Урок WEB. REST-API

REST-API. Понятие. Делаем простое REST-API

Аннотация

На этом занятии мы научимся создавать собственное API для наших веб-приложений.

1. REST API на flask. 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 подняться на уровень выше и рассмотреть архитектуру веб-приложения: 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 учетом того, что все больше приложений развертываются на облачных серверах, после внесенных изменений нужно заново развертывать 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 оправданным.

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 прои 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 еще один подход, «современный взгляд» на проектирование распределенных 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 путем 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. REST

REST — архитектурный стиль программного обеспечения, который определяет правила управления информационными потоками и правила взаимодействия компонентов распределенного 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-я 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 пять 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 соз… первый RESTful веб-сервис.

Перед написанием кода нужно продумать, с какой информацией мы будем работать и на основе каких правил будут строиться URL. Давайте вернемся 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://127.0.0.1:5000/api/news
```

3. Blueprint

Чтобы не смешивать код обработчиков API для новостей с кодом основного приложения, мы воспользуемся механизмом разделения приложения Flask на независимые модули или Blueprint. Как правило, blueprint — логически выделяемый набор обработчиков адресов, который можно вынести в отдельный файл или модуль. Blueprint работает аналогично объекту приложения Flask, но в действительности он не является приложением. Обычно это лишь эскиз для сборки или расширения приложения.

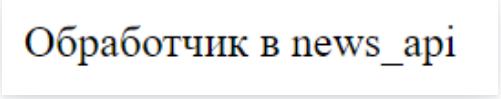
Давайте создадим файл news_api.py и добавим в него следующий код:

```
import flask

from . import db_session
from .news import News

blueprint = flask.Blueprint(
    'news_api',
    __name__,
    template_folder='templates'
)


@blueprint.route('/api/news')
def get_news():
    return "Обработчик в news_api"
```

Мы создали схему, в которую добавили обработчик. Обратите внимание: мы воспользовались не декоратором `app.route`, а аналогичным для схемы — `blueprint.route`. Теперь нужно, чтобы о нашей схеме узнало основное приложение, для чего в main.py перед запуском добавим регистрацию схемы:

```
from data import db_session, news_api


def main():
    db_session.global_init("db/blogs.db")
    app.register_blueprint(news_api.blueprint)
    app.run()
```

Перейдем по адресу http://127.0.0.1:5000/api/news и проверим, что у нас получилось.



4. Сериализация в json

Обычно REST-сервисы обмениваются информацией в формате json. Поэтому надо научить объекты на… превращаться в словарь, для чего надо добавить каждой из моделей метод `to_dict`. Это достаточно … руками, но мы воспользуемся возможностями модуля SQLAlchemy-serializer. Установим его:

```
pip install SQLAlchemy-serializer
```

Теперь мы снова сможем воспользоваться множественным наследованием и добавить к каждой из наших моделей еще один класс-примесь `SerializerMixin`, добавляющий метод `to_dict`.

Заголовок класса `Users` теперь у нас выглядит так:

```
class User(SqlAlchemyBase, UserMixin, SerializerMixin):
```

Не забудьте добавить `SerializerMixin` и в остальные модели.

5. Создание и тестирование сервисов

Теперь допишем в функцию `get_news()` код для получения всех новостей. Давайте пока опустим проверку авторизации пользователя.

```
@blueprint.route('/api/news')
def get_news():
    db_sess = db_session.create_session()
    news = db_sess.query(News).all()
    return jsonify(
        {
            'news':
                [item.to_dict(only=('title', 'content', 'user.name'))
                 for item in news]
        }
    )
```

Как видите, мы можем явно указать, какие поля хотим оставить в получившемся словаре.

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`, который 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. Мы уже неплохо знакомы с этим модулем и помним, что 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:5000/api/news').json())
```

Запустим файл (не забудьте запустить сервер) и увидим, что на наш GET-запрос пришел 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 инфор (например, только id и заголовок), а полную информацию (текст и автора) можно посмотреть с помощью запроса,

которыи мы обработаем далее.

Напишем функцию `get_one_news` для получения информации о новости по ее 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`:

```
@blueprint.route('/api/news/<int:news_id>', methods=['GET'])
def get_one_news(news_id):
    db_sess = db_session.create_session()
    news = db_sess.query(News).get(news_id)
    if not news:
        return jsonify({'error': 'Not found'})
    return jsonify(
        {
            'news': news.to_dict(only=(
                'title', 'content', 'user_id', 'is_private'))
        }
    )
```

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

```
print(get('http://localhost:5000/api/news/1').json())

print(get('http://localhost:5000/api/news/999').json())
# новости с id = 999 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:5000/api/news/q').json())
```

Файл test.py завершится с ошибкой. Так как параметр не типа `int`, в функцию `get_one_news` мы даже не попадем — нет полного совпадения декоратора. На такой запрос сервер вернет ответ со статусом 404 (страница не найдена), но не в формате JSON. Поэтому при попытке преобразовать ответ с помощью метода `json()` программа завершается 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 POS… …ак 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 с модулем requests).

```
@blueprint.route('/api/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', 'is_private']):
        return jsonify({'error': 'Bad request'})
    db_sess = db_session.create_session()
    news = News(
        title=request.json['title'],
        content=request.json['content'],
        user_id=request.json['user_id'],
        is_private=request.json['is_private']
    )
    db_sess.add(news)
    db_sess.commit()
    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 словарем. В нашем случае все ключи заранее известны и все поля обязательны. В противном случае для работы со словарем `request.json` нужно использовать метод словарей `get`. Вспомните, в чем его особенность.

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

```
print(post('http://localhost:5000/api/news').json())

print(post('http://localhost:5000/api/news',
           json={'title': 'Заголовок'}).json())

print(post('http://localhost:5000/api/news',
           json={'title': 'Заголовок',
                 'content': 'Текст новости',
                 'user_id': 1,
                 'is_private': False}).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:

```
@blueprint.route('/api/news/<int:news_id>', methods=['DELETE'])
def delete_news(news_id):
    db_sess = db_session.create_session()
    news = db_sess.query(News).get(news_id)
    if not news:
        return jsonify({'error': 'Not found'})
    db_sess.delete(news)
    db_sess.commit()
    return jsonify({'success': 'OK'})
```

И протестируем ее:

```
print(delete('http://localhost:5000/api/news/999').json())
# новости с id = 999 нет в базе

print(delete('http://localhost:5000/api/news/1').json())
```

Такой «ручной» способ создания RESTful-сервисов достаточно объемный и трудоемкий. 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 объемный код и сделать его более лаконичным. Так и появился легкий способ создавать RESTful-сервисы на flask — модуль Flask-RESTful, который мы рассмотрим уже на следующем уроке.

Еще один важный вопрос не был рассмотрен в этом уроке — вопрос авторизации. Сейчас удалять, добавлять и просматривать новости с помощью нашего API может любой пользователь, а точнее любая программа, которая будет использовать этот API. В настоящих API для выполнения таких действий пользователь должен представиться. Эта тема раскрыта в видео к уроку.