Проект WebServer. REST-API

План урока

1 Введение. Микросервисы

2 Знакомство с архитектурой REST

3 Создание RESTful веб-сервисов на Flask

4 Flask-RESTful

Аннотация

На этом занятии мы научимся создавать API для наших веб-приложений, а также познакомимся с библиотекой Flask-RESTful.

1. Введение. Микросервисы

На прошлых уроках вы изучили базовые возможности фреймворка Flask и уже можете создавать полноценные веб-приложения, которые могут взаимодействовать с пользователем и работать с базой данных. На этом уроке мы предлагаем вам подняться на уровень выше

и рассмотреть **архитектуру** веб-приложения: то, по каким правилам будут взаимодействовать компоненты вашего приложения и как организовать код, чтобы в дальнейшем его было легче поддерживать и проще добавлять новые функции.

Можно выделить два основных подхода к организации архитектуры веб-приложений:

- монолитная архитектура (Monolithic Style);
- сервис-ориентированная архитектура (SOA, Service-oriented Architecture).

При монолитном подходе приложение строится как единое целое. Как правило, монолитное веб-приложение включает три части: веб-сервер, который содержит всю логику приложения, базу данных и пользовательский интерфейс в виде HTML-страниц. Любые изменения в логике сервера, даже самые небольшие, приводят к выпуску новой версии всего приложения.
А с учётом того, что всё больше приложений развёртываются на облачных серверах, после внесённых изменений нужно заново развёртывать в облаке монолитное приложение целиком.
Монолитный подход является самым старым, именно с него началась разработка всего программного обеспечения. Тем не менее, монолитные приложения живы и по сей день, и в некоторых проектах такой подход является подходящим.

Недостатки монолитной архитектуры привели к появлению сначала **модульной**, а потом и сервис-ориентированной архитектуры. **Сервис-ориентированная архитектура** (SOA) основана на использовании отдельных полностью самодостаточных модулей, называемых **сервисами**. Разработка сервисов происходит отдельно друг от друга, а потому изменения в коде одного сервиса будут влиять только на него.

Около 10 лет назад (сам термин устоялся только в 2012 году) появился ещё один подход, «современный взгляд» на проектирование распределённых приложений — **архитектура микросервисов** (MSA, Micro Service Architecture). Можно сказать, что MSA является разновидностью, частным случаем SOA. Такая архитектура подразумевает, что ваше приложение представляет собой много небольших сервисов, которые взаимодействуют между собой путём обмена сообщениями, как правило, по протоколу HTTP (или разные сервисы взаимодействуют по разным протоколам).

MSA предполагает, что каждый микросервис — это полностью независимое приложение, содержащее всего несколько сотен строк кода. Согласно MSA, микросервисы могут располагаться даже на различных серверах. Интересно, что при таком подходе разные микросервисы в рамках одного приложения могут быть написаны на разных языках программирования и с применением разных технологий.

Рекомендуем прочитать подробную статью про микросервисную архитектуру на Хабре.

Можно сказать, что микросервисы — это теория, а на практике она может выражаться по-разному. Мы рассмотрим архитектурный стиль, основанный на идеологии микросервисов, который на настоящее время становится стандартом при разработке веб-приложений — **архитектурный стиль REST** (REpresentational State Transfer).

2. Знакомство с архитектурой REST

REST — это архитектурный стиль программного обеспечения, который определяет правила управления информационными потоками и правила взаимодействия компонентов распределённого приложения. Другими словами, **REST** — это набор ограничений и принципов взаимодействия сервера и клиента в сети, который использует существующие стандарты: протокол HTTP, стандарт построения URL, форматы данных JSON и XML.

Ключевым понятием в архитектуре **REST** является **URL**. Каждая единица информации имеет уникальный идентификатор — URL, который строится по строго заданному формату. Например, вторая книга с книжной полки будет иметь URL /book/2, а 50 страница в этой книге — /book/2/page/50.

То, каким образом происходит управление этой информацией, определяет протокол передачи данных. **REST** использует протокол HTTP, соответственно, управление информацией происходит с помощью HTTP-запросов: GET (получить), PUT (заменить), POST (добавить), DELETE (удалить). Для каждой единицы информации определяется 5 операций:

№	Операция	Запрос	Пример URL	Примечание
1	Получить список всех объектов	GET	`/book/`	Получить список всех книг на полке
2	Получить информацию об объекте	GET	`/book/2`	Получить информацию о книге №2
3	Создать новый объект	POST	`/book/`	Добавить новую книгу на основе информации, переданной с запросом
4	Изменить существующий объект	PUT	`/book/1`	Изменить книгу №1 на основе информации, переданной с запросом
5	Удалить существующий объект	DELETE	`/book/2`	Удалить книгу №2

Данные для изменения и добавления (PUT- и POST-запросы) передаются в теле запроса в формате JSON или XML. Глядя на запрос, можно сразу определить, для чего он служит (к сожалению, в некоторых других архитектурных стилях — SOAP, XML-RPC — это не так).

Приложения, поддерживающие архитектуру REST, называются **RESTful-приложениями**. Давайте создадим наш первый RESTful веб-сервис.

3. Создание RESTful веб-сервисов на Flask

Перед написанием кода нужно продумать, с какой информацией мы будем работать и на основе каких правил будут строиться URL. Давайте вернёмся к нашему приложению по работе с новостями из прошлого урока и построим его по принципам REST. Для получения списка всех новостей можно использовать такой URL:

```
http://127.0.0.1:8080/news
```

Теперь давайте напишем функцию **get_news()** для получения всех новостей. В декораторе нужно указать наш URL. Также давайте пока опустим проверку того, что пользователь авторизован.

```
@app.route('/news',  methods=['GET'])
def get_news():
    news = NewsModel(db.get_connection()).get_all()
    return jsonify({'news': news})
```

Вместо точки входа index() теперь используется функция get_news() , связанная с URL /news/, для HTTP-метода GET. Также сейчас мы пока не будем работать с шаблонами (позже вы измените интерфейс и шаблоны самостоятельно), а посмотрим «сырые» данные от сервера. Согласно архитектуре REST, обмен данными между клиентом и сервером осуществляется в формате JSON (реже — XML). Поэтому мы изменили формат ответа сервера — Flask с помощью метода **jsonify** преобразует наши данные в JSON.

Посмотреть, что вернул сервер, можно и не открывая браузер. По сути, браузер — это средство для отправки запросов на сервер и отображения ответа. Отправить запрос на сервер можно и другими способами (например, консольная утилита curl или GUI-программа Postman), но, как программисты на Python, мы воспользуемся его средствами, а именно — модулем **requests**.

Модуль **requests** нужно предварительно установить с помощью утилиты pip. Он позволяет отправлять на сервер разные типы запросов, передавать данные вместе с POST- и PUT-запросами. Для этого используются специальные функции, их имена совпадают с именем HTTP-запроса. Для тестирования нашего RESTful-сервиса создадим файл test.py со следующим содержимым:

```
from requests import get

print(get('http://localhost:8080/news').json())
```

Запустим файл (не забудьте запустить сервер) и увидим, что на наш GET-запрос пришёл ответ от сервера со всеми новостями из базы данных.

Согласно REST, далее нужно реализовать получение информации об одной новости. Фактически, мы уже получили из списка всю информацию о каждой новости. При

проектировании приложений по архитектуре REST обычно поступают таким образом: когда возвращается список объектов, он содержит только краткую информацию (например, только id и заголовок), а полную информацию (текст и автора) можно посмотреть с помощью запроса, который мы обработаем далее.

Напишем функцию **get_one_news()** для получения информации о новости по её идентификатору. Как нам говорит REST, идентификатор объекта содержится в URL, по которому мы обращаемся. Как же передать этот идентификатор, если он различный у всех новостей, а в декораторе функции-обработчика мы указываем статический адрес? Flask позволяет решить эту проблему с помощью параметров адреса: мы можем передать параметр в URL, в декораторе заключив его в угловые скобки и указав тип и имя (например, <int:news_id>), и он будет передан функции-обработчику как аргумент. Обратите внимание, теперь наша функция принимает аргумент **news_id**:

```
@app.route('/news/<int:news_id>',  methods=['GET'])
def get_one_news(news_id):
    news = NewsModel(db.get_connection()).get(news_id)
    if not news:
        return jsonify({'error': 'Not found'})
    return jsonify({'news': news})
```

Посмотрим ответ сервера с корректным id и с ошибочным, добавив ещё два запроса в файл test.py:

```
from requests import get

print(get('http://localhost:8080/news').json())
print(get('http://localhost:8080/news/1').json())
print(get('http://localhost:8080/news/8').json()) # ошибка
# новости с id=8 нет в базе
```

Как видим, в первом случае нам вернулась одна новость, во втором — сообщение об ошибке. В типе параметра URL мы указали **int**. Что же будет, если мы передадим не число? Давайте проверим.

```
print(get('http://localhost:8080/news/q').json())
```

Файл test.py завершится с ошибкой. Так как параметр не типа int, в функцию get_one_news мы даже не попадём — нет полного совпадения декоратора. На такой запрос сервер вернёт ответ со статусом 404 (страница не найдена), но не в формате JSON. Поэтому при попытке преобразовать ответ с помощью метода json() программа валится с ошибкой, так как клиентское приложение, в нашем случае, ожидает от сервера ответ в формате JSON. Уберите приведение ответа к JSON и посмотрите, что вернул сервер.

Чтобы такой ошибки не возникало, воспользуемся модулем **make_response**. Импортируем модуль и добавим функцию с декоратором 404 ошибки:

```
from flask import make_response

@app.errorhandler(404)
def not_found(error):
    return make_response(jsonify({'error': 'Not found'}), 404)
```

После этого даже при передаче неправильного параметра ответ от сервера будет приходить в формате JSON, и клиентское приложение не будет падать. Верните метод json() и проверьте результат.

Следующая операция, которую нам нужно реализовать — добавление новой новости с помощью POST-запроса. Как уже было сказано, данные для создаваемой новости будут переданы в теле запроса. Чтобы добраться до тела запроса, нам понадобиться модуль **request** из flask (не путайте со стандартным модулем **requests**).

```
from flask import request

@app.route('/news', methods=['POST'])
def create_news():
    if not request.json:
        return jsonify({'error': 'Empty request'})
    elif not all(key in request.json for key in ['title', 'content', 'user_id']):
        return jsonify({'error': 'Bad request'})
    news = NewsModel(db.get_connection())
    news.insert(request.json['title'], request.json['content'],
                request.json['user_id'])
    return jsonify({'success': 'OK'})
```

Проверив, что запрос содержит все требуемые поля, мы заносим новую запись в базу данных. **request.json** содержит тело запроса, с ним можно работать, как со словарём. В нашем случае все ключи заранее известны и все поля обязательны, в противном случае для работы со словарём **request.json** нужно использовать метод словарей **get**. Вспомните, в чём его особенность.

При тестировании проверим пустой запрос, передачу только заголовка новости и корректный запрос. Все данные передаются в параметре **json**:

```
# test.py

from requests import get, post
```

```
print(post('http://localhost:8080/news').json())
print(post('http://localhost:8080/news',
           json={'title': 'Заголовок'}).json())
print(post('http://localhost:8080/news',
           json={'title': 'Заголовок',
                 'content': 'Текст новости',
                 'user_id': 1}).json())
```

Проверьте, что в каждом случае от сервера приходит нужное сообщение. Также давайте отправим запрос на получение всех новостей и убедимся, что новость добавлена.

Добавим функцию удаления новости:

```
@app.route('/news/<int:news_id>', methods=['DELETE'])
def delete_news(news_id):
    news = NewsModel(db.get_connection())
    if not news.get(news_id):
        return jsonify({'error': 'Not found'})
    news.delete(news_id)
    return jsonify({'success': 'OK'})
```

И протестируем её:

```
# test.py

print(delete('http://localhost:8080/news/8').json()) # ошибка
# новости с id=8 нет в базе
print(delete('http://localhost:8080/news/3').json())
```

Функцию изменения новости попробуйте написать самостоятельно.

Такой «ручной» способ создания RESTful-сервисов достаточно объёмный и трудоёмкий. Нужно держать в голове все URL и методы, создавать много функций-обработчиков с декораторами для каждого типа запросов (эти функции как раз и являются микросервисами). Программисты на Python не были бы программистами на Python, если бы не стремились упростить объёмный код и сделать его более лаконичным. Так и появился лёгкий способ создавать RESTful-сервисы на Python — модуль **Flask-RESTful**.

4. Flask-RESTful

Всё то, что мы сделали ранее при создании RESTful-сервиса, можно обернуть в более лаконичную, а главное — объектно-ориентированную оболочку, забыв о множестве разбросанных функций-обработчиков. Для этого установим дополнительный модуль **flask-restful** с помощью pip. Из него нам понадобятся следующие модули:

```
from flask_restful import reqparse, abort, Api, Resource
```

Создадим вторую версию нашего REST-сервера (в новом файле). После создания flask-приложения создадим объект RESTful-API:

```
app = Flask(__name__)
api = Api(app)
```

Для каждого ресурса (единица информации в REST называется **ресурсом**: новости, пользователи и т.д.) создаётся два класса: один объект и список объектов. В нашем случае это классы **News** и **NewsList** соответственно. Оба этих класса наследуются от класса **Resource**, который мы импортировали выше.

В классе одного объекта (**News**) определяются операции, которые мы можем сделать с **одним** объектом: получить информацию об объекте, изменить информацию, удалить объект. **flask-restful** подразумевает, что эти методы будут иметь имена, аналогичные соответствующим HTTP-запросам: get, put, delete. И все они, конечно, должны принимать в качестве аргумента идентификатор объекта.

Когда мы проектировали REST-сервис вручную, в каждой функции-обработчике мы проверяли, существует ли новость с таким идентификатором, и если нет, то отправляли ошибку. А также определили функцию **not_found** с декоратором @app.errorhandler(404), чтобы изменить формат ответа сервера в случае ошибки. Давайте вынесем эту проверку в отдельную функцию:

```
def abort_if_news_not_found(news_id):
    if not NewsModel(db.get_connection()).get(news_id):
        abort(404, message="News {} not found".format(news_id))
```

Функция **abort** генерирует HTTP-ошибку с нужным кодом и возвращает ответ в формате JSON, поэтому функция not_found нам больше не нужна.

В предыдущем нашем варианте мы никак не использовали коды ответа сервера, это неверно. Наше сообщение об ошибке мог прочитать только человек, а для программы-клиента оно ничего не значит, ответ пришёл со статусом OK (200). Давно знакомый вам код 404 означает, что запрашиваемый ресурс не найден. Подробнее про статусы состояния протокола HTTP можно почитать здесь.

Класс **News** с методами получения информации и удаления будет иметь вид:

```
class News(Resource):
    def get(self, news_id):
        abort_if_news_not_found(news_id)
        news = NewsModel(db.get_connection()).get(news_id)
        return jsonify({'news': news})

    def delete(self, news_id):
        abort_if_news_not_found(news_id)
        NewsModel(db.get_connection()).delete(news_id)
        return jsonify({'success': 'OK'})
```

В классе списка объектов **NewsList** определяются операции, которые мы можем сделать с **набором** объектов: показать список объектов и добавить объект в список. В этом классе нам потребуется реализовать два метода: get и post без аргументов. Доступ к данным, переданным в теле POST-запроса, осуществляется с помощью парсера аргументов из модуля **reqparse**, который предварительно нужно создать и добавить в него аргументы:

```
parser = reqparse.RequestParser()
parser.add_argument('title', required=True)
parser.add_argument('content', required=True)
parser.add_argument('user_id', required=True, type=int)
```

По-прежнему считаем, что все поля новости являются обязательными. Также укажем тип идентификатора пользователя — целое число. Теперь всю проверку аргументов запроса за нас будет делать модуль **reqparse**.

Таким образом, класс **NewsList** будет иметь вид:

```
class NewsList(Resource):
    def get(self):
        news = NewsModel(db.get_connection()).get_all()
        return jsonify({'news': news})

    def post(self):
        args = parser.parse_args()
        news = NewsModel(db.get_connection())
        news.insert(args['title'], args['content'], args['user_id'])
        return jsonify({'success': 'OK'})
```

После того, как мы создали классы ресурсов, нам надо внести их в настройки нашего **RESTful-API**, указав имя класса и URL. Параметр URL также указывается в угловых скобках:

```
api.add_resource(NewsList, '/news') # для списка объектов
api.add_resource(News, '/news/<int:news_id>') # для одного объекта
```

Давайте проверим такую реализацию RESTful-сервиса с помощью запросов в нашем тестовом файле test.py. Убедитесь, что всё работает, как нужно.

Документация по модулю **flask-RESTful** есть на официальном сайте.

Если ваше приложение построено по правилам REST, вы получите логичную организацию работы с ресурсами, даже если их в приложении большое количество. Клиенты используют простые и понятные URL, а новым клиентам не составит труда разобраться с интерфейсом вашего приложения. Также в мире веб-разработки стандартом становится предоставлять RESTful API для сторонних приложений.

Есть ли подводные камни при использовании REST? Да, и этим камнем является HTML. Дело в том, что спецификация HTML позволяет создавать формы, отправляющие только GET- или POST-запросы. Поэтому для нормальной работы с другими методами (PUT, DELETE) приходится имитировать их искусственно. Часто в этих случаях используют JavaScript-библиотеки, например, JQuery.