# Контрольная точка 0

**1. Описание архитектуры мессенджера**

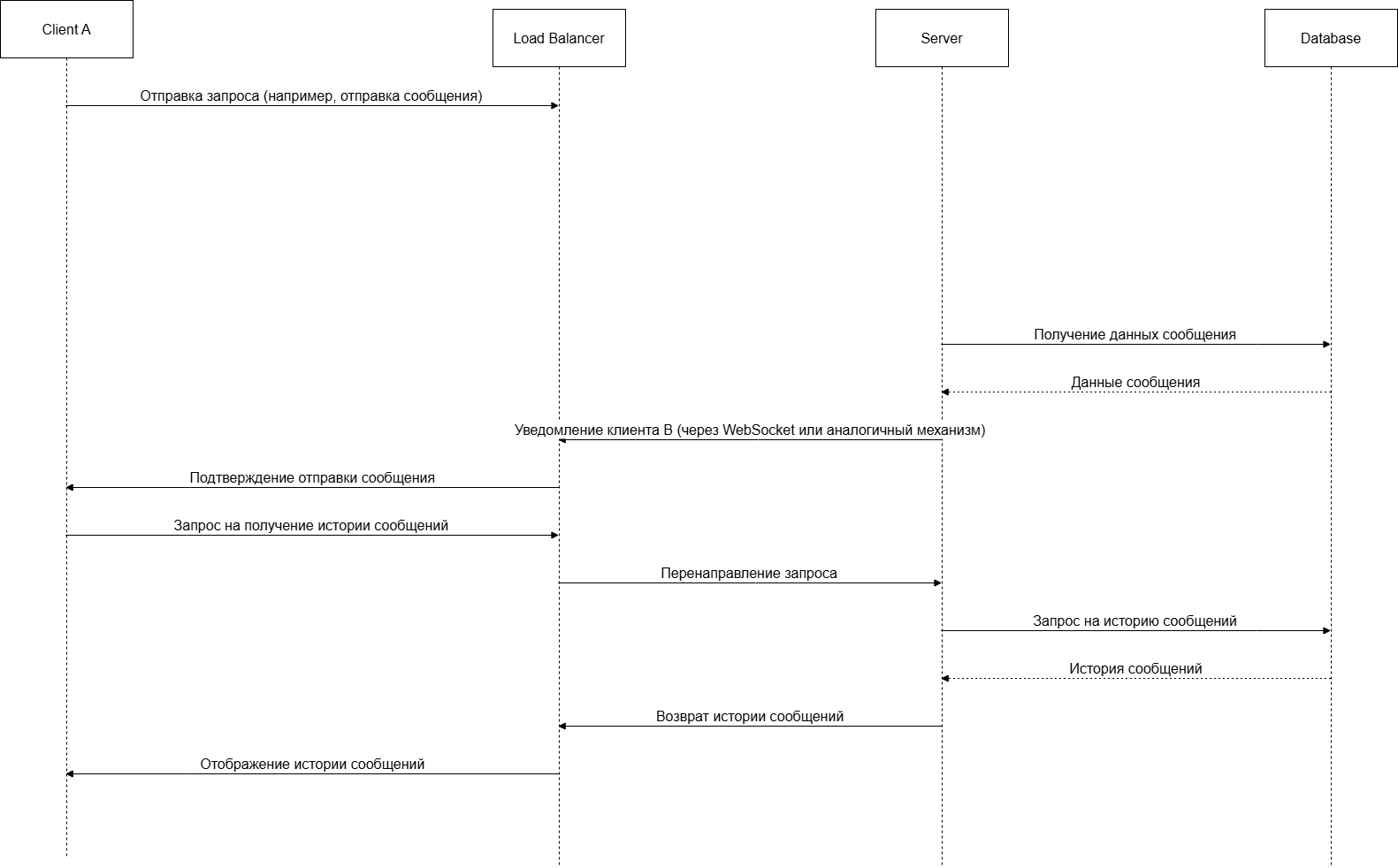
**Целью программы является** отправка и получение сообщений в реальном времени.

Компоненты:

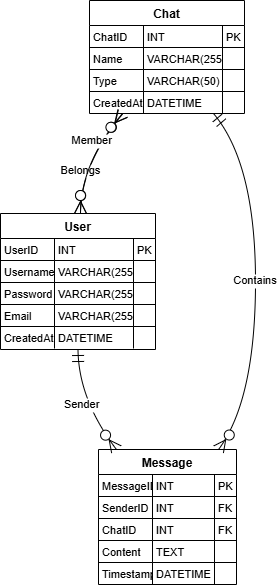
* Клиент: Приложение пользователя.
* Сервер приложений: Обрабатывает запросы, взаимодействует с БД и брокером.
* База данных: Хранит данные (пользователи, сообщения, чаты).
* Брокер сообщений: Отправляет уведомления в реальном времени.

Клиенты отправляют REST API запросы через балансировщик на серверы приложений. Серверы взаимодействуют с базой данных для хранения данных и публикуют уведомления в брокер сообщений. Брокер доставляет уведомления серверам, которые передают их клиентам через WebSocket или SSE.

**2. Схема взаимодействия компонентов**



**3. Логическая схема базы данных**



**4. Структура API (REST)**

**Базовый URL:** https://api.messenger.com/v1 (пример)

**Формат запросов:** JSON

**Формат ответов:** JSON

**Примеры API endpoints:**

1. **Аутентификация:**

**POST /auth/register** - Регистрация нового пользователя.

**Request Body:**

{

"username": "john\_doe",

"password": "secure\_password",

"email": "john.doe@example.com"

}

**Response (201 Created):**

{

"user\_id": 123,

"username": "john\_doe",

"email": "john.doe@example.com"

}

**Response (400 Bad Request):** (Например, username уже существует)

{

"error": "Username already exists"

}

**POST /auth/login** - Аутентификация пользователя.

**Request Body:**

{

"username": "john\_doe",

"password": "secure\_password"

}

**Response (200 OK):**

{

"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c",

"user\_id": 123,

"username": "john\_doe"

}

**Response (401 Unauthorized):**

{

"error": "Invalid credentials"

}

1. **Пользователи:**

**GET /users/{user\_id}** - Получение информации о пользователе.

**Parameters:**

user\_id (path): ID пользователя.

**Headers:**

Authorization: Bearer <token> (Authentication required)

**Response (200 OK):**

{

"user\_id": 123,

"username": "john\_doe",

"email": "john.doe@example.com"

}

**Response (404 Not Found):**

{

"error": "User not found"

}

**GET /users?search={query}** - Поиск пользователей.

**Parameters:**

search (query): Строка поиска (например, по имени).

**Headers:**

Authorization: Bearer <token> (Authentication required)

**Response (200 OK):**

[

{

"user\_id": 123,

"username": "john\_doe",

"email": "john.doe@example.com"

},

{

"user\_id": 456,

"username": "jane\_doe",

"email": "jane.doe@example.com"

}

]

1. **Чаты:**

**GET /chats** - Получение списка чатов пользователя.

**Headers:**

Authorization: Bearer <token> (Authentication required)

**Response (200 OK):**

[

{

"chat\_id": 1,

"name": "John & Jane",

"type": "personal",

"last\_message": {

"content": "Hello!",

"timestamp": "2023-10-27T10:00:00Z"

}

},

{

"chat\_id": 2,

"name": "Developers Group",

"type": "group",

"last\_message": {

"content": "Let's have a meeting",

"timestamp": "2023-10-27T09:00:00Z"

}

}

]

**GET /chats/{chat\_id}** - Получение информации о чате.

**Parameters:**

chat\_id (path): ID чата.

**Headers:**

Authorization: Bearer <token> (Authentication required)

**Response (200 OK):**

{

"chat\_id": 1,

"name": "John & Jane",

"type": "personal",

"participants": [

{

"user\_id": 123,

"username": "john\_doe"

},

{

"user\_id": 456,

"username": "jane\_doe"

}

]

}

**POST /chats** - Создание нового чата.

**Request Body:**

{

"type": "personal",

"participant\_ids": [456] *// list of other user id(s) involved*

}

*// or*

{

"type": "group",

"name": "My New Group",

"participant\_ids": [456, 789]

}

**Headers:**

Authorization: Bearer <token> (Authentication required)

**Response (201 Created):**

{

"chat\_id": 3,

"name": "My New Group",

"type": "group"

}

1. **Сообщения:**

**GET /chats/{chat\_id}/messages?limit={limit}&offset={offset}** - Получение истории сообщений чата.

**Parameters:**

chat\_id (path): ID чата.

limit (query, optional): Максимальное количество сообщений для получения (default: 20).

offset (query, optional): Смещение для пагинации (default: 0).

**Headers:**

Authorization: Bearer <token> (Authentication required)

**Response (200 OK):**

[

{

"message\_id": 1,

"sender\_id": 123,

"content": "Hello!",

"timestamp": "2023-10-27T10:00:00Z"

},

{

"message\_id": 2,

"sender\_id": 456,

"content": "Hi there!",

"timestamp": "2023-10-27T10:01:00Z"

}

]

**POST /chats/{chat\_id}/messages** - Отправка нового сообщения в чат.

**Parameters:**

chat\_id (path): ID чата.

**Request Body:**

{

"content": "This is a new message"

}

**Headers:**

Authorization: Bearer <token> (Authentication required)

**Response (201 Created):**

{

"message\_id": 3,

"sender\_id": 123,

"content": "This is a new message",

"timestamp": "2023-10-27T10:02:00Z"

}

**5. Стек технологий**

Для разработки мы будем использовать:

* **Фронтенд:** HTML, CSS (интерфейс)
* **Backend:** Python(обработка логики).
* **БД:** PostgreSQL, Docker (хранение данных).
* **Связь:** Github (контроль версий)

# Контрольная точка 1

1. **Описать структуру базы данных;**

Структура базы данных, основанная на предоставленных моделях Python (SQLAlchemy ORM), включает в себя три таблицы: users, chats и messages.

1. Таблица users:

Хранит информацию о зарегистрированных пользователях системы.

Столбцы:

* + id (Integer, PRIMARY KEY): Уникальный идентификатор пользователя. Автоинкрементное целое число.
  + username (String, UNIQUE, NOT NULL): Имя пользователя. Должно быть уникальным и не может быть пустым.
  + email (String, UNIQUE, NOT NULL): Адрес электронной почты пользователя. Должен быть уникальным и не может быть пустым.
  + password\_hash (String, NOT NULL): Хэш пароля пользователя. Важно: Хранить пароли в открытом виде категорически запрещено. Здесь должен храниться только хэш.

2. Таблица chats:

Хранит информацию о чатах, существующих в системе.

Столбцы:

* + id (Integer, PRIMARY KEY): Уникальный идентификатор чата. Автоинкрементное целое число.
  + name (String, NOT NULL): Название чата. Не может быть пустым.

3. Таблица messages:

Хранит сообщения, отправленные в чатах.

Столбцы:

* + id (Integer, PRIMARY KEY): Уникальный идентификатор сообщения. Автоинкрементное целое число.
  + chat\_id (Integer, FOREIGN KEY -> chats.id): Идентификатор чата, к которому относится сообщение. Является внешним ключом, ссылающимся на столбец id в таблице chats.
  + sender\_id (Integer, FOREIGN KEY -> users.id): Идентификатор пользователя, отправившего сообщение. Является внешним ключом, ссылающимся на столбец id в таблице users.
  + content (String, NOT NULL): Текст сообщения. Не может быть пустым.
  + timestamp (DateTime, DEFAULT datetime.utcnow): Дата и время отправки сообщения. По умолчанию устанавливается текущее время в UTC.

1. **Скрипты**

from sqlalchemy.orm import sessionmaker

from sqlalchemy import create\_engine

from models import User, Chat, Message

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

from sqlalchemy.orm import relationship

from sqlalchemy.ext.declarative import declarative\_base

from datetime import datetime

from faker import Faker

faker = Faker()

DATABASE\_URL = "postgresql://user:password@localhost:5432/messenger"

engine = create\_engine(DATABASE\_URL)

Session = sessionmaker(bind=engine)

session = Session()

Base = declarative\_base()

# Таблица пользователей

class User(Base):

    \_\_tablename\_\_ = 'users'

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

    username = Column(String, unique=False, nullable=False)

    email = Column(String, unique=True, nullable=False)

    password\_hash = Column(String, nullable=False)

# Таблица чатов

class Chat(Base):

    \_\_tablename\_\_ = 'chats'

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

    name = Column(String, nullable=False)

# Таблица сообщений

class Message(Base):

    \_\_tablename\_\_ = 'messages'

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

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

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

    content = Column(String, nullable=False)

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

    chat = relationship("Chat")

    sender = relationship("User")

Base.metadata.create\_all(engine)

'''with engine.connect() as conn:

    conn.execute(text("ALTER SEQUENCE users\_id\_seq RESTART WITH 1;"))

    conn.execute(text("ALTER SEQUENCE chats\_id\_seq RESTART WITH 1;"))

    conn.execute(text("ALTER SEQUENCE messages\_id\_seq RESTART WITH 1;"))

    conn.commit()'''

# Создание пользователей

user1 = User(username=faker.name(), email=faker.email(), password\_hash="hashed\_password")

user2 = User(username=faker.name(), email=faker.email(), password\_hash="hashed\_password")

# Создание чатов

chat1 = Chat(name=faker.text())

chat2 = Chat(name=faker.text())

# Добавление и коммит пользователей и чатов (чтобы получить их id)

session.add\_all([user1, user2, chat1, chat2])

session.commit()  # Важно: commit нужен для получения id!

# Создание сообщений

message1 = Message(chat\_id=chat1.id, sender\_id=user1.id, content=faker.words(6))

message2 = Message(chat\_id=chat1.id, sender\_id=user2.id, content=faker.words(6))

message3 = Message(chat\_id=chat2.id, sender\_id=user1.id, content=faker.words(6))

message4 = Message(chat\_id=chat2.id, sender\_id=user2.id, content=faker.words(6))

# Добавление сообщений

session.add\_all([message1, message2, message3, message4])

session.commit() # Закоммитить сообщения после добавления

# Чтение данных (сообщений из общего чата)

print("\nСообщения из общего чата:")

for message in session.query(Message).filter(Message.chat\_id == chat1.id).all():

    sender = session.query(User).filter(User.id == message.sender\_id).first() # Retrieve sender from the user table

    print(f"  {sender.username}: {message.content} (at {message.timestamp})") # Use sender.username

# Чтение данных (сообщений из разговора John & Jane)

print("\nСообщения из личного чата John & Jane:")

for message in session.query(Message).filter(Message.chat\_id == chat2.id).all():

    sender = session.query(User).filter(User.id == message.sender\_id).first() # Retrieve sender from the user table

    print(f"  {sender.username}: {message.content} (at {message.timestamp})") # Use sender.username

# Печать списка пользователей в каждом чате

print("\nПользователи в чате:", chat1.name)

users\_in\_chat1 = session.query(User).join(Message).filter(Message.chat\_id == chat1.id).distinct().all()

for user in users\_in\_chat1:

    print(f"  {user.username}")

print("\nПользователи в чате:", chat2.name)

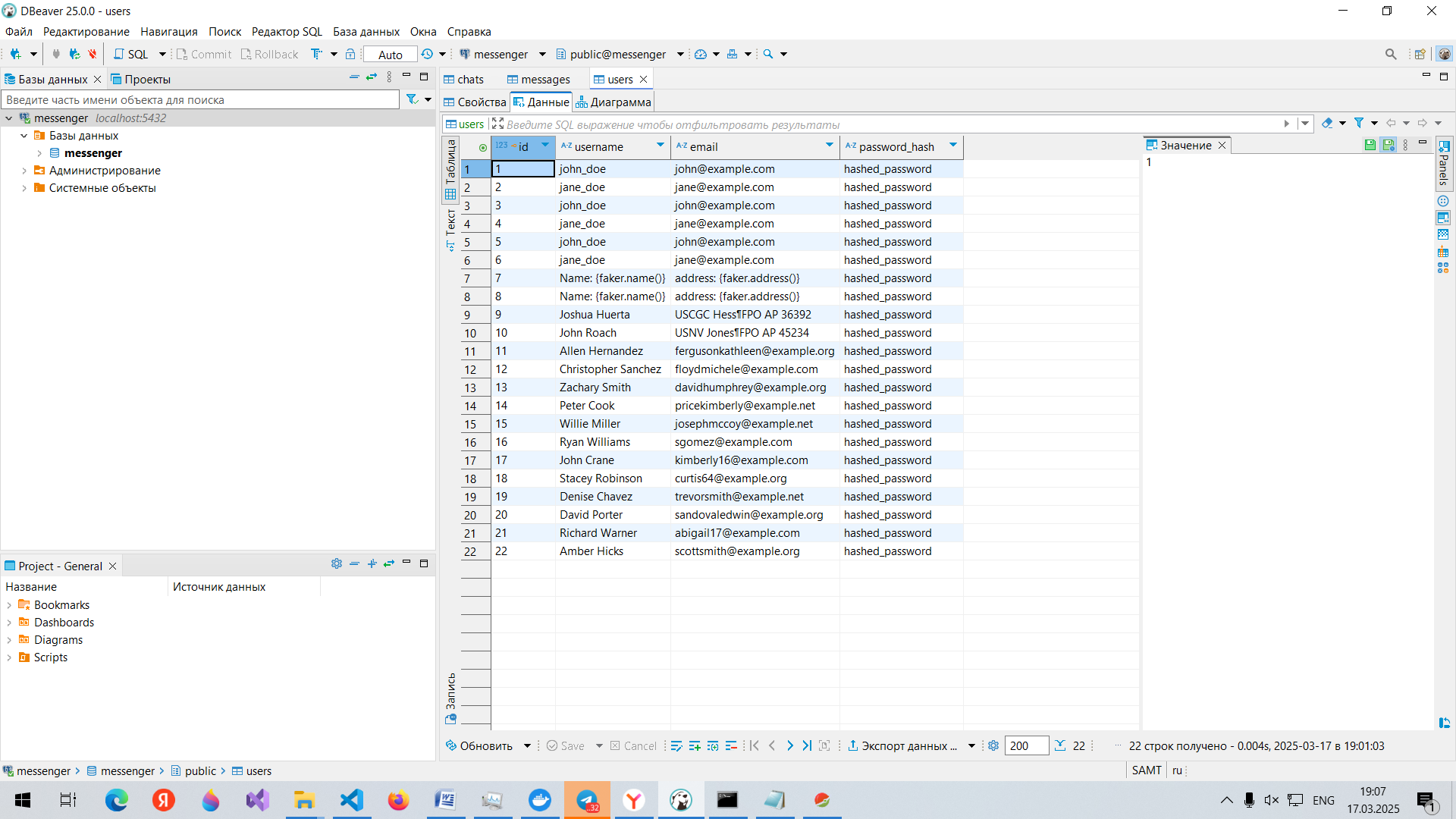
users\_in\_chat2 = session.query(User).join(Message).filter(Message.chat\_id == chat2.id).distinct().all()

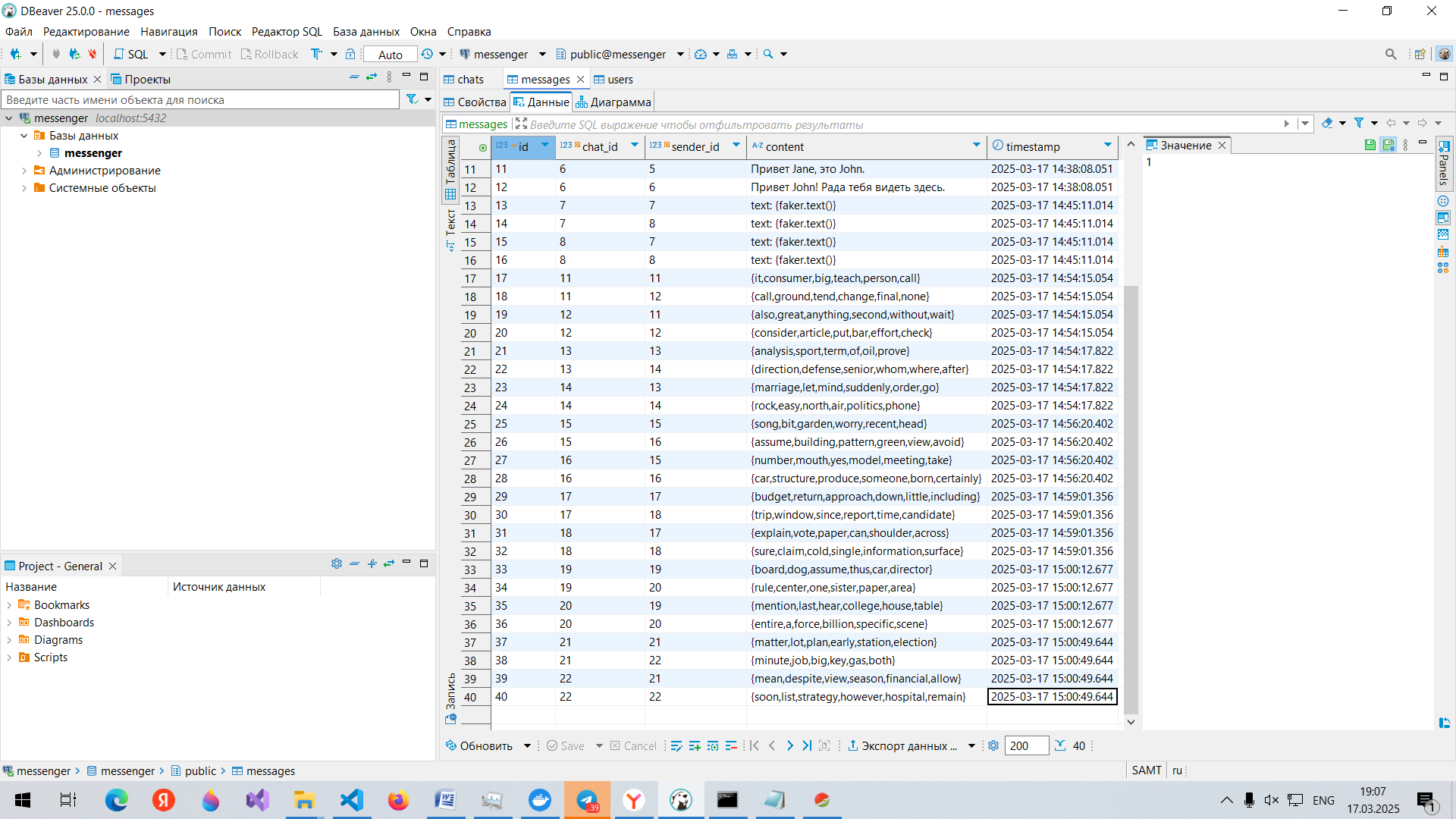
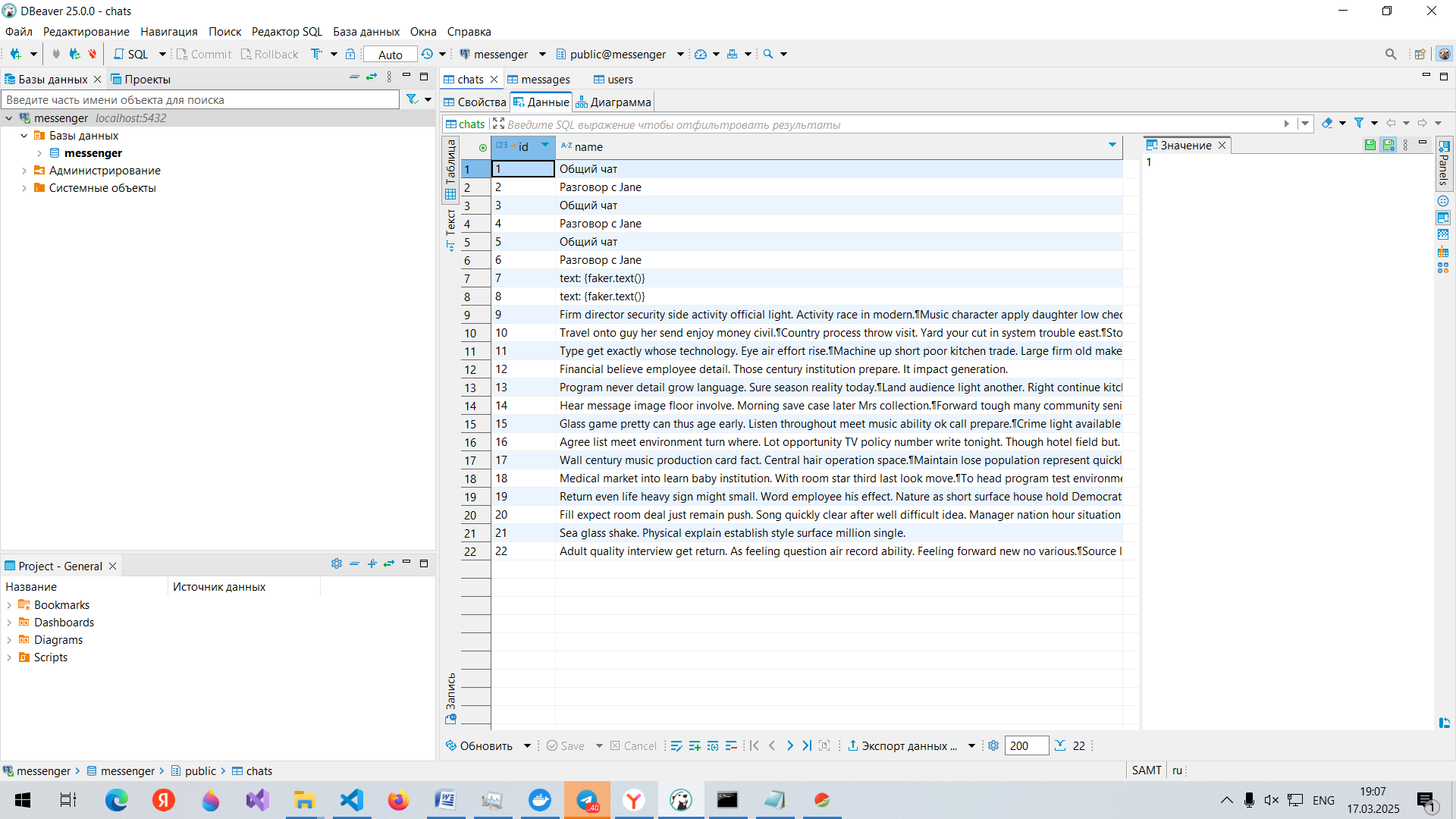
for user in users\_in\_chat2:

    print(f"  {user.username}")

session.close()

Результат:





Сообщения из общего чата:

Catherine Miller: {wait,heart,to,air,prepare,whose} (at 2025-03-17 15:06:30.790023)

Juan Benson: {sport,near,evening,success,admit,hour} (at 2025-03-17 15:06:30.790023)

Сообщения из личного чата John & Jane:

Catherine Miller: {individual,old,dog,necessary,him,power} (at 2025-03-17 15:06:30.790535)

Juan Benson: {ahead,hard,matter,seem,year,executive} (at 2025-03-17 15:06:30.790535)

Пользователи в чате: Big else economy not key dream us. Painting check front maintain. Third once firm family floor parent.

Even race represent but.

Difference PM dream build.

Catherine Miller

Juan Benson

Пользователи в чате: No trade light car trouble. Stop hot choose. Choice home suffer describe.

Consumer eye at. Scientist eat offer participant might yourself could.

Catherine Miller

Juan Benson

1. **Настроены миграции (Alembic):**

alembic init migrations

sqlalchemy.url = postgresql://user:password@localhost:5432/messenger

alembic revision --autogenerate -m "Initial migration"

alembic upgrade head

# Контрольная точка 2

1. **Код API**

from flask import Flask, request, jsonify

from flask\_sqlalchemy import SQLAlchemy

from flask\_marshmallow import Marshmallow

from datetime import datetime

#from marshmallow\_sqlalchemy import SQLAlchemyAutoSchema

# Инициализация приложения

app = Flask(\_\_name\_\_)

@app.route('/')

def home():

    return "Welcome to the Messenger API!"

# Настройки базы данных

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'postgresql://user:password@localhost:5432/messenger'

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

# Инициализация базы данных и Marshmallow

db = SQLAlchemy(app)

ma = Marshmallow(app)

class UserSchema(ma.Schema):

    id = ma.Int()

    username = ma.Str()

    email = ma.Str()

    password\_hash = ma.Str()

class ChatSchema(ma.Schema):

    id = ma.Int()

    name = ma.Str()

class MessageSchema(ma.Schema):

    id = ma.Int()

    chat\_id = ma.Int()

    sender\_id = ma.Int()

    content = ma.Str()

    timestamp = ma.DateTime()

# Сущности моделей из вашего проекта

class User(db.Model):

    \_\_tablename\_\_ = 'users'  # Update this to 'users', not 'user'

    id = db.Column(db.Integer, primary\_key=True)

    username = db.Column(db.String(80), unique=True, nullable=False)

    email = db.Column(db.String(120), unique=True, nullable=False)

    password\_hash = db.Column(db.String(128))

    def \_\_repr\_\_(self):

        return f'<User {self.username}>'

class Chat(db.Model):

    \_\_tablename\_\_ = 'chats'

    id = db.Column(db.Integer, primary\_key=True)

    name = db.Column(db.String, nullable=False)

    def \_\_repr\_\_(self):

        return f'<Chat {self.username}>'

class Message(db.Model):

    \_\_tablename\_\_ = 'messages'

    id = db.Column(db.Integer, primary\_key=True)

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

    sender\_id = db.Column(db.Integer, db.ForeignKey('user.id'))

    content = db.Column(db.String, nullable=False)

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

    def \_\_repr\_\_(self):

        return f'<Message {self.username}>'

# Схемы для сериализации

class UserSchema(ma.SQLAlchemyAutoSchema):

    class Meta:

        model = User

class ChatSchema(ma.SQLAlchemyAutoSchema):

    class Meta:

        model = Chat

class MessageSchema(ma.SQLAlchemyAutoSchema):

    class Meta:

        model = Message

# Реализация маршрутов для CRUD-операций

# Получение всех пользователей

@app.route('/users', methods=['GET'])

def get\_users():

    users = User.query.all()

    user\_schema = UserSchema(many=True)

    return jsonify(user\_schema.dump(users))

# Получение одного пользователя по ID

@app.route('/users/<int:id>', methods=['GET'])

def get\_user(id):

    user = User.query.get(id)

    if user is None:

        return jsonify({"error": "User not found"}), 404

    user\_schema = UserSchema()

    return jsonify(user\_schema.dump(user))

# Создание нового пользователя

@app.route('/users', methods=['POST'])

def add\_user():

    username = request.json['username']

    email = request.json['email']

    password\_hash = request.json['password\_hash']

    new\_user = User(username=username, email=email, password\_hash=password\_hash)

    db.session.add(new\_user)

    db.session.commit()

    user\_schema = UserSchema()

    return jsonify(user\_schema.dump(new\_user)), 201

# Добавление чата

@app.route('/chats', methods=['POST'])

def add\_chat():

    name = request.json['name']

    new\_chat = Chat(name=name)

    db.session.add(new\_chat)

    db.session.commit()

    chat\_schema = ChatSchema()

    return jsonify(chat\_schema.dump(new\_chat)), 201

# Получение всех сообщений в чате

@app.route('/chats/<int:chat\_id>/messages', methods=['GET'])

def get\_messages(chat\_id):

    messages = Message.query.filter\_by(chat\_id=chat\_id).all()

    message\_schema = MessageSchema(many=True)

    return jsonify(message\_schema.dump(messages))

# Отправка нового сообщения

@app.route('/messages', methods=['POST'])

def add\_message():

    chat\_id = request.json['chat\_id']

    sender\_id = request.json['sender\_id']

    content = request.json['content']

    new\_message = Message(chat\_id=chat\_id, sender\_id=sender\_id, content=content)

    db.session.add(new\_message)

    db.session.commit()

    message\_schema = MessageSchema()

    return jsonify(message\_schema.dump(new\_message)), 201

@app.route('/chats/<int:chat\_id>', methods=['DELETE'])

def delete\_chat(chat\_id):

    # Find the chat by its ID

    chat = Chat.query.get(chat\_id)

    if not chat:

        return jsonify({"error": "Chat not found"}), 404

    # Delete all messages in the chat

    Message.query.filter\_by(chat\_id=chat\_id).delete()

    # Delete the chat itself

    db.session.delete(chat)

    db.session.commit()

    return jsonify({"message": f"Chat with ID {chat\_id} and all its messages deleted successfully."}), 200

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(debug=True)

1. **Обзор структуры API**

Базовый URL: API доступен по адресу <http://127.0.0.1:5000/>.

Сущности:

* Пользователь (users): Представляет пользователя в системе, с полями username (имя пользователя), email (почта) и password\_hash (хэш пароля).
* Чат (chats): Представляет чат, имеет поле name для идентификации чата.
* Сообщение (messages): Представляет сообщение в чате, которое связано с пользователем и чатом, имеет поля content (содержимое сообщения) и timestamp (время отправки).

1. **Эндпоинты API**

* Основной эндпоинт:
  + GET /: Возвращает приветственное сообщение.
* Эндпоинты для пользователей:
  + GET /users: Получение списка всех пользователей.
  + GET /users/<int:id>: Получение информации о пользователе по его ID.
  + POST /users: Создание нового пользователя.
* Эндпоинты для чатов:
  + POST /chats: Создание нового чата.
  + GET /chats/<int:chat\_id>/messages: Получение всех сообщений для указанного чата.
  + DELETE /chats/<int:chat\_id>: Удаление чата и всех его сообщений.
* Эндпоинты для сообщений:
  + POST /messages: Создание нового сообщения в чате.

1. **Модели и схемы**

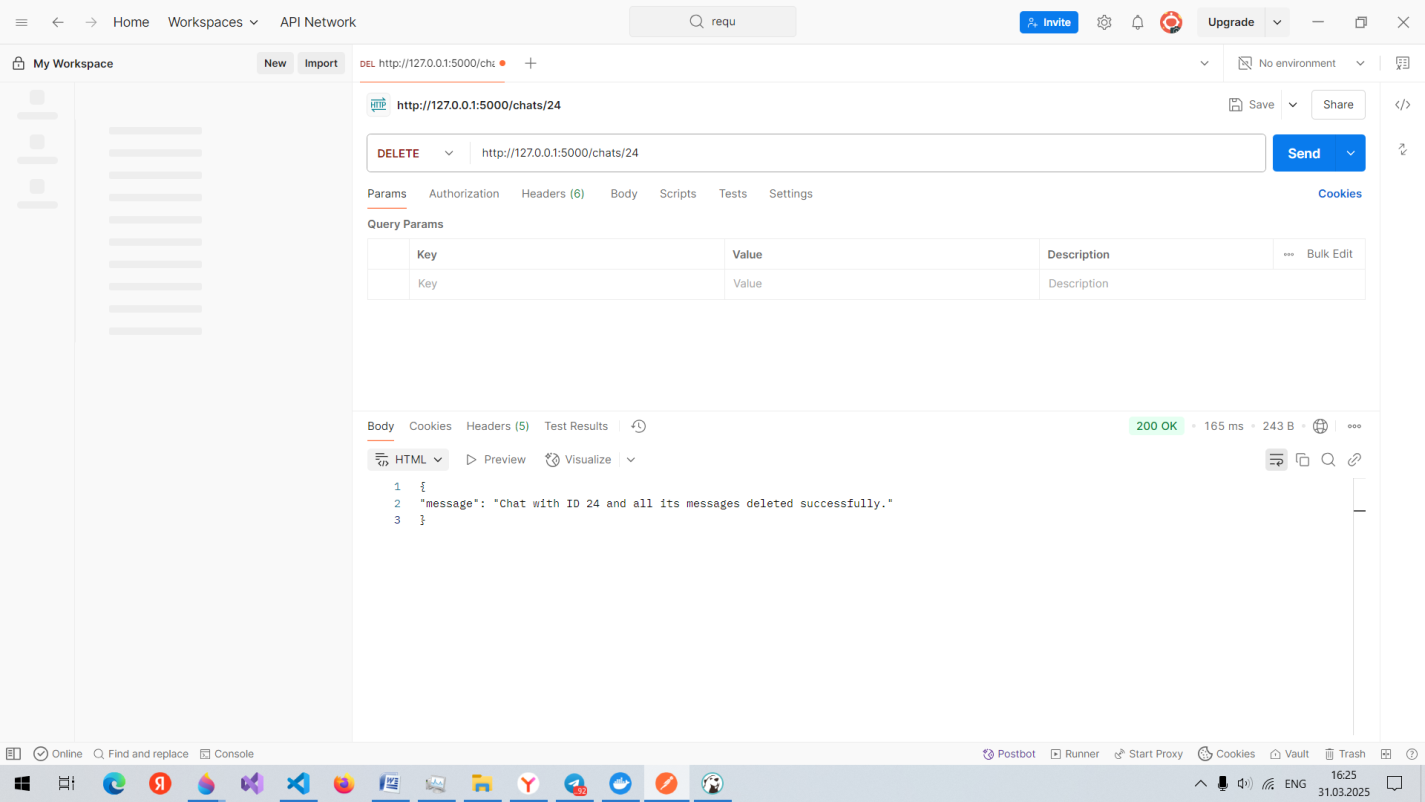
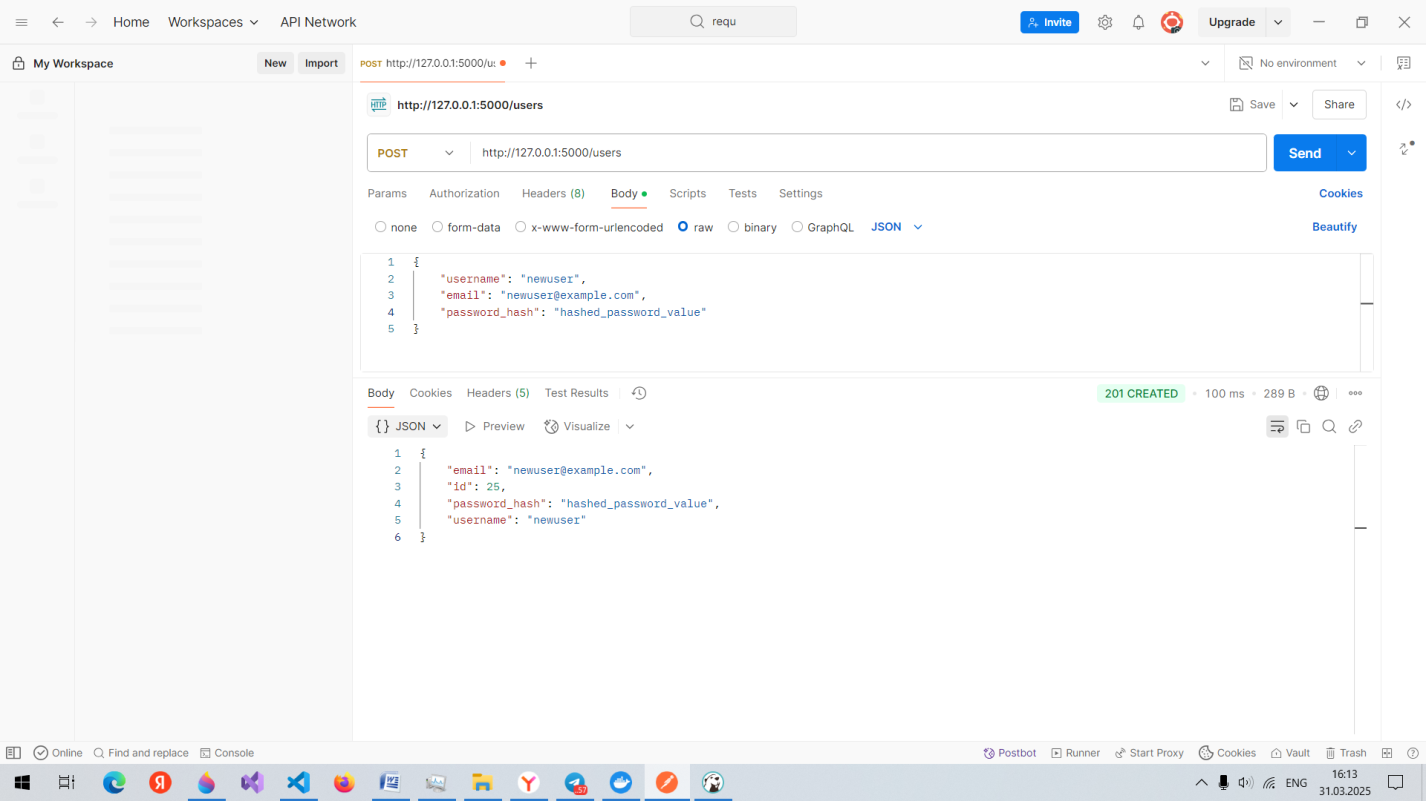
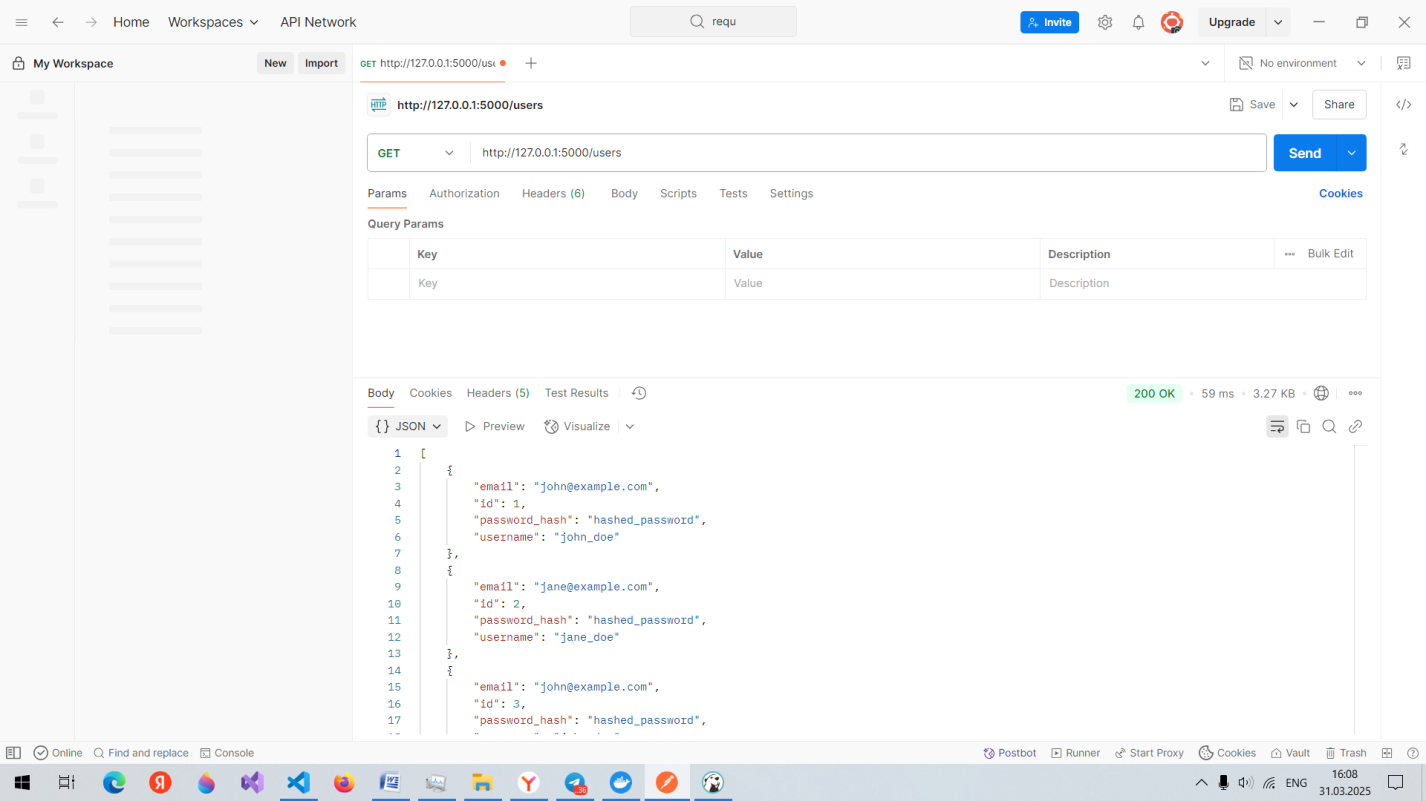
* Модель пользователя (User):
  + id: Первичный ключ, автоинкрементируемое целое число.
  + username: Уникальное строковое поле для имени пользователя.
  + email: Уникальное строковое поле для почты пользователя.
  + password\_hash: Строковое поле для хранения хэшированного пароля.
* Модель чата (Chat):
  + id: Первичный ключ, автоинкрементируемое целое число.
  + name: Имя чата (строковое поле).
* Модель сообщения (Message):
  + id: Первичный ключ, автоинкрементируемое целое число.
  + chat\_id: Внешний ключ, ссылающийся на модель Chat.
  + sender\_id: Внешний ключ, ссылающийся на модель User.
  + content: Содержимое сообщения (строковое поле).
  + timestamp: Время отправки сообщения, по умолчанию текущая дата и время (UTC).

1. **CRUD операции**
2. GET /users: Получение списка всех пользователей, сериализованных с помощью UserSchema.
3. GET /users/int:id: Получение одного пользователя по ID, сериализованного с помощью UserSchema.
4. POST /users: Создание нового пользователя. Для этого необходимо передать username, email и password\_hash в теле запроса, а в ответе вернется созданный объект пользователя.
5. POST /chats: Создание нового чата. Для этого необходимо передать name в теле запроса, а в ответе вернется созданный объект чата.
6. GET /chats/int:chat\_id/messages: Получение всех сообщений для определенного чата, сериализованных с помощью MessageSchema.
7. POST /messages: Создание нового сообщения. Для этого необходимо передать chat\_id, sender\_id и content в теле запроса, а в ответе вернется созданное сообщение.
8. DELETE /chats/int:chat\_id: Удаление чата по его ID. Этот запрос также удаляет все сообщения, связанные с этим чатом.

**Обработка ошибок**

* Пользователь не найден: Если запрос на получение пользователя или чата не находит нужный объект, возвращается код ошибки 404 с соответствующим сообщением ({"error": "User not found"} или {"error": "Chat not found"}).
* Некорректные или отсутствующие данные: Если в запросе отсутствуют необходимые данные для создания пользователя или сообщения, Flask поднимет ошибку. В будущем можно добавить валидацию для обработки этих случаев.

1. **Примеры запросов:**



# Контрольная точка 3

1. **Код API**

from flask import Flask, request, jsonify

from flask\_sqlalchemy import SQLAlchemy

from flask\_marshmallow import Marshmallow

from flask\_jwt\_extended import JWTManager, create\_access\_token, jwt\_required, get\_jwt\_identity, get\_jwt, verify\_jwt\_in\_request

from werkzeug.security import generate\_password\_hash, check\_password\_hash

from datetime import datetime

# Инициализация приложения

app = Flask(\_\_name\_\_)

@app.route('/')

def home():

    return "Welcome to the Messenger API!"

# Настройки базы данных

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'postgresql://user:password@localhost:5432/messenger'

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

# Настройки JWT

app.config['JWT\_SECRET\_KEY'] = 'your\_secret\_key\_here'  # Укажи свой секретный ключ

app.config['JWT\_ACCESS\_TOKEN\_EXPIRES'] = 360000  # Время жизни токена в секундах

jwt = JWTManager(app)

# Инициализация базы данных и Marshmallow и JWT

db = SQLAlchemy(app)

ma = Marshmallow(app)

jwt = JWTManager(app)

# Черный список для токенов

blacklist = set()

@app.before\_request

def check\_token\_in\_blacklist():

    token = request.headers.get('Authorization', None)

    if token:

        token = token.split(" ")[1]  # Извлекаем токен из заголовка Authorization

        try:

            verify\_jwt\_in\_request()

            # Извлекаем данные токена

            decoded\_token = get\_jwt()

            jti = decoded\_token.get('jti')

            # Проверяем, есть ли токен в черном списке

            if jti in blacklist:

                return jsonify({"description": "The token has been revoked.", "error": "token\_revoked"}), 401

        except Exception as e:

            return jsonify({"description": str(e), "error": "invalid\_token"}), 401

@jwt.revoked\_token\_loader

def revoked\_token\_callback(jwt\_header, jwt\_payload):

    return jsonify(

        {"description": "The token has been revoked.", "error": "token\_revoked"}

    ), 401

# Модели

class User(db.Model):

    \_\_tablename\_\_ = 'users'

    id = db.Column(db.Integer, primary\_key=True)

    username = db.Column(db.String(80), unique=True, nullable=False)

    email = db.Column(db.String(120), unique=True, nullable=False)

    password\_hash = db.Column(db.String(128))

    def \_\_repr\_\_(self):

        return f'<User {self.username}>'

class Chat(db.Model):

    \_\_tablename\_\_ = 'chats'

    id = db.Column(db.Integer, primary\_key=True)

    name = db.Column(db.String, nullable=False)

    def \_\_repr\_\_(self):

        return f'<Chat {self.name}>'

class Message(db.Model):

    \_\_tablename\_\_ = 'messages'

    id = db.Column(db.Integer, primary\_key=True)

    chat\_id = db.Column(db.Integer, db.ForeignKey('chats.id'))

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

    content = db.Column(db.String, nullable=False)

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

    def \_\_repr\_\_(self):

        return f'<Message {self.content}>'

# Схемы для сериализации

class UserSchema(ma.SQLAlchemyAutoSchema):

    class Meta:

        model = User

class ChatSchema(ma.SQLAlchemyAutoSchema):

    class Meta:

        model = Chat

class MessageSchema(ma.SQLAlchemyAutoSchema):

    class Meta:

        model = Message

# Регистрация пользователя

@app.route('/register', methods=['POST'])

def register():

    username = request.json.get('username')

    email = request.json.get('email')

    password = request.json.get('password')

    if not username or not email or not password:

        return jsonify({"error": "Missing data"}), 400

    if User.query.filter((User.username == username) | (User.email == email)).first():

        return jsonify({"error": "User already exists"}), 400

    hashed\_password = generate\_password\_hash(password)

    new\_user = User(username=username, email=email, password\_hash=hashed\_password)

    db.session.add(new\_user)

    db.session.commit()

    return jsonify({"message": "User created successfully"}), 201

# Вход пользователя

@app.route('/login', methods=['POST'])

def login():

    username = request.json.get('username')

    password = request.json.get('password')

    user = User.query.filter\_by(username=username).first()

    if not user or not check\_password\_hash(user.password\_hash, password):

        return jsonify({"error": "Invalid username or password"}), 401

    # Создание токена с 'jti'

    access\_token = create\_access\_token(identity=user.id)

    print(f"Generated JWT: {access\_token}")  # Log token for debugging

    return jsonify(access\_token=access\_token)

# Разлогинивание (удаление токена из черного списка)

@app.route('/logout', methods=['POST'])

@jwt\_required()

def logout():

    # Получаем текущий токен

    token = get\_jwt()

    # Извлекаем 'jti' из токена

    jti = token['jti']

    # Добавляем 'jti' в черный список

    blacklist.add(jti)

    return jsonify({"message": "Successfully logged out"}), 200

# Смена пароля

@app.route('/change-password', methods=['POST'])

@jwt\_required()

def change\_password():

    current\_user\_id = get\_jwt\_identity()

    user = User.query.get(current\_user\_id)

    old\_password = request.json.get('old\_password')

    new\_password = request.json.get('new\_password')

    if not check\_password\_hash(user.password\_hash, old\_password):

        return jsonify({"error": "Old password is incorrect"}), 400

    user.password\_hash = generate\_password\_hash(new\_password)

    db.session.commit()

    return jsonify({"message": "Password changed successfully"})

# Пример защищённого эндпоинта

@app.route('/protected', methods=['GET'])

@jwt\_required()

def protected():

    current\_user\_id = get\_jwt\_identity()

    user = User.query.get(current\_user\_id)

    return jsonify(logged\_in\_as=user.username)

# Получение всех пользователей

@app.route('/users', methods=['GET'])

@jwt\_required()

def get\_users():

    users = User.query.all()

    user\_schema = UserSchema(many=True)

    return jsonify(user\_schema.dump(users))

# Получение одного пользователя по ID

@app.route('/users/<int:id>', methods=['GET'])

@jwt\_required()

def get\_user(id):

    user = User.query.get(id)

    if user is None:

        return jsonify({"error": "User not found"}), 404

    user\_schema = UserSchema()

    return jsonify(user\_schema.dump(user))

# Создание нового чата

@app.route('/chats', methods=['POST'])

@jwt\_required()

def add\_chat():

    name = request.json.get('name')

    new\_chat = Chat(name=name)

    db.session.add(new\_chat)

    db.session.commit()

    chat\_schema = ChatSchema()

    return jsonify(chat\_schema.dump(new\_chat)), 201

# Получение всех сообщений в чате

@app.route('/chats/<int:chat\_id>/messages', methods=['GET'])

@jwt\_required()

def get\_messages(chat\_id):

    messages = Message.query.filter\_by(chat\_id=chat\_id).all()

    message\_schema = MessageSchema(many=True)

    return jsonify(message\_schema.dump(messages))

# Отправка нового сообщения

@app.route('/messages', methods=['POST'])

@jwt\_required()

def add\_message():

    chat\_id = request.json.get('chat\_id')

    sender\_id = get\_jwt\_identity()

    content = request.json.get('content')

    new\_message = Message(chat\_id=chat\_id, sender\_id=sender\_id, content=content)

    db.session.add(new\_message)

    db.session.commit()

    message\_schema = MessageSchema()

    return jsonify(message\_schema.dump(new\_message)), 201

# Удаление чата

@app.route('/chats/<int:chat\_id>', methods=['DELETE'])

@jwt\_required()

def delete\_chat(chat\_id):

    chat = Chat.query.get(chat\_id)

    if not chat:

        return jsonify({"error": "Chat not found"}), 404

    Message.query.filter\_by(chat\_id=chat\_id).delete()

    db.session.delete(chat)

    db.session.commit()

    return jsonify({"message": f"Chat with ID {chat\_id} and all its messages deleted successfully."}), 200

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(debug=True)

1. **Модель пользователя**

 id: Уникальный идентификатор пользователя (первичный ключ).

 username: Имя пользователя (уникальное и обязательное).

 email: Электронная почта пользователя (уникальная и обязательная).

 password\_hash: Хеш пароля пользователя, который хранится в базе данных вместо самого пароля.

1. **JWT Аутентификация**

**JWT Настройки:**

* JWT\_SECRET\_KEY: Секретный ключ для создания и проверки токенов. Этот ключ должен быть уникальным и секретным.
* JWT\_ACCESS\_TOKEN\_EXPIRES: Время жизни токена (в данном случае 100 часов).
* JWTManager: Инициализация менеджера JWT, который будет использоваться для создания и проверки токенов.

**Черный список для токенов (для выхода):**

Токены, которые были отозваны (например, при выходе из системы), сохраняются в blacklist. Функция проверяет, находится ли токен в черном списке.

**Роуты для аутентификации:**

* Регистрация:

При регистрации создается новый пользователь с хешированным паролем.

* Вход (Login):

Если имя пользователя и пароль правильные, создается JWT токен, который отправляется клиенту.

* Выход (Logout):

При выходе текущий JWT токен добавляется в черный список, что делает его недействительным.

**Защищенные эндпоинты:**

Для того, чтобы доступ к определенным эндпоинтам был только для авторизованных пользователей, используется декоратор @jwt\_required(). Например:

@app.route('/protected', methods=['GET'])

@jwt\_required()

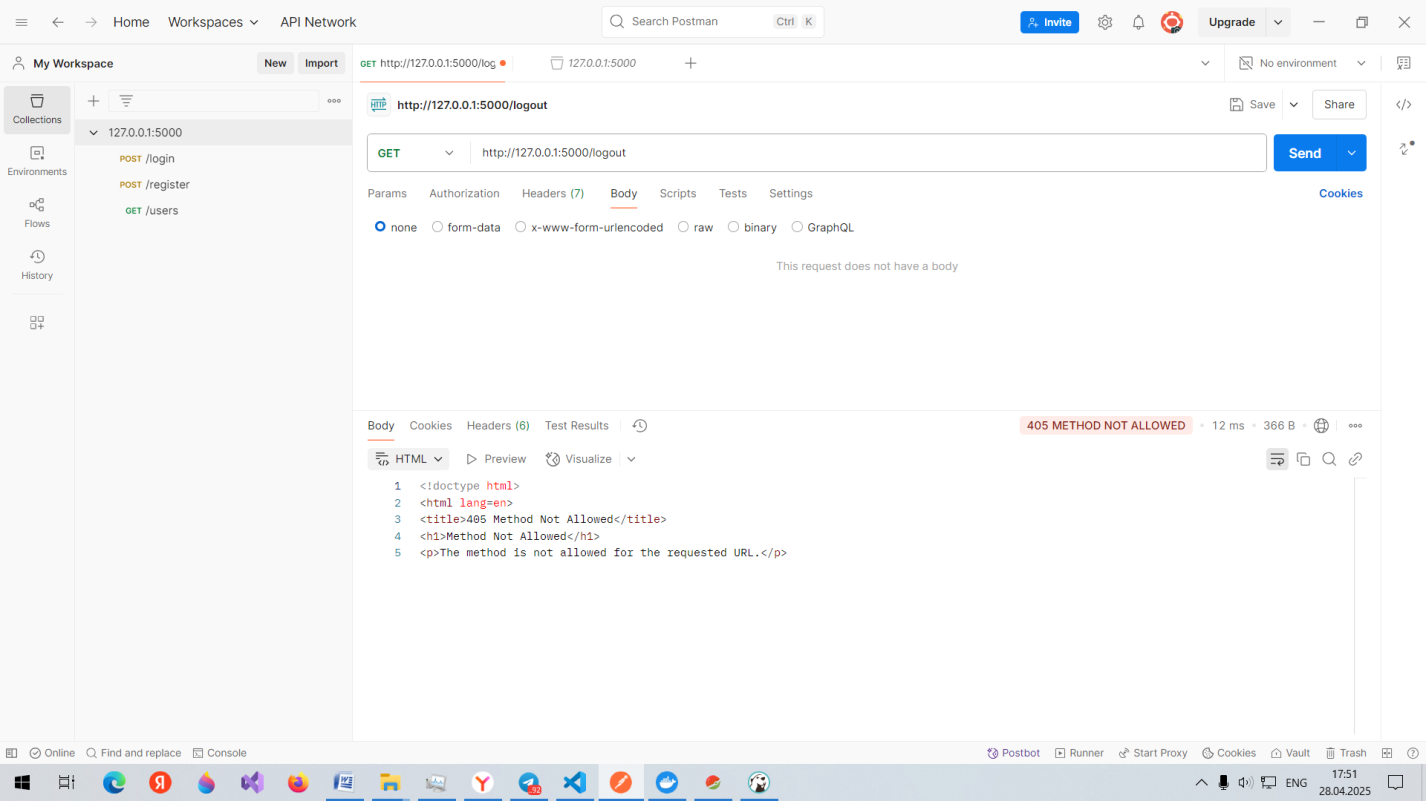
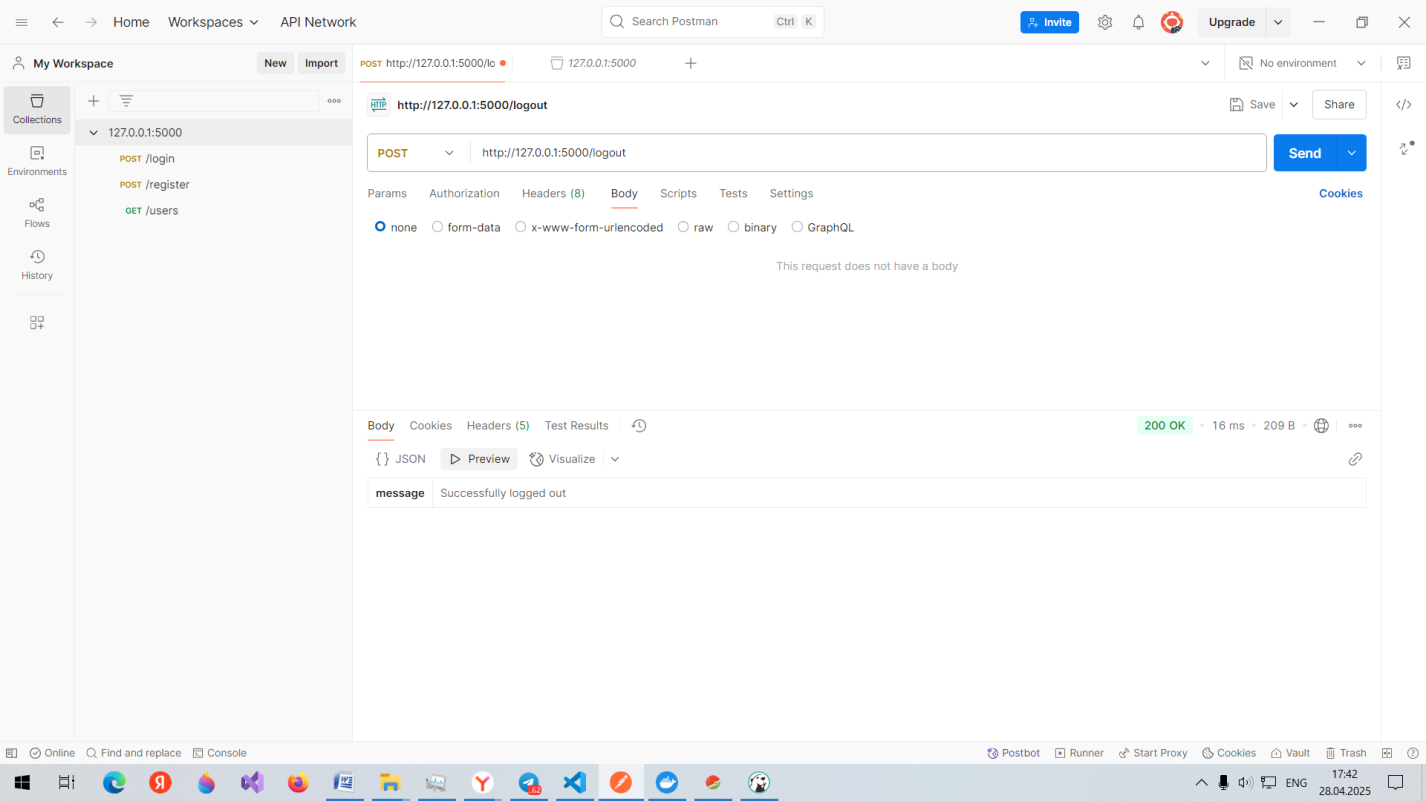
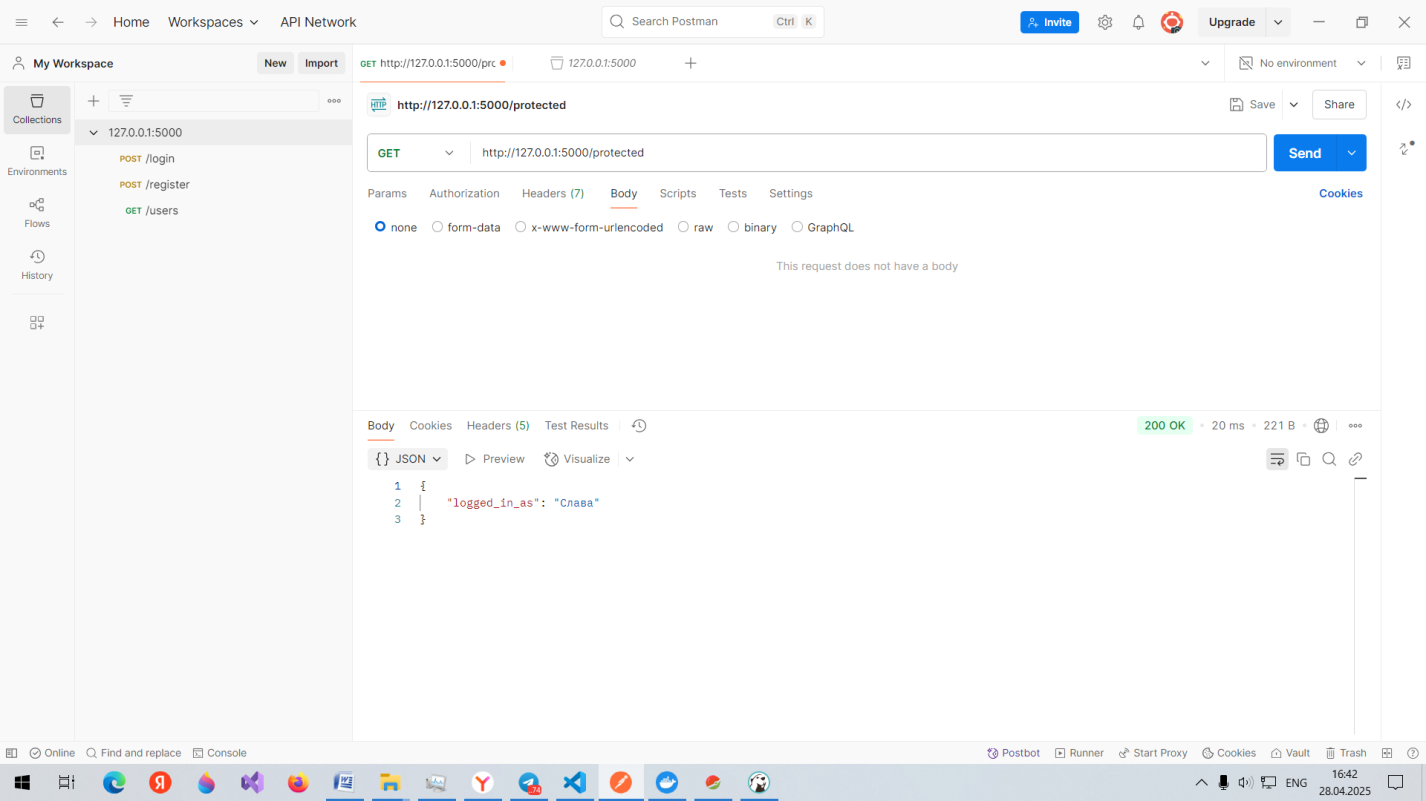
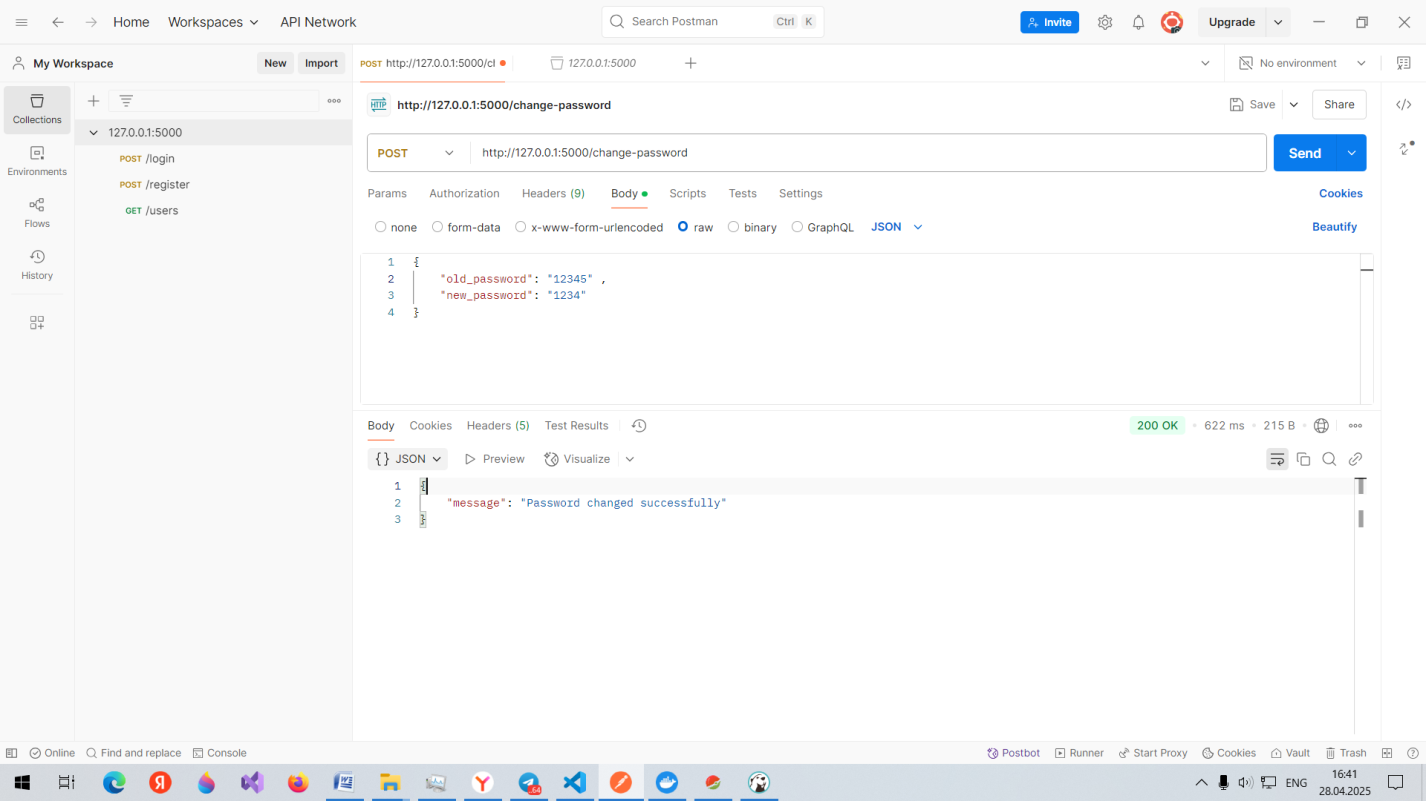
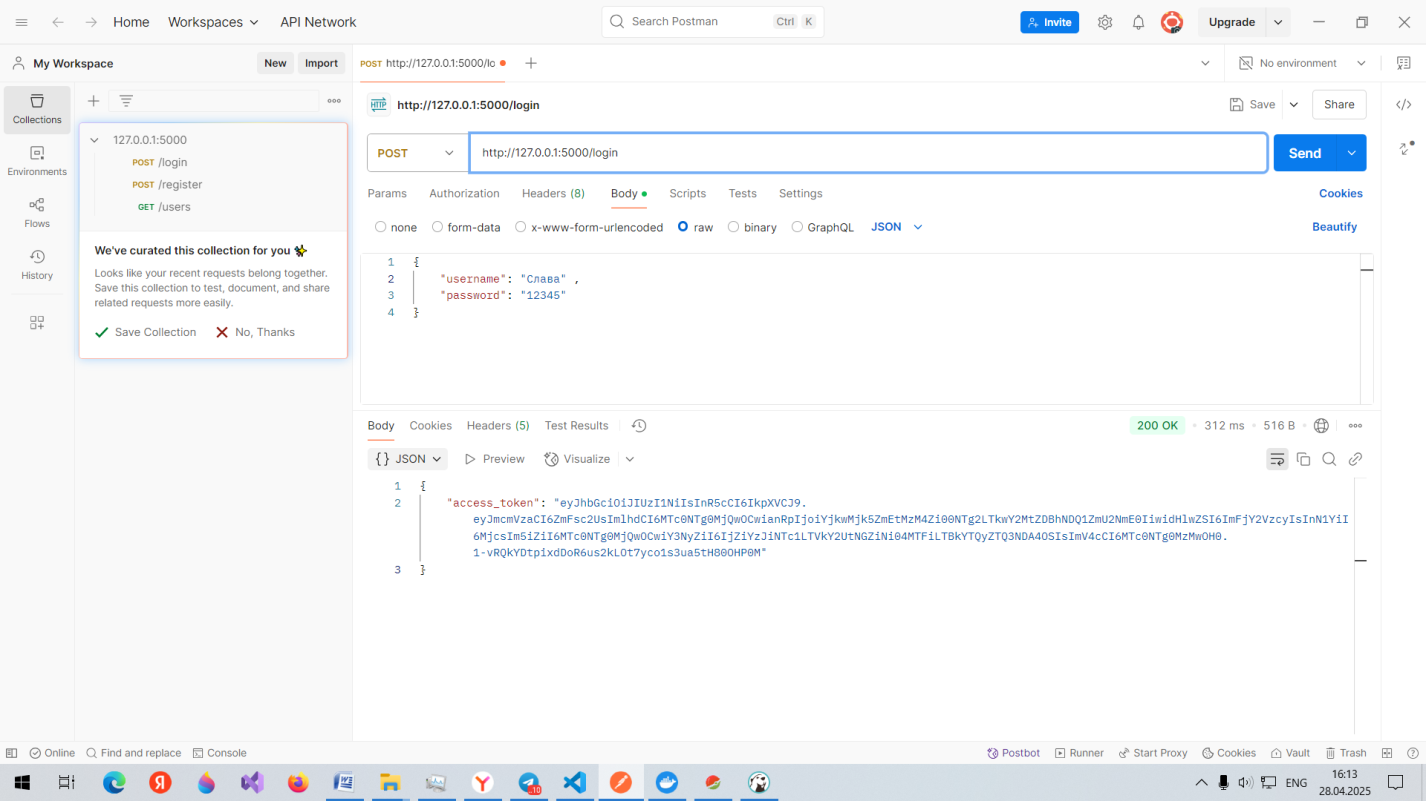
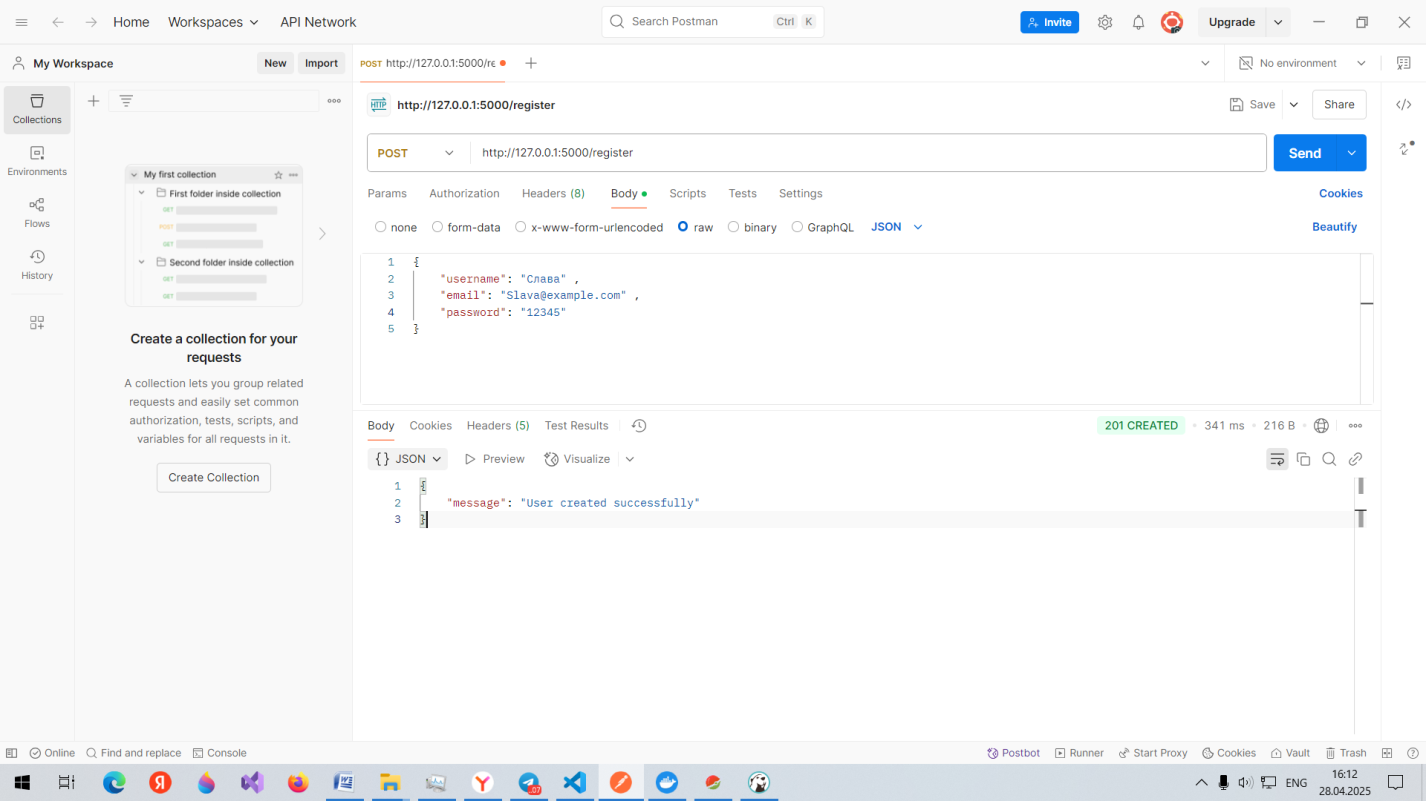
def protected():

current\_user\_id = get\_jwt\_identity() # Получаем ID текущего пользователя из токена

user = User.query.get(current\_user\_id)

return jsonify(logged\_in\_as=user.username)

1. **Примеры**



# Контрольная точка 4

1. **Код API**

fromflaskimportFlask, request, render\_template, redirect, url\_for, session, flash

fromflask\_sqlalchemyimportSQLAlchemy

fromwerkzeug.securityimportgenerate\_password\_hash, check\_password\_hash

fromdatetimeimportdatetime

app=Flask(\_\_name\_\_)

app.secret\_key='your\_super\_secret\_key'

app.config['SQLALCHEMY\_DATABASE\_URI'] ='postgresql://user:password@localhost:5432/messenger'

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] =False

db=SQLAlchemy(app)

classUser(db.Model):

    \_\_tablename\_\_='users'

    id=db.Column(db.Integer, primary\_key=True)

    username=db.Column(db.String(80), unique=True, nullable=False)

    email=db.Column(db.String(120), unique=True, nullable=False)

    password\_hash=db.Column(db.String(128))

classChat(db.Model):

    \_\_tablename\_\_='chats'

    id=db.Column(db.Integer, primary\_key=True)

    name=db.Column(db.String, nullable=False)

classMessage(db.Model):

    \_\_tablename\_\_='messages'

    id=db.Column(db.Integer, primary\_key=True)

    chat\_id=db.Column(db.Integer, db.ForeignKey('chats.id'))

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

    content=db.Column(db.String, nullable=False)

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

classChatMember(db.Model):

    \_\_tablename\_\_='chat\_members'

    id=db.Column(db.Integer, primary\_key=True)

    chat\_id=db.Column(db.Integer, db.ForeignKey('chats.id'), nullable=False)

    user\_id=db.Column(db.Integer, db.ForeignKey('users.id'), nullable=False)

@app.route('/')

defhome():

    returnredirect(url\_for('login\_page'))

@app.route('/register-page', methods=['GET', 'POST'])

defregister\_page():

    ifrequest.method=='POST':

        username=request.form.get('username')

        email=request.form.get('email')

        password=request.form.get('password')

        ifnotusernameornotemailornotpassword:

            flash("Все поля обязательны для заполнения.")

            returnrender\_template('register.html')

        ifUser.query.filter((User.username==username) | (User.email==email)).first():

            flash("Пользователь с таким логином или email уже существует.")

            returnrender\_template('register.html')

        hashed\_password=generate\_password\_hash(password)

        new\_user=User(username=username, email=email, password\_hash=hashed\_password)

        db.session.add(new\_user)

        db.session.commit()

        flash("Успешная регистрация. Войдите в систему.")

        returnredirect(url\_for('login\_page'))

    returnrender\_template('register.html')

@app.route('/login-page', methods=['GET', 'POST'])

deflogin\_page():

    ifrequest.method=='POST':

        username=request.form.get('username')

        password=request.form.get('password')

        user=User.query.filter\_by(username=username).first()

        ifnotuserornotcheck\_password\_hash(user.password\_hash, password):

            flash("Неправильный логин или пароль.")

            returnrender\_template('login.html')

        session['user\_id'] =user.id

        returnredirect(url\_for('chats\_page'))

    returnrender\_template('login.html')

@app.route('/logout-page')

deflogout\_page():

    session.clear()

    returnredirect(url\_for('login\_page'))

@app.route('/chats-page', methods=['GET', 'POST'])

defchats\_page():

    user\_id=session.get('user\_id')

    ifnotuser\_id:

        returnredirect(url\_for('login\_page'))

    ifrequest.method=='POST':

        chat\_name=request.form.get('name')

        ifchat\_name:

            new\_chat=Chat(name=chat\_name)

            db.session.add(new\_chat)

            db.session.commit()

            chat\_member=ChatMember(chat\_id=new\_chat.id, user\_id=user\_id)

            db.session.add(chat\_member)

            db.session.commit()

            initial\_message=Message(chat\_id=new\_chat.id, sender\_id=user\_id, content="Чат создан")

            db.session.add(initial\_message)

            db.session.commit()

            flash("Чат создан.")

        else:

            flash("Имя чата не задано.")

        returnredirect(url\_for('chats\_page'))

    chats= (

        db.session.query(Chat)

        .join(ChatMember, Chat.id==ChatMember.chat\_id)

        .filter(ChatMember.user\_id==user\_id)

        .all()

    )

    returnrender\_template('chats.html', chats=chats)

@app.route('/chats/<int:chat\_id>/leave', methods=['POST'])

defleave\_chat(chat\_id):

    user\_id=session.get('user\_id')

    ifnotuser\_id:

        returnredirect(url\_for('login\_page'))

    # Найдём пользователя

    user=User.query.get(user\_id)

    ifnotuser:

        flash("Пользователь не найден.")

        returnredirect(url\_for('chats\_page'))

    # Удалим участника из чата

    ChatMember.query.filter\_by(chat\_id=chat\_id, user\_id=user\_id).delete()

    db.session.commit()

    # Добавим сообщение о выходе пользователя

    leave\_message=Message(chat\_id=chat\_id, sender\_id=user\_id, content=f"{user.username} покинул чат")

    db.session.add(leave\_message)

    db.session.commit()

    # Проверим, остались ли участники

    remaining\_members=ChatMember.query.filter\_by(chat\_id=chat\_id).count()

    ifremaining\_members==0:

        # Удалим все сообщения

        Message.query.filter\_by(chat\_id=chat\_id).delete()

        # Удалим сам чат

        Chat.query.filter\_by(id=chat\_id).delete()

        db.session.commit()

        flash("Чат удалён, так как в нём не осталось участников.")

    else:

        flash("Вы покинули чат.")

    returnredirect(url\_for('chats\_page'))

@app.route('/chats/<int:chat\_id>/messages-page', methods=['GET', 'POST'])

defview\_chat(chat\_id):

    user\_id=session.get('user\_id')

    ifnotuser\_id:

        returnredirect(url\_for('login\_page'))

    ifrequest.method=='POST':

        action=request.form.get('action')

        ifaction=='send\_as\_self':

            content=request.form.get('content')

            ifcontent:

                new\_message=Message(chat\_id=chat\_id, sender\_id=user\_id, content=content)

                db.session.add(new\_message)

                db.session.commit()

        elifaction=='add\_user':

            new\_user\_id\_str=request.form.get('new\_user\_id')

            try:

                new\_user\_id=int(new\_user\_id\_str)

                user=User.query.get(new\_user\_id)

                ifuser:

                    exists=ChatMember.query.filter\_by(chat\_id=chat\_id, user\_id=new\_user\_id).first()

                    ifnotexists:

                        db.session.add(ChatMember(chat\_id=chat\_id, user\_id=new\_user\_id))

                        db.session.commit()

                        db.session.add(Message(chat\_id=chat\_id, sender\_id=user\_id, content=f"{user.username} добавлен"))

                        db.session.commit()

                        flash("Пользователь добавлен.")

                else:

                    flash("Пользователь не найден.")

            except:

                flash("Неверный ID пользователя.")

        returnredirect(url\_for('view\_chat', chat\_id=chat\_id))

    messages= (

        db.session.query(Message, User)

        .join(User, Message.sender\_id==User.id)

        .filter(Message.chat\_id==chat\_id)

        .order\_by(Message.timestamp.asc())

        .all()

    )

    formatted= [

        {'timestamp': m.timestamp.strftime('%Y-%m-%d %H:%M:%S'), 'sender': u.username, 'content': m.content}

        form, uinmessages

    ]

    returnrender\_template('messages.html', messages=formatted, chat\_id=chat\_id)

if\_\_name\_\_=='\_\_main\_\_':

    app.run(debug=True)

1. **Код интерфейса**

**style.css**

body {

    font-family: Arial, sans-serif;

    padding: 20px;

    background-color: #f4f4f4;

}

header {

    background-color: #333;

    color: #fff;

    padding: 10px20px;

}

headerh1 {

    display: inline;

}

nava {

    color: #fff;

    margin-left: 15px;

    text-decoration: none;

}

nava:hover {

    text-decoration: underline;

}

main {

    background-color: #fff;

    padding: 20px;

    margin-top: 10px;

    border-radius: 5px;

}

forminput, formbutton {

    display: block;

    margin: 10px0;

    padding: 10px;

    width: 100%;

    box-sizing: border-box;

}

ul {

    list-style-type: none;

    padding: 0;

}

li {

    margin-bottom: 10px;

}

.flashes {

    background-color: #ffdddd;

    border-left: 6pxsolid#f44336;

    padding: 10px;

    margin-bottom: 15px;

}

**base.html**

<!DOCTYPEhtml>

<htmllang="ru">

<head>

    <metacharset="UTF-8">

    <title>Messenger</title>

    <linkrel="stylesheet"href="{{ url\_for('static', filename='style.css') }}">

</head>

<body>

    <header>

        <h1>Messenger</h1>

        <nav>

            {% ifsession.get('token') %}

                <ahref="{{ url\_for('chats\_page') }}">Чаты</a>

                <ahref="{{ url\_for('logout\_page') }}">Выйти</a>

            {% else %}

                <ahref="{{ url\_for('login\_page') }}">Войти</a>

                <ahref="{{ url\_for('register\_page') }}">Регистрация</a>

            {% endif %}

        </nav>

    </header>

    <main>

        {% withmessages = get\_flashed\_messages() %}

          {% if messages %}

            <ulclass="flashes">

              {% formessageinmessages %}

                <li>{{ message}}</li>

              {% endfor %}

            </ul>

          {% endif %}

        {% endwith %}

        {% blockcontent %}{% endblock %}

    </main>

</body>

</html>

**chats.html**

{% extends "base.html" %}

{% blockcontent %}

<h2>Ваши чаты</h2>

<ul>

    {% forchatinchats %}

        <li>

            <ahref="{{ url\_for('view\_chat', chat\_id=chat['id']) }}">{{ chat['name'] }}</a>

            <!-- Кнопка "Покинуть чат" -->

            <formmethod="POST"action="{{ url\_for('leave\_chat', chat\_id=chat['id']) }}"style="display:inline;">

                <buttontype="submit">Покинуть</button>

            </form>

        </li>

    {% endfor %}

</ul>

<formmethod="POST">

    <inputtype="text"name="name"placeholder="Название нового чата"required>

    <buttontype="submit">Создать чат</button>

</form>

{% endblock %}

**login.html**

{% extends "base.html" %}

{% blockcontent %}

<h2>Вход</h2>

<formmethod="POST">

    <inputtype="text"name="username"placeholder="Имя пользователя"required>

    <inputtype="password"name="password"placeholder="Пароль"required>

    <buttontype="submit">Войти</button>

</form>

{% endblock %}

**messages.html**

{% extends "base.html" %}

{% blockcontent %}

<h2>Сообщения</h2>

<ul>

    {% formessageinmessages %}

        <li>

            {{ message['timestamp'] }} | <b>{{ message['sender'] }}</b>: {{ message['content'] }}

        </li>

    {% endfor %}

</ul>

<formmethod="POST">

    <inputtype="text"name="content"placeholder="Ваше сообщение"required>

    <buttontype="submit"name="action"value="send\_as\_self">Отправить</button>

</form>

<formmethod="POST">

    <inputtype="text"name="new\_user\_id"placeholder="ID пользователя для добавления"required>

    <buttontype="submit"name="action"value="add\_user">Добавить пользователя</button>

</form>

<ahref="{{ url\_for('chats\_page') }}">Назадкчатам</a>

{% endblock %}

**register.html**

{% extends "base.html" %}

{% blockcontent %}

<h2>Регистрация</h2>

<formmethod="POST">

    <inputtype="text"name="username"placeholder="Имя пользователя"required>

    <inputtype="email"name="email"placeholder="Email"required>

    <inputtype="password"name="password"placeholder="Пароль"required>

    <buttontype="submit">Зарегистрироваться</button>

</form>

{% endblock %}

1. **Описание структуры интерфейса и логики работы с API**

Приложение построено на базе Flask и использует SQLAlchemy для взаимодействия с базой данных PostgreSQL. Интерфейс организован через HTML-шаблоны с использованием render\_template, а данные передаются между сервером и клиентом через формы и URL-параметры.

Пользователь начинает с регистрации или входа в систему. После успешной аутентификации ему открывается доступ к интерфейсу чатов. Чаты создаются пользователями вручную, и каждый новый чат автоматически включает создателя в список участников. Добавление новых пользователей в чат возможно по их ID, что реализовано через POST-запрос на ту же страницу сообщений чата.

Логика API включает следующие действия: регистрация пользователя, вход, выход, создание чатов, просмотр списка чатов, переход к сообщениям, отправка сообщений, добавление участников и выход из чата. Если пользователь выходит, и он был последним участником, чат полностью удаляется вместе с сообщениями.

1. **Примеры рендеринга данных и передачи параметров**

Пример передачи чатов в шаблон происходит при отображении страницы /chats-page. Сначала в базе данных выбираются чаты, в которых состоит пользователь, после чего результат передаётся в HTML-шаблон chats.html:

returnrender\_template('chats.html', chats=chats)

Сообщения отображаются аналогично: при заходе в конкретный чат сервер получает список всех сообщений и информацию об отправителях, форматирует их (дата, имя, текст), а затем отправляет в messages.html:

returnrender\_template('messages.html', messages=formatted, chat\_id=chat\_id)

Отправка данных происходит через HTML-формы с методом POST. На странице сообщений можно выбрать действие — отправить сообщение или добавить пользователя, что различается по значению скрытого поля action. Сервер обрабатывает это значение и выполняет соответствующее действие.

1. **Презентация сценария работы приложения**

После запуска пользователь попадает на страницу входа. Если аккаунт ещё не создан, он переходит на форму регистрации, вводит данные, и они сохраняются в базе. Далее он входит в систему и видит список чатов. Новый чат можно создать, указав имя — он сразу появится в списке.

При открытии чата пользователь видит все сообщения, отсортированные по времени. Он может отправить своё сообщение, а также добавить участника по его ID. В чате отображаются имя отправителя, дата и содержание каждого сообщения.

Если пользователь решает выйти из чата, система удаляет его из участников. Если после этого участников не остаётся, удаляются и все связанные данные — сам чат и сообщения. Это поддерживает чистоту данных и автоматическое завершение "пустых" чатов.

# Контрольная точка 5

1. **Код программы**

app.py

from flask import Flask, request, render\_template, redirect, url\_for, session, flash

from flask\_sqlalchemy import SQLAlchemy

from werkzeug.security import generate\_password\_hash, check\_password\_hash

from datetime import datetime

import os

app = Flask(\_\_name\_\_)

app.secret\_key = 'your\_super\_secret\_key'

app.config['SQLALCHEMY\_DATABASE\_URI'] = os.getenv(

    'DATABASE\_URL',

    'postgresql://user:password@localhost:5432/messenger'  # fallback для локального запуска

)

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

db = SQLAlchemy(app)

class User(db.Model):

    \_\_tablename\_\_ = 'users'

    id = db.Column(db.Integer, primary\_key=True)

    username = db.Column(db.String(80), unique=True, nullable=False)

    email = db.Column(db.String(120), unique=True, nullable=False)

    password\_hash = db.Column(db.String(128))

class Chat(db.Model):

    \_\_tablename\_\_ = 'chats'

    id = db.Column(db.Integer, primary\_key=True)

    name = db.Column(db.String, nullable=False)

class Message(db.Model):

    \_\_tablename\_\_ = 'messages'

    id = db.Column(db.Integer, primary\_key=True)

    chat\_id = db.Column(db.Integer, db.ForeignKey('chats.id'))

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

    content = db.Column(db.String, nullable=False)

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

class ChatMember(db.Model):

    \_\_tablename\_\_ = 'chat\_members'

    id = db.Column(db.Integer, primary\_key=True)

    chat\_id = db.Column(db.Integer, db.ForeignKey('chats.id'), nullable=False)

    user\_id = db.Column(db.Integer, db.ForeignKey('users.id'), nullable=False)

@app.route('/')

def home():

    return redirect(url\_for('login\_page'))

@app.route('/register-page', methods=['GET', 'POST'])

def register\_page():

    if request.method == 'POST':

        username = request.form.get('username')

        email = request.form.get('email')

        password = request.form.get('password')

        if not username or not email or not password:

            flash("Все поля обязательны для заполнения.")

            return render\_template('register.html')

        if User.query.filter((User.username == username) | (User.email == email)).first():

            flash("Пользователь с таким логином или email уже существует.")

            return render\_template('register.html')

        hashed\_password = generate\_password\_hash(password)

        new\_user = User(username=username, email=email, password\_hash=hashed\_password)

        db.session.add(new\_user)

        db.session.commit()

        flash("Успешная регистрация. Войдите в систему.")

        return redirect(url\_for('login\_page'))

    return render\_template('register.html')

@app.route('/login-page', methods=['GET', 'POST'])

def login\_page():

    if request.method == 'POST':

        username = request.form.get('username')

        password = request.form.get('password')

        user = User.query.filter\_by(username=username).first()

        if not user or not check\_password\_hash(user.password\_hash, password):

            flash("Неправильный логин или пароль.")

            return render\_template('login.html')

        session['user\_id'] = user.id

        return redirect(url\_for('chats\_page'))

    return render\_template('login.html')

@app.route('/logout-page')

def logout\_page():

    session.clear()

    return redirect(url\_for('login\_page'))

@app.route('/chats-page', methods=['GET', 'POST'])

def chats\_page():

    user\_id = session.get('user\_id')

    if not user\_id:

        return redirect(url\_for('login\_page'))

    if request.method == 'POST':

        chat\_name = request.form.get('name')

        if chat\_name:

            new\_chat = Chat(name=chat\_name)

            db.session.add(new\_chat)

            db.session.commit()

            chat\_member = ChatMember(chat\_id=new\_chat.id, user\_id=user\_id)

            db.session.add(chat\_member)

            db.session.commit()

            initial\_message = Message(chat\_id=new\_chat.id, sender\_id=user\_id, content="Чат создан")

            db.session.add(initial\_message)

            db.session.commit()

            flash("Чат создан.")

        else:

            flash("Имя чата не задано.")

        return redirect(url\_for('chats\_page'))

    chats = (

        db.session.query(Chat)

        .join(ChatMember, Chat.id == ChatMember.chat\_id)

        .filter(ChatMember.user\_id == user\_id)

        .all()

    )

    return render\_template('chats.html', chats=chats)

@app.route('/chats/<int:chat\_id>/leave', methods=['POST'])

def leave\_chat(chat\_id):

    user\_id = session.get('user\_id')

    if not user\_id:

        return redirect(url\_for('login\_page'))

    # Найдём пользователя

    user = User.query.get(user\_id)

    if not user:

        flash("Пользователь не найден.")

        return redirect(url\_for('chats\_page'))

    # Удалим участника из чата

    ChatMember.query.filter\_by(chat\_id=chat\_id, user\_id=user\_id).delete()

    db.session.commit()

    # Добавим сообщение о выходе пользователя

    leave\_message = Message(chat\_id=chat\_id, sender\_id=user\_id, content=f"{user.username} покинул чат")

    db.session.add(leave\_message)

    db.session.commit()

    # Проверим, остались ли участники

    remaining\_members = ChatMember.query.filter\_by(chat\_id=chat\_id).count()

    if remaining\_members == 0:

        # Удалим все сообщения

        Message.query.filter\_by(chat\_id=chat\_id).delete()

        # Удалим сам чат

        Chat.query.filter\_by(id=chat\_id).delete()

        db.session.commit()

        flash("Чат удалён, так как в нём не осталось участников.")

    else:

        flash("Вы покинули чат.")

    return redirect(url\_for('chats\_page'))

@app.route('/chats/<int:chat\_id>/messages-page', methods=['GET', 'POST'])

def view\_chat(chat\_id):

    user\_id = session.get('user\_id')

    if not user\_id:

        return redirect(url\_for('login\_page'))

    if request.method == 'POST':

        action = request.form.get('action')

        if action == 'send\_as\_self':

            content = request.form.get('content')

            if content:

                new\_message = Message(chat\_id=chat\_id, sender\_id=user\_id, content=content)

                db.session.add(new\_message)

                db.session.commit()

        elif action == 'add\_user':

            new\_user\_id\_str = request.form.get('new\_user\_id')

            try:

                new\_user\_id = int(new\_user\_id\_str)

                user = User.query.get(new\_user\_id)

                if user:

                    exists = ChatMember.query.filter\_by(chat\_id=chat\_id, user\_id=new\_user\_id).first()

                    if not exists:

                        db.session.add(ChatMember(chat\_id=chat\_id, user\_id=new\_user\_id))

                        db.session.commit()

                        db.session.add(Message(chat\_id=chat\_id, sender\_id=user\_id, content=f"{user.username} добавлен"))

                        db.session.commit()

                        flash("Пользователь добавлен.")

                else:

                    flash("Пользователь не найден.")

            except:

                flash("Неверный ID пользователя.")

        return redirect(url\_for('view\_chat', chat\_id=chat\_id))

    messages = (

        db.session.query(Message, User)

        .join(User, Message.sender\_id == User.id)

        .filter(Message.chat\_id == chat\_id)

        .order\_by(Message.timestamp.asc())

        .all()

    )

    formatted = [

        {'timestamp': m.timestamp.strftime('%Y-%m-%d %H:%M:%S'), 'sender': u.username, 'content': m.content}

        for m, u in messages

    ]

    return render\_template('messages.html', messages=formatted, chat\_id=chat\_id)

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(host="0.0.0.0", port=5000)

docker-compose.yml

services:

  db:

    image: postgres:latest

    container\_name: messenger\_db

    restart: always

    environment:

      POSTGRES\_DB: messenger

      POSTGRES\_USER: user

      POSTGRES\_PASSWORD: password

    ports:

      - "5432:5432"

    volumes:

      - pg\_data:/var/lib/postgresql/data

  web:

    build: .

    container\_name: messenger\_web

    restart: always

    depends\_on:

      - db

    ports:

      - "5431:5000"

    environment:

      - FLASK\_APP=app.py

      - FLASK\_RUN\_HOST=0.0.0.0

      - DATABASE\_URL=postgresql://user:password@db:5432/messenger

    volumes:

      - .:/app

volumes:

  pg\_data:

dockerfile

# Базовый образ Python

FROM python:3.10-slim

# Устанавливаем рабочую директорию внутри контейнера

WORKDIR /app

# Копируем зависимости

COPY requirements.txt .

# Устанавливаем зависимости

RUN pip install --no-cache-dir -r requirements.txt

# Копируем всё приложение внутрь контейнера

COPY . .

# Открываем порт (если нужно)

EXPOSE 5000

# Запуск приложения

CMD ["python", "app.py"]

1. **Структура контейнеризации и настройка окружения**

Приложение развернуто с использованием Docker Compose и состоит из двух основных сервисов:

|  |  |  |
| --- | --- | --- |
| **Сервис** | **Образ** | **Назначение** |
| db | postgres:latest | Хранение данных пользователей, чатов, сообщений |
| web | Кастомный (FROM python:3.10-slim) | Flask-приложение (веб-интерфейс) |

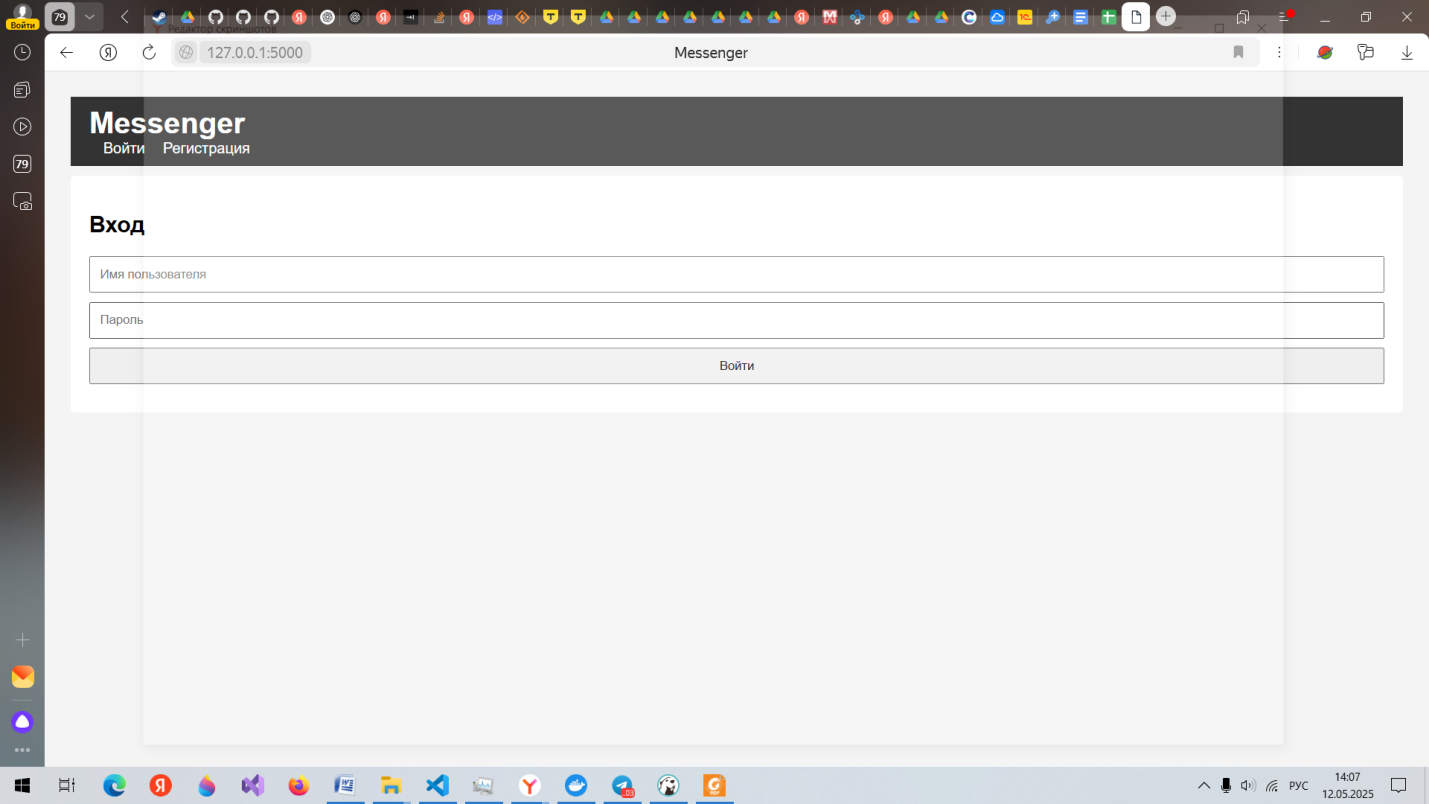
**Файлы:**

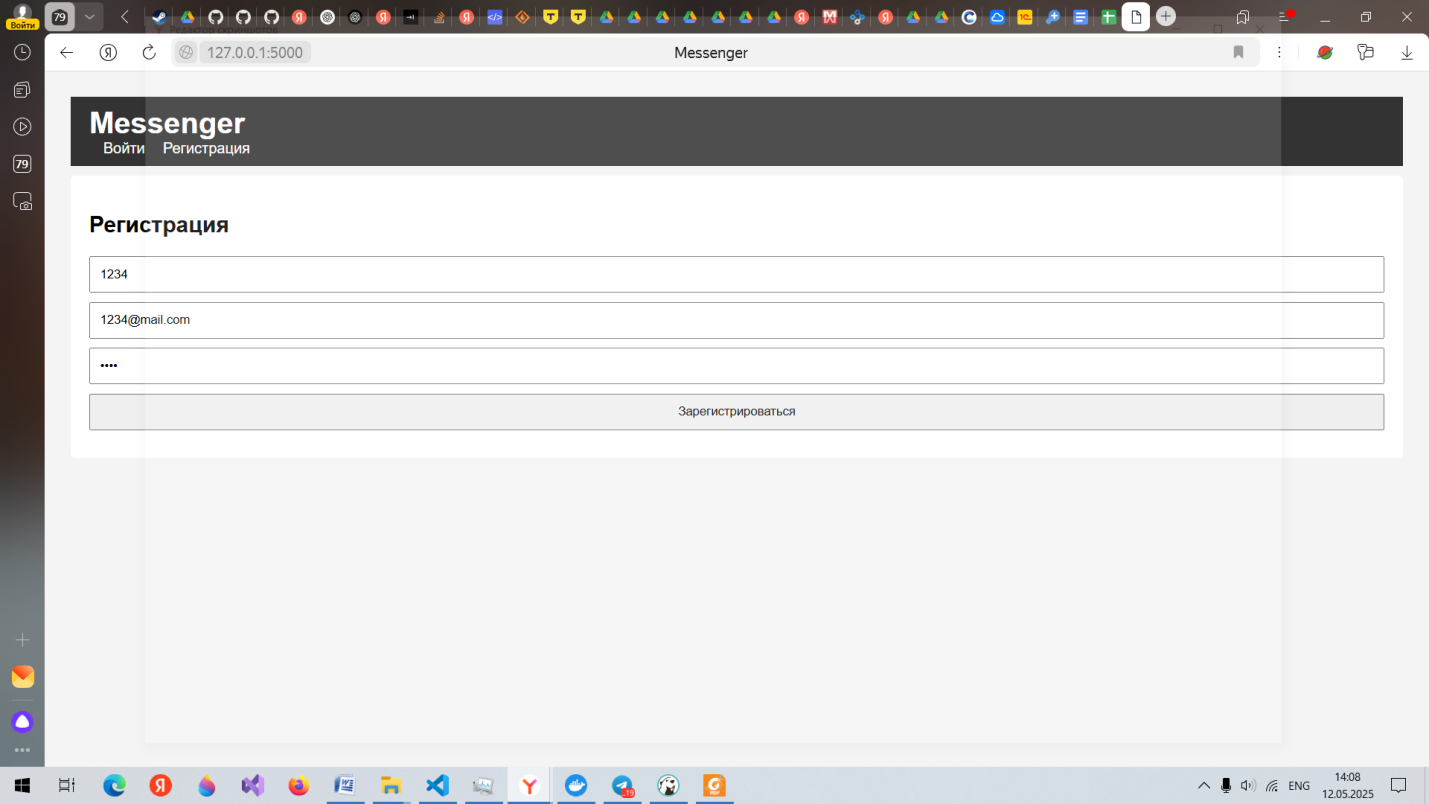
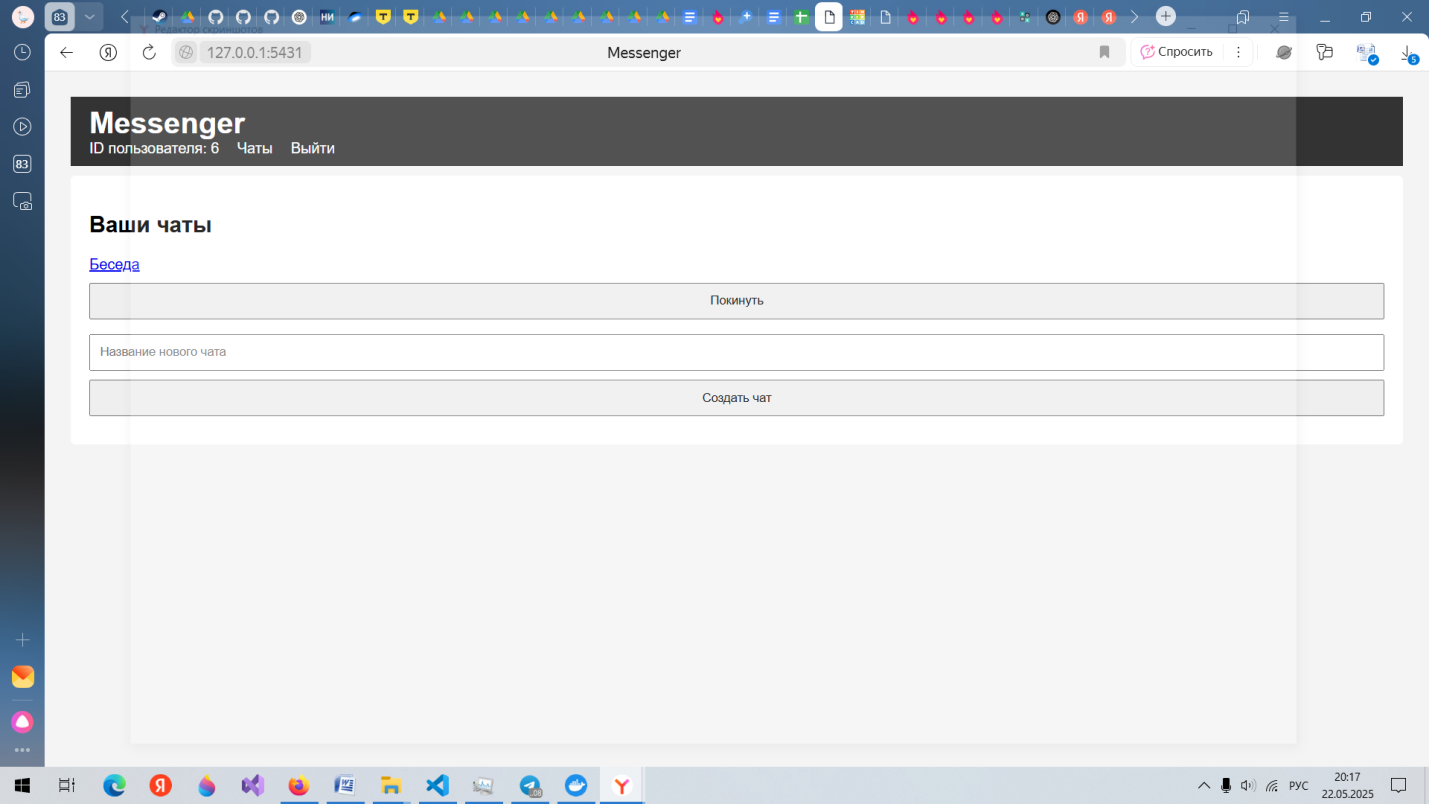
* docker-compose.yml: описывает оба сервиса, монтирует том для PostgreSQL.
* Dockerfile: создает образ Flask-приложения.
* requirements.txt: содержит зависимости (Flask, SQLAlchemy, psycopg2 и др.).

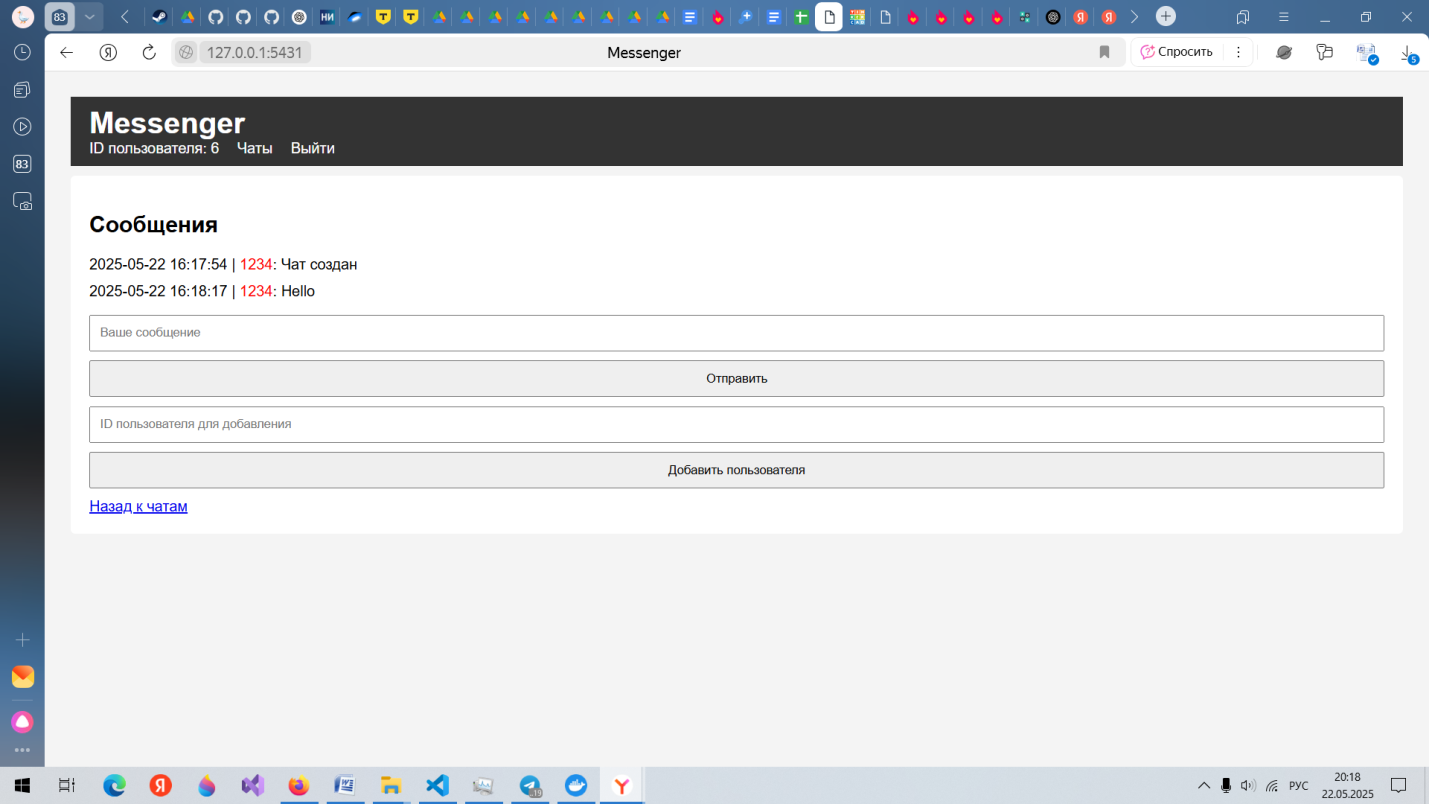
**Пример запуска:**

docker-compose up --build

1. **Скриншоты работающего приложения в контейнерах**





1. **Описание API (в формате таблицы)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Маршрут** | **Метод** | **Назначение** | **Аутентификация** |
| / | GET | Редирект на /login-page | нет |
| /register-page | GET/POST | Регистрация нового пользователя | нет |
| /login-page | GET/POST | Вход в систему | нет |
| /logout-page | GET | Выход из системы | нет |
| /chats-page | GET/POST | Просмотр чатов / создание нового | да |
| /chats/<chat\_id>/leave | POST | Покинуть чат | да |
| /chats/<chat\_id>/messages-page | GET/POST | Просмотр сообщений и взаимодействие | да |

1. **Архитектура разработанного приложения**

**Основные компоненты:**

* Flask (Backend) — веб-приложение, обрабатывающее маршруты, формы и шаблоны.
* Jinja2 (Frontend) — серверная генерация HTML.
* PostgreSQL — СУБД для хранения данных.
* SQLAlchemy — ORM для работы с БД.

**Основные сущности:**

* User: хранит имя пользователя, email и хеш пароля.
* Chat: групповые чаты.
* Message: сообщения, привязанные к чатам.
* ChatMember: связующая таблица many-to-many между пользователями и чатами.

**Поток взаимодействия:**

1. Регистрация / логин.
2. Создание чатов, автоматическое добавление автора.
3. Отправка сообщений.
4. Добавление участников по ID.
5. Удаление чата при выходе последнего участника.