5 Задание 5. Разработка REST веб-сервиса

Цель: на языке высокого уровня (Java, C#, Python и др. – на выбор обучающегося) реализовать REST веб-сервис и клиент для него, обеспечивающий сбор и предоставление клиентам возможностей работы с ресурсами игры «SOA-мафия».

5.1 Критерии оценивания

№	Задача	**Баллы**
1.	Реализовать REST-сервис, который предоставляет возможность добавления, просмотра, редактирования и удаления следующей информации по профилю игрока: 1) Его имя (никнейм) 2) Аватар (изображение) 3) Пол 4) E-mail Должна быть обеспечена возможность получения профиля как отдельного игрока, так и перечня игроков.	4
2.	Реализовать сбор и представление посредством REST-сервиса статистики по игрокам и проведенным сессиям игр. Статистика по игроку должна генерироваться в виде PDF-документа, содержащего информацию: 1) Профиль игрока 2) Количество сессий, в которых участвовал игрок 3) Количество побед 4) Количество поражений 5) Общее время в игре Статистика по игроку должна генерироваться по асинхронному запросу, возвращающему URL, по которому в дальнейшем будет доступен PDF-	6

	документ со сгенерированной статистикой. Генерация статистики должна быть реализована на основе паттерна «Очередь заданий».	
*3**	*Организовать регистрацию, авторизацию и разграничение прав пользователей к редактированию профиля игрока в REST-сервисе.*	*3**
*4**	*Организовать единый механизм регистрации и авторизации пользователей с использованием JWT как для основного приложения, так и для REST-сервиса*	*3**

5.2 Методические указания

В качестве примера реализации REST-сервиса, возьмем фреймворк Flask для языка Python.

Начнем с настройки среды разработки. Будем использовать Visual Studio Code.

1. Установите дополнение для Python.
2. Установите версию Python 3 (для которой написано это руководство). Опции включают в себя:
    - (Все операционные системы) Загрузка с python.org; обычно используйте кнопку **Загрузить Python 3.9.1**, которая появляется первой на странице (или любую другую последнюю версию).
    - (Linux) Встроенная установка Python 3 работает хорошо, но для установки других пакетов Python необходимо запустить `sudo apt install python3-pip` в терминале.
    - (macOS) Установка через Homebrew на macOS с использованием `brew install python3` (системная установка Python на macOS не поддерживается).
    - (Все операционные системы) Загрузка с сайта Anaconda (для целей анализа данных).
3. На Windows убедитесь, что расположение вашего интерпретатора Python включено в переменную окружения PATH. Вы можете проверить это место, запустив `path` в командной строке. Если папка интерпретатора Python не включена, откройте Настройки Windows, найдите "окружение", выберите **Редактировать переменные окружения для вашей учётной записи (Edit**

environment variables for your account), затем отредактируйте переменную **Path**, добавив в нее эту папку.

5.2.1 Подготовка среды для разработки

В этом разделе вы создаете виртуальную среду, в которой установлен Flask. Использование виртуального окружения позволяет избежать установки Flask в глобальное окружение Python и дает вам точный контроль над библиотеками, используемыми в приложении. Виртуальное окружение также упрощает создание файла requirements.txt для этого окружения.

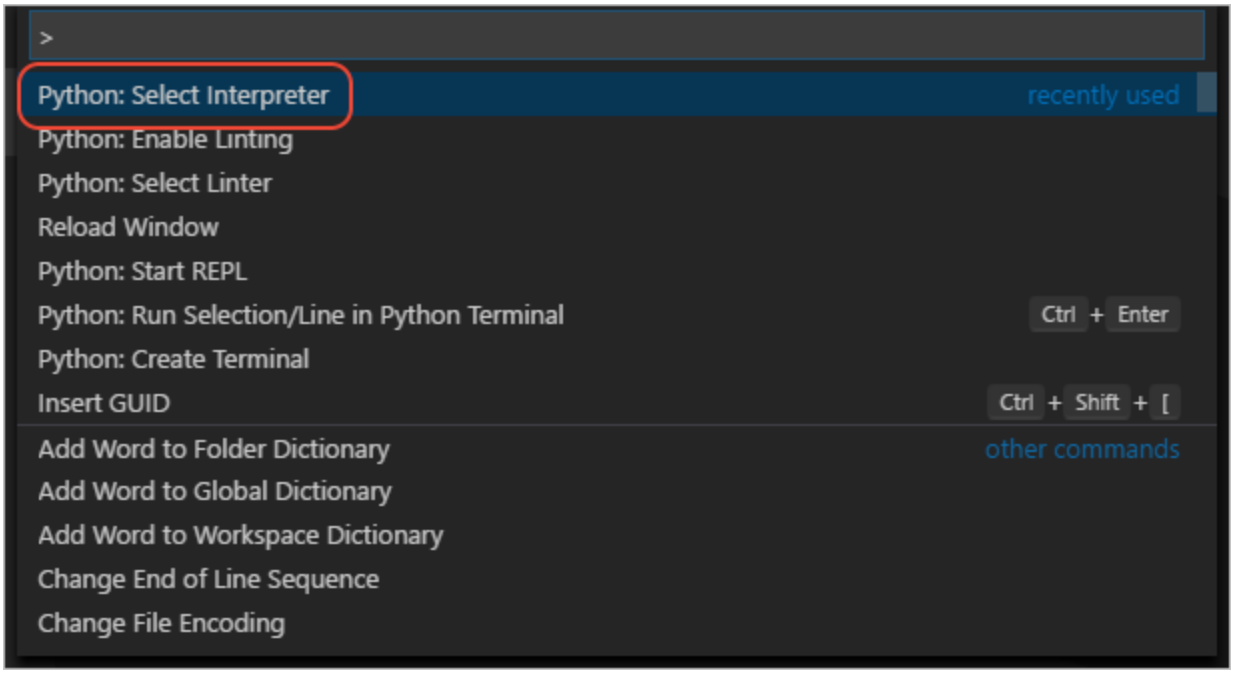
1. В вашей файловой системе создайте папку проекта для этого руководства, например, `hello_flask`.
2. В этой папке используйте следующую команду (соответствующую вашему компьютеру), чтобы создать виртуальную среду с именем `env`, основанную на вашем текущем интерпретаторе:

```
# Linux
sudo apt-get install python3-venv    # If needed
python3 -m venv env


# macOS
python3 -m venv env


# Windows
python -m venv env
```

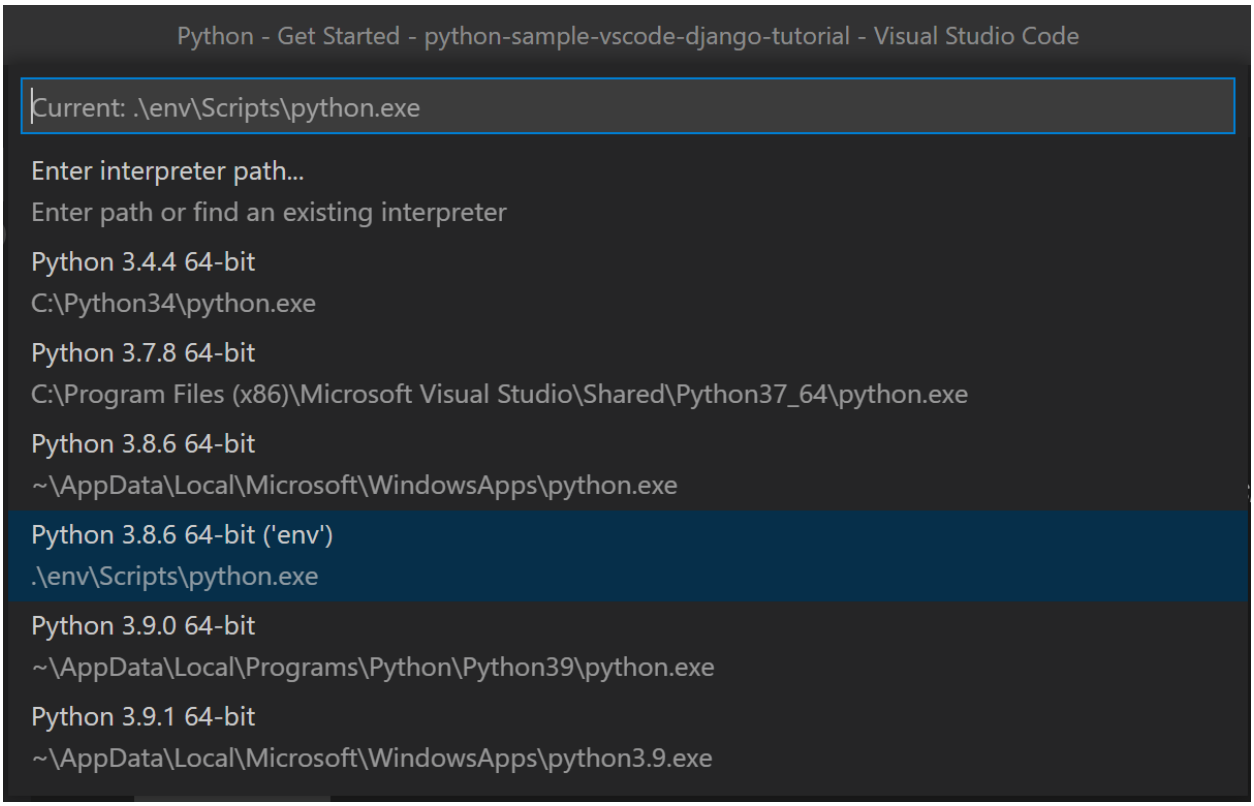
Примечание: Используйте стандартную установку Python при выполнении вышеуказанных команд. Если вы используете `python.exe` из установки Anaconda, вы увидите ошибку, потому что модуль ensurepip недоступен, а окружение остаётся несозданным.

3. Откройте папку проекта в VS Code набрав `code .` (в терминале) или открыв **File** > **Open Folder** .

4. В VS Code: **View** > **Command Palette** или(`Ctrl+Shift+P`)). Выберете **Python: Select Interpreter** :



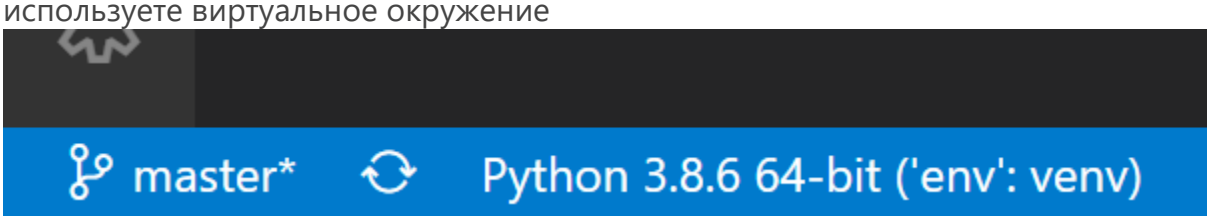
5. Команда представляет список доступных интерпретаторов, которые VS Code может найти автоматически (ваш список будет меняться; если вы не видите нужного интерпретатора, смотрите раздел Настройка окружения Python). Из списка выберите виртуальное окружение в папке проекта, которое начинается с `./env` или `.\env`:



6. Запустите **Terminal: Create New Integrated Terminal** (`Ctrl+Shift+`) из командной строки VS Code. Эта команда создает терминал и автоматически активизирует виртуальную среду, запустив его скрипт активации.

   Примечание: В Windows, если типом терминала по умолчанию является PowerShell, вы можете увидеть ошибку, что он не может запустить activate.ps1, потому что запущенные скрипты отключены в системе. Ошибка дает ссылку на информацию о том, как разрешить сценарии.

   Выбранное окружение появляется в левой части строки состояния VS Code. Вы можете заметить индикатор "(venv)", который говорит о том, что вы используете виртуальное окружение



7. Обновите pip в виртуальной среде, выполнив следующую команду в терминале VS Code Terminal:

```
python -m pip install --upgrade pip
```

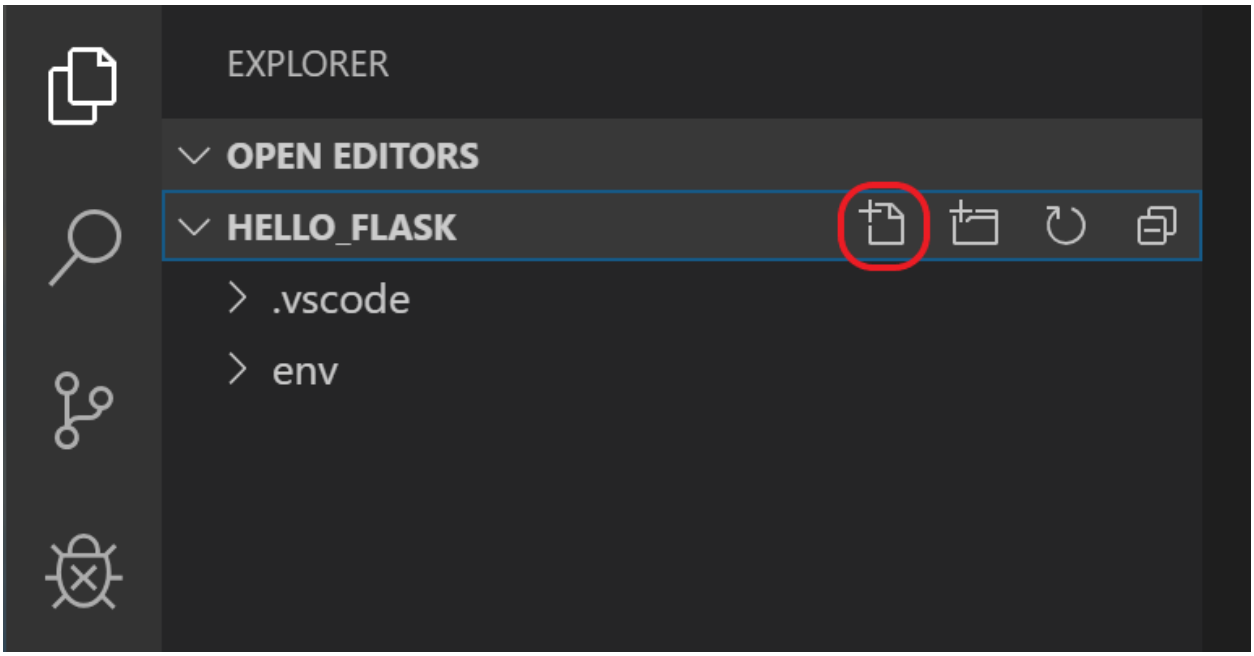
8. Установите Flask в виртуальную среду, выполнив следующую команду в терминале VS Code Terminal:

```
python -m pip install flask
```

Теперь у вас есть автономная среда, готовая к написанию кода Flask. VS Code автоматически активирует среду при использовании **Terminal: Create New Integrated Terminal**. Если вы открываете отдельную командную строку или терминал, активируйте среду, запустив `source env/bin/activate` (Linux/macOS) или `env\Scripts\Activate.ps1` (Windows). Вы знаете, что среда активируется, когда командная строка показывает **(env)** в начале.

5.2.2 Создание и запуск минимального приложение на Flask

1. В VS Code, создайте новый файл в папке вашего проекта под названием `app.py` используя либо **File** > **New** или нажав `Ctrl+N`,).



2. В файле `app.py`, добавьте код для импорта Flask и создайте экземпляр объекта Flask.

```
from flask import Flask

app = Flask(__name__)
```

3. Также в `app.py`, добавьте функцию, которая возвращает содержимое, в данном случае простую строку, и используйте декоратор `app.route` to map для отображения URL маршрута `/` на эту функцию:

```
@app.route("/")

def home():

    return "Hello, Flask!"
```

Совет: Вы можете использовать несколько декораторов на одной и той же функции, по одному на строку, в зависимости от того, сколько различных маршрутов вы хотите отобразить к одной и той же функции

Сохраните файл `app.py` (`Ctrl+S`).

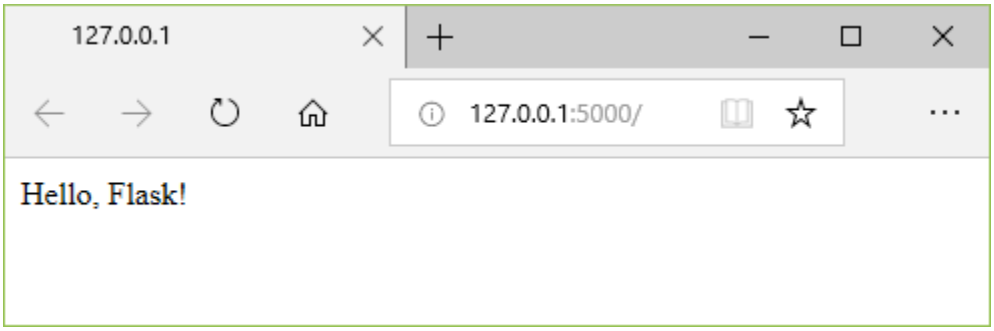
4. В интегрированном терминале запустите приложение, введя команду `python -m flask run`, который запускает сервер разработки Flask. По умолчанию сервер разработки ищет `app.py` by default При запуске Flask вы должны увидеть вывод, аналогичный следующему:

```
(env) D:\py\\hello_flask>python -m flask run

 * Environment: production

   WARNING: Do not use the development server in a production environment.

   Use a production WSGI server instead.

 * Debug mode: off

 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Если вы видите ошибку, что модуль Flask не может быть найден, убедитесь, что вы выполнили `python -m pip install flask` в вашей виртуальной среде, как описано в конце предыдущего раздела.

Также, если вы хотите запустить сервер разработки на другом IP-адресе или порту, используйте аргументы командной строки host и port, например `--host=0.0.0.0 --port=80`.

5. Чтобы открыть браузер по умолчанию для отображенной страницы, `Ctrl+click` на ссылке `http://127.0.0.1:5000/` в терминале.



6. Обратите внимание, что при посещении URL типа / в отладочном терминале появляется сообщение, показывающее HTTP-запрос:

```
127.0.0.1 - - [11/Jul/2018 08:40:15] "GET / HTTP/1.1" 200 -
```

7. Остановите приложение, используя `Ctrl+C` в терминале.

Совет: если вы хотите использовать имя файла, отличное от имени `app.py`, например, `program.py`, определите переменную окружения с именем FLASK_APP и установите ее значение в выбранный вами файл. Сервер разработки Flask затем использует значение FLASK_APP вместо файла по умолчанию `app.py`. Подробнее см. интерфейс командной строки Flask.

5.2.3 Простейшее приложение для списка дел

Исходный код приложения доступен в репозитории по адресу: https://github.com/domage/soa-curriculum-2021/tree/main/examples/flask-rest-tutorial

Чтобы продемонстрировать работу с REST, подготовим учебное приложение-пример. Веб-сервис обеспечивает работу со списком дел "ToDo". При помощи REST API обеспечивается возможность взаимодействия со списком дел путем обмена информацией в формате JSON, включая:

- по запросу ``GET`` к списку ToDo: получение списка ToDo;
- по запросу ``GET`` отдельного ToDo: получение отдельного ToDo по его индексу;
- по запросу ``POST`` к списку ToDo: добавление нового ToDo;
- по запросу ``PUT`` отдельного ToDo: обновление записи отдельного ToDo;
- по запросу ``DELETE`` отдельного ToDo: удаление отдельного ToDo.

Мы будем использовать очень легковесную файловую базу данных SQLite для хранения данных. Вы можете установить Браузер баз данных для SQLite, либо расширение VS Code для легкой работы с этой базой данных.

Назовем эту БД todo.db и поместим ее в директорию нашего проекта. Теперь, чтобы создать таблицу, в которой будут храниться наши данные выполним простой запрос:

```
CREATE TABLE "todos" (
    "todo_id" INTEGER PRIMARY KEY,
    "description" TEXT NOT NULL,
    "status" TEXT NOT NULL
);
```

Кроме того, чтобы все было просто, мы будем записывать все наши маршруты в один файл, хотя это не всегда хорошая практика, особенно для очень больших приложений.

Мы также будем использовать еще один файл, который будет содержать наши вспомогательные функции. Эти функции будут обеспечивать выполнение бизнес-логики для обработки HTML-запросов путем подключения к базе данных и выполнения соответствующих SQL-запросов.

Создадим в каталоге нашего проекта файл `helper.py` и внесем туда код, который позволит создать новые задачи в нашу базу данных

```
import sqlite3
from flask import jsonify, url_for

DB_PATH = './todo.db'   # Update this path accordingly
NOTSTARTED = 'Not Started'

# Добавить элемент в таблицу
def add_to_list(description):
    try:
        conn = sqlite3.connect(DB_PATH)
        c = conn.cursor()
        c.execute('insert into todos(description, status) values(?,?)', (descript
ion, NOTSTARTED))
        conn.commit()
        result = get_todo(c.lastrowid)
        return result
    except Exception as e:
        print('Error: ', e)
```

```
        return None
```

В заголовке мы импортируем пакеты, которые позволяют нам работать с базой данных SQLite, а также утилиты из пакета Flask.

В переменную DB_PATH мы сохранили путь к нашей базе данных.

Комментарии к методу add_to_list излишни – мы соединяемся с базой данных и вносим туда переданное описание задания, после чего получаем сгенерированное для него ID. Единственное, мы еще не реализовали функцию, которая бы возвращала нам запись нашей задачи. Реализуем эту функцию.

```
# Получить отдельный элемент
def get_todo(todo_id):
    try:
        conn = sqlite3.connect(DB_PATH)
        # Обеспечивает работу с названиями колонок в таблице
        conn.row_factory = sqlite3.Row
        c = conn.cursor()
        c.execute("select * from todos where todo_id=?;" , [todo_id])
        r = c.fetchone()
        return jsonify(make_public_todo(r))
    except Exception as e:
        print('Error: ', e)
    return None
```

Тут логика примерна такая-же. Соединяемся с базой данных и читаем из нее строку по индексу.

Но для того, чтобы отобразить эту запись, мало сгенерировать из нее JSON методом jsonify. При реализации API, принято к сущностям, которые идентифицируются по ID, по возможности возвращать не ID, а URL. Для этого реализуем небольшую функцию, которая будет собирать make_public_todo, которая будет формировать URL из ID для наших задач.

```
# Формирование структуры ответа на основе информации, представленной в базе данных
def make_public_todo(row):
    new_todo = {}
    for field in row.keys():
        # Предоставление URL объекта вместо его id - хорошая практика
        if field == 'todo_id':
```

```
            new_todo['uri'] = url_for('get_todo', todo_id = row['todo_id'], _external
 = True)
        else:
            new_todo[field] = row[field]

    return new_todo
```

На этом построение функционала, достаточного для базового запуска приложения будет достаточно. Переходим в файл `app.py.`

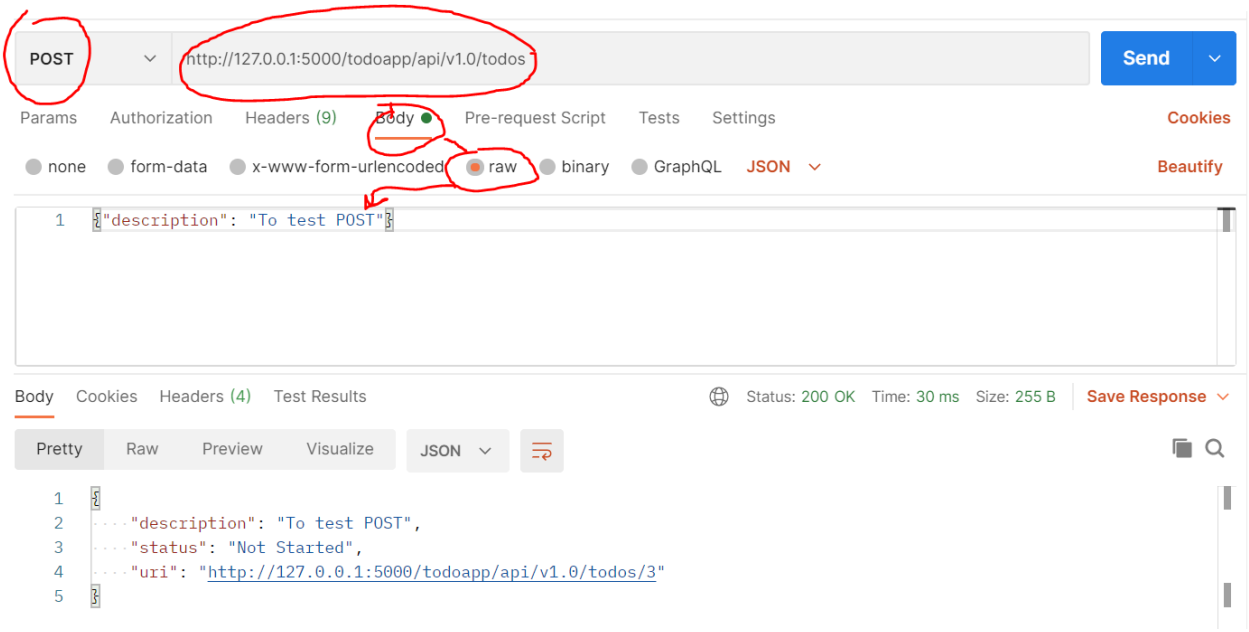
Для того, чтобы создать запись, нам надо обработать запрос POST в корень нашего API. При этом в запросе мы должны получить информацию о задаче. Реализуем это, добавив следующую функцию в `app.py`

```
# Добавить элемент в коллекцию. В теле запроса должен быть передан JSON с полем 'description'
@app.route('/todoapp/api/v1.0/todos', methods=['POST'])
def add_todo():

    # Если в параметрах запроса нет тела, либо нет поля 'description' - отбой
    if not request.json or not 'description' in request.json:
        abort(400)

    # Получаем поле из запроса
    description = request.get_json()['description']

    # Добавляем элемент в базу данных
    response = helper.add_to_list(description)

    # Если не удачно - возвращаем ошибку 400
    if response is None:
        abort(400)

    # Возвращаем полное описание добавленного элемента
    return response
```

Самое время проверить работу нашего приложения. Запустим сервер командой `python -m flask run`

Отлично. Сервер работает. Но для того, чтобы проверить работу метода POST в текущих условиях, браузера будет не достаточно. Для отправки запросов на сервер можно использовать утилиту `curl` либо [postman](). Для добавления новой записи проверим выполнение вызова (при условии, что сервер развернут по адресу `http://127.0.0.1:5000`):

```
curl --location --request POST 'http://127.0.0.1:5000/todoapp/api/v1.0/todos' --
header 'Content-Type: application/json' --data-raw '{"description": "To test POST"}'
```



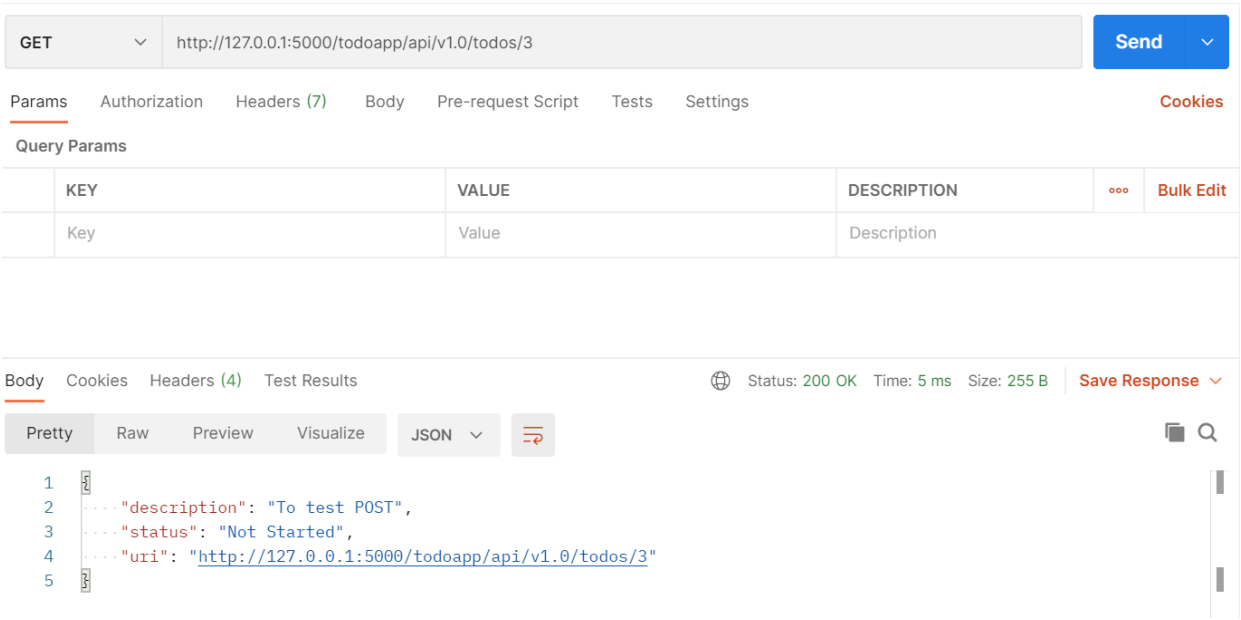
Результатом выполнения этой команды будет JSON объект с описанием новой задачи и с URI, по которой можно эту задачу посмотреть.

Давайте реализуем просмотр задачи, добавив следующую функцию в `app.py`

```
# Получаем отдельную запись по индексу todo_id
@app.route('/todoapp/api/v1.0/todos/<int:todo_id>', methods=['GET'])
def get_todo(todo_id):
    # Получаем запись из базы данных
    response = helper.get_todo(todo_id)

    # Если не найдено - ошибка 404
    if response is None:
        abort(404)

    return response
```

Здесь в конце пути мы используем параметр `<int:todo_id>`, который передается в виде значения в функцию get_todo. В нашем файле-хэлпере мы реализовали функцию get_todo, так что мы можем проверить работу этой фукнции, выполнив метод GET с URI, переданным нам ранее в результате выполнения метода POST. Перезапускаем сервер и выполняем запрос:

```
curl --location --request GET 'http://127.0.0.1:5000/todoapp/api/v1.0/todos/3'
```



Таким образом, мы реализовали операции GET и POST. Реализация PUT, DELETE и формирование списка при вызове GET по URI коллекции доступна по ссылке в репозитории-примере: https://github.com/domage/soa-curriculum-2021/tree/main/examples/flask-rest-tutorial

После ознакомления с этим примером, так же рекомендуется ознакомиться с его продолжением – реализацией аналогичного функционала с расширением Flask-RESTX для систематизации работы с REST-сервисами и автоматической генерации документации по API с использованием OpenAPI. Этот пример также доступен в репозитории по ссылке: https://github.com/domage/soa-curriculum-2021/tree/main/examples/flask-restx-openapi . Для того, чтобы познакомиться с документацией OpenAPI, которая автоматически генерируется по исходному коду сервера, после его запуска достаточно пройти по адресу http://127.0.0.1:5000/

5.3 Задание на самостоятельную работу

На основе предложенных примеров вам предлагается самостоятельно выполнить задачи 1-2 из задания. При подготовке выполнения задания 2, вам может помочь пример приложения, реализующий паттерн «очередь задач»: https://github.com/domage/soa-curriculum-2021/tree/main/examples/grpc-queue

Задачи 3 и 4 – на дополнительные баллы.