



Соломин Данил
Белокобыльский Илья

Web-технологии

Web-сервисы

Содержание

- Что такое SOAP. Сравнение с REST
- Синтаксис WSDL
- Реализация приложения с SOAP на TypeScript
- Что такое GraphQL, примеры использования
- Что такое gRPC
- Сравнение микросервисов и веб-сервисов

<https://www.tutorialspoint.com/wsdl/index.htm>

<https://www.npmjs.com/package/soap>

<https://graphql.org/learn/>

<https://www.apollographql.com/>

<https://grpc.io/docs/>

Что такое SOAP, WSDL и веб-сервисы ³

Веб-сервисы - надстройка поверх HTTP (FTP, SMTP, ...)

Чтобы связать и предоставить возможность одним приложениям обмениваться данными с другими, и были придуманы веб-сервисы.

Иначе, веб-сервисы — это реализация стандартных интерфейсов обмена данными между различными приложениями.

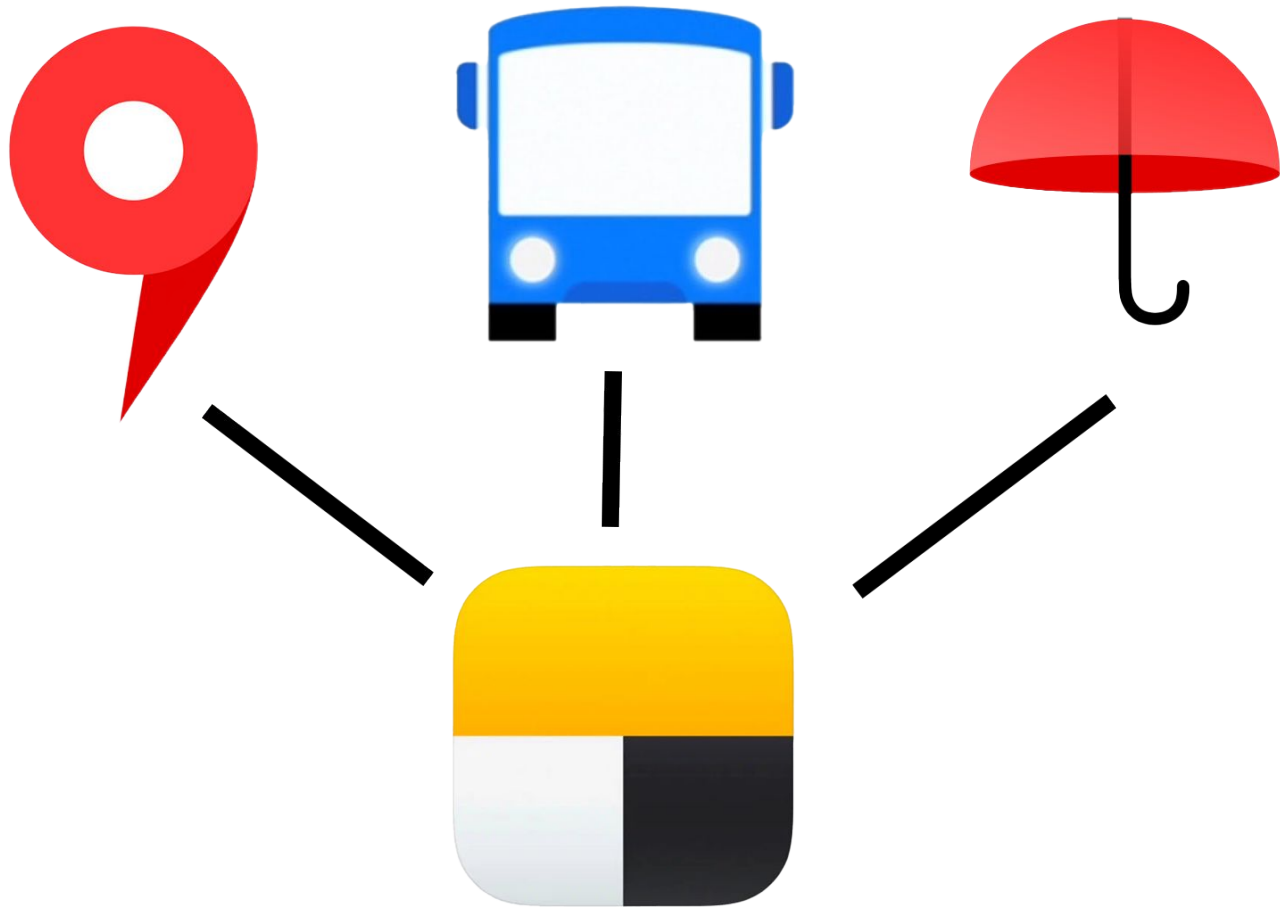
При этом неважно, на каком языке написаны и где располагаются эти приложения

Протоколы веб-сервисов

SOAP (Simple Object Access Protocol) - протокол обмена структурированными сообщениями в формате XML.

SOAP – это не стиль написания кода, как REST, а протокол - надстройка над другим прикладным протоколом (HTTP, FTP ...).

В этой лекции мы остановимся больше на SOAP, но также коснемся и других протоколов: GraphQL и gRPC.



Что такое SOAP, WSDL и веб-сервисы ⁶

WSDL (Web Services Description Language) – Это специальный XML-подобный язык для описания веб сервисов.

```
<message name="getTermRequest">  
  <part name="term" type="xs:string" />  
</message>
```

```
<message name="getTermResponse">  
  <part name="value" type="xs:string"/>  
</message>
```

```
<portType name="glossaryTerms">  
  <operation name="getTerm">  
    <input message="getTermRequest" />  
    <output message="getTermResponse" />  
  </operation>  
</portType>
```



GraphQL

REST

SOAP

SOAP и REST. Специфика.

Сравнение не совсем корректно: REST - стиль проектирования, SOAP - протокол.

Специфика SOAP — это формат обмена данными. С SOAP это всегда SOAP-XML, который представляет собой XML.

Специфика REST — всегда использует HTTP и все его функции — методы запросов, заголовки запросов, ответы, заголовки ответов и т. д.

SOAP и REST. Язык.

SOAP использует WSDL — язык описания веб-сервисов и доступа к ним, основанный на XML.

REST не имеет стандартного языка определения сервиса. Хотя WADL — один из первых стандартов, он не очень популярен. Более популярно использование Swagger или Open API.

SOAP и REST. Протокол.

SOAP не накладывает никаких ограничений на тип транспортного протокола. Вы можете использовать либо HTTP, либо любой другой (FTP, SMTP ...).

REST подразумевает использование только HTTP и HTTPS

SOAP и REST. Реализация.

RESTFful веб-сервисы обычно проще реализовать, чем веб-сервисы на основе SOAP. REST обычно использует JSON, который легче анализировать и обрабатывать.

В случае SOAP необходимо определить сервис с использованием WSDL. При обработке и анализе сообщений SOAP-XML возникают большие накладные расходы.

Преимущества SOAP

- ~~Убивает бактерии~~
- Обладает высокой расширяемостью, по сравнению с REST - используются только те фрагменты, что важны для решения задачи
- Из коробки поддерживает стандарт WS Security
- Встроенная обработка ошибок
- Можно использовать поверх других прикладных протоколов

Где используют SOAP

- Необходима надежная и структурированная передача информации
- Не хватает CRUD системы REST
- Нужно поддерживать outdated системы с большим количеством кода.
- В основном SOAP используют в крупных банках, таких как Сбер. Хотя почти наверное вы не столкнетесь с SOAP, это интересная технология

Структура WSDL

WSDL состоит из 3
ОСНОВНЫХ ТИПОВ
ЭЛЕМЕНТОВ.

- Типы (Types)
- Операторы (Operators)
- Бинды (Bindings)

Далее мы рассмотрим
каждый из типов

```
<definitions>
  <types>
    <!-- definition of types... -->
  </types>
  <message>
    <!-- definition of messages... -->
  </message>
  <portType>
    <operation>
      <!-- definition of operations... -->
    </operation>
  </portType>
  <binding>
    <!-- definition of bindings... -->
  </binding>
  <service>
    <!-- definition of services... -->
  </service>
</defenitions>
```

<definitions>

Элемент <definitions> должен быть корневым элементом всех документов WSDL. Он определяет имя веб-сервиса.

```
<definitions
```

```
  name="ServiceName"
```

```
  targetNamespace="http://localhost:3000/wsdl"
```

```
  xmlns="http://schemas.xmlsoap.org/wsdl/"
```

```
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
```

```
  xmlns:tns="http://localhost:3000/wsdl">
```

```
<!-- ..... -->
```

```
</defenitions>
```


<definitions>

Из примера видно, что definitions —

- Указывает, что этот документ называется `ServiceName` .
- Указывает атрибут `targetNamespace` — соглашение XML-схемы, которое позволяет документу WSDL ссылаться на себя .
- Указывает пространство имен по умолчанию. Поэтому все элементы без префикса пространства имен, такие как `message` или `portType` , считаются частью пространства имен WSDL по умолчанию.
- Определяет многочисленные пространства имен, которые используются в оставшейся части документа.

<types>

Веб-сервис должен определить свои входные и выходные данные. Элемент WSDL <types> отвечает за определение типов данных, используемых веб-службой.

```
<types>
```

```
<schema
```

```
  targetNamespace="http://www.example.org/wsdl/"
```

```
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
<element>
```

```
  <complexType name="Person">
```

```
    <sequence>
```

```
      <element name="firstName" type="string"/>
```

```
      <element name="lastName" type="string"/>
```

```
      <element name="age" type="int"/>
```

```
    </sequence>
```

```
  </xsd:complexType>
```

```
</element>
```

```
<element>
```

```
  <complexType name="TradePrice">
```

```
    <all>
```

```
      <element name="price" type="float"/>
```

```
    </all>
```

```
  </xsd:complexType>
```

```
</element>
```

```
</schema>
```

```
</types>
```

<types>

Элемент `types` описывает все типы данных, используемые между клиентом и сервером.

WSDL не привязан исключительно к конкретной системе типизации.

WSDL использует спецификацию схемы XML W3C в качестве выбора по умолчанию для определения типов данных.

Если служба использует только встроенные простые типы XML-схемы, такие как строки и целые числа, то элемент типов не требуется.

WSDL позволяет определять типы в отдельных элементах, чтобы их можно было повторно использовать с несколькими веб-службами.

<message>

Каждая веб-служба имеет два сообщения: входное и выходное.

Входные данные описывают параметры веб-службы, выходные данные описывают возвращаемые данные веб-службы.

Каждое сообщение содержит ноль или более параметров <part>, по одному на каждый параметр функции веб-сервиса.

Каждый параметр <part> связан с конкретным типом, определенным в элементе контейнера <types> .

<message>

Элемент <message> описывает данные, которыми обмениваются веб-сервис и клиент.

```
<message name="SayHelloRequest">  
  <part name="firstName" type="string"/>  
</message>
```

```
<message name="SayHelloResponse">  
  <part name="greeting" type="string"/>  
</message>
```

<portType>

Элемент <portType> объединяет элементы сообщения.

Например, <portType> может объединить один запрос и один ответ в одну операцию запроса/ответа. Это используется в SOAP. portType может определять несколько операций.

```
<portType name="Hello_PortType">  
  <operation name="sayHello">  
    <input message="tns:sayHelloRequest"/>  
    <output message="tns:sayHelloResponse"/>  
  </operation>  
</portType>
```

<portType>

Элемент portType определяет одну операцию под названием sayHello.

Операция состоит из одного входного сообщения SayHelloRequest и выходное сообщение SayHelloResponse .

Также поддерживается сообщения в 1 сторону

```
<portType name="Hello_PortType">  
  <operation name="sayHello">  
    <input message="tns:sayHelloRequest"/>  
    <output message="tns:sayHelloResponse"/>  
  </operation>  
</portType>
```


<portType>

Поддерживается также формат запроса
“Уведомление”

<portType>

<operation name="nmtoken">

<output message="qname"/>

</operation>

</portType>

<portType>. Ответ на запрос

Служба получает сообщение и отправляет ответ. Таким образом, операция имеет один входной элемент, за которым следует один выходной элемент.

```
<portType >  
  <operation name="nmtoken">  
  
    <input message="qname" />  
    <output message="qname" />  
    <fault message="qname" />  
  
  </operation>  
</portType>
```

<binding>

<binding> предоставляет конкретные сведения о том, как операция portType фактически будет передаваться по сети.

Элемент <binding> имеет два атрибута: атрибут имени и типа .

```
<binding name = "Hello_Binding" type = "tns:Hello_PortType">
```

Атрибут name определяет имя бинда, а атрибут type указывает на portType для привязки, в данном случае порт «tns:Hello_PortType» .

<binding>

```
<binding name="Hello_Binding" type="tns:Hello_PortType">
  <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="sayHello">
    <soap:operation soapAction="sayHello"/>
    <input>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:examples:helloservice" use="encoded"/>
    </input>
    <output>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:examples:helloservice" use="encoded"/>
    </output>
  </operation>
</binding>
```

<binding>

<soap:binding> - указание формата и протокола. Этот элемент указывает, что привязка будет доступна через SOAP. Атрибут `style` указывает общий стиль формата сообщения SOAP. Атрибут `transport` указывает на передачу SOAP-сообщений. *`http://schemas.xmlsoap.org/soap/http`* указывает на HTTP

<soap:operation> - привязывание операции к реализации. `soapAction` указывает, что HTTP-заголовок SOAPAction используется для идентификации службы.

<soap:body> - детали входящих и исходящих сообщений

<port>

Определяет отдельный адрес для binding.
Имеет два атрибута - name, binding.

```
<port name="HelloPort" binding="tns:HelloBinding">  
  <soap:address location="http://localhost:3000/wsdl"/>  
</port>
```

<service>

Определяет поддерживаемые службой порты по одному для каждого из протоколов.
Включает в себя документацию

```
<service name="Hello_Service">  
  <documentation>WSDL File for HelloService</documentation>  
  <port name="Hello_Port" binding="tns:Hello_Binding">  
    <soap:address location="http://localhost:3000/wsdl"/>  
  </port>  
</service>
```


Hello World service

В качестве практики напишем простой сервис для вывода hello world

Hello World. WSDL

31

Напишем definitions, в который укажем пространство имен и используемые типы сообщений

```
// hello_world_service.wsdl
```

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
```

```
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
```

```
xmlns:tns="http://localhost:3000/wsdl" targetNamespace="http://localhost:3000/wsdl">
```

```
  <message name="HelloRequest">
```

```
    <part name="name" type="string"/>
```

```
  </message>
```

```
  <message name="HelloResponse">
```

```
    <part name="greeting" type="string"/>
```

```
  </message>
```

```
  <message name="ByeRequest">
```

```
    <part name="name" type="string"/>
```

```
  </message>
```

```
  <message name="ByeResponse">
```

```
    <part name="farewell" type="string"/>
```

```
  </message>
```

Hello World. WSDL

32

Теперь напишем
portType, в котором
определим
операции sayHello и
sayBye

```
// hello_world_service.wsdl
<portType name="HelloPortType">
  <operation name="sayHello">
    <input message="tns:HelloRequest"/>
    <output message="tns:HelloResponse"/>
  </operation>
  <operation name="sayBye">
    <input message="tns:ByeRequest"/>
    <output message="tns:ByeResponse"/>
  </operation>
</portType>
```

Hello World. WSDL

33

Напишем binding
с указанием
протокола HTTP и
свяжем операции
sayHello и sayBye с
реализациями

```
// hello_world_service.wsdl
<binding name="HelloBinding" type="tns:HelloPortType">
  <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="sayHello">
    <soap:operation soapAction="sayHello"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
  <operation name="sayBye">
    <soap:operation soapAction="sayBye"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
```

Hello World. WSDL

И в итоге закончим определением адреса сервиса, свяжем его с binding

```
// hello_world_service.wsdl
<service name="HelloService">
  <port name="HelloPort" binding="tns:HelloBinding">
    <soap:address location="http://localhost:3000/wsdl"/>
  </port>
</service>
</definitions>
```

Hello World. TS

Теперь реализуем серверную и клиентские части.
Инициализируем стандартный express сервер

```
// server.ts

import express from "express";
import cors from "cors";
import soap from "soap";
import { readFileSync } from "fs";

const app = express();

app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

Hello World. TS

36

Напишем сервисы
для sayHello и
sayBye

```
// server.ts
const myService = {
  HelloService: {
    HelloPort: {
      sayHello: function (args: IHello) {
        return {
          greeting: "Hello " + args.name,
        };
      },
      sayBye: function (args: IBye) {
        return {
          farewell: "Bye " + args.name,
        };
      },
    },
  },
};
```


Запустим сервер с указанием WSDL файла в soap.listen

```
// server.ts
// Routers

const xml = readFileSync("static/service.wsdl", "utf-8");

app.listen(process.env.PORT || 4000, async () => {
  const server = soap.listen(app, "/wsdl", myService, xml, function () {
    console.log("Running on port " + (process.env.PORT || 4000));
  });
});
```

Протестируем работу сервера, написав клиент

```
// client.ts

import soap from "soap";

const url = "http://localhost:4000/wsdl?wsdl";

(async () => {
  const client = await soap.createClientAsync(url, {});
  const [hello] = await client.sayHelloAsync({ name: "Ilya" });
  const [bye] = await client.sayByeAsync({ name: "Ilya" });
  console.log(hello); // { greeting: 'Hello Ilya' }
  console.log(bye); // { farewell: 'Bye Ilya' }
})();
```

GraphQL. Проблема REST

Представьте, что вам нужно отобразить список записей (posts), и под каждым опубликовать список лайков (likes), включая имена пользователей и аватары. На самом деле, это не сложно, вы просто измените API posts так, чтобы оно содержало массив likes, в котором будут объекты-пользователи.

Но затем, при разработке мобильного приложения, оказалось что из-за загрузки дополнительных данных приложение работает медленнее. Так что вам теперь нужно два endpoint, один возвращающий записи с лайками, а другой без них.

Добавим ещё один фактор: оказывается, записи хранятся в базе данных MySQL, а лайки в Redis! Что же теперь делать?!

```
[
  {
    title: 'My First Post',
    likes: [
      {
        name: 'Sacha',
        avatar: 'sacha.jpg'
      },
      {
        name: 'Mike',
        avatar: 'mike.jpg'
      }
    ]
  },
  {
    title: 'Another Great Post',
    likes: [
      {
        name: 'Julia',
        avatar: 'julia.jpg'
      },
      {
        name: 'Sacha',
        avatar: 'sacha.jpg'
      }
    ]
  }
],
]
```

My First Post

Liked by:



Sacha



Mike

Another Great Post

Liked by:



Julia



Sacha

GraphQL. Решение

Решением и оказываются более сложные протоколы, такие как GraphQL.

На практике GraphQL API построен на трёх основных строительных блоках: на схеме (schema), запросах (queries) и распознавателях (resolvers).

Запросы (Queries)

42

```
query ExampleQuery($name: String!) { {  
  viewer {  
    login  
    url  
    updatedAt  
    repository(name: "tetris") {  
      id  
      name  
      isPrivate  
    }  
  }  
}  
  
  "data": {  
    "viewer": {  
      "login": "kyzinatra",  
      "url": "https://github.com/kyzinatra",  
      "updatedAt": "2023-12-01T21:49:39Z",  
      "repository": {  
        "id": "R_kgDOKOz6GA",  
        "name": "tetris",  
        "isPrivate": true  
      }  
    }  
  }  
}
```

Запросы (Queries)

43

```
query ExampleQuery($q: String!,
$type: [SearchType!]!) {
  search(q: $q, type: $type) {
    tracks {
      edges {
        node {
          name,
          href,
          durationMs
          artists {
            name,
            genres
          }
        }
      }
    }
  }
}
```

```
{
  "data": {
    "search": {
      "tracks": {
        "edges": [
          {
            "node": {
              "name": "Space Oddity",
              "href":
                "https://api.spotify.com/v1/tracks/5kBAk4dS
                QX4aXbTqjaPvF6",
              "durationMs": 319186,
              "artists": [
                {
                  "name": "Chris Hadfield",
                  "genres": []
                }
              ]
            }
          },
          {
            "node": {
```

Распознаватели (Resolvers)

GraphQL не может знать что делать с входящим запросом, если ему не объяснить при помощи распознавателя (resolver).

Используя распознаватель GraphQL понимает, как и где получить данные, соответствующие запрашиваемому полю.

Распознаватели (Resolvers)

```
Post: {  
  author(post) {  
    return Users.find({ id: post.authorId })  
  },  
  commentsCount(post) {  
    return Comments.find({ postId: post.id }).count();  
  }  
},
```

Схема запроса GraphQL и структура БД никак не связаны. Другими словами, в базе данных может не существовать полей `author` или `commentsCount`, но мы можем "симулировать" их благодаря силе распознавателей.

Вы можете писать любой код внутри распознавателя. Так что вы можете изменять содержимое базы данных; такие распознаватели называют изменяющими (`mutation`).

Схема (schema)

Все это становится возможным благодаря типизированной схеме данных GraphQL.

Загляните [в документацию GraphQL](#) (англ), если хотите узнать больше.

И познакомьтесь с [Apollo](#) – платформой для построения графовых цепочек, кстати, красиво и удобно данные с Spotify и Github я брал как раз с помощью Apollo.

Sort By: ☒ Popularity ☐ Alphabetical

JavaScript

Go

PHP

**Java /
Kotlin**

C# / .NET

Python

Rust

Ruby

**Swift /
Objective-
C**

Elixir

Scala

Flutter

Clojure

Haskell

C / C++

Elm

**OCaml /
Reason**

Erlang

Ballerina

Julia

R

Groovy

Perl

D

GraphQL и графовые БД

GraphQL не имеет ничего общего с графовыми БД типа Neo4j.

"Graph" отражает идею получения контента, проходя сквозь граф API, используя поля и подполя;

"QL" – "query language" – "язык запросов".

Если вы не достигли болевых точек REST API или SOAP, решением которых занимается GraphQL, то можете не волноваться.

Для внедрения GraphQL в проект поверх REST не нужны будут сложные изменения, поэтому стоит его попробовать в небольшой части проекта, если это возможно.

GraphQL - можно ли без библиотек?

51

Конечно! GraphQL это только спецификация, вы можете использовать его с любой библиотекой на любой платформе, используя готовый клиент (к примеру, у [Apollo](#) есть клиенты для iOS, Angular и другие) или вручную отправляя запросы на сервер GraphQL.

GraphQL создан Meta*

52

Ещё раз, GraphQL является всего лишь спецификацией, это значит что вы можете использовать его не используя ни одной строки кода, написанной Meta*.

Наличие поддержки Meta* это хороший плюс для экосистемы GraphQL. Но сейчас сообщество достаточно большое чтобы поддерживать GraphQL, даже если Meta* перестанет его использовать.

* компания признана экстремистской на территории РФ

Точно ли такой доступ безопасен?

Так как вы пишете свои распознаватели, вы можете разрешать любые проблемы безопасности на этом уровне.

К примеру, если вы позволите клиенту использовать параметр `limit` для указания количества документов, которые он запрашивает, вы скорее всего захотите контролировать это число для избежания атак типа "отказ в обслуживании" когда клиент будет запрашивать миллионы документов снова и снова.

↓ Weekly Downloads

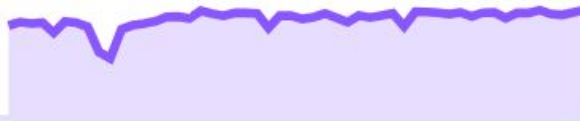
29 246 367



Express(x89)

↓ Weekly Downloads

11 528 858



GraphQL(x32)

↓ Weekly Downloads

1 539 534



NestJS(x4.5)

↓ Weekly Downloads

360 569



SOAP

Что такое gRPC

gRPC - система удаленного вызова процедур. Используется в основном в бэкенде для межсервисного взаимодействия.

Базируется на HTTP v2, а в качестве языка описания использует Protocol Buffers (.proto)

Возможности gRPC

- Унарная (обычная) передача данных
- Поточковая передача данных (stream) - в т.ч. двусторонняя
- Таймауты, доступные обеим сторонам (клиенту и серверу)
- Отмена любой стороной в любое время
- Передача метаданных (аутентификация и пр.)

Что такое Protocol Buffers

Это протокол сериализации (т.е. преобразования данных в биты).

Описывается .proto файлами - это как WSDL - язык описания контракта.

Он обладает статической типизацией

Что такое Protocol Buffers

Основные понятия: сообщение, сервис и грс.

Сообщение - передаваемая информация,

сервис - интерфейс,

грс - методы, необходимые для реализации этого интерфейса.

То есть получается, что мы передаем данные - сообщения в методы - грс, которые объединены в сервис

Пример proto файла

// The greeter service definition.

service Greeter {

// Sends a greeting

rpc SayHello (HelloRequest) **returns** (HelloReply) {}

}

// The request message containing the user's name.

message HelloRequest {

string name = 1;

}

// The response message containing the greetings

message HelloReply {

string message = 1;

}

И где это использовать?

- Передача больших данных частями через стриминг
- Реализация серверных пушей
- Ускорение передачи данных - актуально для микросервисов

Пример открытого gRPC API: [Yandex GPT](#)

gRPC для TypeScript

После написания proto файла необходимо написать сервис.

Есть 2 варианта использования .proto:

1. Статическая генерация с использованием protoc
2. Динамическая генерация

Что такое микросервисы?

Способы организации кода:

Монолит - вся программа - единое целое.

Микросервисы - программа разделена на несколько маленьких частей, (слабо) связанных между собой.

Например, сервис аутентификации, сервис товаров и сервис чего-то там еще

Микросервисы и веб-сервисы

Микросервисы - программная архитектура, не зависит от протоколов

Веб-сервис - программная система, в основе - веб-протоколы.

Веб-сервис - любое приложение, которое работает на основе REST, SOAP, GraphQL, gRPC и т.д. (в роли сервера), независимо от архитектуры.

Микросервисы и веб-сервисы

Обычно микросервис поверх REST или gRPC - веб-сервис - он реализует стандартизированный интерфейс обмена данными

Веб-сервис же может быть чем угодно - как монолитом, так и микросервисом. Главное - чтобы он работал поверх HTTP, имел свой идентификатор в сети (URI) и реализовывал какой-то стандартизированный интерфейс

Задача на реализацию

Реализовать SOAP API на TypeScript. Для примера возьмем API ресторана. Необходимые функции: просмотр меню, создание заказа, получение статуса заказа.

Для написания предлагается использовать <https://www.npmjs.com/package/soap>

Реализация функций в Typescript

66

Для начала
добавим
реализацию
выполнения задач
в виде функций в
typescript.
Добавление заказа,
вывод меню.

```
// server.ts
const myService = {
  ShopService: {
    ShopPort: {
      newOrder: function (args: INewOrder) {
        return newOrderHandler(args);
      },
      getMenu: function (args: IGetMenu) {
        return getMenuHandler()
      },
    },
  },
};
```

Подключение сервиса

Чтобы сервис заработал на HTTP протоколе нам нужно установить express и прописать следующий boilerplate код

```
// server.ts
const xml = readFileSync("pizza/service/service.wsdl", "utf-8");

app.listen(process.env.PORT || 3000, async () => {
  const server = soap.listen(app, "/wsdl", myService, xml, function ()
  {
    console.log("Running on port " + (process.env.PORT || 3000));
  });
  server.log = function (type: string, data: any) {
    console.log("SOAP service " + type + " log");
  };
});
```

Написание wsdl файла

68

По описанным
ранее сценарием
напишем блок
message.

```
// service.wsdl
<message name="newOrderRequest">
  <part name="name" type="string"/>
</message>
<message name="newOrderResponse">
  <part name="number" type="string"/>
</message>
<message name="getMenuRequest">
  <part name="name" type="string"/>
</message>
<message name="getMenuResponse">
  <part name="menu" type="string"/>
</message>
```


Написание wsdl файла

Перейдем к определению <portType> и <binding>

```
// service.wsdl
```

```
<portType name="ShopPortType">
```

```
  <operation name="newOrder">
```

```
    <input message="tns:newOrderRequest"/>
```

```
    <output message="tns:newOrderResponse"/>
```

```
  </operation>
```

```
  <operation name="getMenu">
```

```
    <input message="tns:getMenuRequest"/>
```

```
    <output message="tns:getMenuResponse"/>
```

```
  </operation>
```

```
</portType>
```

Написание wsdl файла

70

```
// service.wsdl
<binding name="ShopBinding" type="tns:ShopPortType">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="newOrder">
    <soap:operation soapAction="newOrder"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
  <operation name="getMenu">
    <soap:operation soapAction="getMenu"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
```

Написание wsdl файла

Наконец определим названия сервиса и путь до него.

```
// service.wsdl
<service name="ShopService">
  <port name="ShopPort" binding="tns:ShopBinding">
    <soap:address location="http://localhost:3000/wsdl"/>
  </port>
</service>
```

Запуск сервера

Пропишем **npm start** для запуска. На запрос за НОВЫМ заказом, получаем его номер в системе

```
// client.ts
import soap from "soap";
const url = "http://localhost:3000/wsdl?wsdl";

(async () => {
  const client = await soap.createClientAsync(url, {});
  const [response] = await client.newOrderAsync({ name: "Ilya" });
  console.log(response);
})();
```

```
$ npm run start_client

> super-api@0.0.0 start_client
> ts-node --esm client.ts

{ number: '88' }
```

Реализация gRPC, .proto-файл

```
syntax = "proto3";
```

```
package delivery;
```

```
message Pizza {  
    int64 price = 1;  
    repeated string ingredients = 2;  
}
```

```
message OrderRequest {  
    repeated Pizza pizzas = 1;  
    int64 number = 2;  
    int64 id = 3;  
}
```

```
message CourierPositionResponse {  
    int64 waitTime = 1;  
}
```

```
service Delivery {  
    rpc CreateDelivery(OrderRequest) returns (stream  
    CourierPositionResponse) {}  
}
```

Реализация gRPC. Генерация TS

74

```
protoc --plugin=$(npm root)/.bin/protoc-gen-ts_proto \  
--ts_proto_out=generated \  
--ts_proto_opt=outputServices=grpc-js \  
--ts_proto_opt=esModuleInterop=true \  
-I=./service/delivery.proto
```

Реализация gRPC. Создание заказа

75

```
// grpc-server.ts

const createDelivery = (call: grpc.ServerWritableStream<OrderRequest,
CourierPositionResponse>): void => {
  orders[call.request.number] = <TOrder>{
    waitTime: 10,
    // ...
  };
  const send = () => {
    orders[call.request.number].waitTime--;
    call.write(<CourierPositionResponse>{
      waitTime: orders[call.request.number].waitTime
    })
    if (orders[call.request.number].waitTime > 0) {
      setTimeout(send, 1000);
    } else {
      call.end();
    }
  };
  setTimeout(send, 1000);
}
```

Написание gRPC клиента

```
// index.ts
// gRPC client
const deliveryClient = new DeliveryClient(
  "localhost:50051",
  grpc.credentials.createInsecure()
);

// gRPC request
const call =
deliveryClient.createDelivery(<OrderRequest>{/* ... */});
call.on('data', data => {
  console.log('Courier will come in ', data.waitTime, '
minutes');
  // print time for cooks...
  // other logic...
});
call.on('status', status => {
  console.log("gRPC request status change", status)
});
```


Реализация GraphQL

77

Напишем схему для GraphQL API. Она включает информацию о заказе с возможностью его получения.

```
// graphql.ts
const schema = buildSchema(`
  type Query {
    count: Int!
    order(order: Int!): Order
  }
  type Order {
    id: ID!
    order: Int!
    list: [String!]!
    price: Float!
    status: String
    waitTime: Int!
  }
`);
```

Реализация GraphQL

78

Подключим сервер к express простой конструкцией

```
var app = express();  
app.use(  
  "/graphql",  
  graphqlHTTP({  
    schema: schema,  
    rootValue,  
    graphql: true,  
  })  
);  
  
app.listen(3030);
```

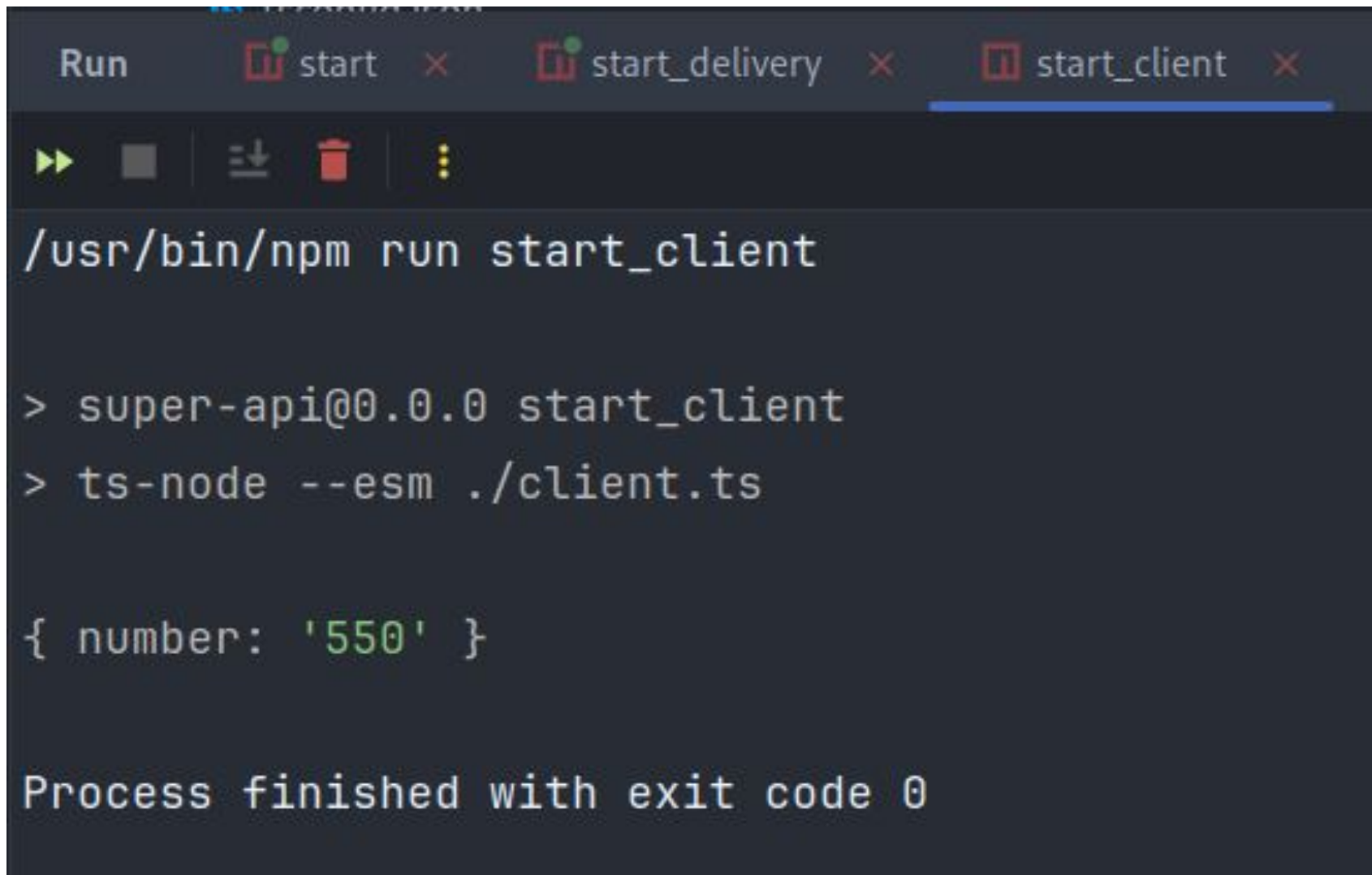
Демонстрация работы

Запустим сервер и postman и попробуем получить данные. Видно, что Postman загрузил описание всего API, что позволяет удобно с ним работать



Демонстрация работы. Клиент

80



The image shows a VS Code terminal window with three tabs: 'Run', 'start', and 'start_client'. The 'start_client' tab is active. The terminal output shows the command `/usr/bin/npm run start_client` being executed, followed by the command `> super-api@0.0.0 start_client` and `> ts-node --esm ./client.ts`. The output of the script is a JSON object `{ number: '550' }`. At the bottom, it says 'Process finished with exit code 0'.

```
Run start x start_delivery x start_client x
>> [icons]
/usr/bin/npm run start_client

> super-api@0.0.0 start_client
> ts-node --esm ./client.ts

{ number: '550' }

Process finished with exit code 0
```

Демонстрация работы. Ресторан

81

```
New order {  
  id: 'Ilya',  
  number: 550,  
  waitTime: 1200,  
  list: [  
    { price: 12, ingredients: [Array] },  
    { price: 15, ingredients: [Array] }  
  ]  
}
```

SOAP service replied log

```
Courier will come in 9 minutes  
Courier will come in 8 minutes  
Courier will come in 7 minutes  
Courier will come in 6 minutes
```

```
Courier will come in 5 minutes  
Courier will come in 4 minutes  
Courier will come in 3 minutes  
Courier will come in 2 minutes  
Courier will come in 1 minutes  
Courier will come in 0 minutes  
gRPC request status change {  
  code: 0,  
  details: 'OK',  
  metadata: Metadata { internalRepr: Map(0) {}, options: {} }  
}
```

Демонстрация работы. Доставка

82

```
Run start x start_delivery x start_client x
>> [stop] [debug] [trash] [menu]
/usr/bin/npm run start_delivery

> super-api@0.0.0 start_delivery
> ts-node --esm ./delivery.ts

Running a GraphQL API server at http://localhost:3030/graphql
New delivery {
  waitTime: 10,
  number: 286,
  id: 'Ilya',
  list: [
    { price: 14, ingredients: [Array] },
    { price: 12, ingredients: [Array] }
  ]
}
```

Демонстрация работы

83

The screenshot displays a GraphQL client interface with the following components:

- URL Bar:** `http://localhost:3030/graphql`
- Query Tab:** Active tab showing the query editor and field selection.
- Search:** A search bar labeled "Search fields".
- Field Selection:**
 - ☒ `count` `Int!`
 - ☒ `order` `Order`
 - ☒ `order` `Int!` (value: `550`, marked `ARG`)
 - ☒ `waitTime` `Int!`
- Query Editor:**

```
1 query Order {  
2   count  
3   order(order: 550) {  
4     waitTime  
5     number  
6     id  
7   }  
8 }
```
- Variables:** An empty section for defining variables.
- Body Tab:** Active tab showing the JSON response.
 - Format:** Pretty
 - Response:**

```
1 {  
2   "data": {  
3     "count": 1,  
4     "order": {  
5       "waitTime": 0,  
6       "number": 550,  
7       "id": "Ilya"  
8     }  
9   }  
10 }
```
- Status Bar:** `Status: 200 OK Time: 10.31 ms Size: 250 B`

Вопросы для самопроверки

- Из каких компонентов состоит сообщение в SOAP?
- Зачем нужен SOAP, если есть REST?
- Какой протокол лежит в основе SOAP?
- Из каких частей состоит WSDL-документ?
- Как в JS поднять SOAP-сервис на основе WSDL-документа?
- Какую проблему REST решает GraphQL?
- Что такое Распознаватели?
- Чем gRPC лучше REST, а чем хуже?
- Как реализовать стриминг в gRPC?
- Что такое Protocol Buffers?
- Что такое .proto файлы?
- Какие способы интеграции .proto файлов есть в JS?
- Как генерировать typescript-код на основе .proto?
- В чем разница между микросервисами и веб-сервисами?