**Министерство науки и высшего образования Российской Федерации**

**ФГАОУ ВО «УрФУ имени первого Президента России Б.Н. Ельцина»**

Кафедра «школа бакалавриата (школа)»

Оценка работы\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Руководитель от УрФУ Свинцов Д.В.

Тема задания на практику

**Основы Веб API и базовые Веб протоколы**

ОТЧЕТ

Вид практики Учебная практика

Тип практики Учебная, ознакомительная

Студент Хуснутдинов Р.А.

**ФИО студента**

Специальность (направление подготовки) 09.03.01 Информатика и вычислительная техника

Группа РИ-210912

Екатеринбург 2023

СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 2](#_Toc141814800)

[ВВЕДЕНИЕ 3](#_Toc141814801)

[ПЕРВИЧНАЯ ПОДГОТОВКА 3](#_Toc141814802)

[ПАРСИНГ 5](#_Toc141814803)

[БАЗА ДАННЫХ 7](#_Toc141814804)

[FAST API 8](#_Toc141814805)

[РЕЗУЛЬТАТЫ ПРАКТИКИ 14](#_Toc141814806)

[ЗАКЛЮЧЕНИЕ 14](#_Toc141814807)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 16](#_Toc141814808)

ВВЕДЕНИЕ

На данной практике были рассмотренные основы веб программирования и построения REST API. Мы изучили веб протоколы, из чего состоит запрос, и использование клиент-серверной архитектуры. И так же познакомились с созданием REST API, для конечной цели практики – создание REST API приложения на Python с подключенной базой данных.

ПЕРВИЧНАЯ ПОДГОТОВКА

В начале практики мы начали изучать основы веб программирования и HTTP. Что такое HTTP протокол, клиент-серверная архитектура, из чего состоит HTTP сообщение, структура URL, и какие методы используются для работы веб приложений.

HTTP (HyperText Transfer Protocol) — это протокол передачи данных, который используется для обмена информацией между клиентами и серверами в сети Интернет. Он является основным протоколом для передачи данных в вебе и обеспечивает взаимодействие между веб-браузерами (клиентами) и веб-серверами.

HTTP определяет различные методы запросов, которые клиент может использовать для взаимодействия с сервером. Некоторые из распространенных методов включают GET (получение данных), POST (отправка данных на сервер), PUT (обновление данных на сервере) и DELETE (удаление данных на сервере). Это методы дают нам возможность построить REST API. (RESTful API).

После чего мы использовали Telnet и OpenSSL для формирования вручную запросов на доступные сервера.

Telnet — это протокол прикладного уровня, который позволяет устанавливать текстовые сетевые соединения и взаимодействовать с удаленными узлами. Он предоставляет возможность удаленного доступа к устройствам, серверам или компьютерам через командную строку.

Пример подключения через Telnet:

$ telnet wikipedia.org 443

Trying 91.198.174.192...

Connected to wikipedia.org.

Escape character is '^]'.

GET /ip HTTP/1.1

Host: wikipedia.org

OpenSSL — это библиотека с открытым исходным кодом для криптографии, которая предоставляет различные криптографические функции, алгоритмы и утилиты. Она поддерживает различные протоколы безопасности, такие как SSL (Secure Sockets Layer) и TLS (Transport Layer Security), и предоставляет широкий спектр функций, включая шифрование, дешифрование, генерацию цифровых подписей, создание и проверку сертификатов и т.д.

Пример подключения через OpenSSL:

$ openssl s\_client -connect wikipedia.org:443

GET /wiki/страница HTTP/1.1

Host: ru.wikipedia.org

User-Agent: Mozilla/5.0 (X11; U; Linux i686; ru; rv:1.9b5) Gecko/2008050509 Firefox/3.0b5

Accept: text/html

Connection: close

(пустая строка)

Как выглядит успешный запрос.



ПАРСИНГ

*Парсинг (англ. parsing) — это процесс анализа и извлечения структурированных данных из исходного текста или документа.*

После того как мы разобрались с тем, как работает передача данных в интернете, мы с помощью Python , и библиотеками requests и selenium, начали парсить информацию с сайта <https://www.onlinetrade.ru>.

Requests — это библиотека Python для выполнения HTTP-запросов. Она обеспечивает простой и удобный способ отправки HTTP-запросов на сервер.

Для начала мы использовали библиотеку requests, чтобы получить доступ к https://www.onlinetrade.ru/. С помощью HTTP-запросов мы получили HTML-код страницы и затем использовали BeautifulSoup, для извлечения нужной информации из HTML-структуры. Конкретно мы извлекли название и цену.

Во время работы мы столкнулись с проблемой, что сайт блокирует обычный request из библиотеки, поэтому нам пришлось добавить заголовки к запросу, которые мы взяли из своего браузера, в инструментах разработчика.

*Python.*

*import* requests *as* r  
*from* bs4 *import* BeautifulSoup *as* bs  
hs = { ...  
}  
URL = "https://www.onlinetrade.ru/catalogue/monitory-c23/dell/monitor\_dell\_s3222dgm\_31.5\_black\_3222\_5120-2833680.html"  
req = r.get(URL, headers=hs)  
html = req.text  
soup = bs(html, "lxml")  
title = soup.find("span", itemprop="name").get\_text()  
price = soup.find("span", itemprop="price").get\_text()

Обойти защиту от парсинга библиотекой requests некоторых сайтов довольно затруднительно, поэтому можно использовать Selenium.

Selenium — это набор инструментов и библиотек для автоматизации действий веб-браузера.

*Python.*

*from* selenium *import* webdriver  
*from* bs4 *import* BeautifulSoup *as* bs  
*def* get\_info(URL):  
 driver = webdriver.Chrome()  
 driver.get(URL)  
 html = driver.page\_source  
 driver.close()  
 driver.quit()  
 soup = bs(html, "lxml")  
 title = soup.find("span", class\_="title-info-title-text").get\_text()  
 price = soup.find("span", itemprop="price").get\_text()  
 rating = soup.find("strong", class\_="styles-module-root-LIAav styles-module-size\_s-awPvv")  
 *return* {  
 'title':title,  
 'price':price,  
 'rating':rating  
 }

БАЗА ДАННЫХ

После изучения способа получать данные, их нужно где-то хранить. В рамках данной практики мы использовали SQLite, и в него загружали структурированную информацию с нужных сайтов.

Мы использовали реляционные базы данных SQL, такие как SQLite, для хранения, управления и организации данных. Реляционные базы данных предоставляют структурированный подход к хранению информации, где данные представлены в виде таблиц с рядами и столбцами.

SQLite - это легковесная, встраиваемая реляционная база данных, которая обеспечивает хранение и управление данными. SQLite является одним из наиболее популярных СУБД (систем управления базами данных) и отличается своей простотой, портативностью и надежностью.

Работа с базой данных происходила с помощью SQLAlchemy – библиотека для Python, которая предоставляет удобный и выразительный способ взаимодействия с базами данных, используя язык Python, вместо языка SQL.

Во время практики мы разобрали два варианта работы с БД *(базой данных)*.

1. Подключение к БД с помощью engine.connect / execute , где нужно прописывать команды на языке SQL
2. Создание Declarative Base, что является работой с БД, с помощью синтаксиса создания класса в Python.

После создания БД, для проверки, в нее была загружена первая информация с парсинга.

FAST API

Последний инструмент, который нужен для создания REST API, это FastAPI.

FastAPI — это современный фреймворк для создания веб-приложений и API на языке программирования Python. Он был разработан для обеспечения высокой производительности, простоты использования и быстрой разработки.

С помощью него мы создали начальный функционал итогового проекта, без подключения с БД (вместо нее был обычный словарь). В начальный функционал входили CRUD методы (create, read, update, delete).

Для создания нам потребуется Python, SQLAlchemy, Pydantic, FastApi

Pydantic - это библиотека для языка программирования Python, предназначенная для валидации данных и создания схем (сериализации и десериализации) данных. Она позволяет определять схемы данных (модели) с помощью стандартных типов Python и дополнительных аннотаций Pydantic. Эти схемы могут использоваться для проверки входных данных, автоматической валидации и преобразования данных в соответствии с этими схемами.

Процесс создания REST API:

1. Создание БД с помощью SQLAlchemy

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

SQLALCHEMY\_DATABASE\_URL = "sqlite:///./sql\_app.db"

# SQLALCHEMY\_DATABASE\_URL = "postgresql://user:password@postgresserver/db"

engine = create\_engine(

SQLALCHEMY\_DATABASE\_URL, connect\_args={"check\_same\_thread": False}

)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative\_base()

1. Создание моделей для валидации

from sqlalchemy import Boolean, Column, ForeignKey, Integer, String

from sqlalchemy.orm import relationship

from .database import Base

class User(Base):

\_\_tablename\_\_ = "users"

id = Column(Integer, primary\_key=True, index=True)

email = Column(String, unique=True, index=True)

hashed\_password = Column(String)

is\_active = Column(Boolean, default=True)

items = relationship("Item", back\_populates="owner")

class Item(Base):

\_\_tablename\_\_ = "items"

id = Column(Integer, primary\_key=True, index=True)

title = Column(String, index=True)

description = Column(String, index=True)

owner\_id = Column(Integer, ForeignKey("users.id"))

owner = relationship("User", back\_populates="items")

1. Создание pydantic моделей

from pydantic import BaseModel

class ItemBase(BaseModel):

title: str

description: str | None = None

class ItemCreate(ItemBase):

pass

class Item(ItemBase):

id: int

owner\_id: int

class Config:

orm\_mode = True

class UserBase(BaseModel):

email: str

class UserCreate(UserBase):

password: str

class User(UserBase):

id: int

is\_active: bool

items: list[Item] = []

class Config:

orm\_mode = True

1. Создание CRUD

from sqlalchemy.orm import Session

from . import models, schemas

def get\_user(db: Session, user\_id: int):

return db.query(models.User).filter(models.User.id == user\_id).first()

def get\_user\_by\_email(db: Session, email: str):

return db.query(models.User).filter(models.User.email == email).first()

def get\_users(db: Session, skip: int = 0, limit: int = 100):

return db.query(models.User).offset(skip).limit(limit).all()

def create\_user(db: Session, user: schemas.UserCreate):

fake\_hashed\_password = user.password + "notreallyhashed"

db\_user = models.User(email=user.email, hashed\_password=fake\_hashed\_password)

db.add(db\_user)

db.commit()

db.refresh(db\_user)

return db\_user

def get\_items(db: Session, skip: int = 0, limit: int = 100):

return db.query(models.Item).offset(skip).limit(limit).all()

def create\_user\_item(db: Session, item: schemas.ItemCreate, user\_id: int):

db\_item = models.Item(\*\*item.dict(), owner\_id=user\_id)

db.add(db\_item)

db.commit()

db.refresh(db\_item)

return db\_item

1. Main файл

from fastapi import Depends, FastAPI, HTTPException

from sqlalchemy.orm import Session

from . import crud, models, schemas

from .database import SessionLocal, engine

models.Base.metadata.create\_all(bind=engine)

app = FastAPI()

# Dependency

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

@app.post("/users/", response\_model=schemas.User)

def create\_user(user: schemas.UserCreate, db: Session = Depends(get\_db)):

db\_user = crud.get\_user\_by\_email(db, email=user.email)

if db\_user:

raise HTTPException(status\_code=400, detail="Email already registered")

return crud.create\_user(db=db, user=user)

@app.get("/users/", response\_model=list[schemas.User])

def read\_users(skip: int = 0, limit: int = 100, db: Session = Depends(get\_db)):

users = crud.get\_users(db, skip=skip, limit=limit)

return users

@app.get("/users/{user\_id}", response\_model=schemas.User)

def read\_user(user\_id: int, db: Session = Depends(get\_db)):

db\_user = crud.get\_user(db, user\_id=user\_id)

if db\_user is None:

raise HTTPException(status\_code=404, detail="User not found")

return db\_user

@app.post("/users/{user\_id}/items/", response\_model=schemas.Item)

def create\_item\_for\_user(

user\_id: int, item: schemas.ItemCreate, db: Session = Depends(get\_db)

):

return crud.create\_user\_item(db=db, item=item, user\_id=user\_id)

@app.get("/items/", response\_model=list[schemas.Item])

def read\_items(skip: int = 0, limit: int = 100, db: Session = Depends(get\_db)):

items = crud.get\_items(db, skip=skip, limit=limit)

return items

РЕЗУЛЬТАТЫ ПРАКТИКИ

В итоговом проекте было необходимо объединить все что было раньше написано:

1. Парсер
2. БД
3. Backend

И это будет являться полноценным REST API. После чего, web приложение нужно загрузить на один из доступных хостингов.

Мое приложение отличает то, что для заполнения БД, не нужно отправлять полные JSON объекты, с названием, ценой и т.д. Достаточно отправить объект с пустыми полями, и ссылкой, после чего скрипт автоматически заполнит объект, и добавит его в БД.

Ссылка на проект:

<https://replit.com/@TELEBON/crudapp>

<https://github.com/RO1T/crudapp>

ЗАКЛЮЧЕНИЕ

В рамках данной практики мы успешно изучили основы веб-программирования и научились строить полноценный фундамент REST API с использованием базы данных и заполнением через парсер. В ходе этой практики, мы ознакомились с ключевыми концепциями веб-разработки, такими как HTTP протокол, клиент-серверное взаимодействие и архитектура RESTful API.

Научившись работать с базой данных, в частности с реляционными базами данных SQL, мы приобрели умения в проектировании схемы базы данных, создании таблиц, выполнении запросов на выборку, вставку, обновление и удаление данных.

Парсинг данных из внешних источников позволил нам автоматизировать процесс заполнения базы данных информацией. Это оказалось особенно полезным при работе с большим объемом данных или при регулярном обновлении информации.

Основы веб-программирования и создание REST API являются важными навыками для разработчика, позволяющими создавать современные веб-приложения и сервисы с открытым интерфейсом. REST API предоставляет удобные и гибкие методы взаимодействия с данными, а использование баз данных обеспечивает эффективное хранение и управление информацией.

В процессе работы над проектом мы столкнулись с различными вызовами и проблемами, которые успешно преодолели, что дало нам ценный опыт и понимание работы с веб-технологиями и базами данных.

В целом, данная практика позволила нам расширить наши знания и навыки в области веб-программирования и подготовила нас к более сложным и захватывающим проектам. Мы теперь осознаем важность правильного проектирования API и баз данных для обеспечения надежности, производительности и удобства использования наших приложений.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. <https://lectureswww.readthedocs.io/>
2. <https://fastapi.tiangolo.com/>
3. <https://www.sqlalchemy.org/>
4. <https://ru.wikipedia.org/wiki/>
5. <https://www.selenium.dev/documentation/>
6. <https://stackoverflow.com/>
7. <https://docs.pydantic.dev/latest/api/base_model/>
8. <https://requests.readthedocs.io/en/latest/>
9. <https://www.openssl.org/docs/>
10. <https://developer.mozilla.org/ru/docs/Web/HTTP/Overview>