Лабораторная работа №4

Создание и подключение базы данных к веб-приложению. Разработка REST-методов взаимодействия с базой данных веб-приложения.

Задание:

1) Создать базу данных (БД) SQLite с таблицей, которая будет хранить данные, отправляемые с веб-формы вашего приложения. Таблица должна иметь не менее 4-х полей. Процедура создания таблицы должна быть сохранена в отдельный скрипт *.sql.

2) Подключить БД к приложению и реализовать CRUD-методы для работы с таблицей, т.е. методы должны использовать следующие типы SQL-запросов: SELECT (с фильтром и без), INSERT, UPDATE, DELETE.

3) Разработать REST API и привязать маршруты (routes) вашего веб-приложения к соответствующим CRUD-методам. При отправке данных с вашей веб-формы методом POST должны добавляться данные в таблицу, используйте отправку данных на основе подхода AJAX из ЛР №3.

4) Провести тестирование разработанного REST API с помощью любого REST-клиента.

Теоретический материал:

1) Понятие CRUD-методов, особенности БД SQLite (основные преимущества и ограничения).

2) Ознакомиться с протоколом HTTP (формат передачи данных, основные коды возвращаемых ошибок).

3) Ознакомиться с понятием REST API (наиболее часто используемые HTTP-методы, понятие URI, клиент-серверное взаимодействие).

Рекомендуемое программное обеспечение:

1) DBeaver CE – универсальный клиент для работы с БД:

https://dbeaver.io/files/dbeaver-ce-latest-x86_64-setup.exe

2) **Insomnia** или **Postman**- утилиты для тестирования REST API:

https://insomnia.rest/download

Методические рекомендации

1. Создание БД SQLite.

Если в разрабатываемом веб-приложении необходимо использовать БД, то реализация спроектированной структуры (реляционной) БД возможна следующими способами:

1) С помощью языка SQL.

2) С помощью технологии «объектно-реляционного отображения» - Object-Relational Mapping (ORM), которая используется в современных языках программирования, поддерживающих объектно-ориентированный подход (Python, Node.js, Java, C# и т.п.).

В рамках данного практического курса рассмотрим первый вариант реализации структуры БД.

БД SQLite является встраиваемой БД, часто применяемой при локальном тестировании приложений в процессе разработки, а также нередко используется в рабочих решениях в случае, когда основная масса запросов к БД – это запросы на чтение.

Для инициализации БД SQLite создайте пустой текстовый файл и измените его имя и расширение, например: **`appdb.sqlite`** . В менеджере БД DBeaver откройте выпадающее меню для нового подключения и выберите подключение к SQLite:

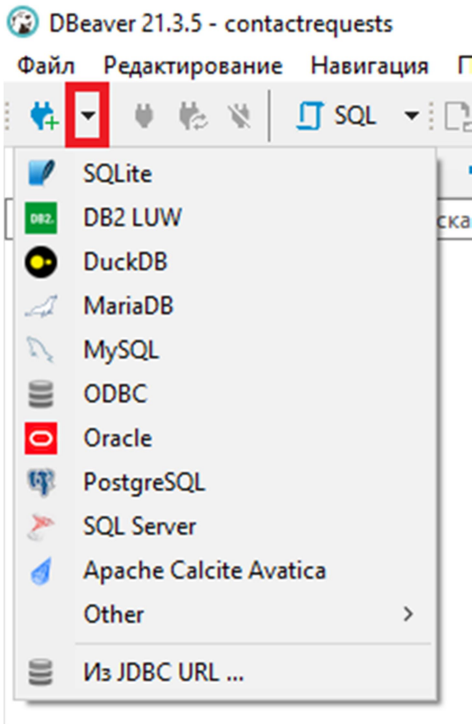


Рис. 1 – Выбор подключения в DBeaver

В окне подключения нажмите «Найти» и выберите созданный файл БД:

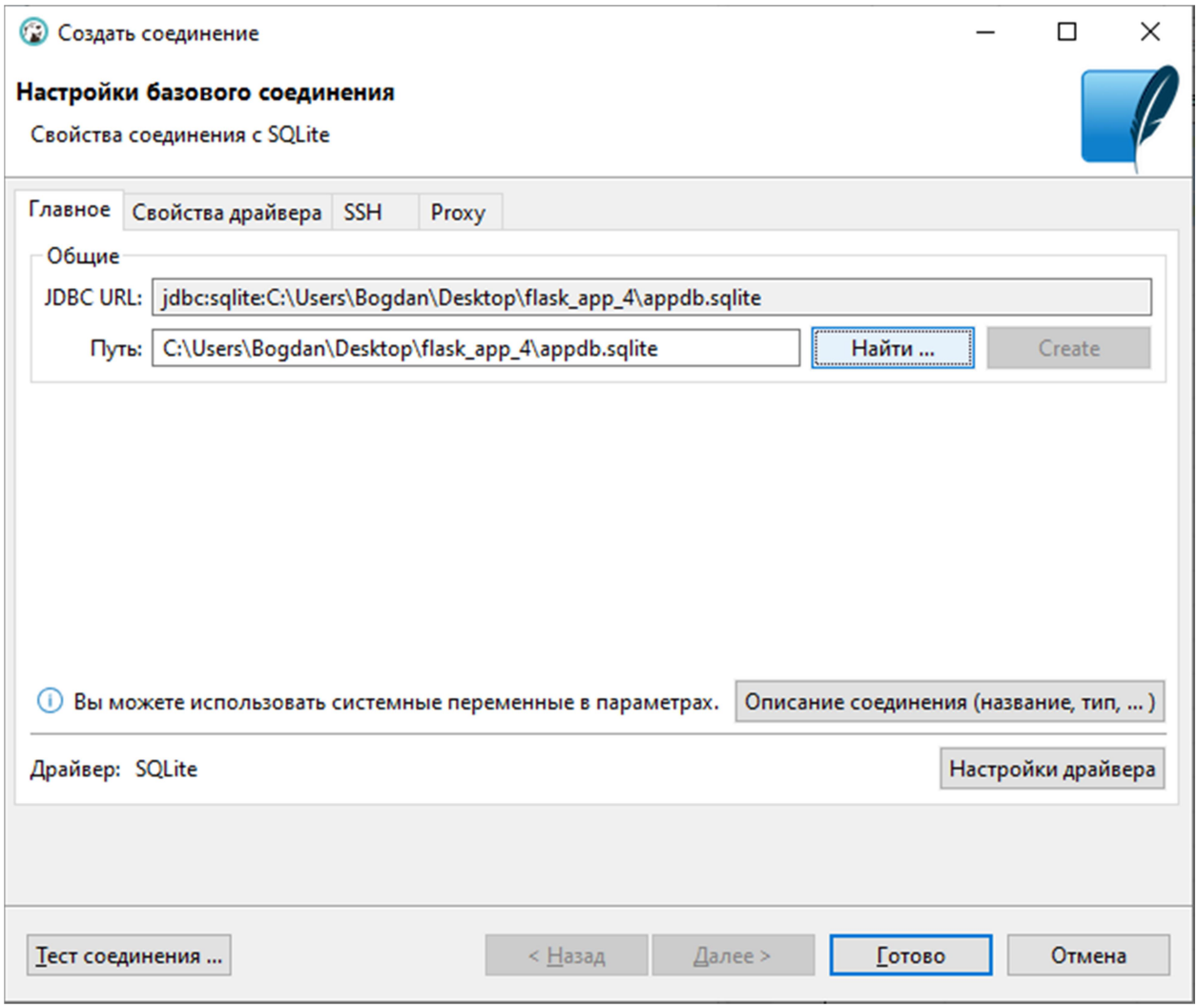


Рис. 2 – Настройка соединения с БД SQLite

После подтверждения клиент (возможно) автоматически предложить скачать драйвер для установления соединения. После установки соединения в списке соединений появится ваша БД. Выделите вашу БД и нажмите кнопку создания скрипта SQL на панели инструментов:

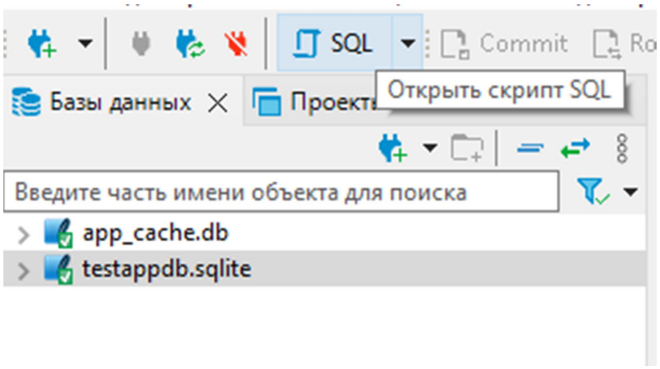


Рис. 3 – Запуск редактора SQL

В окне редактора SQL введите инструкцию для создания таблицы БД. Структура таблицы БД должна соответствовать веб-форме для отправки данных на одной из страниц вашего приложения. Например, на странице CONTACT реализована следующая веб-форма:

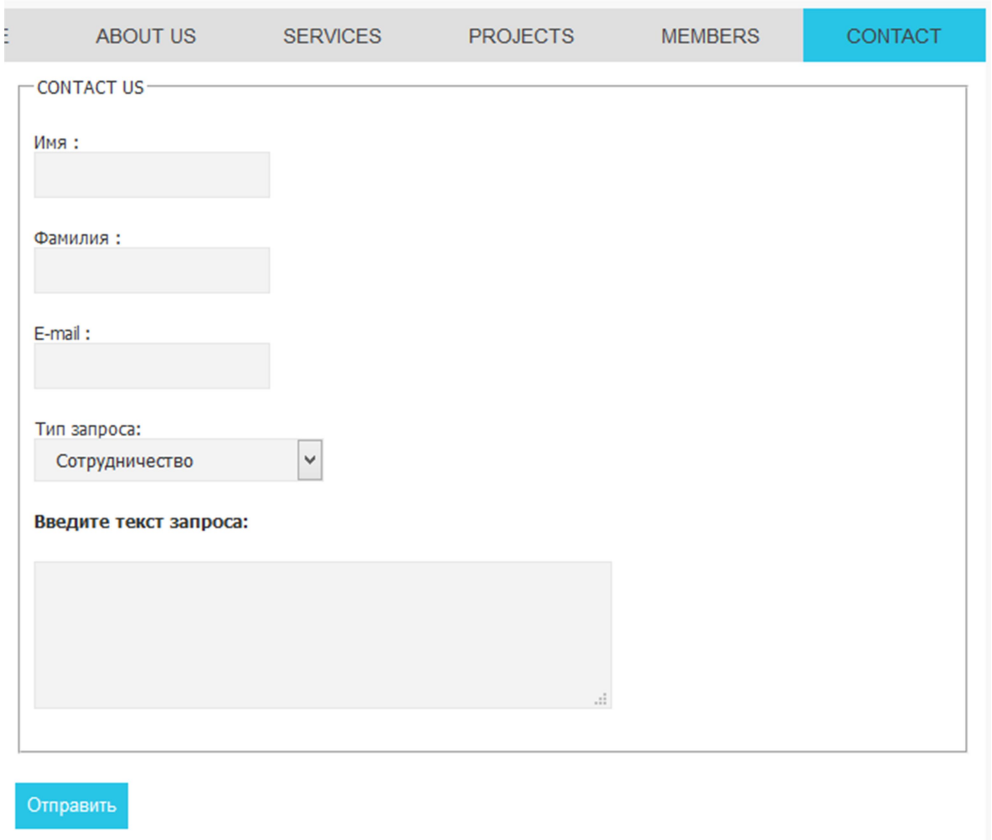


Рис. 4 – Форма создания нового запроса

Тогда инструкция для создания таблицы может иметь следующий вид:

```
create table contactrequests (
      id integer PRIMARY KEY autoincrement,
      firstname varchar(255) NOT NULL,
      lastname varchar(255),
      email varchar(255),
      reqtype varchar(255),
      reqtext varchar(255),
      cratedAt datetime,
      updatedAt datetime
 );
```

Для выполнения скрипта, используйте кнопки слева от поля редактирования. После выполнения скрипта, выберите вашу БД в списке соединения и нажмите F5 для обновления данных. После обновления должна появиться возможность раскрыть структуру БД и просмотреть поля таблицы:

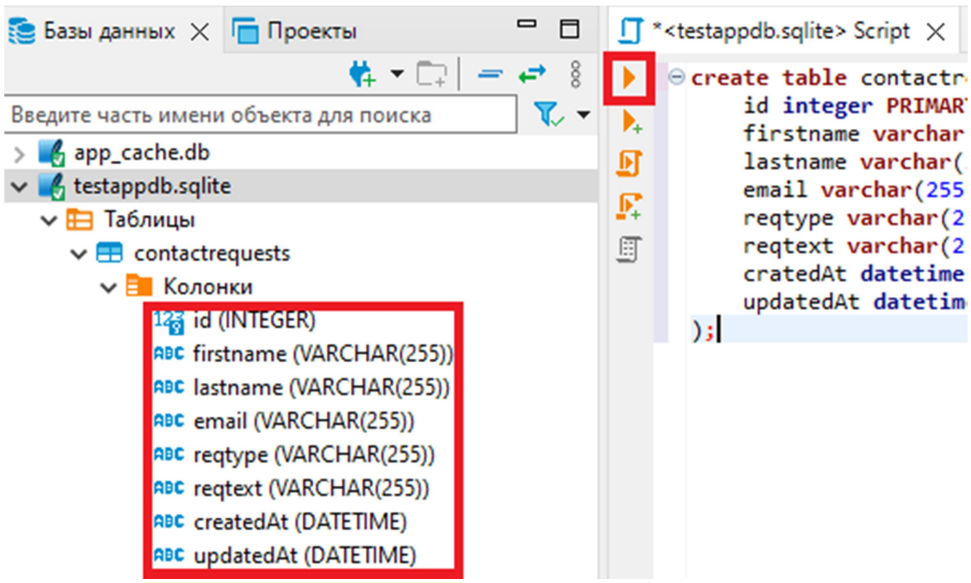


Рис. 5 – Выполнение SQL-скрипта и проверка структуры БД

2. Подключение БД в проекте веб-приложения.

Для подключения БД в **Python Flask** можно воспользоваться популярным фреймворком для работы с БД **SQLAlchemy**. Данный framework предоставляет универсальные методы (как на основе простых SQL-запросов, так и с помощью ORM) для работы со всеми популярными реляционными БД, в т.ч. MSSQL (подробнее см. документацию:

https://docs.sqlalchemy.org/en/14/tutorial/index.html).

Для установки SQLAlchemy в Flask-приложение используем пакетный менеджер pip:

```
pip3 install flask-sqlalchemy
```

Для работы с реляционными БД необходима установка «диалекта» (библиотеки для работы с БД). Например, для MySQL:

```
pip3 install pymysql
```

Поддержка драйвера SQLite в Python есть по умолчанию, поэтому установки дополнительных драйверов не требуется.

Подключение БД в проект Flask осуществляется в следующих файлах:

config.py – файл конфигурации приложения Flask. Здесь указывается основной параметр **SQLALCHEMY_DATABASE_URI** – строка подключения к БД. В примере используется подключение к БД SQLite, информацию по подключению к другим БД можно посмотреть в документации:

https://docs.sqlalchemy.org/en/14/core/engines.html

labapp/__init.py__ – файл инициализации основного пакета приложения. Здесь также инициализируется и провайдер для работы с БД:

```
# Регистрируем приложение Flask
app = Flask(__name__)
# Подключаем конфигурацию приложения
app.config.from_object(Config)
# Данный объект (провайдер) для работы с базой данных, интегрированный в
Flask,
# берет на себя все функции по управлению подключениями
db = SQLAlchemy(app)
```

3. Реализация CRUD-методов.

Модуль **labapp/dbservice.py** (**см. пример приложения**) содержит код функций, которые **реализуют CRUD методы** для работы с вашей таблицей: создание (create), чтение (read), модификация (update), удаление (delete). Если в вашем приложении используется несколько сущностей (таблиц) в БД, то хорошей практикой будет являться реализация отдельных модулей с CRUD-операциями для каждой таблицы, при этом данные модули лучше группировать в отдельном пакете Python, т.е. создавать папку с файлом __init__.py . Например, если у вас есть таблица «books» то для работы с ней создается отдельный модуль в пакете dbservice: **labapp/dbservice/book_service.py**, для таблицы с пользователями («users») соответственно: **labapp/dbservice/user_service.py** и т.д.

Пример функции, реализующей чтение записи из БД по идентификатору:

```
# Получаем запрос с фильтром по id
def get_contact_req_by_id(id):
    result = db.session.execute(f"SELECT * FROM contactrequests WHERE id =
{id}").fetchone()
    return dict(result)
```

Пример функции, считывающей массив строк из БД:

```
# Получаем список всех запросов.
def get_contact_req_all():
    result = []      # создаем пустой список
    # Получаем итерируемый объект, где содержатся все строки таблицы
contactrequests
    rows = db.session.execute("SELECT * FROM contactrequests").fetchall()
    # Каждую строку конвертируем в стандартный dict, который Flask может
трансформировать в json-строку
    for row in rows:
        result.append(dict(row))
    # возвращаем dict, где result - это список с dict-объектов с информацией
    return {'contactrequests': result}
```

Выполнение SQL-запроса происходит с помощью вызова метода execute(…). В качестве аргумента метод принимает обычную строку, подстановку данных в которую можно делать с помощью форматирования (подробнее о методах форматирования строк в python: **https://shultais.education/blog/python-f-strings**)

Результатом выполнения execute() может быть или один объект строки, если инструкция завершается с помощью .fetchone(), или массив строк-объектов, если используется .fetchall(). Каждую строку-объект необходимо преобразовывать в словарь Python (**dict**) для дальнейшей конвертации в json-формат.

Для методов, реализующих операции по изменению данных (INSERT, UPDATE и т.д.) необходимо также предусмотреть обработку исключений через **try-except**, позволит вывести причину ошибки, а также безопасно «откатить» (rollback) изменения в случае ошибки:

```
# Обновить текст запроса по id в таблице
def update_contact_req_by_id(id, json_data):
    try:
        cur_time = datetime.now().strftime("%Y-%m-%d %H:%M:%S")  # текущая
дата и время
        # UPDATE запрос в БД
        db.session.execute(f"UPDATE contactrequests SET reqtext =
'{json_data['reqtext']}', "
                           f"updatedAt = '{cur_time}' WHERE id = {id}")
        db.session.commit()
        return {'message': "ContactRequest Updated!"}
    except Exception as e:
        db.session.rollback()
        return {'message': str(e)}
```

4. Реализация REST API.

REST (Representational State Transfer) – это модель взаимодействия клиент-серверного приложения в сети по протоколу **HTTP**.

API (Application Programming Interface – программный интерфейс приложения) – описание классов, процедур, функций и методов взаимодействия между приложениями. Проще говоря, это «язык общения» между приложениями.

Данная модель взаимодействия позволяет осуществлять вызов удаленных процедур (методов) web-приложения для взаимодействия с ресурсами данного приложения. Вызов данных методов осуществляется с помощью структурированных (унифицированных) адресов, которые обозначаются аббревиатурой **URI (Uniform Resource Identifier)**.

Web-приложение, использующее для предоставления своих ресурсов REST-модель взаимодействия, называется **RESTful веб-сервис**. Такое приложение

использует специальные методы протокола HTTP и соответствующие структурированные веб-адреса.

В **табл. 1** представлены основные методы HTTP для взаимодействия с RESTful веб-сервисом и URI для предоставления ресурсов веб-приложения из примера.

Таблица №1 – Описание REST API веб-приложения из примера

Метод HTTP	Действие	Пример URI
GET	Получить информацию о всех запросах	http://127.0.0.1:3000/api/contactrequest
GET	Получить информацию о запросе по id	http://127.0.0.1:3000/api/contactrequest/123 (информация о запросе №123)
GET	Получить информацию о всех запросах, созданных определённым автором	http://127.0.0.1:3000/api/contactrequest/author/Tom (информация о всех запросах, созданных автором с именем Tom)
POST	Создать новый запрос	http://127.0.0.1:3000/api/contactrequest (создать новый запрос из json-данных переданных с запросом)
PUT	Обновить запрос	http://127.0.0.1:3000/api/contactrequest/123 (обновить запрос №123 json-данными переданными с запросом)
DELETE	Удалить запрос	http://127.0.0.1:3000/api/contactrequest/123 (удалить запрос №123)

Таким образом, данную REST-модель взаимодействия можно использовать для вызова соответствующих CRUD-методов вашей модели через HTTP-протокол.

5. Связывание CRUD-операций с REST API веб-приложения.

Базовые принципы подключения маршрутов в веб-приложении Flask были изложены в методических рекомендациях к лабораторной работе №3.

Подключение маршрутов к вашим CRUD-методам осуществляется в модуле **labapp/routes.py** с помощью импорта модуля dbservice.py и реализации соответствующих функций обработки маршрутов, например:

```
from . import dbservice


@app.route('/api/contactrequest', methods=['GET'])
# Получаем все записи contactrequests из БД
def get_contact_req_all():
    response = dbservice.get_contact_req_all()
    return json_response(response)
```

Обрабатывает **GET**-запрос по адресу:

http://127.0.0.1:8888/api/contactrequest

и вызывает метод **get_contact_req_all()**, который возвращает все записи из БД.

```
@app.route('/api/contactrequest/<int:id>', methods=['GET'])
# Получаем запись по id
def get_contact_req_by_id(id):
    response = dbservice.get_contact_req_by_id(id)
    return json_response(response)
```

Обрабатывает **GET**-запрос по адресу:

http://127.0.0.1:8888/api/contactrequest/1

Последняя часть адреса (**<int:id>**) передается в функцию обработчика маршрута в качестве аргумента (id == 1).

```
@app.route('/api/contactrequest/author/<string:firstname>', methods=['GET'])
# Получаем запись по имени пользователя
def get_get_contact_req_by_author(firstname):
    if not firstname:
        # то возвращаем стандартный код 400 HTTP-протокола (неверный запрос)
        return bad_request()
        # Иначе отправляем json-ответ
    else:
        response = dbservice.get_contact_req_by_author(firstname)
    return json_response(response)
```

Обрабатывает **GET**-запрос по адресу:

http://127.0.0.1:8888/api/contactrequest/author/Tom

Последняя часть адреса (**<string:firstname>**) передается в функцию обработчика маршрута в качестве аргумента (**firstname** == “Tom”).

Также дополнительные параметры можно передавать с помощью следующей строки запроса:

http://127.0.0.1:8888/api/contactrequest/author?firstname=Tom

Тогда в функции обработчика параметр firstname можно получить, используя инструкцию **request.args.get(‘firstname’)**.

```
@app.route('/api/contactrequest', methods=['POST'])
# Обработка запроса на создание новой записи в БД
def create_contact_req():
    # Если в запросе нет данных или неверный заголовок запроса (т.е. нет
'application/json'),
    # или в данных нет обязательного поля 'firstname' или 'reqtext'
    if not request.json or not 'firstname' or not 'reqtext' in request.json:
        # возвращаем стандартный код 400 HTTP-протокола (неверный запрос)
        return bad_request()
    # Иначе добавляем запись в БД отправляем json-ответ
    else:
        response = dbservice.create_contact_req(request.json)
        return json_response(response)
```

Обрабатывает **POST**-запрос по адресу:

http://127.0.0.1:8888/api/contactrequest

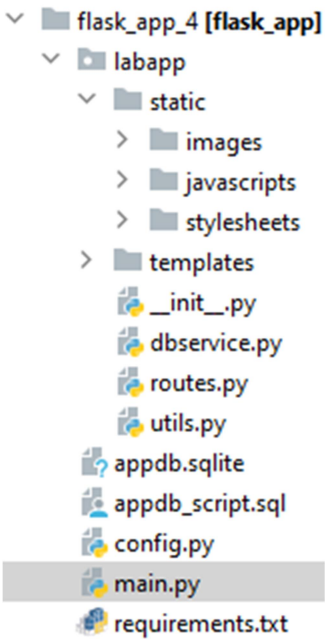
Здесь функция обработчика принимает json-данные, валидирует обязательные поля их, конвертирует в dict-объект с помощью инструкции **request.json** и передает в метод для создания записи в БД.

Аналогично работает обработка маршрутов:

```
@app.route('/api/contactrequest/<int:id>', methods=['PUT'])
@app.route('/api/contactrequest/<int:id>', methods=['DELETE'])
```

Для соответствующих HTTP-методов PUT и DELETE.

Итоговая структура проекта представлена ниже:



6. Тестирование REST API. Примеры, тестирования API с помощью Insomnia представлены на рисунках ниже:

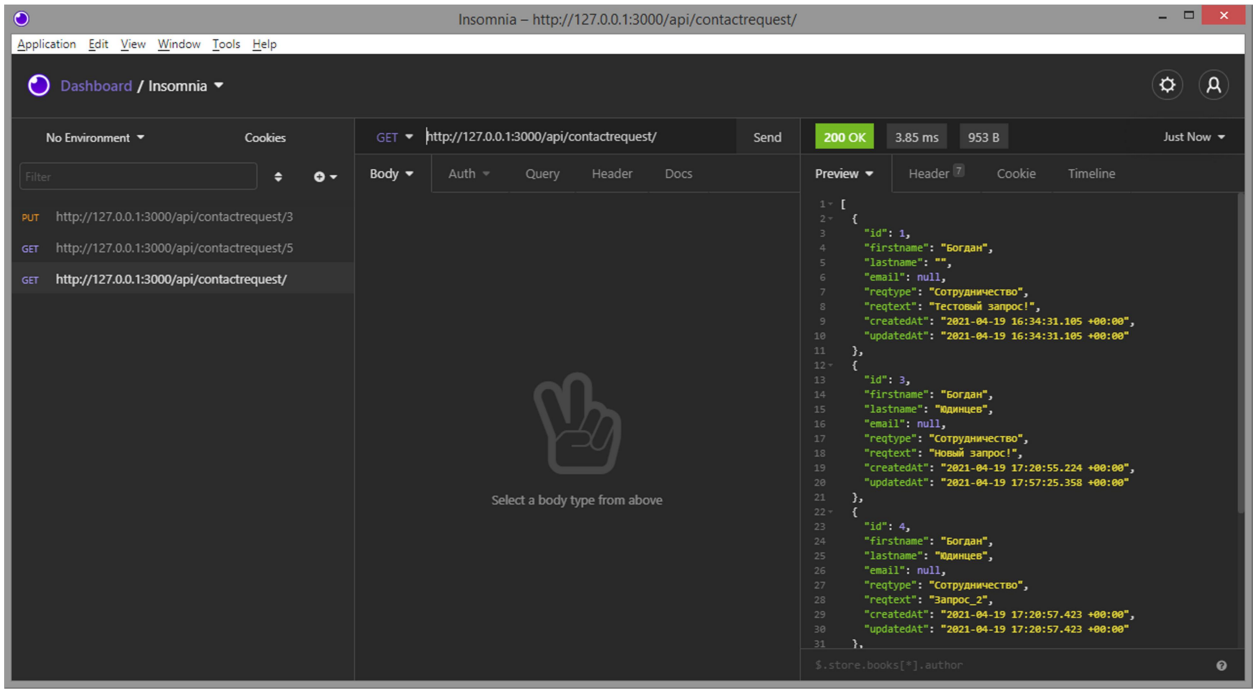


Рис. 6 – Тестирование GET-метода

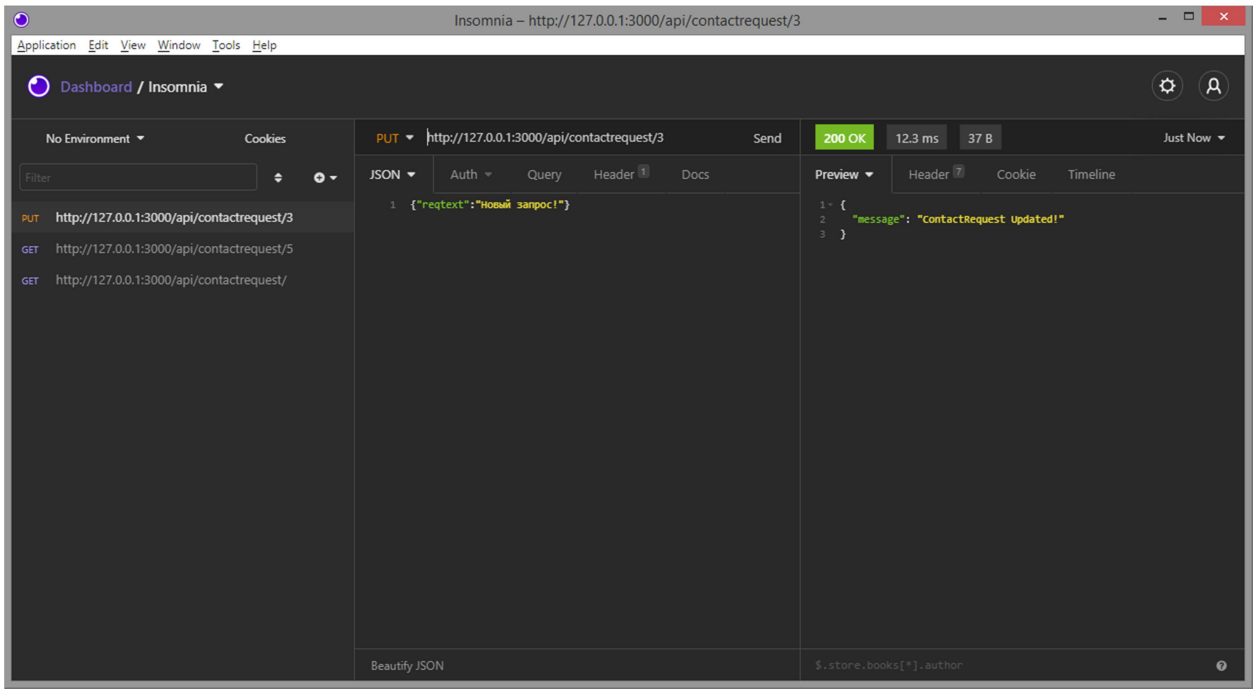


Рис. 7 – Тестирование PUT-метода

Дополнительная литература:

Tutorial по созданию web-приложений Flask + SQLA:

https://flask-sqlalchemy-russian.readthedocs.io/ru/latest/quickstart.html

Немного устаревшая, но более подробная информация:

https://habr.com/ru/post/246699/

https://habr.com/ru/post/193242/

Результатом выполнения задания являются файлы с кодом веб-приложения и отчет, содержащий следующую информацию:

1) Конспект теоретического материала по темам: основы протокола HTTP, основные HTTP-методы, HTTP-коды.
2) Таблица (аналогичная таблице №1 в методических рекомендациях) с описанием API для вашей модели.
3) Скриншоты, содержащие результаты проверки вашего API в Insomnia или любом другом REST-клиенте.

4) Работающее веб-приложение, реализованное по заданию.

Требования к оформлению отчета:

Способ выполнения текста должен быть единым для всей работы. **Шрифт** –

Times New Roman, кегль 14, **межстрочный интервал** – 1,5, **размеры полей**: левое – 30 мм; правое – 10 мм, верхнее – 20 мм; нижнее – 20 мм. Сокращения слов в тексте допускаются только общепринятые.

Абзацный отступ (1,25) должен быть одинаковым во всей работе. **Нумерация страниц** основного текста должна быть сквозной. Номер страницы на титульном листе не указывается. Сам номер располагается внизу по центру страницы или справа.

Разработано: Юдинцев Б.С.