

Веб-программирование

API

API

Проблема: если программы написаны на разных языках, их может быть трудно «подружить» и сделать так, чтобы они могли друг с другом «общаться»

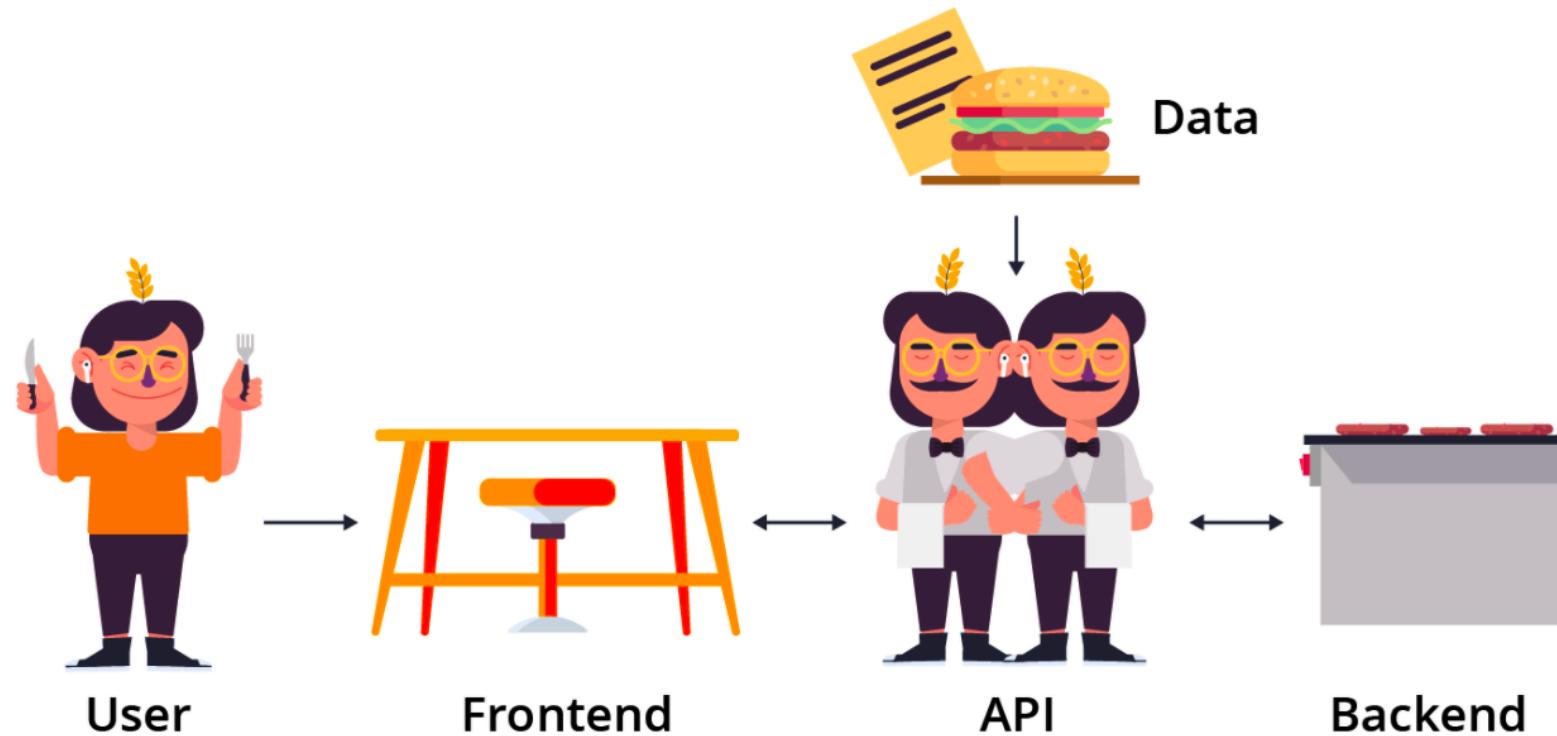
Решение: API

API (Application Programming Interface) — это набор фич, которые одна программа представляет всем остальным.

В случае с клиент-серверным общением API может выступать как набор ссылок, по которым клиент обращается на сервер:

- `POST /api/v1.0/users` — для создания пользователя;
- `GET /api/v1.0/users` — для получения списка пользователей.

API



API

API может использоваться не только для общения браузера и сервера, но и в принципе для общения разных программ друг с другом.

Примеры:

- Модули одной и той же программы
- Микросервисы
- Операционная система и программы
- Arduino SDK

API

Идеальное API:

- **Бесшовное** — единожды написанную функциональность можно использовать везде
- **Быстрое** — чем меньше времени тратится на общение и выполнение нужных действий, тем лучше спроектировано API
- **Понятное** — чем точнее названы функции, методы или ссылки в API, тем меньше заблуждений и ошибок будет возникать при работе с ним
- **Полное** — данных ровно столько, сколько нужно

(Идеального API не существует, но к нему нужно стремиться 😊)

Стандарты API

- SOAP
- REST
- RPC (gRPC)
- GraphQL

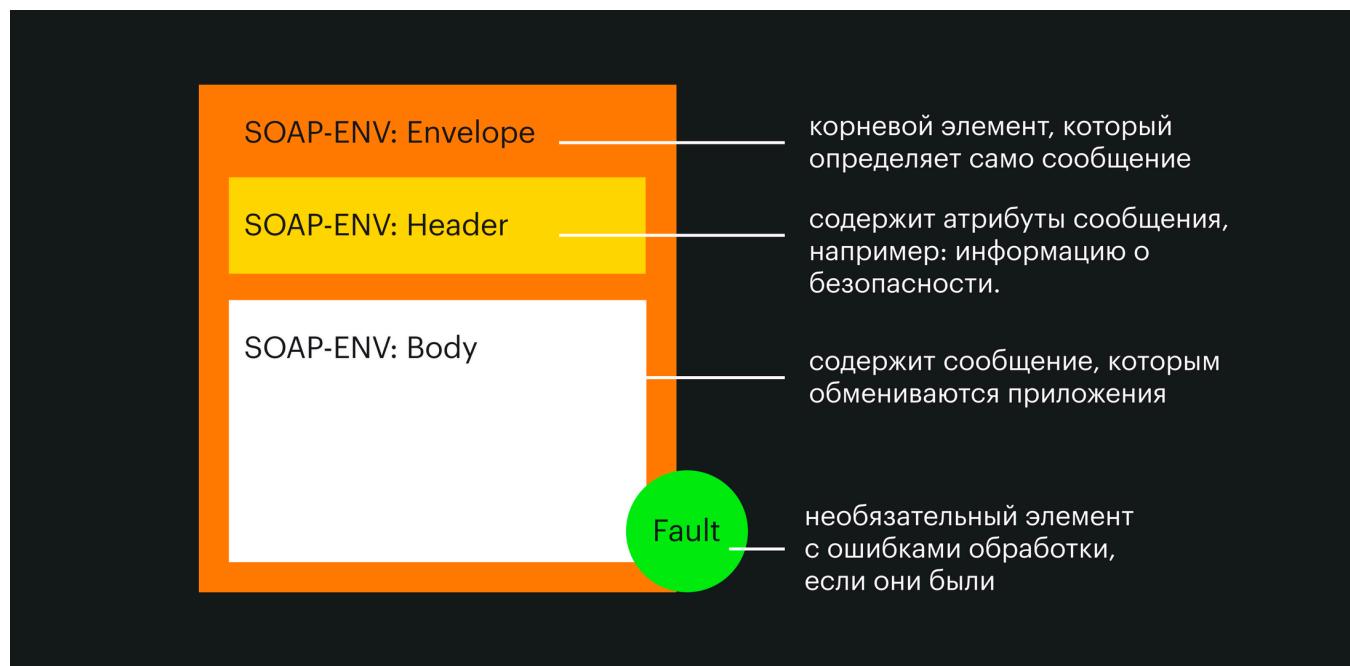
	RPC	SOAP	REST	GraphQL
Organized in terms of	local procedure calling	enveloped message structure	compliance with six architectural constraints	schema & type system
Format	JSON, XML, Protobuf, Thrift, FlatBuffers	XML only	XML, JSON, HTML, plain text,	JSON
Learning curve	Easy	Difficult	Easy	Medium
Community	Large	Small	Large	Growing
Use cases	Command and action-oriented APIs; internal high performance communication in massive micro-services systems	Payment gateways, identity management CRM solutions financial and telecommunication services, legacy system support	Public APIs simple resource-driven apps	Mobile APIs, complex systems, micro-services

SOAP

SOAP (Simple Object Access Protocol) — структурированный формат обмена данными.

Работает поверх протоколов HTTP и SMTP.

Для отражения структуры каждого сообщения используется XML:



SOAP

Сообщение-запрос к интернет-магазину:

```
XML
1 <?xml version="1.0" encoding="utf-8"?>
2 <soap:Envelope
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
6   <soap:Body>
7     <getOrderDetails xmlns="https://example-store.com/orders">
8       <orderID>42</orderID>
9     </getOrderDetails>
10    </soap:Body>
11 </soap:Envelope>
```

Ответ:

```
XML
1 <?xml version="1.0" encoding="utf-8"?>
2 <soap:Envelope
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
6   <soap:Body>
7     <getOrderDetailsResponse xmlns="https://example-store.com/orders">
8       <getOrderDetailsResult>
9         <orderID>42</orderID>
10        <userID>43</userID>
11        <dateTime>2020-10-10T12:00:00</dateTime>
12        <products>
13          <productID>1</productID>
14          <productID>23</productID>
15          <productID>45</productID>
16        </products>
17      </getOrderDetailsResult>
18    </getOrderDetailsResponse>
19  </soap:Body>
20 </soap:Envelope>
```

SOAP

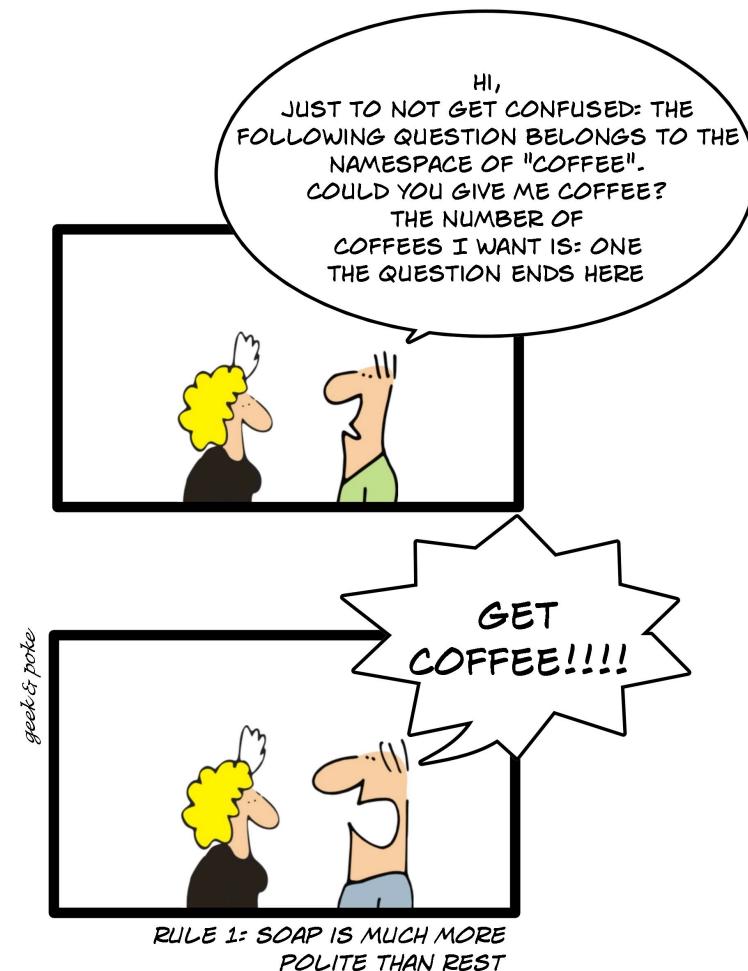
Плюсы:

- не зависит от методов передачи
- есть структура сообщения

Минусы:

- многословен
- проигрывает REST в простоте

SERVICE CALLING MADE EASY



REST

REST (Representational State Transfer) — архитектурный стиль; стиль общения компонентов, при котором все необходимые данные указываются в параметрах запроса.

Работает поверх HTTP. Поддерживает обмен сообщений в форматах: текст; HTML; YAML; XML; **JSON**.

Отличительная особенность этого стиля — это соответствие между данными и URL-адресами:



```
{  
  "userId": 1,  
  "id": 1,  
  "title": "delectus aut autem",  
  "completed": false  
}
```

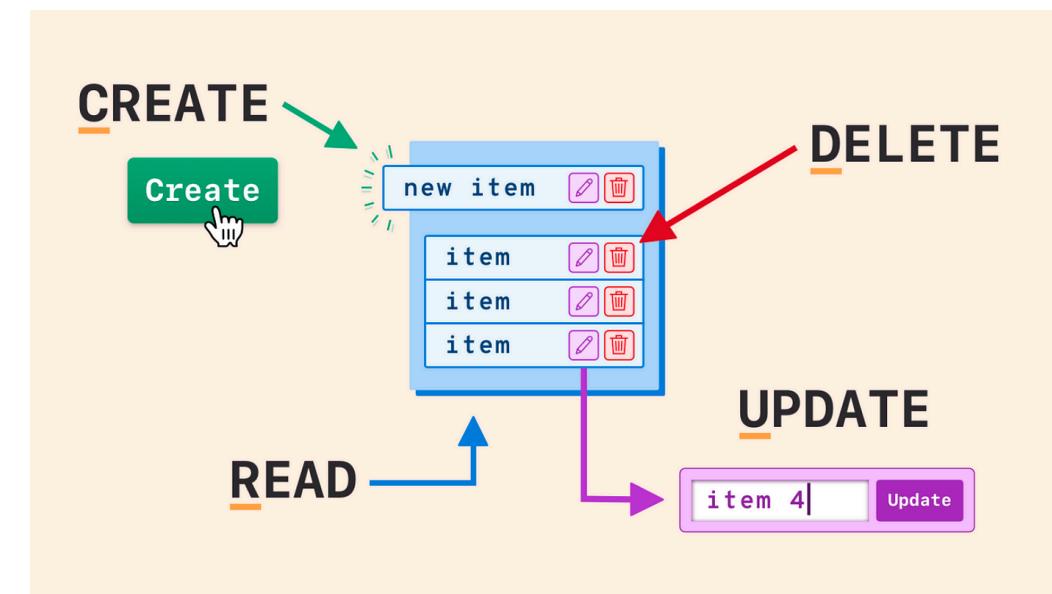
REST

Всё взаимодействие между клиентом и сервером сводится к 4 операциям (CRUD):

- созданию чего-либо, например, объекта пользователя (create, **C**)
- чтению (read, **R**)
- обновлению (update, **U**)
- удалению (delete, **D**)

Для каждой из операций есть собственный HTTP-метод:

- **POST** для создания
- **GET** для чтения
- **PUT, PATCH** для обновления
- **DELETE** для удаления



REST

Если бы мы писали API для интернет-магазина, то CRUD для заказа мог бы выглядеть следующим образом:

- **POST /api/orders/** — создать новый заказ

Как правило, в ответ на POST-запрос сервер возвращает ID созданной сущности, в нашем случае — ID заказа. Пусть будет 42

- **GET /api/orders/42** — получить заказ с номером 42

В ответ мы получим JSON, XML, HTML с данными о заказе (сейчас чаще всего — JSON)

- **PUT /api/orders/42** — обновить заказ с номером 42

Вместе с запросом мы отправляем данные, которыми надо обновить этот заказ. В ответ сервер ответит или статусом 204 (всё хорошо, но контента в ответе нет), или ID обновлённой сущности

- **DELETE /api/orders/42** — удалить заказ с номером 42

Как правило, в ответ присыпается или 204, или ID удалённой сущности

REST

Плюсы:

- самый распространённый стиль
- использует фундаментальную технологию (HTTP), как основу
- достаточно легко читается

Минусы:

- если спроектирован плохо, может отправлять или слишком много информации, либо слишком мало
(но для обхода этой проблемы можно использовать backend for frontend)



RPC

RPC (Remote Procedure Call) — стиль, при котором в сообщении запроса хранится и действие, которое надо выполнить, и данные, которые для этого действия нужны.

Наиболее популярная на данный момент реализация RPC — gRPC.

gRPC — протокол от Google, работающий поверх HTTP/2 и использующий в качестве формата обмена данных буферы протоколов (protobuf).

Плюсы:

- Невероятно производителен

Минусы:

- Не в вебе

```
syntax = "proto2";

package tutorial;

message Person {
    required string name = 1;
    required int32 id = 2;
    optional bool has_mac = 3;
    repeated Phone phones = 4;
}

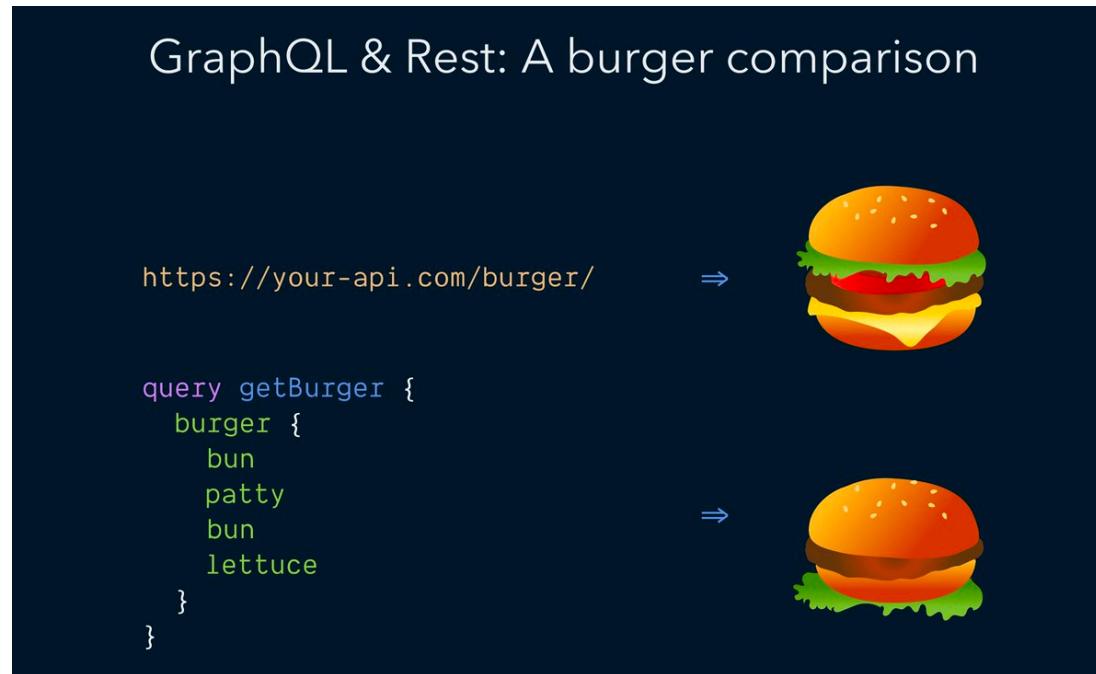
enum PhoneType {
    MOBILE = 0;
    HOME = 1;
}

message Phone {
    required string number = 1;
    optional PhoneType type_ = 2 [default = MOBILE];
}
```

GraphQL

GraphQL — это язык запросов и серверная среда для API с открытым исходным кодом.

Он появился в Facebook в 2012 году и был разработан для упрощения управления конечными точками для API на основе REST.



GraphQL

Плюсы:

- Оптимизация запросов (берём только то, что нужно)
- Более гибкий, чем CRUD
- Один запрос — много ресурсов

Минусы:

- Отсутствие браузерного кэширования
- Безопасность:
 - Уязвимость перед DDoS-атаками
 - Приватность на уровне полей, а не URL
- Недостаточно зрелая экосистема:
 - GraphQL (опенсорс) — 2015
 - GraphQL Foundation — 2019



Maciej Walkowiak @maciejwalk... · 1d

The biggest difference between REST and GraphQL is that when you build GraphQL API nobody tells you "well, actually this is not GraphQL".

27 180

1,458



Jason Carreira
@jasoncarreira

Replies to @maciejwalkowiak

It's only REST if it's developed in the REST region of France, otherwise it's just sparkling JSON.

```
user {  
    username <-- могут видеть все  
    email <-- приватное  
    post {  
        title <-- некоторые посты приватны  
    }  
}
```

HTTP

HTTP (HyperText Transfer Protocol) — протокол прикладного уровня модели OSI, предназначенный для обмена данными между веб-сервером и веб-браузером.

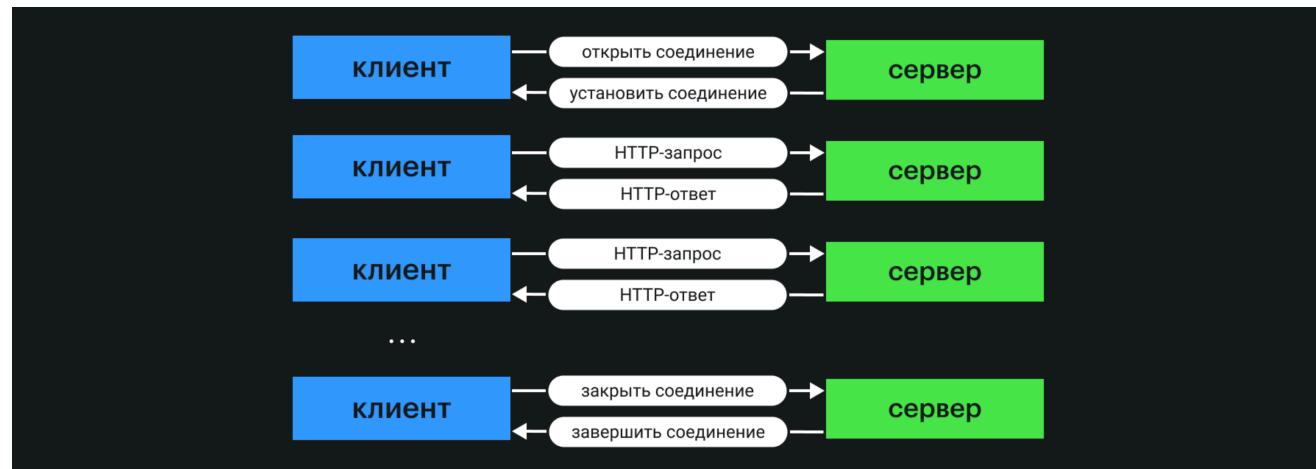
Есть три главных объекта, которые обмениваются сообщениями:

- **Клиент (user agent)** — программа, которая отправляет запросы, получает и обрабатывает ответы (браузер)
- **Сервер** — программа, которая принимает, обрабатывает запросы и отправляет ответы клиенту (веб-сервер)
- **Прокси (прокси-сервер)** — программа, которая работает на сервере, пропускает через себя запросы и ответы и выступает в роли посредника между клиентом и сервером

HTTP

HTTP-сообщение представляет собой обычный текст. Структура сообщения строго определена:

1. Стартовая строка (Starting line) говорит нам, запрос или ответ содержит сообщение
2. Заголовки (Headers) описывают тело сообщения, параметры передачи и прочие сведения
3. Тело сообщения (Message Body) содержит данные (опциональная часть)



HTTP. Стартовая строка

Стартовая строка запроса:

1. Метод запроса
2. URI
3. Используемый протокол

HTTP

1 GET /tools/web-server HTTP/2.0

Стартовая строка ответа:

1. Используемый протокол
2. Код состояния (реакция сервера на запрос)
3. Пояснение к коду ответа (необязательное)

HTTP

1 HTTP/2.0 200 OK

HTTP. Методы запроса

Методы запроса описывают тип обработки данных, который клиент хочет осуществить:

- **OPTIONS** — используется для определения возможностей сервера по преобразованию данных
- **GET** — используется для получения данных от сервера
- **HEAD** — то же, что и GET, но не содержит тело в сообщении ответа
- **POST** — используется для отправки данных на сервер
- **PUT** — используется для добавления новых или изменения существующих данных на сервере
- **PATCH** — то же, что и PUT, но используется для обновления части данных
- **DELETE** — используется для удаления данных на сервере
- **TRACE** — возвращает запрос от клиента таким образом, что в ответе содержится информация о преобразованиях запроса на промежуточных серверах.
- **CONNECT** — переводит текущее соединение в TCP/IP-туннель (обычно используется для установления защищённого SSL-соединения)

HTTP. Коды ответов

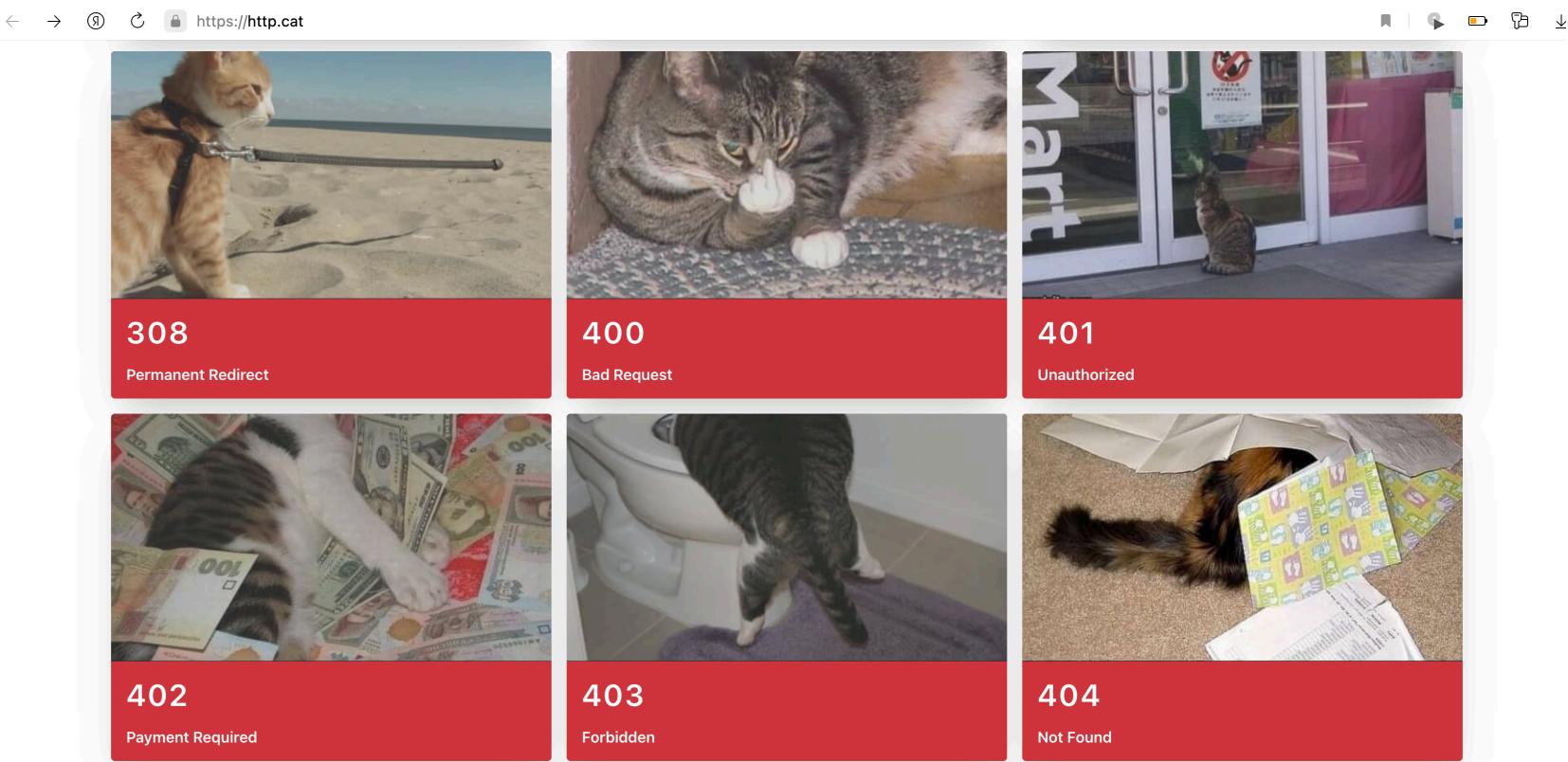
Код состояния в ответе сервера содержит информацию о результате обработки данных.

Существует пять классов кодов состояний:

- **1xx** — информационные коды
- **2xx** — успешная обработка данных
- **3xx** — перенаправление запросов
- **4xx** — ошибка по вине клиента
- **5xx** — ошибка по вине сервера

Information [100 - 199]	Redirect [300 - 399]
100 - Continue	300 - Multiple Choices
101 - Switching Protocols	301 - Moved Permanently
102 - Processing	304 - Not Modified
103 - Checkpoint	307 - Temporary Redirect
	308 - Permanent Redirect
Success [200 - 299]	Client Error [400 - 499]
200 - OK	400 - Bad Request
201 - Created	401 - Unauthorized
202 - Accepted	403 - Forbidden
204 - No Content	404 - Not Found
205 - Reset Content	408 - Request Timeout
206 - Partial Content	409 - Conflict
Server Error [500 - 599]	
500 - Internal Server Error	503 - Service Unavailable
501 - Not Implemented	504 - Gateway Timeout
502 - Bad Gateway	

HTTP. Коды ответов



<https://http.cat/>

HTTP. Заголовки

В заголовках содержится информация о домене, о сжатии, шифровании и формате данных, кодировке и прочая важная информация, которая используется для запроса или ответа.

Каждый заголовок в HTTP-сообщении представляется отдельной строкой с параметром и значением, разделёнными двоеточием.

Заголовки запроса:

HTTP

- 1 `Accept-Language: en-us`
- 2 `Accept-Encoding: gzip, deflate, br`

Заголовки ответа:

HTTP

- 1 `Content-Type: text/html; charset=UTF-8`
- 2 `Content-Encoding: gzip`

HTTP. Тело сообщения

Запрос:

```
HTTP  
1 GET /wiki/Web_server HTTP/1.1  
2 Host: en.wikipedia.org
```

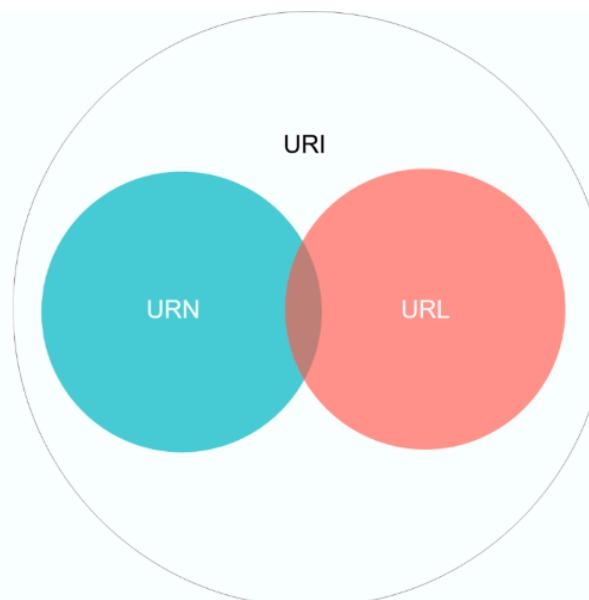
Ответ:

```
HTTP  
1 Date: Wed, 02 Jun 2021 00:41:41 GMT  
2 Content-Language: en  
3 Last-Modified: Tue, 25 May 2021 15:08:46 GMT  
4 Content-Type: text/html; charset=UTF-8  
5 Content-Encoding: gzip  
6 ...  
7 Accept-Ranges: bytes  
8 Content-Length: 27606  
9 Connection: keep-alive  
10  
11 <!DOCTYPE html>  
12 <html class="client-nojs" lang="en" dir="ltr">  
13 <head>  
14 <meta charset="UTF-8">  
15 <title>Web server - Wikipedia</title>  
16 ...
```

URI + URL + URN

Аббревиатуры:

- **URI** — Uniform Resource Identifier (унифицированный идентификатор ресурса)
- **URL** — Uniform Resource Locator (унифицированный определитель местонахождения ресурса)
- **URN** — Uniform Resource Name (унифицированное имя ресурса)



URI + URL + URN

Пример:

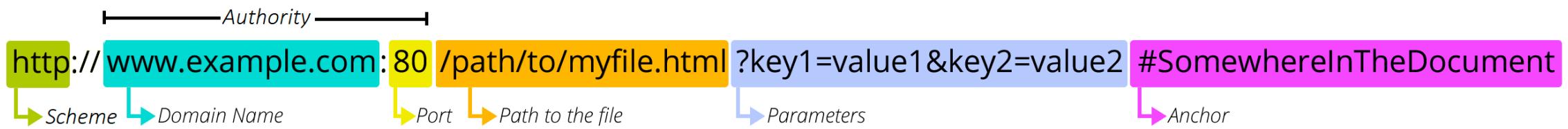
- **URI** — <https://github.com/lyaplyap/frontend-development-course>
- **URL** — <https://github.com/>
- **URN** — /lyaplyap/frontend-development-course

Но <https://github.com/lyaplyap/frontend-development-course> — также URL!

Назначение:

- **URI** — идентифицировать ресурс и отличить его от других ресурсов, используя местоположение или имя
- **URL** — получить адрес или местоположение ресурса

URL



Cookie

HTTP — это stateless-протокол (без сохранения состояния), то есть общение клиента с сервером состоит из независимых пар запрос-ответ.

Для хранения данных о текущей сессии используются **куки (cookie)**.

Куки чаще всего используют для:

- Аутентификации и авторизации
- Хранения данных о пользователях
- Защиты пользователя (CSRF токен)

Eu: *entro em um site*

Site:

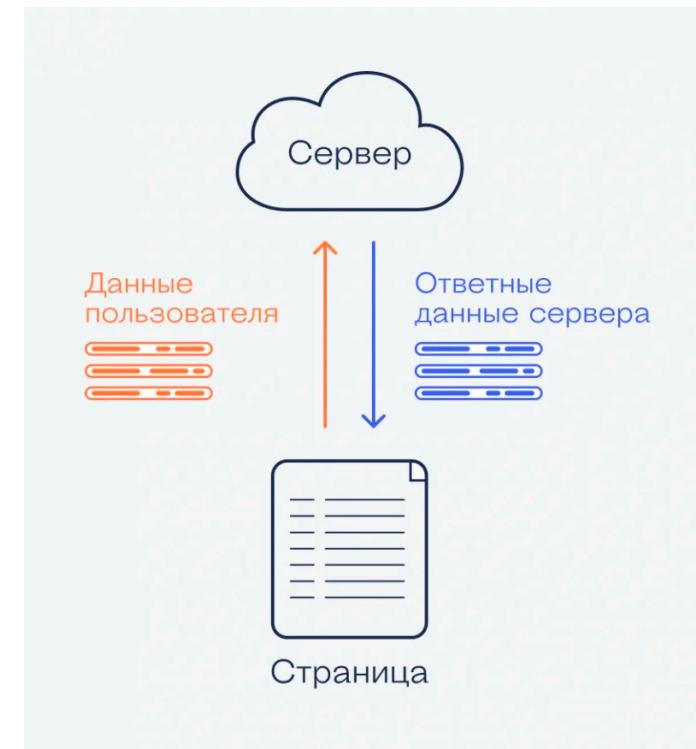


Сетевые запросы. AJAX

AJAX (Asynchronous JavaScript and XML) — подход к использованию существующих технологий, при котором недостающая информацию запрашивается у сервера и добавляется на страницу, при этом сама страница не перезагружается.

Используемые технологии:

- XMLHttpRequest (устаревший)
- Fetch API
- Long Polling
- WebSocket



Сетевые запросы. XMLHttpRequest

XMLHttpRequest (XHR) – это встроенный в браузер объект, который даёт возможность делать HTTP-запросы к серверу без перезагрузки страницы.

XHR может работать с любыми данными, а не только с XML.

XHR — устаревший способ отправки сетевых запросов. Стоит его использовать, только если:

- Нужно поддерживать код с XHR
- Нужно поддерживать старые браузеры и нельзя использовать полифили
- Нужна функциональность, которой нет в fetch (например, отслеживание прогресса отправки)

```
1 // 1. Создаём новый XMLHttpRequest-объект
2 let xhr = new XMLHttpRequest();
3
4 // 2. Настраиваем его: GET-запрос по URL
5 xhr.open('GET', 'https://jsonplaceholder.typicode.com/todos/1');
6
7 // 3. Отсылаем запрос
8 xhr.send();
9
10 // 4. Этот код сработает после того, как мы получим ответ сервера
11 xhr.onload = function() {
12     // Коллбек на успех
13 };
14
15 xhr.onprogress = function(event) {
16     // Коллбек на загрузку
17 };
18
19 xhr.onerror = function() {
20     // Коллбек на ошибку
21 };
```

Сетевые запросы. Fetch API

Функция **fetch()** принимает два параметра:

- url — адрес, по которому нужно сделать запрос
- options (необязательный) — объект конфигурации, в котором можно настроить метод запроса, тело запроса, заголовки и другое

По умолчанию вызов `fetch()` делает GET-запрос по указанному адресу. Базовый вызов для получения данных можно записать таким образом:

JS

```
1 fetch('http://jsonplaceholder.typicode.com/posts')
```

Результатом вызова **fetch()** будет Promise, в котором будет содержаться специальный объект ответа Response. У этого объекта есть два важных поля:

- ok — принимает состояние `true` или `false` и сообщает об успешности запроса
- json — метод, вызов которого, возвращает результат запроса в виде json

Сетевые запросы. Fetch API

```
1  const newPost = {
2    title: 'foo',
3    body: 'bar',
4    userId: 1,
5  }
6
7  fetch('https://jsonplaceholder.typicode.com/posts', {
8    method: 'POST', // Здесь так же могут быть GET, PUT, DELETE
9    body: JSON.stringify(newPost), // Тело запроса в JSON-формате
10   headers: {
11     // Добавляем необходимые заголовки
12     'Content-type': 'application/json; charset=UTF-8',
13   },
14 })
15 .then(response) => response.json()
16 .then(data) => {
17   console.log(data)
18   // {title: "foo", body: "bar", userId: 1, id: 101}
19 };
```

Сетевые запросы. Fetch API

```
1  fetch('https://jsonplaceholder.typicode.com/there-is-no-such-route')
2  .then((response) => {
3      // Проверяем успешность запроса и выкидываем ошибку
4      if (!response.ok) {
5          throw new Error('Error occurred!')
6      }
7
8      return response.json()
9  })
10 // Теперь попадём сюда, т.к выбросили ошибку
11 .catch((err) => {
12     console.log(err)
13 }); // Error: Error occurred!
```

Полезные материалы

<https://doka.guide/> — про веб-разработку понятным языком от профессиональных разработчиков

<https://learn.javascript.ru/> — самый подробный учебник по JS с примерами и задачами

<https://developer.mozilla.org/ru/> — документация по JS и WebAPI от Mozilla

<https://academy.yandex.ru/journal/chto-takoe-graphql> — про GraphQL от Академии Яндекса