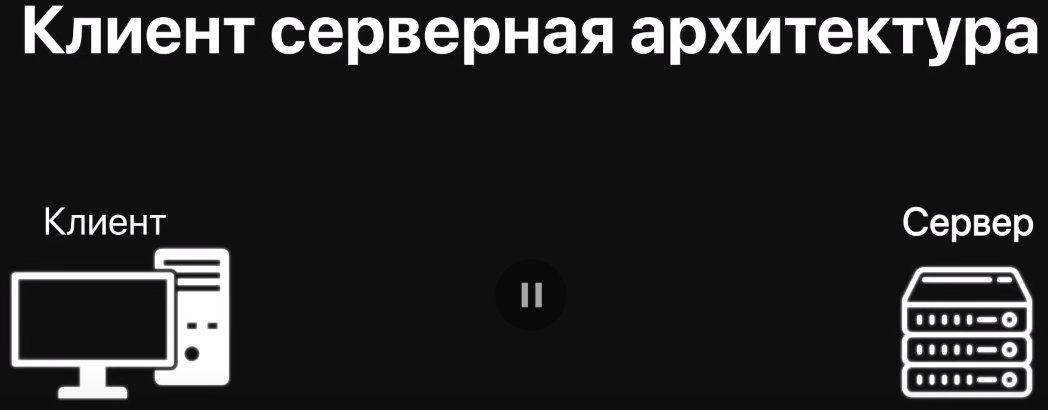
* 1. **Клиентская часть распределённого в сети приложения**………………………...
  2. **Основы клиент-серверной архитектуры** ……………………………………
  3. **Основа** **WEB**……………………………………………………………………..
  4. **Rest** **API**…………………………………………………………………………..
  5. **SOAP**………………………………………………………………………………
  6. **Сравнение** **REST API и SOAP**…………………………………………………
  7. **Клиентская часть распределённого в сети приложения**
  8. **Основы клиент-серверной архитектуры**

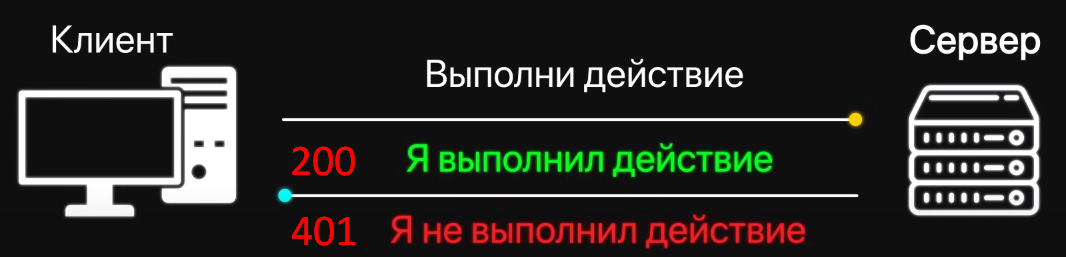
REST API, SOAP, GraphQL - объединяет эти технологии то, что каждая из них так или иначе определяет способ взаимодействия частей распределенного в сети приложения на основе клиент-серверной архитектуры.



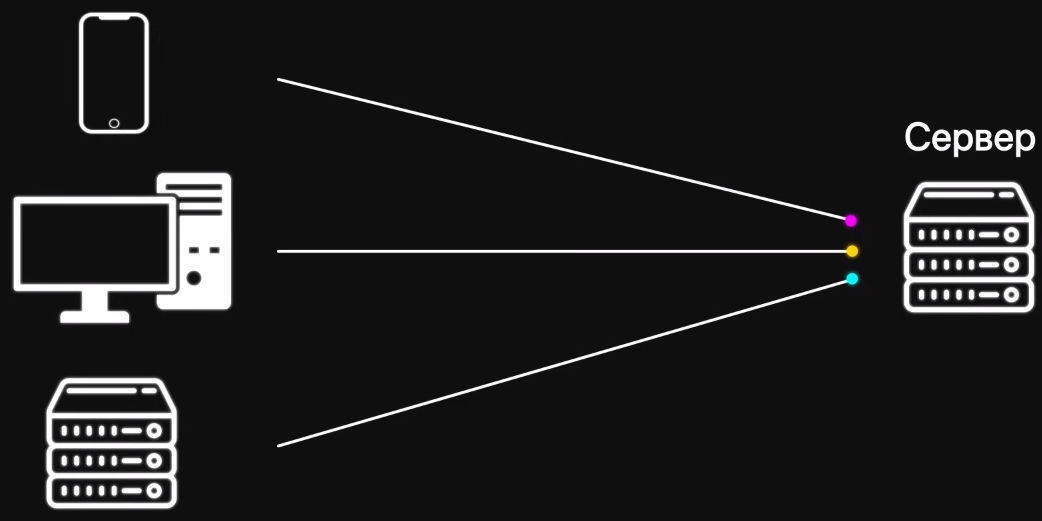
Сервер (поставщик) – источник вычислений, обработки, хранения и кеширования данных. Под сервером понимается некоторый хост, на котором в рантайме выполняется серверная часть приложения. Сервер может быть сколь угодно большим и сложным, будь то мощный виртуальный сервер, который распределяет запросы между дата-центрами или один единственный небольшой сервер. Серверная часть приложения взаимодействует с базой данных, определяет протоколы взаимодействия с ней, определяет API. API – application programming interface, interface – контракт. Обе стороны должны соблюдать некоторый контракт в процессе общения для того чтобы оно было максимально понятным, прозрачным и предсказуемым.   
 Всё это работает в формате «запрос-ответ». Клиент, следуя API, контракту, делает запрос на сервер, в свою очередь, сервер, следуя API, отвечает на клиентский запрос. Например:



Запросы могут быть не только на получение данных, но и на создание новых (например, авторизуй пользователя)



В качестве клиента могут выступать не только браузеры, но и мобильные приложения и даже другие серверы. Следовательно, сервер – поставщик данных/услуг, клиент – потребитель данных/услуг.

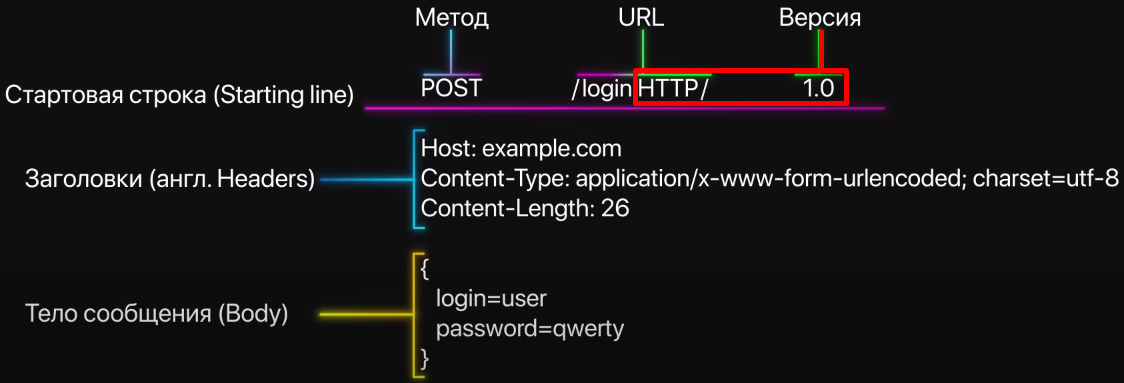


* 1. **Основа WEB**

**Запрос**

HTTP – основа, за счёт которой осуществляется взаимодействие частей распределенного в сети приложения. HTTP – протокол прикладного уровня. По этому пртоколу можно передавать текстовые данные, файлы, данные в формате html,xml,json

Структура произвольного HTTP запроса:

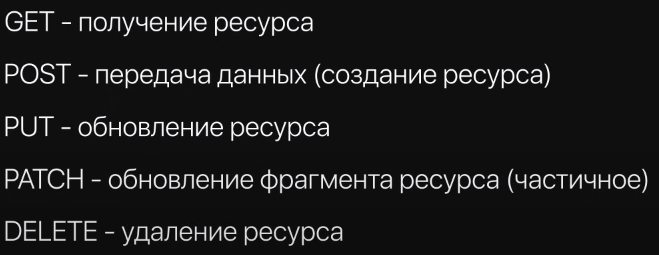


• Стартовая строка – содержит метод (семантику) запроса, т.е. что мы хотим сделать: создать ресурс, обновить, прочитать, удалить. В данном случае, метод «post» говорит о том, что хотим создать ресурс, либо выполнить какое0-то действие. URL – эндпоинт API, к которому обращаемся. Версия – версия HTTP протокола по которому осуществляется запрос.

• Заголовки – несут в себе сведения о данных. Есть обязательные заголовки и необязательные. Заголовки могут быть также кастомными. В заголовках указывается информация о хосте с которого выполняется запрос, информация о браузере, о типе устройства, тип контента (JSON, text). Могут отправляться авторизационные загололовки с токенами (access token, refresh token и т.д.). С помощью загловком обеспечивается безопасность взаимодействия различных источников в сети, т.н. CORS-заголовки.

• Тело сообщения – передача данных, полезной нагрузки (payload). Например, имя пользователя и пароль в процессе аутентификации.

Виды методов для запроса:

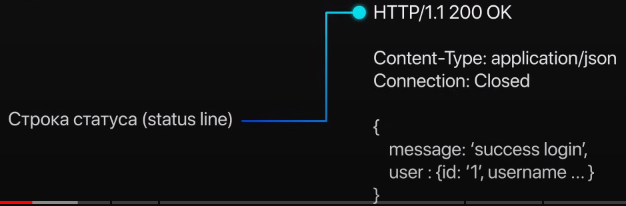


В зависимости от того, какой метод запроса указал клиент, сервер может осуществлять процесс обработки по-разному.

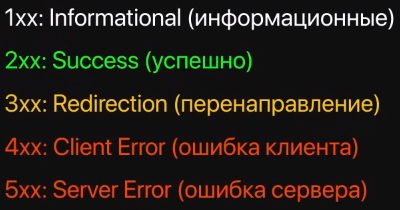
Методы запросов – семантика. Ничто не мешает вместо DELETE использовать POST-метод с пустым body. Или использовать вместо POST-метода – метод GET с непустым body.

**Ответ**

Вместо стартовой строки – строка статуса, в которой указана версия HTTP и статус-код обработки запроса от клиента.



Группы статус-кодов:



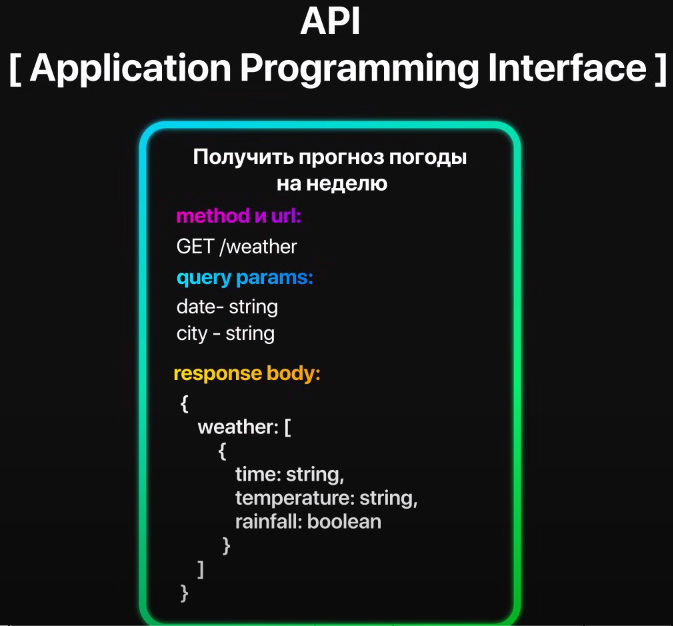
Статус-коды – также про семантику, как и методы запросов. Ничто не мешает на любую клиентскую ошибку серверу возвращать 400 статус код (некорректно составлен запрос с клиентской стороны, т.е. нарушен API).

Вывод: семантика методов запроса и ответных статус-кодов очень важна. Процесс программирования шутка такая, что если разработчикам что-то явно не запретить – они обязательно это могут нарушить. Хотелось бы как-то это все привести к единообразию, стандартизировать, унифицировать, ввести какой-то архитектурный стиль, который будет говорить о том – как делать правильно, по стандарту, и как делать не нужно. И здесь в игру вступает REST API.

* 1. **Rest API**

REST API – архитектурный стиль, проверенный временем, который содержит в себе описание наиболее прозрачного и эффективного взаимодействия клиента с сервером поверх HTTP.

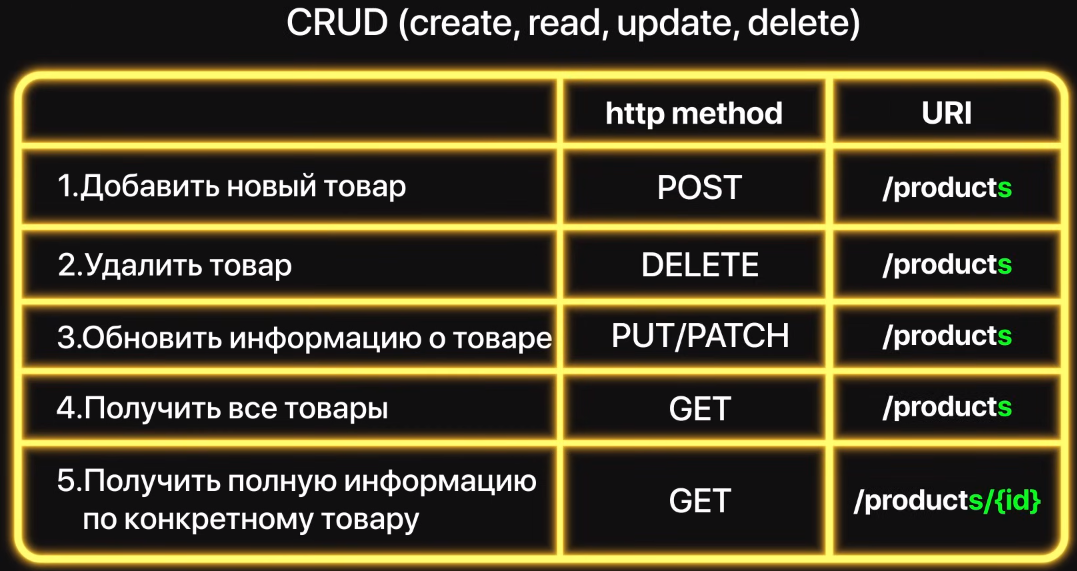
Пример эндпоинта API для получения прогноза погоды. Из описания следует – нужно сделать запрос методом GET по соответствующему эндпоинту API. В query params передать дату и локацию. И приведено описание тела ответного запроса при 200 статус коде.



API может документироваться, могут описываться эндпоинты и входные параметры, различные выходные ответы со статус-кодами.

Концепции REST API:

1. **Модель взаимодействия с REST API** – это клиент-сервер.
2. **Многоуровневость/многослойность системы** (много серверов, микро-сервисов – вот эта структура инкапсулирована от клиента. Клиенту всё равно насколько сложная может быть структуры сервера, как осуществляется процесс балансирования, распределения, проксирования, ротации DNS-имён)
3. **Отсутствие состояния** (stateless). Сервер не должен обладать каким-либо состоянием. При каждом новом запросе клиент и сервер общаются как будто в первый раз. Поэтому при каждом новом запросе от клиента к серверу необходимо снова передавать всю необходимую полную информацию для того, чтобы сервер правильно обработал входящий запрос. Например, пользователь ползает по зоне приложения, которая требует от клиента авторизованного статуса. И при каждом новом запросе со стороны клиента – будут отправляться токены авторизации
4. **Единообразный унифицированный интерфейс,** которым API обладает. Эндпоинты во множественном числе название, так более правильно по REST API. Название токенов авторизации, например, во всех заголовках где они необходимы – одинаково называются, всегда, без исключения. Всё единообразно в соответствии с правилами, которые диктует REST API. Для каждого запроса используется семантически правильный метод, для каждого ответа на запрос используется семантический статус-код.



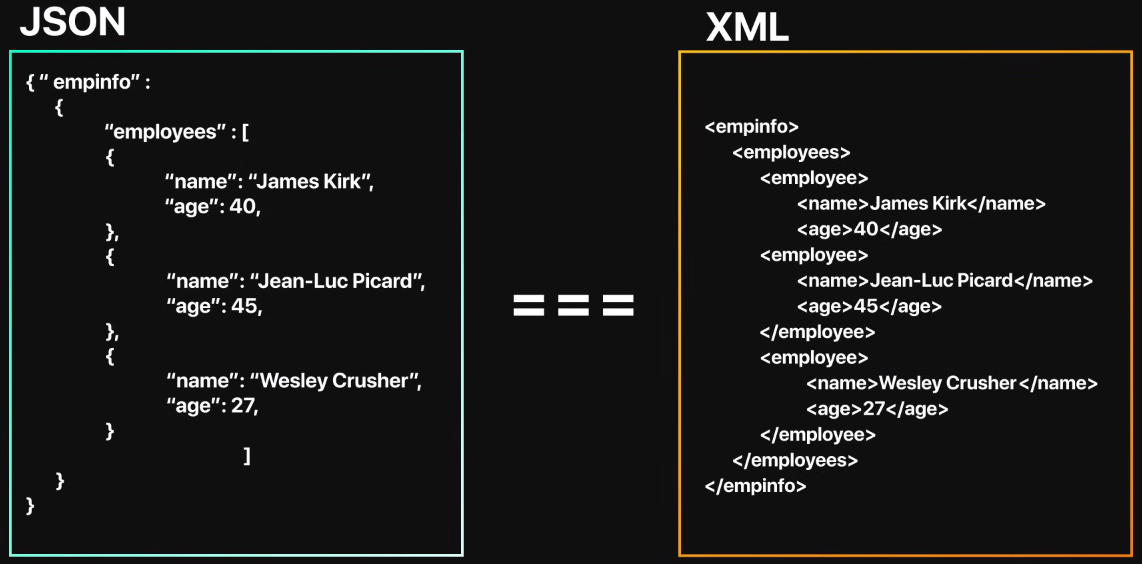
1. **Кэширование**. Может быть осуществлено средствами HTTP, засчёт проставления опредёлнных заголовков, так и средствами сторонними, реализованными на серверной части. Например, Redis.   
   Запросы с методами GET и POST могут быть закешированы, остальные не кэшируются. От кэширования жить всем лучше – и клиенту прилетают данные быстрее и серверу приходится меньше делать работы и нагружаться.
2. **Версионирование**. Внесение правок в API должно быть обратно совместимо. В противном случае, часть приложения, работающая на клиентской стороне, поломается. Расширяемость API достигается посредством разделения API на версии в том случае, если вносимые изменения не могут быть обратно совместимыми.
3. **Документирование**. Свойственно для любого API. У REST есть спецификация по тому, как стоит документировать API. Для документирования API могут быть использованы инструменты: Open API, Swagger. Open API позволяет задокументировать API: какие есть эндпоинты, их методы, статус-коды, какие query-параметры ожидаются на входе, тело запроса, какое тело ответа будет возвращено; версия API. Swagger – имплементация Open API. Набор инструментов для документации и визуализации нашего REST API.

**Идемпотентность запросов**

Идемпотентность – свойство, которое означает, что повторный идентичный запрос, сделанный один или несколько раз подряд, имеет один и тот же эффект, не изменяющий состояние сервера. Корректно реализованные методы GET, PUT, DELETE идемпотенты, но не метод POST. Например, несколько одинаковых, подряд идущих PUT-запросов всегда дадут одинаковый результат, поскольку каждый последующий запрос перезапишет результаты предыдущего. Чего нельзя сказать о методе POST в контексте создания каких-то ресурсов, скажем, регистрация аккаунта. Несколько одинаковых запросов с методом POST и данными для регистрации аккаунта – зарегистрируют аккаунты, создав дубли ресурсов.

**Формат обмена данными**

В большинстве случаев – JSON. Часто используется XML.



* 1. **SOAP**

**SOAP** (от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Simple Object Access Protocol* — протокол доступа к объектам) — [протокол](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D0%B5%D0%B2%D0%BE%D0%B9_%D0%BF%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB) обмена структурированными сообщениями в [распределённой вычислительной среде](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D0%B0%D1%8F_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%B0). Первоначально SOAP предназначался в основном для реализации удалённого вызова процедур ([RPC](https://ru.wikipedia.org/wiki/Remote_Procedure_Call)). Сейчас протокол используется для обмена произвольными сообщениями в формате [XML](https://ru.wikipedia.org/wiki/XML) (Soap-XML), а не только для вызова процедур.

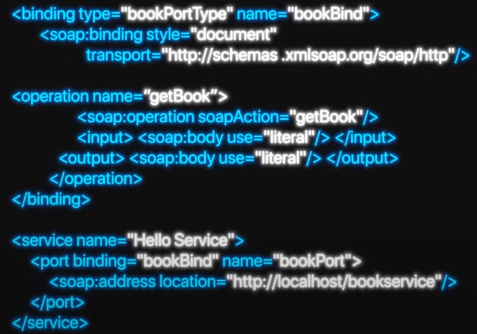


Возможно в рамках одного сервера, в рамках одного бэкэнда, реализовать и REST, и SOAP. Бизнес-логика отдельным ядром, и контроллеры: rest-controller позволяет осуществлять формат обмена данными в формате json, например; soap контроллеры, в которых описан WSDL и которые возвращают XML.



Если REST – набор правил по эффективному использованию HTTP, то SOAP, в свою очередь, может использоваться с любым протоколом прикладного уровня (FTP, HTTP, SMTP и другие)

**WSDL** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) ***W****eb****S****ervices****D****escription****L****anguage* [/ˈwɪz dəl/](https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0_%D0%9C%D0%A4%D0%90_%D0%B4%D0%BB%D1%8F_%D0%B0%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%BE%D0%B3%D0%BE_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0)) — язык описания [веб-сервисов](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B8%D1%81) и доступа к ним, основанный на языке [XML](https://ru.wikipedia.org/wiki/XML). Этот язык используется для описания SOAP-сервисов.



Здесь нет эндпоинтов как в REST, но здесь есть операции. Например, операция «getBook», где в поле «input» пишется то, что ожидается на входе и «output» - то что будет отдано на выходе.

* 1. **Сравнение REST API и SOAP**

REST – множество эндпоинтов.  
SOAP – скорее один эндпоинт, в который надо передать название процедуры, название операции которые надо выполнить. Обладает некоторой строгостью относительно REST, поскольку REST – архитектурный стиль / набор правил, а SOAP – протокол, которой задаёт рамки для разработчиков. Сообщения, которые отправляются в SOAP, обладают определенной структурой. Состоят из 4х частей, 3 обязательных (Envelope, Header, Body) и 1 опциональная (отвечает за ошибки).



• Envelope – корневой элемент, который определяет начало и конец сообщения;

• Header – аналог заголовка HTTP-запроса. Используется для передачи токенов, например;

• Body – используется для передачи полезной нагрузки (payload).

Вывод:

