МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«ИРКУТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «ИГУ»)

Институт математики и информационных технологий

Кафедра информационных технологий

ОТЧЕТ

о курсовой работе по курсу «Разработка WEB-приложений»

Разработка REST API для мессенджера

Студента 3 курса группы 2371–ДБ
Чехова Александра Игоревича
Направление : 02.03.02– Фундаментальная информатика и информационные технологии

Руководитель:
канд. техн. наук доцент
Черкашин Евгений Александрович

Курсовая работа защищена с оценкой

_____

Иркутск – 2021

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной курсовой работы была поставлена разработка веб–приложения — приложение, клиентом которой будет страница в браузере (клиент). Конкретной темой и целью для приложения был выбран мессенджер (М). Суть М состоит в предоставлении возможности обмена текстовыми сообщениями между пользователями.

Такому приложению необходимо постоянно взаимодействовать с сервером: получать и отправлять данные. Для создаваемого М разработан REST API, позволяющий взаимодействовать с базой данных (БД) путём получения запросов от клиента. В данном отчёте представлены результаты разработки именно этой части приложения.

Учитывая учебную направленность данной работы, были сформулированы следующие требования к серверному приложению:

- Смоделировать интерфейс допустимых запросов (API);
- Спроектировать и создать БД;
- Смоделировать и задать необходимые запросы к БД;
- Разработать обработку запросов от клиентов (маршрутизацию);

Так же, исходя из поставленных задач необходимо выбрать БД и фреймворк для реализации маршрутизации. Фреймворк должен обеспечить удобное и лёгкое создание серверного приложения, так как целевое приложение не имеет особых требований к API. БД в свою очередь должна предоставить удобный формат хранения данных. Так как это курсовая работа в рамках обучения, в качестве указанных выше программных инструментов были выбраны те, что рассчитаны на малую нагрузку — число обрабатываемых запросов.

1 Теоретические основы разработки REST API

Большинство приложений использующих интернет сети полагаются на сервера — компьютеры, позволяющие использовать свои ресурсы (аппаратные мощности) и данные другим компьютерам в локальной или глобальной сети. Такую архитектуру часто определяют как клиент–серверной: клиент совершает запросы к серверу и получает ответы в виде данных и/или подтверждений записи данных из запроса [1].

Для обработки запросов на сервере могут создаваться как большие изолированные программы, так и маленькие, предназначенные для простых целей, при чём вторые могут запускаться как отдельно, так и в составе первых.

Для обработки запросов такого приложения как мессенджер будет достаточно и небольшого приложения. Данное приложение должно позволять:

- считывать получаемые компьютером по сети Итернет запросы и приходящие с ними данные,
- легко подключаться к базе данных (БД),
- отправлять ответные данные клиенту и статус коды — зарезервированные числовые комбинации, несущие информацию о состоянии обработки запроса,

В качестве запросов выступают посланные данные на сервер данные по определённому протоколу — HTTP[2]. Сервер считывает эти данные и направляет данные к приложению, которому они назначаются, а тот в свою очередь схожим образом отправляет ответ клиенту. Во многих случаях запросы схожи с ссылками на сайты, которые пользователи вводят в адресной строке браузера. Такое поведение обеспечивают DNS–сервера, которые производят связывание текстовых ссылок с определёнными компьютерами, и серверами в частности. Именно эту возможность и будет использовать разрабатываемое REST API приложение — по своей сути оно является отражением этой идеи на практике: приложение позволяет делать к себе HTTP-запросы и выдаёт ответы.

Сама программа представляет из себя множество методов, которые обрабатывают определённые команды, приходящие в запросе клиента. Такие программы предоставляют определённый интерфейс воздействия на них и их состояние, из-за чего клиент должен знать как именно нужно делать запросы, чтобы программа выполнила их на сервере и ответила необходимыми данными.

Так же серверные приложения часто используют базы данных для хранения информации пользователя, о пользователе или просто для выдачи её при необходимых условиях (оплате, членстве клуба и т. п.). Одним из видов БД являются SQL подобные БД. Такие БД хранят ифнформацию в виде таблиц и поддерживают различные отложения между записями в таблицах.

Между приложением и БД так же необходима программная связь которая обеспечит удобный интерфейс добавления данных в БД и их чтения от туда же. Это необходимо потому, что большинство разработчиков БД стремятся к улучшению функциональности БД, а следственно и добавлению особенностей. Программа–коннектор обеспечит обработку всех особенностей выбранной БД.

После изучения предметной области — приложение–мессенджер, выделены основные операции производимые пользователями:

- Отправка сообщения,
- Получение сообщений из диалога с одним из других пользователей,
- Добавление новых пользователей в список контактов,
- Авторизация.

Также отдельно стоит упомянуть операцию регистрацию — нерегулярное действие, но требующее всей ответственности работы с запросом и БД.

Реализацию операций описанных выше можно считать требованиями к проекту — всему веб–приложению в целом.

2 Реализация REST API Приложения

2.1 Моделирование API

После изучения требований к веб–приложению было необходимо задать виды ссылок, по которым клиентская часть приложения будет делать запросы на серверную. В результате заданы следующие ссылки:

- `/get_massages` — получение сообщений диалога,
- `/getuser` — получение информации о пользователи для создания нового диалога,
- `/send_massage` — отправка сообщения,
- `/login` — для авторизации,
- `/register` — регистрация,
- `/renew_name` — изменение имени,
- `/renew_avatar` — изменение аватара (картинки в профиле).

Последние два пункта позволяют клиенту реализовать функционал, повышающий удобство использования приложения: человеку проще идентифицировать себя и других людей картинкой и именем, которые они могут менять под себя.

2.2 Проектирование БД

В системе мессенджера имеются две сущности: пользователи и сообщения. Так же необходимо хранить связи пользователей друг с другом, как участников диалогов. В связи с этими условиями была выбрана сдледующая архитектура БД:

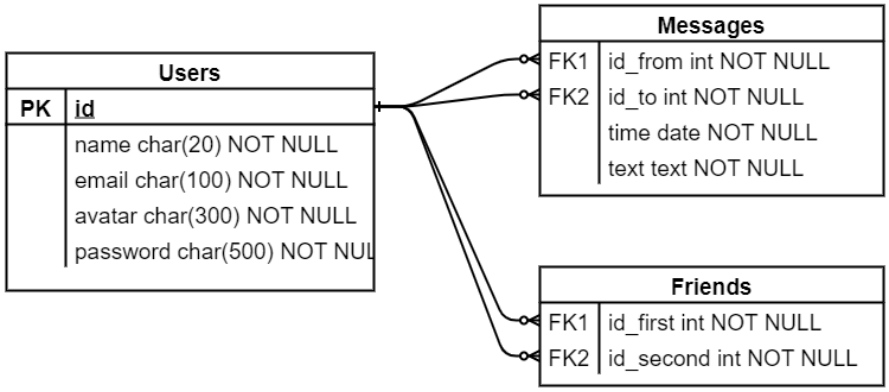

Рисунок 2.1 — Диаграмма отношений таблиц базы данных

В качестве базы данных была выбрана SQLite. Эта БД представляет из себя просто файл, который можно использовать как SQL-подобную БД. Из-за этого SQLite является простой в настройке, малой по объёму занимаемой памяти, но в качестве минуса скорость работы — что не выходит за рамки требований к учебному проекту.

2.3 Язык программирования и его библиотеки

В качестве языка программирования для разработки приложения был выбран Python [3]. Под этот язык разработано много удобных и мощных библиотек, помогающих решать большие задача, без необходимости писать большие объёмы кода.

2.4 Основной фреймворк для создания приложения

В качестве основы выбран легковесный фреймворк `Flask` [4].

`Flask` — это фреймворк, позволяющий создать простое серверное приложение с базовым функционалом:

- Обработка `HTTP` запросов,
- Подключение к БД,
- Возврат по значений из БД.

При создании простого REST API для мессенджера этого будет вполне достаточно.

Во `Flask`, первым делом, создаётся объект приложения (см. приложение листинг 2). Также сразу удобно задать в конфиге приложения переменную `SQLALCHEMY_DATABASE_URI`. Такая настройка позволит удобно использовать другую библиотеку (её модифицированную под `Flask` версию).

На данном этапе имеется представление сруктуры БД. Рассмотрим создание её структуры и запросов к ней для более удобного дальнейшего понимания релизации обработчиков запросов.

2.5 Алгоритм порождения структуры базы данных

Главной библиотекой, помогающей работать с БД стара `sqlalchemy`, а точнее `flask_sqlalchemy`. Версия для `Flask` позволяет автоматизировать некоторые постоянные вещи, как ручное подключение к базе данных, создание состояния для каждого запроса и проведение этого состояния через все запросы к самой БД. Но такое упрощение не обходится без ограничений, по этому и обычная `sqlalchemy` понадобится, например при создании сложных запросов (см. приложение листинг 4).

После того, как была задана конфигурационная переменная для приложения, остаётся три шага:

1. Создать объект класса `SQLalchemy` — переменная `db`, передав ей объект приложения,
2. Воссоздать структуру БД в виде соответствия каждой таблице класса, наследующегося от `db.Module`
3. Вызвать метод `db.create_all()` (см. приложение листинг 3), который, ориентируясь на проделанные ранее шаги, создаст базу данных и необходимые таблицы в ней.

В результате выполнения этих шагов в дирректории проекта появился файл `messanger.db`, который и представляет БД.

2.6 Создание запросов к БД

После проектирования БД, нужно установить те операции с данными внутри неё, которые будут проделываться в будущем. Анализируя требования к проекту можно выделить следующие операции:

- Добавление пользователя,
- Добавление сообщения,
- Добавление связи связи двумя пользователям
- Получение информации о пользователе по его `id` / `email`
- Получение сообщений между двумя пользователями
- Получение друзей пользователя — других пользователей, с которыми он связан

Все эти операции разработаны в виде функций, которые уже реализуют запросы к БД (см. приложение листинги 5–8). Все функции реализуют схожий алгоритм:

1. В блоке `try` проделываются необходимые операции: проверки и изъятия информации,
2. В блоке `except` отлавливаются возможые ощибки при работе с БД,
3. В блоке `finaly` закрывается ссеся подключения к БД.

Все представленные функции позволяют при маршрутизации не задумываться о том, как и от куда берутся данные.

2.7 Реализация маршрутизации

Все функции маршрутизации — это функции, обёрнутые в декораторы вида `@app.route()` (см. приложение листинги 9–11). Аргументом такому декораторам передаётся строка, которую будет необходимо прописать клиенту при запросе. Так же можно передать массив методов запросов, которые будет принимать приложение. В итоге получился набор функций которые обрабатывают определённые запросы и выполняет соответствующие действия. Все функции с запросом получают JSON–объекты, из которых извлекают необходимую информацию. При неудаче: некорректные данные, отказе в доступе или информации нет – функции возвращают код 400, иначе код 200.

Функция	Запрос	Выполняемое действие
`register()`	`/register`	Исходя из полученной информации добавляет нового пользователя в БД
`login()`	`/login`	Проверяет верность полученных данных: email и пароль, и возвращает коды 200 если всё верно
`get_messages()`	`/get_messages`	Возвращает все сообщения между двумя пользователями, идентефикаторы которых получены в зпросе
`get_user()`	`/get_user`	Возвращает информацию о пользователе, email которого был получен в запросе
`add_message()`	`/send_message`	Добовляет собщение в БД
`renew_name()`	`/renew_name`	Обновляет имя пользователя на полученное
`renew_avatar()`	`/renew_avatar`	Обновляет аватарку пользователя (сохраняет новыую ссылку на картинку из интернета)

Таблица 2.1 — Функции маршрутизации

Так же имеется метод `shut_down_connection()`, который после обработки каждого запроса закрывает сессию подключения к БЗ (дополнительная мера предосторожности).

ЗАКЛЮЧЕНИЕ

По итогу проделанной работы было разработано веб-приложение представляющее из себя простой мессенджер, и позволяющее пользователям:

- Регистрироваться и авторизоваться в системе,
- Отправлять друг другу сообщения,
- Добавление новых пользователей в список контактов,
- Изменять имя и аватарку в профиле.

В частности серверная часть приложения предоставляется API для обработки соответсвующих запросов с клиента.

Созданное веб-приложение можно доработать следующими пунктами:

- Получение сообщения без необходимости перезагрузки всего диалога
- Возможность отправки файлов (картинок, аудио и др.)
- Возможность удаления сообщений и контактов
- Отправку уведомлений в браузере
- Зашифрованная передача сообщений
- Наличие отдельных таблиц в базе данных (БД) для хранения персональных данных

По результатам работы над курсовым проектом был получен опят разработки REST API приложения при помощи языка программирования `Python` и библиотеки `Flask`. Также была спроектирована БД и изучена работа с библиотекой `SQLalchemy`, позволяющей воссоздавать структуру БД из создаваемых `Python`–классов и делать различные запросы к БД.

СПИСОК ИПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Снейдер Й. Эффективное программирование TCP/IP: Пер. с англ. – М.: ДМК Пресс.– 320 с.
2. Поллард Б. HTTP/2 в действии / пер. с анг. П. М. Бомбаковой.– М.: ДМК Пресс, 2021.– 424 с.
3. Бэрри, Пол Изучаем программирование на Python / пер. с англ. М. А. Райтман.– М.: Эскимо, 2020.– 624 с.
4. Гринберг М. Разработка веб-приложений с использованием Flask на языке Python / пер. с анг. А. Н. Киселева.– М.: ДМК Пресс, 2014.– 272 с.

ПРИЛОЖЕНИЕ А Исходный код программ

```
DB_CONFIG = {               # it was for PostgreSQL
    'user': 'postgres',
    'password': 'su_12345',
    'host': '127.0.0.1',
    'port': '5432',
    'database': 'messeger'}
SQLITE_FILE_NAME = 'messanger.db'
DEBUG = True
DEFAULT_AVATAR = 'https://vraki.net/sites/default/files/inline/images/30_55.jpg'
```

Listing 1: Файл `config.py`

```
from config import SQLITE_FILE_NAME
from flask import Flask
app = Flask(__name__)
#app.config['SQLALCHEMY_DATABASE_URI'] = \
#    "postgresql+psycopg2://{user}:{password}@{host}:{port}/{database}".format(**DB_CONFIG)
app.config['SQLALCHEMY_DATABASE_URI'] = f"sqlite:///{SQLITE_FILE_NAME}"
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```

Listing 2: Файл `application.py`

```
import db_operations
from database import db
from application import app
from config import DEBUG
from flask_cors import CORS
import routing
db_operations.create_db()
db.create_all()
if DEBUG:
    client = app.test_client()
CORS(app)
if __name__ == '__main__':
    app.run(debug=DEBUG)
```

Listing 3: Файл `start.py`

```
from application import app
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import and_, or_
from sqlalchemy.sql import func
from datetime import datetime
from werkzeug.security import generate_password_hash, check_password_hash
from config import DEFAULT_AVATAR

db = SQLAlchemy(app)


class Users(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(20), nullable=False)
    email = db.Column(db.String(100), nullable=False, unique=True)
    avatar = db.Column(db.String(300), nullable=True)
    password = db.Column(db.String(500), nullable=False)

    def as_dict(self):
        return {'id': self.id,
                'avatar': self.avatar if self.avatar else DEFAULT_AVATAR,
                'name': self.name}

    def __repr__(self):
        return f"<users {self.id}>"


class Friends(db.Model):
    first_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                         primary_key=True)
    second_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                          primary_key=True)

    def __repr__(self):
        return f"<friends {self.first_id} : {self.second_id}>"


class Messages(db.Model):
    id_from = db.Column(db.Integer, db.ForeignKey('users.id'),
                        primary_key=True)
    id_to = db.Column(db.Integer, db.ForeignKey('users.id'),
                      primary_key=True)
    time = db.Column(db.DateTime(timezone=False), primary_key=True,
                     default=func.now())
    text = db.Column(db.Text, nullable=False)

    def as_dict(self):
        return {'id_from': self.id_from,
                'id_to': self.id_to,
                'time': self.time.timestamp(),
                'text': self.text}

    def __repr__(self):
        return f"<Messages {self.id_from} -> {self.id_to}>"
```

Listing 4: Файл `database.py`, часть 1

```
def add_user(name, email, password):
    try:
        if (Users.query.filter(Users.email == email)).all():
            return False
        u = Users(name=name, email=email,
                  password=generate_password_hash(password))
        db.session.add(u)
        db.session.flush()
        db.session.commit()
        return True
    except Exception as e:
        db.session.rollback()
        print(e)
        print('Не удалось добавить пользователя')
        return False
    finally:
        db.session.remove()


def add_message(id_from, id_to, text):
    try:
        if len(Users.query.filter(or_(Users.id == id_from,
                                      Users.id == id_to)).all()) != 2:
            return False

        m = Messages(id_from=id_from, id_to=id_to,
                     text=text)
        db.session.add(m)
        db.session.flush()
        db.session.commit()
        return True
    except Exception as e:
        db.session.rollback()
        print(e)
        print('Не удалось добавить сообщение')
        return False
    finally:
        db.session.remove()
```

Listing 5: Файл `database.py`, часть 2

```
def add_friends(first_id, second_id):
    try:
        if len(Users.query.filter(or_(Users.id == first_id,
                                      Users.id == second_id)).all()) != 2 or \
                len(
                    Friends.query.filter(or_(
                        and_(Friends.first_id == first_id, Friends.first_id == second_id),
                        and_(Friends.first_id == second_id, Friends.first_id == first_id))
                    ).all()
                ) != 0:
            return False

        f = Friends(first_id=first_id, second_id=second_id)
        db.session.add(f)
        db.session.flush()
        db.session.commit()
    except Exception as e:
        db.session.rollback()
        print(e)
        print('Не удалось добавить сообщение')
        return False
    finally:
        db.session.remove()


def renew_name(user_id, new_name):
    try:
        user = Users.query.get(user_id)
        if not user:
            return False
        user.name = new_name
        db.session.commit()
        return True
    except Exception as e:
        print(e)
        print('Не удалось обновить имя')
        return False
    finally:
        db.session.remove()
```

Listing 6: Файл `database.py`, часть 3

```
def renew_avatar(user_id, new_avatar):
    try:
        user = Users.query.get(user_id)
        if not user:
            return False
        user.avatar = new_avatar
        db.session.commit()
        return True
    except Exception as e:
        print(e)
        print('Не удалось обновить фото')
        return False
    finally:
        db.session.remove()


def get_user(user_in_list):
    try:
        if not user_in_list:
            return False
        return user_in_list[0]
    except Exception as e:
        print(e)
        print('Не удалось получить пользователя')
    finally:
        db.session.remove()

def get_user_by_email(email):
    return get_user(Users.query.filter(Users.email == email).all())


def get_user_by_id(user_id):
    return  get_user(Users.query.filter(Users.id == user_id).all())


def get_messages(id_1, id_2):
    try:
        if len(Users.query.filter(or_(Users.id == id_1,
                                      Users.id == id_2)).all()) != 2:
            return []
        return Messages.query.filter(or_(
            and_(Messages.id_from == id_1, Messages.id_to == id_2),
            and_(Messages.id_from == id_2, Messages.id_to == id_1))
        ).order_by(Messages.time).all()
    except Exception as e:
        print(e)
        print('Не удалось получить сообщения')
        return []
    finally:
        db.session.remove()
```

Listing 7: Файл `database.py`, часть 4

```
def get_friends(user_id):
    try:
        if not Users.query.filter(Users.id == user_id).all():
            return []
        return Friends.query.filter(or_(Friends.first_id == user_id,
                                    Friends.second_id == user_id)).all()
    except Exception as e:
        print(e)
        print('Не удалось получить друзей')
        return []
    finally:
        db.session.remove()


def check_user(email, password):
    try:
        if len(Users.query.filter(Users.email == email).all()) < 1:
            return False
        return check_password_hash(
            Users.query.filter(Users.email == email).one().password,
            password)
    except Exception as e:
        print(e)
        print('Не удалось получить друзей')
        return []
    finally:
        db.session.remove()
```

Listing 8: Файл `database.py`, часть 5

```
from application import app
import database as db
from flask import request, make_response


@app.route('/get_messages', methods=['POST'])
def get_messages():
    try:
        id_first = request.get_json().get('myId')
        id_second = request.get_json().get('userId')
        messages = [m.as_dict() for m in db.get_messages(id_first, id_second)]
        status_code = 200
    except KeyError:
        messages = []
        status_code = 400
    response = make_response({'messages': messages}, status_code)
    response.headers.add('Access-Control-Allow-Origin', '*')
    return response


@app.route('/get_user', methods=['POST'])
def get_user():
    try:
        email = request.get_json().get('email')
        user_id = request.get_json().get('id')
        friend = db.get_user_by_email(email)
        if friend:
            friend = friend.as_dict()
            db.add_friends(user_id, friend['id'])
            status_code = 200
        else:
            friend = {}
            status_code = 400
    except KeyError:
        friend = {}
        status_code = 400
    response = make_response(friend, status_code)
    response.headers.add('Access-Control-Allow-Origin', '*')
    return response
```

Listing 9: Файл `routing.py`, часть 1

```
@app.route('/send_message', methods=['POST'])
def add_message():
    try:
        id_form = request.get_json().get('id_from')
        id_to = request.get_json().get('id_to')
        text = request.get_json().get('text')
        status_code = 200 if db.add_message(id_form, id_to, text) else 400
    except KeyError:
        status_code = 400
    response = make_response({}, status_code)
    response.headers.add('Access-Control-Allow-Origin', '*')
    return response


@app.route('/login', methods=['POST'])
def login():
    ret = {}
    try:
        req_json = request.get_json()
        email = req_json.get('email')
        password = req_json.get('password')
        status_code = 200 if db.check_user(email, password) else 400
        if status_code == 200:
            user = db.get_user_by_email(email).as_dict()
            user_id = user['id']
            pairs = db.get_friends(user_id)
            if pairs:
                ids1, ids2 = zip(*[[p.first_id, p.second_id] for p in pairs])
                ids = set(ids1 + ids2)
                friends = [db.get_user_by_id(fid).as_dict()
                           for fid in ids if fid != user_id]
            else:
                friends = []
            ret = {'user': user,
                   'friends': friends}
    except KeyError:
        status_code = 400
    response = make_response(ret, status_code)
    response.headers.add('Access-Control-Allow-Origin', '*')
    return response
```

Listing 10: Файл `routing.py`, часть 2

```
@app.route('/register', methods=['POST'])
def register():
    try:
        req_json = request.get_json()
        name = req_json.get('name')
        email = req_json.get('email')
        password = req_json.get('password')
        status_code = 200 if db.add_user(name, email, password) else 400
    except KeyError:
        status_code = 400
    response = make_response({}, status_code)
    response.headers.add('Access-Control-Allow-Origin', '*')
    return response


@app.route('/renew_name', methods=['POST'])
def renew_name():
    try:
        req_json = request.get_json()
        user_id = req_json.get('id')
        new_name = req_json.get('name')
        status_code = 200 if db.renew_name(user_id, new_name) else 400
    except KeyError:
        status_code = 400
    response = make_response({}, status_code)
    response.headers.add('Access-Control-Allow-Origin', '*')
    return response


@app.route('/renew_avatar', methods=['POST'])
def renew_avatar():
    try:
        req_json = request.get_json()
        user_id = req_json.get('id')
        new_avatar = req_json.get('avatar')
        status_code = 200 if db.renew_avatar(user_id, new_avatar) else 400
    except KeyError:
        status_code = 400
    response = make_response({}, status_code)
    response.headers.add('Access-Control-Allow-Origin', '*')
    return response


@app.teardown_appcontext
def shut_down_connection(exception=None):
    if exception:
        print(exception)
    db.db.session.remove()
```

Listing 11: Файл `routing.py`, часть 3