BACKEND. ПРОТОКОЛ HTTP. API. АРХИТЕКТУРА REST

Оглавление

1 Backend и Frontend

У большинства современных сайтов и сервисов есть два слоя: фронтенд для пользователей и бэкенд для технических действий. С фронтендом взаимодействуют обычные клиенты, а бэкенд находится под капотом и обрабатывает запросы от фронтенда.

Фронтенд (англ. frontend) — это разработка пользовательских функций и интерфейса. К ним относится всё, что пользователи видят на сайте или в приложении, и с чем можно взаимодействовать: картинки, выпадающие списки, меню, анимация, карточки товаров, кнопки, чекбоксы, интерактивные элементы. На любой странице в интернете виден результат работы фронтенд-разработчика.

Чтобы увидеть фронтенд-код сайта, достаточно щёлкнуть по странице правой кнопкой мыши и выбрать «*Просмотреть код*». Фронтенд-разработчики трудятся вместе с дизайнерами и верстальщиками над созданием продукта, с которым пользователю будет удобно взаимодействовать.

Бэкенд (англ. backend) — это логика работы сайта, скрытая от пользователя. Именно там происходит то, что можно назвать работой сайта.

Чтобы понять, в чём разница между **backend** и **frontend**, возьмём пример:

- Визуальное отображение карточки товара и кнопка «заказать» — это **фронтенд**.
- Обновление цены и остатков товара на складе, добавление товара в корзину при нажатии кнопки, функция сравнения двух товаров — это **бэкенд**.
- Результат, который видит пользователь: цена и остатки товара, товар в корзине, сравнение — это снова **фронтенд**.

Код бэкенда увидеть невозможно — он не отправляется пользователю напрямую в смартфон или браузер, а работает на сервере, на котором хранится приложение или сайт. **Сервер** — это специальный мощный компьютер, который подключён к интернету и служит для хранения данных, работы кода и его отправки в браузер.

Какие языки используют фронтенд- и бэкенд-разработчики:

HTML и **CSS** — это не языки программирования, а языки разметки. Они «рассказывают» браузеру, как именно должна выглядеть страница: где расположены блоки, какого они цвета, какого размера шрифт и картинки. От языков программирования HTML и CSS отличаются тем, что в них нет никаких функций, подсчётов, сравнений и других действий — они статично описывают внешний вид страницы.

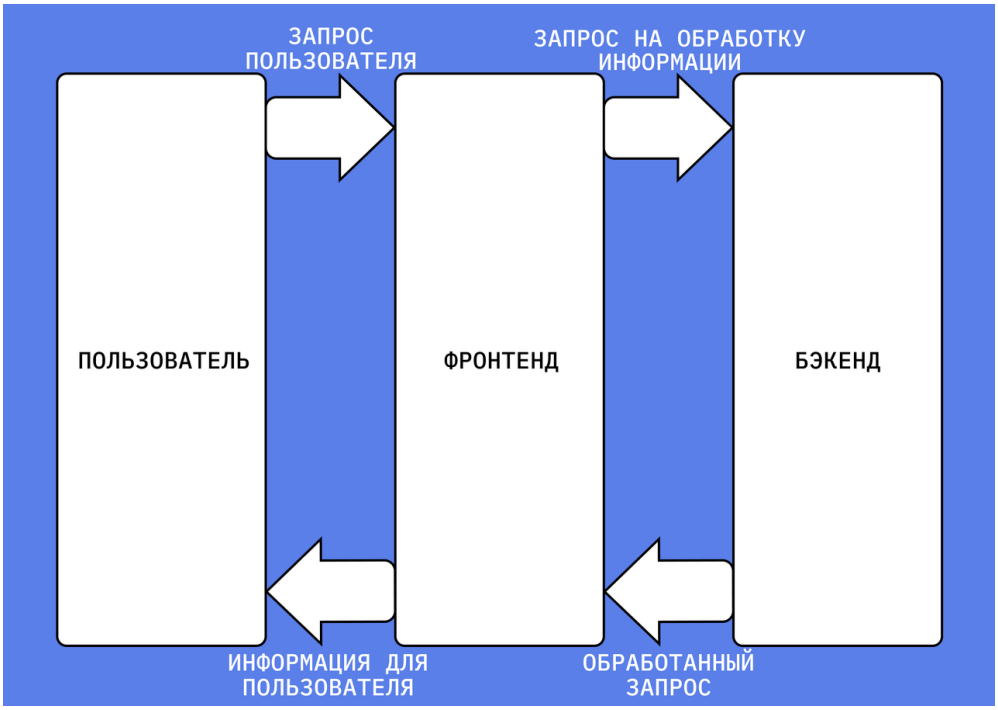
JavaScript — язык программирования. Он содержит функции и методы, которые позволяют получать информацию от сервера, отправлять её назад и выводить для пользователя, создавать интерактивные элементы, например кнопки и формы.

В отличие от фронтенда, бэкенд использует гораздо больше языков программирования. Он может быть написан на любом языке. Сейчас наиболее востребованы **Python**, **Java**, **Go**, всё ещё популярен **PHP**. Иногда бэкенд пишут на **C#** и **Ruby**.

Взаимодействие фронтенда и бэкенда

Разработка фронтенд и бэкенд неотделимы друг от друга. Обычно они связаны по такой схеме:

1. Пользователь что-то делает во фронтенде. Например, нажимает кнопку.
2. Фронтенд отправляет информацию об этом действии в бэкенд.
3. Бэкенд обрабатывает информацию. Например, если пользователь нажал кнопку «Заказать», формирует для него корзину и подсчитывает цену с доставкой.
4. Бэкенд возвращает информацию назад фронтенду.
5. Фронтенд «рисует» для пользователя понятную «картинку» — страницу корзины со стоимостью товара и доставки.



Различия фронтенда и бэкенда

Главная разница между **frontend** и **backend** в том, что первый работает на пользователя, на мощности его компьютера или смартфона, а второй — на сервере, и пользователю отправляет только результат работы. Но есть и другие отличия:

Фронтенд	Бэкенд
Взаимодействует непосредственно с **конечным пользователем**	Обеспечивает **логические функции**, нужные для работы сайта и приложения.
Единообразен. Практически все фронтенд-разработчики работают на связке технологий HTML+CSS+JavaScript.	**Многообразен**. Можно писать практически на любом языке программирования. Открывает доступ к сложным технологиям вроде машинного обучения и анализа данных.
Изменчив. Языки, функции и инструменты для работы часто меняются. Нужно постоянно учиться и следить за трендами.	**Стабилен**. Радикальные изменения происходят редко, можно годами работать по привычной схеме.
Фронтенд-разработчики более тесно контактируют с дизайнерами, маркетологами, менеджерами продукта.	Бэкенд-разработчики больше взаимодействуют с аналитиками, продакт-менеджерами и фронтендерами.

2 Протокол HTTP: ресурсы URL, методы, заголовки, коды ответов

Интернет состоит из компьютеров. Специальные компьютеры — **серверы** — существуют для хранения информации и обработки запросов от клиентов. Браузеры на компьютерах пользователей называются **клиенты**.
Как мы выяснили из прошлой главы - по команде пользователя клиент шлёт на сервер запрос. Чтобы серверы и клиенты понимали друг друга, производители сетевого оборудования и разработчики программ договорились, как именно будет передаваться информация. Такие договорённости называются **протоколы**.

Из сетевых протоколов для бэкенд-разработчика важнее всех **HTTP** (от англ. *HyperText Transfer Protocol*, «*протокол передачи гипертекста*»). Именно он отвечает за передачу написанных гипертекстом веб-страниц с сервера на клиент и обратно.

Разберемся, из чего состоит HTTP протокол.

2.1 URI-адреса

URI (англ. *Uniform Resource Identifier* - «*унифицированный идентификатор ресурса*») — это полный адрес сайта в Сети. Он состоит из двух частей: **URL** (*locator*) и **URN** (*name*).

Первое — это адрес хоста. Например, *www.vk.com*.

Второе — это то, что ставится после **URL** и символа «**/**». Например, для URI *www.vk.com/feed* URN-адресом будет */feed*. **URN** ещё можно назвать адресом до конкретного файла на сайте (конечная точка).

По сути URL - лишь адрес, который выдан уникальному ресурсу в интернете. В теории, **каждый корректный URL ведёт на уникальный ресурс**. Такими ресурсами могут быть HTML-страница, CSS-файл, изображение и т.д.

- **http://** - протокол. Он отображает, какой протокол браузер должен использовать. Обычно это HTTP-протокол или его безопасная версия - HTTPS.
- **www.domain.com** - доменное имя. Оно означает, какой веб-сервер должен быть запрошен. В качестве альтернативы может быть использован и IP-адрес, но это делается редко, поскольку запоминать IP сложнее, и это не популярно в интернете.
- **:1234** - порт. Он отображает технический параметр, используемый для доступа к ресурсам на веб-сервере. Обычно подразумевается, что веб-сервер использует стандартные порты HTTP-протокола (80 для HTTP и 443 для HTTPS) для доступа к своим ресурсам. В любом случае, порт - это факультативная составная часть URL.
- **/path/to/resource** - адрес ресурса на веб-сервере. Чаще всего это абстракция, позволяющая обрабатывать адреса и отображать тот или иной контент из баз данных.
- **?a=b&x=y** - дополнительные параметры запроса, которые браузер сообщает веб-серверу. Эти параметры - список пар *ключ=значение*, которые разделены символом «*&*». Веб-сервер может использовать эти параметры для исполнения дополнительных команд перед тем как отдать ресурс (например, пагинация, поиск и тд). Каждый веб-сервер имеет свои собственные правила обработки этих параметров и узнать их можно, только спросив владельца сервера.

2.2 Методы

URL-ссылки идентифицируют определенный сервер, с которым мы хотим наладить обмен сообщениями, однако действие, которое должно быть выполнено на сервере, указывается при помощи **методов HTTP**. Естественно, что клиент хотел бы выполнить некоторые действия (**методы**) на сервере. В HTTP методы стандартизированы и универсальны для всех видов приложений.

Чаще всего применяются методы, реализующие **CRUD**:

- **GET** получает ресурсы;
- **POST** создаёт ресурс;
- **PUT** заменяет существующий ресурс целиком;
- **PATCH** частично изменяет существующий ресурс;
- **DELETE** удаляет ресурс.

Реже применяют ещё два метода:

- **HEAD** получает только заголовки ответа. **HEAD** похож на **GET**, но в ответе на этот запрос есть только заголовок, а тела ответа нет.
- **OPTIONS** получает перечень HTTP-методов, которые поддерживает сервер.

При **POST, PUT, PATCH** методах в запросе также может передаваться **тело** (*Request/Response body*) – информация о новом/обновленном ресурсе (например, в формате JSON, XML), файлы и тд.

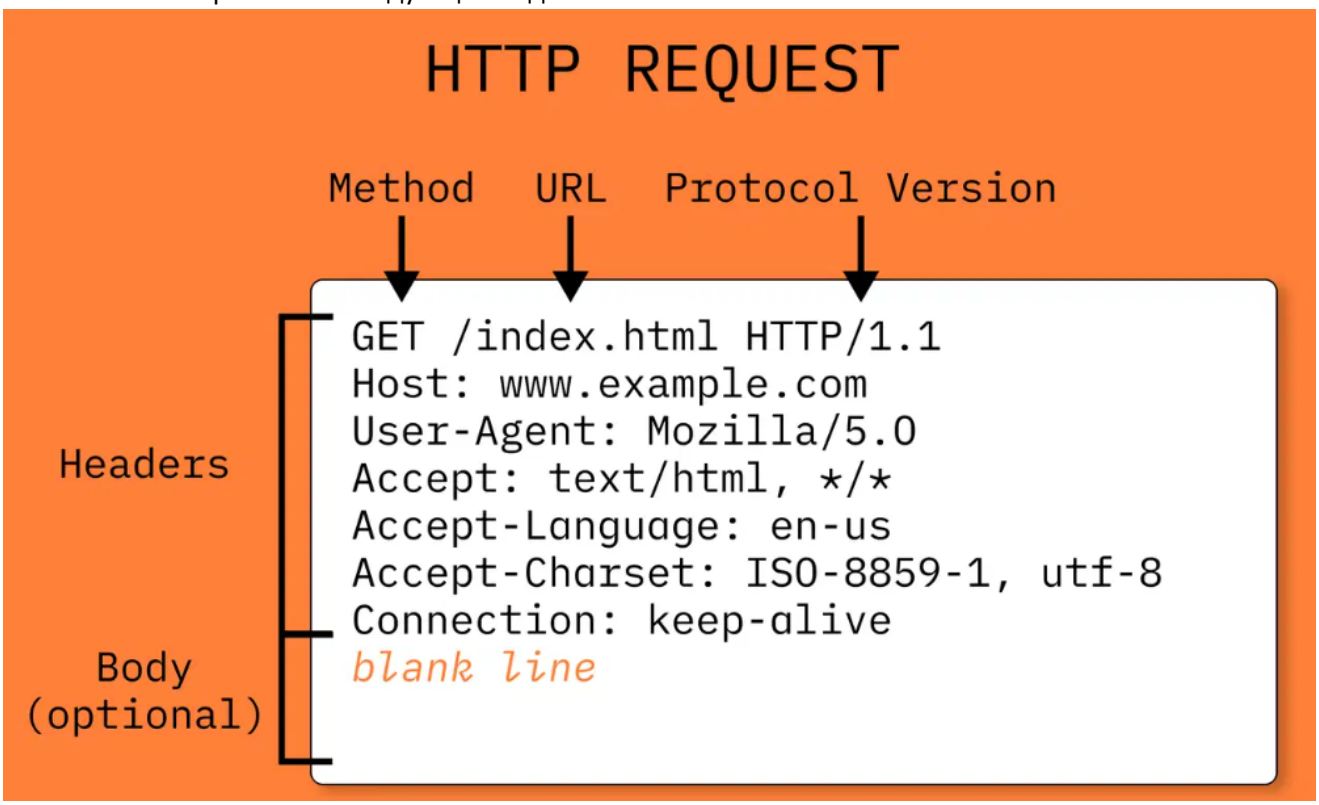
2.3 Заголовки (*Headers*)

Заголовки HTTP позволяют клиенту и серверу отправлять дополнительную информацию с HTTP запросом или ответом. В HTTP-заголовке содержится не чувствительное к регистру название, а затем после (:) непосредственно значение. Пробелы перед значением игнорируются.

Вот пример некоторых HTTP заголовков с кратким описанием:

- **Authorization** - используется для отправки данных авторизации на сервер;
- **Host** - используется для указания доменного имени и порта запрашиваемого ресурса;
- **User-Agent** - содержит в себе полную информацию о клиенте пользователя, например, о браузере;
- **Content-Length** - указывает необходимую длину тела сообщения в байтах;
- **Content-Type** - используется для указания медиа типа данных в теле сообщения;
- **Cookie** - содержит сохраненные файлы, связанные с сервером (дополнительная информация);

В итоге HTTP-запрос имеет следующий вид:



2.4 Коды состояния (*Status Code*)

Имея URL-адреса и методы, клиент может инициировать запросы к серверу. В ответ сервер присылает ответы с кодами состояния и содержимым сообщений. **Код состояния** – важный компонент сообщения; он указывает клиенту, как интерпретировать ответ сервера.

HTTP устанавливаются определенные диапазоны чисел для конкретных типов ответов:

- **100 – 199**: *Информационные* - используется просто для предварительного общения клиента и сервера;
- **200 – 299**: *Успешные* - сообщают клиенту, что его запрос успешно обработан:
    - **200 OK** - "*Успешно*". Запрос успешно обработан.
    - **201 Created** - "*Создано*". Запрос успешно выполнен и в результате был создан ресурс.

    - **204 No Content** - "*Нет содержимого*". Нет содержимого для ответа на запрос, но заголовки ответа, которые могут быть полезны, присылаются.
- **300 – 399**: *Перенаправления* - указывает клиенту, что необходимо будет выполнить дополнительное действие. Самый распространенный вариант – выполнение запроса по другому URL-адресу получения запрашиваемого ресурса;
- **400 – 499**: *Клиентские ошибки* – информация об ошибках со стороны клиента:
    - **400 Bad Request** - "*Плохой запрос*". Этот ответ означает, что сервер не понимает запрос из-за неверного синтаксиса.
    - **401 Unauthorized** - "*Неавторизованно*". Для получения запрашиваемого ответа нужна аутентификация. Статус похож на статус 403, но в этом случае аутентификация возможна.
    - **403 Forbidden** - "*Запрещено*". У клиента нет прав доступа к содержимому, поэтому сервер отказывается дать надлежащий ответ.
    - **404 Not Found** - "*Не найден*". Сервер не может найти запрашиваемый ресурс.
    - **413 Request Entity Too Large** - Размер запроса превышает лимит, объявленный сервером.
- **500 – 599**: *Серверные ошибки* - используется для сообщения о неуспешном выполнении операции по вине сервера:
    - **500 Internal Server Error** - "*Внутренняя ошибка сервера*". Сервер столкнулся с ситуацией, которую он не знает как обработать.
    - **501 Not Implemented** - "*Не выполнено*". Метод запроса не поддерживается сервером и не может быть обработан.

2.5 Преимущества и недостатки HTTP

Преимущества	Недостатки
Расширяемость. В 1992, когда HTTP только появился, он был совсем простым. Но со временем протокол обрастал новыми методами и возможностями, и он всё ещё способен к расширению и изменению	**Отсутствие навигации**. HTTP не позволяет запросить все доступные ресурсы и их параметры.
Подробная документация. HTTP подробно описан на разных языках, и в документации есть ответы на большинство вопросов.	**Проблемы с распределёнными запросами**. Когда HTTP только создавали, время обработки запросов не учитывали, но сейчас с повышением нагрузки на серверы это иногда становится проблемой.
Распространённость. HTTP — самый популярный протокол в интернете. Он считается основным и универсальным, на нём работают практически все сайты в мире.	**Незащищённость**. Базовый HTTP без шифрования совершенно небезопасен — любой может перехватить данные запроса и узнать всё: логины, пароли, данные банковских карт. Поэтому и появился **HTTPS**.

3 API. Форматы передачи данных. Основные принципы REST

3.1 API. Форматы передачи данных

Интерфейс (от англ. *interface*) — это посредник, средство взаимодействия двух систем. Например, регулятор громкости радиоприёмника — это интерфейс регулирования громкости, посредник между системами «человек» и «электронная начинка приёмника».

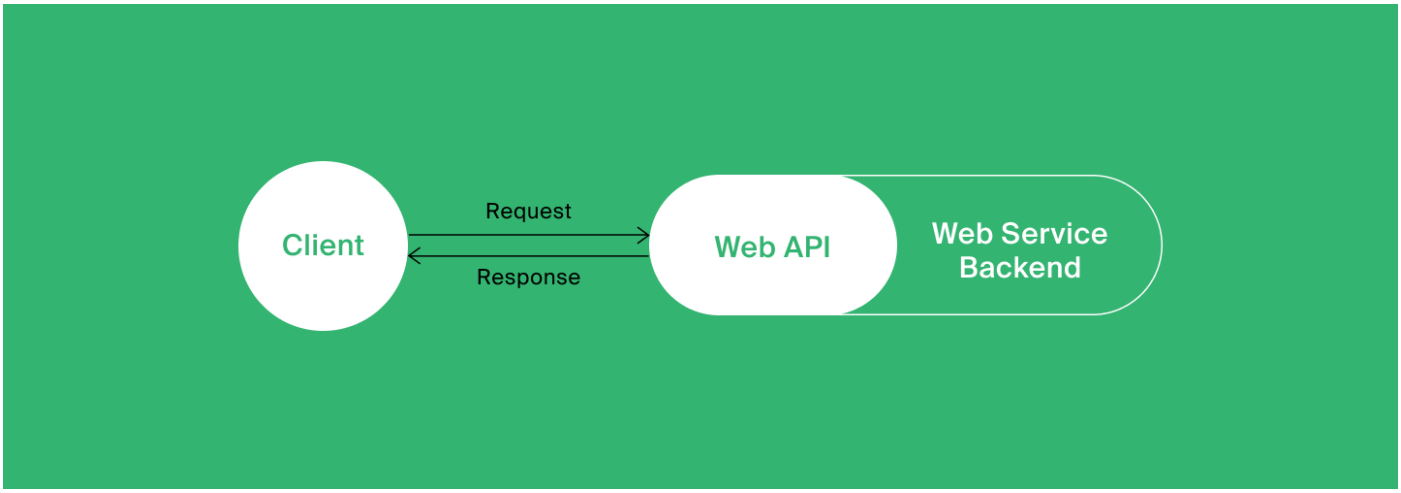
Графическая оболочка операционной системы компьютера — это тоже интерфейс. На экране нарисованы кнопки, пользователь нажимает на них, решая свои повседневные задачи, и при этом не задумывается о внутреннем устройстве самой операционной системы.

API (от англ. *Application Programming Interface*, «*программный интерфейс приложения*») — это интерфейс для обмена данными. Слово «программный» означает, что API служат в первую очередь для взаимодействия программ: с системой взаимодействует не разработчик, а код, написанный им.

API могут общаться друг с другом: например, ваш код запрашивает данные с API Wildberries о появлении новых товаров, а Wildberries в свою очередь обращается к API различных магазинов, чтобы эти данные получить.

Программу, которая обращается к API с запросом, называют «клиент» (или client, если по-английски). Клиентом может быть код на сервере, мобильное приложение, программа для тестирования или даже обычный веб-браузер.

При этом каждый сервис может быть написан на своём языке программирования, но благодаря API они могут легко общаться между собой по сети, используя протокол HTTP и передавая данные в удобном для всех формате.



JSON

Один из самых распространённых форматов передачи данных — это **JSON**.

JSON расшифровывается как *JavaScript Object Notation* (англ. «*объектная запись JavaScript*»). Исторически сложилось так, что этот формат приешл из языка программирования JavaScript. По структуре **JSON** очень похож на тип данных **dict** в Python: это последовательность пар «**ключ-значение**»; как и словари, JSON поддерживает вложенность. Но JSON более стандартизирован: например, ключи словаря в JSON пишутся только в **двойных кавычках**. Значениями ключей могут быть:

- строки;
- числа;
- булевы значения;
- словари;
- списки;
- null.

Пример описания супергероя в формате JSON:

```
[
  {
      "name": "Captain America",
      "realName": "Steve Rogers",
      "yearCreated": 1941,
      "powers": [
        "Strength",
        "Healing ability"
    ]
  }
]
```

Так будет выглядеть JSON, если супергероев несколько:

```
[
  {
    "name": "Captain America",
    "realName": "Steve Rogers",
    "yearCreated": 1941,
    "powers": [
      "Strength",
      "Healing ability"
    ]
  },
  {
    "name": "Spider-Man",
    "realName": "Peter Parker",
    "yearCreated": 1963,
    "powers": [
      "Danger sense",
      "Speed",
      "Jumping"
    ]
  }
]
```

Некоторые типы данных в JSON не формализованы: например, в нём нет специального формата для даты и данные можно передавать в любом удобном формате — например, как строку "*09-10-1988*" или как целое число *1623799668*.

XML

XML (*eXtensible Markup Language*, «*расширяемый язык разметки*») тоже популярен и широко применяется в разработке. Внешне этот язык чем-то похож на HTML, и это неслучайно: язык разметки веб-страниц — прямой потомок XML.

Одно из основных отличий HTML от XML в том, что названия тегов в XML не стандартизированы, их можно называть по собственному желанию. Да и предназначение у этих языков разное: теги HTML служат для отображения информации в браузере, а XML-теги просто структурируют передаваемую информацию.

Синтаксис похож на HTML: теги пишутся в треугольных скобках, есть открывающий тег и (в большинстве случаев) к нему в пару должен быть закрывающий.

Вот та же картотека супергероев, но в формате **XML**:

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
    <superhero>
        <name>Captain America</name>
        <realName>Steve Rogers</realName>
        <yearCreated>1941</yearCreated>
        <powers>Strength</powers>
        <powers>Healing ability</powers>
    </superhero>
    <superhero>
        <name>Spider-Man</name>
        <realName>Peter Parker</realName>
        <yearCreated>1963</yearCreated>
        <powers>Danger sense</powers>
```

```
        <powers>Speed</powers>
        <powers>Jumping</powers>
    </superhero>
</root>
```

И что же выбрать? **JSON** — один из наиболее популярных форматов обмена данными при работе с API. Он проще для чтения и при равном объёме данных весит чуть меньше, чем **XML**. JSON поддерживают все популярные языки программирования; в этом курсе вы будете работать в основном с ним.

3.2 Взаимодействие программ по сети

Чтобы предоставить программе или сервису доступ к информации или возможностям другой программы, используют API, программные интерфейсы приложений.

Зачастую программы и сервисы находятся в разных адресных пространствах: они удалены друг от друга, размещены на разных компьютерах или серверах. Взаимодействие таких программ основано на удаленном вызове процедур или функций: одна система отправляет запрос к другой, и, в результате, в удаленной системе вызывается функция.

Существует целый класс технологий, позволяющих программам удалённо вызывать функции или процедуры. Эти технологии называют **RPC** (англ. *Remote Procedure Call*, «*удаленный вызов процедур*»).

Рассмотрим самые известные технологии этого класса, которые чаще всего применяются для создания API.

Протокол SOAP: данные в конверте

SOAP (англ. *Simple Object Access Protocol*, «*протокол доступа к простым объектам*») — это протокол, который применяется для удаленного вызова процедур, обмена произвольными сообщениями в формате **XML** и для организации API-сервисов. Протокол SOAP был разработан для Microsoft, первый релиз появился в 1998 году.

Сообщения, которые курсируют между клиентом и сервером — это XML-файлы, составленные по определённому стандарту:

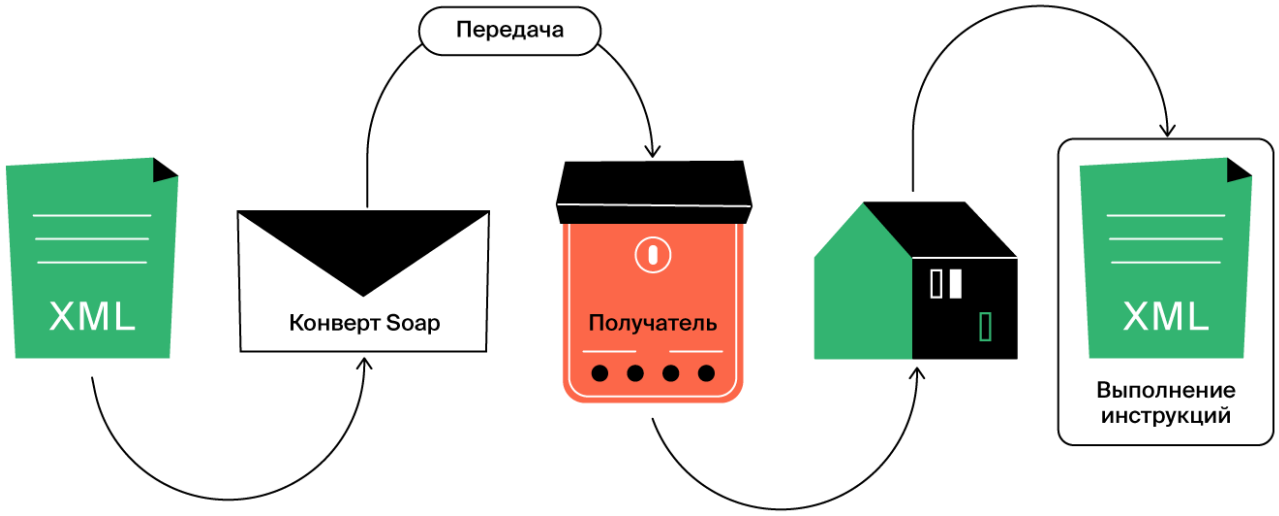
- **Конверт** — это начальный и конечный теги сообщения.
- **Заголовок** — содержит необязательные атрибуты сообщения.
- **Тело** — содержит данные, которые сервер передает получателю.
- **Ошибки** — содержит информацию об ошибках, возникших при обработке сообщения.

Вот как может выглядеть XML, в котором по протоколу SOAP пересылается информация о двух студентах:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <!--Здесь служебная информация, например - данные для авторизации-->
  </soap:Header>

  <soap:Body>
    <!--Здесь тело сообщения, пересылаемые данные-->
    <student>
      <username>Антон Кобелев</username>
      <faculty>Python</faculty>
    </student>

    <student>
      <username>Сергей Петрович</username>
      <faculty>Java</faculty>
    </student>
```

```
  </soap:Body>

  <soap:Fault>
    <!--Здесь может быть список ошибок. Но это необязательно-->
  </soap:Fault>

</soap:Envelope>
```

Само сообщение в SOAP можно представить в виде обычного письма. Такая аналогия поможет лучше понять концепцию SOAP как «конверт»:



Практическая реализация API-сервиса на основе SOAP неразрывно связана с использованием языка **WSDL**.

Язык **WSDL** (англ. *Web Services Description Language*, «*язык описания веб-сервисов*») основан на **XML** и предназначен для создания машиночитаемых «инструкций», в которых описано, как программа-клиент должна взаимодействовать с сервером.

Для этого в файле с расширением ***.wsdl** описываются детали взаимодействия клиента и сервера, в том числе — адрес, куда нужно отправлять запросы и структура конверта с запросом и ответом.

WSDL-файл — основа любого сервиса передачи данных, построенного на SOAP: программа-клиент должна знать адрес этого файла, и тогда она будет знать о веб-сервисе всё, что необходимо для работы с ним.

В настоящее время SOAP чаще всего применяется в сфере финансовых услуг, платежных шлюзов, управления идентификацией — там, где требуется высокая надёжность передачи данных и хорошо документированный стандарт, на который можно было бы опираться в юридических документах.

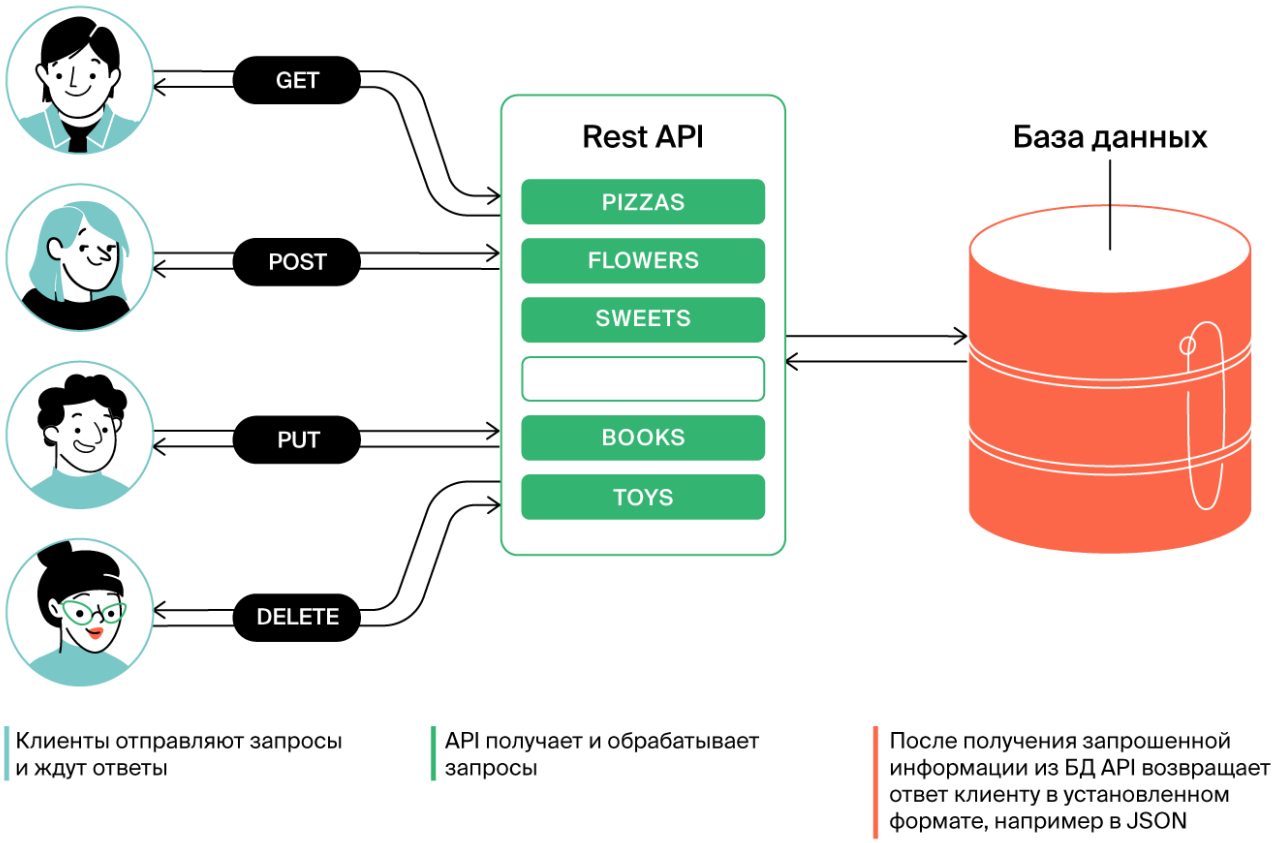
Архитектура REST

REST (англ. *REpresentational State Transfer*, «*передача состояния представления*») — это архитектурный стиль, набор принципов взаимодействия компьютерных систем, основанный на методах протокола HTTP. В отличие от SOAP, REST не подкреплен официальным стандартом.

Одно из основных понятий в REST — **ресурс**. Это довольно абстрактное понятие: ресурсом в REST называют «всё, чему можно дать имя в структуре сервиса». Ресурсом может быть что угодно, к чему разработчик REST API считает важным предоставить доступ клиенту.

Вся логика работы REST API базируется именно на ресурсах: к API отправляются запросы, из них клиент получает информацию.

На схеме — воображаемый сервис, который хранит информацию о пицце, цветах, сладостях, книгах и игрушках. Чтобы получить информацию о пицце — нужно сделать запрос к ресурсу **pizzas**, о конфетах — к ресурсу **sweets**.



Архитектура REST допускает обмен данными в различных форматах, например — в HTML, JSON, XML или даже в формате простого текста, в то время как строгий SOAP допускает только XML.

Язык запросов GraphQL: служба одного окна

GraphQL — это не протокол и не архитектура, это язык запросов к API.

GraphQL был разработан в компании Facebook в 2012 году, а в 2015 году был выпущен в открытый доступ.

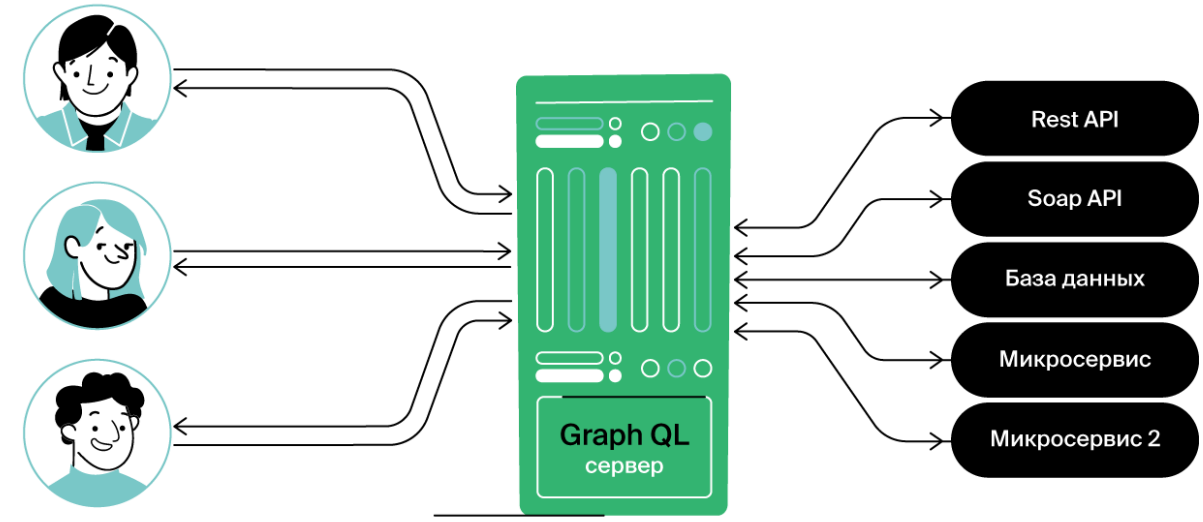
У **GraphQL** есть три основные особенности:

- он позволяет клиенту точно указать, какие данные ему нужны;
- облегчает агрегацию данных из нескольких источников (а REST вернёт данные только одного ресурса);
- использует систему типов для описания данных.

Вернёмся к воображаемому API с пиццей и другими полезными вещами.

Если API основан на архитектуре REST — то для получения информации о пицце, конфетах и книгах будет вынужден сделать три запроса к трём ресурсам.

В API, основанном на **GraphQL**, можно обойтись одним запросом: «дайте мне информацию о большой пицце, об конфетах и ещё пришлите две пару книг»



Клиенты отправляют запросы

Graph QL разбирает запрос клиента и обращается в нужные сервисы или в базу данных и возвращает консолидированный ответ клиенту

Лучший подход для API

У каждого варианта реализации API-сервиса есть свои особенности. Выбор технологии зависит от требований к проекту. Может быть важен язык программирования или формат, в котором передаются данные; могут быть заявлены требования к быстродействию или объёму передаваемых данных; возможно, есть необходимость опираться на строгий стандарт; даже квалификация разработчиков может быть аргументом в пользу выбора той или иной технологии.

Программисты и архитекторы API взвешивают все плюсы и минусы и выбирают вариант, наиболее подходящий конкретному проекту. Лучшая технология — та, которая лучше всего ляжет на требования к вашему проекту.

Далее в курсе мы будем работать именно с REST API — это популярная и востребованная на рынке архитектура; изучение REST даст навыки, которые впоследствии помогут разобраться и с другими технологиями.

3.3 API First. Архитектура REST

Во времена, когда компьютеры были большие, а мониторы — маленькие, в интернет выходили с настольного компьютера, а единственным пользовательским клиентом был веб-браузер. Сайты работали по стандартной системе: сервер получал запрос и отдавал клиенту готовые HTML-страницы.

В современном мире у такого подхода есть два недостатка:

- HTML-код нужен лишь браузеру. Например, мобильное приложение вполне обойдётся без HTML-форматирования, ему нужны лишь структурированные данные; мобильное приложение само позаботится об отображении полученных данных.
- Поскольку HTML-код генерируется на сервере, при переходе от страницы к странице браузер перезагружает сайт целиком. Это неэффективно: приходится перерисовывать одинаковые элементы: шапку, меню, подвал. Разумнее получить данные об изменяющихся частях сайта и отрисовать только их.

Размышляя об этих проблемах, Тим Бернерс-Ли (его называют создателем интернета) и Рой Филдинг (его коллега) придумали принципы, которые позволяли бы масштабировать развитие всемирной сети.

Основная идея в том, что сервер возвращает только запрошенные данные, а клиент сам разбирается, как эти данные отобразить. Мобильное приложение будет использовать свои методы отрисовки, браузер или чат-бот — свои. В результате можно ограничиться одним API для разных платформ.

Такой подход получил название **API First**, то есть **сначала данные, а затем — интерфейсы для их отображения**.

Так появился **REST**.

REpresentational State Transfer, REST (англ. «*передача состояния представления*») — это набор принципов, которых следует придерживаться при создании API. Если API сделан по этим принципам, его называют **RESTful API** (или просто **REST API**). Эти принципы стандартизируют передачу данных по сети.

Основные принципы REST:

1. **Клиент-сервер. Разделение ответственности между клиентом и сервером**
   Клиент и сервер отвечают за разные вещи. Ответственность клиента — пользовательский интерфейс, ответственность сервера — данные. Если API возвращает HTML-страницу, его нельзя назвать REST API: ведь при этом сервер берёт на себя ответственность за интерфейс.
2. **Отсутствие состояния. Сервер не хранит состояние**
   Каждый запрос должен быть независимым, как будто он сделан в первый раз. Сервер не должен хранить какой-либо информации о клиенте. Каждый запрос клиента к серверу должен содержать всю информацию, необходимую для обработки этого запроса: кто запрашивает данные, какие данные запрашиваются.
3. **Единый интерфейс**
   Интерфейс обращения к серверу одинаков для всех и не зависит от клиента. Запрос к данным может быть сформирован из браузера, мобильного приложения и с «умного» чайника по одним и тем же правилам.
4. **Многоуровневость**
   Первый принцип гласит, что в коммуникации участвуют двое: клиент и сервер. Но можно строить более сложные системы, не нарушая этого принципа.
   API сервиса Wildberries может использовать API других магазинов для получения информации. Вы как клиент взаимодействуете только с API Wildberries, а он, в свою очередь, является клиентом магазинов. Здесь есть одно условие — каждый компонент должен видеть только свой уровень. Например, сторонние магазины не должны видеть все данные, которые вы отправили в Wildberries.
5. **Кешируемость**
   Данные ответа могут быть закешированы. Это значит, что можно сохранить полученные данные на клиенте, а при идентичном запросе взять их из памяти клиента — кеша, а не ждать их с сервера. Нет смысла запрашивать данные повторно, если они никак не изменились.

3.4 Правила именования ресурсов

В базе данных проекта может храниться множество разнородной информации, и задача API — обеспечить доступ к этой информации.

Ключевая абстракция в **REST** это **ресурс**. Любая информация, которая может быть названа, может быть ресурсом: пост в социальной сети, коллекция постов, подборка актуальных новостей, пользователь сайта, коллекция любых объектов или других ресурсов.

Унифицированный указатель ресурса, или **URN** (от англ. *Uniform Resource Name*), используют для указания, где находится тот или иной ресурс.

```
/users                # Ресурсом может быть коллекция сущностей
/users/12             # Пользователь с id = 12 — это тоже ресурс
/latest-news          # Ресурс не обязательно должен быть статическим:
                      # новости каждый день разные, но ресурс — один, постоянный
/users/12/playlists # Все плейлисты пользователя с id=12 — это ресурс,
                      # содержащий коллекцию ресурсов
```

В терминах REST URN-адрес, идентифицирующий ресурс, принято называть **эндпоинтом** (англ. endpoint, «конечная точка»).

Вот несколько простых правил именования *ресурсов/эндпоинтов*:

- Почти всегда ресурсы именуют существительными во множественном числе:

`/users`

`/api/starships`

- Иногда для именования ресурсов применяют существительные в единственном числе:

`/users/{user-id}/profile`

`/users/me`

- Слеш в URL используется для указания иерархии ресурсов по принципу «*от общего к частному*»:

`/users/{user-id}/posts #` *Все пользователи → Конкретный пользователь по ID → Все его посты*.

А чтобы обратиться к конкретному посту, нужно указать его ID:

```
/users/{user-id}/posts/{post-id} # Все пользователи → Конкретный пользователь по ID →
                                 # Все его посты → Конкретный пост по ID
```

- В URL не должно быть пробелов, их заменяют дефисами или подчёркиваниями. Лучше применять дефисы:
  - на некоторых устройствах подчёркивание может выйти за базовую линию строки, и его будет не видно;
  - несколько подчёркиваний сливаются в одно.

```
# Лучше делать так:
/users/{user-id}/user-devices
# чем так:
/users/{user-id}/user_devices
```

- Не стоит включать в название ресурса имя HTTP-метода. API должен понимать по HTTP-методу, какие действия от него требуются, поэтому дублировать указания в имени ресурса не нужно.:

```
# Так не надо
GET /get-users
POST /create-user
```