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

**Факультет безопасности информационных технологий**

**Дисциплина:**

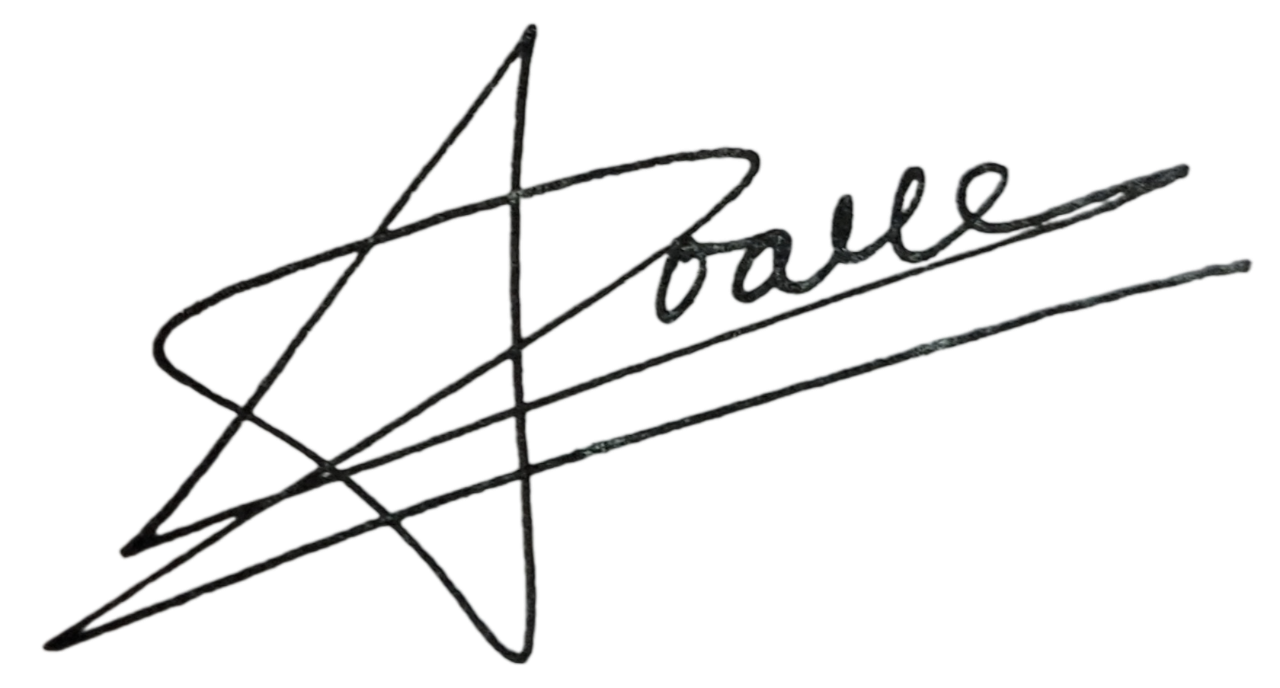
**«Web программирование»**

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №1**

«Разработка Веб сервиса»

**Выполнили:**

Чу Ван Доан, студент группы N3347



*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

(подпись)

**Проверил:**

Менщиков Александр Алексеевич

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

(отметка о выполнении)

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

(подпись)

# Содержание

[1. Задание 3](#_heading=h.puhy4onlavyt)

[2. Ход работы 6](#_heading=h.di7u4ulvin57)

[2.1. Описание архитектуры ПО 6](#_heading=h.ulbutra9hjmj)

[2.2. Описание структуры базы данных. 10](#_heading=h.7trcqc8wjpi)

[2.3. Описание протокола и форматов передачи данных 11](#_heading=h.fuxh91gdwafs)

[2.4. Описание API 13](#_heading=h.f7th913ikioe)

[2.5. Скриншоты функционала (каждая страница и кейс из технического задания). 16](#_heading=h.yalbnljopw2)

[2.6. Коды 22](#_heading=h.1ar7z5y5ddkr)

[Заключение 38](#_heading=h.tyjcwt)

[Список использованных источников 39](#_heading=h.3dy6vkm)

# Введение

Цель работы – Необходимо разработать онлайн чат для пользователей, состоящий из двух микросервисов. Первый на Python FastAPI будет отвечать за регистрацию, аутентификацию, создание и поиск чат-комнат. Второй на Python FastAPI будет отвечать за функционал Websocket чата. Хранение данных должно быть реализовано в PostgreSQL либо в MongoDB. Аутентификация в чате должна происходить на основе JWT токенов. Фронтенд допустимо реализовать на HTML с использованием Jinja2 шаблонов.

## Задание

Сценарий использования

Пользователь заходит на главную страницу сайта и видит приглашение к регистрации либо аутентификации. Пользователь может зарегистрироваться или войти, введя email и пароль. Авторизованный пользователь может создавать новые чат-комнаты (название), видеть список своих созданных комнат и удалять свои комнаты. Авторизованный пользователь видит окно поиска чатов по названию. Можно ввести любой текст от 6 символов и получить список всех чатов, содержащих данное слово.

По клику на название чата (из своего списка, либо из результатов поиска) пользователь переходит на страницу чата, содержащую сообщения (историю поддерживать не требуется) и окно для ввода нового сообщения. Каждое сообщение, которое вводит любой из пользователей в данном чате, сразу отображается у всех подключенных в данный момент пользователей (у которых в браузере открыта страница данного чата). Когда пользователь только открывает страницу. У всех отображается информация о том, что новый пользователь подключился.

Техническое задание

Разработать и настроить следующие компоненты:

* Микросервис **website** – функционал управления пользователями, комнатами;
* Микросервис **chat** – функционал Websocket чатов;
* База данных;
* Веб-сервер.

Требования в сервису website

* Должен быть разработан на языке Python с использованием фреймворка FastAPI;
* Фронтенд должен быть реализован с использованием Jinja2 шаблонов;
* Хранение и извлечение данных должно быть в БД PostgreSQL либо MongoDB по выбору;
* Доступ к БД только с использованием ORM библиотек;
* Фронтенд может быть минималистичным, но должен быть аккуратным;
* Должен использоваться механизм сессий;
* Страница с чатом должна быть реализована на стороне фронтенда с использованием API микросервиса **chat**;
* Взаимодействие с **chat** должно происходить с помощью JavaScript;
* Аутентификация в **chat** решается посредством генерации подписанного JWT токена, содержащего (email пользователя и название чата) и передачи его в **chat** в HTTP заголовке;
* Должен выделяться JWT токен только для существующих чатов и существующих пользователей.

Требования к сервису chat

* Должен быть разработан на языке Python;
* Должен использовать Websocket протокол;
* Не должен хранить никакой информации кроме открытых комнат и подключенных к ним пользователей;
* Аутентификация пользователей должна выполняться через проверку JWT токена;
* Должен иметь метод приема сообщения и поддерживать рассылку сообщения по всем пользователям, подключенным к данной чат комнате.

Требования к базе данных

* Должна быть выбрана БД PostgreSQL или MongoDB;
* Должен быть настроен доступ к БД по логину и паролю;
* Должна быть продумана и описана структура БД в отчете.

Требования к веб серверу

* Должен использоваться веб сервер Nginx;
* Должен быть настроен Proxy pass на оба микросервиса: HTTP для **website** и Websocket для **chat** по пути /ws;
* Должно быть настроено логирование запросов.

Общие требования

* Весь проект должен быть упакован с использованием Docker.
* Должна поддерживаться следующая структура директории:
* В корне расположен docker-compose.yml и README.md;
* Каждый микросервис, его код и Dockerfile расположены в своей директории (website, chat, db, nginx).
* Наружу должен быть доступен только 1 HTTP порт;
* База данных должна обеспечивать persistence в локальную директорию dump;
* Должны быть настроены файлы .gitignore и .dockerignore.

## Ход работы

### Описание архитектуры ПО

project-root/

├── .gitignore

├── .dockerignore

├── docker-compose.yml

│

├── website/

│ ├── app/

│ │ ├── main.py

│ │ ├── database.py

│ │ ├── models.py

│ │ ├── schemas.py

│ │ ├── routers/

│ │ │ ├── auth.py

│ │ │ ├── chat.py

│ │ │ └── websocket\_handler.py

│ │ ├── templates/

│ │ │ ├── index.html

│ │ │ ├── chat.html

│ │ │ ├── login.html

│ │ │ ├── register.html

│ │ │ └── room\_chat.html

│ │ └── static/

│ │ └── favicon.ico

│ ├── Dockerfile

│ ├── requirements.txt

│ └── .gitignore

│

├── chat/

│ ├── app/

│ │ ├── main.py

│ │ └── websocket\_handler.py

│ ├── Dockerfile

│ └── requirements.txt

│

├── db/

│ ├── dump/

│ ├── Dockerfile

│ └── init.sql

│

└── nginx/

├── nginx.conf

└── Dockerfile

**Root-Level Files**

* **.gitignore**: Содержит список файлов и директорий, которые не должны отправляться в Git, например, файлы конфигурации среды .env, временные файлы и папки.
* **.dockerignore**: Исключает ненужные файлы и директории из процесса сборки Docker-образа, такие как .git, журналы, кэш и локальные конфигурации.
* **docker-compose.yml**: Конфигурационный файл Docker Compose для управления всеми сервисами проекта, включая website, chat, db (PostgreSQL) и nginx. Определяет контейнеры, тома, сеть и необходимые переменные окружения.

**website - Микросервис для управления пользователями и чат-комнатами**

* **website/Dockerfile**: Описывает процесс сборки Docker-образа для микросервиса website, устанавливает необходимые библиотеки из requirements.txt и запускает приложение.
* **website/requirements.txt**: Содержит список зависимостей для website, таких как FastAPI, SQLAlchemy, Jinja2 и psycopg2.
* **website/app**: Основная папка с исходным кодом приложения website.
  + **main.py**: Точка входа для приложения FastAPI, подключает маршруты и настраивает приложение.
  + **database.py**: Настройка соединения с базой данных PostgreSQL через SQLAlchemy, содержит функцию для создания сессии.
  + **models.py**: Определяет модели SQLAlchemy для таблиц, таких как User и ChatRoom.
  + **schemas.py**: Определяет схемы Pydantic для валидации входных и выходных данных, например UserCreate и ChatRoomCreate.
  + **routers**: Папка с маршрутами FastAPI для website.
    - **auth.py**: Управление регистрацией и аутентификацией пользователей, создание и проверка JWT-токенов для сессий.
    - **chat.py**: Управление чат-комнатами: создание, поиск и удаление комнат.
    - **websocket\_handler.py**: Обработка соединений WebSocket, управление подключениями и отправка сообщений участникам комнат.
  + **templates**: HTML-шаблоны, рендеринг которых происходит с использованием Jinja2.
    - **index.html**: Шаблон для главной страницы website.
    - **chat.html**: Шаблон для отображения списка комнат и интерфейса для входа в чат.
    - **login.html**: Шаблон для страницы авторизации.
    - **register.html**: Шаблон для страницы регистрации.
    - **room\_chat.html**: Шаблон для интерфейса конкретной комнаты чата, где пользователи могут отправлять и просматривать сообщения.
  + **static**: Содержит статические файлы, такие как иконка сайта (favicon.ico).

**chat - Микросервис WebSocket для чатов**

* **chat/Dockerfile**: Описывает процесс сборки Docker-образа для микросервиса chat, устанавливает необходимые библиотеки из requirements.txt и запускает приложение.
* **chat/requirements.txt**: Список зависимостей для микросервиса chat, включая FastAPI и библиотеки для поддержки WebSocket.
* **chat/app**: Основная папка с исходным кодом приложения chat.
  + **main.py**: Точка входа для WebSocket-сервера. Настраивает конечные точки и обрабатывает соединения WebSocket.
  + **websocket\_handler.py**: Управление событиями WebSocket, такими как подключение пользователей, обработка отправки/получения сообщений и отключение.

db - Настройки для базы данных PostgreSQL

* **db/Dockerfile**: Описывает процесс сборки Docker-образа для сервиса PostgreSQL, включает файлы конфигурации и инициализацию данных.
* **db/dump**: Папка для хранения резервных копий базы данных, позволяющая сохранить данные на постоянном диске.
* **db/init.sql**: Инициализационный SQL-файл, содержащий инструкции для создания таблиц User и ChatRoom в базе данных PostgreSQL, а также может содержать тестовые данные.

nginx - Конфигурация для Nginx в качестве reverse proxy

* **nginx/nginx.conf**: Конфигурационный файл Nginx, который настраивает reverse proxy для микросервисов website и chat. Nginx перенаправляет HTTP-запросы на website, а запросы WebSocket – на chat.
* **nginx/Dockerfile**: Описывает процесс сборки Docker-образа для Nginx, копирует конфигурационный файл nginx.conf в контейнер.

### Описание структуры базы данных.

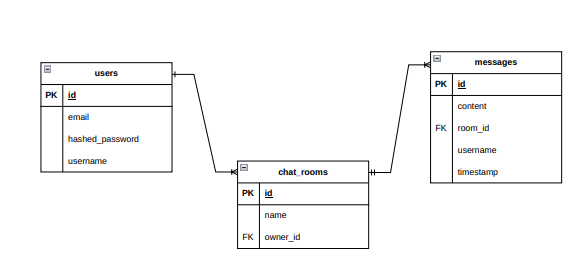


Рисунок 1: Диаграмма отношений базы данных

Обозначения

* **PK** (Primary Key): первичный ключ. Уникальный идентификатор записи в таблице.
* **FK** (Foreign Key): внешний ключ. Указывает на связь с другим полем (полем id другой таблицы).
* **UNIQUE**: уникальное значение, которое нельзя повторить в других записях таблицы.

Пояснение связей:

* **Users → ChatRooms**:
  + Поле owner\_id в таблице ChatRooms связано с полем id в таблице Users.
  + Это означает, что каждый чат создается конкретным пользователем, который является его владельцем.
  + Связь типа "один ко многим" (1 - N), поскольку один пользователь может быть владельцем нескольких чатов.
* **ChatRooms → Messages**:
  + Поле room\_id в таблице Messages связано с полем id в таблице ChatRooms.
  + Это означает, что сообщения привязываются к определенной комнате.
  + Связь также типа "один ко многим" (1 - N), поскольку одна комната может содержать множество сообщений.

### Описание протокола и форматов передачи данных

В данном проекте используется **протокол HTTP** для обычных запросов к REST API (регистрация, авторизация, создание и удаление комнат) и **протокол WebSocket** для обмена сообщениями в реальном времени в чат-комнатах. Формат передачи данных — JSON.

#### HTTP-протокол (REST API)

* + - 1. **Регистрация и Авторизация**
* Метод: POST
* Эндпоинт:
* /register — регистрация нового пользователя.
* /login — авторизация пользователя.
* Формат данных:

На POST /register и POST /login запросы отправляются в формате application/x-www-form-urlencoded с параметрами email, password и username (для регистрации).

* Ответ:
* В случае успешной регистрации или авторизации возвращается статус 303 See Other с перенаправлением на страницу входа или главную страницу.
* В случае ошибки возвращается статус 400 Bad Request с соответствующим сообщением.
  + - 1. **Управление чат-комнатами**

Методы:

* POST /create-room — создание новой комнаты.
* POST /delete-room — удаление существующей комнаты.
* GET /search-rooms — поиск комнат по ключевому слову.
* GET /chat — получение списка всех комнат пользователя.

Формат данных:

* POST /create-room и POST /delete-room принимают данные в формате application/x-www-form-urlencoded с полем room\_name или room\_id.
* GET /search-rooms принимает параметр search\_query в URL-строке для поиска по имени.

Ответ:

* Возвращаются HTML-страницы, сгенерированные с помощью Jinja2, содержащие список комнат и данные пользователя.
* При ошибках возвращается 404 Not Found или 400 Bad Request с описанием проблемы.

#### WebSocket-протокол (для обмена сообщениями в комнатах)

* + - 1. **Соединение с комнатой**
* Эндпоинт: /ws/{room\_name}
* {room\_name} — имя комнаты, с которой устанавливается WebSocket-соединение.
* Метод: WebSocket, подключение к комнате.
  + - 1. **Формат сообщений**:
* При подключении к комнате пользователь отправляет текстовое сообщение с приветствием.
* После установления WebSocket-соединения сообщения между сервером и клиентом отправляются в виде простых текстовых строк.
* Клиент отправляет сообщение в формате: <имя пользователя>: <сообщение>
* Сервер транслирует полученное сообщение всем подключенным пользователям в той же комнате.
  + - 1. **Обработка сообщений**:

На стороне клиента:

* Пользователь отправляет сообщение через WebSocket.
* При отправке клиент формирует строку вида "username: message".
* В случае успешной отправки сообщение отображается в разделе чата.

На стороне сервера:

* Сообщение от клиента принимается и добавляется в базу данных (messages), затем транслируется всем подключенным пользователям в комнате.
* Если сообщение отправлено в некорректном формате, сервер отправляет ошибку и игнорирует сообщение.
  + - 1. **Сообщения об ошибках**:

При ошибках WebSocket-соединения, например, если формат не соответствует ожидаемому, сервер закрывает соединение или отправляет ошибку в формате текста.

Пример обмена данными

* Регистрация пользователя (POST /register):

Запрос: email=email@example.com&password=secure\_password&username=username

Ответ: Перенаправление на страницу логина.

* Отправка сообщения через WebSocket:

Запрос от клиента: ws.send("username: Hello, everyone!")

Ответ сервера (трансляция): "username: Hello, everyone!" — отображается у всех пользователей в комнате.

### Описание API

Проект представляет собой чат-приложение, реализованное на основе FastAPI с использованием WebSocket для обмена сообщениями в реальном времени. API состоит из HTTP-эндпоинтов для управления пользователями и комнатами, а также WebSocket-эндпоинтов для обмена сообщениями.

#### HTTP API

* + - 1. Регистрация пользователя
* Метод: POST
* Эндпоинт: /register
* Описание: Регистрирует нового пользователя с уникальным именем и email.
* Параметры (в теле запроса):
  + username (string, обязательное) — Имя пользователя.
  + email (string, обязательное) — Email пользователя.
  + password (string, обязательное) — Пароль.
* Ответы:
  + 303 See Other — При успешной регистрации, перенаправление на страницу логина.
  + 400 Bad Request — Если email или username уже зарегистрированы.
    - 1. Авторизация пользователя
* Метод: POST
* Эндпоинт: /login
* Описание: Аутентифицирует пользователя по email и паролю, возвращает токен доступа.
* Параметры (в теле запроса):
  + email (string, обязательное) — Email пользователя.
  + password (string, обязательное) — Пароль пользователя.
* Ответы:
  + 303 See Other — При успешной авторизации, перенаправление на страницу с комнатами.
  + 400 Bad Request — Неверный email или пароль.
    - 1. Создание новой комнаты
* Метод: POST
* Эндпоинт: /create-room
* Описание: Создает новую чат-комнату для пользователя.
* Параметры (в теле запроса):
  + room\_name (string, обязательное) — Имя новой комнаты.
* Ответы:
  + 303 See Other — Комната успешно создана, перенаправление на страницу комнат.
  + 400 Bad Request — Комната с таким именем уже существует.
    - 1. Удаление комнаты
* Метод: POST
* Эндпоинт: /delete-room
* Описание: Удаляет комнату, если пользователь является её владельцем.
* Параметры (в теле запроса):
  + room\_id (integer, обязательное) — ID комнаты.
* Ответы:
  + 303 See Other — Комната успешно удалена.
  + 403 Forbidden — Пользователь не является владельцем комнаты.
  + 404 Not Found — Комната не найдена.
    - 1. Поиск комнат
* Метод: GET
* Эндпоинт: /search-rooms
* Описание: Ищет комнаты по ключевому слову в имени.
* Параметры:
  + search\_query (string, обязательное) — Текст для поиска в имени комнаты.
* Ответы:
  + 200 OK — Возвращает страницу с результатами поиска комнат.
    - 1. Страница чата
* Метод: GET
* Эндпоинт: /chat
* Описание: Получает список комнат пользователя и информацию о пользователе.
* Ответы:
  + 200 OK — Страница с доступными чат-комнатами и полем для поиска комнат.
  + 303 See Other — Перенаправление на /login, если пользователь не аутентифицирован.

#### WebSocket API

* + - 1. Подключение к комнате
* Эндпоинт: /ws/{room\_name}
* Описание: Устанавливает WebSocket-соединение для указанной комнаты.
* Параметры пути:
  + room\_name (string) — Имя комнаты для подключения.
* Формат сообщений:
  + Клиент отправляет сообщение в формате: <текстовое сообщение>.
  + Сервер транслирует полученное сообщение всем подключенным пользователям в формате: <имя пользователя>: <сообщение>.
    - 1. Обмен сообщениями
* Сообщение от клиента:
  + Клиент отправляет текстовое сообщение в текущей комнате. Формат: <сообщение>.
* Сообщение от сервера:
  + Сервер транслирует сообщение всем пользователям, подключённым к той же комнате.
* Формат: <имя пользователя>: <сообщение>.
  + - 1. **Отключение от комнаты**

Описание: Если пользователь закрывает соединение, сервер уведомляет других пользователей о его отключении и удаляет соединение из списка активных соединений.

#### Пример запросов и ответов

* + - 1. Регистрация (POST /register):
* Запрос:

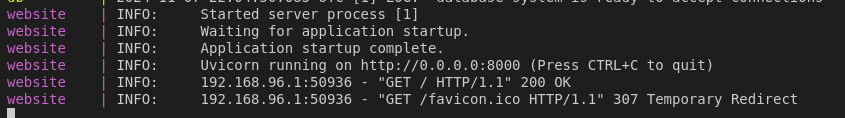
email=user@example.com&password=my\_password&username=user123

* Ответ: 303 See Other, перенаправление на /login.
  + - 1. Отправка сообщения в WebSocket (/ws/room\_name):
* Клиент отправляет: "Привет, как дела?"
* Сервер транслирует: user123: Привет, как дела?

### Скриншоты функционала (каждая страница и кейс из технического задания).



Рисунок 2: построение и запуск сервисов



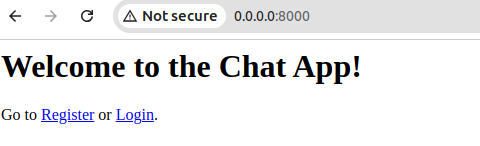


Рисунок 3: интерфейс при открытии веб-страницы



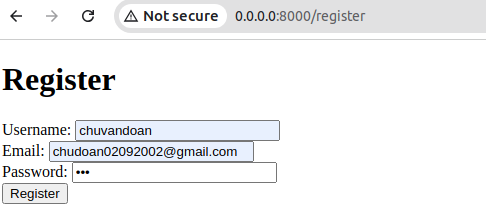


Рисунок 4: выполнение регистрации



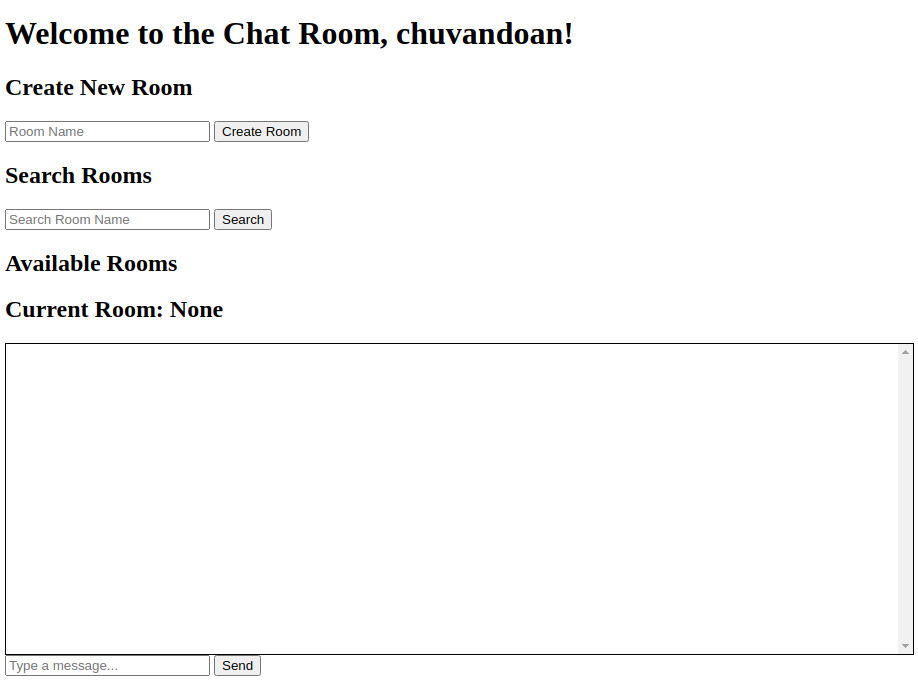


Рисунок 5: успешный вход



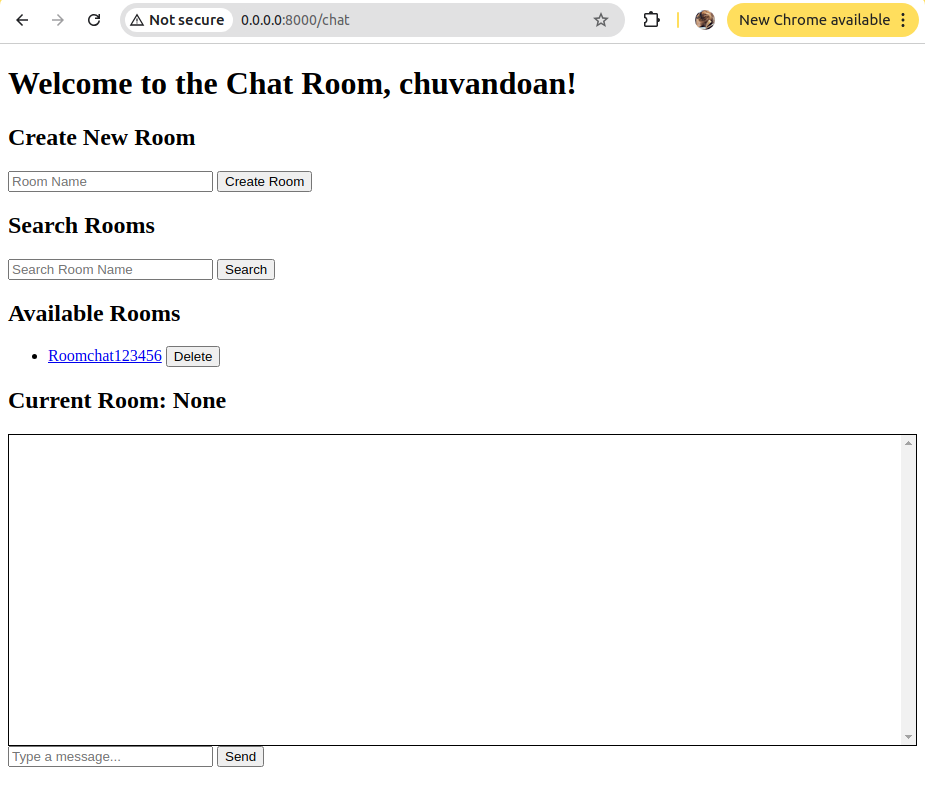


Рисунок 6: создание чата



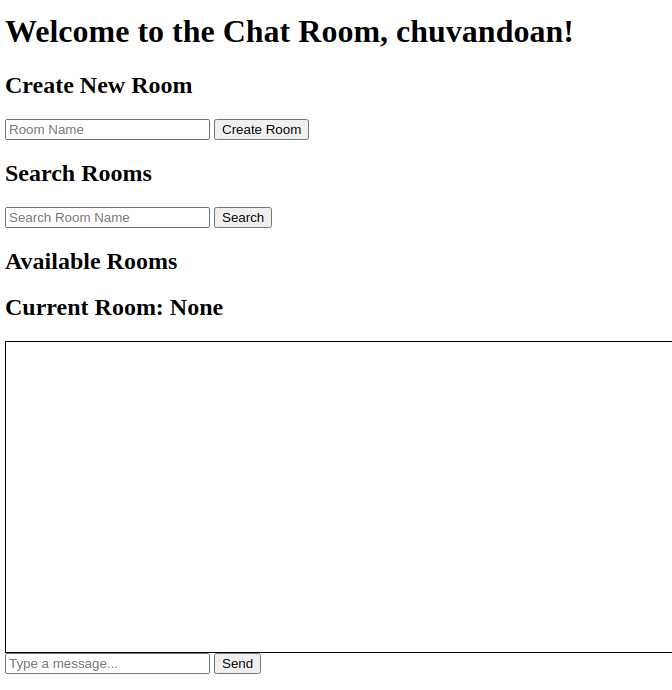
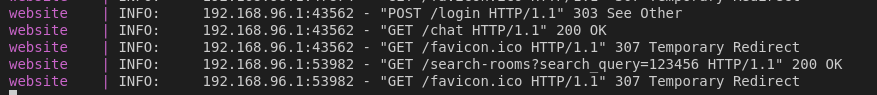


Рисунок 7: удаление чата



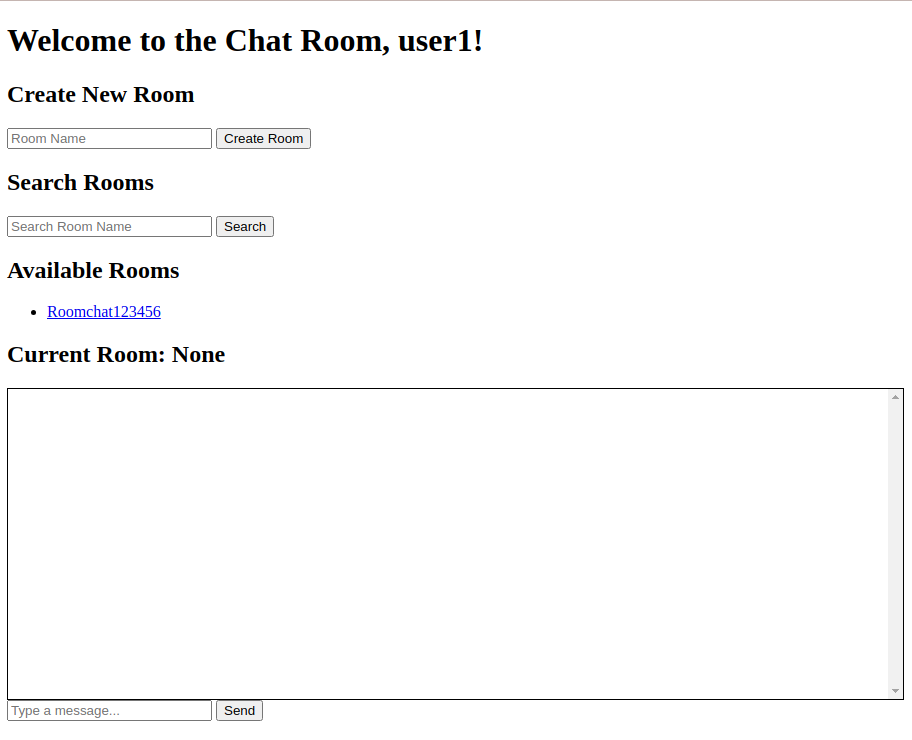
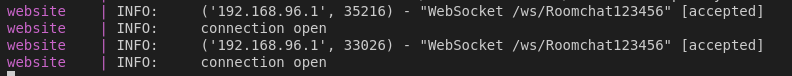


Рисунок 8: создание нового пользователя и поиск чата



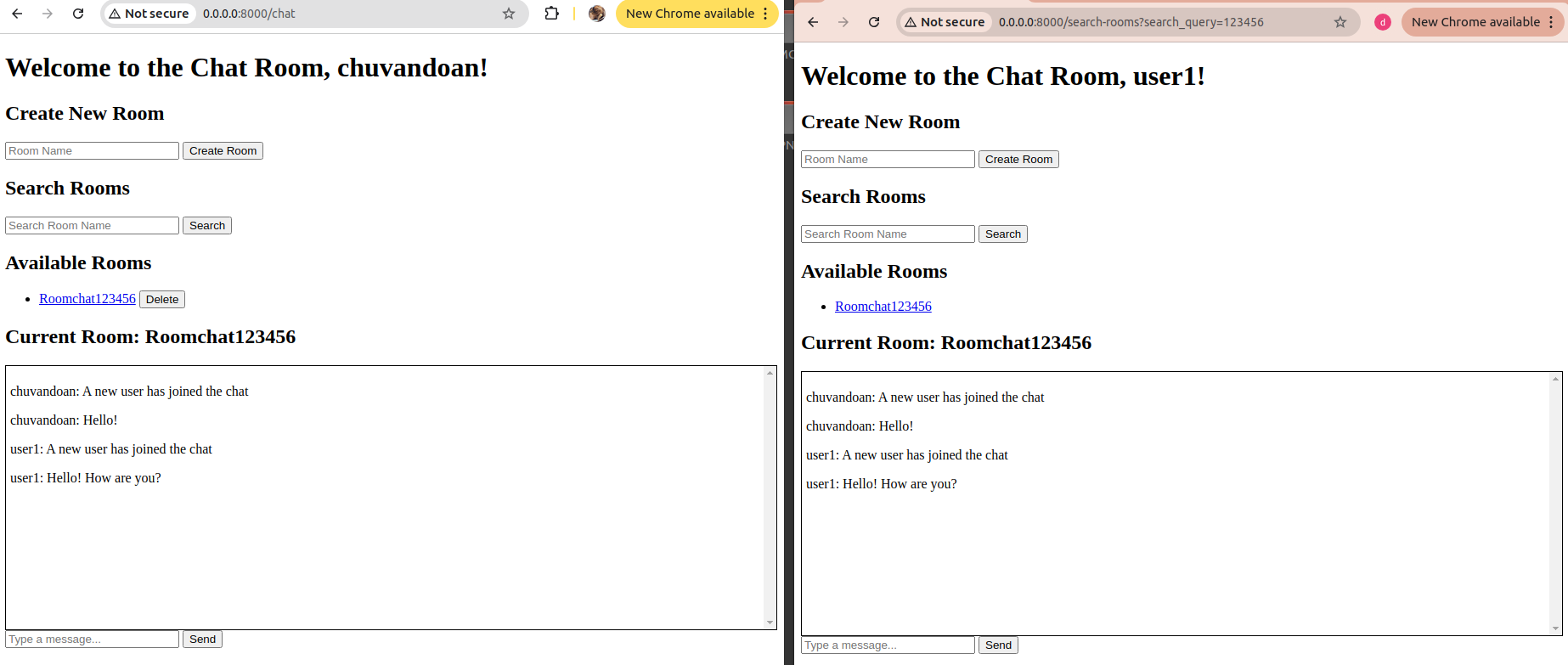


Рисунок 9: два пользователя присоединяются к чату и обмениваются сообщениями

### Коды

#### website backend

* **/website/**
  + **main.py**

from fastapi import FastAPI

from fastapi.responses import HTMLResponse, RedirectResponse

from fastapi.staticfiles import StaticFiles

from .routers import auth, chat

from .database import engine, Base

app = FastAPI()

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

app.include\_router(auth.router)

app.include\_router(chat.router)

@app.get("/", response\_class=HTMLResponse)

async def read\_root():

return """

<html>

<head>

<title>Chat App</title>

</head>

<body>

<h1>Welcome to the Chat App!</h1>

<p>Go to <a href="/register">Register</a> or <a href="/login">Login</a>.</p>

</body>

</html>

"""

# Mount static files

app.mount("/static", StaticFiles(directory="app/static"), name="static")

# Route cho favicon

@app.get("/favicon.ico", include\_in\_schema=False)

async def favicon():

return RedirectResponse(url="/static/favicon.ico")

* + **database.py**

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

import os

DATABASE\_URL = os.getenv("DATABASE\_URL")

engine = create\_engine(DATABASE\_URL)

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

Base = declarative\_base()

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

* + **models.py**

from sqlalchemy import Column, Integer, String, Text, ForeignKey, DateTime

from sqlalchemy.orm import relationship

from .database import Base

from datetime import datetime

class User(Base):

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

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

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

hashed\_password = Column(String)

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

chat\_rooms = relationship("ChatRoom", back\_populates="owner")

class ChatRoom(Base):

\_\_tablename\_\_ = "chat\_rooms"

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

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

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

owner = relationship("User", back\_populates="chat\_rooms")

messages = relationship("Message", back\_populates="room", cascade="all, delete")

class Message(Base):

\_\_tablename\_\_ = "messages"

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

content = Column(Text, nullable=False)

room\_id = Column(Integer, ForeignKey('chat\_rooms.id'))

username = Column(String, nullable=False)

timestamp = Column(DateTime, default=datetime.utcnow)

room = relationship("ChatRoom", back\_populates="messages")

* + **schemas.py**

from pydantic import BaseModel

class UserCreate(BaseModel):

email: str

password: str

username: str

class ChatRoomCreate(BaseModel):

name: str

owner\_id: int

class ChatRoomSearch(BaseModel):

name: str

* **/website/app/routers**
  + **chat.py**

import jwt

from fastapi import APIRouter, Depends, HTTPException, WebSocket, WebSocketDisconnect, Request, Form

from sqlalchemy.orm import Session

from fastapi.responses import HTMLResponse, RedirectResponse

from .. import models, database

from .websocket\_handler import ConnectionManager

from fastapi.templating import Jinja2Templates

router = APIRouter()

templates = Jinja2Templates(directory="app/templates")

manager = ConnectionManager()

SECRET\_KEY = "your\_secret\_key"

ALGORITHM = "HS256"

def get\_db():

db = database.SessionLocal()

try:

yield db

finally:

db.close()

def decode\_jwt(token: str):

if token is None:

return None

try:

token = token.split(" ")[1]

payload = jwt.decode(token, SECRET\_KEY, algorithms=[ALGORITHM])

return payload.get("sub")

except jwt.PyJWTError:

return None

@router.get("/chat", response\_class=HTMLResponse)

async def chat\_page(request: Request, db: Session = Depends(get\_db)):

token = request.cookies.get("access\_token")

if not token:

return RedirectResponse(url="/login")

user\_email = decode\_jwt(token)

if not user\_email:

return RedirectResponse(url="/login")

user = db.query(models.User).filter(models.User.email == user\_email).first()

if not user:

return RedirectResponse(url="/login")

user\_rooms = db.query(models.ChatRoom).filter(models.ChatRoom.owner\_id == user.id).all()

return templates.TemplateResponse("chat.html", {"request": request, "rooms": user\_rooms, "user": user})

@router.post("/create-room", response\_class=HTMLResponse)

async def create\_room(request: Request, room\_name: str = Form(...), db: Session = Depends(get\_db)):

token = request.cookies.get("access\_token")

user\_email = decode\_jwt(token)

user = db.query(models.User).filter(models.User.email == user\_email).first()

existing\_room = db.query(models.ChatRoom).filter(models.ChatRoom.name == room\_name).first()

if existing\_room:

raise HTTPException(status\_code=400, detail="Room already exists")

new\_room = models.ChatRoom(name=room\_name, owner\_id=user.id)

db.add(new\_room)

db.commit()

return RedirectResponse(url="/chat", status\_code=303)

@router.post("/delete-room", response\_class=HTMLResponse)

async def delete\_room(request: Request, room\_id: int = Form(...), db: Session = Depends(get\_db)):

token = request.cookies.get("access\_token")

user\_email = decode\_jwt(token)

user = db.query(models.User).filter(models.User.email == user\_email).first()

room = db.query(models.ChatRoom).filter(models.ChatRoom.id == room\_id, models.ChatRoom.owner\_id == user.id).first()

if not room:

raise HTTPException(status\_code=403, detail="Not authorized to delete this room")

db.delete(room)

db.commit()

return RedirectResponse(url="/chat", status\_code=303)

@router.get("/search-rooms", response\_class=HTMLResponse)

async def search\_rooms(request: Request, search\_query: str, db: Session = Depends(get\_db)):

token = request.cookies.get("access\_token")

user\_email = decode\_jwt(token)

user = db.query(models.User).filter(models.User.email == user\_email).first()

if not user:

return RedirectResponse(url="/login")

rooms = db.query(models.ChatRoom).filter(models.ChatRoom.name.contains(search\_query)).all()

return templates.TemplateResponse("chat.html", {"request": request, "rooms": rooms, "user": user, "search\_query": search\_query})

@router.get("/chat/{room\_name}", response\_class=HTMLResponse)

async def room\_chat\_page(request: Request, room\_name: str, db: Session = Depends(get\_db)):

token = request.cookies.get("access\_token")

user\_email = decode\_jwt(token)

user = db.query(models.User).filter(models.User.email == user\_email).first()

room = db.query(models.ChatRoom).filter(models.ChatRoom.name == room\_name).first()

if not room:

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

messages = db.query(models.Message).filter(models.Message.room\_id == room.id).order\_by(models.Message.timestamp).all()

return templates.TemplateResponse("room\_chat.html", {"request": request, "room": room, "user": user, "messages": messages})

@router.websocket("/ws/{room\_name}")

async def websocket\_endpoint(websocket: WebSocket, room\_name: str, db: Session = Depends(get\_db)):

await manager.connect(websocket, room\_name)

room = db.query(models.ChatRoom).filter(models.ChatRoom.name == room\_name).first()

if not room:

await websocket.close()

return

token = websocket.cookies.get("access\_token")

user\_email = decode\_jwt(token)

user = db.query(models.User).filter(models.User.email == user\_email).first()

if not user:

await websocket.close()

return

username = user.username

messages = db.query(models.Message).filter(models.Message.room\_id == room.id).order\_by(models.Message.timestamp).all()

for message in messages:

await websocket.send\_text(f"{message.username}: {message.content}")

try:

while True:

content = await websocket.receive\_text()

new\_message = models.Message(content=content, room\_id=room.id, username=username)

db.add(new\_message)

db.commit()

await manager.broadcast(f"{username}: {content}", room\_name)

except WebSocketDisconnect:

manager.disconnect(websocket, room\_name)

* + **auth.py**

from fastapi import APIRouter, Depends, HTTPException, Request, Form

from fastapi.responses import HTMLResponse, RedirectResponse

from sqlalchemy.orm import Session

from passlib.context import CryptContext

from .. import models, database

import jwt

from datetime import datetime, timedelta

from fastapi.templating import Jinja2Templates

router = APIRouter()

templates = Jinja2Templates(directory="app/templates")

SECRET\_KEY = "your\_secret\_key"

ALGORITHM = "HS256"

pwd\_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def get\_db():

db = database.SessionLocal()

try:

yield db

finally:

db.close()

def create\_access\_token(data: dict):

to\_encode = data.copy()

expire = datetime.utcnow() + timedelta(hours=1)

to\_encode.update({"exp": expire})

return jwt.encode(to\_encode, SECRET\_KEY, algorithm=ALGORITHM)

@router.get("/register", response\_class=HTMLResponse)

async def get\_register\_form(request: Request):

return templates.TemplateResponse("register.html", {"request": request})

@router.post("/register")

def register(

username: str = Form(...),

email: str = Form(...),

password: str = Form(...),

db: Session = Depends(get\_db)

):

db\_user\_email = db.query(models.User).filter(models.User.email == email).first()

db\_user\_username = db.query(models.User).filter(models.User.username == username).first()

if db\_user\_email or db\_user\_username:

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

hashed\_password = pwd\_context.hash(password)

db\_user = models.User(email=email, hashed\_password=hashed\_password, username=username)

db.add(db\_user)

db.commit()

db.refresh(db\_user)

return RedirectResponse(url="/login", status\_code=303)

@router.get("/login", response\_class=HTMLResponse)

async def get\_login\_form(request: Request):

return templates.TemplateResponse("login.html", {"request": request})

@router.post("/login")

async def login(

request: Request,

email: str = Form(...),

password: str = Form(...),

db: Session = Depends(get\_db)

):

db\_user = db.query(models.User).filter(models.User.email == email).first()

if not db\_user or not pwd\_context.verify(password, db\_user.hashed\_password):

return templates.TemplateResponse("login.html", {

"request": request,

"error\_message": "Invalid email or password. Please try again."

})

access\_token = create\_access\_token(data={"sub": db\_user.email, "username": db\_user.username})

response = RedirectResponse(url="/chat", status\_code=303)

response.set\_cookie(key="access\_token", value=f"Bearer {access\_token}", httponly=True)

return response

* + **websocket\_handler.py**

from fastapi import WebSocket

from typing import Dict, List

class ConnectionManager:

def \_\_init\_\_(self):

self.active\_connections: Dict[str, List[WebSocket]] = {}

async def connect(self, websocket: WebSocket, room\_name: str):

await websocket.accept()

if room\_name not in self.active\_connections:

self.active\_connections[room\_name] = []

self.active\_connections[room\_name].append(websocket)

def disconnect(self, websocket: WebSocket, room\_name: str):

if room\_name in self.active\_connections:

self.active\_connections[room\_name].remove(websocket)

if not self.active\_connections[room\_name]:

del self.active\_connections[room\_name]

async def broadcast(self, message: str, room\_name: str):

if room\_name in self.active\_connections:

for connection in self.active\_connections[room\_name]:

await connection.send\_text(message)

#### chat backend

* /chat/app/

from fastapi import FastAPI, WebSocket, WebSocketDisconnect, Depends

from .websocket\_handler import ConnectionManager

import jwt

app = FastAPI()

manager = ConnectionManager()

SECRET\_KEY = "your\_secret\_key"

ALGORITHM = "HS256"

def decode\_jwt(token: str):

try:

payload = jwt.decode(token, SECRET\_KEY, algorithms=[ALGORITHM])

return payload.get("username")

except jwt.PyJWTError:

return None

@app.websocket("/ws/{room\_name}")

async def websocket\_endpoint(websocket: WebSocket, room\_name: str, token: str = None):

username = decode\_jwt(token)

if not username:

await websocket.close(code=1008)

return

await manager.connect(websocket, username)

try:

await manager.broadcast(f"{username} joined the chat")

while True:

data = await websocket.receive\_text()

await manager.broadcast(f"{username}: {data}")

except WebSocketDisconnect:

manager.disconnect(websocket)

await manager.broadcast(f"{username} left the chat")

* + **websocket\_handler.py**

from fastapi import WebSocket

from typing import Dict, List

class ConnectionManager:

def \_\_init\_\_(self):

self.active\_connections: Dict[str, List[WebSocket]] = {}

async def connect(self, websocket: WebSocket, room\_name: str):

await websocket.accept()

if room\_name not in self.active\_connections:

self.active\_connections[room\_name] = []

self.active\_connections[room\_name].append(websocket)

def disconnect(self, websocket: WebSocket, room\_name: str):

if room\_name in self.active\_connections:

self.active\_connections[room\_name].remove(websocket)

async def broadcast(self, message: str, room\_name: str):

for connection in self.active\_connections[room\_name]:

await connection.send\_text(message)

## Заключение

В ходе выполнения данного проекта был разработан и реализован функционал веб-приложения для обмена сообщениями в реальном времени, включающего регистрацию пользователей, аутентификацию, создание и управление чат-комнатами, а также подключение к комнатам через WebSocket для непосредственного обмена сообщениями между пользователями. Были проработаны основные элементы клиент-серверного взаимодействия, включая HTTP и WebSocket API, с применением современных инструментов и технологий, таких как FastAPI, PostgreSQL и Docker.

Реализация проекта позволила закрепить навыки работы с:

* построением RESTful и WebSocket API,
* управлением сессиями и аутентификацией через JWT токены,
* взаимодействием с базами данных через ORM,
* применением Docker для контейнеризации микросервисов,
* проектированием структуры базы данных и построением связанных таблиц для обеспечения целостности данных.

Этот опыт укрепил практические навыки разработки микросервисной архитектуры и взаимодействия компонентов приложения, а также способствовал лучшему пониманию организации реального проекта с использованием современных технологий для обеспечения масштабируемости и устойчивости веб-приложения.

## Список использованных источников

* FastAPI
  + Документация FastAPI<https://fastapi.tiangolo.com/>
  + FastAPI Tutorial <https://realpython.com/fastapi-python-web-apis/>
* PostgreSQL
  + Официальная документация PostgreSQL <https://www.postgresql.org/docs/>
  + SQLAlchemy ORM <https://docs.sqlalchemy.org/>
* Docker
  + Документация Docker <https://docs.docker.com/>
  + Docker Compose Guide <https://docs.docker.com/compose/>
* Nginx
  + Официальная документация Nginx <https://nginx.org/ru/docs/>
  + Nginx и настройка обратного прокси <https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/>
* JWT JSON Web Tokens
  + Документация JWT <https://jwt.io/introduction/>
  + Использование JWT в FastAPI <https://github.com/fastapi/fastapi>
* WebSocket
  + Введение в WebSocket <https://developer.mozilla.org/ru/docs/Web/API/WebSockets_API>
  + WebSocket в FastAPI <https://fastapi.tiangolo.com/advanced/websockets/>
* Аутентификация и авторизация
  + OAuth 2.0 и OpenID Connect <https://auth0.com/docs/get-started> – рекомендации по протоколам безопасности
  + Обеспечение безопасности API в FastAPI <https://fastapi.tiangolo.com/tutorial/security/>
* Микросервисная архитектура
  + Обзор микросервисной архитектуры <https://martinfowler.com/microservices/>
  + Микросервисы и Docker <https://docs.docker.com/get-started/overview/>
* Проектирование и нормализация баз данных
  + Принципы нормализации базы данных <https://www.guru99.com/database-normalization.html>
  + ER диаграммы для проектирования базы данных <https://vertabelo.com/blog/what-is-an-entity-relationship-diagram/>
* Шаблоны и генерация HTML на сервере
  + Использование Jinja2 в FastAPI <https://fastapi.tiangolo.com/advanced/templates/>
* Работа с асинхронными функциями
  + Асинхронное программирование в Python <https://docs.python.org/3/library/asyncio.html>
  + Асинхронный ввод/вывод в FastAPI <https://fastapi.tiangolo.com/async/>