to store the events. In this chapter, we will migrate the application to use a proper database.

A database can be simply referred to as a storehouse for data. In this context, a database enables us to store data permanently, as opposed to an in-app database, which is wiped off upon any app restart or crash. **A database is a table housing columns, referred to as fields, and rows, referred to as records.**

By the end of this chapter, you will be equipped with the knowledge of how to connect a FastAPI application to a database. This chapter will explain how to connect to a SQL database using **SQLModel** and a **MongoDB** database via **Beanie**. (However, the application will make use of MongoDB as its primary database in later chapters.) In this chapter, you'll be covering the following topics:

- Setting up SQLModel
- CRUD operations on a SQL database using SQLModel
- Setting up MongoDB
- CRUD operations on MongoDB using Beanie

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/ch06/planner`.

Setting up SQLModel

The first step to integrate a SQL database into our planner application is to install the SQLModel library. The SQLModel library was built by the creator of FastAPI and is powered by Pydantic and SQLAlchemy. Support from Pydantic will make it easy for us to define models, as we learned in *Chapter 3*, *Response Models and Error Handling*.

Since we'll be implementing both SQL and NoSQL databases, we'll create a new GitHub branch for this section. In your terminal, navigate to the project directory, initialize a GitHub repository, and commit the existing files:

```
$ git init
$ git add database models routes main.py
$ git commit -m "Committing bare application without a
database"
```

Next, create a new branch:

```
$ git checkout -b planner-sql
```

Now, we are ready to set up SQLModel in our application. From your terminal, activate the virtual environment and install the SQLModel library:

```
$ source venv/bin/activate
(venv)$ pip install sqlmodel
```

Before diving into adding a database to our planner application, let's look at some of the methods contained in SQLModel that we'll be using in this chapter.

Tables

A table is essentially an object that contains data stored in a database – for example, events data will be stored in an event table. The table will consist of columns and rows where the data will eventually be stored.

To create a table using SQLModel, a table model class is first defined. As with Pydantic models, the table is defined but, this time around, as subclasses of the SQLModel class. The class definition also takes another config variable, `table`, to indicate that this class is a SQLModel table.

The variables defined in the class will represent the columns by default unless denoted as a field. Let's look at how the event table will be defined:

```
class Event(SQLModel, table=True):
    id: Optional[int] = Field(default=None,
    primary_key=True)
    title: str
    image: str
    description: str
    location: str
    tags: List[str]
```

In this `table` class, all the variables defined are columns except `id`, which has been defined as a field. Fields are denoted using the `Field` object from the SQLModel library. The `id` field is also the primary key in the database table.

> **What Is a Primary Key?**
> A primary key is a unique identifier for a record contained in a database table.

Now that we have learned what tables are and how to create them, let's look at rows in the next section.

Rows

Data sent to a database table is stored in rows under specified columns. To insert data into the rows and store them, an instance of the table is created and the variables are filled with the desired input. For example, to insert event data into the events table, we'll create an instance of the model first:

```
new_event = Event(title="Book Launch",
                  image="src/fastapi.png",
                  description="The book launch event will
                  be held at Packt HQ, Packt city",
```

```
                    location="Google Meet",
                    tags=["packt", "book"])
```

Next, we create a database transaction using the `Session` class:

```
with Session(engine) as session:
    session.add(new_event)
    session.commit()
```

The preceding operation may seem alien to you. Let's look at what the **Session** class is and what it does.

Sessions

A session object handles the interaction from code to a database. It primarily acts as an intermediary in executing operations. The `Session` class takes an argument that is the instance of a SQL engine.

Now that we have learned how tables and rows are created, we will look at how a database is created. Some of the methods of the `session` class we'll be using in this chapter include the following:

- `add()`: This method is responsible for adding a database object to memory pending further operations. In the previous code block, the `new_event` object is added to the session's memory, waiting to be committed into the database by the `commit()` method.
- `commit()`: This method is responsible for flushing transactions present in the session.
- `get()`: This method takes two parameters – the model and the ID of the document requested. This method is used to retrieve a single row from a database.

Now that we know how to create tables, rows, and columns, as well as insert data using the `Session` class, let's move on to creating a database and performing CRUD operations in the next section.

Creating a database

In SQLModel, connecting to a database is done via a SQLAlchemy engine. The engine is created by the `create_engine()` method, imported from the SQLModel library.

The `create_engine()` method takes the database URL as the argument. The database URL is in the form of `sqlite:///database.db` or `sqlite:///database.sqlite`. It also takes an optional argument, `echo`, which when set to `True` prints out the SQL commands carried out when an operation is executed.

However, the `create_engine()` method alone isn't sufficient to create a database file. To create the database file, the `SQLModel.metadata.create_all(engine)` method whose argument is an instance of the `create_engine()` method is invoked, such as the following:

```
database_file = "database.db"
engine = create_engine(database_file, echo=True)
SQLModel.metadata.create_all(engine)
```

The `create_all()` method creates the database as well as the tables defined. It is important to note that the file containing the tables is imported into the file where the database connection takes place.

In our planner application, we perform CRUD operations for events. In the database folder, create the following file:

`connection.py`

In this file, we'll configure the necessary data for the database:

```
(venv)$ touch database/connection.py
```

Now that we have created the database connection file, let's create the functions required to connect our application to the database:

1. We'll start by updating the events model class defined in `models/events.py` to a SQLModel table model class:

```
from sqlmodel import JSON, SQLModel, Field, Column
from typing import Optional, List

class Event(SQLModel, table=True):
    id: int = Field(default=None, primary_key=True)
    title: str
    image: str
    description: str
    tags: List[str] = Field(sa_column=Column(JSON))
```