Building a Data API with FastAPI and SQLAlchemy

How to use modular database logic to load data into a database and perform CRUD operations in an App

Edward Krueger
May 8 · 6 min read ★

By: Edward Krueger Data Scientist and Instructor and Douglas Franklin Teaching Assistant and Technical Writer.

Link to Github repo with app code: https://github.com/edkrueger/sars-fastapi

Photo by Chris Liverani on Unsplash

What is FastAPI?

FastAPI is a high-performance API based on Pydantic and Starlette. FastAPI integrates well with many packages, including many ORMs. With FastAPI, you can use most relational databases. FastAPI easily integrates with SQLAlchemy and SQLAlchemy supports PostgreSQL, MySQL, SQLite, Oracle, Microsoft SQL Server and others.

Other python microservice frameworks like Flask don't integrate with SQLAlchemy easily. It is common to use Flask with a package called Flask-SQLAlchemy. Flask-SQLAlchemy isn't necessary and has problems of its own. For more information on this, give this article a read!

There is no FastAPI-SQLALchemly because FastAPI integrates well with vanilla SQLAlchemy!

Modular App Structure

When writing an app, it is best to create independent and modular python code. Here we will discuss the following constituent files of our app, database.py, models.py, schemas.py, main.py, and load.py.

Ideally, you should only have to define your database models once! Using SQLAlchemy's `declarative_base()` and `Base.metadata.create_all()` allows you to write just one class per table to use in the app, to use in Python outside of the app and to use in the database. With a separate **database.py** and **models.py** file, we establish our database table classes and connection a single time, then call them later as needed.

To avoid confusion between the SQLAlchemy *models* and the Pydantic *models*, we will have the file `models.py` with the SQLAlchemy models, and the file `schemas.py` with the Pydantic models. Also of note is that SQLAlchemy and Pydantic use slightly different syntax to define models, as seen in the below files.

Database.py

Here is the file that defines our database connection using SQLAlchemy.

```
import os

```

```python
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = os.getenv("DB_CONN")

engine = create_engine(SQLALCHEMY_DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()
```

database.py hosted with ♡ by GitHub view raw

database.py

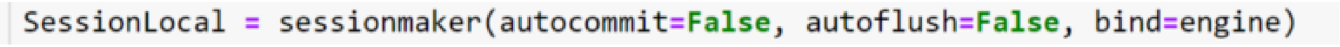
Declarative Base and MetaData

The `declarative_base()` base class contains a `MetaData` object where newly defined `Table` objects are collected. This MetaData object is accessed when we call the line `models.Base.metadata.create_all()` to create all of our tables.

Session Local: Handling Threading Issues

SQLAlchemy includes a helper object that helps with the establishment of user-defined `Session` scopes. This is useful for eliminating threading issues across your app.

To create a session, below we use the `sessionmaker` function and pass it a few arguments. Sessionmaker is a factory for initializing new Session objects. Sessionmaker initializes these sessions by requesting a connection from the engine's connection pool and attaching a connection to the new Session object.



```python
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
```

SessionLocal from database.py

Initializing a new session object is also referred to as “checking out” a connection. The database stores a list of these connections/processes. So when you begin a new session, be mindful you are also starting a new process within the database. If the database doesn’t have these connections closed, there is a maximum number of connections that can be reached. The database will eventually kill idle processes like stale connections; however, it can take hours before that happens.

SQLAlchemy has some pool options to prevent this, but removing the connections when they are no longer needed is best! The FastAPI docs include a `get_db()` function that allows a route to use the same session through a request and then close it when the request is finished. Then `get_db()` creates a new session for the next request.

Once we have our database connection and session set up, we are ready to build our other app components.

Models.py

Notice that we import the `Base` class, defined in the database.py file above, into the models.py file below to use `declarative_base()`.

```python
from sqlalchemy import Column, Integer, String
from sqlalchemy.types import Date
from .database import Base


class Record(Base):
    __tablename__ = "Records"

    id = Column(Integer, primary_key=True, index=True)
    date = Column(Date)
    country = Column(String(255), index=True)
    cases = Column(Integer)
    deaths = Column(Integer)
    recoveries = Column(Integer)
```

models.py hosted with ♡ by GitHub view raw

models.py

This file creates the model or schema for the table `Records` in our database.

Using SQLAlchemy's `declarative_base()` allows you to write just one model for each table that app uses. That model is then used in Python outside of the app and in the database.

Having these separate Python files is good because you can use the same model to query or load data outside of an app. Additionally, you'll have one version of each model, which simplifies development.

These modular Python files can be used to reference the same models or databases in data pipelines, report generation, and anywhere else they are needed.

Schemas.py

Here we write our schema for Pydantic. Remember that FastAPI is built upon Pydantic. The primary means of defining objects in Pydantic is via models that inherit from `BaseModel`.

Pydantic guarantees that the data fields of the resultant model conform to the field types we have defined, using standard modern Python types, for the model.

```python
from datetime import date
from pydantic import BaseModel


class Record(BaseModel):
    id: int
    date: date
    country: str
    cases: int
    deaths: int
    recoveries: int

    class Config:
        orm_mode = True
```

schemas.py hosted with ♡ by GitHub view raw

schemas.py

The line `orm_mode = True` allows the app to take ORM objects and translate them into responses automatically. This automation saves us from manually taking data out of ORM, making it into a dictionary, then loading it in with Pydantic.

Main.py

Here is where we bring all the modular components together.

After importing all of our dependencies and modular app components we call `models.Base.metadata.create_all(bind=engine)` to create our models in the database.

Next, we define our app and set up CORS Middleware.

```python
from typing import List

from fastapi import Depends, FastAPI, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from sqlalchemy.orm import Session
from starlette.responses import RedirectResponse
from starlette.responses import RedirectResponse

from . import models, schemas
from .database import SessionLocal, engine

models.Base.metadata.create_all(bind=engine)

app = FastAPI()


app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
    allow_headers=["*"],
    allow_credentials=True,
)

# Dependency
def get_db():
    try:
        db = SessionLocal()
        yield db
    finally:
        db.close()


@app.get("/")
def main():
    return RedirectResponse(url="/docs/")


@app.get("/records/", response_model=List[schemas.Record])
def show_records(db: Session = Depends(get_db)):
    records = db.query(models.Record).all()
    return records
```

main.py

CORS Middleware

CORS or "Cross-Origin Resource Sharing" refers to the situations where a frontend running in a browser has JavaScript code that communicates with a backend, and the backend is in a different "origin" than the frontend.

To configure CORS middleware in your **FastAPI** application

- Import `CORSMiddleware` .
- Create a list of allowed origins as strings, i.e., “http://localhost,” “http://localhost:8080.”
- Add it as a “middleware” to your **FastAPI** application.

You can also specify if your backend allows:

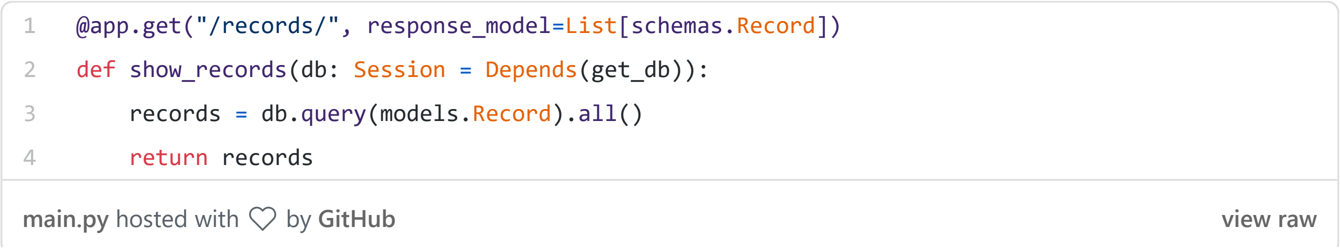
- Credentials (Authorization headers, Cookies, etc.).
- Specific HTTP methods (`POST` , `PUT`) or all of them with the wildcard `"*"` .
- Specific HTTP headers or all of them with the wildcard `"*"` .

For everything to work correctly, it’s best to specify the allowed origins explicitly. Additionally, for security reasons, it is good practice to be explicit about which origins may access your app.

Properly setting up the CORS middleware will eliminate any CORS issues within your app.

The `get_db()` function ensures that any route passed this function ought to have our SessionLocal database connection when needed and that the session is closed after use.

The `/records` route is for viewing our app’s data. Notice that we use schemas.py for the frontend and models.py to query our backend in this route.



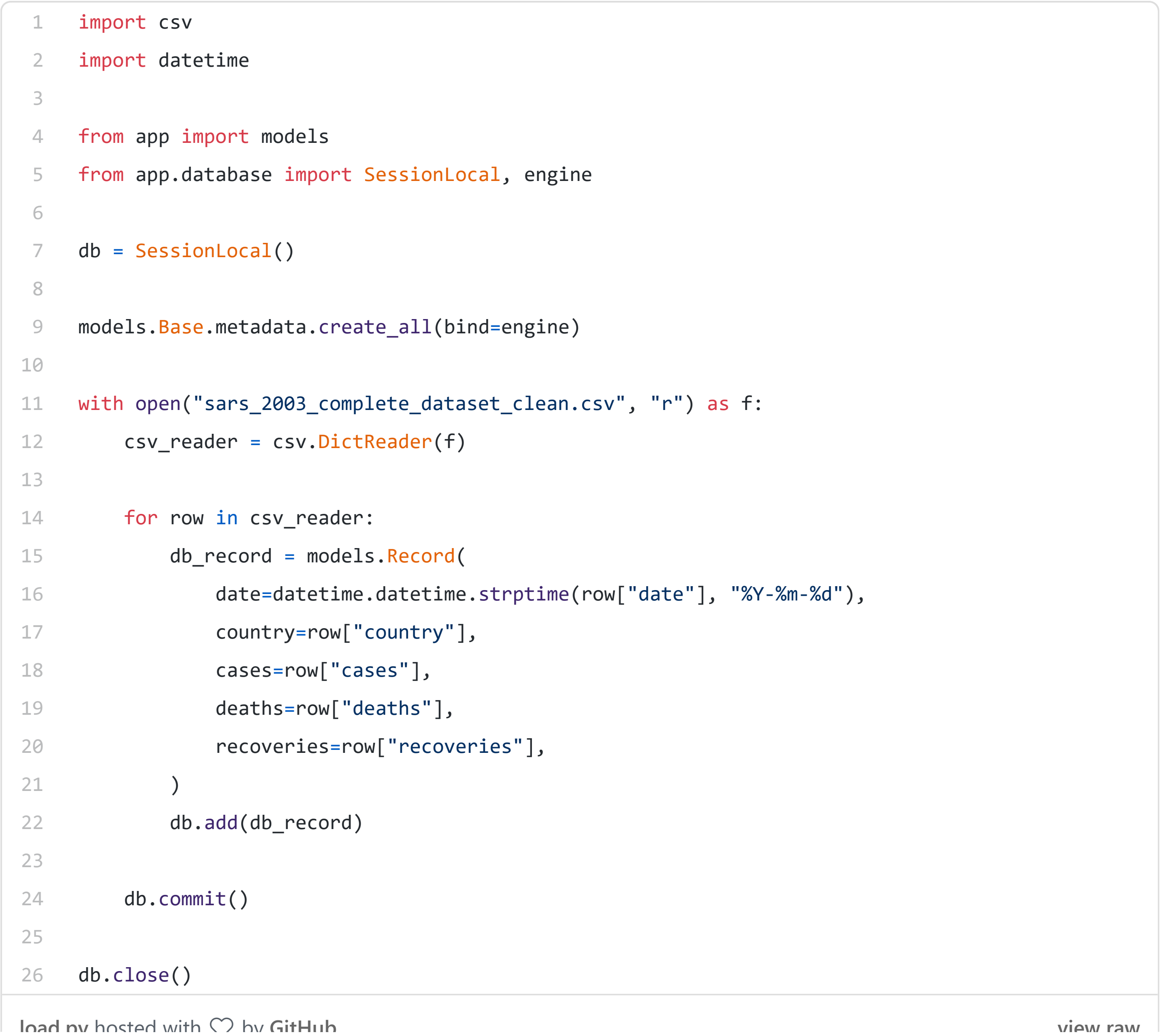
```python
@app.get("/records/", response_model=List[schemas.Record])
def show_records(db: Session = Depends(get_db)):
    records = db.query(models.Record).all()
    return records
```

FastAPI /records route

External Data Loading

Instead of using the app to load data, we load the database with a separate Python file.

Here is an example Python file that reads data from a CSV and inserts that data into a database.



```python
import csv
import datetime

from app import models
from app.database import SessionLocal, engine

db = SessionLocal()

models.Base.metadata.create_all(bind=engine)

with open("sars_2003_complete_dataset_clean.csv", "r") as f:
    csv_reader = csv.DictReader(f)

    for row in csv_reader:
        db_record = models.Record(
            date=datetime.datetime.strptime(row["date"], "%Y-%m-%d"),
            country=row["country"],
            cases=row["cases"],
            deaths=row["deaths"],
            recoveries=row["recoveries"],
        )
        db.add(db_record)

    db.commit()

db.close()
```

load.py

Notice that we import the models, our custom session SessionLocal, and our engine that we've defined in other Python files. Then we read the CSV and using the `models.Record` schema, add `db_record` to the database through the `SessionLocal()` connection.

Database Loading

If your app is set up properly, including your database connection string you may call:

```
python load.py
```

This will load your local or remote database without ever having to run the app!

Testing the app

To run the app locally using a remote DB, in the terminal run:

```
uvicorn app.main:app
```

This runs your app locally. This local instance is connected to the cloud database we just loaded, so check the /records route to see the data!

Conclusion

The modular app structure used allows us to define a model once then use it as needed. This practice makes development much easier because you only have one file to debug if something goes awry. Additionally, your code will be much more reusable and ready for another project!

For your next data project, whether it is a dashboard, data journal, or an API, be sure to give FastAPI a try! It is very fast and easy to pick up, and they have excellent documentation to guide you along the way.

We hope this has been helpful, thank you for reading, and best of luck!

API Database Software Development Sqlalchemy Apps