Создание первой миграции

1. Настройка подключения к БД

Для создания миграции нам понадобятся модели, которые будут описывать таблицы в базе данных. Создадим в базовой директории папку `api`, в ней мы уже будет создавать все, связанное с нашим API.
Дальше создайте файл `database.py`, в нем мы создадим подключение к нашей базе данных и будем ипортировать его при необходимости.

```
#Импорты
from typing import AsyncGenerator
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.orm import sessionmaker

#Импортируем DataBase URL из config.py
from config import DATABASE_URL

#Остальной код
```

Дальше нам понадобится создать объект аснхронного движка подключения к БД - `engine`, для этого добавьте следующий код:

```
engine = create_async_engine(DATABASE_URL, echo=True)
```

- **DATABASE_URL** - URL для подключения к БД
- **echo** - опция, которая включает логирование всех SQL-запросов, которые выполняются через этот движок. Это полезно для отладки и мониторинга запросов.

После создания `engine` нам понадобится так называемая "фабрика" для создания асинхронных сессий - `async_session`, для её создания добавим следующий код:

```
async_session = sessionmaker(
    bind=engine,
    class_=AsyncSession,
    expire_on_commit=False,
)
```

- **bind=engine** - указывает, что сессии будут использовать ранее созданный движок `engine` для взаимодействия с базой данных.
- **class_=AsyncSession** - указывает, что сессия будет асинхронной (использует класс `AsyncSession`)
- **expire_on_commit=False** - отключает автоматическое "истечение" объектов после фиксации транзакции (После сохранения нового объекта в БД или сохранениц изменений уже существующего). Это нужно, чтобы избежать повторных обращений к базе данных при повторном использовании объектов после `commit()`

Теперь мы можем перейти к созданию асинхронной функции `get_session` - это функция, которая возвращает генератор сессий. Это общепринятый способ предоставления сессий в асинхронных приложениях, таких как FastAPI, где каждый запрос может иметь свою сессию для взаимодействия с базой данных. Давайте добавим эту функцию:

```
async def get_session() -> AsyncGenerator[AsyncSession, None]:
    async with async_session() as session:
        yield session
```

- `-> AsyncGenerator[AsyncSession, None]` - указывает, что будет возвращать эта функция. В этом проекте мы везде будем использовать явную типизацию.
- `async with async_session() as session` - здесь создается асинхронная сессия. В этом контексте сессия автоматически закрывается после завершения работы с ней (даже в случае ошибок)
- `yield session` - позволяет временно передать управление сессией коду, который вызывает `get_session` . Это сделано для того, чтобы сессия могла быть использована внутри блоков `async for` или в других частях программы. После завершения работы сессией управление возвращается обратно в `get_session` , и сессия закрывается.

2. Создание первой модели

Нацонец мы закончили с настройкой подключения к БД. Можно перейти к созданию первой модели. Для этого в папке `api` создадим папку `user` , а в ней файл `models.py`

> **Warning**
>
> Мы и в дальнейшем будем предерживаться одной структуры. У нас будут добавляться папки для отдельных "модулей", например захотите вы группы, создаем `group` или `auth` и тд. и внутри него соответствующие модулю файлы.

2.1. Добавление нужных пакетов

В файле `user/models.py` добавим следующий код:

```
from datetime import datetime, timedelta, timezone
# datetime: для работы с текущим временем (создание временных меток).
# timedelta: для работы с интервалами времени (например, срок действия токенов).
# timezone: для работы с часовыми поясами.

from typing import Optional
# Optional: используется для указания того, что переменная может быть либо определенного типа, л

from fastapi import HTTPException, status
# HTTPException: для возврата HTTP-ошибок в FastAPI (например, 401 Unauthorized).
# status: содержит константы HTTP-статусов для удобства использования.

import jwt
# jwt: библиотека для работы с JSON Web Tokens (создание, декодирование и проверка токенов).

import bcrypt
# bcrypt: библиотека для хэширования паролей и их проверки.

from sqlalchemy import Column, Integer, String, ForeignKey, Enum
# Column: определяет столбцы в модели базы данных.
# Integer: тип данных для целочисленных столбцов.
# String: тип данных для строковых столбцов.
# ForeignKey: используется для создания внешнего ключа (связь между таблицами).
# Enum: для работы с перечислениями в базе данных.

from sqlalchemy.orm import DeclarativeBase, relationship
# DeclarativeBase: базовый класс для всех моделей базы данных, которые будут отображаться на таб
# relationship: используется для определения связей между моделями (например, "один ко многим" и

from sqlalchemy.ext.asyncio import AsyncSession
# AsyncSession: асинхронная сессия для выполнения операций с базой данных без блокировки.

from enum import Enum as BaseEnum
# BaseEnum: базовый класс для создания собственных перечислений (enum) в Python.

from config import SECRET_KEY
# SECRET_KEY: секретный ключ, который используется для подписывания JWT-токенов или других крипт
```

Это все нужные нам импорты, комментарии оставляйте на ваш вкус, у меня их дальше у не будет.

2.2. Объявление базовых классов

Сейчас мы объявим базовые классы для дальнейшего наследования от них

```
class Base(DeclarativeBase):
    pass

class BaseEnum(BaseEnum):
    @classmethod
    async def get_values(cls):
        return [breed.value for breed in cls]
```

- `Base` - базовый класс модели, помните в `migrations/env.py` мы импортируем самый последний в цепочке.
- `BaseEnum` - базовый класс для создания классов перечислений, в нем я определил метод `get_values` для получение списка его `value`

2.3. Создание первого Enum

Создадим класс Role, в котором перечислим роли, использующиеся в нашем проекте.

```
class Role(BaseEnum):
    USER = "Пользователь"
    CASHIER = "Кассир"
    ADMIN = "Администратор"
```

> **Tip**
>
> - `Role.USER` вернёт объект USER
> - `Role.USER.value` вернёт Пользователь

2.4. Создание первой модели и таблицы

Теперь создадим модель `User` :

```
class User(Base):
    __tablename__ = "users"

    #primary_key - первичный ключ
    id = Column(Integer, primary_key=True)
    #unique - уникальность, nullable - может ли быть Null
    username = Column(String(20), unique=True, nullable=False)
    hashed_password = Column(String(512), nullable=False)
    #default - дефолтное значение, например если при посздании
    #явное не указана роль, то создасться USER
    role = Column(Enum(Role), default=Role.USER, nullable=False)
    full_name = Column(String(40), nullable=False)
```

- `__tablename__` - имя таблички в БД
- `username` - имя пользователя, в дальнейшем логин для авторизации
- `role` - роль пользователя, мы используем созданный ранее Enum, дефолтно указывая на `USER`
- `full_name` - ФИО пользователя

> **Tip**
>
> Почему именно `hashed_password` ? Мы с вами в БД будем хранить захешированный пароль, это будет некая последовательность символов. При создании авторизации мы определим функции хеширования пароля и работы с ним. А для получения мы создадим метод `password` , который будет его декешировать, по этому поле в модели так называется

2.5. Создание первой миграции

Теперь в файле `migrations/env.py` импортируем наш `Base` , напомню:

```
from api.user.models import Base
```

> **Warning**
>
> И раскомментируйте, если комментировали `target_metadata = Base.metadata`

Теперь нам осталось создать и применить миграции при помощи `alembic` , давайте посзнакомимся с основными командами:

```
alembic revision --autogenerate -m "Create users table"
```

Создает файл миграции в директории versions. Флаг --autogenerate анализирует изменения в моделях SQLAlchemy и автоматически генерирует SQL-запросы для изменений, таких как добавление/удаление столбцов или таблиц. Параметр -m "" добавляет описание для этой миграции.

```
alembic upgrade head или alembic upgrade <revision>
```

Применяет все миграции от текущей ревизии до указанной. head — это последняя ревизия, или можно указать конкретный ID ревизии. Полезно для развертывания новых версий схемы базы данных.

```
alembic downgrade base или alembic downgrade <revision>
```

Откатывает изменения в базе данных до указанной ревизии. base — это самая начальная ревизия (схема будет откатана до пустой базы данных), а — это конкретный ID ревизии.

```
alembic current
```

Показывает идентификатор текущей ревизии, до которой обновлена база данных.

Теперь мы знаем основные команды. Создадим и применим миграцию. Если вы все делали правильно, то в папке `migration/version` появится миграция, а ваша модель `User` в виде таблицы появится в БД

```
alembic revision --autogenerate -m "Initial Migrate" && alembic upgrade head
```