

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 фреймворка FastAPI и библиотеки SQLAlchemy	3
2 Установка библиотек	4
3 Необходимое ПО	9
4 Что такое ORM?	10
5 Определение моделей	11
6 Подключение к базе данных	13
7 Миграция базы данных	14
8 Реализация API.....	17
9 Самостоятельное задание.....	23

1 Общее описание веб фреймворка 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))

```
C:\Users\Ackerman>pip install poetry
Collecting poetry
  Downloading poetry-1.8.4-py3-none-any.whl.metadata (6.9 kB)
Collecting build<2.0.0,>=1.0.3 (from poetry)
  Downloading build-1.2.2.post1-py3-none-any.whl.metadata (6.5 kB)
Collecting cachecontrol<0.15.0,>=0.14.0 (from cachecontrol[filecache]<0.15.0,>=0.14.0->poetry)
  Downloading cachecontrol-0.14.1-py3-none-any.whl.metadata (3.1 kB)
Collecting cleo<3.0.0,>=2.1.0 (from poetry)
  Downloading cleo-2.1.0-py3-none-any.whl.metadata (12 kB)
Collecting crashtest<0.5.0,>=0.4.1 (from poetry)
  Downloading crashtest-0.4.1-py3-none-any.whl.metadata (1.1 kB)
Collecting dulwich<0.22.0,>=0.21.2 (from poetry)
  Downloading dulwich-0.21.7.tar.gz (448 kB)
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
```

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.

The screenshot shows a terminal window with several tabs at the top: PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is selected), and PORTS. The terminal content is as follows:

```
● PS C:\Users\dimak\pyprojects\method> poetry init

This command will guide you through creating your pyproject.toml config.

Package name [method]:
Version [0.1.0]:
Description []:
Author [kawaragi <dima.kretinin.05@mail.ru>, n to skip]:
License []:
Compatible Python versions [^3.10]:

Would you like to define your main dependencies interactively? (yes/no) [yes]
You can specify a package in the following forms:
- A single name (requests): this will search for matches on PyPI
- A name and a constraint (requests@^2.23.0)
- A git url (git+https://github.com/python-poetry/poetry.git)
- A git url with a revision (git+https://github.com/python-poetry/poetry.git#develop)
- A file path (../my-package/my-package.whl)
- A directory (../my-package/)
- A url (https://example.com/packages/my-package-0.1.0.tar.gz)

Package to add or search for (leave blank to skip):

Would you like to define your development dependencies interactively? (yes/no) [yes]
Package to add or search for (leave blank to skip):

Generated file

[tool.poetry]
name = "method"
version = "0.1.0"
description = ""
authors = ["kawaragi <dima.kretinin.05@mail.ru>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.10"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"

Do you confirm generation? (yes/no) [yes]
● PS C:\Users\dimak\pyprojects\method>
● PS C:\Users\dimak\pyprojects\method>
```

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
```

```
● PS C:\Users\dimak\pyprojects\method> poetry add sqlalchemy uvicorn alembic pydantic-settings psycopg2 asyncio fastapi  
Creating virtualenv method-40XsDete-py3.10 in C:\Users\dimak\AppData\Local\pypoetry\Cache\virtualenvs  
Using version ^2.0.36 for sqlalchemy  
Using version ^0.32.0 for uvicorn  
Using version ^1.14.0 for alembic  
Using version ^2.6.1 for pydantic-settings  
Using version ^2.9.10 for psycopg2  
Using version ^3.4.3 for asyncio  
Using version ^0.115.5 for fastapi  
  
Updating dependencies  
Resolving dependencies... (3.6s)  
  
Package operations: 23 installs, 0 updates, 0 removals  
  
- Installing exceptiongroup (1.2.2)  
- Installing idna (3.10)  
- Installing sniffio (1.3.1)  
- Installing typing-extensions (4.12.2)  
- Installing annotated-types (0.7.0)  
- Installing anyio (4.6.2.post1)  
- Installing colorama (0.4.6)  
- Installing greenlet (3.1.1)  
- Installing markupsafe (3.0.2)  
- Installing pydantic-core (2.23.4)  
- Installing click (8.1.7)  
- Installing h11 (0.14.0)  
- Installing mako (1.3.6)  
- Installing pydantic (2.9.2)  
- Installing python-dotenv (1.0.1)  
- Installing sqlalchemy (2.0.36)  
- Installing starlette (0.41.2)  
- Installing alembic (1.14.0)  
- Installing asyncio (3.4.3)  
- Installing fastapi (0.115.5)  
- Installing psycopg2 (2.9.10)  
- Installing pydantic-settings (2.6.1)  
- Installing uvicorn (0.32.0)  
  
Writing lock file  
○ PS C:\Users\dimak\pyprojects\method>
```

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)

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\dimak\pyprojects\method> poetry shell
Spawning shell within C:\Users\dimak\AppData\Local\pypoetry\Cache\virtualenvs\method-40XsDete-py3.10
PowerShell 7.4.6
(method-py3.10) PS C:\Users\dimak\pyprojects\method> poetry env info

Virtualenv
Python:      3.10.10
Implementation: CPython
Path:        C:\Users\dimak\AppData\Local\pypoetry\Cache\virtualenvs\method-40XsDete-py3.10
Executable:  C:\Users\dimak\AppData\Local\pypoetry\Cache\virtualenvs\method-40XsDete-py3.10\Scripts\python.exe
Valid:       True

Base
Platform:   win32
OS:          nt
Python:      3.10.10
Path:        C:\Users\dimak\AppData\Local\Programs\Python\Python310
Executable:  C:\Users\dimak\AppData\Local\Programs\Python\Python310\python.exe
(method-py3.10) PS C:\Users\dimak\pyprojects\method>
```

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.

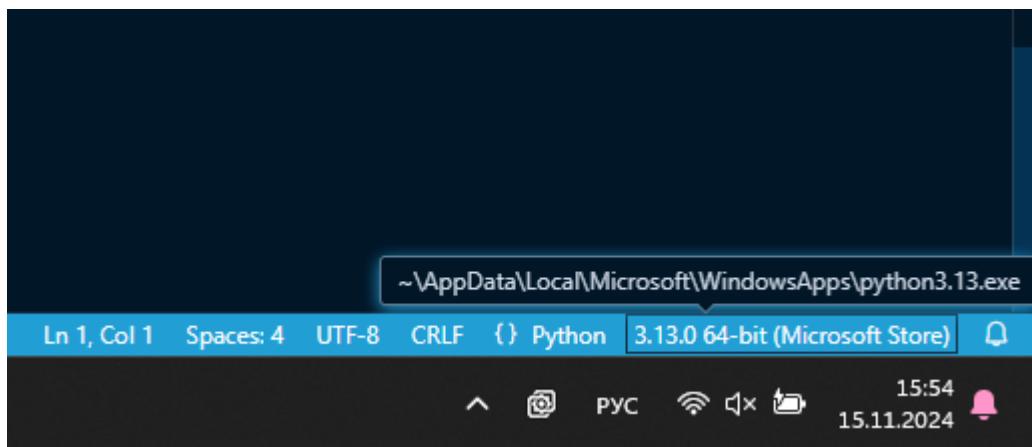


Рисунок 5 – кнопка выбора интерпретатора

Сверху появится следующее окно:

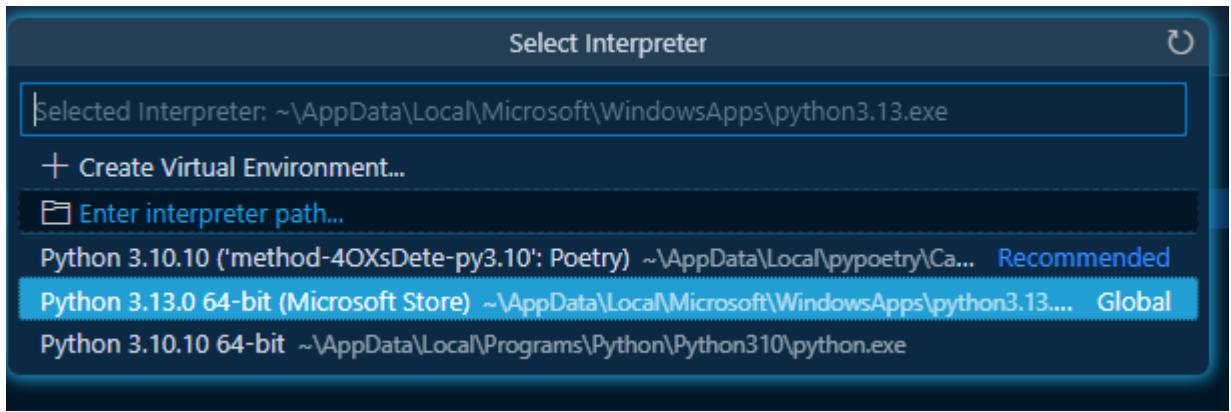


Рисунок 6 – добавление нового интерпретатора

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 браузер: 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("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»

```
PS C:\Users\dimak\pyprojects\alchemy> 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.

```
63  
64 sqlalchemy.url = postgresql+psycopg2://postgres:postgres@localhost:5554/testtest  
65
```

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».

```

alembic > env.py > ...
  1  from logging.config import fileConfig
  2
  3  from models import *
  4
  5  from sqlalchemy import engine_from_config
  6  from sqlalchemy import pool
  7
  8  from alembic import context
  9
 10 # this is the Alembic Config object, which provides
 11 # access to the values within the .ini file in use.
 12 config = context.config
 13
 14 # Interpret the config file for Python Logging.
 15 # This line sets up Loggers basically.
 16 if config.config_file_name is not None:
 17     fileConfig(config.config_file_name)
 18
 19 # add your model's MetaData object here
 20 # for 'autogenerate' support
 21 # from myapp import mymodel
 22 # target_metadata = mymodel.Base.metadata
 23 target_metadata = 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.

Столбцы								+
	Имя	Тип данных	Length/Precision	Масштаб	Не NULL?	Первичный кл...	По умолч...	
	id	integer			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="text" value="nextval('r"/>	
	name	character varying			<input checked="" type="checkbox"/>	<input type="checkbox"/>		

Рисунок 13 – Созданная таблица roles

Столбцы								+
	Имя	Тип данных	Length/Precision	Масштаб	Не NULL?	Первичный кл...	По умолч...	
		id	integer		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	nextval('l
		name	character varying		<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
		role_id	integer		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Рисунок 14 – Созданная таблица users

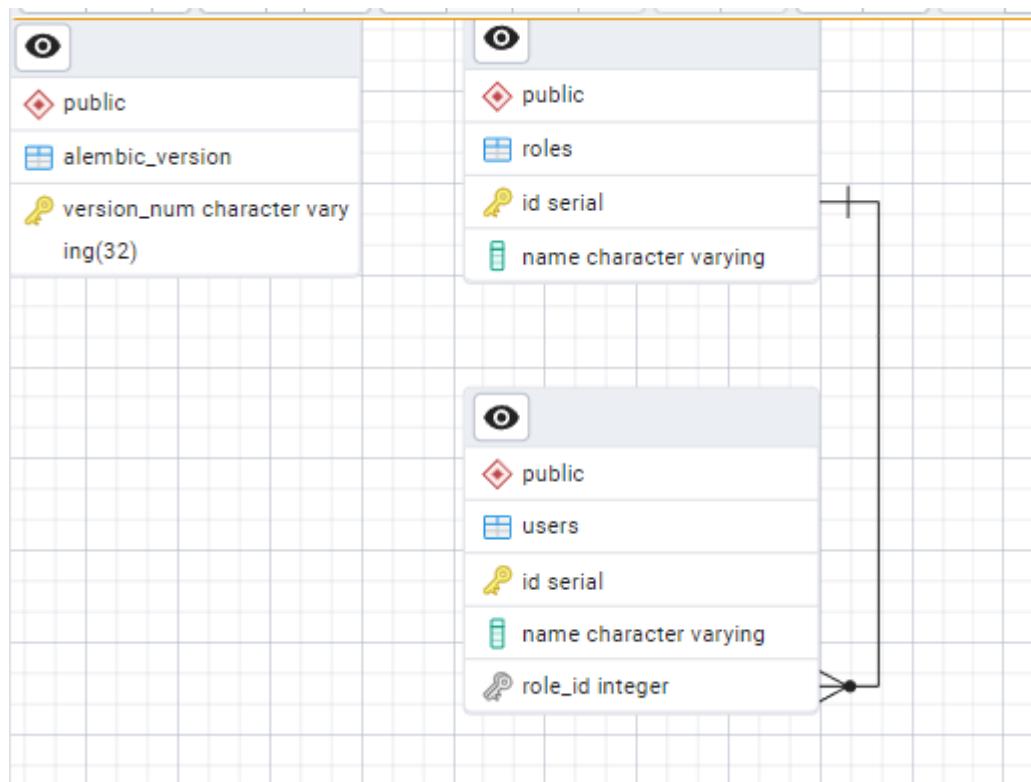


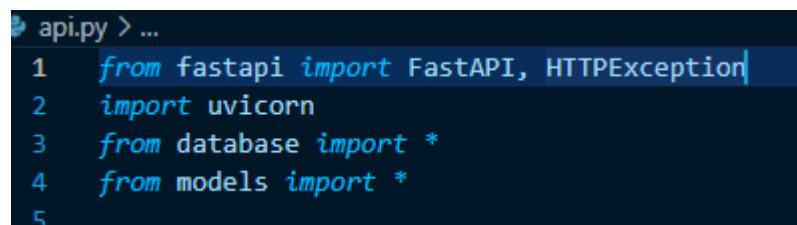
Рисунок 15 – ERD-диаграмма

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 получает 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).



```
api.py > ...
1  from fastapi import FastAPI, HTTPException
2  import uvicorn
3  from database import *
4  from models import *
5
```

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»



```
response_models.py > ...
1  from pydantic import BaseModel
2
3  class UserCreate(BaseModel):
4      name:str
5      role:str
6
7  class RoleCreate(BaseModel):
8      name:str
9
10 class UserRead(BaseModel):
11     id:int
12     name:str
13     role_id:int
14
15 class RoleRead(BaseModel):
16     id:int
17     name:int
18
```

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».

```
INFO:     Started server process [82176]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

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.

```
@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 соответству, 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.

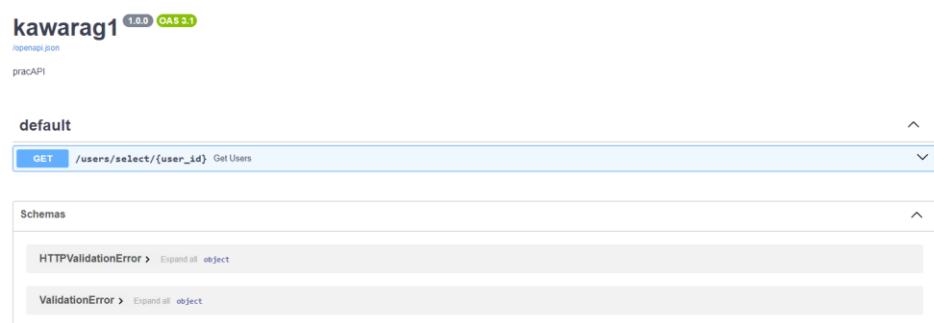


Рисунок 21 – Swagger с GET запросом

The screenshot shows the Swagger UI interface for a GET request. In the 'Parameters' section, there is a single parameter named 'user_id' which is required and of type integer (path). In the 'Responses' section, there are two entries: a 'Successful Response' (200) with a media type of application/json and a schema of 'string', and a 'Validation Error' (422) with a media type of application/json and a complex schema involving 'detail', 'loc', 'msg', and 'type' fields.

Рисунок 22 – GET запрос в Swagger’е

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»

The screenshot shows the execution results of the GET request. The status code is 200, and the response body is a JSON object with the following structure: { "name": "sasha", "role_id": 2, "id": 3 }. There are download and copy icons next to the response body.

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.

```
@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 оказываться 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):

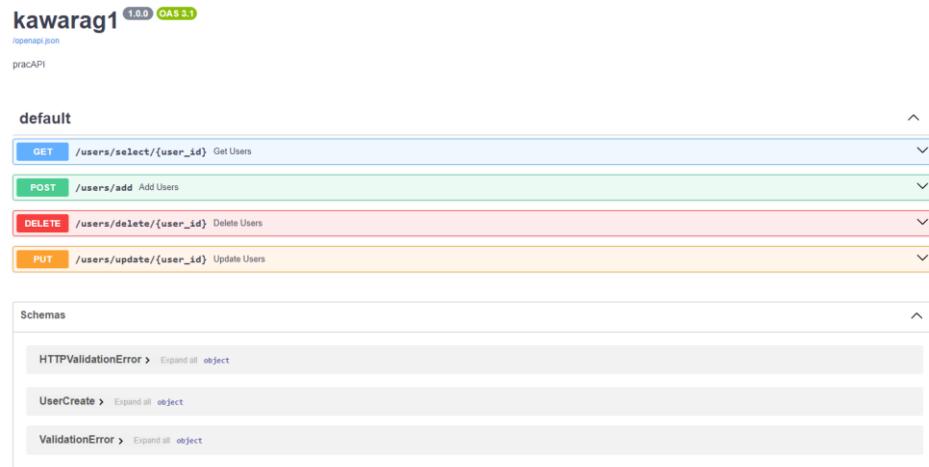


Рисунок 25 – итоговый вид Swagger