Введение в создание API. Настройка директории

1. Основные библиотеки и зависимости

В этом проекте мы будем использовать множество библиотек, чтобы реализовать полноценное асинхронное API. Давайте рассмотрим ключевые зависимости, которые помогут нам в процессе разработки.

1. FastAPI (`fastapi==0.115.0`)

FastAPI — это современный, высокопроизводительный фреймворк для создания API на Python с поддержкой асинхронности. Он позволяет писать декларативные маршруты с минимальными затратами времени на разработку и обеспечивает автоматическую документацию через Swagger и Redoc.

2. SQLAlchemy (`SQLAlchemy==2.0.35`)

SQLAlchemy — мощный ORM (Object Relational Mapping), который позволяет работать с базами данных как с объектами Python. В нашем проекте используется асинхронная версия SQLAlchemy в сочетании с `asyncpg` для работы с PostgreSQL.

3. Alembic (`alembic==1.13.3`)

Alembic — это инструмент для управления миграциями базы данных. Он тесно интегрирован с SQLAlchemy и позволяет легко обновлять схему базы данных по мере изменения модели данных.

4. Asyncpg (`asyncpg==0.29.0`)

Asyncpg — это высокопроизводительный драйвер для работы с PostgreSQL, который поддерживает асинхронные операции. Это позволяет значительно повысить производительность нашего API при работе с базой данных.

5. Bcrypt (`bcrypt==4.2.0`)

Bcrypt — это библиотека для безопасного хеширования паролей. Она используется для защиты данных пользователей путем безопасного хранения их паролей в базе данных.

6. Uvicorn (`uvicorn==0.31.0`)

Uvicorn — это асинхронный HTTP-сервер, который работает на основе ASGI (Asynchronous Server Gateway Interface). Он обеспечивает поддержку асинхронного выполнения задач и необходим для работы FastAPI.

7. Pydantic (`pydantic==2.9.2`)

Pydantic — это библиотека для работы с валидацией данных. Она позволяет легко проверять данные, поступающие в наш API, и определять строгие типы для маршрутов FastAPI.

8. Alembic и миграции

Alembic необходим для управления изменениями структуры базы данных. С помощью этого инструмента мы сможем управлять версиями нашей базы данных и легко вносить изменения в схему.

Другие важные зависимости

- **PyJWT** (`PyJWT==2.9.0`): библиотека для работы с токенами JSON Web Tokens (JWT), которая поможет нам в аутентификации и авторизации.
- **Python-dotenv** (`python-dotenv==1.0.1`): помогает загружать переменные окружения из файла `.env` .

Эти библиотеки составляют основу нашего проекта, но мы также используем другие пакеты для тестирования, управления конфигурациями и форматирования кода.

2. Создание виртуального окружения и установке необходимых пакетов

`venv` (виртуальное окружение) — это инструмент Python, который создаёт изолированную среду для проектов. В каждом проекте можно установить свои зависимости и версии библиотек,

которые не будут влиять на другие проекты на вашем компьютере. Это полезно для управления версиями библиотек и поддержания совместимости.

```
python -m venv .venv
```

> **Tip**
>
> Чтобы создать `venv` с определенной версией Python, например 3.11 python нужно написать:
>
> ```
> python -3.11 -m venv .venv
> ```

Дальше необходимо его активировать

```
.venv/Scripts/Activate
```

> **Warning**
>
> Если при активации `venv` выдает ошибку, возможно у терминала не достаточно прав для активации с его помощью скриптов, используйте в этом случае следующую команду
>
> ```
> Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
> ```

После активации установим все нужные нам зависимости при помощи файла `requirements.txt`

`requirements.txt` — это файл, который содержит список зависимостей (библиотек) проекта с указанием их версий. Этот файл используется для воспроизведения окружения проекта на другом компьютере с помощью команды:

```
pip install -r requirements.txt
```

> **Tip**

> Для создания файла можно использовать следующую комамнду:
>
> ```
> pip freeze > requirements.txt
> ```

3. Alembic: Управление миграциями базы данных

Что такое Alembic?

Alembic — это инструмент для управления миграциями базы данных. Миграции позволяют разработчикам контролировать и отслеживать изменения в структуре базы данных по мере развития приложения. Alembic тесно интегрирован с SQLAlchemy и используется для создания, обновления и отката схемы базы данных без потери данных.

Почему важны миграции?

Когда вы работаете с базой данных, структура таблиц и связей между ними со временем меняется. Например, добавляются новые поля, удаляются ненужные, обновляются типы данных. Чтобы эти изменения применялись к базе данных плавно, необходимо управлять ими с помощью миграций. Миграции обеспечивают:

- Легкость обновления базы данных при изменении моделей.
- Возможность отката изменений, если что-то пошло не так.
- Централизованное хранение истории всех изменений схемы.

Настройка Alembic

1. Инициализация

```
alembic init migrations
```

Эта команда в директории проекта создаст следующую структуру:

```
project-root/
├── alembic.ini
└── migrations/
    ├── env.py
    ├── versions/
        └── (migration files will appear here)
    ├── README
    └── script.py.mako
```

- **alembic.ini** — основной конфигурационный файл Alembic.
- **migrations/** — директория для файлов Alembic.
  - **env.py** — файл, отвечающий за настройку окружения миграций.
  - **script.py.mako** — шаблон для создания новых миграций.
  - **versions/** — папка, в которой будут храниться файлы миграций.

2. alembic.ini

Файл `alembic.ini` управляет конфигурацией Alembic. Здесь вы можете указать `script_location` - папку, которую использует alembic, строку подключения к базе данных и другие параметры. Найдите строку, которая начинается с `sqlalchemy.url` . В ней вы можете указать URL для подключения к базе данных, в нашем случае мы будем использовать PostgreSQL с `asyncpg` :

```
sqlalchemy.url = postgresql+asyncpg://username:@localhost:5432/dbname?async_fallback=True
```

- **postgresql** - используем PostgreSQL
- **asyncpg** - используем асинхронный движок для PostgreSQL
  - **username** - имя вашего пользователя в БД
  - **password** - пароль от вашего пользователя в БД
  - **localhost** - хост для БД
  - **5432** - порт для хоста
  - **dbname** - название базы данных
    - **async_fallback** - опция отката к использованию синхронных операций, в случае отсутствия асинхронных

Но мы с вами не будем так делать, а настроим конфигурационный файл чуть дальше, а пока укажем следующий код:

```
sqlalchemy.url = %(DATABASE_URL)s?async_fallback=True
```

> **Tip**
>
> Также вы можете раскоментить следующую строку:
>
> ```
> file_template = %%(year)d_%%(month).2d_%%(day).2d_%%(hour).2d%%(minute).2d-%%(rev)s_%%(slu
> ```
>
> И тогда формат названий файлов миграций изменится и будет содежрать точную дату ее создания

3. migrations/env.py

Файл отвечает за настройку окружения миграций. Здесь необходимо указать наши модели и подключение к базе данных, и в общем будет иметь слелующий вид:

```python
#Импорты
from logging.config import fileConfig
from sqlalchemy import engine_from_config
from sqlalchemy import pool
from alembic import context

from config import DATABASE_URL #Это будет импортировано из config.py

#Здесь мы импортируем наш class,
#наследованный от DeclarativeBase(Базовый класс для всех моделей в sqlalchemy)
from api.reservation.models import Base #Закоментируйте, чтоб не светилась ошибка

config = context.config
section = config.config_ini_section

#Если вы вспомните, то в alembic.ini мы указывали:
#sqlalchemy.url = %(DATABASE_URL)s?async_fallback=True
#Здесь мы передаем в DATABASE_URL значение config.DATABASE_URL,
#импортированное выше
config.set_section_option(section, "DATABASE_URL", DATABASE_URL)

if config.config_file_name is not None:
    fileConfig(config.config_file_name)

#Здесь мы передаем metadeta из нашего класс Base
#metadata содержит схемы табличек, унаследованных от этого класса
target_metadata = Base.metadata #Закоментируйте, чтоб не светилась ошибка

def run_migrations_offline() -> None:
    #Базовый функционал


def run_migrations_online() -> None:
    #Базовый функционал

#Выбор режима, оставляем также без изменений
if context.is_offline_mode():
    run_migrations_offline()
else:
    run_migrations_online()
```

> **Warning**
>
> В структуре нашего проекта модели будут находиться не в 1 файле, после того как в самом первом мы создадим класс `Base`, унаследованный от `DeclerativeBase` мы будем передавать его в остальные, и уже от него наследовать последующие модели. А в файл `env.py` мы импортируем `Base` из самого последнего файла с моделями в этой цепочке, иначе вы не все модели будут учитываться и в БД будет не хватать таблиц

2. Настройка Config

В директории проекта создадим файл `.env`, этот файл будет содержать переменные окружения, хранящиеся в виртуальном окружении. В нашем проекте он будет иметь следующую структуру:

```
DB_USER="PolinaScrbbs"
DB_PASSWORD="Que337"
DB_NAME="Cinema"

DATABASE_URL="postgresql+asyncpg://PolinaScrbbs:Que337@localhost:5432/Cinema"

# Этот ключ мы будем использовать для хеширования пароля.
# Вы можете задать любую последоватьльность символов.
SECRET_KEY="c87d095c0d020fba6d5671484664c1825b9fe9c69ef59f324f95b96a2fa7ae0b"
```

Далее создадим файл `config.py`, содержащий следующую структуру:

```python
# dotenv - библиотека для получения данных из виртуального окружения
from dotenv import load_dotenv
import os

#Для корректного обновления переменных
# при изменении переменных мы их сначала очистим
os.environ.pop("DB_USER", None)
os.environ.pop("DB_PASSWORD", None)
os.environ.pop("DB_NAME", None)

os.environ.pop("DATABASE_URL", None)
os.environ.pop("SECRET_KEY", None)

#Загружаем данные
load_dotenv()

#Создаём перменные для дальнейшего использования
DB_USER = os.getenv("DB_USER")
DB_PASS = os.getenv("DB_PASSWORD")
DB_NAME = os.getenv("DB_NAME")

DATABASE_URL = os.getenv("DATABASE_URL")
SECRET_KEY = os.getenv("SECRET_KEY")
```

В итоге дирректория проекта должна преобрести следующий вид:

```
project-root/
├── migrations/
    ├── env.py
    ├── versions/
        └── (migration files will appear here)
    ├── README
    └── script.py.mako
├── .env
├── alembic.ini
└── config.py
```