**Министерство науки и высшего образования Российской Федерации**
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

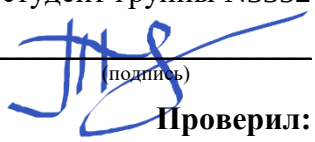
**Факультет безопасности информационных технологий**

**Дисциплина:**
«Web программирование»

# ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

**Выполнил:**

Рядовой Т.С., студент группы N3352



(подпись)

**Проверил:**
Менщиков Александр Алексеевич

(отметка о выполнении)

(подпись)

Санкт-Петербург
2024 г.

**СОДЕРЖАНИЕ**

# ВВЕДЕНИЕ

Цель работы: разработка онлайн-чата на языке программирования Python с использованием веб-фреймровка FastAPI.

# 1 ОПИСАНИЕ АРХИТЕКТУРЫ ПО

Проект представляет собой онлайн-чат для пользователей, состоящий из двух микросервисов:

1. Website – отвечает за регистрацию, аутентификацию пользователей, создание и управление чат-комнатами.
2. Chat – реализует функционал WebSocket для мгновенного обмена сообщениями между пользователями.

Хранение данных осуществляется в PostgreSQL. Взаимодействие с пользователями реализовано через HTML-шаблоны Jinja2 на фронтенде и JavaScript для работы с WebSocket API. Веб-сервер Nginx выполняет проксирование запросов к соответствующим сервисам.

Проект упакован в Docker, с использованием docker-compose, где каждый компонент развернут в своем контейнере.

## 1.1 Микросервис chat

Задачи:

- Управление WebSocket-подключениями пользователей;
- Проверка JWT-токена для аутентификации;
- Подключение пользователей к чат-комнате;
- Отправка сообщений всем участникам комнаты;
- Оповещение о входе/выходе пользователей.

Технологии:

- FastAPI (WebSockets) – сервер WebSocket;
- JWT – проверка аутентификации пользователей;
- JSON – формат сообщений;
- WebSockets (JavaScript) – фронтенд-связь с сервером.

Основной WebSocket-эндпоинт:

- ws://host/ws/roomId– подключение к чату.

Механика работы WebSocket:

1. Пользователь передает JWT-токен при подключении;
2. Сервер проверяет подлинность токена и извлекает email пользователя;

3. Пользователь подключается к комнате и получает уведомление о входе нового участника;
4. Все сообщения от пользователей рассылаются в JSON-формате всем участникам комнаты;
5. При разрыве соединения отправляется уведомление о выходе пользователя.

### 1.2 Микросервис website

Задачи:

- Управление пользователями (регистрация, аутентификация);
- Создание и удаление чат-комнат;
- Поиск комнат по названию;
- Генерация JWT-токенов для аутентификации в WebSocket-чате;
- Рендеринг HTML-страниц с использованием Jinja2;
- Взаимодействие с WebSocket-сервисом через JavaScript API.

Технологии:

- FastAPI – бэкенд API;
- Jinja2 – шаблоны для рендеринга страниц;
- SQLAlchemy – ORM для работы с PostgreSQL;
- JWT – аутентификация пользователей;
- HTML, JavaScript – фронтенд.

### 1.3 База данных

Выбрана PostgreSQL. Доступ к ней осуществляется по логину и паролю.

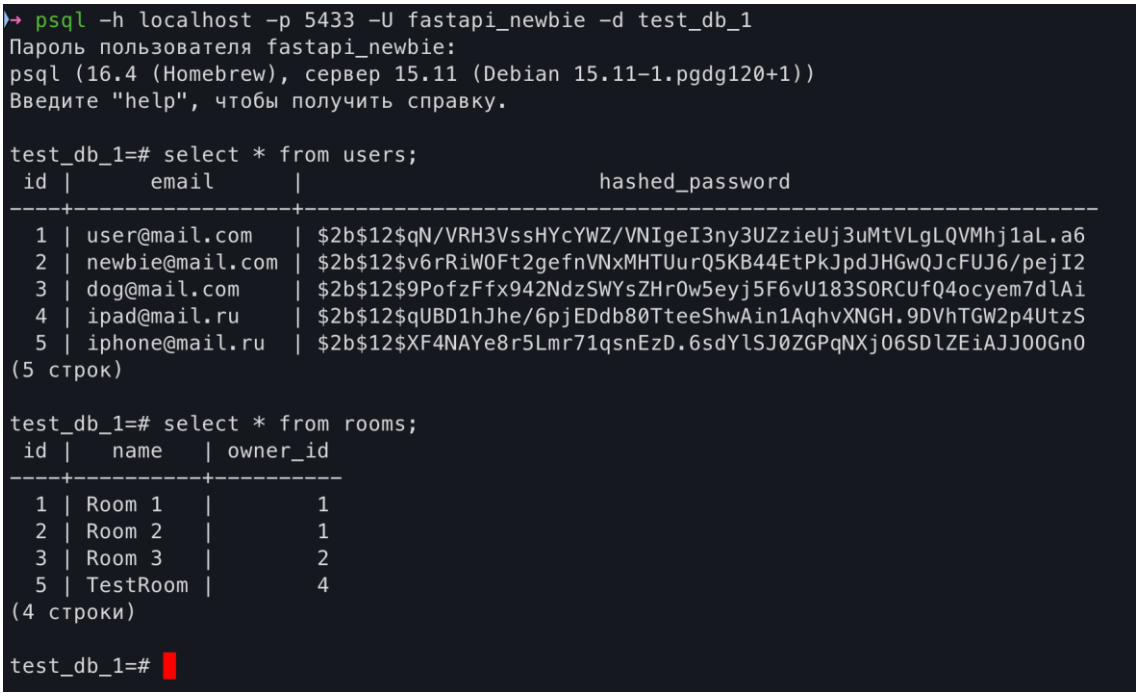



Рисунок 1 – Подключение к базе данных локально

Листинг 1 – init.sql

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR UNIQUE NOT NULL,
    hashed_password VARCHAR NOT NULL
);

CREATE TABLE rooms (
    id SERIAL PRIMARY KEY,
    name VARCHAR NOT NULL,
    owner_id INTEGER,
    FOREIGN KEY (owner_id) REFERENCES users(id)
);

INSERT INTO users (email, hashed_password) VALUES
('user@mail.com',
'$2b$12$qN/VRH3VssHYcYWZ/VNIgeI3ny3UZzieUj3uMtVLgLQVMhj1aL.a6'),  -- 1111
('newbie@mail.com',
'$2b$12$v6rRiWOFt2gefnVNxMHTUurQ5KB44EtPkJpdJHGwQJcFUJ6/pejI2');  -- 1122

INSERT INTO rooms (name, owner_id) VALUES
('Room 1', 1),
('Room 2', 1),
('Room 3', 2);
```

Листинг 2 – Фрагмент из docker-compose.yml для настройки базы данных

```
db:
    image: postgres:15
    container_name: postgres-lab1
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
```

```
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    ports:
      - "5433:5432"
    volumes:
      - db_data:/var/lib/postgresql/data
      - ./db/init.sql:/docker-entrypoint-initdb.d/init.sql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d ${POSTGRES_DB}"]
      interval: 1s
      timeout: 5s
      retries: 10
```

### 1.4 Веб-сервер (Nginx)

Функции:

- Проксирование HTTP-запросов к сервису Website;
- Проксирование WebSocket-запросов к сервису Chat;
- Логирование запросов.

Листинг 3 – Конфигурация сервера Nginx

```
server {
    listen 80;

    location / {
        proxy_pass http://website:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    location /ws {
        proxy_pass http://chat:8001;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_set_header Host $host;
    }

    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;
}
```

## 2 ОПИСАНИЕ СТРУКТУРЫ БАЗЫ ДАННЫХ

В проекте используется база данных PostgreSQL. Она хранит информацию о пользователях и чат-комнатах. Все данные управляются через ORM (SQLAlchemy) для удобства взаимодействия с кодом на Python.

Связи:

- Один пользователь может владеть несколькими комнатами (один-ко-многим);
- Ключ owner_id является внешним (FOREIGN KEY), ссылающимся на users.id.

### 2.1 Таблица users (Пользователи)

Содержит данные о зарегистрированных пользователях.

Таблица 1 – Таблица пользователей

| Поле | Тип данных | Описание |
|---|---|---|
| id | SERIAL PRIMARY KEY | Уникальный идентификатор пользователя |
| email | VARCHAR(255) UNIQUE NOT NULL | Почта пользователя |
| hashed_password | VARCHAR(255) NOT NULL | Хешированный пароль |

### 2.2 Таблица rooms (Чат-комнаты)

Содержит данные о созданных чат-комнатах.

Таблица 2 – Таблица комнат

| Поле | Тип данных | Описание |
|---|---|---|
| id | SERIAL PRIMARY KEY | Уникальный идентификатор комнаты |
| name | VARCHAR(255) NOT NULL | Название комнаты |
| owner_id | INTEGER | Владелец комнаты (ссылка на users.id) |

# 3 ОПИСАНИЕ ПРОТОКОЛА И ФОРМАТОВ ПЕРЕДАЧИ ДАННЫХ

В проекте используются два основных протокола передачи данных:

- HTTP (REST API) – для регистрации, аутентификации и управления чат-комнатами;
- WebSocket – для обмена сообщениями в режиме реального времени между пользователями в чатах.

## 3.1 HTTP

HTTP-протокол используется для взаимодействия между клиентом и сервером website. Все данные передаются в формате JSON.

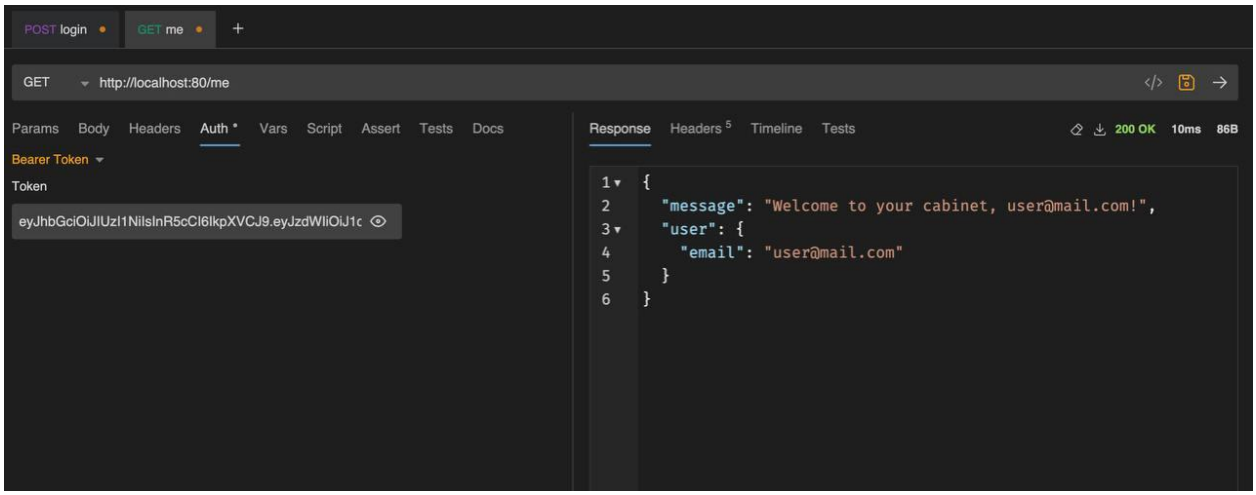



Рисунок 2 – Пример GET-запроса

## 3.2 WebSocket

WebSocket-протокол используется для мгновенного обмена сообщениями в чатах.

```
User email: user@mail.com
WebSocket connection established
Message received: {"type": "notification", "message": "user@mail.com has joined the chat"}
Message received: {"type": "message", "sender": "user@mail.com", "message": "Hello, world!"}
```

Рисунок 3 – Пример справочных данных (логи) из чата

### 3.3 Аутентификация и безопасность

- Вся аутентификация выполняется с помощью JWT-токенов;
- JWT передается в заголовке Authorization для REST API и в параметрах запроса для WebSocket;
- Доступ к чат-комнатам возможен только при наличии корректного токена.

# 4 ОПИСАНИЕ API

API реализовано в виде RESTful-сервиса с поддержкой аутентификации через JWT и передачей данных в формате JSON.

## 4.1 Эндпоинт me

Приветствуем пользователя, прошедшего аутентификацию (JWT).

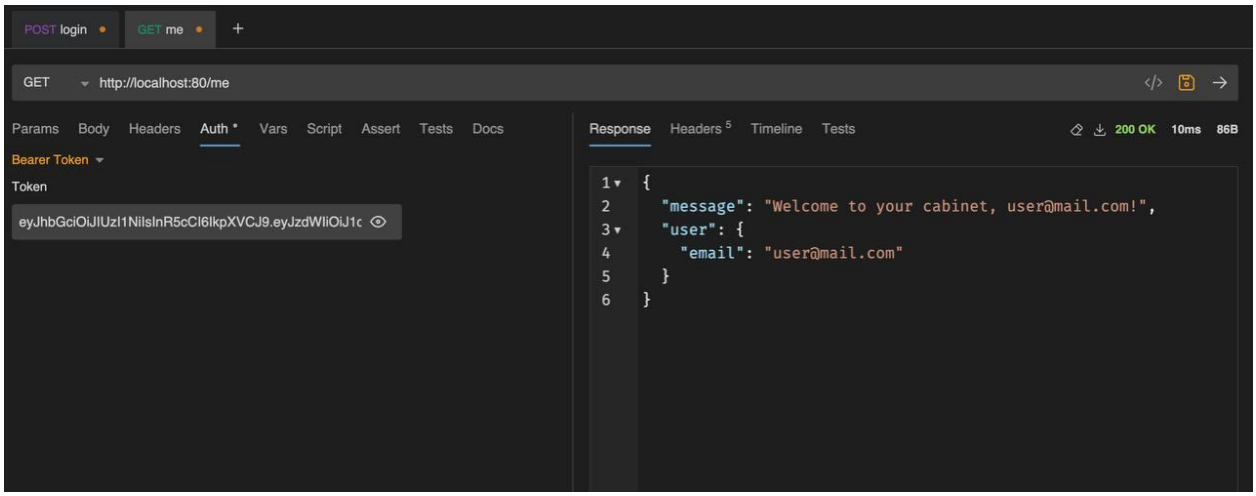



Рисунок 4 – Успешный GET-запрос эндпоинта me

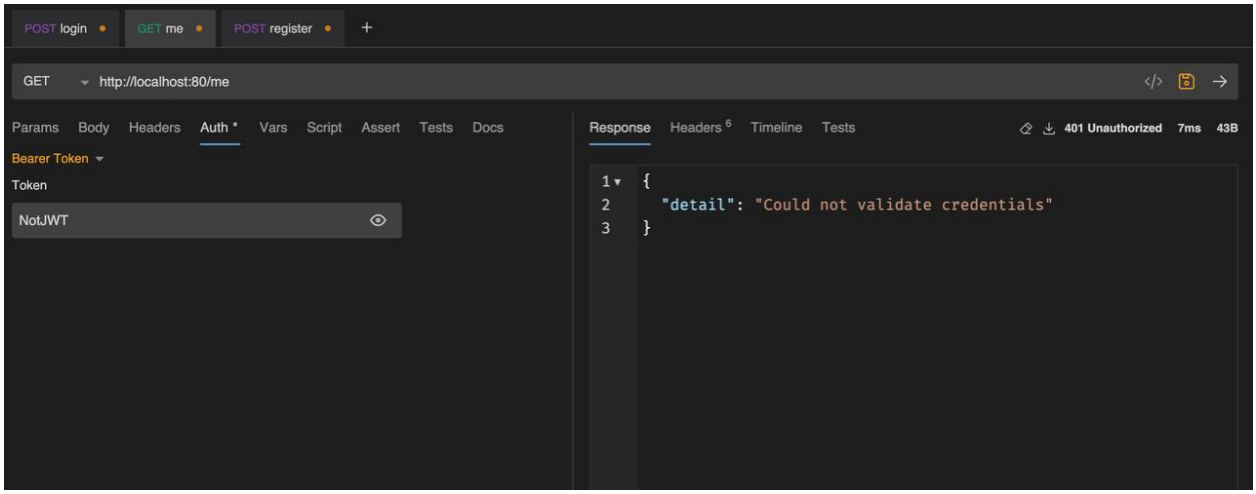



Рисунок 5 – Неуспешный GET-запрос эндпоинта me

### 4.2 Эндпоинт login

Пример успешной и неуспешной попытки входа в личный кабинет.

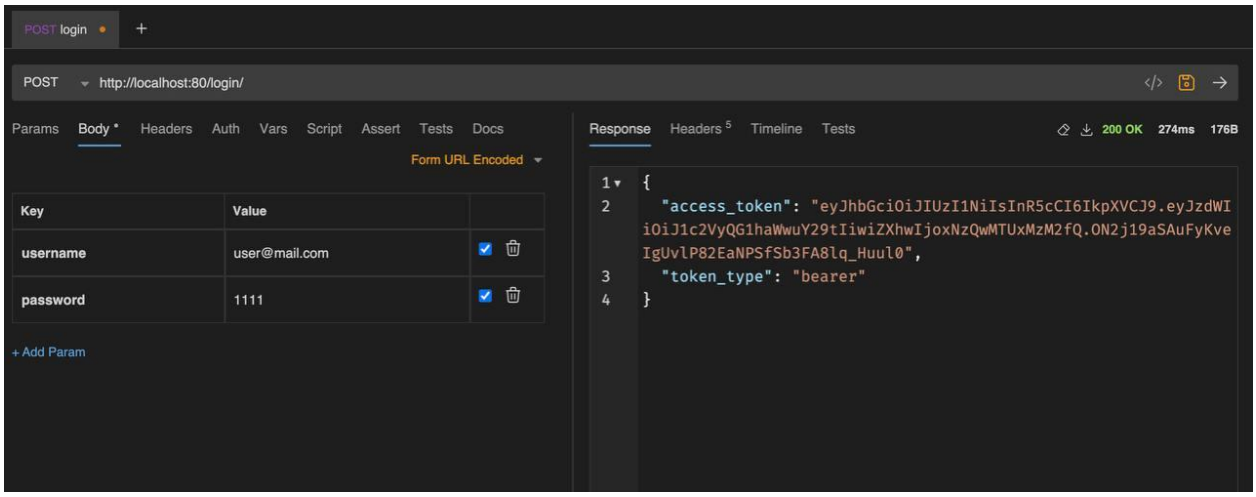



Рисунок 6 – Успешный POST-запрос эндпоинта login

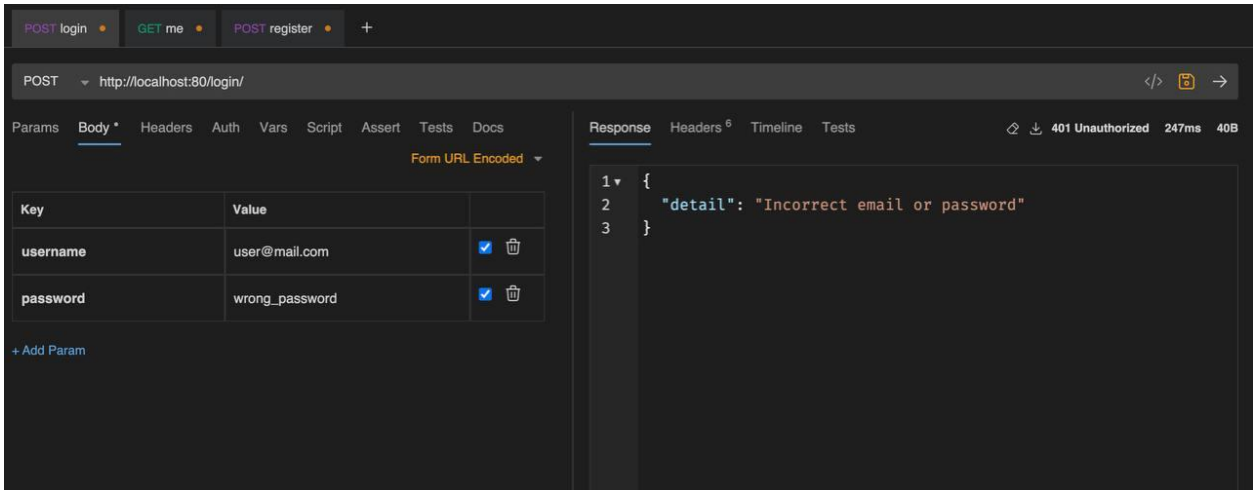



Рисунок 7 – Неуспешный POST-запрос эндпоинта login

### 4.3 Эндпоинт register

Пример успешной и неуспешной попытки регистрации.

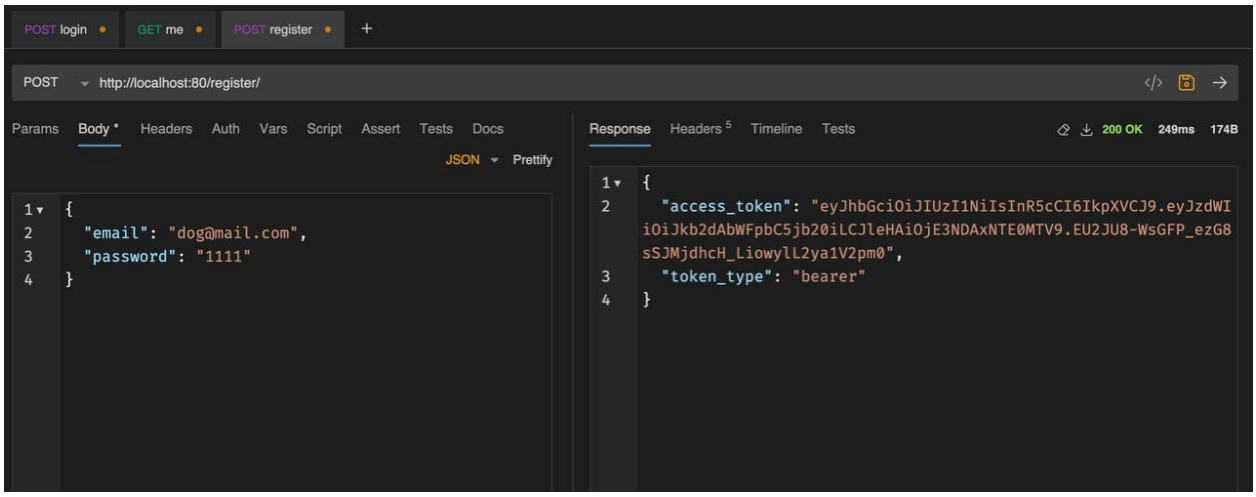



Рисунок 8 – Успешный POST-запрос эндпоинта register

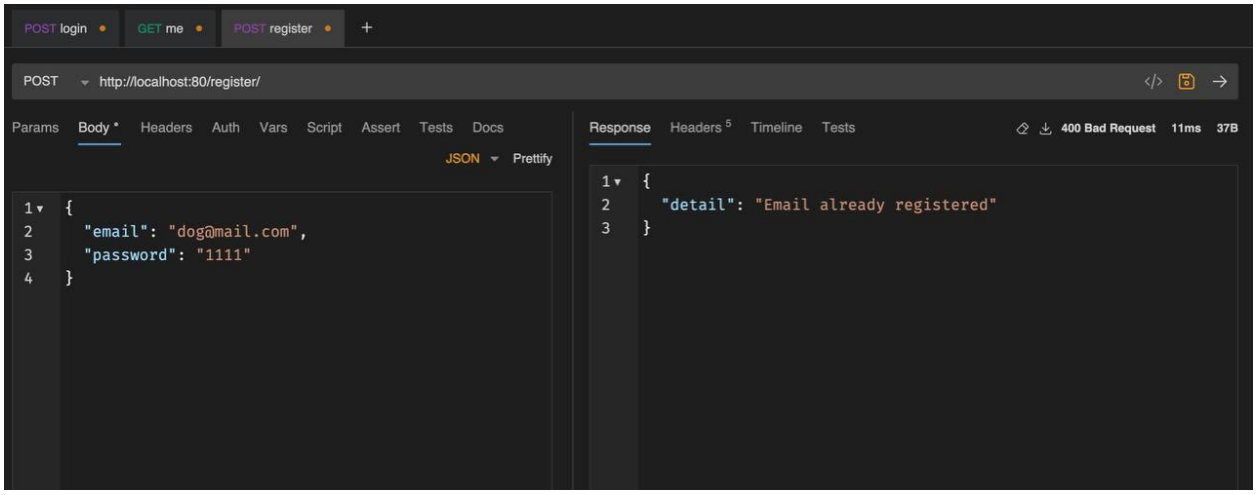



Рисунок 9 – Неуспешный (повторный) POST-запрос эндпоинта register

## 4.4 Эндпоинт chats

Пример успешной и неуспешной попытки получить список всех комнат.

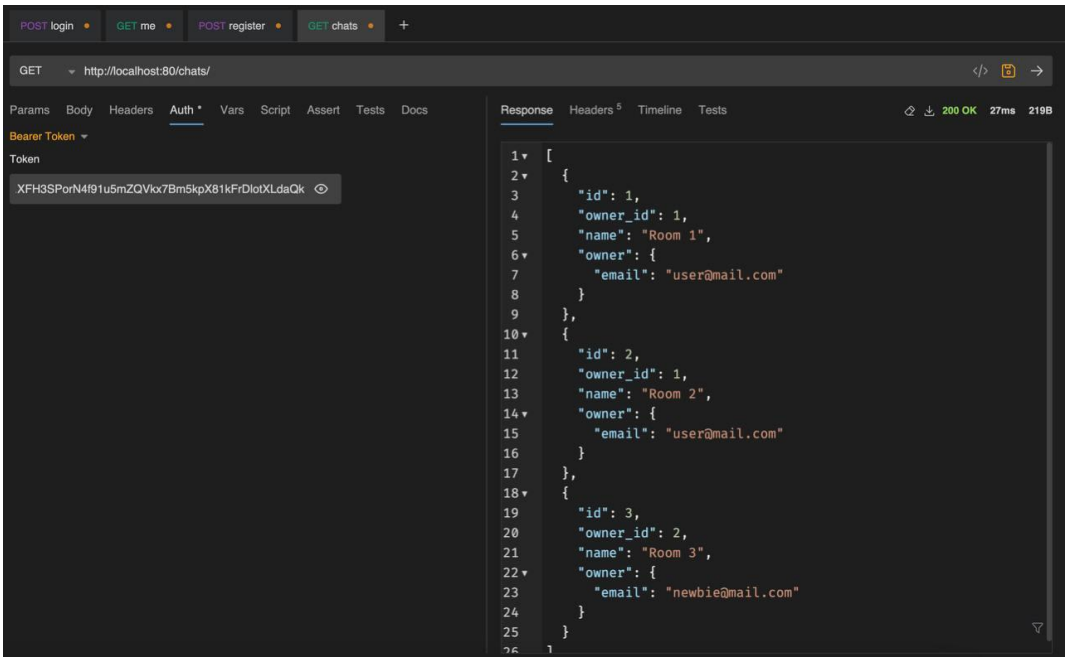



Рисунок 10 – Успешный GET-запрос эндпоинта chats

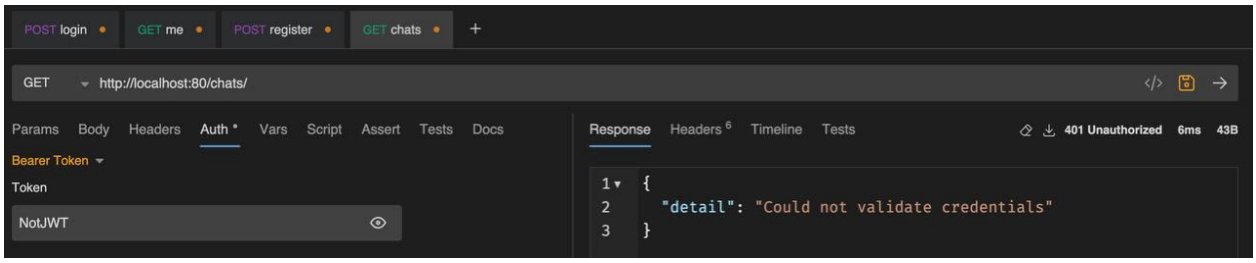



Рисунок 11 – Неуспешный GET-запрос эндпоинта chats

### 4.5 Эндпоинт create

Пример успешной и неуспешной попытки создания комнаты.

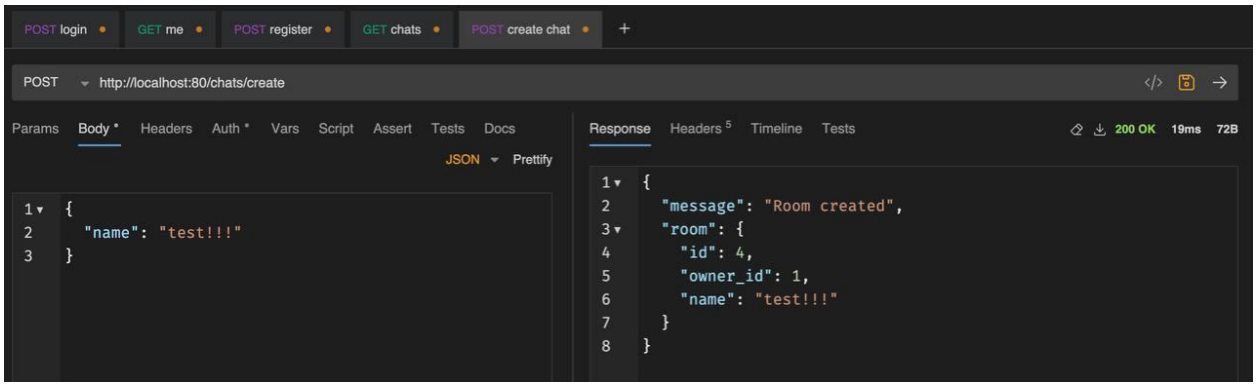



Рисунок 12 – Успешный POST-запрос эндпоинта create

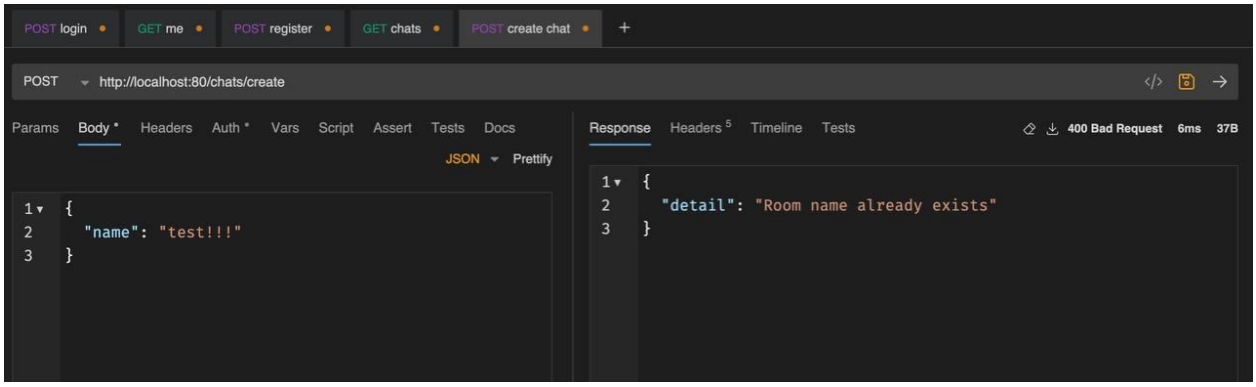



Рисунок 13 – Неуспешный (повторный) POST-запрос эндпоинта create

### 4.6 Эндпоинт delete

Пример успешной и неуспешной попытки удаления комнаты.

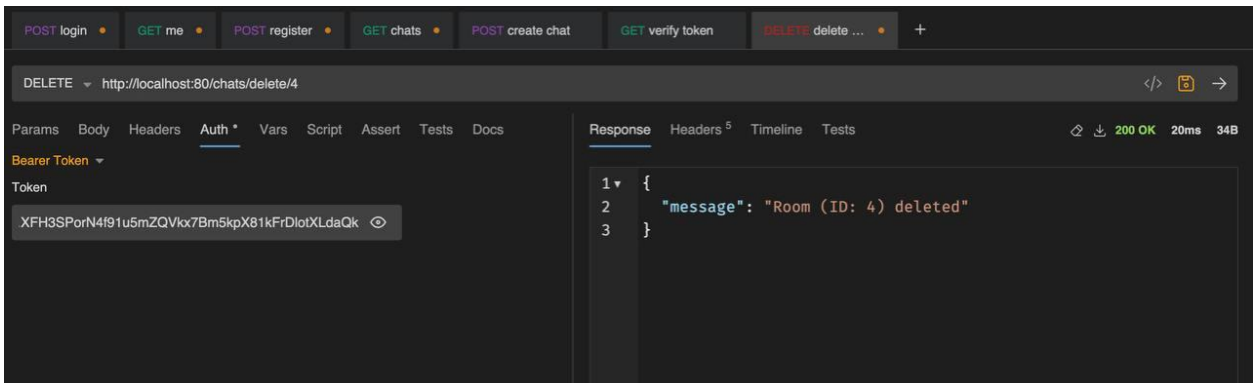



Рисунок 14 – Успешный DELETE-запрос эндпоинта delete

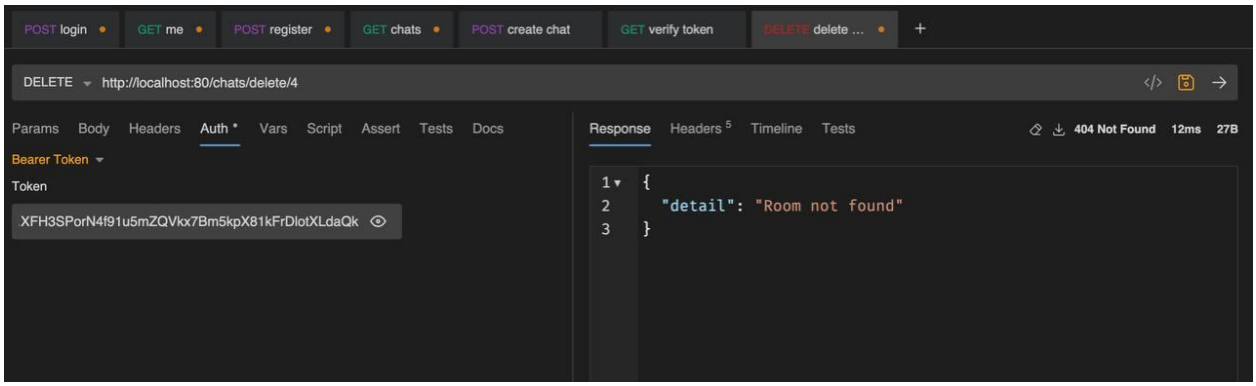



Рисунок 15 – Неуспешный (повторный) DELETE-запрос эндпоинта delete

### 4.7 Эндпоинт search

Пример успешной и неуспешной попытки поиска комнаты.

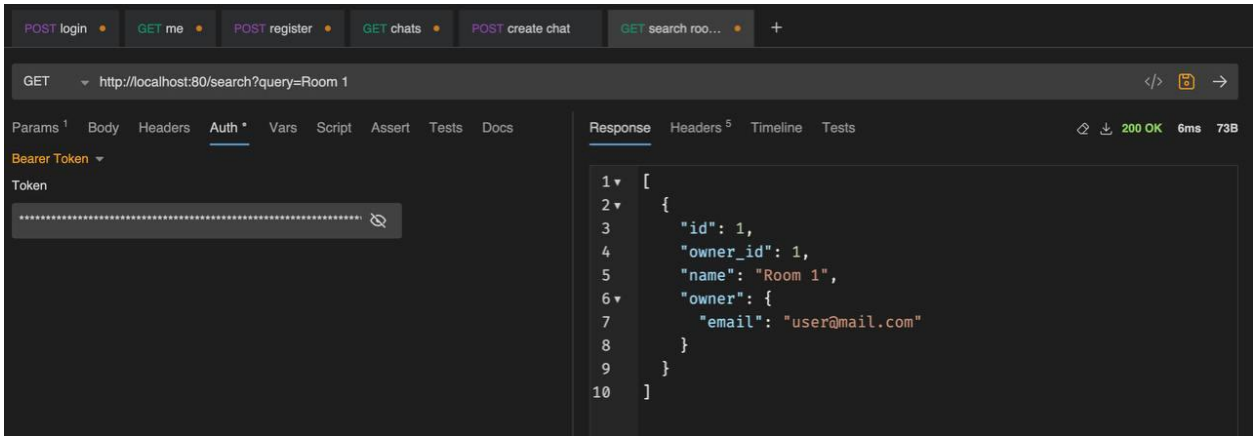
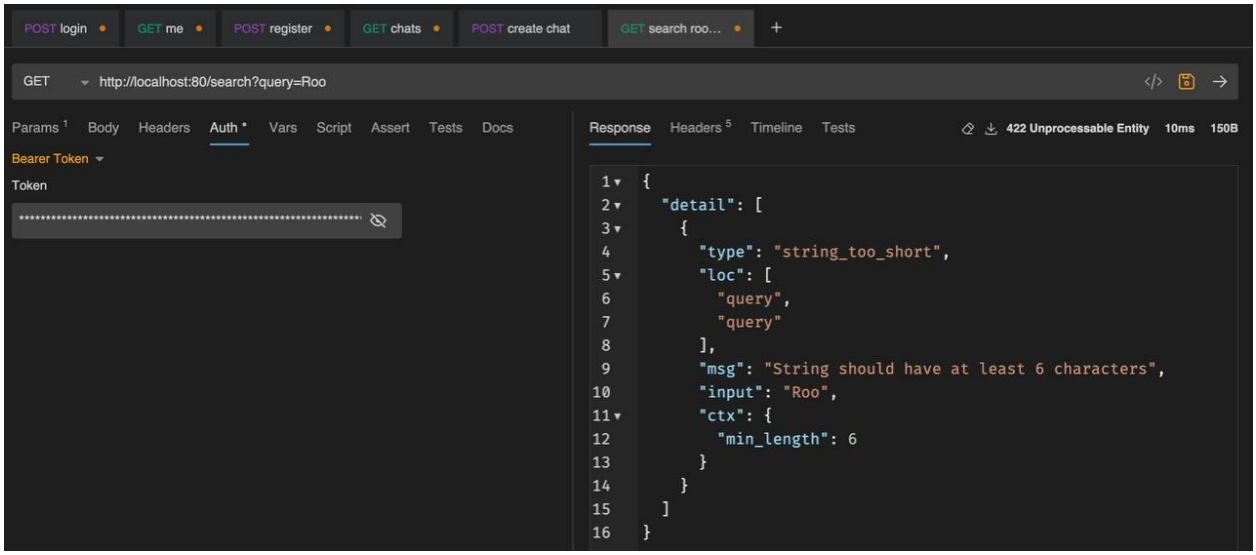



Рисунок 16 – Успешный GET-запрос эндпоинта search




Неуспешный GET-запрос эндпоинта search

# 5 ПРИМЕРЫ ФУНКЦИОНАЛА

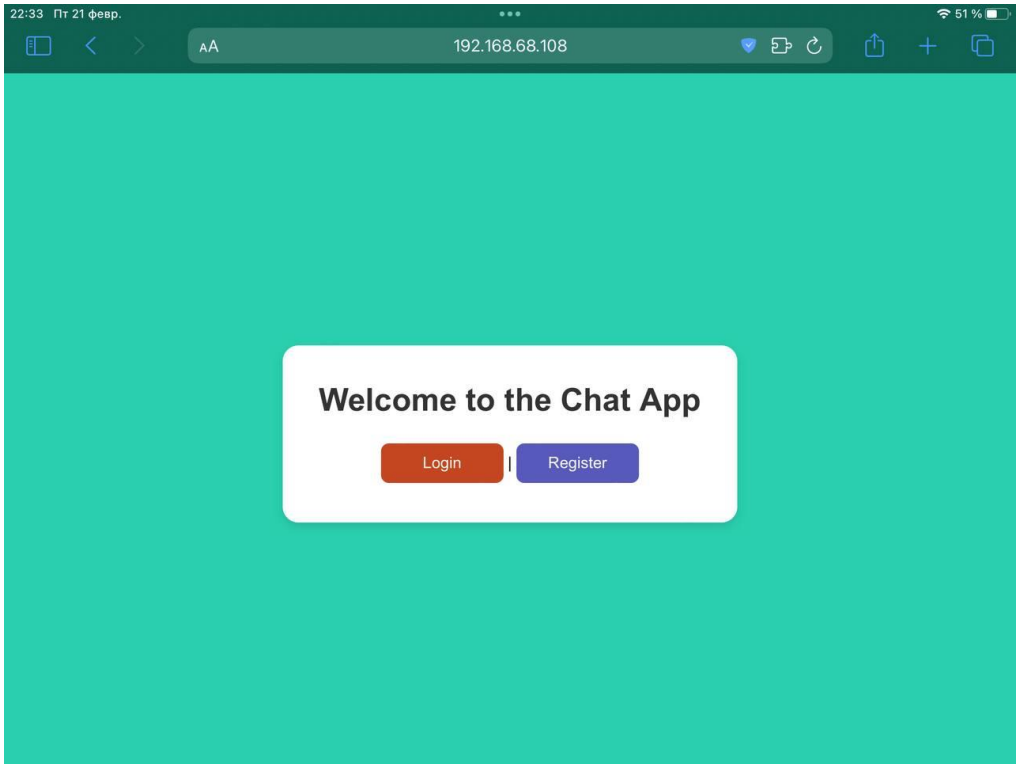



Рисунок 17 – Приветственная страница веб-приложения

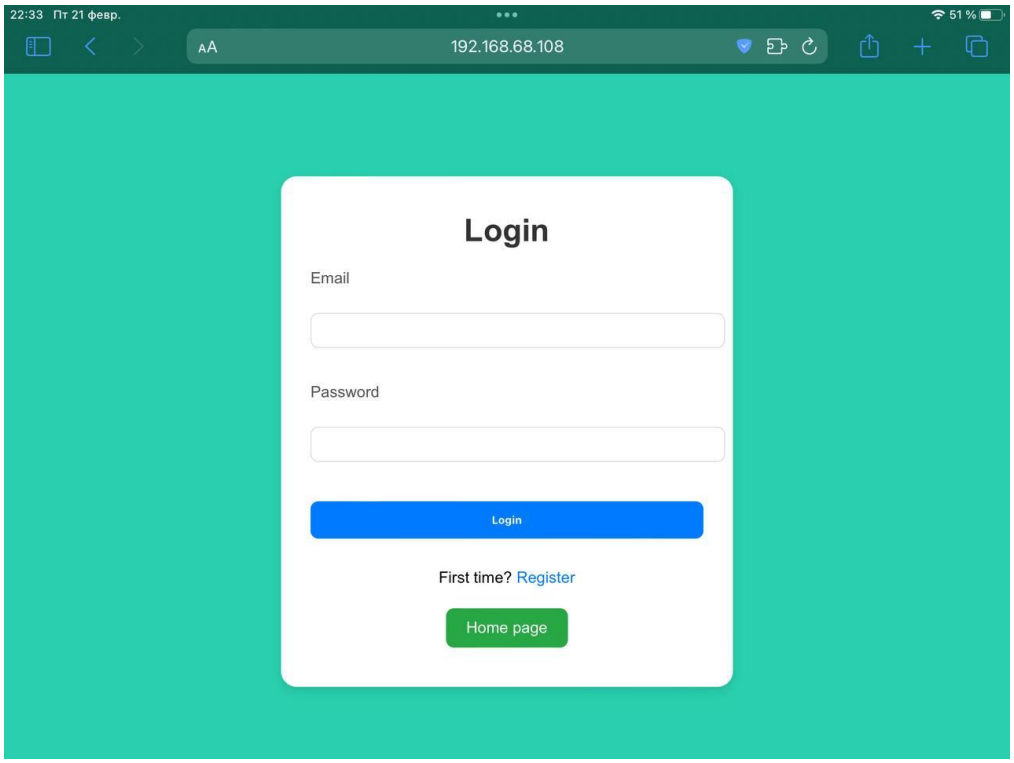



Рисунок 18 – Вход в аккаунт

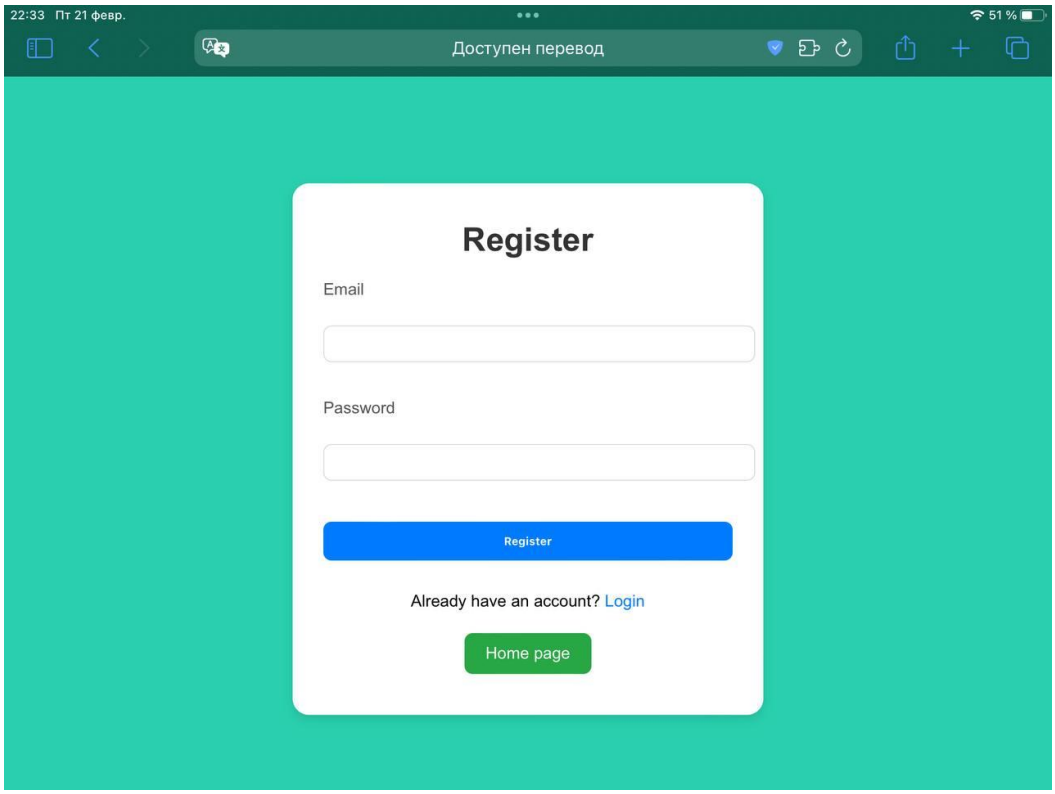



Рисунок 19 – Регистрация

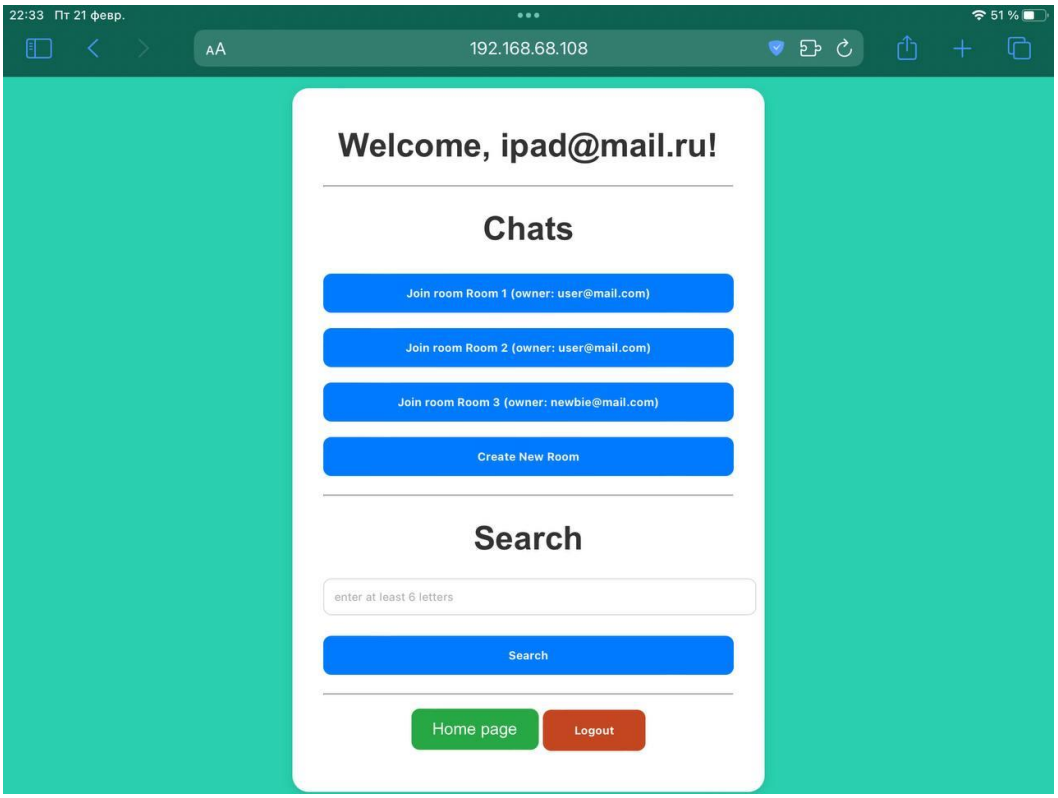



Рисунок 20 – Домашняя страница пользователя

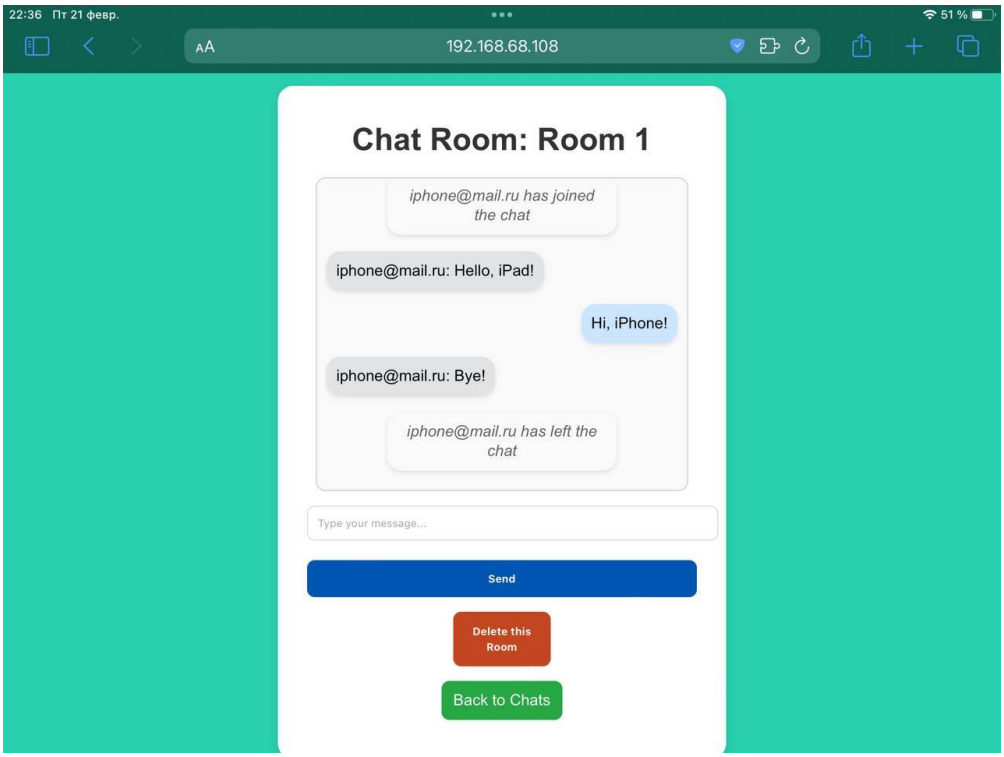



Рисунок 21 – Пример чата с другим пользователем

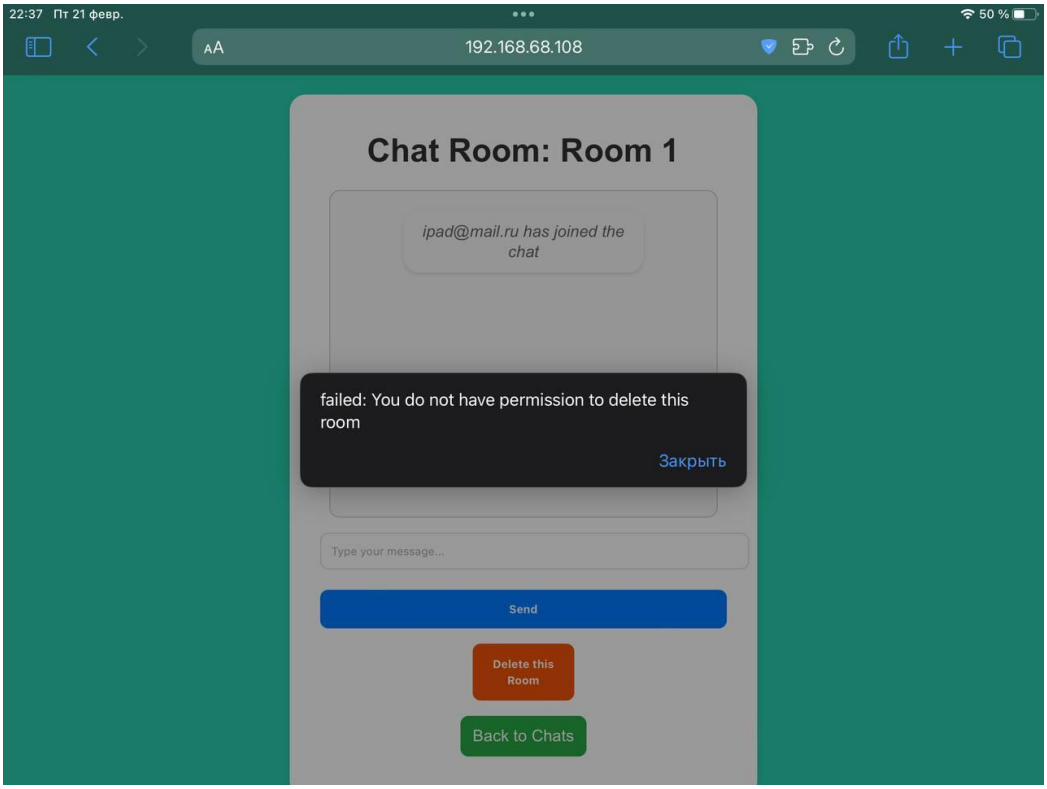



Рисунок 22 – Попытка удалить чат, которая не принадлежит пользователю

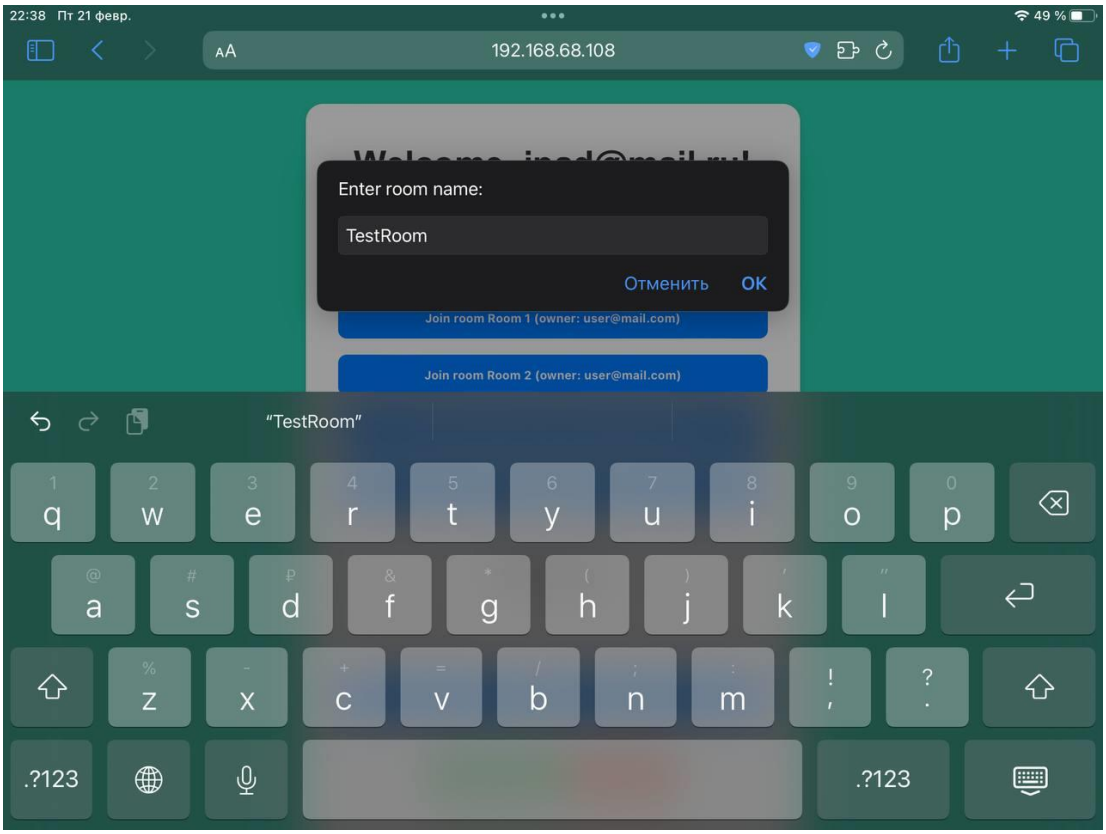



Рисунок 23 – Создание чата

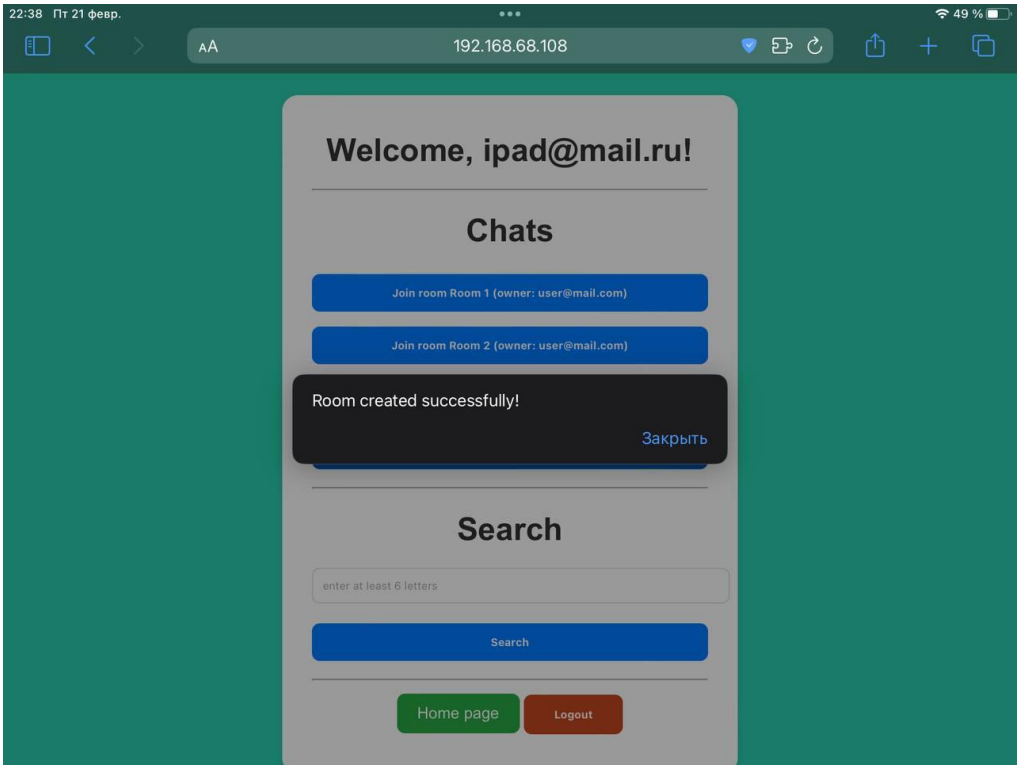



Рисунок 24 – Сообщение об успешном создании чата

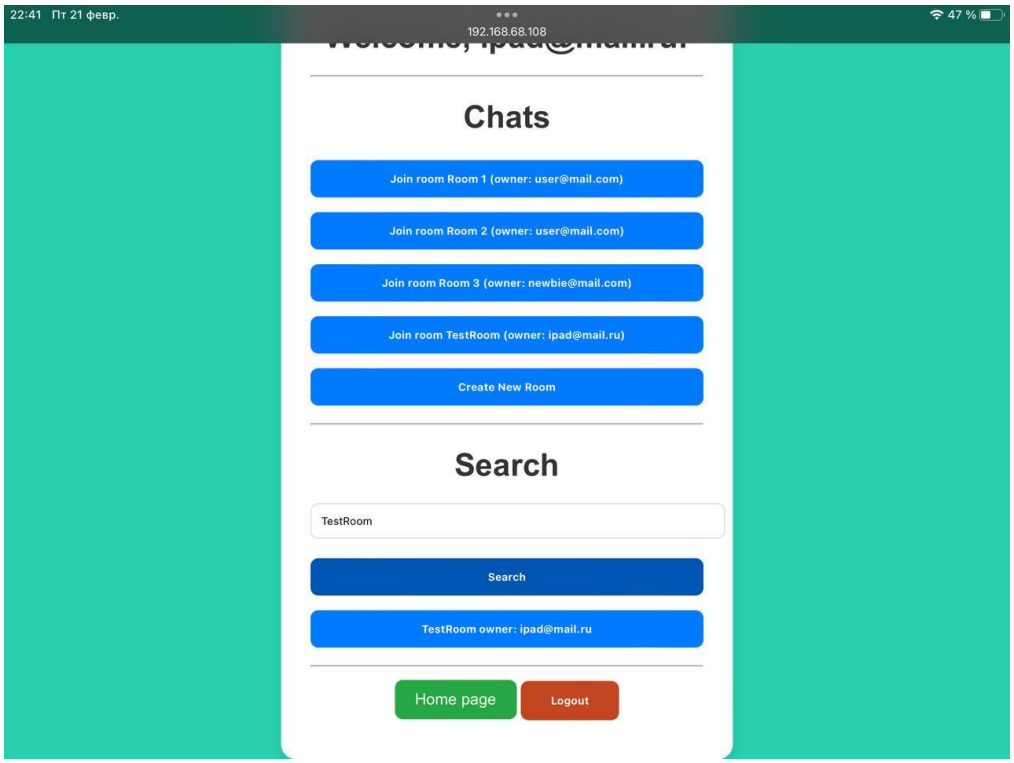



Рисунок 25 – Поиск чатов

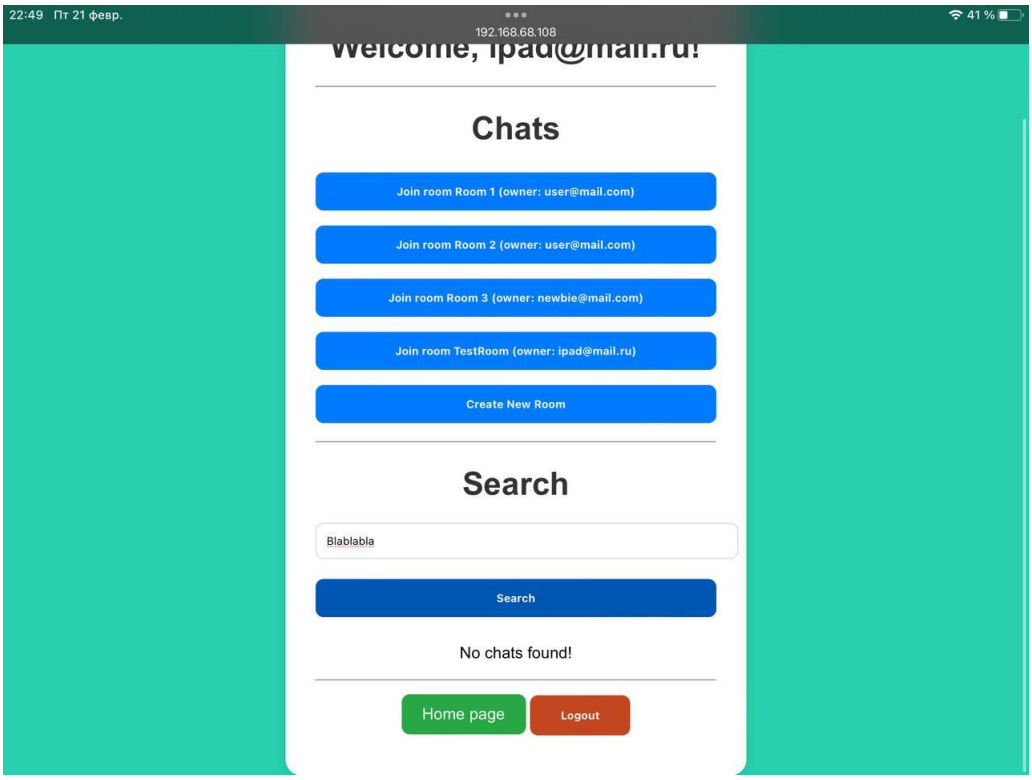



Рисунок 26 – Поиск несуществующих чатов

## ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были изучены основы создания простых веб-приложений с использованием фреймворка FastAPI. Были освоены технологии такие как JWT, WebSockets и механизмы обработки HTTP-запросов, работа с базами данных через SQLAlchemy ORM, а также создание маршрутов и шаблонов с Jinja2.

## СПИСОК ИСТОЧНИКОВ


1. Документация FastAPI [Электронный ресурс]. – URL: https://fastapi.tiangolo.com/ (Дата обращения: 25.01.2025).
2. FastAPI WebSockets [Электронный ресурс]. – URL: https://fastapi.tiangolo.com/advanced/websockets/ (Дата обращения: 25.01.2025).
3. Документация Jinja [Электронный ресурс]. – URL: https://jinja.palletsprojects.com/en/stable/ (Дата обращения: 25.01.2025).
4. Примеры и шаблоны с GitHub [Электронный ресурс]. – URL: https://github.com/cs-itmo/webdev_2024/tree/master/lessons (Дата обращения: 25.01.2025).

## ПРИЛОЖЕНИЕ А

Листинг 4 – Программный код main.py (микросервис chat)

```
from fastapi import FastAPI, WebSocket, WebSocketDisconnect, Request,
HTTPException, status, Depends, Query
from fastapi.responses import HTMLResponse
from fastapi.middleware.cors import CORSMiddleware
from fastapi.security import OAuth2PasswordBearer

from typing import List
from fastapi.templating import Jinja2Templates
from utils import get_user_from_token
import logging
import json

app = FastAPI()
templates = Jinja2Templates(directory="templates")
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

class ConnectionManager:
    def __init__(self):
        self.active_connections: List[WebSocket] = []

    async def connect(self, websocket: WebSocket):
        await websocket.accept()
        self.active_connections.append(websocket)

    def disconnect(self, websocket: WebSocket):
        self.active_connections.remove(websocket)

    async def send_personal_message(self, message: str, websocket:
WebSocket):
        await websocket.send_text(message)

    async def broadcast(self, message: str):
        for connection in self.active_connections:
            await connection.send_text(message)

manager = ConnectionManager()


# @app.get("/")
# async def get(request: Request):
#     return templates.TemplateResponse(request=request, name="test.html")

@app.websocket("/ws/{room_id}")
async def websocket_endpoint(websocket: WebSocket, room_id: int):
    token = websocket.query_params.get("token")
    user_email = get_user_from_token(token)
    logger.info(f"Token received: {token}")
    logger.info(f"email received: {user_email}")
    await manager.connect(websocket)
```

```
    # await manager.broadcast(f"{user_email} has joined the chat")
    await manager.broadcast(json.dumps({
        "type": "notification",
        "message": f"{user_email} has joined the chat"
    }))
    try:
        while True:
            data = await websocket.receive_text()
            # await manager.send_personal_message(f"You wrote: {data}",
websocket)
            # await manager.broadcast(f"{user_email}: {data}")
            await manager.broadcast(json.dumps({
                "type": "message",
                "sender": user_email,
                "message": data
            }))
    except WebSocketDisconnect:
        manager.disconnect(websocket)
        logger.info(f"{user_email} disconnected")
        # await manager.broadcast(f"{user_email} has left the chat")
        await manager.broadcast(json.dumps({
            "type": "notification",
            "message": f"{user_email} has left the chat"
        }))
```

## ПРИЛОЖЕНИЕ Б

Листинг 5 – Программный код main.py (микросервис website)

```
from fastapi import FastAPI, Depends, HTTPException, status, Request, Query
from fastapi.security import OAuth2PasswordRequestForm
from fastapi.responses import HTMLResponse, RedirectResponse
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
from sqlalchemy.orm import Session, joinedload
import logging

from database import SessionLocal, engine, Base, get_db
from schemas import UserCreate, Token, UserResponse, RoomCreate, RoomResponse
from utils import verify_password, get_password_hash
from auth import create_access_token, get_current_user
from models import User, Room
from typing import List

# create app
app = FastAPI()
app.mount("/static", StaticFiles(directory="static"), name="static")
templates = Jinja2Templates(directory="templates")

# create database tables
Base.metadata.create_all(bind=engine)

# create logger
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# ==================================== LOGIN LOGIC
====================================
@app.get("/login", response_class=HTMLResponse)
async def login_page(request: Request):
    return templates.TemplateResponse(request=request, name="login.html",
context={"request": request})

@app.post("/login/", response_model=Token)
def login(form_data: OAuth2PasswordRequestForm = Depends(), db: Session =
Depends(get_db)):
    user = db.query(User).filter(User.email == form_data.username).first()

    if not user or not verify_password(form_data.password,
user.hashed_password):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect email or password",
            headers={"WWW-Authenticate": "Bearer"},
        )

    # generate JWT token for authentication
    access_token = create_access_token(data={"sub": user.email})
    return {"access_token": access_token, "token_type": "bearer"}

# ==================================== REGISTRATION LOGIC
====================================
@app.get("/register", response_class=HTMLResponse)
async def register_page(request: Request):
    return templates.TemplateResponse(request=request, name="register.html",
context={"request": request})

@app.post("/register/", response_model=Token)
```

```
def register(user: UserCreate, db: Session = Depends(get_db)):
    # check if user already exists
    db_user = db.query(User).filter(User.email == user.email).first()
    if db_user:
        raise HTTPException(status_code=400, detail="Email already
registered")

    # hash the password and create the user
    hashed_password = get_password_hash(user.password)
    new_user = User(email=user.email, hashed_password=hashed_password)
    db.add(new_user)
    db.commit()
    db.refresh(new_user)

    # generate JWT token
    access_token = create_access_token(data={"sub": new_user.email})
    return {"access_token": access_token, "token_type": "bearer"}

# ==================================== CHAT LOGIC
====================================
@app.get("/chats", response_class=HTMLResponse)
def chats_page(request: Request):
    return templates.TemplateResponse(request=request, name="chats.html",
context={"request": request})

@app.get("/chats/", response_model=List[RoomResponse])
def get_chats(current_user: User = Depends(get_current_user), db: Session =
Depends(get_db)):
    # to show only owner's rooms
    # chats =
db.query(Room).options(joinedload(Room.owner)).filter(Room.owner_id ==
current_user.id).all()

    # to show only all rooms
    chats = db.query(Room).options(joinedload(Room.owner)).all()
    return chats

@app.get("/chat/{room_id}", response_class=HTMLResponse)
async def chat_page(request: Request, room_id: int, db: Session =
Depends(get_db)):
    room = db.query(Room).filter(Room.id == room_id).first()
    if not room:
        raise HTTPException(status_code=404, detail="Room not found")
    return templates.TemplateResponse("chat.html", {"request": request,
"room": room})

@app.post("/chats/create")
def create_chat_room(room: RoomCreate, db: Session = Depends(get_db),
current_user: User = Depends(get_current_user)):
    # check if room name already exists
    db_room = db.query(Room).filter(Room.name == room.name, Room.owner_id ==
current_user.id).first()
    if db_room:
        raise HTTPException(status_code=400, detail="Room name already
exists")

    new_room = Room(name=room.name, owner_id=current_user.id)
    db.add(new_room)
    db.commit()
    db.refresh(new_room)
    return {"message": "Room created", "room": new_room}

@app.delete("/chats/delete/{room_id}")
```

```
def delete_chat_room(room_id: int, db: Session = Depends(get_db),
current_user: User = Depends(get_current_user)):
    # id and owner_id
    # room = db.query(Room).filter(Room.id == room_id, Room.owner_id ==
current_user.id).first()

    # id and THEN owner_id
    room = db.query(Room).filter(Room.id == room_id).first()
    if not room:
        raise HTTPException(status_code=404, detail="Room not found")
    if room.owner_id != current_user.id:
        raise HTTPException(status_code=403, detail="You do not have
permission to delete this room")

    # delete room from database
    db.delete(room)
    db.commit()
    return {"message": f"Room (ID: {room_id}) deleted"}

@app.get("/search", response_model=List[RoomResponse])
def search_chats(
    query: str = Query(..., min_length=6, description="search query (minimum
6 letters)"),
    db: Session = Depends(get_db),
    current_user: User = Depends(get_current_user)
):
    rooms = db.query(Room).filter(Room.name.ilike(f"%{query}%")).all()
    # rooms = db.query(Room).filter(Room.name.contains(query))
    if not rooms:
        raise HTTPException(status_code=404, detail="No chats found")
    return rooms

# ==================================== ANOTHER LOGIC
====================================

@app.get("/home", response_class=HTMLResponse)
async def read_root(request: Request):
    # return {"message": "Welcome to the FastAPI Auth Demo"}
    return templates.TemplateResponse(request=request, name="home.html",
context={"request": request})

@app.get("/")
def redirect_to_home():
    # redirect
    return RedirectResponse(url="/home", status_code=301)

# Protected route to check JWT token validity
@app.get("/me", response_model=UserResponse)
def access_cabinet(current_user: User = Depends(get_current_user)):
    return {"message": f"Welcome to your cabinet, {current_user.email}!",
"user": current_user}
```