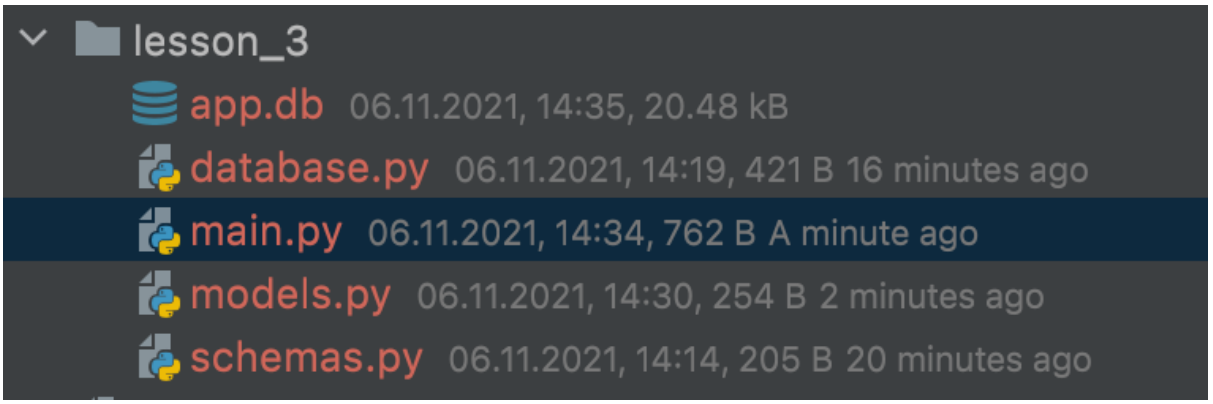
Работаем с БД

В прошлом уроке мы научились обрабатывать входящие параметры. Неплохо бы уметь делать с ними что-то ещё, например сохранять их в базе.

Давайте для начала опишем структуру проекта. Создадим ряд модулей.



Модуль database будет отвечать за всю логику работы с базой данных: URL, создание объекта базы, работа с метаданными и так далее. Модуль models будет содержать модели SQLAlchemy, модуль schemas будет хранить модели pydantic и наконец модуль main — наши эндпоинты.

Как всегда, сначала установим недостающие библиотеки:

```
sqlalchemy==1.4.26
aiosqlite==0.17.0
greenlet==1.1.2
```

Сначала установим модуль SQLAlchemy. Мы с ним уже умеем работать и знаем, что он работает в синхронной манере. Но с недавнего времени этот модуль научился работать асинхронно. Так что нам очень повезло. Раньше бы нам пришлось устанавливать сторонние библиотеки, которые бы запускали синхронные запросы к базе в асинхронной манере. Плюс нужно установить пару вспомогательных библиотек, которые позволят делать сами sql-запросы к базе асинхронно, так называемый sql-драйвер.

Приступим к реализации. Начнем с модуля database.

```
from sqlalchemy.ext.asyncio import create_async_engine,
AsyncSession
from sqlalchemy.ext.declarative import declarative_base
```

```
from sqlalchemy.orm import sessionmaker

DATABASE_URL = "sqlite:///./app.db"

engine = create_async_engine(
   DATABASE_URL, connect_args={"check_same_thread": False}
)
# expire_on_commit=False will prevent attributes from being
expired
# after commit.
async_session = sessionmaker(
   engine, expire_on_commit=False, class_=AsyncSession
)

Base = declarative_base()
```

Тут нет ничего для нас нового, за исключением инициализации слегка других объектов для работы с базой асинхронно.

Создаём engine, session и объект базы.

Теперь опишем модели.

```
from sqlalchemy import Column, String, Integer

from database import Base

class Book(Base):
   __tablename__ = 'Book'
   id = Column(Integer, primary_key=True, index=True)
   title = Column(String, index=True)
   author = Column(String, index=True)
```

Тут тоже ничего для нас нового нет: описываем таблицу, используя как родительский класс объект базы.

Теперь приступим к описанию схем.

```
from pydantic import BaseModel
```

```
class BaseBook(BaseModel):
    title: str
    author: str

class BookIn(BaseBook):
    ...

class BookOut(BaseBook):
    id: int

    class Config:
        orm_mode = True
```

Схема, которая придёт в эндпоинт, довольно очевидна — это мы уже делали. Но вот со схемой, которая вернётся пользователю в ответ, всё чуточку сложнее. Результатом post-операции будет созданная в базе запись, у которой появится ID. Именно поэтому мы вынесли в базовый класс все общие атрибуты — так мы избегаем дублирования кода. И тут самая важная деталь: классы, объявленные с помощью базового класса из модуля pydantic, позволяют указать внутри себя специальный объект конфиг. И вот в нём мы говорим, что этот класс будет использован для сериализации объектов ORM.

И наконец последнее: опишем модуль с эндпоинтами.

```
from typing import List

from fastapi import FastAPI
from sqlalchemy.future import select

import models
import schemas
from database import engine, session



app = FastAPI()


@app.on_event("startup")
async def shutdown():
    async with engine.begin() as conn:
        await conn.run_sync(models.Base.metadata.create_all)
```

```
@app.on_event("shutdown")
async def shutdown():
    await engine.dispose()
```

Первым делом создаём приложение FastAPI.
И сразу видим очень интересную фичу приложения: мы можем запрограммировать некоторое поведение на момент запуска и выключения приложения.
Прежде всего нужно создать базу и таблицы, которые мы только что описали. Но метод create_all по своей реализации — синхронный метод. Так что нам нужно его вызвать асинхронно. Именно это мы и делаем, когда вызываем run_sync метод у полученного в асинхронном контекстном менеджере соединения. В этот метод мы передаём синхронную функцию, которую нужно дождаться. Но тут же можно сконфигурировать логирование, создать соединения с другими сервисами и так далее.
Ну и аналогично, при выключении приложения вы разрываем соединение с базой.

Теперь опишем сами эндпоинты.

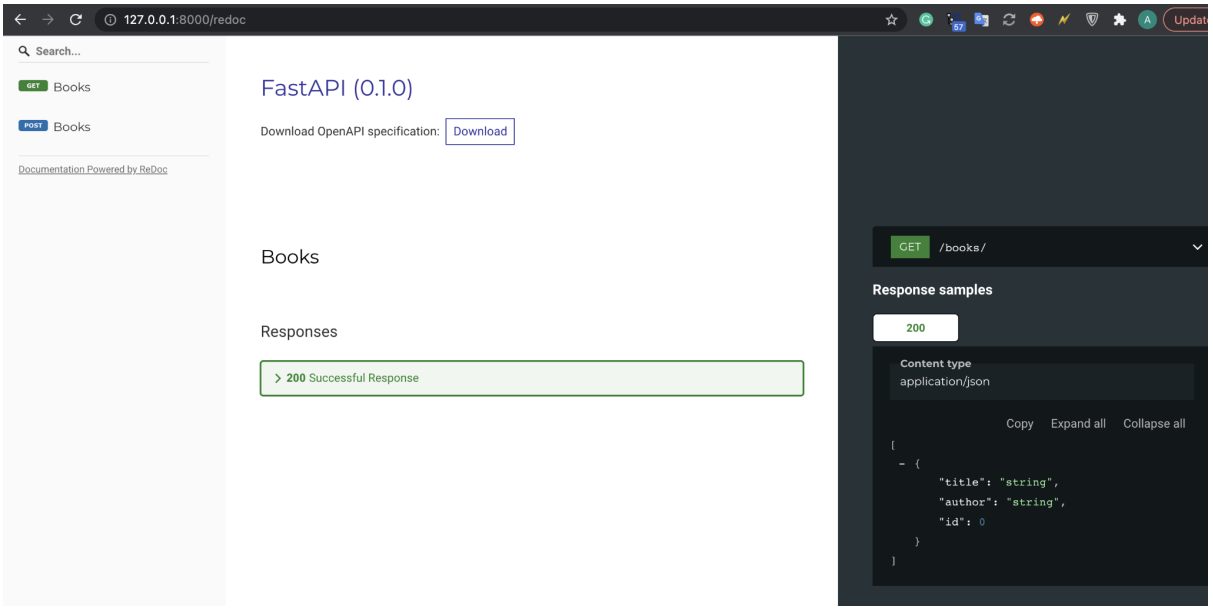
```
@app.post('/books/', response_model=schemas.BookOut)
async def books(book: schemas.BookIn) -> models.Book:
    new_book = models.Book(**book.dict())
    async with session.begin():
        session.add(new_book)
    return new_book


@app.get('/books/', response_model=List[schemas.BookOut])
async def books() -> List[models.Book]:
    return await session.execute(select(models.Book))
```

Прежде всего обратим внимание на то, что мы указываем новый параметр в декоратор — response_model. Тут мы указываем тип, к которому нужно сериализовать данные, что возвращает эндпоинт. В первом случае указываем наш класс BookOut, где есть ID. А во втором случае List of BookOut. FastAPI поймёт, что ему нужно обернуть объекты в список. И заметим: то, что возвращает сама функция-обработчик, не то же самое, что вернёт path-декоратор. Можно сказать, что аннотация функции — только для разработчиков, и FastAPI на это никак не смотрит.

Ну а дальше всё достаточно просто. В зависимости от того, что нам нужно сделать, нужна ли нам транзакционность, вызываем нужные методы у объекта сессии. И тут заметим, что если нужно нужно выполнить агрегацию, то придётся пользоваться так называемым 2.0 стилем агрегации. Вместо того чтобы вызывать метод query у объекта сессии, мы пользуемся объектом, инкапсулирующим в себе логику селекта. Подробнее про его работу и отличия смотрите здесь и тут. Весь этот код работает асинхронно. Всё, что нам осталось, — вернуть объекты, причём вся сериализация из ORM в json будет работать автоматически.

Посмотрим, как оно работает.

И ещё один момент: нам доступна документация в двух форматах. С одним мы уже поработали, но есть второй: redoc. По смыслу они идентичны, но немного различаются визуально. Изучите на досуге.



Итак, в этом модуле мы научились работать с библиотекой FastAPI. Мы знаем, как написать базовую структуру проекта с основными эндпоинтами, умеем делать валидацию параметров: URL, query, и body. Научились работать с БД асинхронно, используя SQLAlchemy. Впереди немного домашней работы, до встречи!