1. Протокол HTTP: клиент-сервер; типы сообщений, структура запроса, структура ответа, статус (серии значений), методы, заголовки, параметры. Понятие stateless-протокола

Формы взаимодействия:

Клиент -http- Сервер

Клиент -http- Несколько серверов

Несколько клиентов -http- Сервер

Клиент -http- Сервер -http- Сервер

Основные свойства протокола http:

1) ассиметричный протокол - сообщения, которые идут от клиента к серверу, отличаются от сообщений, которые идут от сервера к клиенту

Образуют полудуплексный канал: в каждый момент времени сообщения могут передаваться только в одну сторону или в другую

Http - полудуплекс

WebSocket, Tcp - дуплекс

2) два типа сообщения(request, response)

3) версия протокола

4) два типа абонентов(клиент-сервер)

5) от клиента к серверу реквкст, от сервера клиенту респонс

6) на один реквест один респонс и наоборот

7) порты 80, 443

8) для адресации - URI

9) поддерживается W3C

Request:

1) метод

2) uri

3) версия протокола

4) заголовки

5) параметры

6) расширение

Response:

1) Код состояния

2) Пояснение к коду

3) Версия протокола

4) Заголовки

5) Расширения

Заголовки:

1) прописанные в протоколе http

- request

* **Accept (какой тип контента клиент может обрабатывать)**
* **Accept-Charset (какой тип кодировки клиент может…)**
* **Accept-Encoding (кодировка содержимого (алгоритм сжатия))**
* **Accept-Language (какой язык понимает клиент)**
* **Cookie (доп параметры выданные сервером)**
* **Referer (URI ресурса с которого сделан запрос)**
* **User-Agent (ПО клиента)**

- response

* Age (время сколько клиент находился в proxy-cache)
* **Allow (поддерживаемые методы всего сервера или ресурса)**
* **Location (URL для переадресации запроса)**
* **Server (информация про ПО сервера)**
* **Set-Cookie (отправка браузеру файлов cookie)**
* **Content-Type (указания исходного типа носителя ресурса)**
* Keep-alive (timeout, запрещён в http 2.0)
* Vary (части сообщения которые повлияли на запрос кроме метода и URL)
* Date
* Connection(определяет будет ли соединение открытым после завершения текущей транзакции, запрещён в http 2.0)

- general

* **Connection (указывает должно ли соединения оставаться открытым или закрытым)**
* **Content-Type (тип контента)**
* Upgrade (обновление установленного протокола на другой, в версии HTTP 2.0 уже не используется)

- entity

* **Allow (поддерживаемые методы всего сервера или ресурса)**
* **Content-Encoding (все кодировки применимые для полезной нагрузки сообщения)**
* **Content-Language (язык(и) для носителей которых они предназначены)**
* **Content-Length (размер тела сообщения)**
* **Content-Type (указания исходного типа носителя ресурса)**
* **Expires (дата и время после которого ответ считается просроченным)**

2) кастомные

Коды состояния 5 серий:

1xx: информационные

2хх: успешный ответ

3хх: переадресация

4хх: ошибка клиента

5хх: ошибка сервера

Методы запроса:

GET, POST, PUT, DELETE, PATCH (partial changed), OPTIONS (set params and settings). …

Параметры:

Указываются после URI после знака ?, определяются как ключ-значение, разделяются параметры через знак &. (<https://...info?name=Aleksey&surname=Kravchenko>)

Stateless:

"HTTP is stateless" means that each HTTP transaction (request-response pair) can be processed **independently**(независимы) of any state from previous request-response pair… Stateful – наоборот.

1. Протокол HTTPS: TLS, шифронаборы, сертификаты, процедура рукопожатия. Отличия HTTP-версий протоколов.

Https - это протокол, который часто называют расширение протокола http.

Он добавляет две возможности:

1) аутентифицироваться

2) зашифровать канал

Помогает протокол **TLS – transport layer security** (SSL – secure socket layer) - протокол между HTTP и TCP, настройка над tcp с своим API, который позволяет шифровать.

TLS – это безопасный протокол связи, обеспечивающий шифрование и аутентификацию, и это относилось к SSL до того, как он устарел. Как TLS, так и SSL используют цифровые сертификаты, которые облегчают подтверждение подключения (рукопожатия) и обеспечивают зашифрованную связь между браузером и веб-сервером.

Процедура рукопожатия в результате, которой происходит аутентификация.

Рукопожатие – это процесс аутентификации браузером сертификата SSL или TLS сервера. В ходе этого процесса происходит аутентификация обеих сторон, а затем обмен криптографическими ключами.

Рукопожатие SSL было явным соединением, в то время как рукопожатие TLS – неявным. Процесс рукопожатия SSL состоял из большего количества шагов, чем процесс TLS. TLS ускорил этот процесс, устранив дополнительные шаги и сократив общее количество наборов шифров.

**Сертификат** - это электронный документ, который разработан организацией международный институт телекоммуникаций и который гарантирует, что сервер безопасный

Сертификат содержит:

1) версия

2) издатель

3) серийный номер

4) имя субъекта(кому выдан)

5) перечень доменов(имен URI, на которые распространяется сертификат)

6) открытый ключ

7) период действия

Получаем сертификат от удостоверяющий центров - спец организации, которые выдают этот сертификат (CA – Certificate authority). Никто не уполномочен выдавать сертификаты, кроме этих организаций, они сами себя признали, что они выдают и стали авторитетами.

Обладатель сертификата держит его у себя и мы можем его всегда проанализировать.

На обоих сторонах должен быть установлен сертификат центра сертификации.

Процедура handshake (рукопожатия):

Цель клиент должен убедиться, что сервер это тот, с кем хотят работать

Выработать общий сеансовый ключ

1) ClientHello - клиент отправляет список шифр наборов

**Шифр набор** - это такая совокупность с номер, в котором описывается, как вырабатывается ключ, какое шифрование, какой метод хеширования, какой метод сжатия.

**Набор шифров** – это набор алгоритмов, создающих ключи для шифрования информации между браузером и сервером. Обычно набор шифров включает алгоритм обмена ключами, алгоритм проверки, алгоритм массового шифрования и алгоритм MAC. Несколько алгоритмов в протоколе TLS были модернизированы из SSL по соображениям безопасности.

**Cipher Suites – шифронаборы;** в шифронабор входит: криптосистема для аутентификации сервера и сеансового секрета; шифр для защиты передаваемых данных; хеш-функция для кода аутентификации HMAC.

2) Сервер отправляет тот же набор и свой сертификат с публичным ключом

Генерит сам сеансовый ключ

Шифрует его с помощью публичного ключа сервера

И отправляется его серверу

**TLS Handshake** (no SSL) [Microsoft]:

1. The client sends a "Client hello" message to the server, along with the client's random value and supported cipher suites.
2. The server responds by sending a "Server hello" message to the client, along with the server's random value.
3. The server sends its certificate to the client for authentication and may request a certificate from the client. The server sends the "Server hello done" message.
4. If the server has requested a certificate from the client, the client sends it.
5. The client creates a random Pre-Master Secret and encrypts it with the *[public key](https://learn.microsoft.com/en-us/windows/win32/secgloss/p-gly)* from the server's certificate, sending the encrypted Pre-Master Secret to the server.
6. The server receives the Pre-Master Secret. The server and client each generate the Master Secret and *[session keys](https://learn.microsoft.com/en-us/windows/win32/secgloss/s-gly)* based on the Pre-Master Secret.
7. The client sends "Change cipher spec" notification to server to indicate that the client will start using the new *[session keys](https://learn.microsoft.com/en-us/windows/win32/secgloss/s-gly)* for *[hashing](https://learn.microsoft.com/en-us/windows/win32/secgloss/h-gly)* and encrypting messages. Client also sends "Client finished" message.
8. Server receives "Change cipher spec" and switches its record layer security state to *[symmetric encryption](https://learn.microsoft.com/en-us/windows/win32/secgloss/s-gly)* using the *[session keys](https://learn.microsoft.com/en-us/windows/win32/secgloss/s-gly)*. Server sends "Server finished" message to the client.
9. Client and server can now exchange application data over the secured channel they have established. All messages sent from client to server and from server to client are encrypted using session key.

Все браузеры поддерживают только https/2. Только безопасное соединение

Внутри одного соединения может быть несколько потоков

Псевдо заголовок вначале : например :method

Мультиплексирование - в рамнах одного соединения несколько каналов на разных частотах

Push сервер - когда делаем ответ на сторону клиенты мы можем вместе с респонсом отправить данные которые закешируюся на стороне клиента

Разница между протоколами HTTP:

HTTP/0.9 – 1991.

HTTP/1.0 – 1996, добавлены заголовки, управление версиями (явное указание с какой версией работаем), код состояний, разные типы документов передавать, методы POST и HEAD.

HTTP/1.1 – 1997, постоянное соединение для серии запросов, новые методы (PUT, DELETE, PATCH…)

HTTP/2.0 – 2015, бинарный формат, асинхронные запросы, сброс соединений.

HTTP/3.0 – 2020, используется QUIC соединение (UPD подключение), а не как в предыдущих версиях TCP-соединения с TLS.

1. HTML. Структура HTML-страницы. Каскадные таблицы стилей (CSS). Модель DOM. Модель BOM. Пример.

HTML (от англ. HyperText Markup Language — «язык гипертекстовой разметки») — стандартизированный язык гипертекстовой разметки документов для просмотра веб-страниц в браузере.

**Структура HTML документа:**

<HTML>

Head (settings, meta, imports)

Body (content + scripts)

</HTML>

Версии HTML: 1.0, …, 5.0 (актуальная, 2008), 5.1, 5.2 (2017)

Разработчики: WWWC, Тим Бернерс Ли

Элементы HTML: <теги/>, атрибуты (классы, id, стили, доп информация) (<a href=…/>, <img src=’…/>)

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Тестовая страница</title>

</head>

<body>

<p>Это — моя страница</p>

</body>

</html>

**CSS (Cascading Style Sheets)** — это код, который вы используете для стилизации вашей веб-страницы.

Разработка Хокон Виум Ли

Актуальная версия 3.0

Подключение:

<link href="styles/style.css" rel="stylesheet" type="text/css" />

* Селектор (Selector): Имя HTML-элемента в начале набора правил. Он выбирает элемент(ы) для применения стиля (в данном случае, элементы p ). Для стилизации другого элемента, просто измените селектор.
* Свойства (Properties): Способы, которыми вы можете стилизовать определённый HTML-элемент (в данном случае, color является свойством для элементов <p>). В CSS вы выбираете, какие свойства вы хотите затронуть в вашем правиле.
* Селектор класса, ID, селектор атрибута (img[src] Выбирает <img src="myimage.png"> но не <img>), селектор псевдокласса (a:hover Выбирает <a>, но только тогда, когда указатель мыши наведён на ссылку.)

**DOM** - модель представление браузером HTML страницы.

1) Парсит HTML теги в объекты

2) API

**DOM (Document Object Model)** — это специальная древовидная структура, которая позволяет управлять HTML-разметкой из JavaScript-кода. Управление обычно состоит из добавления и удаления элементов, изменения их стилей и содержимого.

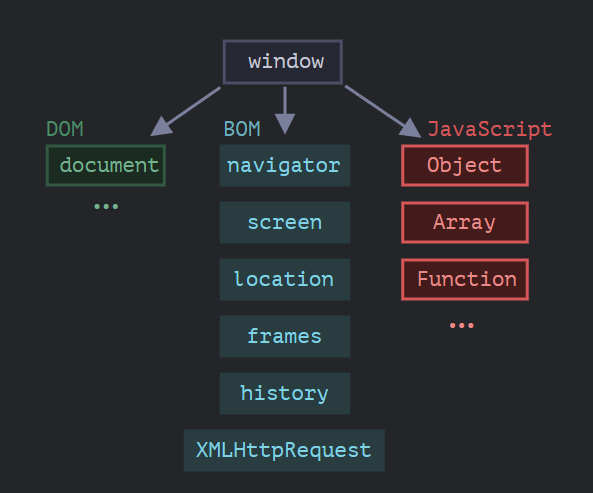
Объект document содержит большое количество свойств и методов, которые позволяют работать с HTML. Чаще всего используются методы, позволяющие найти элементы страницы.

*It represents the page in a tree structure so that programs can read, access and change the document structure, style and content. The DOM is an object-oriented representation in case of the web page, which can be modified with a scripting language such as JavaScript. Every web browser uses some document object model to make web pages accessible via JavaScript.*

Объектная модель браузера **(Browser Object Model, BOM)** – это дополнительные объекты, предоставляемые браузером (окружением), чтобы работать со всем, кроме документа.

Например:

* Объект navigator даёт информацию о самом браузере и операционной системе. Среди множества его свойств самыми известными являются: navigator.userAgent – информация о текущем браузере, и navigator.platform – информация о платформе (может помочь в понимании того, в какой ОС открыт браузер – Windows/Linux/Mac и так далее).
* Объект location позволяет получить текущий URL и перенаправить браузер по новому адресу.



\*React работает с виртуальным DOM, подтягивает изменения только тогда, когда они произошли, т.е. не делает обновление всей объектной структуры при каждом перерендеринге компонента. Обновляется вначале виртуальный DOM, затем вычисляется оптимальный алгоритм для обновления реального DOM и только затем обновляется.

1. Протокол WebSockets: принципы работы и применения. Пример

**Веб-сокеты** это продвинутая технология прикладного уровня, позволяющая открыть постоянное двунаправленное сетевое соединение между браузером пользователя и сервером. С помощью его API вы можете отправить сообщение на сервер и получить ответ без выполнения http запроса, причём этот процесс будет событийно-управляемым.

**WebSocket** – протокол прикладного уровня, основанный на TCP, использующий дуплексный канал связи.

Есть ws(==http), wss(==https).

**Установка соединения:**

1. Клиент посылает обычный HTTP-запрос, называемый рукопожатием, с заголовками:

Connection: Upgrade

Upgrade: Websocket

2. Сервер решает, устанавливать WebSocket-соединение или нет. Если да, то сервер пересылает те же заголовки с кодом ответа 101 Switching Protocols.

3. HTTP-соединение заменяется на WebSocket-соединение, данные можно передавать.

После создания WebSocket мы должны слушать его события, их всего 4:

* Open (connection start)
* Message (received data)
* Error
* Close (connection close)

Применения: мессенджеры, многоюзерские игры, инструменты разработчика, карты…

1. JavaScript. Основные стандарты. Типы данных. Программные структуры. Принцип применения. Пример.

**JavaScript** — это кроссплатформенный, интерпретируемый язык программирования, который используется для создания интерактивных веб-страниц и приложений. Функционал JavaScript может улучшить удобство взаимодействия пользователя с веб-сайтом.

\*Когда JavaScript создавался, у него было другое имя – «LiveScript». Однако, язык Java был очень популярен в то время, и было решено, что позиционирование JavaScript как «младшего брата» Java будет полезно.

Со временем JavaScript стал полностью независимым языком со своей собственной спецификацией, называющейся **ECMAScript**, и сейчас не имеет никакого отношения к Java.

JavaScript создавался как скриптовый язык для Netscape. После чего он был отправлен в ECMA International для стандартизации (ECMA — это ассоциация, деятельность которой посвящена стандартизации информационных и коммуникационных технологий). Это привело к появлению нового языкового стандарта, известного как ECMAScript.

Последующие версии JavaScript уже были основаны на стандарте ECMAScript. Проще говоря, ECMAScript — стандарт, а JavaScript — самая популярная реализация этого стандарта.

ES — это просто сокращение для ECMAScript. Каждое издание ECMAScript получает аббревиатуру ES с последующим его номером.

Версия ES6/ES2015 вышла в июне 2015 года. Это также принесло некую путаницу в связи с названием пакета, ведь ES6 и ES2015 — это одно и то же. С выходом этого пакета обновлений комитет принял решение перейти к ежегодным обновлениям. Поэтому издание было переименовано в ES2015, чтобы отражать год релиза. Последующие версии также называются в соответствии с годом их выпуска.

Текущая версия ES2023.

**Типы данных:**

* String
* Number
* Symbol
* BigInt
* Null
* Undefined
* Object
* Boolean

JS преимущественно применяется для разработки клиентских веб-приложений. Данный язык позволяет взаимодействовать с сервером и динамически обновлять страницу.

1. Методология Ajax. Структура Ajax-приложения, принципы разработки и применения. Пример.

**AJAX** - это методология разработки, которая основывается на основе специального обьъекта xmlhttprequest для ассинхроных запросов.

**Методология** - принципы и правила написания кода.

Структура Ajax-приложения включает следующие элементы:

-XMLHttpRequest Object: Это API в виде объекта, методы которого помогают в передаче данных между веб-браузером и веб-сервером.

-HTML DOM: Когда загружается веб-страница, браузер создает модель объекта документа страницы.

**XmlHttpRequest** - специальный объект, который относится только к браузеру.

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

XMLHttpRequest имеет два режима работы: синхронный и асинхронный.

**Порядок создания запроса**:

* Создать XMLHttpRequest

let xhr = new XMLHttpRequest();

* Инициализировать его (async – true, sync – false (default false))

xhr.open(method, URL, [async, user, password])

* Послать запрос

xhr.send([body])

* Слушать события на xhr, чтобы получить ответ.

onload, onprogress, onerror, onReadyStateChange

* После ответа можем получить результат запроса:

Status – код состояния (200, 401, 500…)

StatusText

ResponseType (text, blob, json…)

readyState

(

UNSENT = 0; // исходное состояние

OPENED = 1; // вызван метод open

HEADERS\_RECEIVED = 2; // получены заголовки ответа

LOADING = 3; // ответ в процессе передачи (данные частично получены)

DONE = 4; // запрос завершён

)

На одной странице можно создать несколько ассинхроных запросов.

Можем динамически изменять содержимое страницы.

**Применение Ajax** включает в себя следующие возможности:

Обновление веб-страницы без ее перезагрузки.

Запрос данных с сервера после загрузки страницы.

Получение данных с сервера после загрузки страницы.

Отправка данных на сервер в фоновом режиме.

**async: оставил заявку и получил\обработал результат**

1. Web-приложение. Архитектура web-приложения. Особенности реализации web-приложения. Web-сервер и web-клиент. Пример.

Разработка веб-приложений - разработка **служб**

Служба Интернет = сервер + протокол.

**Веб-приложение** — это программное обеспечение, которое запускается в веб-браузере.

**Web-программирование** - разработка приложений клиент-сервер, с взаимодействием по протоколу HTTP.

**Web сервис**- такие же веб-приложения, которые предназначены для работы других веб-приложений.

**Web-ресурс приложения**: сущность, расположенная на стороне сервера и имеющая URL/URI, к которой можно сделать http-запрос и получить http-ответ. Одно web-приложение представлено одним или более ресурсов.

Функциональность веб-приложений подразумевает, что они могут работать с несколькими разновидностями страниц, среди которых:

**статистические** — серверная часть создаёт страницу в ответ на запрос и отправляет её в браузер вне зависимости от действий пользователя, так что разные пользователи увидят по одному и тому же запросу одинаковый материал;

**динамические** — серверная часть формирует страницу в ответ на запрос, только материал предварительно проходит через сервер приложений и формируется в зависимости от того, какие команды были отправлены.

**Состав веб приложения**:

* клиентская часть — отвечает за действия, выполняемые пользователем;
* серверная часть — отвечает за процессы, происходящие на сервере;
* база данных — структура для упорядоченного хранения информации и доступа к ней.

**Виды веб приложений** по построению сайта:

* многостраничные (MPA) — запрос отправляется на сервер, а страница полностью обновляется в результате ответа, заменяется на новую;
* одностраничные (SPA) — после отправки запроса на сервер обновляется часть той страницы, из которой состоит приложение, без полной перезагрузки;

Web-приложение: web client + web server

Front: JS+HTML/React/Vue/Angular…

Back: .Net ASP, JAVA Spring, JS NodeJs, Python Django…

1. Сеть Интернет: 4 компонента.

**Интернет** - сеть, в которой использует стек протоколов TCP/IP.

4 компоненты:

**1) Стек протоколов TCP/IP**

**2) Службы интернет** — сервер + протокол / протокол, который описывает, как работать с этим сервером

**3) Набор организаций**, обеспечивающих и улучшающих работу Интернета

Консорциум - организации, для согласования стандартов.

IETF (Internet Engineering Task Force) - организация, инженерная группа, которая предназначена для проектирования интернет, выпуска документации. Публикует RFC (Request for comments).

ICANN (Internet Corporation for Assigned Names and Numbers) - организация, которая ведает управлением именами, распределением IP адресов. Управляет доменными именами.

W3C(World wide web corporation) - организация, разрабатывающая web-стандарты (HTTP, HTML, CSS, DOM, XML…). Возглавляет Тим Бернерс Ли.

IANA (Internet Assigned Numbers Authority) - Администрация адресного пространства Internet. Под контролем ICANN. Кроме того регистрирует типы данных MIME. MIME - формат данных, пересылаемых данных.

ISO - международная стандартизирующая организация

IAR - совет по архитектуре интернета

**4) Документация**

RFC, ISO

1. Понятие службы Интернет.

**Internet служба**(Служба = сервис) - internet ресурс

**Служба Интернет** = сервер + протокол/ протокол, который описывает, как работать с этим сервером.

**Протокол** – набор согласованных правил, которые устанавливают порядок установки соединения и обмена данными между пользователями сети Интернет.

**Се́рвер** (англ. server – обслуживающий), компьютер (условно) для обслуживания поступающих от других компьютеров запросов (например, посредством вычислительной сети), а также программа, обеспечивающая управление доступом к сетевым ресурсам. Основная задача сервера – распределение какого-либо ресурса между использующими данный сервер клиентами. (Готовые веб-сервера: IIS Express, Apache Tomcat, GlassFish)

1. Понятие ресурса Интернет.

**Internet ресурс** - все, что имеет адрес в интернете

**Программирование в интернет** - взаимодействие между **ресурсами** интернет по протоколу http, pop3, imap

**Программирование в Internet** - разработка распределяемого приложения, представляющий собой один или множество связанных **internet ресурсов.**

**Internet ресурс** - сущность, расположенная на стороне сервера и имеющая URL/URI, к которой можно сделать http-запрос и получить http-ответ. Одно web-приложение представлено одним или более ресурсов.

**Узел Интернет** – это устройство, имеющее IP-адрес и подключенное к сети Интернет (как правило, к сети Интернет-провайдера). Каждый узел характеризуется своей программно-аппаратной платформой (аппаратура + ОС).

1. Понятие MIME.

Медиа тип (так же известный как Multipurpose Internet Mail Extensions или **MIME** тип) является стандартом, который описывает природу и формат документа, файла или набора байтов. Он определён и стандартизирован в спецификации RFC.

**MIME**: **Multipurpose Internet Mail Extensions** - многоцелевые расширения Internet-почты. Используется и как стандарт кодирования Internet-сообщений.

Это формат данных, который может пересылаться в Internet.

1. HTTP-переадресация: принцип работы, заголовки, статусы

**HTTP-переадресация** - это процесс, при котором сервер отправляет специальный ответ на запрос, указывая браузеру перейти на другой URL. Это достигается с помощью статусов ответа, начинающихся с 3, и заголовка Location, содержащего URL для переадресации.

Принцип работы HTTP-переадресации:

* Сервер отправляет специальный ответ на запрос, который называется переадресацией.
* Ответы на переадресацию имеют статусы, которые начинаются с 3, и заголовок Location, содержащий URL для переадресации.
* Когда браузеры получают переадресацию, они немедленно загружают новый URL, указанный в заголовке Location.

**Заголовки HTTP-переадресации:**

Location: Этот заголовок содержит URL, на который следует перенаправить.

**Статусы HTTP-переадресации**:

* 301 Moved Permanently: Этот статус указывает, что запрашиваемый ресурс был окончательно перемещен на новый URL, указанный в заголовке Location.
* 302 Found: Этот статус указывает, что запрашиваемый ресурс временно доступен по другому URL, указанному в заголовке Location.
* 303 See Other: Этот статус используется для переадресации после выполнения операций PUT или POST, чтобы обновление страницы результатов не приводило к повторному выполнению операции.
* 307 Temporary Redirect: Этот статус указывает, что запрашиваемый ресурс временно доступен по другому URL, указанному в заголовке Location. Это лучше, чем 302, когда на сайте доступны операции, отличные от GET.
* 308 Permanent Redirect: Этот статус указывает, что запрашиваемый ресурс был окончательно перемещен на новый URL, указанный в заголовке Location. Метод и тело не изменяются.

301 и 302 считаются устаревшими. В спецификации HTTP/1.1 им на замену пришли аналогичные 307 и 308.

Вся разница в том, что 301 и 302 могли изменить метод запроса: например, делается запрос POST /api/users, возвращается 302 status code, и делается новый запрос GET /api/users. В 307 и 308 такой проблемы нет, и метод запроса всегда будет сохраняться при редиректе.

1. HTTP: кеширование на стороне клиента, принцип, заголовки.

**Кеширование на стороне клиента** в HTTP - это процесс, при котором браузер пользователя сохраняет ресурсы, которые он уже получил, такие как изображения и скрипты1. Таким образом, когда пользователь возвращается на ваш сайт, его браузер может быстро извлечь кешированные ресурсы, вместо того чтобы снова их загружать.

**Принцип работы кеширования** на стороне клиента:

При первом посещении сайта браузер сохраняет кешируемые ресурсы, такие как веб-страница.

При следующем посещении этой страницы ресурсы загружаются из локального хранилища, а не загружаются заново с сервера.

Это снижает нагрузку на сервер и уменьшает потребление пропускной способности, что увеличивает производительность.

HTTP заголовки, связанные с кешированием:

**Cache-Control**: Этот заголовок содержит директивы (инструкции), которые контролируют кеширование в браузерах и общих кешах (например, прокси, CDN). Он может содержать множество директив, которые определяют поведение кеширования8. Некоторые из них включают:

max-age: Определяет максимальное время (в секундах), в течение которого ресурс считается свежим.

no-cache: Указывает, что кеш должен перепроверить свежесть ресурса перед его использованием.

no-store: Указывает, что ресурс не может быть кеширован.

private: Указывает, что ресурс может быть кеширован только в приватном кеше (например, в браузере пользователя).

public: Указывает, что ресурс может быть кеширован в любом кеше.

1. Основные объекты web-сервера: кэш, кэш-менеджер.

Кэш (Cache): серверный объект, предназначенный для временного хранения данных с целью ускорения выполнения запроса. Кэширование – процессы записи и извлечения данных в/из Cache. Различают кэширование данных и кэширование вывода. Кэширование данных – кэширование часто используемых данных. Кэширование вывода – кэширование объекта Response.

Мы кешируем response.

**Cachmanager**: приходит запрос, он его запускает на обработку и наш ресурс запоминает в спец. хранилище(кеш-хранилише). Запоминает по определенному ключу.



**Кеширование на сервере** не является нарушением принципа отсутствия состояния в REST. В контексте REST, “отсутствие состояния” означает, что каждый запрос от клиента к серверу должен содержать всю необходимую информацию для обработки этого запроса. Сервер не должен хранить информацию о предыдущих запросах от этого клиента.

Кеширование на сервере относится к хранению ответов на запросы для их повторного использования, что уменьшает нагрузку на сервер и увеличивает производительность. Это не связано с хранением информации о состоянии клиента между запросами.

Важно отметить, что кеширование на сервере должно быть прозрачным для клиента. Клиент не должен знать или полагаться на то, что определенный ответ был получен из кеша. Каждый запрос, который клиент отправляет на сервер, должен быть полным и самодостаточным.

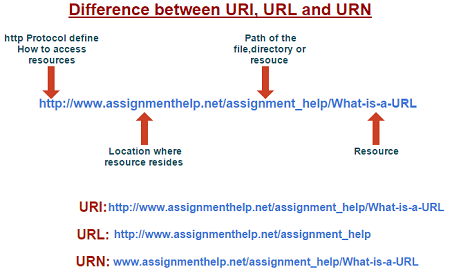
1. Понятия URI, URL, URN.

**URI: Uniform Resource Identifier** – унифицированный идентификатор ресурса (документ, изображение, файл, служба, электронная почта,…).

**URL: Uniform Resource Location** - унифицированный локатор ресурса , содержащий местонахождение ресурса и способ обращения (протокол) к ресурса, описывает множество URI.

**URN: Uniform Resource Name** - унифицированное имя ресурса – URI, имя ресурса, не содержащее месторасположение и способ доступа к ресурсу. В будущем URN должен заменить URL (для решения проблем с перемещением ресурсов в Internet).

**URI, URL, URN** –рекомендуется использовать термин URI



1. HTML: перечень тегов, генерирующих HTTP-запросы.

В HTML есть несколько тегов, которые могут генерировать HTTP-запросы:

* <img src="url">: Генерирует HTTP GET запрос для получения изображения.
* <iframe src="url">: Генерирует HTTP GET запрос для загрузки встроенной веб-страницы.
* <video src="url">: Генерирует HTTP GET запрос для загрузки видео.
* <a href="url">: Генерирует HTTP GET запрос при клике по ссылке.
* <input type="submit">: Может генерировать HTTP POST или GET запрос при отправке формы.
* <link href="url">: Генерирует HTTP GET запрос для загрузки стилей.
* <script src="url">: Генерирует HTTP GET запрос для загрузки JavaScript файла.
* <form action="url">: Генерирует HTTP POST или GET запрос при отправке формы.
* <audio src="url">: Генерирует HTTP GET запрос для загрузки аудиофайла.
* <source src="url">: Генерирует HTTP GET запрос для загрузки медиафайла.
* <video poster="url">: Генерирует HTTP GET запрос для загрузки постера видео.
* <object data="url">: Генерирует HTTP GET запрос для загрузки ресурса.
* <embed src="url">: Генерирует HTTP GET запрос для загрузки встроенного контента.

Важно отметить, что любой элемент может генерировать HTTP-запросы с помощью AJAX, в зависимости от того, как реализован JavaScript.

1. Основные объекты web-сервера (не менее 6).

HttpContext, HttpRequest, HttpResponse, Filter, Listener, Session, Cache…

1. HTTP-аутентификации: общая схема.

**1. Идентификация** – заявление пользователя о себе.

**2. Аутентификация** – процедура проверки подлинности идентификации пользователя.

**3. Авторизация** - процедура проверки прав аутентифицированного

пользователя.

Всё вместе – **верификация.**

**Общая схема HTTP-аутентификации** включает следующие шаги:

* Сервер отвечает клиенту со статусом 401 (Unauthorized) и предоставляет информацию о порядке авторизации через заголовок **WWW-Authenticate**, содержащий хотя бы один метод авторизации.
* Клиент, который хочет аутентифицироваться, может сделать это, включив в следующий запрос заголовок **Authorization** с требуемыми данными.
* Обычно, клиент отображает пользователю запрос (prompt) пароля, и после получения ответа отправляет запрос (request) с пользовательскими данными в заголовке Authorization.
* Если сервер получает корректные учётные данные, но они не подходят для доступа к данному ресурсу, сервер должен отправить ответ со статус кодом 403 Forbidden.



1. HTTP-аутентификации: отличие BASIC и DIGEST.

**BASIC** и **DIGEST** - это два метода HTTP-аутентификации, которые используются для проверки пользователей, имеющих доступ к веб-ресурсу. Они оба используют заголовки WWW-Authenticate и Authorization для передачи информации об аутентификации между клиентом и сервером.

**BASIC аутентификация:**

**Сервер отвечает:** WWW-Authenticate: Basic realm="Fake Realm" (scheme name + realm).

В BASIC аутентификации, вместе с каждым HTTP запросом вместе с заголовками передается связка логин/пароль (userid:password), которые позволяют однозначно аутентифицировать пользователя.

Она передает учетные данные (логин и пароль) в незашифрованном виде, используя кодировку base64.

Из-за этого, BASIC аутентификация должна использоваться только с HTTPS для обеспечения безопасности.



**DIGEST аутентификация:**

DIGEST аутентификация похожа на BASIC, но вместо связки логин/пароль передается контрольная сумма, вычисленная на основе параметров запроса (метода, URI, имя пользователя, область (realm), иногда тела запроса), логина, пароля и, возможно, еще нескольких дополнительных параметров.

Это делает DIGEST аутентификацию более безопасной, поскольку она включает хэширование (шифрование MD5), поэтому инструменты фрагмента пакетов не могут фрагментировать пароль.

The server sends the following parameters in the www-authenticate header field.

* nonce — It is the server-specified string sent in each 401 response. This should be sent again in the client request to avoid replay attacks.
* qop — (Quality of Protection) The values are ‘auth’(authentication) and ‘auth-int’(authentication with integrity protection).
* opaque — The data specified by the server should be returned by the client in the authorization field in the next request.
* algorithm — specified the hashing algorithm
* stale — a flag to indicate whether the previous request is valid or invalid.
* Domain



1. HTTP-аутентификации: FORMS.

Метод авторизации: Forms

Делаем неавторизованный запрос

Ответ: Форма авторизации

Запрос Post логин и пароль

Ответ 302 Found Set-Cookie с токеном

Далее запросы с токеном



1. JWT-токены. Подпись JWT токена. Refresh/access token. Идемпотентность.

**JSON Web Token (JWT)** — это открытый стандарт (RFC 7519) для создания токенов доступа, основанный на формате JSON.

**Token** -битовая последовательность, построенная по определенному принципу.

**Token-аутентификация -** аутентификация, использующая token для идентификации пользователя или авторизации операции пользователя. Применяется, как правило, для реализации Single Sign-On в распределенных системах.

**Identity-provider** - сервер, генерирующий token.

**Service-provider** – сервер, предоставляющий сервис клиенту.

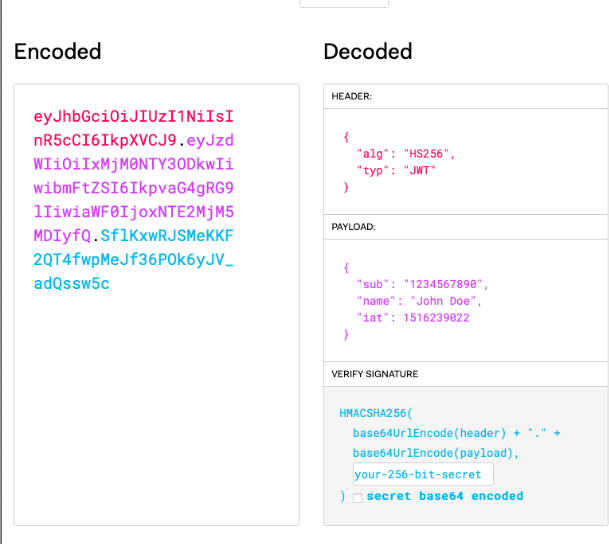


**Token** обычно строится на базе Issuer(эмитент), Audience (аудитория), **Expires On** (время жизни), **Claim** (сведение о пользователе) & **Statements** (дополнительные утверждения о пользователе), HMAC (хеш).

**Подпись JWT-токена –** заголовок + полезная нагрузка (payload)

Вот как выглядит **структура JWT токена:**

* Заголовок (Header): Заголовок обычно состоит из двух частей: типа токена, который является JWT, и алгоритма подписи, такого как HMAC SHA256 или RSA.
* Полезная нагрузка (Payload): Вторая часть токена, полезная нагрузка, содержит утверждения. Утверждения - это заявления о сущности (обычно пользователе) и дополнительные метаданные.
* Подпись (Signature): Чтобы создать подпись, вы должны взять закодированный заголовок, закодированную полезную нагрузку, секрет, применить алгоритм, указанный в заголовке, и затем закодировать этот результат.



**Access/Refresh** – понятно

**Идемпотентность (в http, токены не причём) –** несколько запросов с одинаковыми параметрами и всегда один ответ.

1. Протокол TLS: уровень, назначение, рукопожатие.

**TLS:** 1)обеспечиваетконфиденциальность; 2) обеспечивает целостность (обнаружение подмены); 3) аутентификация узлов (проверка подлинности источника сообщений); последняя версия 1.2, в разработке 1.3 (значительные изменения от 1.2); поверх потокового надежного соединения (для ненадежной передачи есть DTSL);

1. Протокол TLS: X509-сертификаты, назначение.

X.509 certificates are digital documents that represent a user, computer, service, or device. A certificate authority (CA), subordinate CA, or registration authority issues X.509 certificates. The certificates contain the public key of the certificate subject. They don't contain the subject's private key, which must be stored securely. RFC 5280 documents public key certificates, including their fields and extensions. Public key certificates are digitally signed and typically **contain the following information**:

* Information about the certificate subject
* The public key that corresponds to the subject's private key
* Information about the issuing CA
* The supported encryption and/or digital signing algorithms
* Information to determine the revocation and validity status of the certificate

1. Протокол OAuth 2.0: назначение, принципы применения.

**OAuth 2** — это протокол авторизации, предназначенный для организации доступа клиентских приложений к ресурсам, или данным учетных записей, пользователя на другом сервисе. В качестве клиентских приложений выступают веб-сервисы, мобильные и десктопные приложения. В качестве сервисов — mail.ru, GitHub, Bitbucket и др. Протокол используют разработчики сторонних приложений.

**Используемые роли в OAuth 2**

В рамках описываемого протокола выделяются следующие типы ролей:

* владелец (пользователь): авторизует клиентское приложение на доступ к данным своего аккаунта;
* сервер ресурсов/API: здесь располагаются данные пользовательских аккаунтов, а также бизнес-логика авторизации, отвечающая за выдачу новых OAuth-токенов и проверку их подлинности при обращении клиентских приложений к ресурсам. Целесообразно объединять эти роли, так как физически это один сервис;
* клиентское приложение: собственно сервис, которому пользователь делегирует права доступа к своим данным на сервере ресурсов. Пользователь должен авторизовать приложение, а со стороны сервера API оно должно получить подтверждение в виде ключа (токена) доступа.



* Пользователь перенаправляется на страницу авторизации, где у него запрашиваются разрешения для приложения на работу с данными его аккаунта.
* После предоставления необходимых разрешений пользователь попадает на callback URL — адрес, указанный при регистрации приложения, предназначенный для завершения авторизации. При этом происходит подстановка кода авторизации в GET-параметры адреса.
* Сервер клиентского приложения формирует POST-запрос к серверу авторизации API с кодом авторизации в качестве параметра.
* Сервер авторизации проверяет код и возвращает приложению токен доступа (access token).
* Используя токен, приложение авторизуется на сервере API и получает доступ к запрашиваемым пользовательским ресурсам.



1. Протокол WebDAV: назначение, принцип применения.

**Web Distributed Authoring and Versioning** – расширение протокола HTTP/HTTPS, поддерживающее совместную работу по управление файлами на удаленных web-северах; применяется для создания сетевой файловой системы; в системах документооборота (document management system).

**Протокол webdav** - позволяет работать с удаленным хранилищем как с файловой системой.

**WebDAV -** это альтернатива FTP, SMB.

**WebDAV поддерживает**: OS X (Apple) iDisk, Яндекс.Диск, Box.net, Google Drive, Amazon Microsoft IIS, Apache HTTP Server, Dropbox.

**WebDAV предоставляет семь команд:**

PROPFIND — получение свойств объекта на сервере в формате XML;

PROPPATCH — изменение свойств объекта;

MKCOL — создать папку на сервере;

COPY — копирование на стороне сервера;

MOVE — перемещение на стороне сервера;

LOCK — заблокировать объект;

UNLOCK — снять блокировку с объекта.

Таким образом, WebDAV позволяет изменять свойства хранящихся на сервере объектов, выполнять поиск с учетом свойств, блокировать объект (в нашем случае — файл) для организации возможности его редактирования только одним пользователем в распределенной среде, в которой доступ могут иметь много юзеров, управлять версиями файлов (посредством унаследованных команд check -in, -out), а также производить расширенный контроль доступа к файлам на основе списков.

Кроме того, WebDAV поддерживает **унаследованные команды**: GET — для скачивания файла, PUT — для заливки на сервер и DELETE — для удаления серверного объекта.

1. Архитектура браузера, перечень и назначение основных компонентов.

**Браузер** - прикладное программное средство, предназначенное:

- формирования и выполнения http-запросов;

- получения и обработки http-ответов;

- отображения компьютерных файлов;

- интерпретации js-файлов.

**Браузерное программирование** – разработка приложений, работающих в рамках браузера.



**User Interface** –модуль пользовательского интерфейса, позволяющий пользователю управлять работой браузера (вводить URI, движение вперед/назад по истории, закладки и пр.)

**Browser Engine** – модуль управления браузером: управление закладками, скачивание, проверка орфографии, поиск на странице, …).

**Render Engine** – модуль отображения контента: WebKit, Blink, Gecko. Стандарты HTML, CSS, XML, DOM

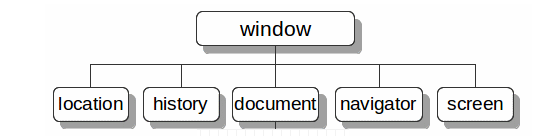
**Data Persistence** – модуль обеспечивающий работу с хранилищем: Local storage, Session Storage, Cookies, WebSQL, IndexedDB, File System, AppCache, Service Workers.

**JS Engine** – модуль-интерпретатор JavaScript: V8,Gecko, SpiderMonkey, Rhino, Tamarin, Chakra, Carakan. Стандарт JS и Browser API.

**Networking** – модуль взаимодействия с сетью.

программисту браузер представляется в виде модели **Browser Object Model**(BOM), модели **Document Object Model** (DOM) и **Browser API**.

**Browser Object Model:**

****

**BOM предоставляет:**

Console, Alert, Prompt, document, localStorage, location (href, pathname, host, protocol, hostname), history (length, back, go, forward), name, screenX/screenY, print, open, setTimeout, setInterval, navigator (platform, appName, language).

1. Назначение и возможности IndexDB API.

**IndexedDB** – это встроенная NoSQL база данных, более мощная, чем localStorage.

* Хранит практически любые значения по ключам, несколько типов ключей.
* Формат данных JSON.
* Поддерживает транзакции для надёжности.
* Поддерживает курсоры.
* Использует события DOM.
* Поддерживает запросы в диапазоне ключей и индексы.
* Позволяет хранить больше данных, чем localStorage.

Для традиционных клиент-серверных приложений эта мощность обычно чрезмерна. IndexedDB предназначена для оффлайн приложений, можно совмещать с ServiceWorkers и другими технологиями.

Открытие (создание) БД:



После этого вызова необходимо назначить обработчик событий для объекта openRequest:

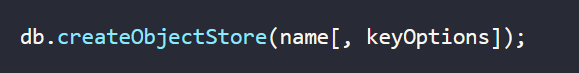
* success: база данных готова к работе, готов «объект базы данных» openRequest.result, его следует использовать для дальнейших вызовов.
* error: не удалось открыть базу данных.
* upgradeneeded: база открыта, но её схема устарела



Удаление Базы данных:



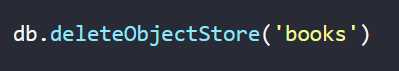
Создание хранилища:



keyOptions – это необязательный объект с одним или двумя свойствами:

* keyPath – путь к свойству объекта, которое IndexedDB будет использовать в качестве ключа, например id.
* autoIncrement – если true, то ключ будет формироваться автоматически для новых объектов, как постоянно увеличивающееся число.

Удаление хранилища:



Если при создании хранилища не указать keyOptions, то нам потребуется явно указать ключ позже, при сохранении объекта.

Пример:



\*ID -ключ в данном случае.

Добавление данных в таблицу/транзакции:



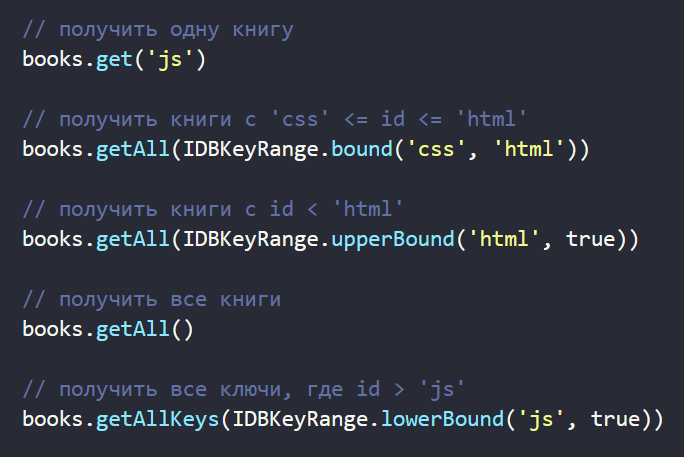
Создание транзакции:



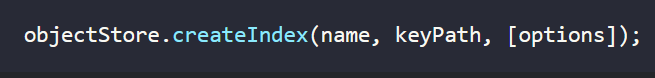
type – тип транзакции, один из:

* readonly – только чтение, по умолчанию.
* readwrite – только чтение и запись данных, создание/удаление самих хранилищ объектов недоступно.

Получение из базы:



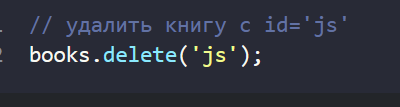
Создание индекса:



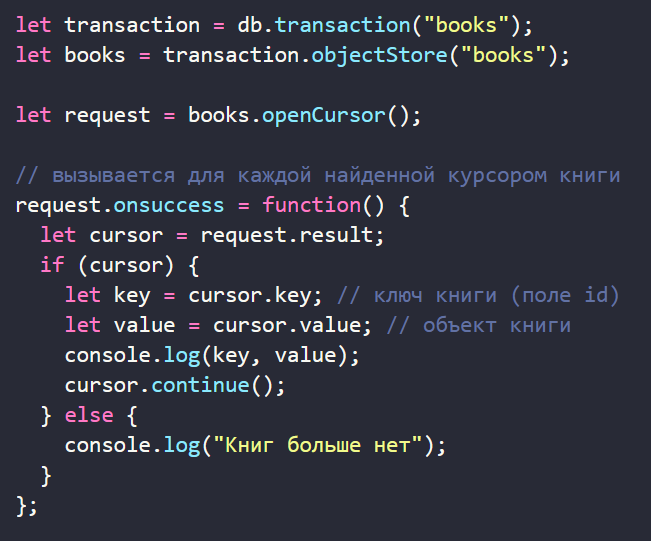
Поиск по индексу:



Удаление данных из БД:



Работа с курсорами:



Хранилище объектов можно создавать/изменять только при обновлении версии базы данных в обработчике upgradeneeded.

Это техническое ограничение. Вне обработчика мы сможем добавлять/удалять/обновлять данные, но хранилища объектов могут быть созданы/удалены/изменены только во время обновления версии базы данных.

Транзакции автокомплитятся после выполнения всех микротасок => нет необходимости ими управлять, но внутри нельзя использовать асинхронные операции (например отправка запроса через fetch или установка таймера setTimeout).

Все операции выполняются синхронно.

Есть обёртка для промисов, можно писать так:



1. WebRTC: назначение, перечень протоколов.

**WebRTC (Web Real-Time Communication** - коммуникация в режиме реального времени) - это API (Application Programming Interface - программный интерфейс приложения) и протокол. Протокол WebRTC - это набор правил, позволяющий двум агентам WebRTC (браузерам) вести двунаправленную (bi-directional) безопасную коммуникацию в реальном времени. WebRTC API позволяет разработчикам использовать протокол WebRTC.

При использовании RTC не нужен сервер! Все данные передаются между двумя клиентами и обрабатываются браузерами.

RTC предоставляет возможность общения в реальном времени.

RTC знает IP-адреса.

**Плюсы**: простое решение, 60 строк кода == общение между двумя клиентами, P2P, сервер практически не нужен т.к. браузеры договариваются напрямую.

**Минусы:** нужен STUN, только звонки (запись или телефонная сеть нельзя).

**Протокол WebRTC** - это собрание других технологий и протоколов.

В процессе установки соединения с помощью WebRTC можно выделить 4 этапа:

* Сигнализация (signalling)
* Подключение (установка соединения) (connection)
* Безопасность (securing)
* Коммуникация (взаимодействие) (communication)
* Переходы между этапами происходят последовательно. Обязательным условием для начала следующего этапа является успешное завершение предыдущего.

Для того, чтобы WebRTC работал, мы должны иметь возможность идентифицировать или определять местонахождение друг друга по сети. Это часто называют обнаружением одноранговых узлов. Это просто причудливый способ сказать — как мне найти кого-то, с кем можно поговорить или обменяться данными? Механизмы обнаружения одноранговых узлов не определены в WebRTC, хотя процесс может быть таким же простым, как совместное использование URL-адреса, который одноранговые узлы могут использовать для общения.

Было бы идеально, если бы две машины могли быть напрямую адресованы с использованием их общедоступных IPv4-адресов. Это невозможно из-за исчерпания адресов IPv4 и того факта, что нам также нужны брандмауэры для контроля доступа через порты и IP-адреса на наших машинах.

**NAT (Network Address Translation)** используется для присвоения устройству общедоступного IP-адреса. Сетевые маршрутизаторы будут иметь публичные IP-адреса, где каждое подключенное к ним устройство будет иметь частные IP-адреса. Устройствам не потребуется уникальный общедоступный IP-адрес, так как их частные IP-адреса преобразуются в общедоступный IP-адрес маршрутизатора с уникальным портом.

WebRTC использует технологии Interactive Connectivity Establishment (ICE) для преодоления сложностей работы в реальном сетевом мире. Чтобы это произошло, приложение должно передавать URL-адреса сервера **ICE** в RTCPeerConnections.

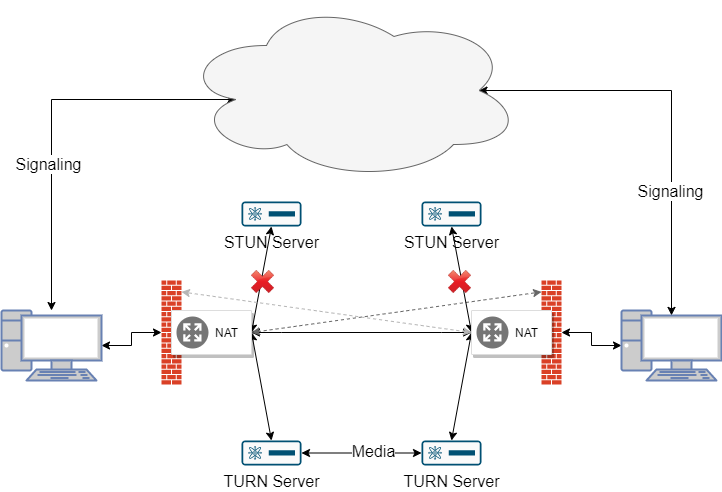
ICE сначала попытается установить соединение, используя адрес хоста, полученный из операционной системы и сети устройства, если это не удастся, ICE получит внешний адрес с помощью **STUN-сервера**, где в случае неудачи трафик направляется через сервер-ретранслятор **TURN**.

**STUN** (Session Traversal Utilities for NAT) — это протокол, который используется для обнаружения публичных адресов и определяет любые ограничения в вашем маршрутизаторе, которые препятствуют прямому соединению с одноранговым узлом. Клиенты получают свои публичные адреса по запросу от STUN-серверов. Это происходит независимо от того, доступен ли клиент за NAT маршрутизатора. Другими словами, приложение использует STUN-сервер для обнаружения своего публичного адреса.

Большинство WebRTC-вызовов успешно устанавливают соединение с помощью STUN-серверов.

**TURN** (Traversal Using Relays around NAT) предназначен для обхода ограничения симметричного NAT путем открытия соединения с сервером TURN и ретрансляции всей информации через этот сервер. Требуется соединение с TURN-сервером, который укажет всем одноранговым узлам посылать пакеты на сервер, которые затем будут пересылаться запрашивающей стороне. Будут некоторые накладные расходы, поэтому он используется только тогда, когда нет других альтернатив.

TURN используется для ретрансляции потоковой передачи аудио/видео/данных между одноранговыми узлами, а не для передачи сигнальных данных!



На этой схеме полностью показано, как работает WebRTC, где TURN запускается после сбоя STUN, каждый пир прибегает к использованию TURN-сервера.

Другими словами: STUN-сервер используется для получения внешнего сетевого адреса, а TURN-сервер используется для ретрансляции трафика в случае сбоя прямого (однорангового) соединения.

То есть используются протоколы:

* ICE (Interactive Connectivity Establishment): Этот протокол позволяет вашему веб-браузеру соединяться с узлами.
* STUN (Session Traversal Utilities for NAT): Этот протокол используется для нахождения и определения вашего публичного адреса и любых ограничений в вашем маршрутизаторе, которые препятствуют прямому соединению с узлом.
* NAT (Network Address Translation): NAT используется для того, чтобы дать вашему устройству публичный IP-адрес.
* TURN (Traversal Using Relays around NAT): TURN предназначен для обхода ограничения “Симметричный NAT” путём открытия соединения с TURN сервером и ретрансляции всей информации через этот сервер.

1. Документ RFC.

Рабочее предложение (англ. **Request for Comments, RFC**) — документ из серии пронумерованных информационных документов Интернета, содержащих технические спецификации и стандарты, широко применяемые во всемирной сети. Название «Request for Comments» ещё можно перевести как «заявка (запрос) на отзывы» или «тема для обсуждения». В настоящее время первичной публикацией документов RFC занимается **IETF** под эгидой открытой организации Общество Интернета (англ. **Internet Society, ISOC**). Правами на RFC обладает именно Общество Интернета.

**Содержимое RFC:**

Несмотря на название, запросы на отзывы RFC сейчас рассматриваются как стандарты Интернета (а рабочие версии стандартов обычно называют драфтами, от англ. draft — черновой проект). Согласно RFC 2026, жизненный цикл стандарта выглядит следующим образом:

* Выносится на **всеобщее рассмотрение** интернет-проект (Internet Draft). Проекты не имеют официального статуса и удаляются из базы через шесть месяцев после последнего изменения.
* Если проект стандарта оказывается достаточно удачным и непротиворечивым, он получает статус **предложенного стандарта** (Proposed Standard), и свой номер RFC. Наличие программной реализации стандарта желательно, но не обязательно.
* Следующая стадия — проект стандарта (Draft Standard) — означает, что предложенный **стандарт принят** сообществом, в частности, существуют две независимые по коду совместимые реализации разных команд разработчиков. В проекты стандартов ещё могут вноситься мелкие правки, но они считаются достаточно стабильными и рекомендуются для реализации.
* Высший уровень — **стандарт Интернета** (Internet Standard). Это спецификации с большим успешным опытом применения и зрелой формулировкой. Параллельно с нумерацией RFC они имеют свою собственную нумерацию STD. Список стандартов имеется в документе STD 1 (сейчас это RFC 5000, но нумерация может измениться). Из более чем трёх тысяч RFC этого уровня достигли только несколько десятков.
* Многие старые RFC замещены более новыми версиями под новыми номерами или вышли из употребления. Такие документы получают статус **исторических (Historic)**

1. Стандарты ISO. Какой стандарт в Беларуси.

Стандарты **ISO (International Organization for Standardization)** - это международные стандарты, которые разрабатываются для обеспечения качества, безопасности и эффективности продуктов, услуг и систем.

В Беларуси стандарты ISO представлены **Государственным комитетом по стандартизации (BELST)**. Беларусь является полноправным членом ISO с 1993 года и участвует в 168 технических комитетах и подкомитетах по стандартизации.

Государственные стандарты Республики Беларусь имеют аббревиатуру **“STB”.** Также применяются стандарты с аббревиатурой **“GOST”,** которые используются Евро-Азиатским советом по стандартизации, метрологии и сертификации в рамках СНГ. Уровень гармонизации государственных стандартов, принятых в Беларуси, с международными достигает 80%

1. Сокеты. Основные понятия. Принцип работы.

* У каждого компьютера в сети есть IP-адрес, даже если это просто локальная сеть.
* IP-адрес — это четыре числа от 0 до 255, разделённые точками, например 77.88.55.88 (это адрес сервера Яндекса).
* С помощью этих адресов компьютеры знают, куда направить свои запросы и ответы.
* Когда один компьютер соединяется с другим, они это делают через сетевой порт. Можно сказать, что порт — это номер соединения.
* Сетевые порты в компьютере нумеруются от 1 до 65535, а компьютер сам следит за тем, как распределяются эти номера.
* С помощью портов компьютер понимает, какие данные предназначены какой программе.
* Некоторые программы и соединения всегда используют один и тот же порт, а другие получают его случайным образом.

**Сокет** — это виртуальная конструкция из IP-адреса и номера порта. Её придумали для того, чтобы разработчикам было проще писать код, а программы могли передавать данные друг другу даже в пределах одного компьютера.

Совокупность IP-адреса и номера порта называется **сокетом**. Сокет однозначно идентифицирует прикладной процесс в сети TCP/IP. Следует помнить, что одни и те же номера портов могут быть использованы как для протокола UDP, так и для протокола TCP.

Смысл в том, чтобы программист работал не с IP-адресами и портами, разбираясь в тонкостях работы протоколов, а использовал что-то попроще. В итоге получается так:

* программист пишет в программе, что он хочет сделать новый сокет;
* указывает для него IP-адрес, если это необходимо;
* программа собирает это в виртуальную конструкцию, и получается сокет;
* после этого программист может отправлять данные просто в сокет и принимать их оттуда, а компьютер берёт на себя все вопросы по передаче данных.

Сокеты используют для двух вещей:

* для передачи данных по сети;
* и для связи между приложениями.

**Сокет находится на транспортном уровне модели ISO.**

1. Протоколы TCP и UDP. Основные свойства. 7 уровней модели ISO/OSI. 4 уровня стека протоколов TCP/IP. Соответствия и различия. Что находится на каждом уровне. NAT. FTP. VPN. PROXY.

**TCP (англ. Transmission Control Protocol** — протокол управления передачей) — один из основных протоколов передачи данных интернета. Предназначен для управления передачей данных интернета. Пакеты в TCP называются сегментами.

**Ненадежные протоколы** - это протоколы, которые не гарантируют доставку сообщений. Они могут быть использованы в условиях, когда потеря сообщения не сильно повлияет на работу приложения. Ненадежные протоколы обычно используются для передачи данных в реальном времени, например, видео- и аудиопотоков.

Примеры ненадежных протоколов: **UDP(**User Datagram Protocol ), ICMP(Internet Control Message Protocol).

1) отсутствие механизмов обеспечения надежности;

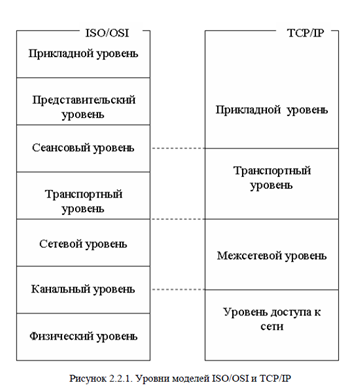
2) отсутствие гарантий доставки;

3) отсутствие обработки соединений;

4) отсутствие буферизации;

5) отсутствие фрагментации;

6) отсутствие пересчёта контрольной суммы.



**Прикладной уровень:** Основное назначения уровня: определить способы взаимодействия пользователей с системой (определить интерфейс).

**Представительский уровень.** На представительский уровне определяется формат данных, используемых приложениями. Процедуры этого уровня описывают способы шифрования, сжатия и преобразования наборов символов данных.

**Сеансовый уровень.** Сеансовый уровень определят способы установки и разрыва соединений (называемых сеансами) двух приложений, работающих в сети.

Следует отметить, что сеансовый уровень - это точка взаимодействия программ и компьютерной сети.

**Транспортный уровень.** Основным назначением процедур транспортного уровня является подготовка и доставка пакетов данных между конечными точками без ошибок и в правильной последовательности. Процедуры транспортного уровня формируют файлы для сеансового уровня из пакетов, полученных от сетевого уровня.

**Сетевой уровень.** Сетевой уровень определяет методы адресации и маршрутизации компьютеров в сети.

**Канальный уровень**. На канальном уровне модели рассматривается два подуровня: подуровень управления доступом к среде передачи данных и подуровень управления логическим каналом.

**Физический уровень.** Физический уровень определяет свойства среды передачи данных (коаксиальный кабель, витая пара, оптоволоконный канал и т.п.) и способы ее соединения с сетевыми адаптерами

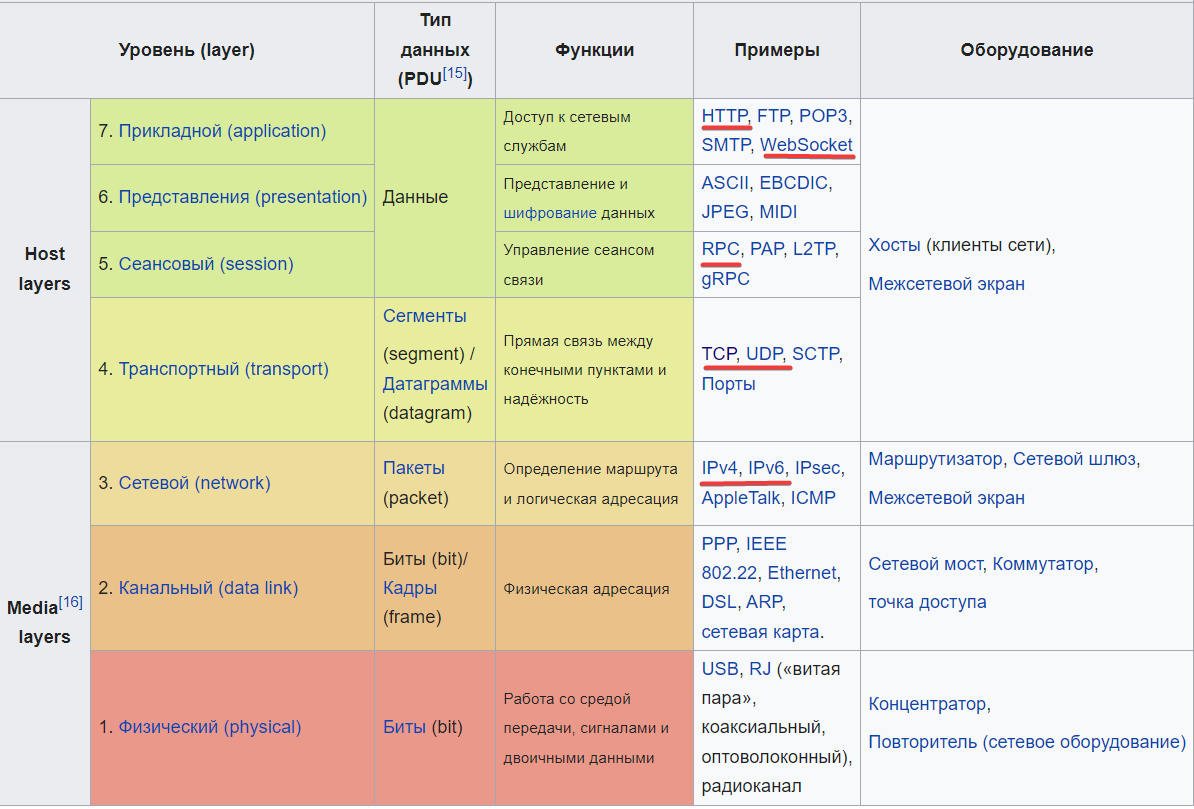
**Модель TCP/IP состоит из 4 уровней:**

1. **Уровень доступа к сети** (Network Access Layer): Определяет физические способы передачи данных по сети. Примеры протоколов: Ethernet, Wi-Fi, Token Ring.

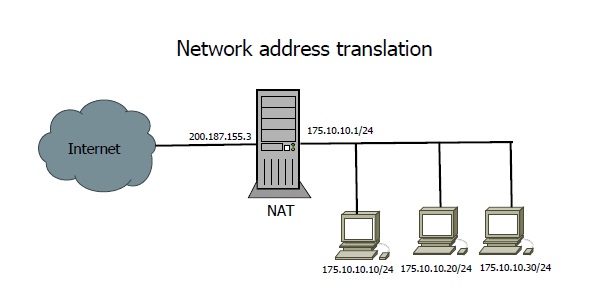
2. **Интернет-уровень (межсетевой)** (Internet Layer): Обеспечивает маршрутизацию пакетов между сетями. Примеры протоколов: IP (Internet Protocol), ICMP (Internet Control Message Protocol), ARP (Address Resolution Protocol), RARP (Reverse ARP).

3. **Транспортный уровень** (Transport Layer): Обеспечивает передачу данных между приложениями на разных узлах сети. Примеры протоколов: TCP, UDP.

4. **Прикладной уровень** (Application Layer): Обеспечивает взаимодействие между приложениями и пользователями. Примеры протоколов: HTTP, FTP, SMTP, DNS, DHCP.

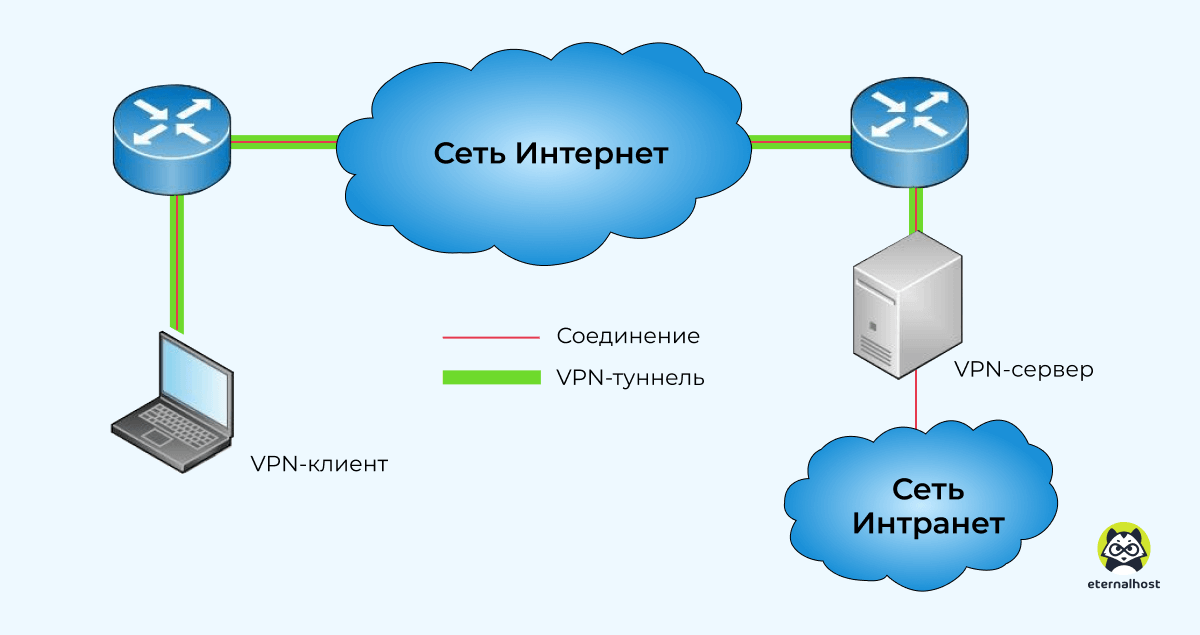


**NAT (от англ. Network Address Translation** — «преобразование сетевых адресов») — это механизм в сетях TCP/IP, позволяющий преобразовывать IP-адреса транзитных пакетов.

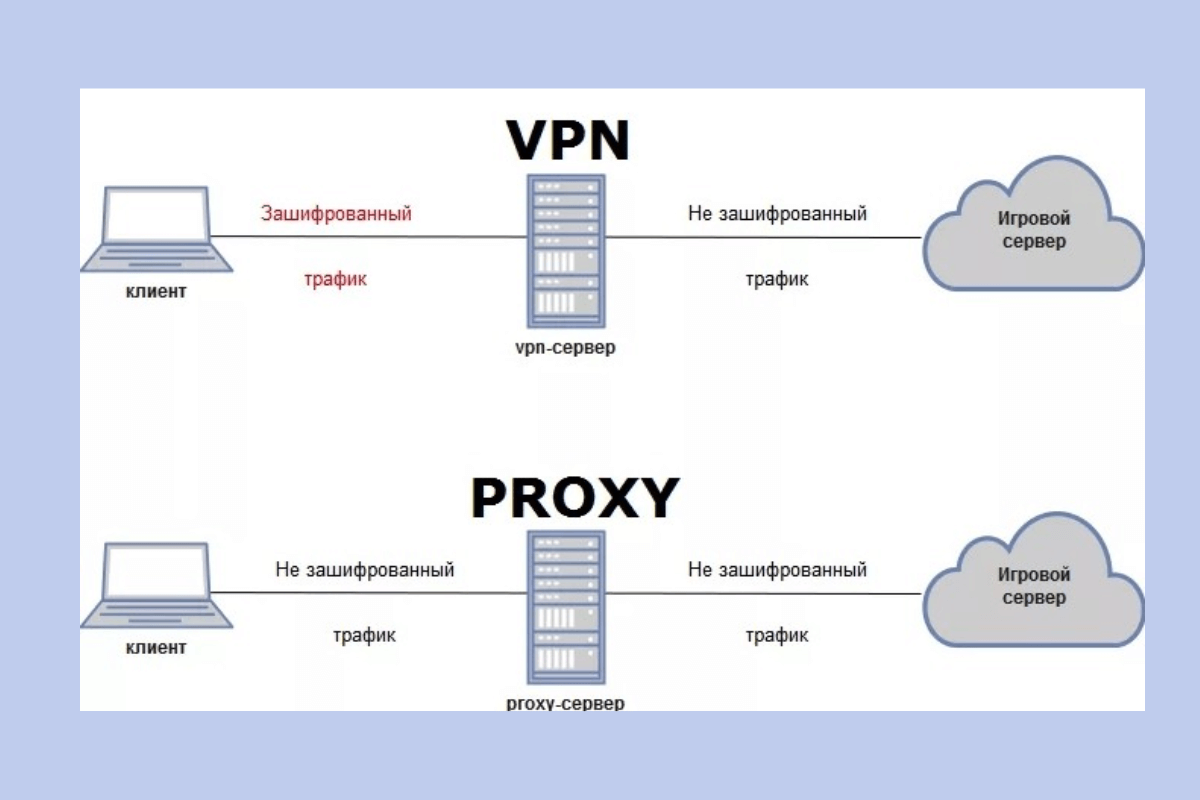


**FTP (File Transfer Protocol)** используется для связи и передачи файлов между компьютерами в сети TCP/IP (Transmission Control Protocol/Internet Protocol), также известной как Интернет. Пользователи, которым предоставлен доступ, могут получать и передавать файлы на сервере протокола передачи файлов.

**VPN (англ. virtual private network** — «виртуальная частная сеть») — обобщённое название технологий, позволяющих обеспечить одно или несколько сетевых соединений поверх чьей-либо другой сети.



**Прокси-сервер (от англ. proxy** — представитель, уполномоченный; часто просто прокси, сервер-посредник) — промежуточный сервер (комплекс программ) в компьютерных сетях, выполняющий роль посредника между пользователем и целевым сервером (при этом о посредничестве могут как знать, так и не знать обе стороны), позволяющий клиентам как выполнять косвенные запросы (принимая и передавая их через прокси-сервер) к другим сетевым службам, так и получать ответы. Сначала клиент подключается к прокси-серверу и запрашивает какой-либо ресурс (например e-mail), расположенный на другом сервере. Затем прокси-сервер либо подключается к указанному серверу и получает ресурс у него, либо возвращает ресурс из собственного кэша (в случаях, если прокси имеет свой кэш).



1. Понятия сессии.

**Сессия (Session):** серверный объект, хранящий информацию о соединении с клиентом, создается при первом обращении время жизни: **timeout** (системный параметр, обычно равен 10 – 30 минутам) – максимальное время между запросами клиента. Если **timeout** превышен, то Session разрушается и при следующем запросе создается новый экземпляр. Каждая сессия имеет собственный идентификатор (**Session ID**, 16 или более байт (128 бит)). Каждый Requestпринадлежит, какой-то сессии (имеет ссылку на объект Session или содержит Session ID). Обычно объект Session предоставляет приложению возможность хранить данные в формате ключ/значение.

Необходима для последовательных запросов, хранения промежуточных данных, e.g. не спрашивать у клиента логин и пароль при каждом запросе.

Относится к **сеансовому уровню OSI/ISO**: на этом уровне выполняется шифрование, и генерируется сеансовый ключ (Session ID), который актуален в течение всей сессии.

Сессия = сеанс

Параметр **session-timeout**(определеяет максимальный интервал времени между запросами в рамках одной сессии, если этот интервал превышен, то сессия разрывается)

Для того, что определить начало серии запросов, относящихся к одной сессии, используется специальный механизм cookie.

* Сервер создает сессию, когда на него приходит первый request:
* без заголовка Cookie;
* в заголовке Cookie не записан Session ID;
* в заголовке Cookie записан неправильный или несуществующий Session ID.
* Сервер создаёт объект сессии, генерирует Session ID, и отправляет клиенту заголовок Set-Cookie со значением Session ID;
* Клиент, получивший заголовок Set-Cookie, обязан запомнить значение Session ID;
* Во все последующие запросы клиент будет добавлять заголовок Cookie со значением Session ID;
* Сам сервер никогда не отправляет заголовок Cookie, он только обрабатывает их от клиента.

1. Основные объекты web-сервера: контекст приложения.

**Конфигурационный файл web-приложения**: файл содержащий системные параметры приложения, обычно в XML-формате, служит основой для создания контекста web-приложения.

**Контекст web-приложения**: серверный объект, предназначенный для хранения информации об одом web-приложении, общий для всех файлов. Как правило, формируется сразу при загрузке web-сервера, основные данные (параметры приложения) копируются из конфигурационного файла приложения, общий для всех сессий приложения, Обычно контекст предоставляет возможность хранить данные в формате ключ/значение.

**Резидентный объект** - тот объект, который находится постоянно в оперативной памяти

Сервер загружается и **создает контекст (резидентный объект).** Значения берет из переменных окружения и конфигураторов

**Хранилище** - эта память, которая будет хранить информацию до выключения сервера

1. DI (dependency injection – внедрение зависимостей). Жизненный цикл. Определение.

**Dependency injection (DI)** или внедрение зависимостей представляет механизм, который позволяет сделать компоненты программы слабосвязанными, а всю программу в целом более гибкой, более адаптируемой и расширяемой.

**Внедрение зависимости** (англ. Dependency injection, DI) — процесс предоставления внешней зависимости программному компоненту.

В центре подобного механизма находится понятие зависимость - некоторая сущность, от которой зависит другая сущность.

Используя различные методы внедрения зависимостей, можно управлять жизненным циклом создаваемых сервисов. Сервисы, которые создаются механизмом Depedency Injection, могут представлять один из следующих типов (так в ASP .Net core):

* Transient: при каждом обращении к сервису создается новый объект сервиса. В течение одного запроса может быть несколько обращений к сервису, соответственно при каждом обращении будет создаваться новый объект. Подобная модель жизненного цикла наиболее подходит для легковесных сервисов, которые не хранят данных о состоянии
* Scoped: для каждого запроса создается свой объект сервиса. То есть если в течение одного запроса есть несколько обращений к одному сервису, то при всех этих обращениях будет использоваться один и тот же объект сервиса.
* Singleton: объект сервиса создается при первом обращении к нему, все последующие запросы используют один и тот же ранее созданный объект сервиса.

1. Промежуточные конвейеры обработки (middleware, filter…).

**Фильтр или middleware** - конвейер обработки запроса и ответа. На прямом пути обрабатываются запрос, на обратном пути обрабатывается ответ. Обычно создает app, а в него цепочки

Контекст, лисенеры, фильтры, сессия создаются после первого запроса серии.

**Фильтр (Filter)**: серверный объект – препроцессор запроса, предназначен для предварительной обработки объекта Request. К одному ресурсу может быть построена цепочка фильтров, последний в цепочке – ресурс. Фильтр может прервать цепочку и сам сформировать ответ клиенту. Один и тот же фильтр может быть применен к нескольким ресурсам. В качестве параметров фильтр получает объекты Request и Response, которые от передает дальше по цепочке или обрывает цепочку и заполняет объект Response.

1. REST. REST API. Определение, применение, особенности.

**REST**(REpresentational State Transfer) — это архитектура, т.е. принципы построения распределенных гипермедиа систем, того что другими словами называется World Wide Web, включая универсальные способы обработки и передачи состояний ресурсов по HTTP.

**Преимущества REST:**

* Отсутствие дополнительных внутренних прослоек, что означает передачу данных в том же виде, что и сами данные. Т.е. данные не оборачиваются в XML, как это делает SOAP и XML-RPC, не используется AMF, как это делает Flash и т.д. Просто отдаются сами данные.
* Каждая единица информации (ресурс) однозначно определяется URL — это значит, что URL по сути является первичным ключом для единицы данных. Причем совершенно не имеет значения, в каком формате находятся данные по адресу — это может быть и HTML, и jpeg, и документ Microsoft Word.
* Как происходит управление информацией ресурса — это целиком и полностью основывается на протоколе передачи данных. Наиболее распространенный протокол конечно же HTTP. Для HTTP действие над данными задается с помощью методов : GET (получить), PUT (добавить, заменить), POST (добавить, изменить, удалить), DELETE (удалить). Таким образом, действия CRUD (Create-Read-Update-Delete) могут выполняться как со всеми 4-мя методами, так и только с помощью GET и POST.

**Что такое RESTful:**

Чтобы распределенная система считалась сконструированной по REST архитектуре (Restful), необходимо, чтобы она удовлетворяла следующим критериям:

* Client-Server. Система должна быть разделена на клиентов и на серверов. Разделение интерфейсов означает, что, например, клиенты не связаны с хранением данных, которое остается внутри каждого сервера, так что мобильность кода клиента улучшается. Серверы не связаны с интерфейсом пользователя или состоянием, так что серверы могут быть проще и масштабируемы. Серверы и клиенты могут быть заменяемы и разрабатываться независимо, пока интерфейс не изменяется.
* Stateless. Сервер не должен хранить какой-либо информации о клиентах. В запросе должна храниться вся необходимая информация для обработки запроса и если необходимо, идентификации клиента.
* Cache․ Каждый ответ должен быть отмечен является ли он кэшируемым или нет, для предотвращения повторного использования клиентами устаревших или некорректных данных в ответ на дальнейшие запросы.
* Uniform Interface. Единый интерфейс определяет интерфейс между клиентами и серверами. Это упрощает и отделяет архитектуру, которая позволяет каждой части развиваться самостоятельно. Ресурсом является всё, чему можно дать имя, манипуляция с ресурсами через представления, само-документируемые сообщения, статус ресурса передаётся через body, query string, headers, resource name.
* Layered System. В REST допускается разделить систему на иерархию слоев но с условием, что каждый компонент может видеть компоненты только непосредственно следующего слоя. Например, если вы вызывайте службу PayPal а он в свою очередь вызывает службу Visa, вы о вызове службы Visa ничего не должны знать.

**Важно !** Сама архитектура REST не привязана к конкретным технологиям и протоколам, но в реалиях современного Веб, построение RESTful API почти всегда подразумевает использование HTTP и каких-либо распространенных форматов представления ресурсов, например JSON, или, менее популярного сегодня, XML.

С точки зрения RESTful-сервиса, операция (или вызов сервиса) **идемпотентна** тогда, когда клиенты могут делать один и тот же вызов неоднократно при одном и том же результате на сервере. Другими словами, создание большого количества идентичных запросов имеет такой же эффект, как и один запрос. Заметьте, что в то время, как идемпотентные операции производят один и тот же результат на сервере, ответ сам по себе может не быть тем же самым (например, состояние ресурса может измениться между запросами).

Если обобщить, то restful web api – это приложение, где есть клиент и сервер, которые независимы друг от друга, они общаются через http с соблюдением правил выше, все состояния сохраняется на клиенте. (ASP .Net CORE MVC – цэ не RESTFul).

1. SOAP. Определение, применение, особенности.

**SOAP** — это протокол, по которому веб-сервисы взаимодействуют друг с другом или с клиентами. Название происходит от сокращения Simple Object Access Protocol («простой протокол доступа к объектам»). SOAP API — это веб-сервис, использующий протокол SOAP для обмена сообщениями между серверами и клиентами. При этом сообщения должны быть написаны на языке XML в соответствии со строгими стандартами, иначе сервер вернет ошибку.

Протокол SOAP был представлен в 1998 году и быстро стал одним из главных стандартов веб-служб, когда Microsoft продвигала платформу .NET, приложения которой взаимодействовали с помощью SOAP API. Сейчас протокол и API уступают по популярности архитектурному стилю REST. Но веб-приложения, использующие SOAP API, все еще пользуются спросом, особенно в банковском и телекоммуникационном секторах.

SOAP может использоваться с протоколами SMTP, FTP, HTTP, HTTPS. Чаще всего — с HTTP как с наиболее универсальным: его поддерживают все браузеры и серверы. Корректное SOAP-сообщение состоит из нескольких структурных элементов: Envelope, Header, Body и Fault.

* Envelope («конверт»). Это корневой элемент. Определяет XML-документ как сообщение SOAP с помощью пространства имен xmlns\_soap=»http://www.w3.org/2003/05/soap-envelope/». Если в определении будет указан другой адрес, сервер вернет ошибку.
* Header («заголовок»). Включает в себя атрибуты сообщения, связанные с конкретным приложением (аутентификация, проведение платежей и так далее). В заголовке могут использоваться три атрибута, которые указывают, как принимающая сторона должна обрабатывать сообщение, — mustUnderstand, actor и encodingStyle. Значение mustUnderstand — 1 или 0 — говорит принимающему приложению о том, следует ли распознавать заголовок в обязательном или опциональном порядке. Атрибут actor задает конкретную конечную точку для сообщения. Атрибут encodingStyle устанавливает специфическую кодировку для элемента. По умолчанию SOAP-сообщение не имеет определенной кодировки.
* Body («тело»). Сообщение, которое передает веб-приложение. Может содержать запрос к серверу или ответ от него. (в XML формате!)
* Fault («ошибка»). Опциональный элемент. Передает уведомление об ошибках, если они возникли в ходе обработки сообщения.

**Отличия SOAP от REST**

**SOAP** — протокол, а **REST** — архитектурный стиль, набор правил по написанию кода. REST был представлен в 2000 году. К этому времени недостатки SOAP были очевидны:

* объемные сообщения;
* поддержка только одного формата — XML;
* схема работы по принципу «один запрос — один ответ»;
* смена описания веб-сервиса может нарушить работу клиента.

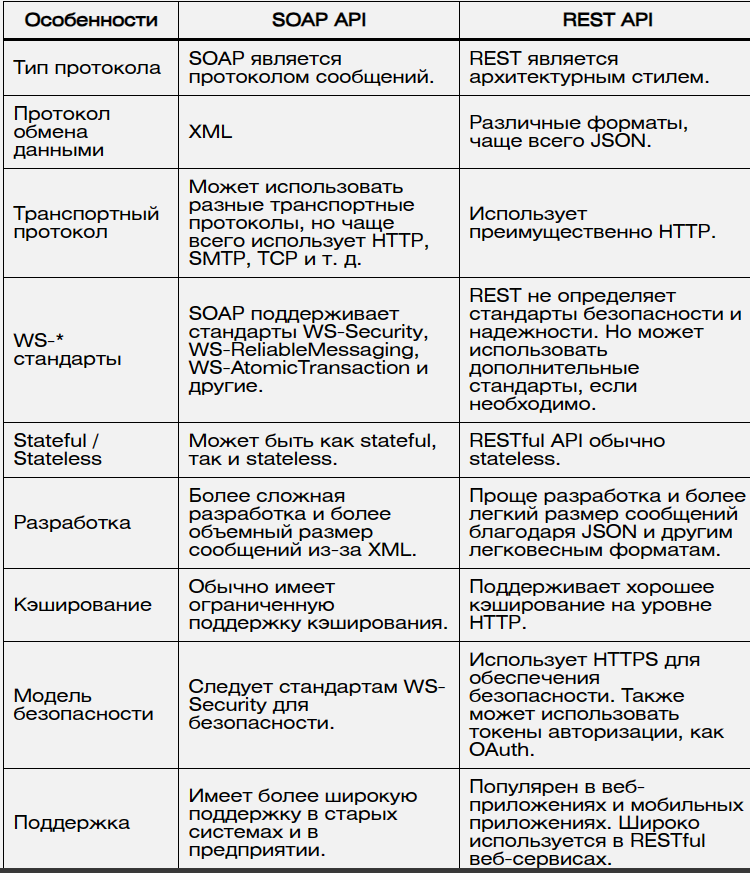
Разработчик стиля REST Рой Филдинг учел недостатки SOAP. REST поддерживает несколько форматов помимо XML: JSON, TXT, CSV, HTML. Вместо создания громоздкой структуры XML-запросов при использовании REST чаще всего можно передать нужный URL. Эти особенности делают стиль REST простым и понятным, а приложения и веб-сервисы, использующие его, отличаются высокой производительностью и легко масштабируются.

Пример простого URL-запроса, возвращающего результаты поиска по ключевому слову DNA («ДНК»), можно посмотреть в международной базе научных статей.

Несмотря на простоту использования, у REST есть ряд **недостатков**, которые отсутствуют у SOAP:

* при использовании REST сложнее обеспечить безопасность конфиденциальных данных;
* трудности с проведением операций, которым необходимо сохранение состояния. Как, например, в случае с корзиной в онлайн-магазине, которая должна сохранять добавленные товары до момента оплаты.

Основные различия SOAP и REST:



1. RPC. Определение, применение, особенности.

**A remote Procedure Call or RPC** is used in distributed computing, and this call occurs when a procedure is executed in a different address space. This protocol is helpful when the process does not know other computer’s network details and the service is requested in a different computer. This is also called a subroutine call or function call. This has a client-server model. The arguments are passed to a remote procedure and the call is sent to the server by the client. RPC can be used in Windows, Apple, and UNIX. The authentication of RPC is DES so that the application can be made secure.

**A Remote Procedure Call (in short RPC)** is a protocol used by operating systems to allow processes to communicate across a network. **RPC allows a process running on one computer to call a procedure on another computer, as if the procedure was running locally**

**When a process makes an RPC call, it sends a message containing the procedure name and any necessary arguments to the remote computer.** The remote computer receives the message, executes the procedure, and returns the results back to the calling process. The calling process then continues execution as if the procedure had been executed locally.

RPC provides a high-level abstraction for communication between processes, allowing developers to write distributed applications without having to worry about the underlying networking details. RPC is often used in client-server architectures, where a client process makes RPC calls to a server process to access its services.

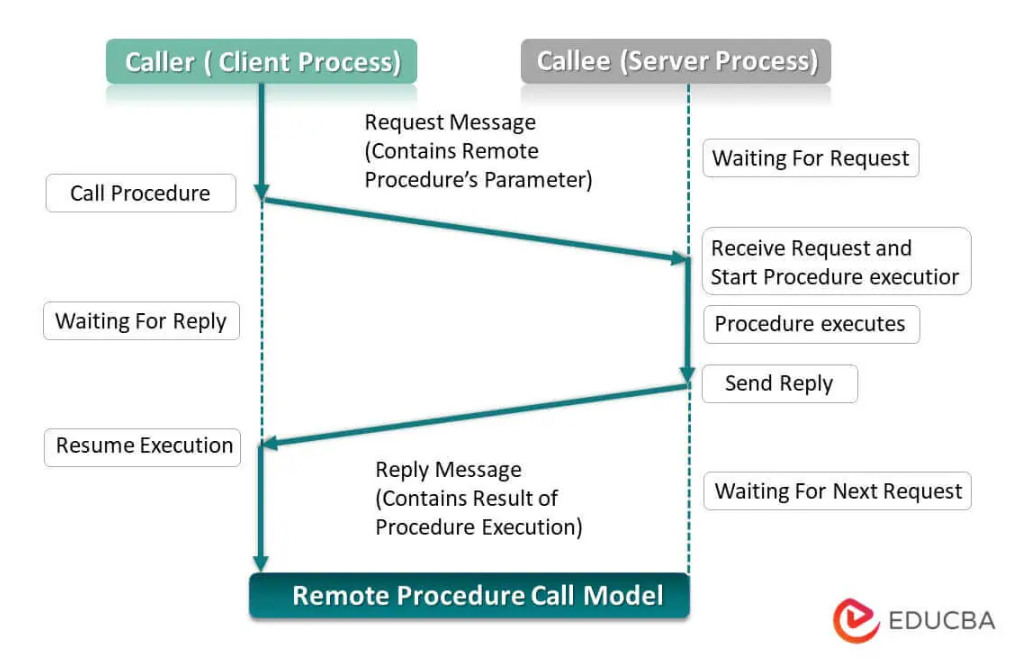
Some common implementations of RPC include Sun RPC (used in Unix-based systems), Microsoft’s DCOM (Distributed Component Object Model), and CORBA (Common Object Request Broker Architecture).

When the program is executed, and a remote procedure call is made, The stub intercepts the remote procedure call request from the client program and forwards it to the client runtime program, which is responsible for handling network communication and sending the request across the network to the server application. When the client stub is first invoked, it queries a name server to obtain the transport address of the server application, which it then uses to send the remote procedure call request across the network.

Once the client runtime program knows the address of the server, it sends the request message containing the name of the remote procedure and any arguments to the server over the network. The server receives the request and passes it to its own runtime program and stub that interfaces with the remote procedure.

The remote procedure is executed on the server, and the results are sent back to the client using the response-request protocol. The server runtime program and stub package the results into a response message, which is sent back to the client through the network.

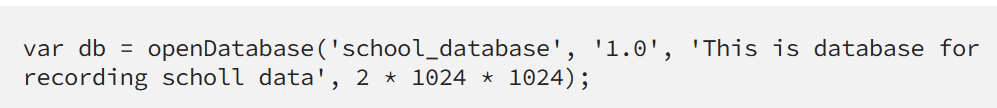
Upon receiving the response message, the client runtime the program and stub unpack the results and return them to the calling program. This allows the calling program to access the results of the remote procedure call as if it had been executed locally.



1. WebSQL.

**WebSQL** - новый метод хранения структурированных данных в традиционном стиле (формат строк и столбцов) на стороне клиента, вот что такое webSQL, также вы можете читать и записывать это пространство хранения с помощью SQL. Это реляционная БД!

* openDatabase(): этот метод используется для открытия существующей базы данных или создания новой



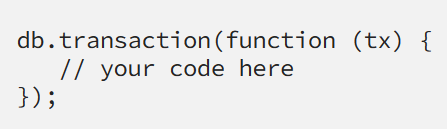
Имя базы данных

Версия базы данных

Описание базы данных

Размер базы данных

* transaction(): метод, используемый для создания транзакции данных от клиентского конца к БД



* executeSql(): это метод, который выполняет наши SQL-запросы

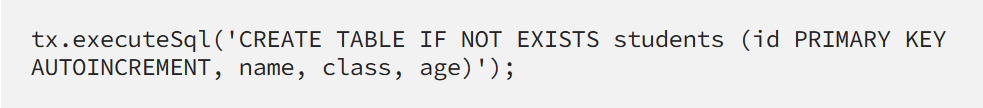
Этот метод принимает 3 параметра

1ts: SQL-запрос

2-й: поле отображения, если вы хотите отобразить поля, вы можете добавить его во 2-й параметр, если нет, вы можете передать ' [] ' пустой массив

3-й: функция обратного вызова

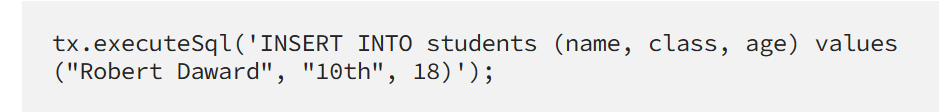
**Создание таблицы**



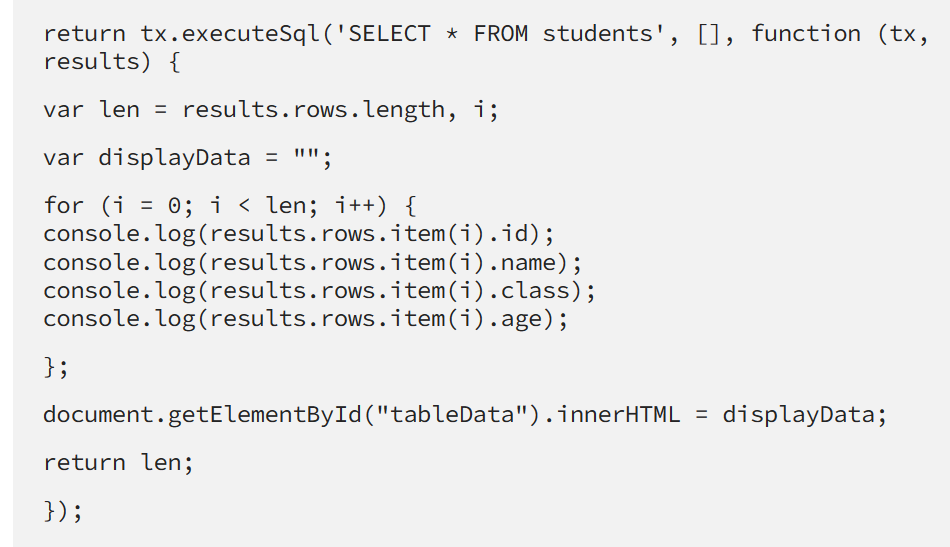
Здесь у вас есть преимущество, вы могли заметить здесь, вы можете избавиться от типов 🕶 данных

\* здесь tx - переменная из функции обратного вызова, методы executeSql работают над объектом транзакции

**Вставка данных**



**Получение данных**



**!** Стоит помнить, если вы храните свои данные на стороне клиента, скорость получения этих данных будет выше по сравнению с любым вызовом API, и ваш сайт/приложение будет загружаться быстро, и у вас будет довольный клиент 😃

1. Хранения данных на стороне клиента (Cookie, indexDb, localStorage, sessionDB…).

Хранение данных на стороне клиента позволяет веб-приложениям сохранять данные прямо в веб-браузере пользователя. Это может улучшить производительность приложения, обеспечивая быстрый доступ к данным, и позволяет приложениям работать в автономном режиме. Вот несколько технологий, которые используются для хранения данных на стороне клиента:

* **Cookies**: Cookies - это небольшие текстовые файлы, которые веб-сайты могут создавать на компьютере пользователя. Они обычно используются для сохранения предпочтений пользователя, таких как язык или статус входа в систему. Куки автоматически отправляются с каждым запросом!!!
* **localStorage**: localStorage предоставляет простой API для хранения данных на долгосрочной основе. Данные, сохраненные в localStorage, не исчезают после закрытия браузера, они остаются доступными до тех пор, пока их явно не удалить.
* **sessionStorage**: sessionStorage очень похож на localStorage, но данные, сохраненные в sessionStorage, удаляются, когда пользователь закрывает вкладку или окно браузера.
* **IndexedDB**: IndexedDB - это более сложный API для хранения значительных объемов структурированных данных, включая файлы и блобы. Это транзакционная база данных системы с функциями поиска, которые могут быть изменены или обновлены.

Важно отметить, что существуют ограничения на объем данных, которые вы можете хранить с помощью этих API1. Точный лимит варьируется в зависимости от браузера и, возможно, в зависимости от пользовательских настроек.

1. Асинхронное программирование. Что это, когда применяется. Определение. Принцип работы.

**Асинхронность (asynchrony)** подразумевает, что операция может быть выполнена кем-то на стороне: удаленным веб-узлом, сервером или другим устройством за пределами текущего вычислительного устройства.

Основное свойство таких операций в том, что начало такой операции требует значительно меньшего времени, чем основная работа. Что позволяет выполнять множество асинхронных операций одновременно даже на устройстве с небольшим числом вычислительных устройств.

**Схема работы выполнения асинхронной операции**: заявка на получение – получения результата. Если один процессор - нет возможности выполнения асинхронных операций.

**Асинхронная операция** может выполняться как в основном потоке программы, так и в другом. Кроме того, программист явно не знает, как будет выполняться данная опера-ция. Например, при обращении к пулу потоков на платформе .Net компилятор самостоятельно определяет в каком потоке будет происходить выполнение программы.

Вам нужно выкопать во дворе бассейн.

Вы взяли лопату и копаете. Это однопоточная работа

Вы пригласили друга Васю и копаете вместе, периодически задевая друг-друга лопатами. Это многопоточная работа

Пока вы копаете бассейн, Вася копает канаву под водопровод. Никто никому не мешает. Это распараллеливание

**Вы пригласили бригаду землекопов, а сами с Васей пошли пить пиво. Когда бригада все сделает, к вам придут за деньгами. Это асинхронная работа.**

Количество лопат в хозяйстве - это количество ядер в системе

1. Различия между монолитной и микро-сервисной архитектурой.

Монолитная архитектура и микросервисная архитектура - это два различных подхода к структурированию приложений.

**Монолитная архитектура** представляет собой единый модуль, в котором все компоненты приложения (клиентская и серверная логика, база данных и т.д.) объединены в одну систему. Это облегчает развертывание и управление кодом, особенно на начальных этапах проекта. Однако, по мере роста приложения, монолитная архитектура может стать сложной и неповоротливой, затрудняя внесение изменений и обновлений.

**Преимущества монолита:**

Легко реализовать, сквозные тесты, легко разворачивать и масштабировать.

**Недостатки монолита:**

Большой шарик говна – если приложение большое, сложнее сделать хорошую архитектуру, каждая новая функция вызывает необходимость менять существующий код (полностью всё связано друг с другом), нет изоляции.

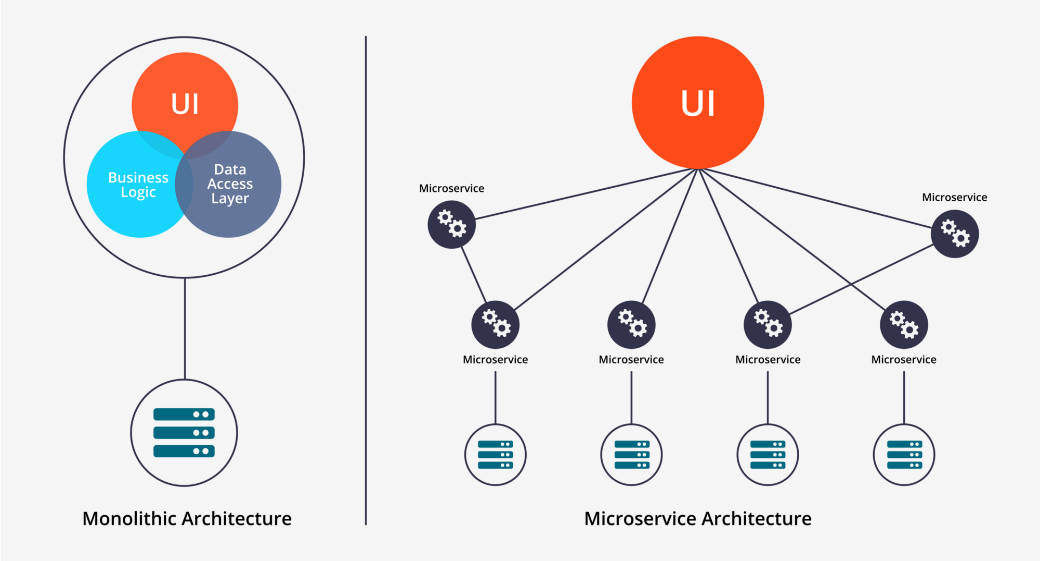
**Микросервисная архитектура**, с другой стороны, состоит из набора небольших, независимо развертываемых служб. Каждый микросервис выполняет конкретную функцию и взаимодействует с другими сервисами через четко определенные интерфейсы. Это позволяет обновлять, изменять, развертывать или масштабировать каждую службу отдельно по мере необходимости.

**Преимущества микросервисов:**

Модульность, в больших компаниях каждая команда отвечает за свой модуль, микросервисы меньше – их легче проверять, время компиляции и запуска выше, развертывание микросервисов независимо друг от друга, разные модули можно писать на разных технологиях, слабая связанность между компонентами, можно выносить отдельные модули в отдельные библиотеки и подключать их (или нет) как плагины.

**Недостатки микросервисов:**

Тяжело тестировать (выяснить где именно ошибка), микросервисы сложны в эксплуатации, больше оборудования для развертывания необходимо.



1. SOLID.

**S: Single Responsibility Principle** (Принцип единственной ответственности).

**O: Open-Closed Principle** (Принцип открытости-закрытости).

**L: Liskov Substitution Principle** (Принцип подстановки Барбары Лисков).

**I: Interface Segregation Principle** (Принцип разделения интерфейса).

**D: Dependency Inversion Principle** (Принцип инверсии зависимостей).

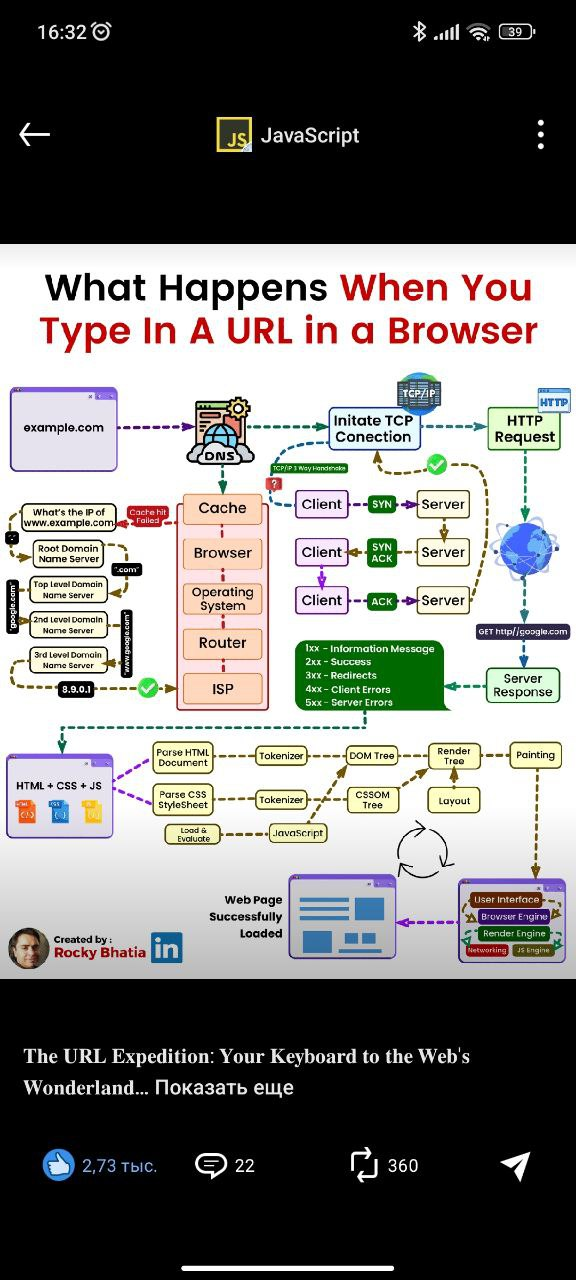
1. Swagger. Определение и понятия.

**Swagger** is an opensource framework for defining your API using a simple language that everyone can understand. It automatically helps in creating attractive and clear documentation of API. It is understandable for developers and non-developers. Anyone can give input into the design of your API because they can see it in swagger UI. Developers always want to work with the APIs that are easy to understand and perform as required.

**Advantages Of Swagger**

* Swagger is very helpful for an API Design.
* Swagger is saving the time of developers and avoid errors while writing Code.
* Because of this framework, server, client and documentation team can be in synchronization concurrently.
* Clients can quickly understand and consume API without any prior knowledge.
* The Swagger UI framework gives a clear idea of request and response parameter.
* **Swagger can auto-generates client code**
* Swagger helps testing API (makes routes (Endpoints) and params (query or body in Post-request))

1. Последовательность выполнения поискового запроса в браузере.



1. Событийный цикл: микрозадачи и макрозадачи.

Идея событийного цикла очень проста. Есть бесконечный цикл, в котором движок JavaScript ожидает задачи, исполняет их и снова ожидает появления новых.

**Общий алгоритм движка:**

1)Пока есть задачи:

выполнить их, начиная с самой старой

2)Бездействовать до появления новой задачи, а затем перейти к пункту 1

Это формализация того, что мы наблюдаем, просматривая веб-страницу. Движок JavaScript большую часть времени ничего не делает и работает, только если требуется исполнить скрипт/обработчик или обработать событие.

Задачи поступают на выполнение – движок выполняет их – затем ожидает новые задачи (во время ожидания практически не нагружая процессор компьютера)

Может так случиться, что задача поступает, когда движок занят чем-то другим, тогда она ставится в очередь.

Очередь, которую формируют такие задачи, называют «очередью **макрозадач**» (macrotask queue, термин v8).

**Отметим две детали:**

Рендеринг (отрисовка страницы) никогда не происходит во время выполнения задачи движком. Не имеет значения, сколь долго выполняется задача. Изменения в DOM отрисовываются только после того, как задача выполнена.

Если задача выполняется очень долго, то браузер не может выполнять другие задачи, обрабатывать пользовательские события, поэтому спустя некоторое время браузер предлагает «убить» долго выполняющуюся задачу. Такое возможно, когда в скрипте много сложных вычислений или ошибка, ведущая к бесконечному циклу.

**Event loop состоит из 3 частей.**

Движок **JavaScript** — компонент, который выполняет код JavaScript и обеспечивает доступ к Web API, стеку (stack) и куче (heap).

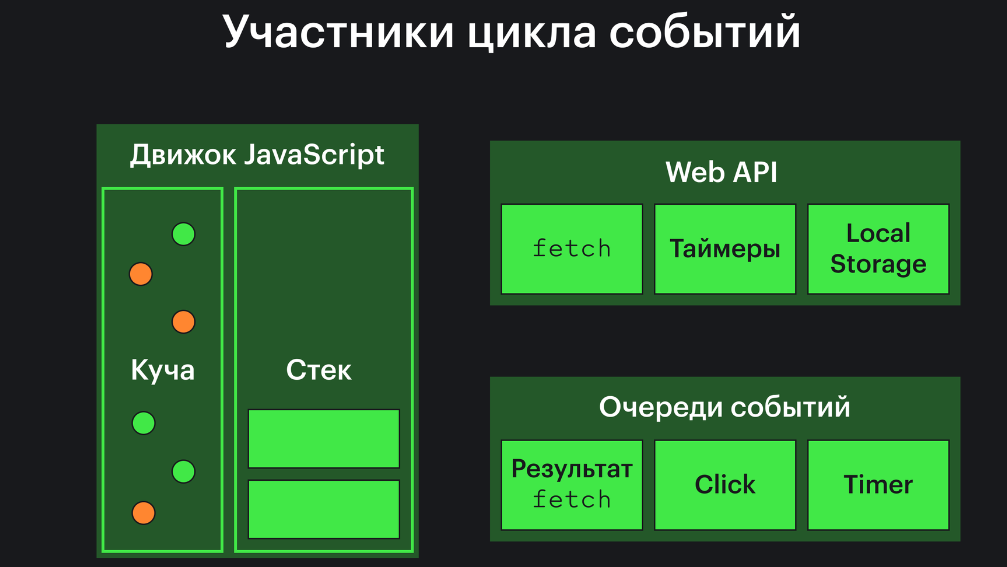
**Стек** — это место, куда попадает наш код, выполняющийся по принципу **LIFO** (последним вошёл и первым вышел).

Стек определяет порядок выполнения кода и хранит локальные переменные. Для переменных ссылочного типа на стеке хранится только указатель на данные в куче. Когда функция вызывает другую функцию, контекст первой добавляется в стек, и выполнение передаётся следующей. Когда выполнение завершается, контекст извлекается из стека, и код продолжает выполнение с предыдущей функции.

**Куча** — это область памяти, в которой хранятся ссылочные типы данных.

**Web API** — это интерфейсы браузера, такие как fetch(), таймеры и другие функции.

**Очереди задач** — место, где накапливаются задачи, соответствующие различным событиям пользовательского интерфейса и событиям, инициированным Web API. Например, результат выполнения fetch.



**Event loop работает следующим образом:**

* Выполнение задач или кода в стеке: прежде всего цикл событий ждёт, пока стек вызовов не станет пустым.
* Проверка очереди событий: как только стек вызовов опустошён, цикл событий начинает проверять очереди событий. В браузерах существует несколько очередей для разных типов задач, таких как макрозадачи и микрозадачи.
* Выбор задачи из очереди: специальный механизм в браузере определяет, какую задачу из очереди следует выполнить следующей. При выполнении задачи она удаляется из очереди, и код обработки помещается на стек.

Выполнение микрозадач: цикл событий обработает все микрозадачи до того, как перейдёт к следующей макрозадаче в очереди. После этого шага очередь микрозадач станет пустой;

Переход к следующей макрозадаче: после того как все микрозадачи выполнены, цикл событий возвращается к выбору и запуску следующей макрозадачи из очереди. Стоит заметить, что, если в процессе выполнения макрозадачи возникают микрозадачи, они получают приоритет. Только после обработки всех микрозадач будет взята в работу одна следующая макрозадача.

* Повторение цикла – бесконечное выполнение шагов цикла событий. Цикл событий постоянно следит за стеком и очередью событий, обеспечивая эффективное выполнение задач.

Если обобщить, то микрозадачи – это то, что создается внутри JS(Promise, async/await, nextTick), а макрозадачи – создаются в результате действий, происходящих вне контекста выполнения JS (setTimeout, setInterval и гэтак далей, events, AJAX, IO, UI render).

1. CORS и кросс-доменные запросы.

**CORS** существует для защиты интернета от хакеров.

Пример cross-origin запроса: HTML страница, обслуживаемая сервером с http://domain-a.com, запрашивает <img> src по адресу http://domain-b.com/image.jpg. Сегодня многие страницы загружают ресурсы вроде CSS-стилей, изображений и скриптов с разных доменов, соответствующих разным сетям доставки контента (Content delivery networks, CDNs).

В целях безопасности браузеры ограничивают cross-origin запросы, инициируемые скриптами. Например, XMLHttpRequest и Fetch API следуют политике одного источника (same-origin policy). Это значит, что web-приложения, использующие такие API, могут запрашивать HTTP-ресурсы только с того домена, с которого были загружены, пока не будут использованы CORS-заголовки.

Браузеры ограничивают кросс-доменные запросы из соображений безопасности. Такие ограничения предотвращают атаки, когда злоумышленники могут получить доступ к данным пользователя на других сайтах.

Однако, иногда вам может потребоваться выполнить кросс-доменный запрос. В этом случае вы столкнетесь с так называемой «политикой одного источника» (Same-Origin Policy), которая блокирует выполнение таких запросов.

**CORS (Cross-Origin Resource Sharing)** – это технология, которая позволяет обойти ограничения политики одного источника и выполнить кросс-доменные запросы. Для этого сервер, с которого запрашивается ресурс, должен включить специальные заголовки CORS в свои ответы.

Пример заголовка, разрешающего кросс-доменные запросы с любого источника:

**Access-Control-Allow-Origin**: \*

Если вы хотите разрешить доступ только для определенного домена, укажите его вместо звездочки:

**Access-Control-Allow-Origin**: <https://example.com>

На клиенте кросс-доменный запрос можно выполнить с помощью Fetch API или XMLHttpRequest. Важно учесть, что для работы с CORS вам может потребоваться настроить сервер таким образом, чтобы он отправлял корректные заголовки CORS. В противном случае запрос будет заблокирован политикой одного источника.

Кросс-доменное взаимодействие это React (работающий на одном порту) + ASP .Net core/NodeJs… (работающее на другом порту).

1. Назначение и возможности Geolocation API.

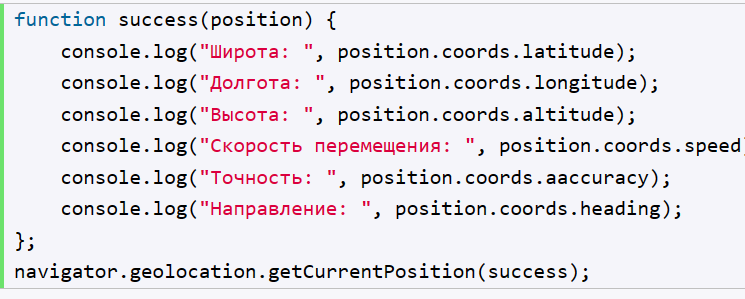
**API геолокации** позволяет пользователю предоставлять свое местоположение веб-приложениям, если он того пожелает. Из соображений конфиденциальности у пользователя запрашивается разрешение на сообщение информации о местоположении.

Доступ к API геолокации осуществляется через вызов navigator.geolocation (объект BOM); Это приведет к тому, что браузер пользователя запросит у него разрешение на доступ к данным о местоположении. Если они согласятся, то браузер будет использовать наилучшие доступные функции на устройстве для доступа к этой информации (например, GPS).

Теперь разработчик может получить доступ к этой информации о местоположении несколькими различными способами:

**Geolocation.getCurrentPosition():** извлекает текущее местоположение устройства.

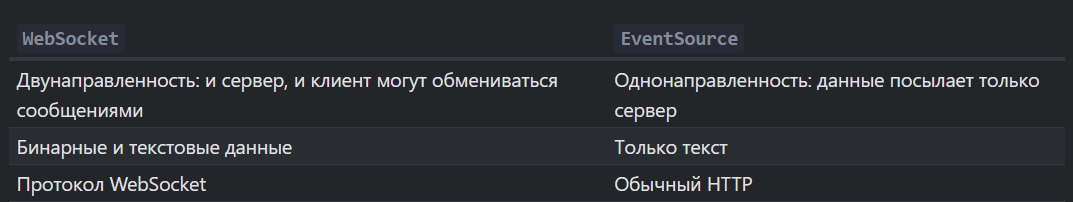
**Geolocation.watchPosition():** регистрирует функцию-обработчик, которая будет вызываться автоматически при каждом изменении положения устройства, возвращая обновленное местоположение.



1. Назначение и возможности Server-Sent Event API.

Спецификация Server-Sent Events описывает встроенный класс **EventSource**, который позволяет поддерживать соединение с сервером и получать от него события.

Как и в случае с WebSocket, соединение постоянно. Но есть несколько важных различий:



EventSource не настолько мощный способ коммуникации с сервером, как WebSocket.

**Зачем нам его использовать?**

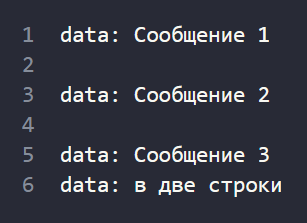
Основная причина: он проще. Многим приложениям не требуется вся мощь WebSocket.

Если нам нужно получать поток данных с сервера: неважно, сообщения в чате или же цены для магазина – с этим легко справится EventSource. К тому же, он поддерживает автоматическое переподключение при потере соединения, которое, используя WebSocket, нам бы пришлось реализовывать самим. Кроме того, используется старый добрый HTTP, а не новый протокол.

Чтобы начать получать данные, нам нужно просто создать new EventSource(url).

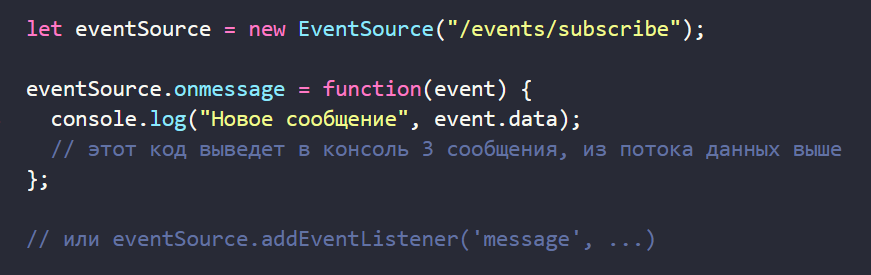
Браузер установит соединение с url и будет поддерживать его открытым, ожидая события.

Сервер должен ответить со статусом 200 и заголовком Content-Type: text/event-stream, затем он должен поддерживать соединение открытым и отправлять сообщения в особом формате:



* Текст сообщения указывается после data:, пробел после двоеточия необязателен.
* Сообщения разделяются двойным переносом строки \n\n.
* Чтобы разделить сообщение на несколько строк, мы можем отправить несколько data: подряд (третье сообщение).

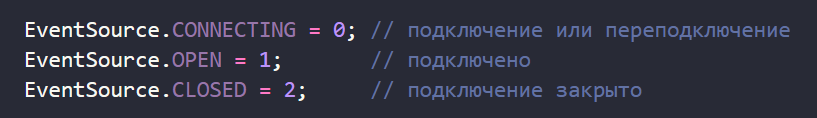
На практике сложные сообщения обычно отправляются в формате JSON, в котором перевод строки кодируется как \n, так что в разделении сообщения на несколько строк обычно нет нужды.



Кросс-доменный запрос:



У объекта EventSource есть свойство readyState, имеющее одно из трёх значений:



По умолчанию объект EventSource генерирует 3 события:

* message – получено сообщение, доступно как event.data.
* open – соединение открыто.
* error – не удалось установить соединение, например, сервер вернул статус 500.

1. Назначение и возможности Drag&Drop API.

**Drag-and-Drop API** позволяет переносить различные элементы мышью на определенную позицию на веб-странице. При перемещении элементов у нас есть источник перемещения - элемент, который перемещаем мышью, и цель перемещения - целевая область на веб-странице (другой элемент), на которую надо переместить источник перемещения.

Чтобы определить элемент на веб-странице, который можно перемещать (источник перетаскивания), нужно для этого элемент определить атрибут draggable со значением true. Теоретически в качестве перетаскиваемого элемента может выступать любой элемент веб-страницы. Например:



По умолчанию элементы не являются перетаскиваемыми.

В качестве цели перетаскиванию может служить произвольный элемент веб-страницы.

После того, как элемент определен как перемещаемый, надо также определить действия, выполняемые при перемещении элемента. Во время операции перетаскивания запускается ряд различных событий:

dragstart: генерируется, когда начинается перетаскивание элемента

drag: генерируется постоянно по мере того, как элемент перетаскивается

dragend: генерируется, когда перетаскивание элемента завершено

dragenter: генерируется, когда элемент входит в границы целевой области

dragover: генерируется постоянно (несколько раз в секунду) по мере того, как элемент перетаскивается над целевой областью

dragleave: генерируется, когда элемент покидает целевую область

drop: генерируется, перетаскиваемый элемент отпускается на целевой области

Обработчики всех выше перечисленных событий перемещения в качестве параметра получают объект типа DragEvent. Этот тип наследует свойства от MouseEvent и соответственно типа Event



1. Назначение и возможности WebAssembly API.

**WebAssembly(WAsm)** - это низкоуровневый язык, похожий на ассемблер, с компактным двоичным форматом, который работает с почти естественной производительностью и предоставляет низкоуровневые модели памяти для таких языков как C++ и Rust, с целью компиляции, чтобы они могли работать в интернете.

Когда кто-то говорит о Wasm, как правило речь идёт о его потенциальной высокой производительности. Объясняют высокую производительность тем, что браузер компилирует Wasm в эффективный машинный код. Однако высокая производительность — не единственное объяснение эффективности WebAssembly.

Wasm изначально проектировался с учётом формата компактных бинарных файлов. Поэтому его можно быстро загружать. Но, что более важно, код Wasm может быть преобразован в машинный прямо во время загрузки. Это называется потоковой компиляцией.

До появления WebAssembly код, написанный на C++ или Rust, перед использованием в вебе приходилось компилировать в JavaScript. Современные виртуальные машины компилируют JavaScript в машинный код, но начать этот процесс, по сути, можно только тогда, когда весь JS-код будет загружен. WebAssembly, напротив, компилируется в машинный код по мере загрузки, что сильно влияет на скорость запуска кода на исполнение. Настолько сильно, что узким местом уже становится скорость загрузки данных по сети, особенно на мобильных устройствах.

**WebAssembly** создавался в первую очередь как цель для компиляции, а не как язык, на котором будут писать вручную, хоть это и возможно. В основном вы будете писать код на более человекочитаемых языках, вроде C++ и Rust, и лишь затем компилировать ваш код в код машиночитаемый, который ещё называют "байт-кодом" (bytecode).

1. Назначение и возможности Shadow DOM API.

**Теневой DOM («Shadow DOM»)** используется для инкапсуляции. Благодаря ему в компоненте есть собственное «теневое» DOM-дерево, к которому нельзя просто так обратиться из главного документа, у него могут быть изолированные CSS-правила и т.д.

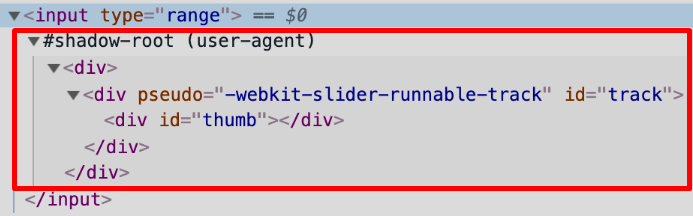
Задумывались ли вы о том, как устроены и стилизованы сложные браузерные элементы управления?

Например, <input type="range">:



Браузер рисует их своими силами и по своему усмотрению. Их DOM-структура обычно нам не видна, но в инструментах разработчика можно её посмотреть. К примеру, в Chrome для этого нужно активировать пункт «Show user agent shadow DOM».

После этого <input type="range"> выглядит так:



То, что находится под #shadow-root – и называется «shadow DOM» (теневой DOM).

Мы не можем получить доступ к теневому DOM встроенных элементов с помощью обычных JavaScript-вызовов или с помощью селекторов. Это не просто обычные потомки, это мощное средство инкапсуляции.

**Теневой DOM отделён от главного документа:**

* Элементы теневого DOM не видны из обычного DOM через querySelector. В частности, элементы теневого DOM могут иметь такие же идентификаторы, как у элементов в обычном DOM (light DOM). Они должны быть уникальными только внутри теневого дерева.
* У теневого DOM свои стили. Стили из внешнего DOM не применятся.

1. Назначение и возможности WebGL API.

**WebGL** — это API JavaScript для рендеринга 2D- и 3D-графики в любом браузере.

WebGL может работать как на встроенных, так и на автономных графических картах любого ПК.

**WebGL** переносит 3D-графику в веб-браузер. Крупнейшие поставщики браузеров Apple (Safari), Google (Chrome), Microsoft (Edge) и Mozilla (Firefox) являются членами рабочей группы WebGL.

WebGL, API 3D-рендеринга на основе браузера, предоставляет разработчикам низкоуровневый доступ к графическому процессору (GPU) устройства. Эта мощная возможность проложила путь к быстрому развитию различных библиотек высокого уровня и игровых движков. Хотя низкоуровневая природа WebGL требует определенного уровня понимания, появление этих библиотек и движков сделало разработку 3D-графики более эффективной и доступной.

Низкоуровневый характер WebGL означает, что ему не хватает определенных встроенных функций для быстрого создания высококачественной 3D-графики. Это привело к разработке многочисленных библиотек, которые абстрагируют некоторые из более сложных задач, таких как загрузка графов сцены и трехмерных объектов или обеспечение функциональности шейдеров, преобразований представлений и усеченных представлений. Некоторые популярные библиотеки, предлагающие множество функций высокого уровня (playCanvas, three.js…)

WebGL исполняется как элемент HTML5 и поэтому является полноценной частью объектной модели документа (DOM API) браузера. Может использоваться с любыми языками программирования, которые умеют работать с DOM API

**WebGL** — это контекст элемента canvas HTML, который обеспечивает API 3D графики без использования плагинов. Первая спецификация была выпущена 3 марта 2011 года. Современная версия 2.0 (несовместима с версией 1.0) доступна с 27 февраля 2017 года.