Методичка

Данная методичка по лабораторной работе дополняет информацию из лекции, но ни в коем случае её не заменяет. > Если чего-то не найдёте в этом файле, ищите информацию в лекции, обращайтесь к GPT(но с умом!!)) или пишите мне в телегу @MeInOS.

Чтобы начать собирать приложение, воспользуемся пакетным менеджером. Он управляет версиями пакетов и создаёт виртуальное окружение, в котором мы будем работать (подробнее что такое виртуальное окружение, смотрите тут). Мы будем использовать пакетный менеджер `uv`.

Создаём директорию, переходим в неё и инициализируем `uv`
`uv init`

После выполнения команды в директории появятся пять файлов. Разберёмся с каждым подробнее.

- `.gitignore` — файл используется для игнорирования ненужных файлов в Git, которые не потребуются в вашем удалённом репозитории. Если не планируете использовать `Git`, этот файл можно удалить.
- `.python-version` — файл, в котором `uv` указывает используемую версию Python по умолчанию.. Тем не менее, если этого файла не будет, `uv` сам определит версию Python (подробнее)
- `hello.py` — файл-пример, содержащий одну функцию для вывода приветствия. Он не нужен для работы приложения, поэтому его можно удалить.
- `README.md` — файл для описания проекта, можно удалить при желании
- `pyproject.toml` — конфигурационный файл для управления настройками и зависимостями проекта (подробнее)

Создаём виртуальное окружение
`uv venv`

Активируем виртуальное окружение
`source .venv/bin/activate` — для MacOS/Linux
`.venv\Scripts\activate` — для Windows

Теперь будем работать внутри него, и сюда же добавлять зависимости
Добавим всё, что нам нужно для работы
`uv add fastapi sqlalchemy asyncpg aiosqlite pydantic-settings alembic greenlet`

Следующий шаг — инициализировать `alembic` для миграций.
`alembic init -t async alembic` — применяем команду для инициализации алембика и работы с асинхронным кодом. Вы увидите в директории файл `alembic.ini` и папку `alembic`, в которой содержатся другие конфигурационные файлы и папка `versions` — где будут находиться ваши миграции и вы всегда сможете к ним обратиться.

Создайте папку app и в ней файл `models.py` и опишите базовую модель алхимии, вот так:

```
from sqlalchemy.orm import DeclarativeBase
from sqlalchemy.ext.asyncio import AsyncAttrs


class Base(DeclarativeBase, AsyncAttrs):
    pass
```

Создайте в корне файл `settings.py` и объявим там простую конфигурацию, где будет строчка с подключением к базе данных

```
from pydantic_settings import BaseSettings, SettingsConfigDict


class Settings(BaseSettings):
    DATABASE_URI: str = 'sqlite+aiosqlite:///:memory:"'

    model_config = SettingsConfigDict(env_file='.env')

settings = Settings()
```

Затем перейдите в `alembic/env.py` и добавьте эти строчки

```
from settings import settings
from app.models import Base
```

И эти строчки чуть ниже

```
target_metadata = Base.metadata
config.set_main_option("sqlalchemy.url", settings.DATABASE_URI)
```

Начало файла должно выглядеть так:

```python
import asyncio
from logging.config import fileConfig

from sqlalchemy import pool
from sqlalchemy.engine import Connection
from sqlalchemy.ext.asyncio import async_engine_from_config

from settings import settings
from app.models import Base

from alembic import context

# this is the Alembic Config object, which provides
# access to the values within the .ini file in use.
config = context.config

# Interpret the config file for Python logging.
# This line sets up loggers basically.
if config.config_file_name is not None:
    fileConfig(config.config_file_name)

# add your model's MetaData object here
# for 'autogenerate' support
# from myapp import mymodel
# target_metadata = mymodel.Base.metadata
target_metadata = Base.metadata
config.set_main_option("sqlalchemy.url", settings.DATABASE_URI)
```

Добавим код для соединения с БД. Не буду сильно углубляться в его пояснения, иначе можно застрять в синтаксисе питона, лишь скажу, чтомы создаём "движок" алхимии для работы с БД, а также фабрику асинхронных сессий, которая с помощью кода, описанного в `get_session()` будет возвращаться сессию. Для того чтобы получить сессию в эндпоинте, нужно вызвать эту функцию в эндпоинте, через `Depends` — `db_session: AsyncSession = Depends(get_session)` (смотреть пример в лекции)

```python
from sqlalchemy.ext.asyncio import AsyncSession, create_async_engine,
async_sessionmaker
from sqlalchemy.exc import SQLAlchemyError
from settings import settings


engine = create_async_engine(settings.DATABASE_URI, echo=False)

async_session = async_sessionmaker(bind=engine, autocommit=False, autoflush=False,
class_=AsyncSession)


async def get_session():
    async with async_session() as session:
        try:
            yield session
        except SQLAlchemyError:
```

```
            await session.rollback()
            raise
        finally:
            await session.close()
```

> Внимание: мы установили Б8 **in-memory** БД, которая хранится в оперативной памяти. Для ЛР рекомендуется создать постоянную базу данных (смотреть README файл в проекте к лекции). Если же вы не установите постоянную базу данных, добавьте следующие строки в том месте, где у вас инициализировано `FastAPI` приложение:

```
@app.on_event('startup')
async def startup():
    if settings.DATABASE_URI == 'sqlite+aiosqlite:///:memory:"':
        async with engine.begin() as conn:
            await conn.run_sync(models.Base.metadata.create_all)
```

Подготовка закончена! Теперь перейдём к заданию.

Задание

Вы разрабатываете backend-сервис для кафе, в котором необходимо хранить информацию о клиентах и их заказах. Один клиент может сделать **много заказов**, но **каждый заказ принадлежит только одному клиенту** (связь “один ко многим”). Вам нужно сделать две модели SQLAlchemy: Client (Клиенты) и Order (Заказы).

Модели

Поля для `Client`:

- `id`
- `name`
- `phone`
- `created_at`

Поля для `Order`:

- `id`
- `total_price`
- `status` (не заморачивайтесь с `enum`, оформите как просто строчку, по типу "pending", "completed" и тд, на ваше усмотрение)
- `created_at`
- `client_id` (FK к Client)

Также подумайте об ORM-связи между ними (через `relationship`).

Реализовать REST API с методами:

3.1. Для клиентов (Clients):

• POST /clients — создать клиента
• GET /clients/{client_id} — получить клиента по ID
• PUT /clients/{client_id} — обновить данные клиента
• DELETE /clients/{client_id} — удалить клиента (заказы для этого клиента тоже должны удаляться)
• GET /clients — получить список всех клиентов

3.2. Для заказов (Orders):

• POST /orders — создать заказ для клиента (client_id)
• GET /orders/{order_id} — получить заказ по ID
• PUT /orders/{order_id} — обновить заказ (изменить статус или сумму)
• DELETE /orders/{order_id} — удалить заказ
• GET /orders — получить список всех заказов
• GET /clients/{client_id}/orders — получить все заказы конкретного клиента

Если у вас всё получилось — отлично. Если вы хотите узнать правильно ли вы написали приложение, тонкости архитектуры или спросить если что-то непонятно, прошу ко мне в личку — @MelnOS