GeekBrains

Клиент-серверная архитектура

Тестирование API

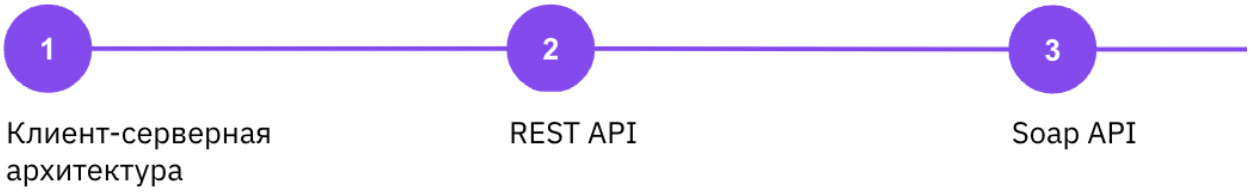
Урок 1

Оглавление

Введение

Представьте, что вы взяли в руки, например, мобильное приложение любимой пиццерии и приготовились выбирать пиццу в меню. Прежде чем, вы приступите к поиску, ваше мобильное приложение запросит актуальное меню с сервера пиццерии. Как только вы закажете пиццу, со склада спишется необходимое количество ингредиентов, пиццу начнут готовить, затем курьер получит уведомление, что необходимо забрать и доставить заказ. Чтобы отдельные составляющие большой сети пиццерии могли взаимодействовать друг с другом, существуют определенные договоренности, форматы и контракты. Наша основная задача - изучить какими бывают контракты и научиться их тестировать.

План курса



На этом уроке

1. Рассмотрим понятия клиент-серверная архитектура, балансировщик и познакомимся с видами клиент-серверной архитектуры.

Клиент-серверная архитектура

«Клиент — сервер» (англ. client–server) — вычислительная или сетевая архитектура, в которой сетевая нагрузка распределена между поставщиками услуг, называемыми серверами, и заказчиками услуг, называемыми клиентами. Иными словами, клиент и сервер — это обычное программное обеспечение. Такие программы расположены на разных вычислительных машинах и взаимодействуют между собой через вычислительную сеть посредством сетевых протоколов, но они могут быть расположены также и на одной машине.

Например, у вас установлено мобильное приложение Ozon, Юла, Wildberries – это клиент. А сервер – это вычислительная машина, которая может находиться в любой точке Земли и которая хранит данные о вашем профиле и ваших заказ и множество других данных, таких, как названия товаров, цены, картинки и многое другое.

Программы-серверы ожидают от клиентских программ запросы и предоставляют им свои ресурсы в виде данных (например, загрузка файлов посредством HTTP, FTP, потоковое мультимедиа или работа с базами данных) или в виде сервисных функций (например,

работа с электронной почтой, общение посредством систем мгновенного обмена сообщениями или просмотр web-страниц во всемирной паутине). Поскольку одна программа-сервер может выполнять запросы от множества программ-клиентов, ее размещают на специально выделенной вычислительной машине, настроенной особым образом, как правило, совместно с другими программами-серверами, поэтому производительность этой машины должна быть высокой. Из-за особой роли такой машины в сети, специфики ее оборудования и программного обеспечения, ее также называют сервером, а машины, выполняющие клиентские программы, соответственно, клиентами.

Балансировщик

Балансировщик нагрузки (Load Balancer) — сервис, помогающий серверам эффективно перемещать данные, оптимизирующий использование ресурсов доставки приложений и предотвращающий перегрузки.

Он управляет потоком информации между клиентом и сервером. Этот сервис проводит непрерывные проверки работоспособности серверов, чтобы убедиться в их работоспособности. При необходимости подсистема балансировки удаляет неисправные серверы из пула.

Также балансировщики выполняют следующие функции::

- функция разгрузки — защищает от распределенных атак типа «отказ в обслуживании» (DDoS);
- функция прогнозной аналитики — определяет узкие места трафика до того, как они возникнут;
- функция запуска новых виртуальных хранилищ данных при превышении лимитов входящего трафика.

Балансировщики могут быть как отдельными физическими аппаратными устройствами, так и поставляться в виде программы. Аппаратные устройства работают на основе программного обеспечения, оптимизированного под специализированные процессоры. По мере увеличения трафика, поставщик просто добавляет дополнительные устройства балансировки нагрузки для обработки необходимого объема.

Виды клиент-серверной архитектуры

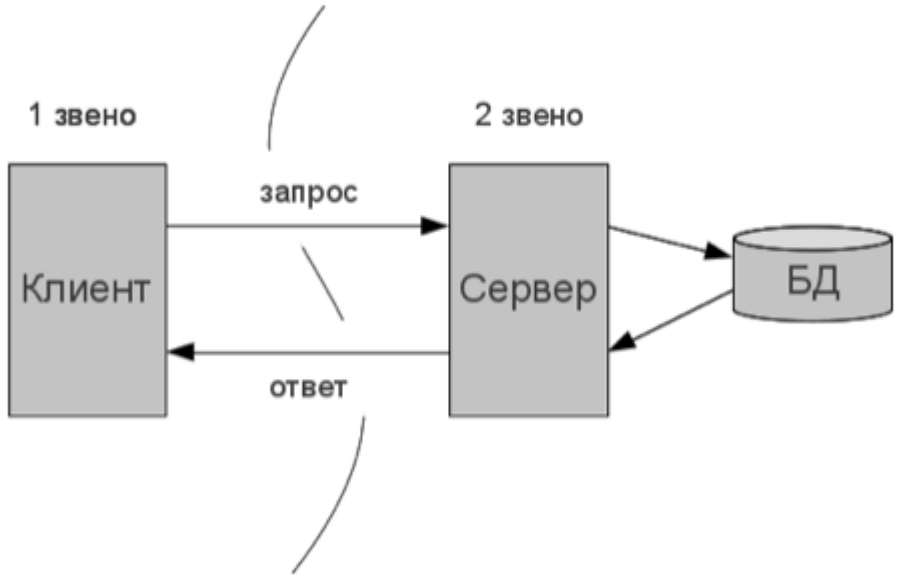
Архитектура «клиент-сервер» определяет общие принципы организации взаимодействия в сети, где имеются серверы, узлы-поставщики некоторых специфичных функций (сервисов) и клиенты (потребители этих функций).
Практические реализации такой архитектуры называются клиент-серверными технологиями.

Существует несколько реализаций:

- Двухзвенная архитектура
- Трехуровневая архитектура
- Многозвенная архитектура
- Монолитная архитектура
- Микросервисная архитектура

Двухзвенная (двухуровневая) архитектура

Двухзвенная архитектура - распределение трех базовых компонентов между двумя узлами (клиентом и сервером). Двухзвенная архитектура используется в клиент-серверных системах, где сервер отвечает на клиентские запросы напрямую и в полном объеме.



Расположение компонентов на стороне клиента или сервера определяет следующие основные модели их взаимодействия в рамках двухзвенной архитектуры:

- Сервер терминалов — распределенное представление данных.
- Файл-сервер — доступ к удаленной базе данных и файловым ресурсам.
- Сервер БД — удаленное представление данных.
- Сервер приложений — удаленное приложение.
- Клиент – это браузер.

Трехзвенная (трехуровневая) архитектура

Трехзвенная архитектура - сетевое приложение разделено на две и более частей, каждая из которых может выполняться на отдельном компьютере. Выделенные части приложения взаимодействуют друг с другом, обмениваясь сообщениями в заранее согласованном формате.

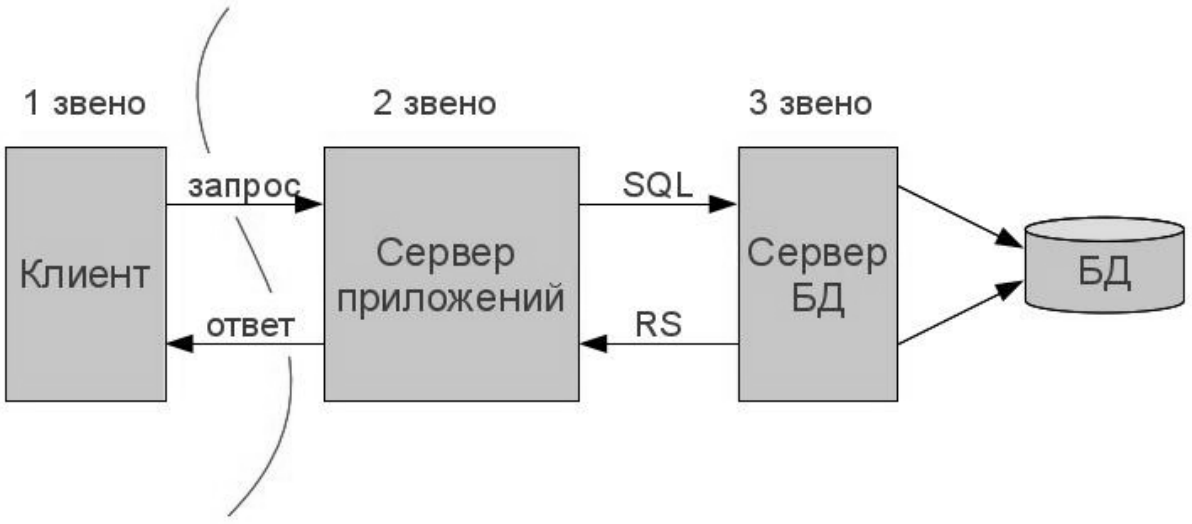
Третьим звеном в трехзвенной архитектуре становится сервер приложений, т.е. компоненты распределяются следующим образом:

- Представление данных — на стороне клиента.
- Прикладной компонент — на выделенном сервере приложений (как вариант, выполняющем функции промежуточного ПО).
- Управление ресурсами — на сервере БД, который и представляет запрашиваемые данные.

Веб-сервер – это сервер, принимающий HTTP-запросы от клиентов и выдающий им HTTP-ответы. Веб-сервером называют как программное обеспечение, выполняющее функции веб-сервера, так и непосредственно компьютер, на котором это программное обеспечение работает. Наиболее распространенными видами ПО веб-серверов являются Apache, IIS и NGINX. На веб-сервере функционирует тестируемое приложение, которое может быть реализовано с применением самых разнообразных языков программирования: PHP, Python, Ruby, Java, Perl и пр.

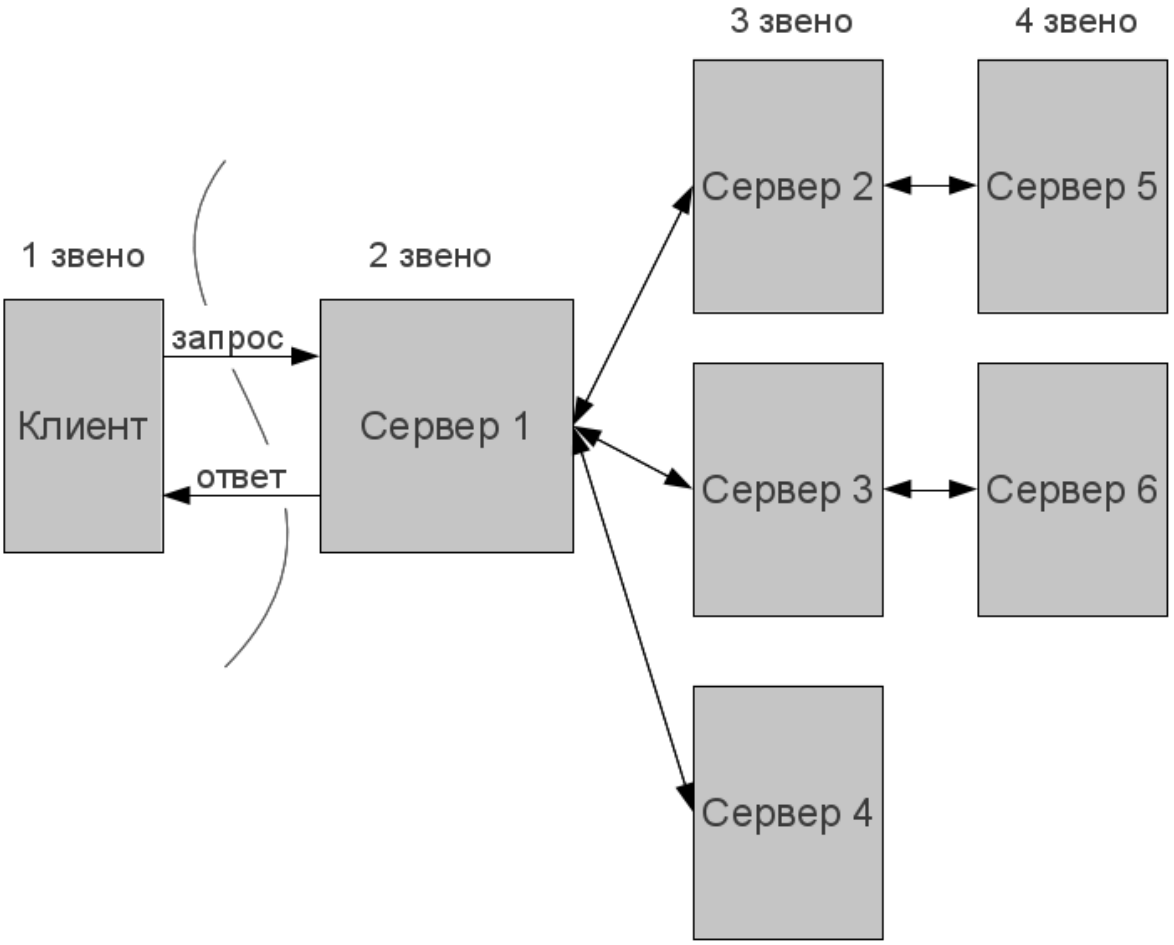
База данных фактически не является частью веб-сервера, но большинство приложений просто не могут выполнять все возложенные на них функции без нее, так как именно в базе данных хранится вся динамическая информация приложения (учетные, пользовательские данные и пр).

База данных – это информационная модель, позволяющая упорядоченно хранить данные об объекте или группе объектов, обладающих набором свойств, которые можно категоризировать. Базы данных функционируют под управлением так называемых систем управления базами данных (далее – СУБД). Самыми популярными СУБД являются MySQL, MS SQL Server, PostgreSQL, Oracle (все – клиент-серверные).



Многозвенная архитектура

Трехзвенная архитектура может быть расширена до многозвенной (N-tier, Multi-tier) путем выделения дополнительных серверов, каждый из которых будет представлять собственные сервисы и пользоваться услугами прочих серверов разного уровня.



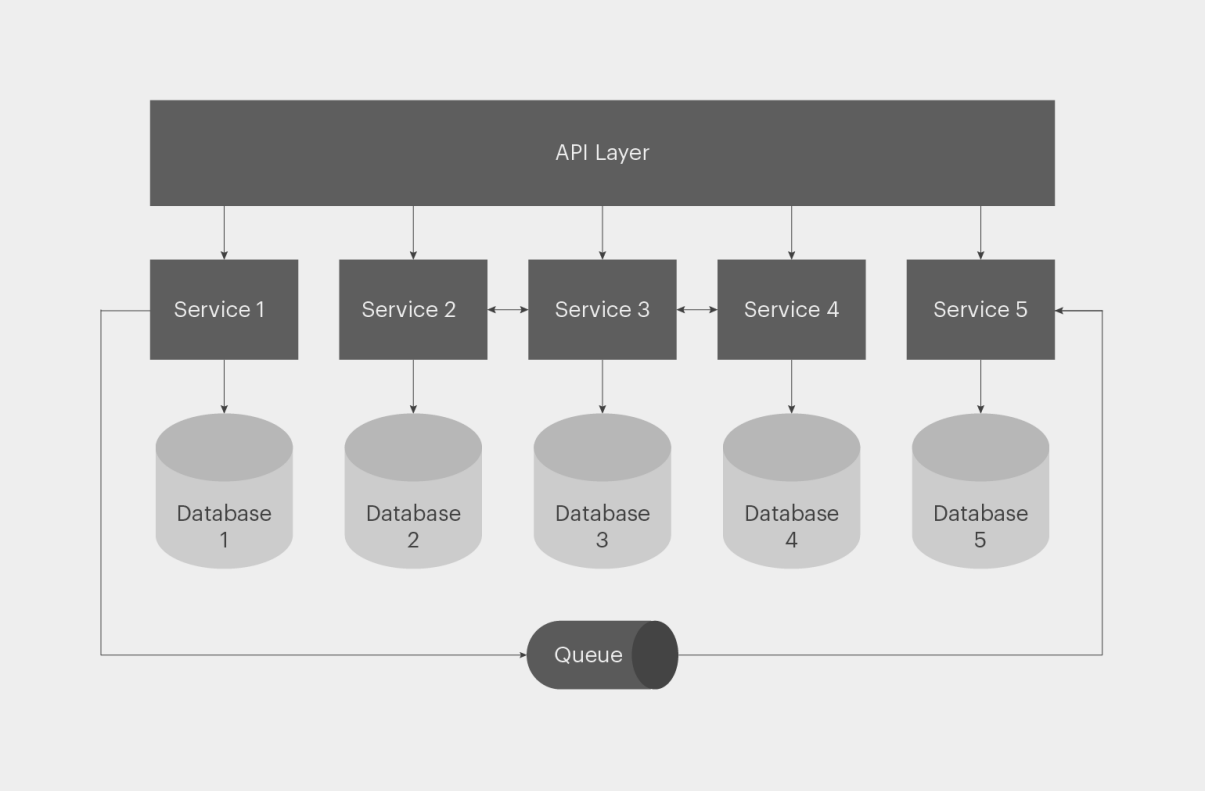
Монолитная архитектура

Монолит — это древнее слово, обозначающее огромный каменный блок. Хотя этот термин широко используется сегодня, представление остается одинаковым во всех областях. В программной инженерии монолитная модель относится к единой неделимой единице. Концепция монолитного программного обеспечения заключается в том, что различные компоненты приложения объединяются в одну программу на одной платформе.

Монолитная архитектура – это архитектура, где приложение представлено в виде единого компонента и представляет собой разрез бизнес-логики, которая модульно прошита. Все клиенты, как правило, обращаются только к одному приложению. Обычно монолитное приложение состоит из:

- базы данных,
- клиентского пользовательского интерфейса,
- серверного приложения.

Микросервисная архитектура



Микросервисная архитектура — это подход к созданию приложения, подразумевающий отказ от единой, монолитной структуры. Приложения с микросервисной архитектурой (MSA) состоят из n небольших подсистем, или микросервисов. Эти подсистемы и их экземпляры могут выполняться как на одной физической машине, так и на нескольких. Во втором случае отказоустойчивость сервиса выше.

Часто в микросервисной архитектуре можно встретить API layer — это общая точка входа в систему. Она балансирует нагрузку и ограничивает доступ к сервисам, но не является обязательным компонентом.

В основе дизайна MSA лежат принципы domain-driven design, в частности ограниченные контексты и домены. Таким образом, каждый микросервис реализует домен или поддомен, имеет собственную базу данных (или не имеет, но у сервисов нет общей БД) и не имеет общей кодовой базы с другими микросервисами. Хотя есть и исключения, например шаблон sidecar.

Микросервисы слабо связаны друг с другом. Чаще всего они общаются синхронно через HTTP или асинхронно с помощью очередей сообщений (RabbitMQ, Redis, Amazon SQS) или логстрима (Kafka, Amazon Kinesis). У такого подхода много преимуществ:

- ➔ Масштабируемость. Благодаря высокой гранулярности и отсутствию централизованных баз данных масштабируемость микросервисной архитектуры достигает наивысшего показателя. Дополнительные экземпляры микросервисов

поднимаются легко, и при этом база нагружается запросами лишь от одного микросервиса, а не от всей системы. А еще мы экономим деньги, потому что масштабируем только те сервисы, на которые идет трафик.

- ➔ Эластичность. Этот показатель тоже достигает экстремально высокого уровня. Новые экземпляры запускаются в течение десятых, а то и сотых долей секунды. При резком увеличении нагрузки на определенные микросервисы система может автоматически поднять дополнительные реплики.
- ➔ Гибкость и изменяемость. Как бы хорошо ни был спроектирован монолит, когда система разрастается, его все тяжелее поддерживать. Поэтому рано или поздно код придется рефакторить. Микросервисы в этом плане очень удобны. Они похожи на кубики лего, из которых можно пересобрать новую фигуру, а если нужно, дополнить проект новыми «кубиками».
- ➔ Скорость и изолированность доставки. Микросервисы просто разрабатывать и деплоить. Отдельный микросервис устроен довольно просто, а значит, его гораздо легче дорабатывать, тестировать и развертывать, чем монолит с комплексной логикой.
- ➔ Изолированность разработки. Много маленьких команд справятся с общей задачей быстрее, чем одна большая. Согласитесь, менеджерить работу в команде из трех человек куда проще, чем в команде из 30.
- ➔ Надежность. Тут тоже все на высоком уровне, хотя не все так однозначно. С одной стороны, изолированность микросервисов повышает отказоустойчивость всей системы, а с другой — чем больше интеграций, тем выше риск отказа. Тем не менее на практике этот показатель у проектов на MSA — выше среднего.

Принцип работы API

API (Application programming interface) — это контракт, который предоставляет программа. «Ко мне можно обращаться так и так, я обязуюсь делать то и это».

Что можно сказать, в целом все API – обмен данными. Обмен данными означает, что обе стороны могут взаимодействовать и есть подход к описанию того как это взаимодействие будет происходить, т.е. есть некоторый формат.

Рассмотрим пример с заказом пиццы в пиццерии. Вы пришли в ресторан, осуществляете заказ, например:

```
{
   "pizza" :
    {
      "dough" : "тонкое",
      "cheese" : "двойной сыр",
      "sausage" : false
    },
```

```
   "price" : 999
}
```

Ресторан принимает заказ, повар готовит пиццу, передает официанту, вам приносят пиццу и ответ будет примерно следующий:

```
{
   "message" : "Заказ сформирован успешно!"
}
```

А если, например, вы передаете вот такие параметры?

```
{
   "pizza" :
    {
      "dough" : утюг,
      "cheese" : "двойной айфон",
      "sausage" : false
    },
   " " : большая цена
}
```

Сервер (в данном случае кухня ресторана, а именно повар) не сможет вас понять и тут, если разработчик учел такой кейс, то будет примерно следующий ответ:

```
{
   "message" : "Такой пиццы нет"
}
```

А если нет? То возможно будет ошибка, вплоть до того, что сервер ляжет.

Как мы уже поняли, то существует некоторый контракт, если он будет нарушен, то запрос не сможет выполниться. Контракт включает в себя:

- саму операцию, которую мы можем выполнить,
- данные, которые поступают на вход,
- данные, которые оказываются на выходе (контент данных или сообщение об ошибке).

Обмен данными был критически важным аспектом корпоративной архитектуры с момента зарождения цифровых вычислений. В основном, обмен информацией происходил внутри компаний, но в какой-то момент потребовалось, чтобы эту информацию пришлось передать другому компьютеру. Ранние формы обмена данными были физическими. Операторы загружали данные на катушки с магнитной лентой. Эти пленки затем перевезли с одного

объекта на другой на расстоянии одной мили. Когда сетевая коммуникация стала стандартизированной, информация стала передаваться в цифровом виде по телефонным линиям и сетевым проводам с использованием протоколов общего назначения, таких как Telnet, SMTP, FTP и HTTP.

Тем не менее, в то время как средства обмена данными стандартизировались, фактические данные, поступающие по проводам, были гораздо менее единообразными. Общепринятых форматов данных не было. У каждой компании и технологии был свой способ структурирования данных.

Но, как говорится, необходимость – мать изобретательности. Отрасль нуждалась в стандартизации – так и произошло. Сегодня существует несколько традиционных форматов данных. Вначале начинали с XML, который используется до сих пор.

Эта стандартизация форматов данных привела к распространению архитектурного дизайна, который позиционирует API как стержень в архитектуре приложения. Сегодняшняя тенденция состоит в том, чтобы клиенты взаимодействовали с уровнем API, представляющим приложение на стороне сервера.

Преимущество подхода на основе API к проектированию архитектуры приложения состоит в том, что он позволяет большому количеству физических клиентских устройств и типов приложений взаимодействовать с данным приложением. Один API можно использовать не только для вычислений на базе ПК, но также для мобильных телефонов и устройств Интернета вещей. Общение не ограничивается взаимодействием между людьми и приложениями. С развитием машинного обучения и искусственного интеллекта взаимодействие между сервисами, поддерживаемое API, станет основным видом деятельности Интернета.

API-интерфейсы открывают новое измерение архитектурному дизайну. Однако, несмотря на то, что сетевое взаимодействие и структуры данных со временем стали более традиционными, форматы API все еще остаются разнообразными. Не существует «единого кольца, которое бы им управляло». Вместо этого существует множество форматов API, наиболее популярными из которых являются SOAP, REST, GraphQL и gRPC.

Типы API

Существует несколько типов API: REST, SOAP, GraphQL, gRPC

SOAP

Simple Object Access Protocol (SOAP) — простой протокол доступа к объектам. Это стандартизированный API с высоким уровнем безопасности. В основном используется в финансовой сфере. Сообщения передаются в формате XML, который представляет собой текстовый формат файла. Он состоит из набора условных меток.

REST

Самый распространенный тип API – **REST API** (Representational State Transfer) — передача состояния представления. В отличие от SOAP, REST является архитектурным стилем, а не протоколом. **Архитектурный стиль** — это согласованный набор ограничений. Для веб-служб, построенных с учетом требований REST, применяют термин «RESTful».

REST API может передавать сообщения в разных форматах — HTML, JSON, XML, или YAML. Самый распространенный — JSON (JavaScript Object Notatio). Он не привязан к языку программирования, в нем меньше слов, его проще писать и читать и у него выше скорость передачи сообщений. API передающее данные в формате JSON называется JSON REST API.

GraphQL

GraphQL – это язык запросов для API для получения данных. Это альтернатива REST API. Он не специфичен для одной платформы и работает для всех типов клиентов, включая Android, iOS или веб. Он располагается между сервером и клиентом и помогает запрашивать данные более оптимизированным способом.

Запросы GraphQL — это сущности, представляющие собой запрос к серверу на получение неких данных. Например, у нас есть некий пользовательский интерфейс, который мы хотим заполнить данными. За этими данными мы и обращаемся к серверу, выполняя запрос. При использовании традиционных REST API наш запрос принимает вид GET-запроса. При работе с GraphQL используется новый синтаксис построения запросов и запросы в этому случае POST:

```
{
	"extensions": {
		"persistedQuery": {
			"version": 1
		}
	},
	"id": "2332a58683ec9efc3c54a64489a7551d610cd1f7a3e485591bdf3fbf4a0c6cd9",
	"operationName": "Feed",
	"variables": {
		"after": "",
		"input": {
			"attributes": [{
				"slug": "categories",
				"value": []
			}],
			"datePublished": null,
			"exb": null,
			"location": {
				"city": null,
				"distanceMax": 50000,
				"latitude": 55.750629000000004,
```

```
                                "longitude": 37.618665
                        },
                        "price": {
                                "from": null,
                                "to": null
                        },
                        "resultType": null,
                        "search": null,
                        "sort": "DEFAULT",
                        "suggestedSubcategory": 0,
                        "suggestedText": null,
                        "viewType": "tile"
                },
                "skipAdvertisements": false,
                "skipCarousels": false,
                "skipStories": false
        }
}
```


Ответ от сервера:


```
{
        "data": {
                "feed": {
                        "__typename": "Feed",
                    "items": [],
                        "pageInfo": {
                                "__typename": "PageInfo",
                                "threshold": 15,
                                "hasNextPage": true,
                                "cursor":
"{\"page\":0,\"totalProductsCount\":30,\"dateUpdatedTo\":1647041341}",
                                "personalSearchId": null,
                                "productsAnalytics": {
                                        "__typename": "AllProductsAnalytics",
                                        "searchId": "5e3a601db28c310002e0911e31616fe5"
                                }
                        }
                }
        },
        "extensions": {
                "tracing": {
                        "version": 1,
                        "startTime": "2022-03-11T23:29:05.731Z",
                        "endTime": "2022-03-11T23:29:08.175Z",
                        "duration": 2443633712,
                        "execution": {
                                "resolvers": []
```

```
                    }
                }
            }
}
```

Это что, JSON? Или JavaScript-объект? Ни то и ни другое. Как мы уже говорили, в названии технологии GraphQL две последние буквы, QL, означают «query language», то есть — язык запросов. Речь идет, в буквальном смысле, о новом языке написания запросов на получение данных. Звучит все это как описание чего-то довольно сложного, но на самом деле ничего сложного тут нет.

gRPC

gRPC – это новый и современный фреймворк для разработки масштабируемых, современных и быстрых API и дословно переводится как система удаленного вызова процедур, разработанный компанией Google еще в далеком 2015 году. Используется многими ведущими компаниями, такими как Google, Square и Netflix, и позволяет программистам писать микросервисы на любом языке, который они хотят, сохраняя при этом возможность легко устанавливать связь между этими сервисами.

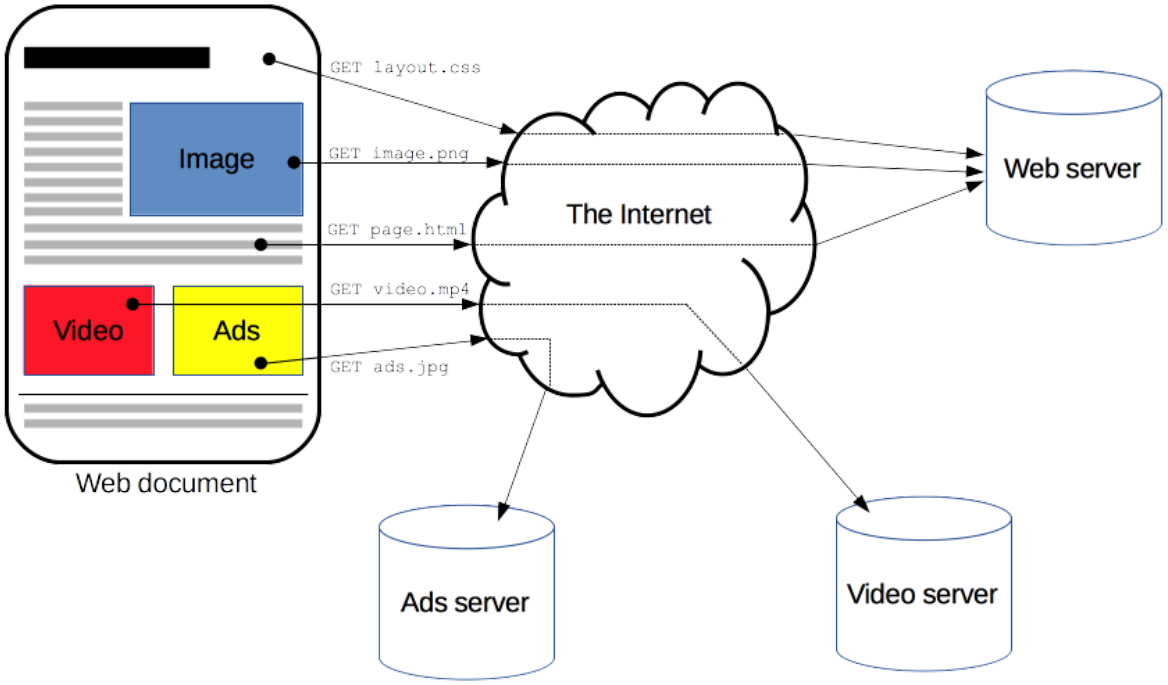
Протоколы HTTP/HTTPS

Протокол HTTP предназначен для передачи содержимого в Интернете. **HTTP** — это простой протокол, который использует для передачи содержимого надежные службы протокола TCP. Благодаря этому HTTP считается очень надежным протоколом для обмена содержимым. Также HTTP является одним из самых часто используемых протоколов приложений. Все операции в Интернете используют протокол HTTP. Подключение в HTTP устанавливается по стандартному TCP-порту 80.

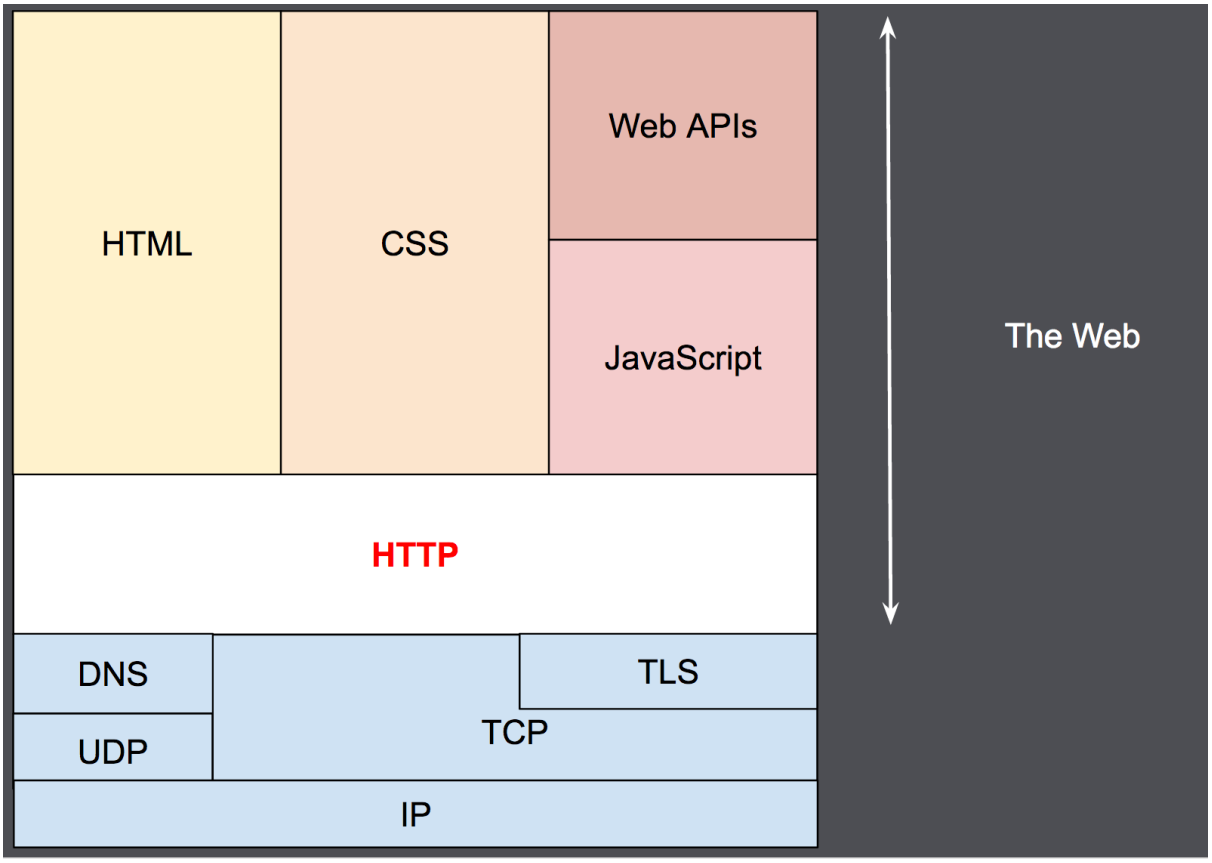
HTTPS — это безопасная версия протокола HTTP, которая реализует протокол HTTP с использованием протокола TLS для защиты базового TCP-подключения. За исключением дополнительной конфигурации, необходимой для настройки TLS, использование протокола HTTPS по сути не отличается от протокола HTTP. Подключение в HTTPS устанавливается по TCP-порту 443.

HTTP

HTTP — это протокол, позволяющий получать различные ресурсы, например HTML-документы. Протокол HTTP лежит в основе обмена данными в Интернете. HTTP является протоколом клиент-серверного взаимодействия, что означает инициирование запросов к серверу самим получателем, обычно веб-браузером (web-browser). Полученный итоговый документ будет (может) состоять из различных документов, являющихся частью итогового документа: например, из отдельно полученного текста, описания структуры документа, изображений, видео-файлов, скриптов и многого другого.



Клиенты и серверы взаимодействуют, обмениваясь одиночными сообщениями (а не потоком данных). Сообщения, отправленные клиентом, обычно веб-браузером, называются запросами, а сообщения, отправленные сервером, называются ответами.



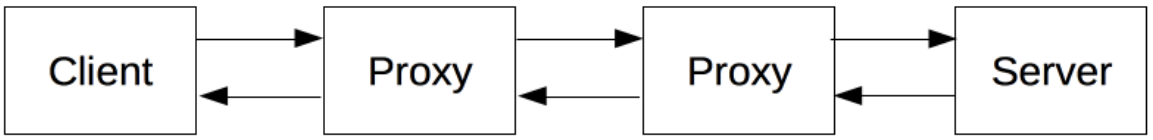
Хотя HTTP был разработан еще в начале 1990-х годов, за счет своей расширяемости в дальнейшем он все время совершенствовался. HTTP является протоколом прикладного уровня, который чаще всего использует возможности другого протокола – TCP (или TLS - защищенный TCP) – для пересылки своих сообщений, однако любой другой надежный

транспортный протокол теоретически может быть использован для доставки таких сообщений. Благодаря своей расширяемости, он используется не только для получения клиентом гипертекстовых документов, изображений и видео, но и для передачи содержимого серверам, например, с помощью HTML-форм. HTTP также может быть использован для получения только частей документа с целью обновления веб-страницы по запросу (например, посредством AJAX запроса).

Составляющие систем, основанных на HTTP

HTTP — это клиент-серверный протокол, то есть запросы отправляются какой-то одной стороной — участником обмена (user-agent) (либо прокси вместо него). Чаще всего в качестве участника выступает веб-браузер, но им может быть кто угодно, например, робот, путешествующий по Сети для пополнения и обновления данных индексации веб-страниц для поисковых систем.

Каждый запрос (англ. request) отправляется серверу, который обрабатывает его и возвращает ответ (англ. response). Между этими запросами и ответами как правило существуют многочисленные посредники, называемые прокси[1], которые выполняют различные операции и работают как шлюзы или кэш, например.



Обычно между браузером и сервером гораздо больше различных устройств-посредников, которые играют какую-либо роль в обработке запроса: маршрутизаторы, модемы и так далее. Благодаря тому, что Сеть построена на основе системы уровней (слоев) взаимодействия, эти посредники "спрятаны" на сетевом и транспортном уровнях. В этой системе уровней HTTP занимает самый верхний уровень, который называется "прикладным" (или "уровнем приложений"). Знания об уровнях сети, таких как представительский, сеансовый, транспортный, сетевой, канальный и физический, имеют важное значение для понимания работы сети и диагностики возможных проблем, но не требуются для описания и понимания HTTP.

Клиент

Участник обмена (user agent) — это любой инструмент или устройство, действующие от лица пользователя. Эту задачу преимущественно выполняет веб-браузер; в некоторых случаях

[1] **Прокси-сервер** - это промежуточная программа или компьютер, используемый при навигации по разным сетям Интернета. Они облегчают доступ к контенту во всемирной паутине. Прокси-сервер перехватывает запросы и возвращает ответы; он может пересылать запросы или нет (например, в случае кеша), и он может изменять его (например, изменяя его заголовки на границе между двумя сетями). Прокси-сервер может находиться на локальном компьютере пользователя или в любом месте между компьютером пользователя и конечным сервером в Интернете.

участниками выступают программы, которые используются инженерами и веб-разработчиками для отладки своих приложений.

Браузер всегда является той сущностью, которая создает запрос. Сервер обычно этого не делает, хотя за многие годы существования сети были придуманы способы, которые могут позволить выполнить запросы со стороны сервера.

Чтобы отобразить веб страницу, браузер отправляет начальный запрос для получения HTML-документа этой страницы. После этого браузер изучает этот документ и запрашивает дополнительные файлы, необходимые для отображения содержания веб-страницы (исполняемые скрипты, информацию о макете страницы - CSS таблицы стилей, дополнительные ресурсы в виде изображений и видео-файлов), которые непосредственно являются частью исходного документа, но расположены в других местах сети. Далее браузер соединяет все эти ресурсы для отображения их пользователю в виде единого документа — веб-страницы. Скрипты, выполняемые самим браузером, могут получать по сети дополнительные ресурсы на последующих этапах обработки веб-страницы, и браузер соответствующим образом обновляет отображение этой страницы для пользователя.

Веб-страница является гипертекстовым документом. Это означает, что некоторые части отображаемого текста являются ссылками, которые могут быть активированы (обычно нажатием кнопки мыши) с целью получения и соответственно отображения новой веб-страницы (переход по ссылке). Это позволяет пользователю "перемещаться" по страницам сети (Internet). Браузер преобразует эти гиперссылки в HTTP-запросы и в дальнейшем полученные HTTP-ответы отображает в понятном для пользователя виде.

Веб-сервер

На другой стороне коммуникационного канала расположен сервер, который обслуживает (англ. serve) пользователя, предоставляя ему документы по запросу. С точки зрения конечного пользователя, сервер всегда является некой одной виртуальной машиной, полностью или частично генерирующей документ, хотя фактически он может быть группой серверов, между которыми балансируется нагрузка, то есть перераспределяются запросы различных пользователей, либо сложным программным обеспечением, опрашивающим другие компьютеры (такие как кеширующие серверы, серверы баз данных, серверы приложений электронной коммерции и другие).

Сервер не обязательно расположен на одной машине, и наоборот - несколько серверов могут быть расположены (поститься) на одной и той же машине. В соответствии с версией HTTP/1.1 и имея Host заголовок, они даже могут делить тот же самый IP-адрес.

Прокси

Между веб-браузером и сервером находятся большое количество сетевых узлов, передающих HTTP сообщения. Из-за слоистой структуры большинство из них оперируют также на транспортном сетевом или физическом уровнях, становясь прозрачным на HTTP

слое и потенциально снижая производительность. Эти операции на уровне приложений называются прокси. Они могут быть прозрачными или нет, (изменяющие запросы не пройдут через них), и способны исполнять множество функций:

- caching (кеш может быть публичным или приватными, как кеш браузера)
- фильтрация (как сканирование антивируса, родительский контроль, ...)
- выравнивание нагрузки (позволить нескольким серверам обслуживать разные запросы)
- аутентификация (контролировать доступом к разным ресурсам)
- протоколирование (разрешение на хранение истории операций)

Основные аспекты HTTP

- HTTP - прост

  Даже с большей сложностью, введенной в HTTP/2 путем инкапсуляции HTTP-сообщений в фреймы, HTTP, как правило, прост и удобен для восприятия человеком. HTTP-сообщения могут читаться и пониматься людьми, обеспечивая более легкое тестирование разработчиков и уменьшенную сложность для новых пользователей.

- HTTP - расширяемый

  Введенные в HTTP/1.0 HTTP-заголовки сделали этот протокол легким для расширения и экспериментирования. Новая функциональность может быть даже введена простым соглашением между клиентом и сервером о семантике нового заголовка.

- HTTP не имеет состояния, но имеет сессию

  HTTP не имеет состояния: не существует связи между двумя запросами, которые последовательно выполняются по одному соединению. Из этого немедленно следует возможность проблем для пользователя, пытающегося взаимодействовать с определенной страницей последовательно, например, при использовании корзины в электронном магазине. Но хотя ядро HTTP не имеет состояния, куки позволяют использовать сессии с сохранением состояния. Используя расширяемость заголовков, куки добавляются к рабочему потоку, позволяя сессии на каждом HTTP-запросе делиться некоторым контекстом или состоянием.

HTTP и соединения

Соединение управляется на транспортном уровне, и потому принципиально выходит за границы HTTP. Хотя HTTP не требует, чтобы базовый транспортного протокол был основан на соединениях, требуя только надежность, или отсутствие потерянных сообщений (т.е. как минимум представление ошибки). Среди двух наиболее распространенных транспортных

протоколов Интернета, TCP надежен, а UDP — нет. HTTP впоследствии полагается на стандарт TCP, являющийся основанным на соединениях, несмотря на то, что соединение не всегда требуется.

HTTP/1.0 открывал TCP-соединение для каждого обмена запросом/ответом, имея два важных недостатка: открытие соединения требует нескольких обменов сообщениями, и потому медленно, хотя становится более эффективным при отправке нескольких сообщений, или при регулярной отправке сообщений: теплые соединения более эффективны, чем холодные.

Для смягчения этих недостатков, HTTP/1.1 предоставил конвейерную обработку (которую оказалось трудно реализовать) и устойчивые соединения: лежащее в основе TCP соединение можно частично контролировать через заголовок Connection. **HTTP/2** сделал следующий шаг, добавив мультиплексирование сообщений через простое соединение, помогающее держать соединение теплым и более эффективным.

HTTP поток

Когда клиент хочет взаимодействовать с сервером, являющимся конечным сервером или промежуточным прокси, он выполняет следующие шаги:

1. Открытие TCP соединения: TCP-соединение будет использоваться для отправки запроса (или запросов) и получения ответа. Клиент может открыть новое соединение, переиспользовать существующее или открыть несколько TCP-соединений к серверу.
2. Отправка HTTP-сообщения: HTTP-сообщения (до HTTP/2) являются человекочитаемыми. Начиная с HTTP/2, простые сообщения инкапсулируются во фреймы, делая невозможным их чтение напрямую, но принципиально остаются такими же.

```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

3. Читает ответ от сервера:

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html
```

```
<!DOCTYPE html... (here comes the 29769 bytes of the requested web page)
```

4. Закрывает или переиспользует соединение для дальнейших запросов.

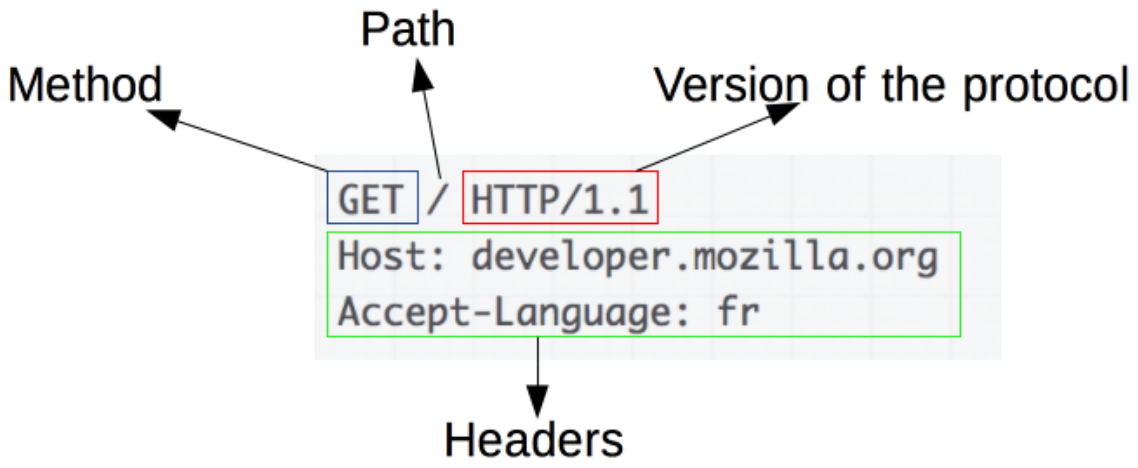
HTTP сообщения

HTTP/1.1 и более ранние HTTP сообщения человекочитаемые. В версии HTTP/2 эти сообщения встроены в новую бинарную структуру, фрейм, позволяющий оптимизации, такие как компрессия заголовков и мультиплексирование. Даже если часть оригинального HTTP сообщения отправлена в этой версии HTTP, семантика каждого сообщения не изменяется и клиент воссоздает (виртуально) оригинальный HTTP-запрос. Это также полезно для понимания HTTP/2 сообщений в формате HTTP/1.1.

Существует два типа HTTP сообщений, запросы и ответы, каждый в своем формате.

Запросы

Примеры HTTP запросов:



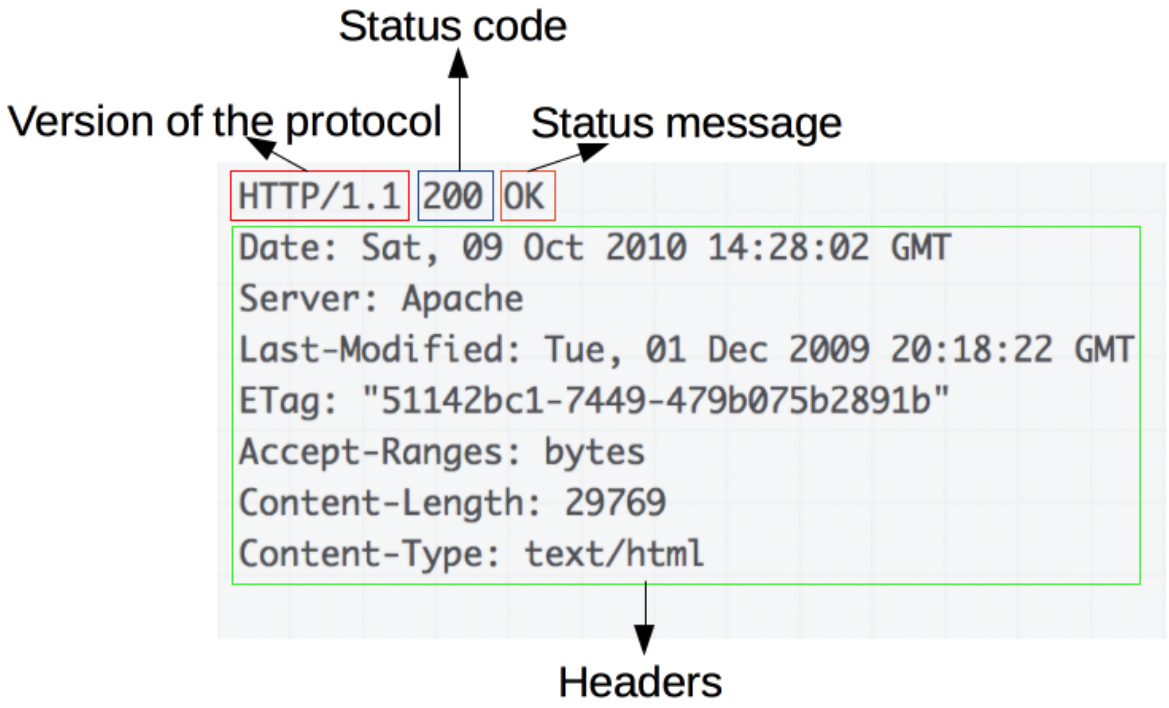
Запросы содержат следующие элементы:

- HTTP-метод, обычно глагол подобно GET, POST или существительное, как OPTIONS или HEAD, определяющее операцию, которую клиент хочет выполнить. Обычно, клиент хочет получить ресурс (используя GET) или передать значения HTML-формы (используя POST), хотя другие операция могут быть необходимы в других случаях.
- Путь к ресурсу: URL ресурсы лишены элементов, которые очевидны из контекста, например без протокола (http://), домена (здесь developer.mozilla.org), или TCP порта (здесь 80).
- Версию HTTP-протокола.
- Заголовки (опционально), предоставляющие дополнительную информацию для сервера.

- Или тело, для некоторых методов, таких как POST, которое содержит отправленный ресурс.

Ответы

Примеры ответов:



Ответы содержат следующие элементы:

- Версию HTTP-протокола.
- HTTP код состояния, сообщающий об успешности запроса или причине неудачи.
- Сообщение состояния — краткое описание кода состояния.
- HTTP заголовки, подобно заголовкам в запросах.
- Опционально: тело, содержащее пересылаемый ресурс.

Swagger

The OpenAPI Specification (с англ. — «спецификация OpenAPI»; изначально известная как Swagger Specification) — формализованная спецификация и экосистема множества инструментов, предоставляющая интерфейс между front-end системами, кодом библиотек низкого уровня и коммерческими решениями в виде API. Вместе с тем, спецификация построена таким образом, что не зависит от языков программирования, и удобна в использовании как человеком, так и машиной.

Относительно назначения, OpenAPI рассматривается как универсальный интерфейс для пользователей (клиентов) по взаимодействию с сервисами (серверами). Если спроектирована спецификация для некоторого сервиса, то на ее основании можно генерировать исходный код для библиотек клиентских приложений, текстовую

документацию для пользователей, варианты тестирования и др. Для этих действий имеется большой набор инструментов для различных языков программирования и платформ.

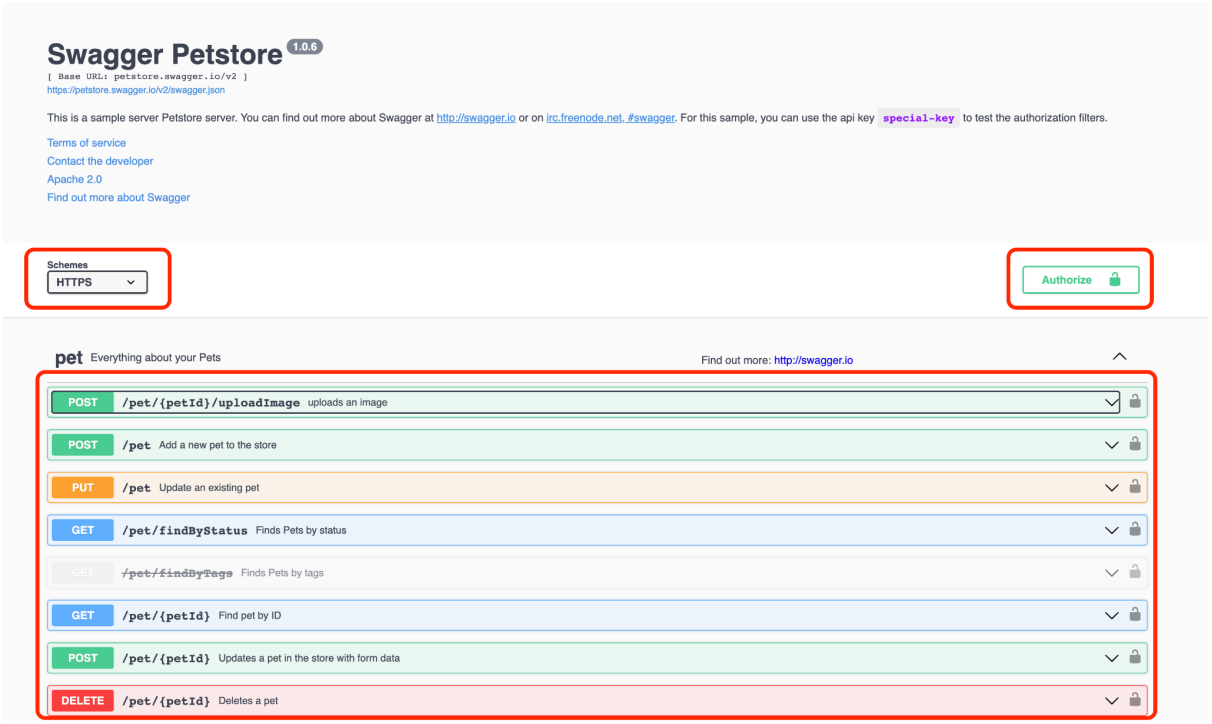
Изначально разработка спецификации под названием Swagger Specification проводилась с 2010 года компанией SmartBear. В ноябре 2015 года SmartBear объявила, что она работает над созданием новой организации Open API Initiative при спонсорской поддержке Linux Foundation.

Swagger – это фреймворк для спецификации RESTful API. Его прелесть заключается в том, что он дает возможность не только интерактивно просматривать спецификацию, но и отправлять запросы – так называемый Swagger UI.

Написание документации осуществляется двумя способами:

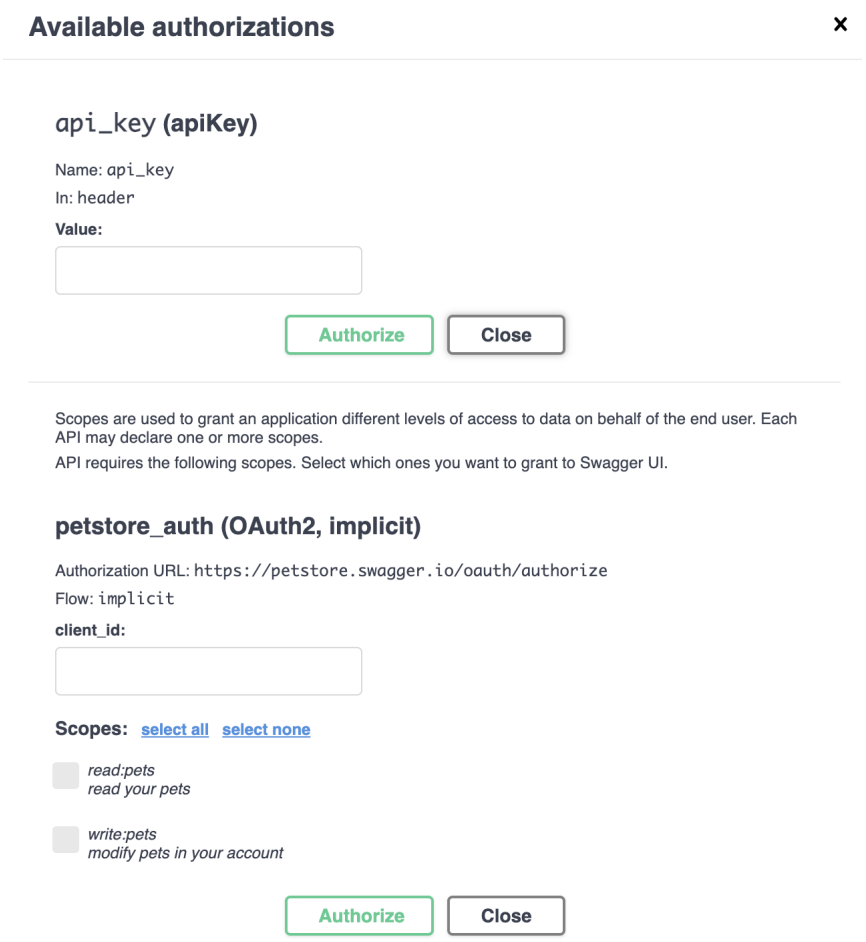
- Автогенерация на основе кода.
- Документация пишется отдельно от кода. Данный подход требует знать синтаксис Swagger Specification. Документация пишется либо в YAML/JSON файле, либо в редакторе Swagger Editor.

Например, имеется swagger одного из сервисов https://petstore.swagger.io/ .

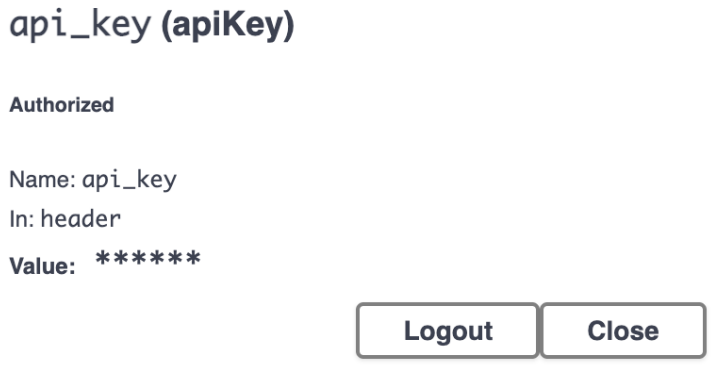


В swagger сервиса Petstore имеется возможность выбора схемы https/http, авторизоваться, а также выполнять запросы к ручкам сервиса.

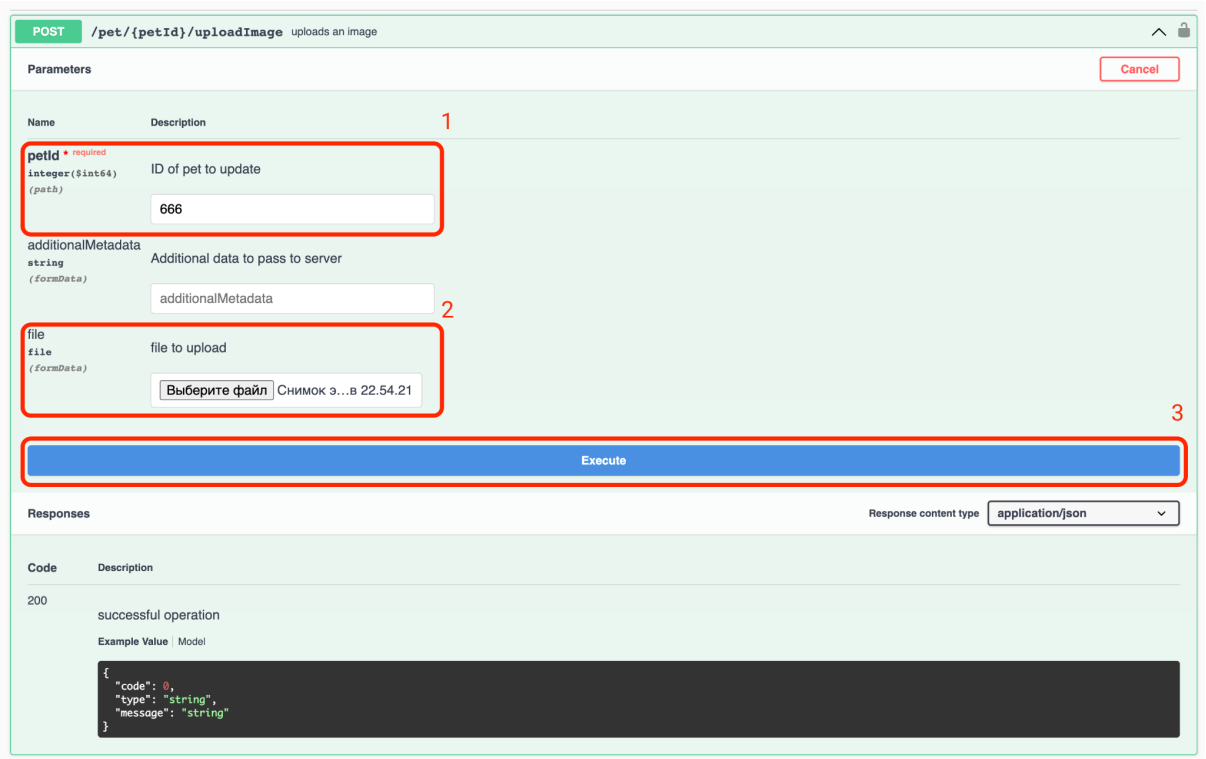
В качестве примера, давайте авторизуемся в сервисе, для этого необходимо выбрать authorize.



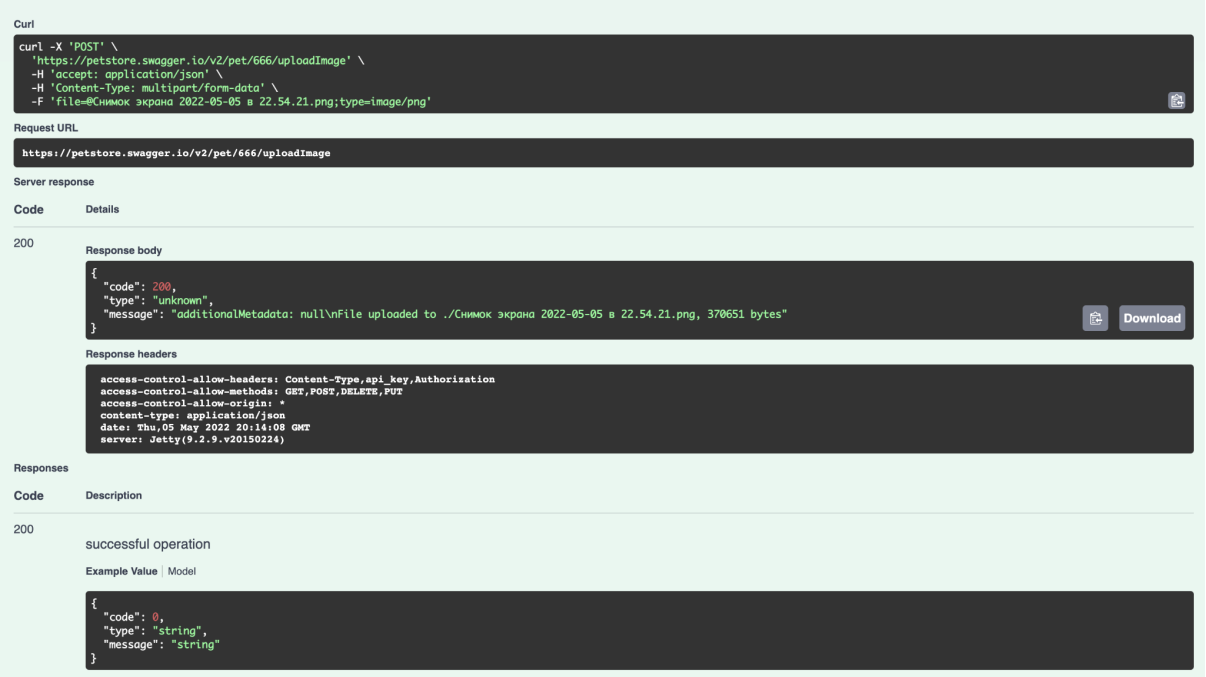
Заполним поля Value, например 123 и нажмем authorize. После этого, пользователь авторизуется.



Следующим шагом нам необходимо выполнить запрос к API, к ручке: /pet/{petId}/uploadImage, заполнив все необходимые для этого поля:



Ниже мы видим выполненный запрос, статус выполненного запроса и ответ.



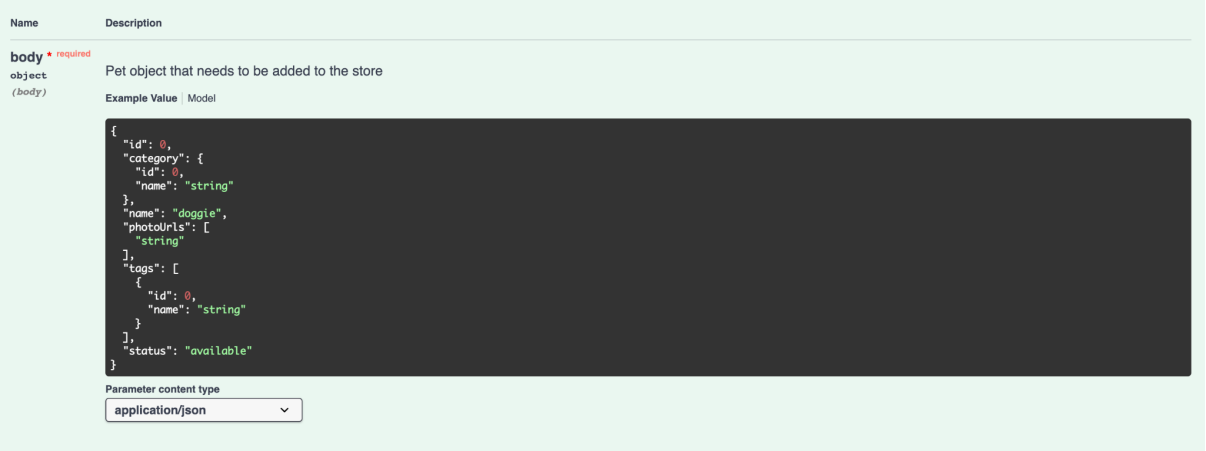
Curl запроса:

```
curl -X 'POST' \
  'https://petstore.swagger.io/v2/pet/666/uploadImage' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'file=@Снимок экрана 2022-05-05 в 22.54.21.png;type=image/png'
```

Ответ от сервера:

```
{
  "code": 200,
  "type": "unknown",
    "message": "additionalMetadata: null\nFile uploaded to ./Снимок экрана 2022-05-05 в 22.54.21.png, 370651 bytes"
}
```

Попробуем добавить нового питомца, для этого выполним запрос к ручке /pet:

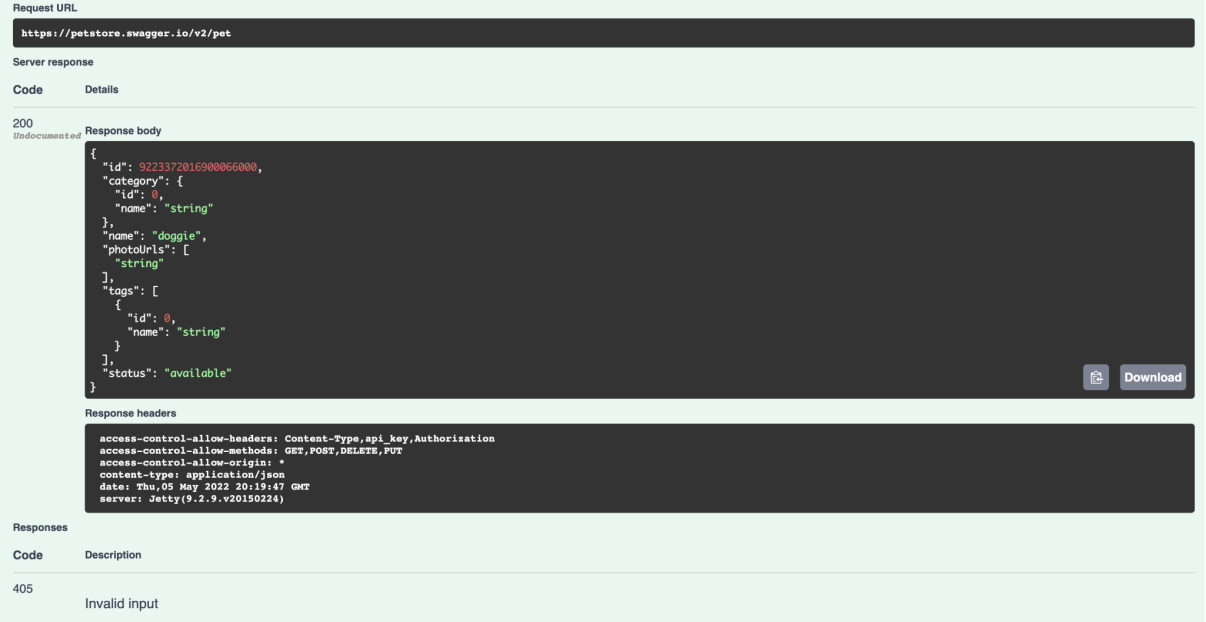


Выполним следующий запрос и передадим следующее тело запроса:

```
curl -X 'POST' \
  'https://petstore.swagger.io/v2/pet' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "id": 0,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "doggie",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}'
```

В ответе получим успешный ответ с созданным питомцем:


```
{
  "id": 9223372016900066000,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "doggie",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}
```

Заключение

Термины, используемые в лекции

«Клиент — сервер» (англ. client–server) — вычислительная или сетевая архитектура, в которой сетевая нагрузка распределена между поставщиками услуг, называемыми серверами, и заказчиками услуг, называемыми клиентами.

Балансировщик нагрузки (Load Balancer) — сервис, помогающий серверам эффективно перемещать данные, оптимизирующий использование ресурсов доставки приложений и предотвращающий перегрузки.

Двухзвенная архитектура - распределение трех базовых компонентов между двумя узлами (клиентом и сервером).

Трехзвенная архитектура - сетевое приложение разделено на две и более частей, каждая из которых может выполняться на отдельном компьютере.

Веб-сервер – это сервер, принимающий HTTP-запросы от клиентов и выдающий им HTTP-ответы.

База данных – это информационная модель, позволяющая упорядоченно хранить данные об объекте или группе объектов, обладающих набором свойств, которые можно категоризировать.

Монолитная архитектура – это архитектура, где приложение представлено в виде единого компонента и представляет собой разрез бизнес-логики, которая модульно прошита.

Микросервисная архитектура — это подход к созданию приложения, подразумевающий отказ от единой, монолитной структуры.

API (Application programming interface) — это контракт, который предоставляет программа. «Ко мне можно обращаться так и так, я обязуюсь делать то и это».

Simple Object Access Protocol (SOAP) — простой протокол доступа к объектам. Это стандартизированный API с высоким уровнем безопасности.

REST API (Representational State Transfer) — передача состояния представления. В отличие от SOAP, REST является архитектурным стилем, а не протоколом.

GraphQL – это язык запросов для API для получения данных.

Запросы GraphQL — это сущности, представляющие собой запрос к серверу на получение неких данных.

gRPC – это новый и современный фреймворк для разработки масштабируемых, современных и быстрых API и дословно переводится как система удаленного вызова процедур, разработанный компанией Google еще в далеком 2015 году.

HTTP — это простой протокол, который использует для передачи содержимого надежные службы протокола TCP.

HTTPS — это безопасная версия протокола HTTP, которая реализует протокол HTTP с использованием протокола TLS для защиты базового TCP-подключения.

Участник обмена (user agent) — это любой инструмент или устройство, действующие от лица пользователя.

The OpenAPI Specification (с англ. — «спецификация OpenAPI»; изначально известная как Swagger Specification) — формализованная спецификация и экосистема множества инструментов, предоставляющая интерфейс между front-end системами, кодом библиотек низкого уровня и коммерческими решениями в виде API.

Swagger – это фреймворк для спецификации RESTful API.

Контрольные вопросы

1. Какие существуют виды клиент-серверной архитектуры?
2. Какие типы API вы знаете? Чем они между собой различаются?
3. Чем отличается HTTP от HTTPS?
4. Пользуется ли тестировщик Swagger и можно ли его считать полноценной документацией на которую можно опираться при тестировании?

Что можно почитать еще?

1. https://habr.com/ru/post/495698/
2. https://www.ittelo.ru/news/tekhnologiya-klient-server-chto-eto-takoe/
3. https://software-testing.ru/edu/1-schedule/271-rest-api

Используемая литература

1. https://developer.mozilla.org/ru/docs/Web/HTTP
2. https://help.reg.ru/hc/ru/articles/4408054713617-%D0%A7%D0%B5%D0%BC-%D0%BE%D1%82%D0%BB%D0%B8%D1%87%D0%B0%D1%8E%D1%82%D1%81%D1%8F-%D0%BF%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB%D1%8B-HTTP-%D0%B8-HTTPS