Министерство образования Новосибирской области ГБПОУ НСО
«Новосибирский авиационный технический колледж имени Б. С. Галущака»

Методические указания к разработке API на Python с использованием библиотек FastAPI и SQLAlchemy
Учебная дисциплина: МДК.01.04 Системное программирование

Разработали:

Студенты группы ПР-23.106

Кретинин Дмитрий Александрович

Мешков Александр Андреевич

2024

Содержание

1 Общее описание веб фреймоврка FastAPI и библиотеки SQLAlchemy

FastAPI – это веб – фреймворк, написанный для языка Python. Он используется для быстрого и эффективного создания веб-приложений. Его преимущество заключается в высокой производительности и простоте использования, также он умеет автоматически генерировать Swagger(о нём чуть позже), что упрощает разработку и тестирование API.

SQLAlchemy – это библиотека для Python, созданная для синхронизации объектов Python с записями различных реляционных баз данных. Она состоит из двух компонентов:

— SQLAlchemy Core – компонент для взаимодействия библиотеки с SQL-базами данных.

— SQLAlchemy ORM – интерфейс для управления базами данных через модели и объекты

Также мы будем использовать вспомогательные библиотеки:

— Uvicorn – это ASGI – сервер, он нам нужен для запуска нашего FastAPI приложения.

— Psycopg2 – это драйвер для подключения к PostgreSQL (учтите, что данная библиотека не поддерживает версию Python выше 3.10), если же вы используете UNIX системы (Mac, Linux), то устанавливайте psycopg2-binary

— Alembic – инструмент для миграции базы данных, используемый в SQLAlchemy

— Pydantic-settings – упрощает управление настройками приложения FastAPI.

— Poetry – это аналог pip, для управления зависимостями в проектах

— Asyncio – библиотека для использования асинхронных методов

2 Установка библиотек

Для того, чтобы установить все вышеперечисленные библиотеки, установим инструмент управления «poetry».

Для его установки открываем любой удобный вам терминал (cmd, powershell, можно и сразу в среде разработки) и пишем:

— pip install poetry (или pip3 install poetry (для UNIX))

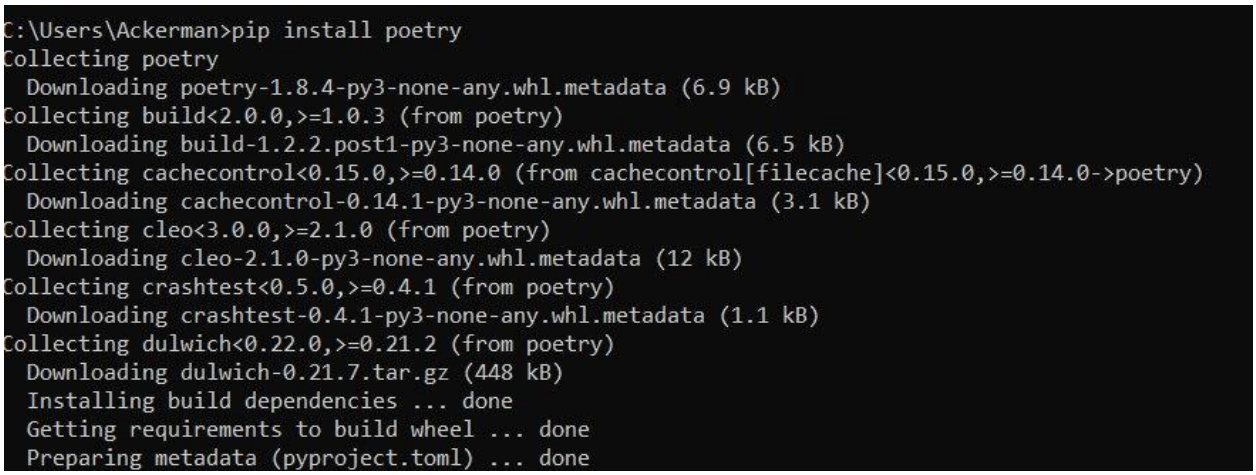

Рисунок 1 – Установка Poetry

После ввода вышеуказанной команды, вы получите подобное сообщение об установке инструмента на ваш ПК. Далее нужно будет с его помощью добавить зависимости в проект.

Для удобства посоветую открыть терминал в вашей среде разработки, в заранее созданной директории проекта.

Открываем терминал (в данном примере всё рассматривается в среде разработки Visual Studio Code, в ней это можно сделать путём нажатия сочетания клавиш CTRL + J).

Далее прописываем в терминал команду

— poetry init

И прожимаем клавишу Enter, пока не увидите свою директорию.

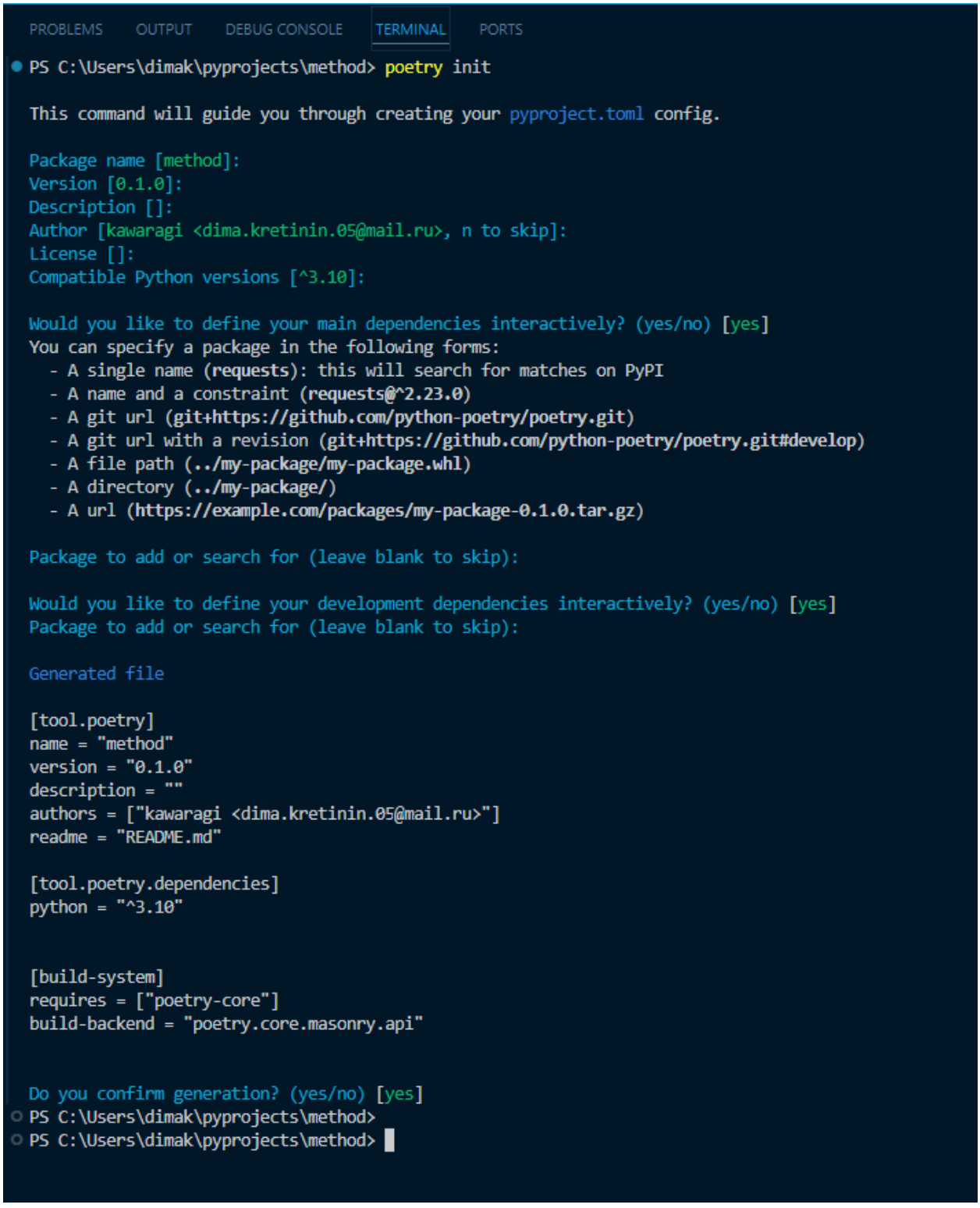

Рисунок 2 – Инициализация poetry в директории проекта

После инициализации нам нужно добавить зависимости в наш проект, для этого прописываем команду в терминал:

— poetry add sqlalchemy uvicorn alembic pydantic-settings psycopg2 asyncio fastapi

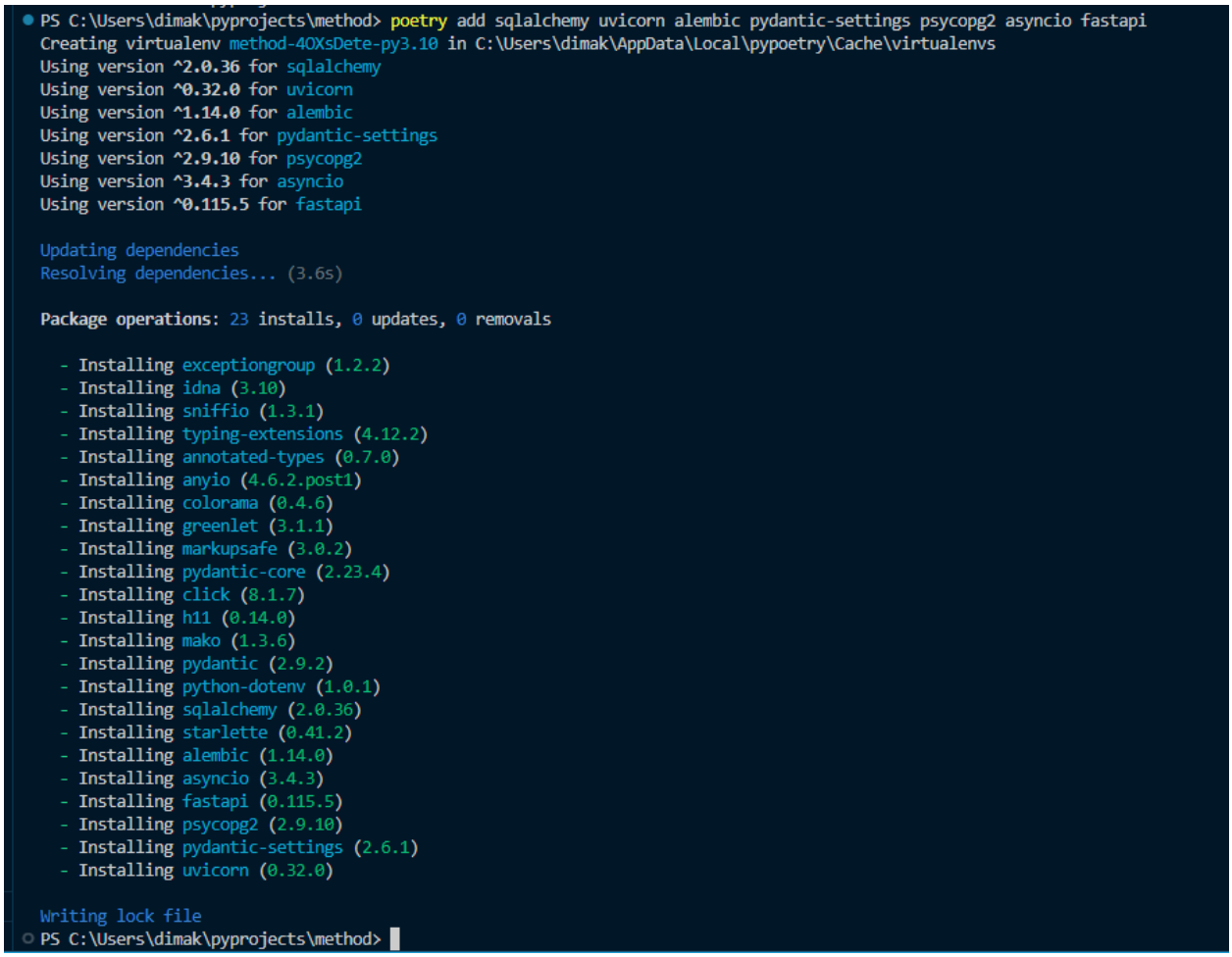

Рисунок 3 – Добавление зависимостей в проект

Зависимости добавлены, осталось только выбрать нужный интерпретатор. Для этого нам нужно будет прописать в терминал 2 команды:

— poetry shell

— poetry env info

После выполнения этих двух команд копируем из терминала строку«Executable» из блока Virtualenv (как показано на рисунке 4)

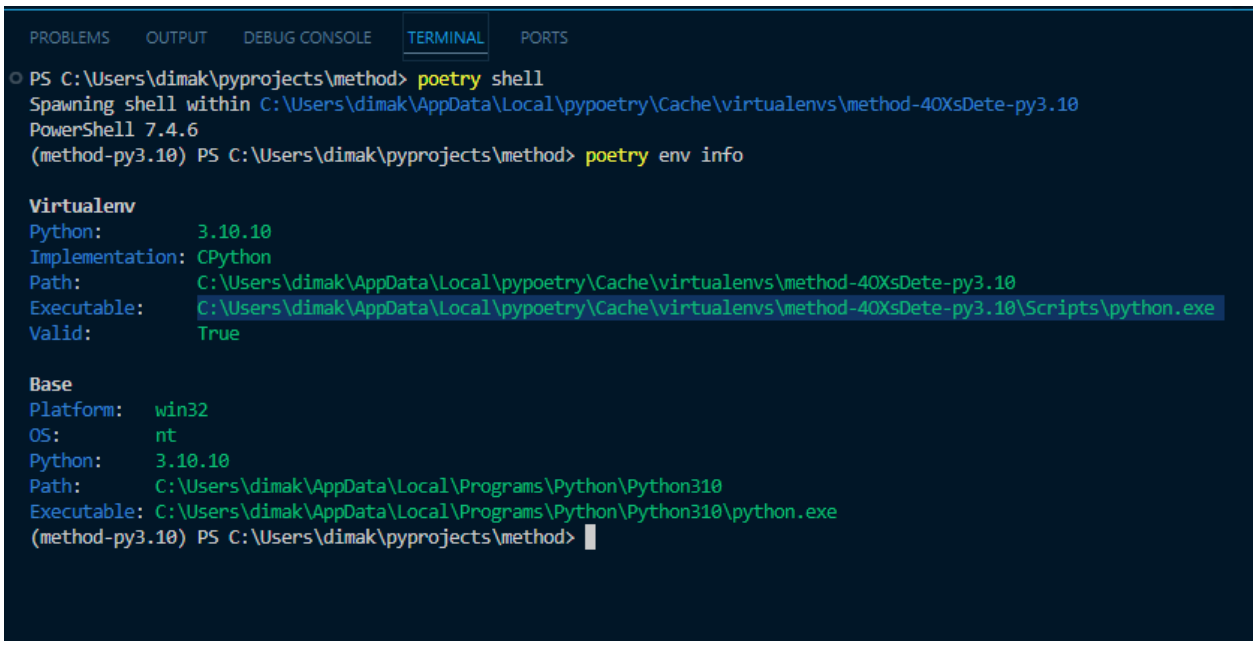

Рисунок 4 – Выбор директории интерпретатора

Копируем этот путь. Создаём файл «main.py» и открываем его.

Обычно Visual Studio Code сама выбирает новый интерпретатор, но может быть такое, что она его не обнаружит.

Для выбора интерпретатора нажимаем на кнопку, показанную на рисунке 5.

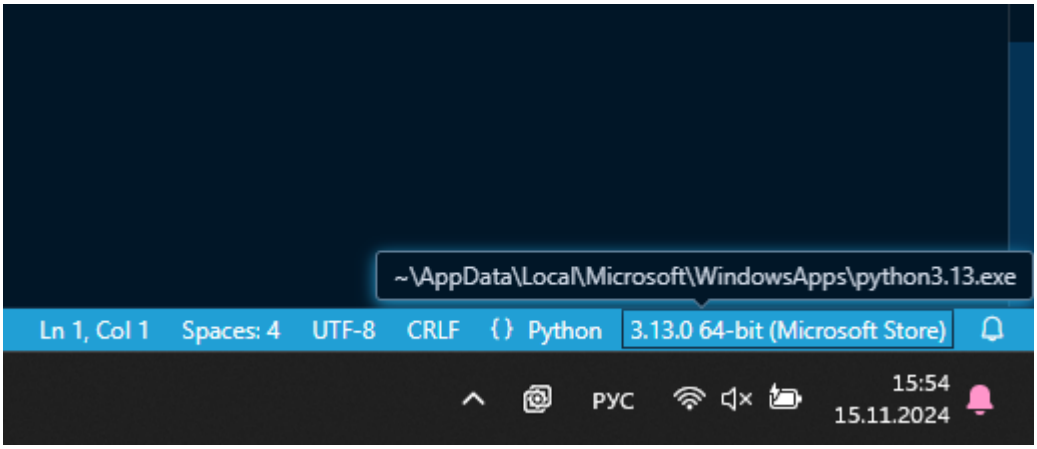

Рисунок 5 – кнопка выбора интерпретатора

Сверху появится следующее окно:

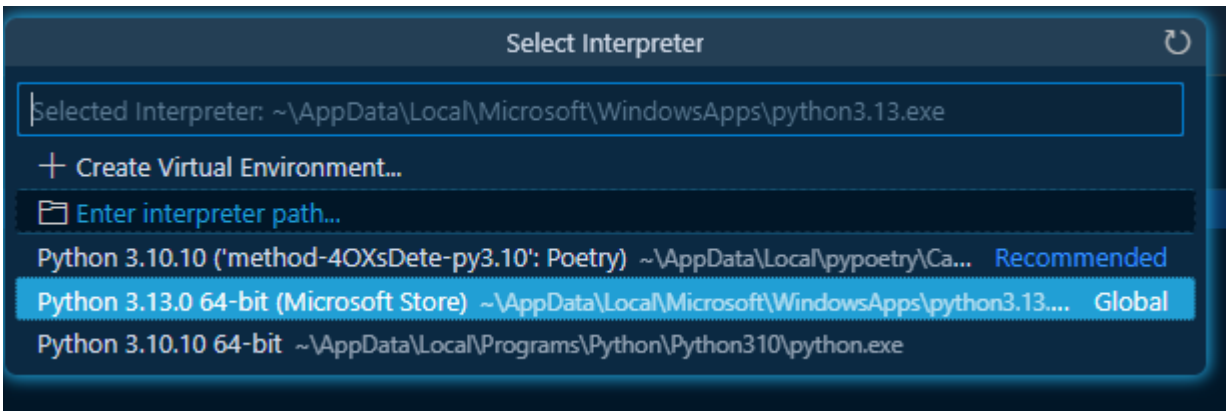

Рисунок 6 – добавление нового интерпретатора

Далее нажимаем на самую верхнюю кнопку и вставляем туда ранее скопированную директорию.

После выполнения всех вышеуказанных действий всё готово к работе.

3 Необходимое ПО

Для работы с Python, FastAPI и SQLAlchemy необходимо следующее ПО:

— IDE: PyCharm или Visual Studio Code

— Postman: для проверки запроса программно

— Любой бразуер: для проверки запроса в браузере

4 Что такое ORM?

В данной практической работе, мы будем связывать наше FastAPI приложение и базу данных.

Базу данных мы будем создавать при помощи ORM. ORM(Object Relational Mapping, объектно-реляционное отображение) –это представление объектов базы данных в виде классов. В данном случае мы представим таблицы нашей базы в виде классов Python.

А чтобы загрузить нашу базу, нам нужно будет использовать так называемые «миграции». Это что-то вроде системы контроля версий для вашей схемы БД. Реализовывать это мы будем при помощи инструмента Alembic.

5 Определение моделей

Допустим, что нам нужно учитывать всех пользователей в системе, у каждого из которого есть определённая роль. Пускай в нашей системе будет всего 3 роли: пользователь, админ и помощник.

Создаём файл «models.py», в котором мы будем инициализировать наши модели.

В данном файле нам пригодятся такие методы, как: declarative_base(), Mapped[], mapped_column(), relationship из модуля sqlalchemy.orm, а также нам пригодится всё, что есть в sqlalchemy

```
models.py > ...
1   from sqlalchemy.orm import declarative_base, Mapped, mapped_column, relationship
2   from sqlalchemy import *
3
```

Рисунок 7 – Импорт модулей в проект

Для определения нашей базы создадим переменную Base, в которую передадим метод declarative_base(). Данный метод предоставляет всем моделям базовый класс, который упрощает работу с базой данных.

Далее создаём два класса User и Role, которые наследуются от нашей переменной Base.

```
8   Base = declarative_base()
9
10  class User(Base):
11      __tablename__ = "users"
12      id:Mapped[int] = mapped_column(Integer, primary_key=True)
13      name:Mapped[str] = mapped_column()
14      role_id:Mapped[int] = mapped_column(Integer, ForeignKey("roles.id"), nullable=True)
15      roles = relationship("Role", uselist=False, back_populates="user")
16
17  class Role(Base):
18      __tablename__= "roles"
19      id:Mapped[int] = mapped_column(Integer, primary_key=True)
20      name:Mapped[str] = mapped_column()
21      user = relationship("User", back_populates="roles")
22
```

Рисунок 8 – Определение таблиц

В каждом классе указываем имя таблицы и после этого определяем столбцы, как поля класса.

Наследуем каждое поле класса от «Mapped[нужный тип данных]». Присваиваем этому полю метод «mapped_column()», так делаем с каждым столбцом. (Если же вы определяете числовое поле, то в метод «mapped_column()» передайте туда конкретный числовой тип данных, так как в postgres есть много числовых типов, в нашем примере возьмём простой Integer.)

Также в ключевые поля передаём два аргумента: «primary_key=True».

Теперь определим связь между таблицами. У нашего пользователя есть роль, которая будет браться из таблицы ролей. Значит, нам нужно определить между ними связь.

У поля «role_id» определяем внешний ключ, передав как аргумент «ForeignKey()», в который передаём поле id класса role, то есть «role.id».

Также прописываем объекты ролей и пользователя для каждой таблицы, путём добавления одноимённых полей в каждый класс. В них мы передадим метод «relationship()», в котором укажем класс, из которого будем брать данные и укажем объект, в который вернём ранее указанное поле класса. Проще говоря: «user = relationship(“User”, back_populates =«roles»»

«User» - это таблица, с которой будет связана таблица, а «roles» это поле, в которое будут возвращаться данные. Главное указать это в обоих таблицах, иначе миграции не получатся.

6 Подключение к базе данных

Создаём файл «config.py». В нём у нас будет прописана строка подключения к нашей базе. Для начала импортируем всё из sqlalchemy, и метод Session() из sqlalchemy.orm

В нём опишем класс подключения, назовём его «DBSettings».

```
class DBSettings():
    @staticmethod
    def get_session():
        engine = create_engine(f"postgresql+psycopg2://postgres:postgres@localhost:5554/testtest")
        return Session(bind=engine)
```

Рисунок 9 – Класс для подключения к базе данных

В данном примере взята строка подключения для локальной базы на PostgreSQL, формируется она так:

```
postgresql+psycopg2://USER_NAME:PASSWORD@localhost:5432/DB_NAME
```

Вместо «USER_NAME» подставляете пользователя, под которым заходите на сервер, «PASSWORD» - пароль, который задавали при установке сервера и вместо «DB_NAME» - название базы данных, с которой планируете работать.

Статический метод «get_session()» создаёт подключение к базе данных и открывает сессию подключения, в которой мы можем обращаться к БД.

7 Миграция базы данных

После определения моделей таблиц и подключения к базе данных можно начать миграцию базы данных. Миграция базы данных – это перенос базы данных из кода в саму СУБД.

Для переноса нашей БД в PostgreSQL нужно использовать ранее упомянутую библиотеку «Alembic».

В терминале прописываем команду «alembic init alembic»

Рисунок 10 – Инициализация alembic в проекте

После этого в нашем проекте будет создана директория «alembic» и файл вне это директории «alembic.ini».

Открываем файл «alembic.ini», на 64 строчке вставляем ссылку подключения к нашей бд.

Рисунок 11 – Добавление строки подключения в файл «alembic.ini»

Для дальнейшей настройки раскрываем директорию «alembic» и открываем в ней файл «env.py». Импортируем в него наш файл «models.py» и на 23 строчке в «target_metadata» передаём нашу переменную «Base.metadata».

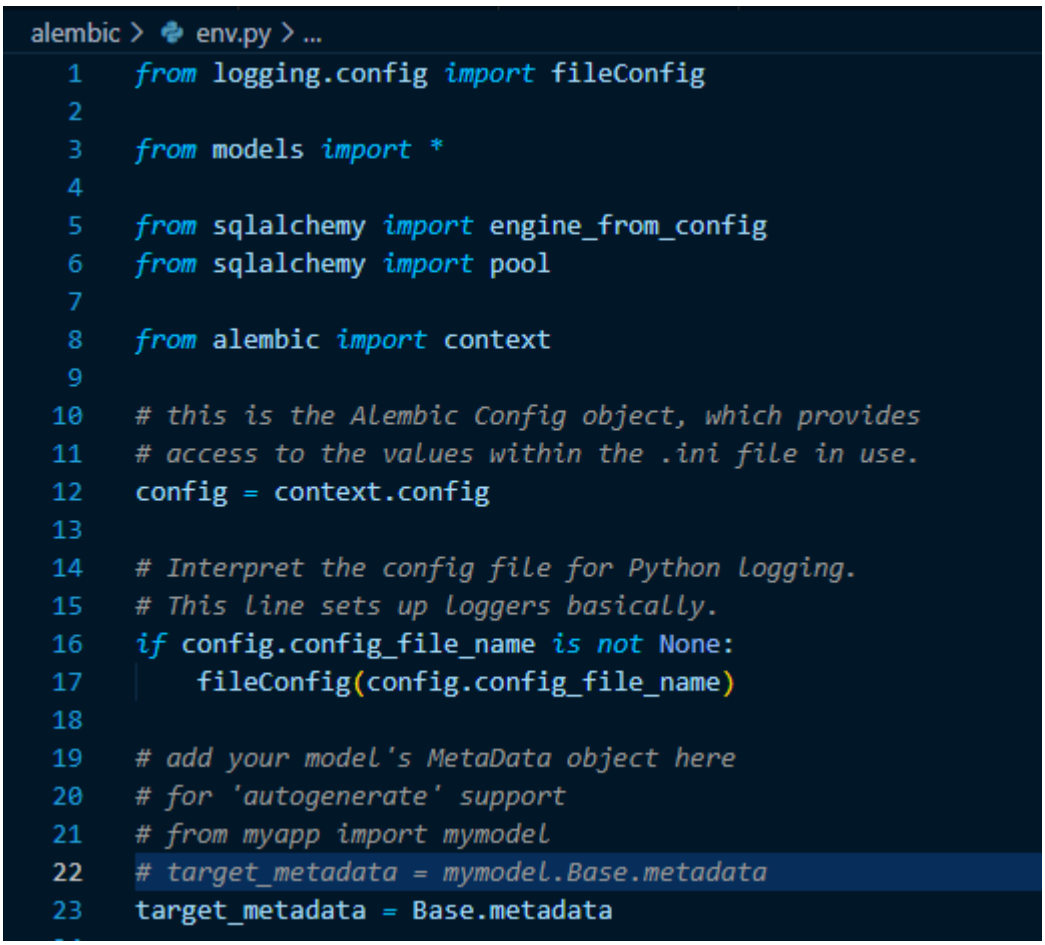

Рисунок 12 – Добавление сессии подключения к БД в файл «env.py»

После этих действий мы можем спокойно мигрировать нашу БД.Для этого нам нужно всего 2 команды. «alembic revision –autogenerate -m “ваш комментарий”». Данная команда фиксирует версию вашей БД(как в GIT). Далее нам нужно обновить нашу миграцию командой «alembic upgrade head».

Теперь у нас создались таблицы в нашей базе данных.

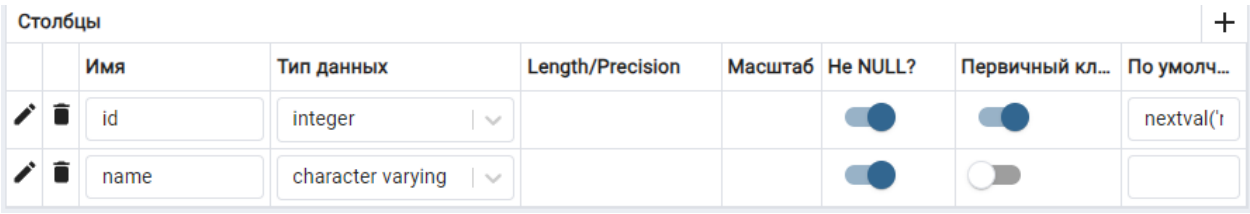

Рисунок 13 – Созданная таблица roles

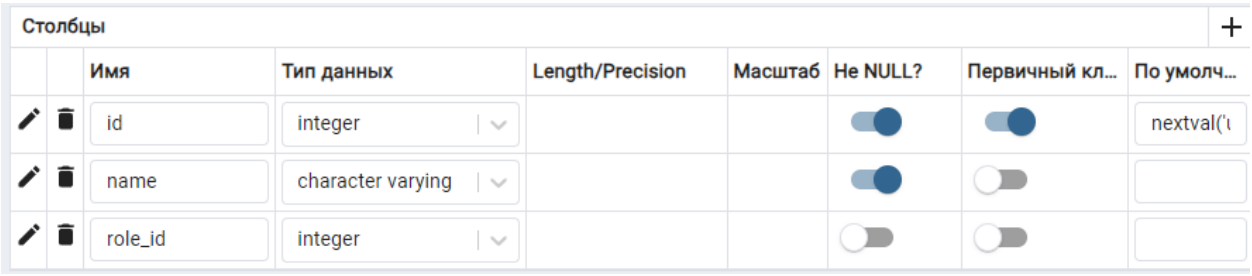

Рисунок 14 – Созданная таблица users

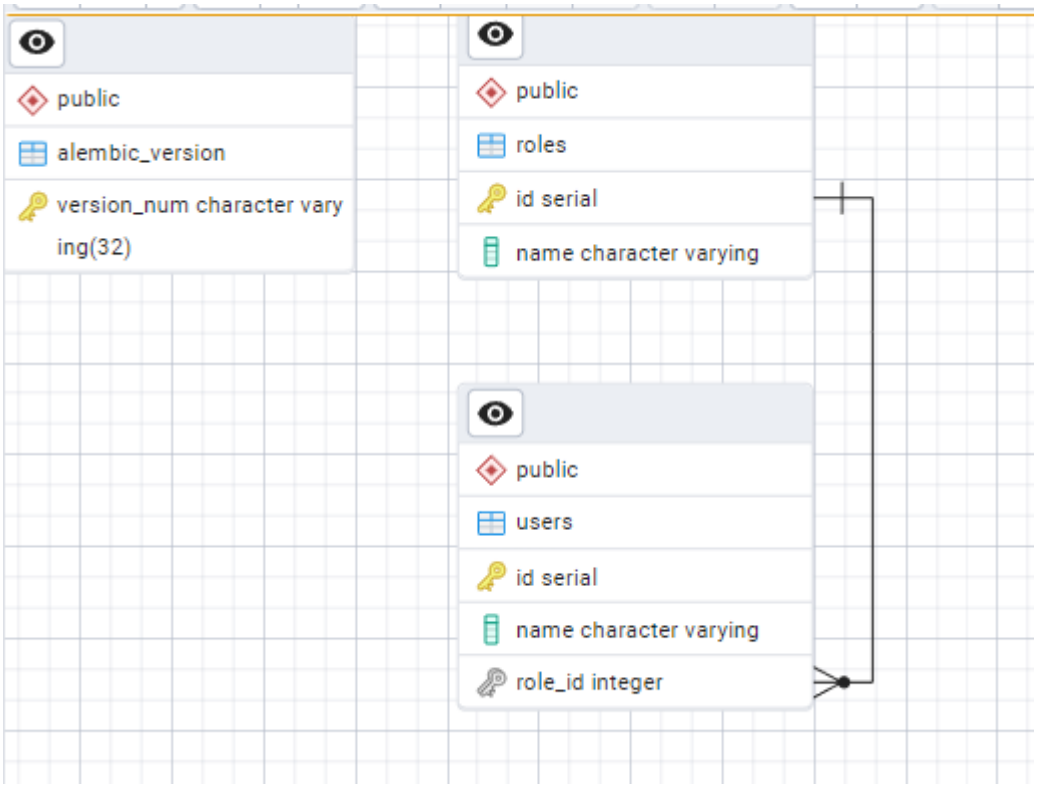

Рисунок 15 – ERD-диаграмма

8 Реализация API

Для начала стоит разобраться с тем, что вообще такое API.

API – это веб приложение, которое является прослойкой между Frontend и Backend частями приложения. Обычно API получается данные из Frontend, обрабатывает их через Backend и возвращает обратно во Frontend. Проще говоря, оно является прослойкой между клиентом и базой данных.

Создаём файл «api.py». В который импортируем библиотеки (см. рисунок 15).

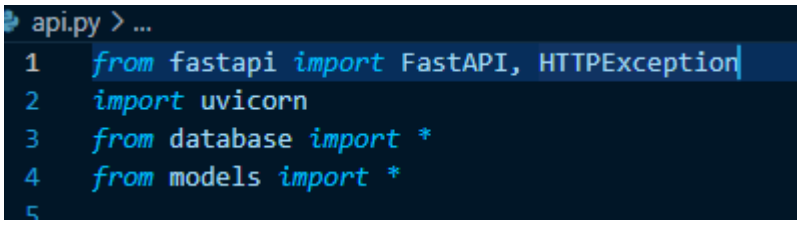

Рисунок 15 – Зависимости для файла «api.py»

Для удобства реализации запросов, удобнее будет создать ещё один файл с классами, под названием «response_models.py», который послужит нам как модели ответов, которые будут выводиться при выполнении запросов. Импортируйте их в файл «api.py»

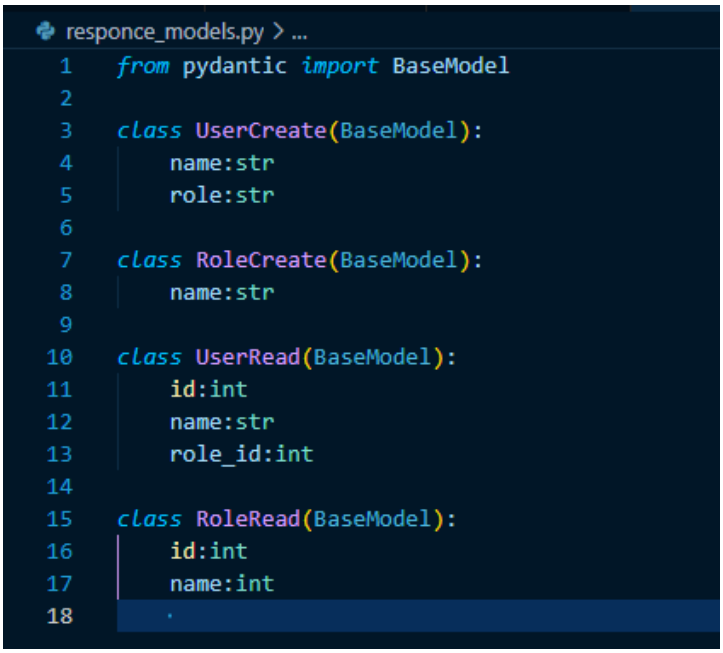

Рисунок 16 – Файл response_models.py

Продолжим написание нашего FastAPI приложения. Инициализируем переменную «app = FastAPI()».

Запуск FastAPI приложения происходит через инструмент uvicorn, поэтому прописываем в конец файла данную строчку:

```
uvicorn.run(app, host="127.0.0.1", port=8000)
```

```
app = FastAPI(
    title="yourtitle",
    description="pracAPI",
    version="1.0.0",
    docs_url="/docs",
    redoc_url="/redoc"
)
```

Рисунок 17 – Переменная app

Библиотека FastAPI славится быстрым и автоматическим формированием документации для API(Swagger). Передав в переменную «app» аргументы, как на фото, мы сформировали Swagger.

Коротко говоря, Swagger – это инструмент тестирования API приложений, в котором можно увидеть все используемые модели и методы.

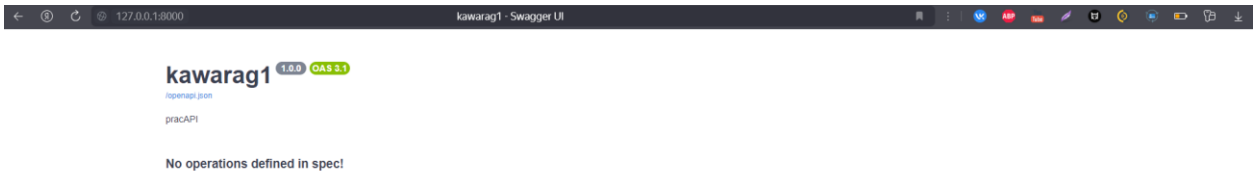

Рисунок 18 – Swagger

На данный момент он пустой, так как мы не прописали ни один метод. Чтобы перейти в Swagger, нужно открыть ссылку, которая выводится в консоль(см. рисунок 19) с конечной точкой «/docs».

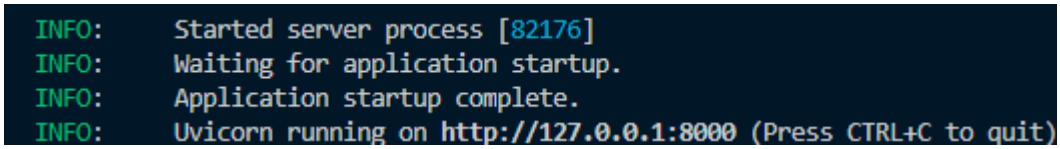

Рисунок 19 – Консольный вывод при запуске FastAPI приложения

Мы будем реализовывать в нашем API 4 вида запросов:

— GET
— POST
— PUT
— DELETE

В FastAPI для обозначения типа запроса используется декоратор. В нашем случае это будет «app.query», вместо query пишем тип запроса. В него мы будем передавать endpoints(конечные точки).

GET запрос:

Это запрос для получения данных. С его помощью мы будем вытягивать данные из нашей БД.

```python
@app.get("/users/select/{user_id}")
async def get_users(user_id:int):
    try:
        with DBSettings.get_session() as conn:
            user = conn.query(User).filter(User.id == user_id).first()
            return user
    except:
        raise HTTPException(status_code=404, detail="User not found")
```

Рисунок 20 – GET запрос

Указываем декоратор «@app.get» и в аргументы передаём нашу конечную точку, которая будет брать «user_id» из функции, привязанной к

этой конечной точке. (Важно в такие моменты указывать одинаковое название, иначе ничего не заработает.

Советую всю логику прописывать в обработчике исключений, чтобы непредвиденные ошибки не крашили наше приложение.

В этой функции мы будем вытягивать из нашей БД пользователя по его id. Для этого нам нужно раскрыть подключение к БД, в котором мы будем обращаться к нашей базе данных. Лучше при работе с ORM использовать подключение к БД через конструкцию «with open», иначе может произойти конфликт сессий и у вас возникнут непредвиденные ошибки. Инициализируем переменную «user», которой присваиваем объект, полученный из запроса. Строчка запроса реализует SQL код «SELECT FROM WHERE», обязательно в конце нужно будет указать метод «first()», который вытягивает нам объект по первому соответствию, ну и конечно, нам нужно вернуть этого пользователя.

В блоке «except», пропишем ситуацию, когда id пользователя не существует.

Поздравляю, у вас получилось написать первый API запрос, чтобы его проверить запустим наше приложение и откроем Swagger.

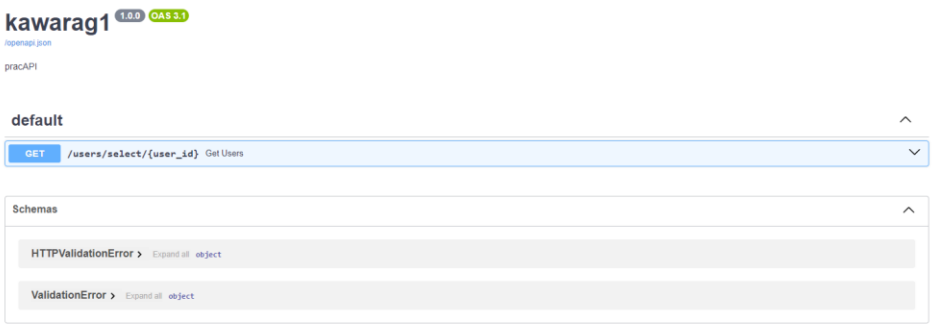

Рисунок 21 – Swagger с GET запросом

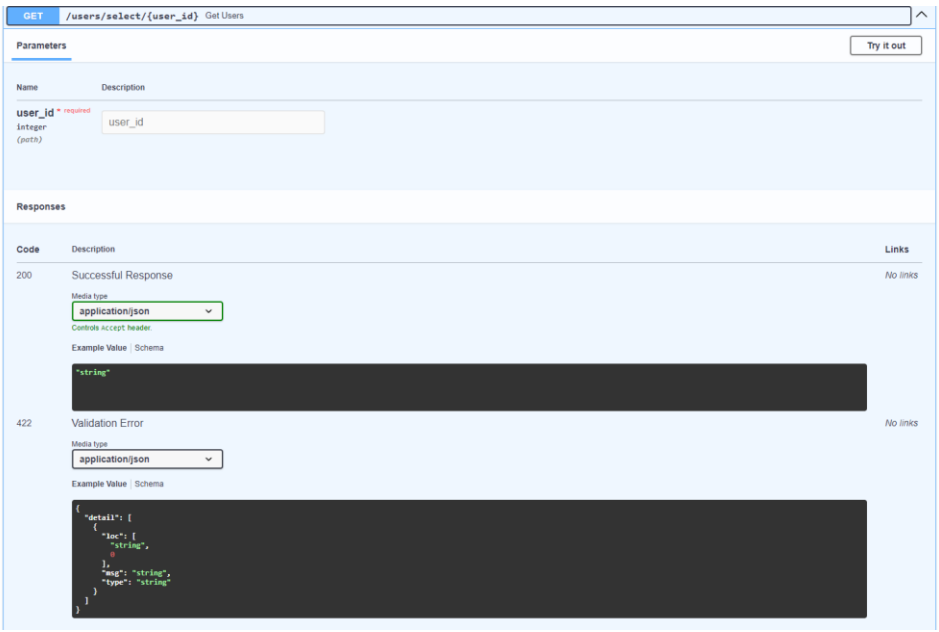

Рисунок 22 – GET запрос в Swagger’e

Раскрываем вкладку нашего запроса (см. рисунок 22) и видим его «внутренности», чтобы его протестировать, нажимаем кнопку «Try it out», вводим id пользователя и нажимаем кнопку «execute», если вы ввели id, который существует в вашей базе, то в блоке «Response body», нам выведется наш пользователь, иначе там будет «null»

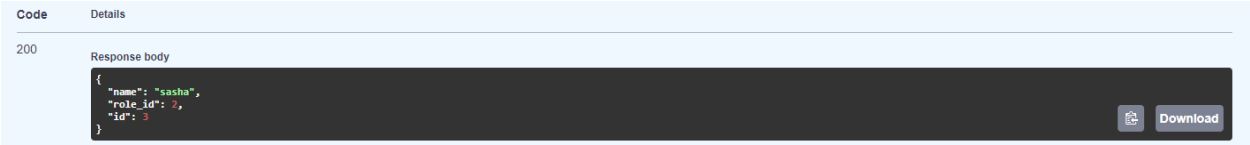

Рисунок 23 – Вывод по выполнению запроса

Перейдём к разработке POST запроса. Существенной разницы нет, но придётся немного подумать над логикой, так как у нас есть связь между таблицей users и таблицей roles. Значит, чтобы добавить пользователя, нам нужно будет сделать несколько запросов в базу данных, первый это получение id роли по её названию, а второй это добавление нашего пользователя.

```python
@app.post("/users/add", response_model= UserCreate)
async def add_users(user_name:str, user_role:str):
    user = UserCreate(name=user_name, role = user_role)
    with DBSettings.get_session() as conn:
        roleDB = conn.query(Role).filter(Role.name == user.role).first()
        if (roleDB == None):
            raise HTTPException(status_code=404, detail="We haven't this role")
        else:
            new_user = User(name = user.name, role_id = roleDB.id)
            conn.add(new_user)
            conn.commit()
            print("Успешно")
            return(user)
```

Рисунок 24 – POST запрос

Как вы могли заметить, в декораторе появился новый аргумент «response_model», в который мы передали класс «UserCreate» из файла «response_models.py». Этот аргумент задаёт конечной точке формат вывода ответа, таким образом мы сможем увидеть добавленного пользователя в формате JSON и сможем использовать приложения без возникновения каких-либо ошибок.

Передаём в функцию название пользователя и роли, внутри функции формируем объект класса «UserCreate». Далее раскрываем подключение в базу данных. В нём нам нужно вытянуть ID нашей роли, по названию этой роли, так же, учитываем, что может указываться не существующая роль, если же роль существует, то добавляем данные в базу и обязательно фиксируем эти данные, строчкой «conn.commit()». Для удобства возвращаем добавленного пользователя.

Запускаем наше приложение, открываем Swagger и по аналогии с GET запросом тестируем наш запрос. Если вы сделали всё правильно, то после выполнения запроса у вас в базе появится новая запись.

9 Самостоятельное задание

Самостоятельно реализуйте удаление и обновление данных в базе данных. Подсказка: обновление – это PUT запрос, а удаление – DELETE.

Конечный Swagger должен выглядеть так (см. рисунок 25):

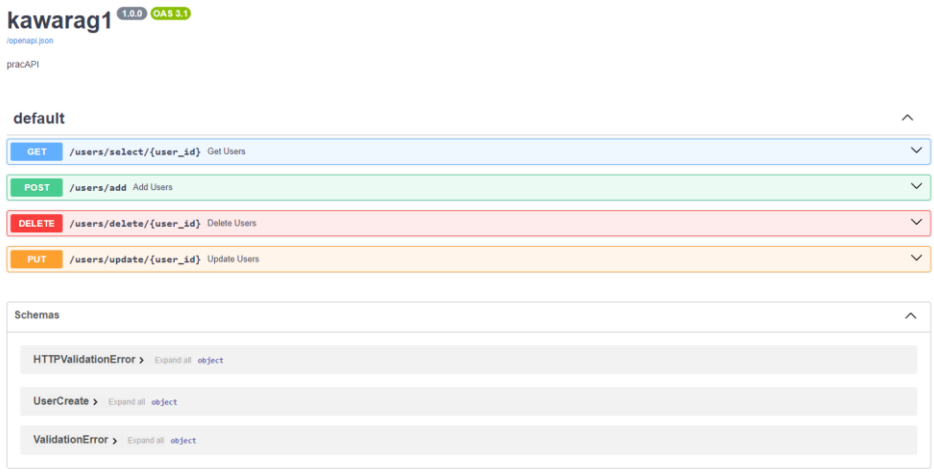

Рисунок 25 – итоговый вид Swagger