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:**



