**Вопросы к экзамену «Технологии разработки web-приложений»**

Оглавление

[1. Сеть Интернет. Определение, основные понятия. 2](#_Toc168836529)

[2. Службы Интернет. Определение, основные понятия, примеры. 3](#_Toc168836530)

[3. Основные организации управления сетью Интернет. 4](#_Toc168836531)

[4. Протокол HTTP. Основные свойства, версии и их особенности. 5](#_Toc168836532)

[5. Понятия: URL/URI, MIME. 7](#_Toc168836533)

[6. Структура HTTP-запроса. 10](#_Toc168836534)

[7. Структура HTTP-ответа. 10](#_Toc168836535)

[8. HTTP-заголовки: типы, назначение, примеры. 10](#_Toc168836536)

[9. Протокол HTTPS: свойства, назначение, применение. 15](#_Toc168836537)

[10. Протокол TLS: свойства, назначение, применение, шифронаборы, процедура рукопожатия. 15](#_Toc168836538)

[11. Протокол TLS: X509-сертификаты, назначение. 19](#_Toc168836539)

[12. HTTP-аутентификация: Basic - процедура. 20](#_Toc168836540)

[13. HTTP-аутентификация: Digest - процедура. 22](#_Toc168836541)

[14. HTTP-аутентификация: Forms - процедура. 24](#_Toc168836542)

[15. HTTP-аутентификация: Token – процедура. 25](#_Toc168836543)

[16. HTTP: принципы кэширование на стороне клиента. 28](#_Toc168836544)

[17. HTTP: сохранение состояния на стороне клиента. 30](#_Toc168836545)

[18. HTML: стандарты, BOM, DOM, CSS, HTML 5 API. 31](#_Toc168836546)

[19. AJAX: асинхронный запрос, объекты XMLHTTPREQUEST и FETCH, форматы передачи данных. 36](#_Toc168836547)

[20. Протокол WebSocket: назначение, применение. 42](#_Toc168836548)

[21. Web-приложение: определение, назначение, архитектура, применение. 44](#_Toc168836549)

[22. Web-сервер: основные объекты. 46](#_Toc168836550)

[23. Web-сервер: объекты Request и Response. 47](#_Toc168836551)

[24. Web-сервер: объект Session и его жизненный цикл. 47](#_Toc168836552)

[25. Web-сервер: сохранение состояния. 48](#_Toc168836553)

[26. Web-сервер: объект Filter/Middleware его жизненный цикл. 48](#_Toc168836554)

[27. Web-сервер: объект Context его жизненный цикл. 50](#_Toc168836555)

[28. Web-сервер: принципы кэширования. 50](#_Toc168836556)

[29. Web-сервер: объект connection pool и его жизненный цикл. 51](#_Toc168836557)

[30. Web-сервер: объект listener и его жизненный цикл. 52](#_Toc168836558)

[31. Web-браузер: структура, принципы функционирования, основные объекты. 53](#_Toc168836559)

[32. Web-браузер: назначение и возможности IndexDB API. 55](#_Toc168836560)

[33. Протокол OAuth 2.0: назначение, принципы применения. 57](#_Toc168836561)

[34. Протокол WebDAV: назначение, принцип применения. 60](#_Toc168836562)

[35. WebRTC: назначение, протоколы и их назначение, API. 60](#_Toc168836563)

[36. Сервер NGINX: назначение и применение. 63](#_Toc168836564)

[В конспекте норм 63](#_Toc168836565)

1. Сеть Интернет. Определение, основные понятия.

Состоит из 4 основных компонентов:

1. **Стек протоколов TCP-IP** — это основа Интернета.
2. **Интернет службы** (DNS, DHCP, SMTP, POP3 и т.д.).
3. **Документация** в формате RFC и STD.

RFC (Request for Comments) — это серия документов, используемых для описания и публикации новых концепций, исследований, методов, поведения, или инноваций в области сетевых технологий и интернета. Документы RFC издаются и поддерживаются организацией IETF (Internet Engineering Task Force).

STD (Standard) — это подмножество RFC-документов, которые были официально приняты как интернет-стандарты. Эти документы проходят через строгий процесс оценки и утверждения, чтобы гарантировать их качество и надежность.

1. **Система организаций**, которые поддерживают сеть:

* IETF (разрабатывает стандарты интернета в RFC)
* ICANN (распространяет доменные имена)
* IANA (регистрирует MIME)
* ISOC (развитие, внедрение и распространение новых интернет-технологий) ISOC владелец RFC-стандартов.
* W3C (WWW **Consortium, некоммерческая организация для согласования стандартов)** — согласовывают *всё*, что касается веб-программирования (HTTP, CSS, SVG, URI/URL, XML, PNG, JPEG, …). Также они изготавливают стандарты и регистрируют их в IETF.

RFC: жизненный цикл описан в RFC 2026, этапы:

* Draft Internet,
* Proposed Standard,
* Draft Standard,
* Internet Standard (RFC/STD),
* Historic (вышедшие из употребления).

RFC может содержать не только стандарты, но и концепции (Experimental – результаты экспериментов, Informational – информационные, Best Current Practice – опыт применения (рекомендации и наилучшие текущие практики)).

**сущность в сети Internet, имеющая адрес** (опубликованная в Internet сущность).

Бывают:

* статические (html)
* динамические (js)
* полудинамические (html + js)

**статические** - отправляются клиенту без изменения (html-страницы, рисунки, видео-файлы, …),

**динамические** – динамически (программно) формируются на сервере и отправляются клиенту (сервлеты, JSP, http-обработчики, aspx-страницы,…).

Ресурс может быть статическим относительно сервера и динамическим относительно клиента (html-страницы с JavaScript).

**Web-приложение**: клиент-серверное приложение, применяющее для обмена данными протокол HTTP; может быть просто web-приложением (HTML+HTTP) или web-службой (API, HTTP-транспорт, формат XML, JSON)

1. Службы Интернет. Определение, основные понятия, примеры.

- это программа, система, предоставляющая услуги клиентам.

- другое название Internet-сервис, один из видов Internet-ресурса, имеющий специальное назначение

- Сервер + протокол. Протокол, описывающий доступ к этому серверу

Стандартные серверы, которые прослушивают стандартные порты (от 0 до 1024).

Службы отличаются по порту

(DNS, WWW, E-mail, FTP, ICQ, Telnet).

**DNS (Domain Name System)**: Преобразование доменных имен в IP-адреса

**WWW (World Wide Web)**: предоставляет доступ к документам и другим веб-ресурсам через интернет-протоколы, такие как HTTP/HTTPS. Доступ к веб-страницам через браузеры

**E-mail (Electronic Mail)**: Система для обмена электронными сообщениями между пользователями через интернет

**FTP (File Transfer Protocol)**: для передачи файлов между компьютерами в сети

**Telnet**: Протокол для удаленного доступа к компьютерам и управления ими через текстовый интерфейс. **Пример использования**: Администрирование серверов и сетевых устройств.

**SMTP (Simple Mail Transfer Protocol)**: Протокол для отправки электронной почты между серверами. Пересылка писем от клиента к почтовому серверу и между почтовыми серверами для доставки писем конечным получателям.

**DHCP (Dynamic Host Configuration Protocol)** для автоматического назначения IP-адресов, Управление арендой IP-адресов

1. Основные организации управления сетью Интернет.

**ISOC**: Internet Society – международная организация занимающаяся развитием сети Internet. ISOC владелец RFC-стандартов. ISOC обеспечивает правовую поддержку и финансирует все другие организации, связанные с деятельностью Internet (IETF, IAB,…).

**IETF**: Internet Engineering Task Force - рабочая группа проектирования Internet. Публикует RFC (Request for Comments – заявка на отзывы = тема для обсуждения). Задачи IETF описаны в RFC 4677.

**IAB**: Internet Architecture Board - совет по архитектуре Internet, одна из комиссий IETF, имеет консультативный статус при ISOC.

**ICANN**: Internet Corporation for Assigned Names and Numbers – корпорация по управлению доменными именами и IP-адресами.

**IANA: Internet Assigned Numbers Authority –** Администрация адресного пространства Internet. Под контролем ICANN. Кроме того регистрирует типы данных **MIME**.

Стандартизирующие:

**W3C**: **World Wide Web Consortium** – организация разрабатывающая и внедряющая web-стандарты (HTTP, HTML, URI/URL, CSS, DOM, XML, PNG, SVG,…). Возглавляет Тимоти Джон Бернерс-Ли.

ISO - международная стандартизирующая организация

1. Протокол HTTP. Основные свойства, версии и их особенности.

* версии HTTP/1.1 – действующий (текстовый), HTTP/2 – черновой (не распространен, бинарный);
* два типа абонентов: клиент и сервер;
* два типа сообщений: request и response;
* от клиента к серверу – request;
* от сервера к клиенту – response;
* на один request всегда один response, иначе ошибка;
* одному response всегда один request, иначе ошибка;
* TCP-порты: 80, 443;
* для адресации используется URI или URN;
* поддерживается W3C, описан в нескольких RFC.

**HTTP** — протокол прикладного уровня передачи данных, изначально — в виде гипертекстовых документов в формате HTML, в настоящее время используется для передачи произвольных данных.

Полудуплексный протокол прикладного уровня. в каждый момент времени сообщение передается или в одну, или в другую сторону.

*TCP-дуплексный  
Web-Socket – дуплексный*

Stateless протокол – который не сохраняет состояние между запросом и ответом (например, HTTP или UDP).

Асимметричность – сообщения, которые идут от клиента к серверу отличаются от сообщений, которые идут от сервера к клиенту (Request, Response).

***\\ Версии***

**НTTP/2:** обратно совместим с HTTP/1.1.

**НTTP/2:** бинарный формат.

**НTTP/2:** разрабатывается IETF, основан (IETF-ответ) на протоколе **SPDY** (Google), c 2016 Google отказался от поддержки SPDY в пользу HTTP/2 (Chromium 51+).

**НTTP/2:** без шифрования (HTTP URI) и с шифрованием (HTTPS URI) над TLS 1.2+, большинство браузеров поддерживает только HTTPS/2.

**НTTP/2:** Псевдозагаловки

**мультиплексирование**, в браузере количество одновременно работающих TCP-соединений ограничено

**мультиплексирование**;в HTTP/1.1 применяли спрайты (при загрузке несколько файлов объединяются в один спрайт, например JS или CSS); в HTTP/2 спрайты можно применять, но в некоторых случаях снижает эффект мультиплексирования (спрайт загружается полностью, а при мультиплексировании только те которые запрашиваются), но улучшает сжатие.

**приоритезация** – запросу можно поставить приоритет; 2 типа: 1) каждый поток имеет определенный вес, в зависимости от веса сервер распределяет нагрузку; 2) браузер просит загружать некоторые элемента контента в заданной последовательности (по умолчанию).

**сжатие HTTP-заголовков** – заголовки передаются в сжатом виде (метод HPACK).

**push-сервер** – сервер может отсылать данные клиенту (браузеру) для записи в кэш.

клиентможет сбросить запрос с большим Content-Length (это невозможно сделать в HTTP/1.1).

**HTTP3:** библиотеки с открытым кодом для разработки серверной и клиентской компоненты.

**QUIC:** протокол поверх UDP, поддерживающий все возможности TCP, TLS, HTTP/2. Новый транспортный протокол разработать не реально, т.к. сетевое hardware, работающее на транспортном уровне заменить не возможно.

**QUIC:** QUIC-connection над UDP, несколько независимых потоков, ошибки передачи в одном потоке не влияют на другой поток.

**QUIC:** легкие потоки**,** позволяетлегко переключаться на другие IP, бесшовный WiFi.

**QUIC:** шифрование внутри.

**QUIC:** реализован на уровне приложения, не встроен в OC.

**QUIC:** компрессия заголовков похожа, но отличается от HTTP2.

1. Понятия: URL/URI, MIME.

**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 ресурсу, 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

**Структура и компоненты URI:**

**1)Scheme (схема).** Каждый URI начинается с имени схемы, которое относится к спецификации для присвоения идентификаторов в этой схеме. **http:**

**2)Authority(Узел).** Компонента authority начинается с двойного слеша(//) и заканчивается следующим слешем(/), знаком вопроса(?) или октоторпом или концом URI.

**Userinfo —** под-компонент authority, использующийся для авторизации пользователя на ресурсе. Состоит из username и необязательного password, от остальной части authority отделяется символом "@". Не смотря на то, что параметр password указан в спецификации, его использование крайне не рекомендуется, т. к. фактически производится передача пароля к учетной записи username, в незашифрованном виде.

**Host — компонент** authority, Имя хоста или IP-адрес, указывающий на целевой узел**,** использующийся для определения целевого узла, который может находиться как в сети интернет, так и вне её, в зависимости от указанной схемы. Данная компонента не чувствительна к регистру. Хост может представлять из себя либо IP-адрес, либо регистрационное имя (reg-name) и, опционально, следующий за ними порт(port). **example.com**

**3)Path (Путь).** определяет местоположение ресурса на сервере. Компонента пути содержит данные, обычно, организованные в иерархической форме, которые, вместе с данными в неиерархическом компоненте запроса (Query), служит, чтобы идентифицировать ресурс в рамках схемы URI и authority (если таковая компонента указана).Путь начинается со слеша(/) и заканчивается знаком вопроса(?), октоторпом(#) или концом URI. **/path/to/resource**

**4)Query (Запрос).** Компонента запроса содержит данные, организованные в неиерархической форме, которые, вместе с данными в иерархическом компоненте пути (Path) Начинается со знака вопроса (?) и продолжается до октоторпа (#) или конца URI, состоит из пар "ключ-значение", разделённых символом амперсанда (&). **?key1=value1&key2=value2**

**5)Fragment (Фрагмент).** Компонента фрагмент позволяет осуществить косвенную идентификацию вторичного ресурса по отношению к первому. Семантика фрагмента никак не ограничена, фрагмент начинается октоторпом(#) и заканчивается концом URI, при этом может состоять из абсолютно любого набора символов. **#section2**

**Стуруктура URN:**

*Т. е., в отличие от URL, который ссылается на како-то место, где хранится документ,* ***URN ссылается*** *на сам документ, и при перемещении документа в другое место ссылка не изменится.*

*В силу того, что URN концептуально отличается от URL, то и система разрешения имен у него другая — DDDS, которая преобразует URN в URL, по которым можно найти ресурс/объект или что бы то ни было, на что ссылается URN.*

**PURPL:** Persistent Uniform Resource Locator – постоянный унифицированный локатор ресурса. Например: http://purpl.by/bstu/faculties/lh/lv.html, http://purpl.by – указывает на БД, в которой по имени bstu можно найти URL http://www.bstu.by и получить искомый URL - http://www.bstu.by/faculties/lh/lv.html. Доступ к конечному ресурсу через redirect.

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

Структура MIME-типа состоит из типа и подтипа, разделенных косой чертой (/) без пробелов.

**Тип** определяет категорию данных (например, text, image, audio).

**Подтип** определяет конкретный формат данных внутри этой категории (например, plain для text, jpeg для image).

Примеры

text/plain — обычный текст.

text/html — HTML-документ.

image/jpeg — изображение в формате JPEG

video/mp4: Видео файл в формате MP4

application/json — JSON-данные.

application/xml:

MIME-тип нечувствителен к регистру, но обычно записывается в нижнем регистре. Необязательный параметр может быть добавлен для указания дополнительных деталей тип/подтип;параметр=значение

1. Структура HTTP-запроса.

* метод;
* URI;
* версия протокола (HTTP/1.1);
* заголовки (пары: имя/заголовок);
* параметры (пары: имя/заголовок);
* расширение.

1. Структура HTTP-ответа.

* версия протокола (HTTP/1.1);
* код состояния (1xx, 2xx, 3xx, 4xx, 5xx);
* пояснение к коду состояния;
* заголовки (пары: имя/заголовок);
* расширение

1. HTTP-заголовки: типы, назначение, примеры.

HTTP-заголовки содержат метаданные о запросе или ответе. Они помогают клиенту и серверу обмениваться информацией и управлять поведением соединения.

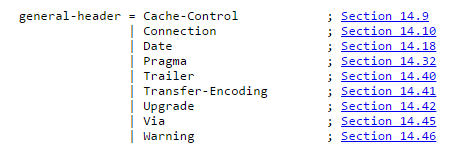
Изначально есть 2 типа заголовков:

1)Определены в HTTP

2)Кастомные (До недавнего времени было принято добавлять префикс X- к именам пользовательских заголовков,)

Заголовки делятся на 4 типа:

1. **General**: общие заголовки, используются в запросах и ответах;



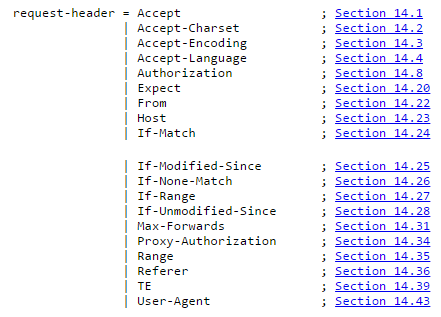
**Connection** – определяет, остаётся ли сетевое соединение активным после завершения текущей транзакции (запроса). Если в запросе отправлено значение keep-alive, то соединение остаётся и не завершается, позволяя выполнять последующие запросы на тот же сервер. (Кроме keep-alive еще может быть close (закрыть соединение), upgrade (переключение на другой протокол))

**Date** – дата и время, когда было создано сообщение

**Pragma** – используется для обратной совместимости с HTTP/1.0, где заголовок Cache-Control ещё не присутствует

**Cache-Control** – определяет поведение браузера при кэшировании. Он может содержать различные директивы, определяющие, должен ли браузер кэшировать данные и насколько свежими должны быть эти данные (Может содержать: no-store (запрещает кеширование), no-cache (необходимость отправить запрос на сервер для валидации ресурса перед использованием закешированных данных), must-revalidate (кешированный ответ должен быть проверен на актуальность перед использованием))

1. **Request**: используются только в запросах;



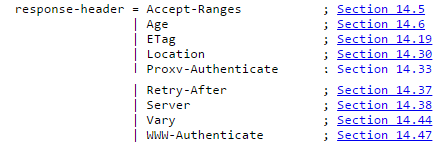
**User-Agent** – содержит информацию о клиенте, который отправляет запрос. Это может включать в себя детали о браузере, операционной системе и другую информацию, которая может быть полезна для сервера при формировании ответа.

**Accept** – какие типы контента (выраженные как MIME-типы), клиент может понять. Сервер затем выбирает одно из предложений и информирует клиента о своём выборе с помощью заголовка ответа Content-Type. (Если \*/\* - то клиент, у нас это браузер, может принять любой тип контента (html, json, png …)

**Accept-Encoding** – какие методы сжатия контента поддерживаются клиентом. Сервер затем может использовать эту информацию, чтобы определить, как сжать ответ для передачи по сети. (Если gzip, deflate, br – то это просто алгоритмы сжатия, инфа по каждому думаю излишне

**Authorization**: передаёт данные авторизации для доступа к защищённым ресурсам

1. **Response**: используются только в ответах;



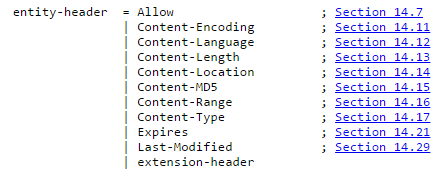
**Server** – содержит информацию о программном обеспечении, используемом сервером для обработки запросов (В данном случае, у меня на сервере работает веб-сервер Apache, еще может быть nginx, GWS, Microsoft IIS)

**Vary** – определяет, как сопоставить будущие заголовки запроса, чтобы решить, можно ли использовать кешированный ответ, а не запрашивать новый с исходного сервера (Если установлен Accept-Encoding, то сервер может отправлять разные версии ресурса в зависимости от того, какие кодировки сжатия поддерживаются клиентом, может еще быть User-Agent (разные версии в зависимости от, к примеру, браузера), Accept-Language (разные версии в зависимости от языка пользователя), Origin (в зависимости от источника запроса)

**WWW-Authenticate:** Указывает метод аутентификации, поддерживаемый сервером (например, WWW-Authenticate: Basic realm="Access to the staging site").

**Keep-Alive** – используется для управления постоянными соединениями. Он может указывать, сколько времени и сколько запросов может оставаться открытым текущее неиспользованное соединение (Timeout – кол-во секунд между двумя последовательными запросами на одном и том же соединении, max – максимальное кол-во запросов, которое можно отправить по одному соединению)

1. **Entity:** для сущности в ответах и запросах.



**Expires** – содержит дату/время, по истечении которой ответ сервера считается устаревшим. Прошедшая или невалидная дата, например 0, обозначает, что ресурс уже устарел

**Content-Encoding** – указывает на кодировку, применённую к телу сообщения. Это позволяет клиенту информацию как декодировать тело, чтобы получить медиа-тип ссылающийся на заголовок Content-Type

**Content-Type** – описывает MIME-тип содержимого, передаваемого в теле сообщения

**Content-Length** – Указывает длину содержимого в байтах

**Date** General HTTP заголовок, содержащий дату и время, в которое сообщение было создано.

Заголовок **Server** описывает программное обеспечение, используемое исходным сервером, который обработал запрос, то есть сервером, сгенерировавшим ответ.

Заголовок **Transfer-Encoding** указывает форму кодирования, используемую для безопасной передачи тела полезных данных пользователю.

Заголовок представления **Content-Type** используется для указания исходного типа мультимедиа ресурса (до любого кодирования контента, примененного для отправки).

Заголовок **Content-Length** указывает размер отправленного получателю тела объекта в байтах.

HTTP заголовок запроса **Accept** указывает, какие типы контента, выраженные как MIME типы, клиент может понять.

HTTP-заголовок запроса **Accept-Encoding** указывает кодировку контента (обычно алгоритм сжатия), которую может понять клиент.

HTTP-заголовок запроса **Accept-Language** указывает естественный язык и локаль, которые предпочитает клиент.

Общий заголовок **Cache-Control** используется для задания инструкций кеширования как для запросов, так и для ответов. Инструкции кеширования однонаправленные: заданная инструкция в запросе не подразумевает, что такая же инструкция будет указана в ответе

Общий заголовок **Connection** определяет желаемые опции для настраиваемого подклюбчения!!!!, останется ли сетевое соединение открытым после завершения текущей транзакции. Если отправленное значение поддерживает активность, соединение является постоянным и не закрывается, что позволяет выполнять последующие запросы к тому же серверу.

Заголовок HTTP-запроса **Cookie** содержит сохраненные файлы cookie HTTP, связанные с сервером (т. е. ранее отправленные сервером с заголовком Set-Cookie или установленные в JavaScript с помощью Document.cookie).

Заголовок запроса **Host** указывает хост и номер порта сервера, на который отправляется запрос. Если порт не указан, подразумевается порт по умолчанию для запрошенной службы (например, 443 для URL-адреса HTTPS и 80 для URL-адреса HTTP).

Заголовок HTTP-запроса **Referer** содержит абсолютный или частичный адрес, с которого был запрошен ресурс. Заголовок Referer позволяет серверу идентифицировать ссылающиеся страницы, с которых люди посещают или где используются запрошенные ресурсы.

Заголовок запроса HTTP **Upgrade-Insecure-Requests** отправляет на сервер сигнал, выражающий предпочтение клиента в отношении зашифрованного и аутентифицированного ответа, а также о том, что он может успешно обрабатывать директиву CSP небезопасных обновлений. Если 1 – то браузер хочет получать сайт в защищенном режиме.

**User-Agent** – это строка с характеристиками, по которым сервера и сетевые узлы могут определить тип приложения, операционную систему, производителя и/или версию пользовательского агента.

1. Протокол HTTPS: свойства, назначение, применение.

**HTTPS** *— это расширение HTTP, обеспечивающее защиту передачи данных с использованием* ***TLS*** *(Transport Layer Security).*

HTTPS обеспечивает:

1. Аутентификацию (обеспечивает аутентификацию сервера, что позволяет клиентам убедиться в том, что они соединены с правильным веб-сервером, а не с поддельным или вредоносным. Это достигается благодаря использованию цифровых сертификатов, выдаваемых доверенными центрами сертификации (Certificate Authorities, CA).)
2. Шифрование канала с помощью протокола TLS

Процесс установления защищенного соединения состоит из нескольких этапов, известных как TLS Handshake. После завершения TLS Handshake все данные, передаваемые между клиентом и сервером, шифруются с использованием симметричных ключей, созданных во время Handshake.

*Основные аспекты HTTPS включают:*

*- TLS*

*- Шифронаборы*

*- Сертификаты*

*- Процедура рукопожатия TLS*

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

HTTPS гарантирует, что данные не были изменены или повреждены в процессе передачи. Это достигается через использование хэш-функций и цифровых подписей.

1. Протокол TLS: свойства, назначение, применение, шифронаборы, процедура рукопожатия.

**Transport Layer Security (TLS)** — это протокол, предназначенный для обеспечения безопасности передачи данных в сети

**RFC 2246,** новое название **Transport Layer Security (TLS).** Последние обновления RFC 5246, 6176

**Свойства TLS**

1. **Шифрование:** TLS использует симметричные и асимметричные алгоритмы для шифрования данных, передаваемых между клиентом и сервером. Это предотвращает перехват данных третьими сторонами.

2. **Аутентификация:** TLS обеспечивает аутентификацию серверов и клиентов, используя цифровые сертификаты. Это позволяет сторонам проверить подлинность друг друга.

3. **Целостность данных:** TLS использует механизмы целостности (обычно HMAC) для обеспечения того, что данные не были изменены во время передачи.

4. **Обеспечение конфиденциальности:** Шифрование данных гарантирует, что информация остается конфиденциальной и не может быть прочитана третьими лицами.

**Назначение TLS**

Протокол TLS (Transport Layer Security) предназначен для обеспечения безопасности данных, передаваемых по сети. Его основные задачи включают:

1. **Защита данных от перехвата и подслушивания. (Конфиденциальность)**

2. **Проверка подлинности участников обмена данными. (Аутентификация)**

3. **Обеспечение целостности передаваемых данных. (Целостность)**

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

**в шифронабор входят**: криптосистема для получения общего секрета; криптосистема для аутентификации сервера и сеансового секрета; шифр для защиты передаваемых данных; хеш-функция для кода аутентификации HMAC

**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.

**Шифронабор** (cipher suite) — это набор алгоритмов, используемых для обеспечения безопасности сетевого соединения с использованием протоколов шифрования, таких как TLS. Шифронабор определяет, какие криптографические алгоритмы будут использоваться для аутентификации, шифрования данных и обеспечения их целостности.

Шифронабор обычно включает в себя следующие компоненты:

1. Алгоритм Ключевого Обмена определяет, как клиент и сервер будут обмениваться ключами, которые затем будут использоваться для шифрования данных.

2. Алгоритм Аутентификации ­ используется для проверки подлинности сторон. Обычно это метод цифровой подписи или обмен ключами.

3. Алгоритм Симметричного Шифрования для шифрования данных, передаваемых между клиентом и сервером.

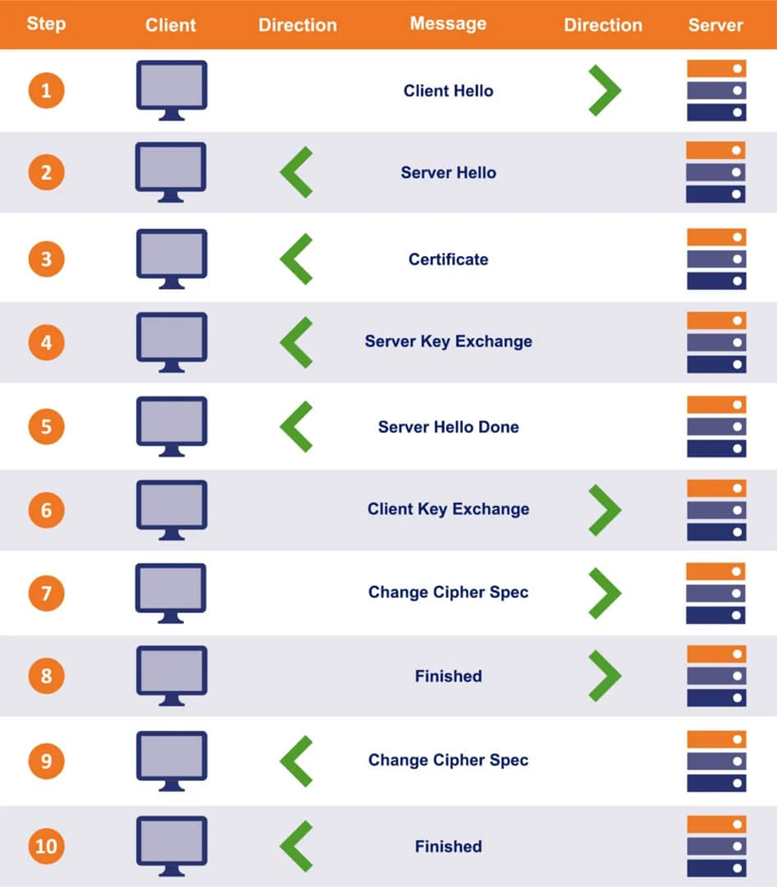
4. Алгоритм Хэширования используется для создания кода аутентификации сообщений (MAC) для обеспечения целостности данных и аутентификации сообщений.

**Процедура Рукопожатия TLS (TLS Handshake)**

Цель:

1. Установка соединения
2. Договорится о ключе шифрования

Процедура рукопожатия TLS — это процесс, в ходе которого клиент и сервер устанавливают безопасное соединение. Она состоит из нескольких этапов:



1. **Клиент** выдает запрос серверу (Client Hello).

2. **Сервер** подписывает свой сертификат и высылает клиенту (Server Hello).

3. **Клиент** проверяет сертификат в центре сертификации, которому доверяет.

4. **Клиент** сравнивает данные сертификата с информацией центра сертификации.

5. **Клиент** сообщает серверу, какие ключи шифрования он поддерживает.

6. **Сервер** выбирает подходящую длину ключа.

7. **Клиент** генерирует симметричный ключ, шифрует его открытым ключом.

**Сервер** получает симметричный ключ и расшифровывает его.

Способы вычисления хэша:  
 **ECDHE -** Диффи-Хеллман на эллиптических кривых для вычисления общего секрета;

**ECDSA -** аутентификацияданных на этапе установления соединения на основе цифровой подписи на эллиптических кривых;

**AES\_128** – шифрование полезной нагрузки с помощью алгоритма AES с 128-битным ключом в режиме GCM;

**SHA256** -для хеширования применяется алгоритм SHA с 256-битным ключом

1. Протокол TLS: X509-сертификаты, назначение.

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

X.509 — это стандарт, определяющий структуру цифровых сертификатов. Он описывает, как должны выглядеть сертификаты, какие поля они должны содержать и как их можно использовать.

**Сертификат X.509:** стандартныйформат хранения и  транспортировки  атрибутов безопасности; главное – открытый ключ; сертификаты выдают центры сертификации (Certificate Authority, CA). [http://www.thwate.com](http://www.thwate.com/), [http://www.Verysign.com](http://www.verysign.com/);  если это intranet, то можно использовать **Certificate Server Active Directory**; сертификат содержит:  имя держателя, адрес, серийный номер сертификата,  даты проверки, открытый ключ держателя.

**сертификат содержит**:

·         версию

·         Имя издателя

·         серийный номер

·         имя субъекта(кому выдан)

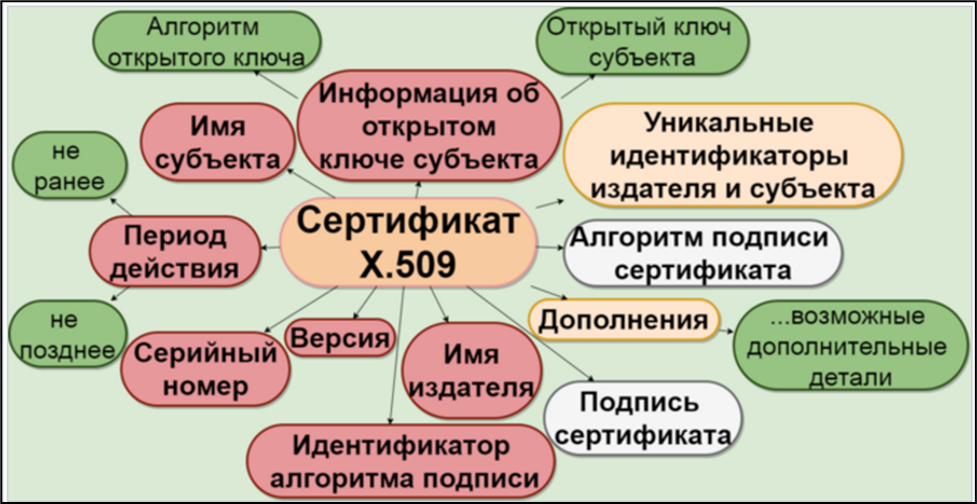
·         перечень доменов(имен URI, на которые распространяеца сертификат)

·         открытый ключ, который будет использоваться для шифрования и проверки подписей.

·         период действия

·         Цифровая подпись

·         Алгоритм, используемый для создания цифровой подписи



Выдает центр сертификации (СА). [http://www.thwate.com](http://www.thwate.com/), [http://www.Verysign.com](http://www.verysign.com/). Если это intranet, то можно использовать **Certificate Server Active Directory**. Сертификат содержит:  имя держателя, адрес, серийный номер сертификата,  даты проверки, открытый ключ держателя

1. HTTP-аутентификация: Basic - процедура.

**Идентификация** – заявление пользователя о себе.

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

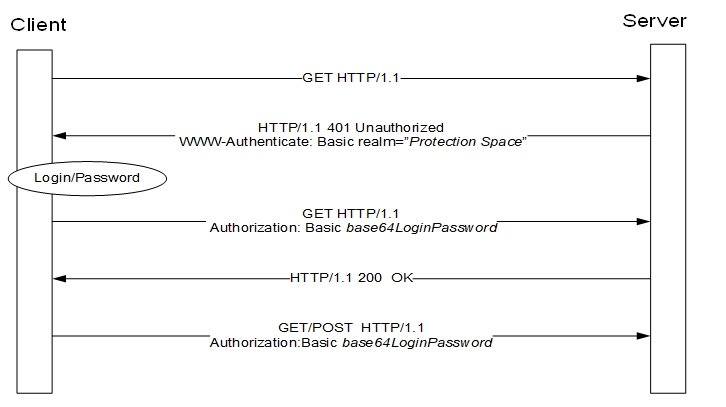
**Авторизация** - процедура проверки прав аутентифицированного пользователя.

Самый распространенный и поддерживаемый протокол.

При обычной аутентификации имя пользователя и пароль объединяются в одну строку и разделяются двоеточием (:). После этого он кодирует их, используя кодировку Base64. Несмотря на то, как это выглядит, зашифрованная последовательность символов небезопасна, и вы можете легко ее декодировать.

Целью кодировки Base64 является не шифрование, а обеспечение совместимости имени пользователя и пароля с HTTP. Основная причина этого в том, что вы не можете использовать международные символы в заголовках HTTP.

HTTP Basic аутентификация передает учетные данные в открытом виде через сеть, поэтому она не является самым безопасным методом аутентификации. Использование HTTPS вместо HTTP сильно улучшает безопасность, так как все данные передаются в зашифрованном виде. Безопасно только в зашифрованном канале.



1. Клиент делает запрос к ресурсу, который требует аутентификации.

GET /protected-resource HTTP/1.1

Host: example.com

2. Сервер отвечает с кодом состояния 401 Unauthorized и заголовком WWW-Authenticate, указывающим, что требуется Basic Authentication.

HTTP/1.1 401 Unauthorized

WWW-Authenticate: Basic realm="example"

3. Клиент отправляет повторный запрос с заголовком Authorization, содержащим закодированные учетные данные (имя пользователя и пароль).

Учетные данные кодируются с использованием Base64. Формат заголовка следующий:

если имя пользователя — user и пароль — password, строка user:password кодируется в dXNlcjpwYXNzd29yZA==.

**Пример заголовка:**

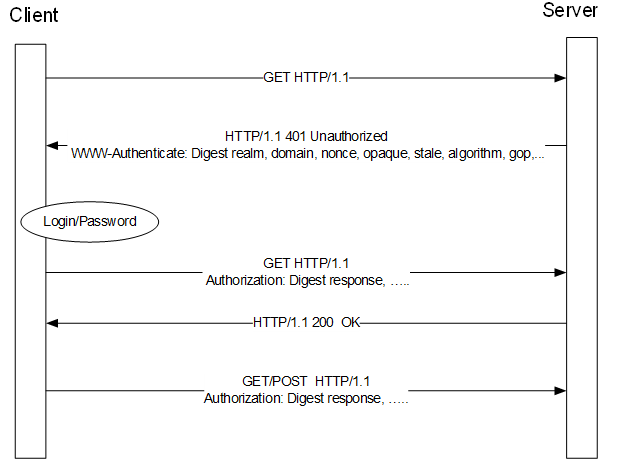
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=

4. Сервер проверяет учетные данные Сервер декодирует строку из Base64 и проверяет полученные учетные данные (имя пользователя и пароль).

5. Если учетные данные верны, сервер возвращает запрашиваемый ресурс с кодом состояния 200 OK.

Если учетные данные неверны, сервер снова возвращает ответ с кодом состояния 401 Unauthorized

1. HTTP-аутентификация: Digest - процедура.

****

**Realm – обычно область действия аутентификации.**

**WWW-Authenticate: Digest realm + параметры для хеширования**

**Схема Дайджеста** основана на простой парадигме «вызов-ответ». Схема дайджеста запрашивает использование значения nonce и может указывать на то, что хеширование имени пользователя поддерживается. Допустимый ответ содержит дайджест без ключа, содержащий имя пользователя, пароль, заданное значение nonce, метод HTTP и запрошенный URI. Таким образом, пароль никогда не передается в открытом виде, а имя пользователя может быть хешировано, в зависимости от указания, полученного от сервера. Имя пользователя и пароль должны быть заранее заданы каким-либо образом, не предусмотренным в этом документе.

**Дайджест-аутентификаци**я использует **криптографическое хеширование MD5** в сочетании с использованием **одноразовых номеров**. Таким образом, информация о пароле скрывается для предотвращения различных видов злонамеренных атак.

1. **Клиен**т отправляет неаутентифицированный запрос
2. **Сервер** отвечает с кодом состояния **401 Unauthorized** и заголовком **WWW-Authenticate**, указывающим, что требуется Digest Authentication. Этот заголовок содержит ряд параметров, которые будут использованы для создания хеша. **nonce (nonce i.e. number used only once)**: Одноразовая строка, выданная сервером для предотвращения повторных атак.
3. **Клиент** вычисляет хэш-значение, основанное на nonce, имени пользователя, пароле, HTTP-методе и URI, и отправляет это значение в заголовке **Authorization: Digest**.
4. **Сервер** также вычисляет хэш на своей стороне и сравнивает его с хэшем, присланным клиентом. Если значения совпадают, сервер аутентифицирует клиента

**nonce** и **opaque** - определенные сервером строки, которые клиент возвращает после их получения

**qop (качество защиты)** - одно или несколько предопределенных значений («auth» | «auth-int» | токен). Эти значения влияют на вычисление дайджеста.

**cnonce** - одноразовый номер клиента, должен генерироваться, если задано qop. Он используется, чтобы избежать [атак на выбранный открытый текст](https://en.wikipedia.org/wiki/Chosen-plaintext_attack) и предоставить сообщению защиту целостности.

**nc** - счетчик одноразовых номеров, должен быть отправлен, если задано qop. Эта директива позволяет серверу обнаруживать повторы запросов, сохраняя свою собственную копию этого счетчика - если одно и то же значение nc появляется дважды, то запрос является повтором.

1. HTTP-аутентификация: Forms - процедура.



1.  **Запрос страницы входа (GET /Index):** Клиент (браузер) запрашивает начальную страницу, которая возвращает HTML-форму для входа.

2.  **Отправка учетных данных (POST /Login):** Пользователь вводит свои учетные данные и отправляет их на сервер с помощью POST-запроса. Данные логина и пароля включены в тело запроса.

3.  **Создание и отправка токена (HTTP/1.1 302 Found):** Если сервер успешно проверяет учетные данные, он создает токен сессии (например, закодированный в base64) и отправляет его обратно клиенту в cookie. Код 302 Found указывает на перенаправление, часто на защищенную страницу или основную страницу после входа.

4.  **Доступ к защищенному ресурсу (GET/POST /Index):** После успешной аутентификации клиент использует полученный токен для доступа к защищенным ресурсам. Токен отправляется в заголовке cookie. Сервер проверяет этот токен и, если он действителен, предоставляет доступ к ресурсу.

Forms (нет стандарта, но применяется)

теперь веб-сайт сверяет не логин и пароль, а токен из куки — если совпал, значит это авторизованный пользователь и ему можно показать ограниченный для доступа контент

1. HTTP-аутентификация: Token – процедура.

**Token:** битовая последовательность, построенная по определенному принципу

**Token-аутентификация:** аутентификация, использующая token для идентификации пользователя или авторизации операции пользователя

**Token-аутентификация:** применяется, как правило, для реализации Single Sign-On в распределенных системах

JWT состоит из трех частей, разделенных точками (.):

1. **Header (Заголовок)**
2. **Payload (Полезная нагрузка)**
3. **Signature (Подпись)**

base64

1. Header (Заголовок)

Заголовок обычно содержит два поля:

* тип токена (указывает, что это JWT)
* алгоритм хэширования, используемый для подписи токена (например, HMAC SHA256).

2. Payload (Полезная нагрузка)

* Эмитент – тот кто выдал токен.
* Субъект – идентифицирует пользователя.
* Аудитория – указывает, для кого предназначен токен.
* время истечения – время, после которого токен недействителен.
* не ранее – время, до которого токен недействителен.
* время выдачи токена.
* JWT ID – уникальный идентификатор токена.

**Как это работает**

1. **Создание токена**: Сервер создает JWT, включающий нужные утверждения, подписывает его и отправляет клиенту.
2. **Использование токена**: Клиент хранит токен (обычно в локальном хранилище или куки) и отправляет его в заголовке Authorization для доступа к защищенным ресурсам.
3. **Проверка токена**: Сервер проверяет подпись и срок действия токена при каждом запросе к защищенным ресурсам.

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

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

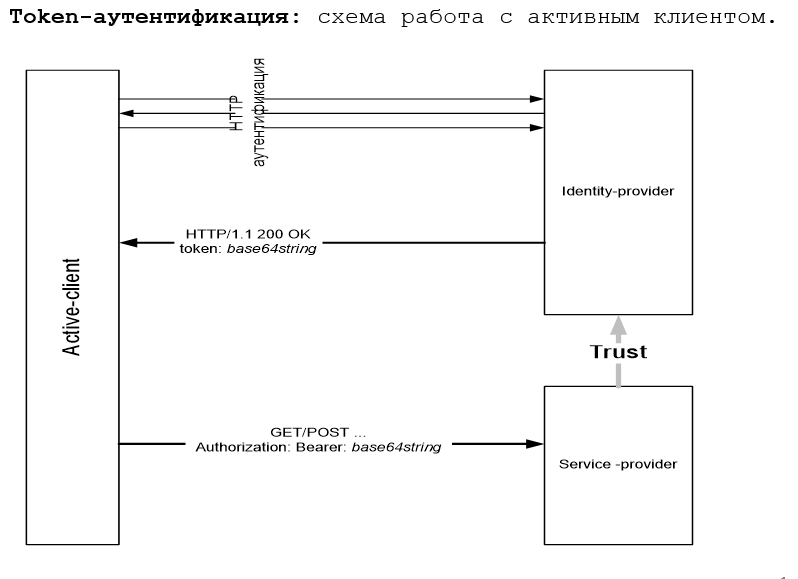
**активный клиент** – программный код, который может выполнять любые запросы и обрабатывать любые ответы. Обычно – это пользовательская программная реализация http-клиента.

**Пассивный клиент** – браузер

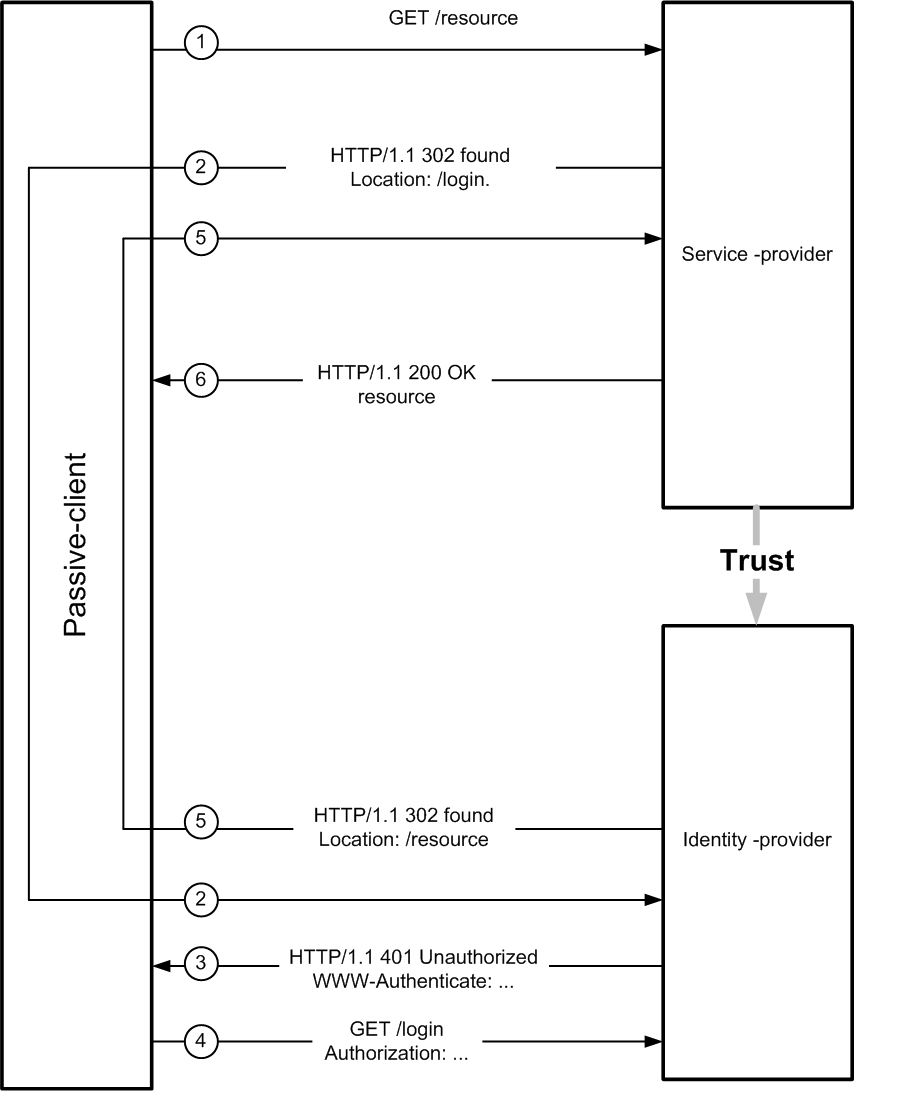
Service-provider и Identity-provider должны иметь общий секретный ключ для шифрования/проверки token’а.

**Активный клиент** – программный код, который может выполнять любые запросы и обрабатывать любые ответы. Обычно – это пользовательская программная реализация http-клиента.

**Пассивный клиент** – браузер.



**Token-аутентификация**: схема работа с пассивным клиентом (браузером).

****

1. HTTP: принципы кэширование на стороне клиента.

**Кеширование** – перемещение данных из менее быстрой памяти с более быструю память.

HTTPS не кэшируется

**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.

**Заголовок ответа Last-modified и заголовок запроса if-Modified-Since.**

Сервер добавляет заголовок Last-modified к файлу (ответу), который он отдает браузеру.

Теперь браузер знает, что файл был создан (или изменен) 1 декабря 2014. В следующий раз, когда браузеру понадобится тот же файл, он отправит запрос с заголовком if-Modified-Since.

Если файл не изменялся, сервер отправляет браузеру пустой ответ со статусом 304 (Not Modified). В этом случае, браузер знает, что файл не обновлялся и может отобразить копию, которую он сохранил в прошлый раз.

**заголовок ответа Etag - заголовок запроса If-None-Match**

Принцип работы Etagочень схож с Last-modified, но, в отличии от него, не привязан ко времени.

Идея заключается в том, что при создании и каждом изменении сервер помечает файл особой меткой, называемой ETag, а также добавляет заголовок к файлу (ответу), который он отдает браузеру.

Теперь браузер знает, что файл актуальной версии имеет ETag равный “686897696a7c876b7e”. В следующий раз, когда брузеру понадобится тот же файл, он отправит запрос с заголовком If-None-Match: "686897696a7c876b7e".

Сервер может сравнить метки и если файл не изменялся, отправить браузеру пустой ответ со статусом 304 (Not Modified). Как и в случае с Last-modified браузер выяснит, что файл не обновлялся и сможет отобразить копию из кэша.

**Заголовок Expired**

Принцип работы этого заголовка отличается от вышеописанных Etag и Last-modified. При помощи Expired определяется “срок годности” (“срок акуальности”) файла. Т.е. при первой загрузке сервер дает браузеру знать, что он не планирует изменять файл до наступления даты, указанной в Expired

В следующий раз браузер, зная, что “дата истечения срока годности” еще не наступила, даже не будет пытаться делать запрос к серверу и отобразит файл из кэша.

**Заголовок Cache-control с директивой max-age**.

Принцип работы Cache-control: max-age очень схож с Expired. Здесь тоже определяется “срок годности” файла, но он задается в секундах и не привязан к конкретному времени, что намного удобнее в большинстве случаев

**Кэширование на стороне браузера** — это метод хранения веб-ресурсов на локальном устройстве пользователя (например, HTML, CSS, JavaScript, изображения) для уменьшения времени загрузки страниц и снижения нагрузки на сервер. Это помогает ускорить повторные посещения веб-сайтов, делая их более отзывчивыми

1. HTTP: сохранение состояния на стороне клиента.

Смелов:

**Сохранить состояние можно в URI и в Cookie**

**Cookie** - фрагмент данных ключ-значение, который м.б. сохранен на стороне клиента по указанию сервера. Set-Cookie: имя-значение, path (шаблон URI). Браузер в качестве ключа берет шаблон URI. Теперь браузер обязан по URI из path вставить заголовок Cookie.

**Куки** (cookies) — это фрагменты данных в формате "ключ-значение", которые веб-сервер отправляет клиенту (браузеру). Браузер сохраняет эти данные и отправляет их обратно на сервер при каждом последующем запросе на тот же домен. Это позволяет серверу «запоминать» информацию о пользователе между запросами.

Используется для аутентификации, хранения пользовательских предпочтений, статистики, **информации о сеансе** (обычно Session ID). Обычно имеет имя, содержащее URL, может иметь срок действия.

**Основные атрибуты куки**

1. **Имя и значение (name=value):**
   * Это основные данные куки. Например: user=JohnDoe.
2. **Path (путь):**
   * Определяет путь на сервере, для которого куки будут доступны. Например: **path=/mypath** означает, что куки будут отправляться только для запросов, начинающихся с **/mypath**.
   * Этот путь должен быть абсолютным, то есть начинаться с **/**.
3. **Domain (домен):**
   * Определяет домен, для которого куки будут доступны. Например: **domain=site.com**.
   * Существует ограничение: мы не можем указать произвольный домен. Например, куки, установленные для **example.com**, не будут доступны для **another.com**.
4. **Expires и Max-Age:**
   * Expires: указывает дату истечения куки. Например: expires=Tue, 19 Jan 2038 03:14:07 GMT.
   * Max-Age: указывает количество секунд, через которое куки истекут. Например: max-age=3600 означает, что куки будут жить один час.
   * Если ни один из этих параметров не указан, куки будут удалены при закрытии браузера. Такие куки называются сессионными (session cookies).

**Пример установки куки**

Сервер может установить куки, отправив заголовок **Set-Cookie** в ответе:

* **user=JohnDoe:** имя и значение куки.
* **path=/mypath:** куки будут доступны только для запросов, начинающихся с **/mypath**.
* **domain=site.com:** куки будут доступны только для домена **site.com**.
* **expires=Tue, 19 Jan 2038 03:14:07 GMT:** куки истекут в указанную дату.
* **secure:** куки будут передаваться только по защищённому соединению (HTTPS).
* **HttpOnly:** куки не будут доступны через JavaScript (только для HTTP-запросов).

Когда браузер получает ответ с заголовком **Set-Cookie**, он сохраняет куки и будет автоматически отправлять их в каждом последующем запросе к соответствующему домену и пути.

1. HTML: стандарты, BOM, DOM, CSS, HTML 5 API.

HTML (от англ. HyperText Markup Language — «язык гипертекстовой разметки») — стандартизированный язык разметки веб-страниц. Код HTML интерпретируется браузерами; полученная в результате интерпретации страница отображается на экране монитора компьютера или мобильного устройства. Спецификации HTML5 формулируются в терминах DOM (объектной модели документа).

**Стандарты**

**HTML** 4. Эта версия ввела возможность использования CSS

**XHTML 1.0.** HTML основанный на XML

**XHTML 1.1.** Удаление устаревших элементов и атрибутов**.** Поддержка модульного подхода для включения только необходимых элементов.

**HTML 5.** Введение новых семантических элементов: <header>, <footer>, <article>, <section>, <nav>, <aside> и другие. Адаптивный дизайн, API, встроенная поддержка аудио и видео

**Структура HTML-страницы**

HTML-документ состоит из различных элементов, которые организованы в логическую структуру. Основные части HTML-документа:

1. **<!DOCTYPE html>**: Определяет тип документа и версию HTML.
2. **<html>**: Корневой элемент, который содержит все другие элементы.
3. **<head>**: Секция, содержащая метаданные документа (например, заголовок страницы, ссылки на стили и скрипты).
4. **<title>**: Определяет заголовок документа, отображаемый в заголовке браузера.
5. **<body>**: Основная часть документа, содержащая весь контент, который отображается пользователю.

**модель DOM (W3C):** представление HTML-документа web-браузером, интерфейс JavaScript для доступа к содержимому HTML-документа

**Модель DOM**: три уровня DOM0, DOM1, DOM2, DOM3; современные браузеры уровня 2 с элементами уровня 3.

**Модель DOM (Document Object Model)** DOM представляет собой объектную модель, которую браузеры используют для представления и взаимодействия с HTML-документами. DOM позволяет JavaScript манипулировать содержимым, структурой и стилями документов.

**Основные характеристик**и объектной модели:

• основана на валидном HTML-коде;

• может быть модифицирована из JavaScript;

• не включает псевдоэлементы, созданные из CSS;

• включает скрытые элементы (display: none).

Он представляет документ как дерево объектов, что позволяет программам и скриптам изменять структуру, стиль и содержимое документа.

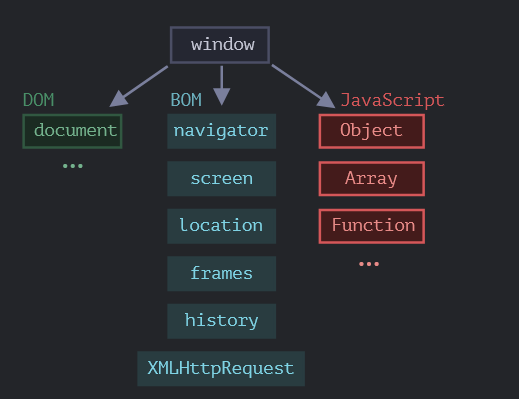
Основные аспекты DOM:

1. **Дерево узлов**: HTML-документ представлен как дерево узлов, где каждый узел соответствует элементу, атрибуту или тексту в документе.
2. **Доступ к элементам**: С помощью DOM можно получить доступ к элементам HTML, изменять их свойства, содержимое и стили.
3. **События**: DOM позволяет назначать и обрабатывать события, такие как клики, загрузка страницы, изменения формы и другие взаимодействия пользователя.

\*\***Модель BOM (Browser Object Model)** \*\*BOM предоставляет интерфейсы для взаимодействия с браузером вне контекста HTML-документа. BOM включает объекты, такие как **window**, **navigator**, **screen**, **location**, и **history**.

[BOM (Browser Object Model)](https://learn.javascript.ru/browser-environment#bom-browser-object-model)

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



Как мы видим, имеется корневой объект window, который выступает в 2 ролях:

1. Во-первых, это глобальный объект для JavaScript-кода, об этом более подробно говорится в главе [Глобальный объект](https://learn.javascript.ru/global-object).
2. Во-вторых, он также представляет собой окно браузера и располагает методами для управления им.
3. Объект [navigator](https://developer.mozilla.org/en/DOM/window.navigator) содержит общую информацию о браузере и операционной системы. Особенно примечательны два свойства:navigator.userAgent– содержит информацию о браузере иnavigator.platform содержит информацию о платформе, позволяет различать Windows/Linux/Mac и т.п.
4. Объект [location](https://developer.mozilla.org/en-US/docs/Web/API/Window.location) содержит информацию о текущем URL страницы и позволяет перенаправить посетителя на новый URL.
5. Функции alert/confirm/promptтоже входят в BOM.

**CSS** **(Cascading Style Sheets)** – это язык стилей, используемый для описания внешнего вида и форматирования документов, написанных на языках разметки, таких как HTML. Основная задача CSS – отделить содержимое документа от его представления, что позволяет улучшить управление стилями

Селектор – это часть CSS-правила, которая сообщает браузеру, к какому элементу (или элементам) веб-страницы будет применён стиль.

Стиль — это совокупность правил, применяемых к элементу гипертекста и определяющих способ его отображения. Стиль включает все типы элементов дизайна: шрифт, фон, текст, цвета ссылок, поля и расположение объектов на странице.

Таблица стилей — это совокупность стилей, применимых к гипертекстовому документу.

Каскадирование — это порядок применения различных стилей к веб-странице. Браузер, поддерживающий таблицы стилей, будет последовательно применять их в соответствии с приоритетом: сначала связанные, затем внедренные и, наконец, встроенные стили. Другой аспект каскадирования — наследование (inheritance), — означает, что если не указано иное, то конкретный стиль будет применен ко всем дочерним элементами гипертекстового документа. Например, если вы примените определенный цвет текста в теге <div>, то все теги внутри этого блока будут отображаться этим же цветом.

Использование каскадных таблиц дает возможность разделить содержимое и его представление и гибко управлять отображением гипертекстовых документов путем изменения стилей.

**Методы Размещения CSS на Странице**

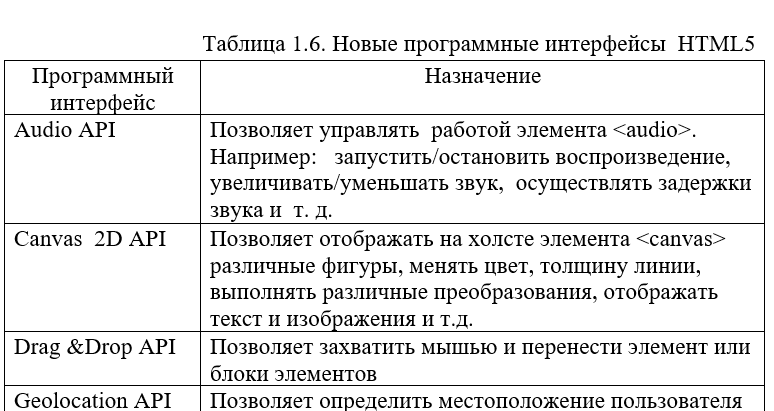
1. **Встроенные стили (Inline Styles)**: <p style="color: red;">Это красный текст.</p>
2. **Внутренние стили (Internal Styles)**: Определяются в секции **<style>** внутри **<head>** HTML-документа.
3. **Внешние стили (External Styles)**: Определяются в отдельном CSS-файле, который подключается к HTML-документу с помощью тега **<link>**

**Селекторы CSS**

Селекторы используются для выбора элементов HTML, к которым будут применены стили. Существует множество различных типов селекторов.

1. **Селекторы по тегу**:
2. **Селекторы по классу**:
3. **Селекторы по идентификатору (ID)**:
4. **Комбинированные селекторы**:

HTML API (Application Programming Interface) предоставляет методы и свойства для работы с HTML-документами через JavaScript. API включает в себя функции для взаимодействия с элементами, управления их поведением и изменениями.





Некоторые важные HTML API:

1. **Element API**: Позволяет взаимодействовать с HTML-элементами, изменять их атрибуты, содержимое и стиль.
2. **Event API**: Позволяет управлять событиями, такими как клики, загрузка страницы, изменение содержимого и т.д.
3. **Form API**: Специфический для работы с HTML-формами, включает методы и свойства для управления полями ввода, отправкой форм и валидацией данных.
4. **Canvas API**: Предоставляет методы для рисования и анимации на веб-странице.
5. **Web Storage** **API** предлагает разработчику локальное хранилище (local storage) и сессионное хранилище (session storage). localStorage сохраняет данные без срока действия, а sessionStorage - только на время текущей сессии.
6. AJAX: асинхронный запрос, объекты XMLHTTPREQUEST и FETCH, форматы передачи данных.

**!!!ИЗ ЛЕКЦИИ: AJAX – это методология (принципы и правила написания кода) разработки, которая основывается на возможности объекта XMLHttpRequest выдавать асинхронные запросы**

**AJAX – асинхронный JavaScript and XML – методология (подход) построения динамических приложений, которая основана на возможности XMLHttpRequest выдавать асинхронные запросы. Следствие: не осуществляется полная перезагрузка html-страниц. AJAX: XMLHTTPRequest, DOM, формат: XML и JSON.  XMLHttpRequest не относится к BOM, DOM, но в рамках браузера.**

Метод open (метод, URI, bool async | sync)

Send метод отправляет заявку.

Получить ответ: в свойство onChange записать ссылку на callback функцию, которая будет вызываться браузером в зависимости от 4 состояний. На 4 стадии можно обрабатывать сам ответ. Можно использовать для progress-bar.

Из книжки смелова

Основной недостаток AJAX заключается в том, что он основывается на протоколе HTTP, который требует работы в режиме «запрос-ответ». Другими словами, получить порцию данных от сервера можно только сделав к нему запрос. В приложениях, основная задача которых реагировать на события, происходящие на сервере, клиентской части приложения приходится периодически выполнять запросы к серверу, чтобы «увидеть» эти события. Для того, чтобы «не пропустить» серверное событие и своевременно на него отреагировать, приходится увеличивать частоту запросов. Это приводит к загрузке сетевого трафика и нерациональному расходу ресурса сервера, который вынужден постоянно отвечать на присланные клиентом запросы.

**AJAX**: Asynchronous JavaScript and XML – асинхронный JavaScript and XML – методология (подход) построения динамических приложений, при которых не осуществляется полная перезагрузка html-страниц. AJAX: XMLHTTPRequest, DOM, формат: XML и JSON.

**В основе методологии Ajax лежат следующие технологии:** язык HTML, язык JavaScript, язык XML, модель DOM, протокол HTTP, протокол JSON, объект XMLHttpRequest.

СТРУКУТУРА:

**XML –** расширяемый язык разметки данных. Предназначен для структуризации данных с целью хранения или/и передачи. В Ajax язык XML является одним из форматов, который используется для структуризации данных пересылаемых между JavaScript-сценарием и серверным приложением.

**JSON (JavaScript Object Notation)** - текстовый формат обмена данными, применяемый обычно в сценариях JavaScript**.** В Ajax формат JSON является одним из форматов, который используется для структуризации данных пересылаемых между JavaScript сценарием и серверным приложением. Формат JSON основывается на функции **eval()** языка JavaScript.

**XMLHttpRequest**– **браузерный объект,** специальный API (предопределенный объект), используемый в языке JavaScript для обмена данными между сценарием JavaScript и серверным приложением по протоколу HTTP. В Ajax методы объекта XMLHttpRequestиспользуется для отправки и получения данных между JavaScript-сценарием и серверным приложением. Данные могут получены в виде XML-документа и виде обыкновенного текста (в частном случае могут быть представлены в формате JSON).

*Структура AJAX*

Для каждой части страницы должен создавать свой объект xmlHttpRequest (он живёт в браузере).

Чтобы сделать запрос, нам нужно выполнить три шага:

1. Создать XMLHttpRequest.

let xhr = new XMLHttpRequest();

1. Инициализировать его.

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

Заметим, что вызов open, вопреки своему названию, не открывает соединение. Он лишь конфигурирует запрос, но непосредственно отсылается запрос только лишь после вызова send.

1. Послать запрос.

xhr.send([body])

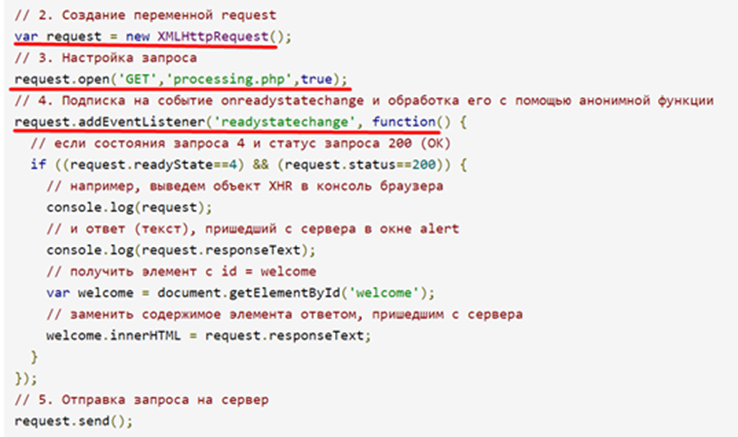
Этот метод устанавливает соединение и отсылает запрос к серверу. Необязательный параметр body содержит тело запроса.

1. Слушать события на xhr, чтобы получить ответ.

Три наиболее используемых события:

* load – происходит, когда получен какой-либо ответ, включая ответы с HTTP-ошибкой, например 404.
* error – когда запрос не может быть выполнен, например, нет соединения или невалидный URL.
* progress – происходит периодически во время загрузки ответа, сообщает о прогрессе.

У XMLHttpRequest есть состояния, которые меняются по мере выполнения запроса. Текущее состояние можно посмотреть в свойстве xhr.readyState.



А как теперь получить ответ, когда мы отравили запрос? Любая асинхронная обработка всегда устроена так, что обработка ответов осуществляется с помощью функции обратного вызова (функция callback).

**Методы объекта XmlHttpRequest**

* getAllResponseHeaders() — получить все заголовки ответа от сервера.
* getResponseHeader(«имя\_заголовка») — получить указаный заголовок.
* setRequestHeader(«имя\_заголовка»,«значение») — установить значения заголовка запроса.
* open(«тип\_запроса»,«URL»,«асинхронный»,«имя\_пользователя»,«пароль») — инициализация запроса к серверу, указание метода запроса. Тип запроса и URL — обязательные параметры. Третий аргумент — булево значение. Обычно всегда указывается true или не указывается вообще (по умолчанию — true). Четвертый и пятый аргументы используются для аутентификации (это очень небезопасно, хранить данные об аутентификации в скрипте, так как скрипт может посмотреть любой пользователь). (вызывается первым после создания XMLHttpRequest, вызов open – не открывает соединение. Он настраивает запрос.)
* send(«содержимое») — послать HTTP запрос на сервер и получить ответ. (открывает соединение и отправляет запрос на сервер.)
* abort() — отмена текущего запроса к серверу.

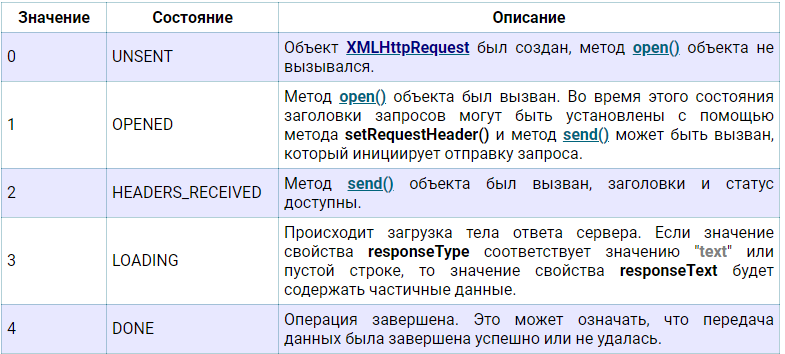
**Свойства объекта XmlHttpRequest**

* Status – http-код ответа.
* StatusText – текстовое описание http-кода.
* ResponseText – текст ответа сервера.
* onreadystatechange — одно из самых главных свойств объекта XMLHttpRequest. С помощью этого свойства задаётся обработчик, который вызывается всякий раз при смене статуса объекта.
* readyState — число, обозначающее статус объекта.

Ответ с сервера приходит в JSON.

5 состояния запроса:

* 0 – начальное состояние
* 1 – вызван open
* 2 – получены заголовки ответа
* 3 – загружается тело
* 4 – запрос завершён



Асинхронная функция – это функция, после вызова которой JavaScript приложение продолжает работать, потому что функция сразу выполняет возврат. Результат работы асинхронной функции становится известным позже, и для того, чтобы оповестить наше приложении о полученных значениях, асинхронная функция вызывает другую функцию (callback), которую мы передаем в аргументах при запуске.

Fetch в себе содержит [XMLHttpRequest](https://ru.wikipedia.org/wiki/XMLHttpRequest).

let promise = fetch(url, [options]),

где url - URL для отправки запроса,

options - дополниельные параметры: метод, заголовки и так далее.

**Функция обратного вызова: простыми словами:** коллбэк — это функция, которая должна быть выполнена после того, как другая функция завершила выполнение (отсюда и название: callback – функция обратного вызова).

**Чуть сложнее:** В JavaScript функции — это объекты. Поэтому функции могут принимать другие функции в качестве аргументов, а также функции могут возвращать функции в качестве результата. Функции, которые это умеют, называются **функциями высшего порядка**. А любая функция, которая передается как аргумент, называется **callback-функцией**. Чтобы лучше разобраться, давайте посмотрим на примерах, как это выглядит.

В С++ передать указатель на функцию.В С# дилегат.

**Fetch API**: Это современный API для выполнения сетевых запросов в браузере. Он предоставляет более гибкий и мощный способ выполнения запросов по сравнению с XMLHttpRequest

fetch() - это метод. Возвращает promise.

Promise - это js-объект, представляющий результат асинхронной операцию.

let promise = fetch(url, [options])

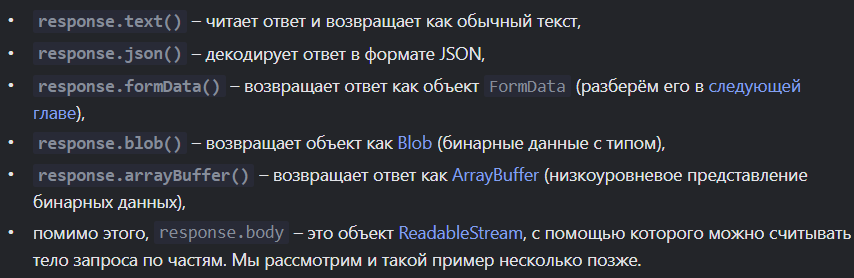
* **url** – URL для отправки запроса.
* **options** – дополнительные параметры: метод, заголовки и так далее.

Без options это простой GET-запрос, скачивающий содержимое по адресу url.

Процесс получения ответа обычно происходит в два этапа.

**Во-первых, promise выполняется с объектом встроенного класса**[**Response**](https://fetch.spec.whatwg.org/#response-class)**в качестве результата, как только сервер пришлёт заголовки ответа.** На этом этапе мы можем проверить статус HTTP-запроса и определить, выполнился ли он успешно, а также посмотреть заголовки, но пока без тела ответа.

**Во-вторых, для получения тела ответа нам нужно использовать дополнительный вызов метода.** Response предоставляет несколько методов, основанных на промисах, для доступа к телу ответа в различных форматах:



Мы можем выбрать только один метод чтения ответа. Если мы уже получили ответ с response.text(), тогда response.json() не сработает, так как данные уже были обработаны.

fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits')

.then(response => response.json())

.then(commits => alert(commits[0].author.login));

let response = await fetch('/article/fetch/post/user', {

**method**: 'POST',

**headers**: {

'Content-Type': 'application/json;charset=utf-8'

},

body: JSON.stringify(user)

});

**Промисы в JavaScript** — это объекты, представляющие завершение или неудачу асинхронной операции и ее результат.

**Методы промиса**:

* **then()**: Метод для обработки успешного выполнения промиса. Принимает два аргумента: колбэк для обработки успешного результата и колбэк для обработки ошибки (необязательный).
* **catch()**: Метод для обработки ошибок. Принимает один аргумент: колбэк для обработки ошибки.
* **finally()**: Метод, который вызывается после завершения промиса (успешно или с ошибкой). Принимает один аргумент: колбэк, который будет выполнен в любом случае.

fetch API возвращает промис, что делает обработку асинхронных HTTP запросов удобной и более читаемой. Использование промисов с fetch позволяет легко обрабатывать как успешные ответы, так и ошибки.

1. Протокол WebSocket: назначение, применение..

Протокол передачи данных, основанный на TCP, для установки длительного соединения и обмена сообщениями между клиентом и сервером в режиме дуплексной связи. Находится на прикладном уровне модели OSI.

Соединение устанавливается следующим образом:

1.  Клиент посылает обычный HTTP-запрос, называемый рукопожатием, с заголовками Upgrade и Connection (Connection: Upgrade, Upgrade: websocket)

2.  Сервер отвечает HTTP-ответом со статусом 101 (Switching Protocols), показывая, что он согласен переключить протокол на WebSocket. Ответ сервера содержит заголовок Upgrade: websocket и ключи для подтверждения протокола (например, Sec-WebSocket-Accept), сгенерированные на основе ключа, отправленного клиентом в заголовке Sec-WebSocket-Key.

Если не поддерживает, то наверно 400 Bad Request

3. Сразу после отправки сервером ответа,  TCP-соединение остается открытым, клиент и сервер могут начинать двунаправленный обмен сообщениями  по этому же TCP-соединению.

При этом протокол определяет две URL-схемы: ws – для незашифрованного (80 порт дефолт) соединения и  wss – для зашифрованного (443 порт дефолт).

Применения WebSockets

WebSockets находят применение в:

* **Чат-приложениях**: Обеспечивают моментальный обмен сообщениями между пользователями.
* **Реального времени обновлениях**: Например, в приложениях для отслеживания фондовых рынков или спортивных событий.

**Онлайн-играх**: Обеспечивают низкую задержку и высокий уровень взаимодействия между игроками.

Протокол определяет управляющие кадры, такие как "ping" и "pong", для поддержания соединения и проверки его состояния.

WebSocket — протокол связи поверх TCP-соединения,предназначенный для обмена сообщениями между браузером и веб-сервером ~~в режиме реального времени~~.

Дуплексный протокол – 2 однонаправленных протокола в разные стороны, которые работают независимо друг от друга.

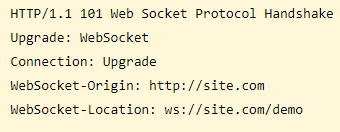
Изначально синхронный HTTP протокол, построенный по модели «запрос — ответ», становится **полностью асинхронным и симметричным**. Теперь уже нет клиента и сервера с фиксированными ролями, а есть два равноправных участника обмена данными. Каждый работает сам по себе, и когда надо отправляет данные другому. Одна сторона отправит данные и продолжит работу дальше, ничего ждать не надо. Вторая сторона ответит, когда захочет — может не сразу, а может и вообще не ответит.

Подключение:

Браузер подключается по протоколу TCP на 80 порт сервера и дает немного необычный GET-запрос:



Если сервер поддерживает ВебСокеты, то он отвечает таким образом:



Когда сервер и клиент послали **handshake** запросы, и проверка пройдена, то начинается этап обмена данными.

Если браузер это устраивает, то он просто оставляет TCP-соединение открытым.

Как только одна сторона хочет передать другой какую-то информацию, она отправляет дата-фрейм следующего вида:



То есть просто строка текста — последовательность байт, к которой спереди приставлен нулевой байт 0x00, а в конце — 0xFF. И все — никаких заголовков, метаданных! Что именно отправлять, разработчики полностью оставили на ваше усмотрение: хотите XML, хотите JSON.

С помощью WebSockets так же можно передавать и бинарные данные и картинки.

1. Web-приложение: определение, назначение, архитектура, применение.

**Веб-приложения** — клиент-серверное приложение, в котором клиент взаимодействует с сервером по протоколу HTTP.

**HTTP** - протокол прикладного уровня, который нужен для того чтобы описывать правила. Эти правила нужны для того, чтобы клиент и сервер могли пересылать друг другу сообщения. HTTP прослушивается через 80 порт или 443 порт – это HTTPS.

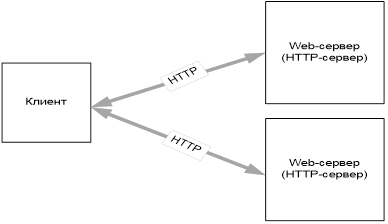
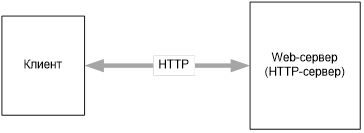
**Порт** — программа на сервере, которая прослушивает входящие сообщения.

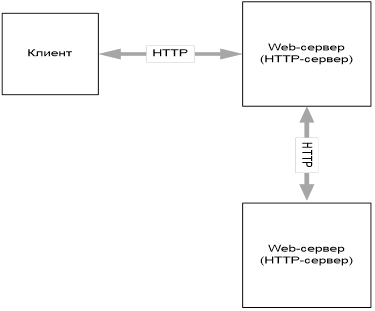
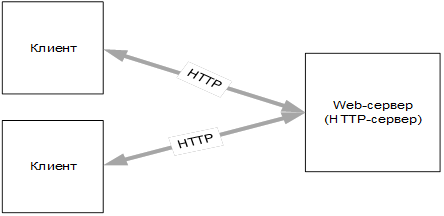
Web-приложения представляют собой особый тип программ, построенных по архитектуре "клиент-сервер". Особенность их заключается в том, что само Web-приложение находится и выполняется на сервере - клиент при этом получает только результаты работы. Работа приложения основывается на получении запросов от пользователя (клиента), их обработке и выдачи результата. Передача запросов и результатов их обработки происходит через Интернет.

На стороне сервера Web-приложение выполняется специальным программным обеспечением (Web-сервером), который и принимает запросы клиентов, обрабатывает их, формирует ответ в виде страницы, описанной на языке HTML, и передает его клиенту

**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.

**Web ресурсы приложения**: **статические** - отправляются клиенту без изменения (html-страницы, рисунки, видео-файлы, …), динамические – **динамически** (программно) формируются на сервере и отправляются клиенту (сервлеты, JSP, http-обработчики, aspx-страницы,…). Ресурс может быть статическим относительно сервера и динамическим относительно клиента (html-страницы с JavaScript).





1. Web-сервер: основные объекты.

Основные объекты:

* запросы и ответы;
* соединение(сессия);
* контекст приложения;
* фильтры;
* кэш (данных и вывода);
* слушатели событий;

**!!!Порядок создания объектов сервера:**

Контекст => listener => filter. Затем уже сессия и д.р.

**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.

1. Web-сервер: объекты Request и Response.

**Запрос(Request):** серверныйобъект, который образуется в результате обработки сервером http-запроса, поступающего от клиента и передается серверному программному коду для обработки. Содержит: всю информацию из http-запроса: метод, коллекция заголовков, коллекция параметров, поток данных … Обычно объект Request предоставляет возможность хранить данные в формате ключ/значение.

**HTTP-драйвер преобразует последовательность битов HTTP-запроса в объект Request.**

**Ответ(Response):** серверныйобъект, который автоматически формируется сервером, при получении http-запроса (одновременно с объектом Request), заполняется данными серверными программным кодом, преобразуется в http-ответ и отправляется клиенту. Содержит: всю информацию, которая должна быть помещена в http-ответ: статус, коллекция заголовков, поток данных, …

1. Web-сервер: объект Session и его жизненный цикл.

**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).

Каждый Requestпринадлежит, какой-то сессии (имеет ссылку на объект Session или содержит Session ID). sessionID, соответствует серии запросов, идентифицирует серию запросов. Запросы в рамках одной сессии (сеанса).

Обычно объект Session предоставляет приложению возможность хранить данные в формате ключ/значение.

! Как сервер определяет начало сессии: если нет куки – значит начало сессии; если есть куки в них есть sessionID, но он просрочен, то начинается новая сессия, новый Set-Cookie. Сервер может убить куки если надо.

**жизненный цикл**

создается при первом запросе пользователя (при этом генерируется session ID), умирает в конце срока действия (Если **timeout** превышен) или Пользователь может явно завершить сессию, например, выйдя из системы (logout).

1. Web-сервер: сохранение состояния.

3 уровня хранилища

1. **В Request**(самая малая, инфа хранится в рамках 1 запроса)
2. В Session (информация между запросами в рамках 1 сессии)
3. В Context (информация между сессиями)

**Персистентность** – система умеет сохранять свое состояние. Обычно обеспечивается контекстом, но есть системы, которые сохраняют и в сессии

1. Web-сервер: объект Filter/Middleware его жизненный цикл.

Filter = Middleware. Это конвейер обработки request и response.

**Filter** - серверный объект, препроцессор запроса, предназначен для предварительной обработки объекта Request.

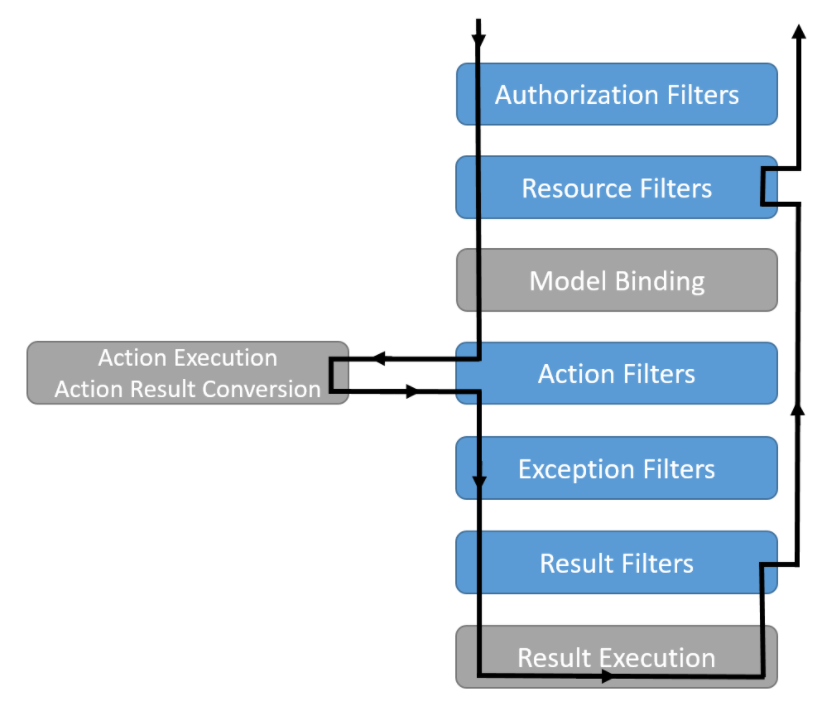
К одному ресурсу может быть построена цепочка фильтров, последний в цепочке – ресурс. Фильтр может прервать цепочку и сам сформировать ответ клиенту. Один и тот же фильтр может быть применен к нескольким ресурсам.

В качестве параметров фильтр получает объекты Request и Response, которые от передает дальше по цепочке или обрывает цепочку и заполняет объект Response.

Допустим первый чекает аутентифицирован ли, второй чекает валидность данных, которые передаются в запросе, третий что-нибудь тоже. Могут по ходу работы изменять ответ, добавляя/удаляя заголовки. Можно для кеширования.

**Типы фильтров (фильтр каждого типа выполняется на определенном этапе конвейера фильтров):**

* Фильтры *авторизации*: определяют, авторизован ли пользователь для выполнения текущего запроса. Если пользователь не авторизован для доступа к ресурсу, то фильтр завершает обработку запроса.
* Фильтры *ресурсов*: выполняются после фильтров авторизации. Его метод OnResourceExecuting() выполняется до всех остальных фильтров и до привязки модели, а его метод OnResourceExecuted() выполняется после всех остальных фильтров
* Фильтры *действий*: применяется только к действиям контроллера, запускается после фильтра ресурсов как до, так и после выполнения метода контроллера
* Фильтры *исключений*: определяют действия в отношении необработанных исключений
* Фильтры *результатов действий*: фильтр применяется к результатам методов контроллера и страниц Razor Pages, выполняется как до, так и после получения результата



**Жизненный цикл**

1. Инициализация: Фильтры создаются на этапе инициализации приложения.

2. Обработка запроса

3. Обработка ответа

4. Объект фильтра или промежуточного слоя в веб-приложении обычно существует на протяжении всего времени работы приложения

1. Web-сервер: объект Context его жизненный цикл.

**Контекст web-приложения**: серверный объект, предназначенный для хранения информации об одном 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

**Контекст** – серверный объект, который хранит информацию об одном web-приложении, считывается с config, переменных окружения. В разных системах может называться по-разному: ASP – HttpApplication, Java – ServletContext, Nodejs – process. Резидентный объект. В нем есть хранилище (сохраняется пока сервер запущен).

Обычно контекст предоставляет возможность хранить данные в формате ключ/значение.

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

**жизненный цикл**

формируется сразу при загрузке web-сервера

резидентный — находится в оперативной памяти (на протяжении работы сервера)

1. Web-сервер: принципы кэширования.

**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.(Результаты запросов к базе данных, данных, полученных из внешних AP)

**Кэширование вывода** – кэширование объекта Response. (Вместо повторной генерации полного ответа при каждом запросе, сервер может хранить готовые объекты ответа (например, HTML-страницы, JSON-ответы) и отдавать их напрямую из кэша.)

Smelov:

**Кеширование** – перемещение данных из менее быстрой памяти с более быструю память.

**Кеширование** выполняется на стороне сервера. Ресурс может обрабатываться долго. Кешируем response. Не во всех системах допустим кеш.

**Cache Manager** – приходит запрос, запускает обработку, проходит обработку, запоминает response в хэш хранилище по ключу (самый простой вариант – URL) и указываем сколько оно будет хранится.

Когда запоминаем, мы говорим сколько времени должно храниться. Например, 5 сек. После кеш самоочищается. Это **время абсолютное**.

**Время скользящее** – когда отсчитывается время между запросами. Если в течение 5 сек приходит – то сдвигается ещё на 5 сек.

Составные ключи: URI + параметры и их значения, может ещё заголовок и его значение

1. Web-сервер: объект connection pool и его жизненный цикл.

**Пул соединений с базой данных -** несколько  предварительно и  постоянно открытых соединений с  сервером СУБД, которые используют приложения.  Выбор подключения из пула по Open, возврат в пул Close. Если все подключения пула заняты, запрос на соединение ставится в очередь. Применение пула позволяет увеличить производительность за счет отсутствия процесса подключения к серверу.

Альтернатива: держать постоянное открытое соединение для каждого подключения (в общем случае для web-приложения не приемлемо) или открывать/закрывать соединение при каждом запросе (большие накладные расходы на установку соединения).

Смелов: Пул соединений: сервер сразу должен установить min кол-во соединений. Можно: сделать через Singleton, записать в контекст или листенер. Может расти до max, затем выстраивается в очередь. Ограничивает кол-во соединений.

!! нужно сказать, что при создании пула сразу открываются min кол-во соединений. (что они именно открываются, а не создаются)

**Жизненный цикл Connection Pool:**

1. **Инициализация**: На этом этапе создается пул соединений, обычно в начале работы сервера или при первом обращении к базе данных. Количество соединений в пуле определяется параметрами конфигурации и требованиями приложения.
2. **Запрос соединения**: Когда приложение нуждается в выполнении операции, требующей соединения с базой данных (например, выполнение SQL-запроса), оно запрашивает свободное соединение из пула.
3. **Выдача соединения**: Пул соединений предоставляет приложению доступное соединение из своего пула. Если все соединения заняты, пул может создать новое соединение, если это допускается настройками, или ждать, пока не освободится соединение.
4. **Использование соединения**: После получения соединения приложение выполняет необходимые операции, такие как выполнение запросов к базе данных.
5. **Возврат соединения в пул**: После завершения операции приложение возвращает соединение обратно в пул, чтобы оно могло быть использовано другими запросами.
6. **Разрыв соединения**: В некоторых случаях соединение может быть разорвано из-за истечения тайм-аута, ошибок сети или других причин. Пул соединений может периодически проверять соединения на активность и пересоздавать их при необходимости.
7. **Завершение работы**: При завершении работы сервера или приложения пул соединений освобождает все ресурсы, связанные со собой, включая активные соединения, и завершает свою работу.
8. Web-сервер: объект listener и его жизненный цикл.

**Listener** - серверные объекты – для обработки событий жизненного цикла web-приложения.

Не всегда существует явно.

События которые обрабатывает:

* Загрузка сервера
* Завершение работы сервера
* Создание сессии
* Завершение сессии
* Авторизация
* аутентификация

**жизненный цикл**

Создётся при инициализации сервера, данные для создания берутся из контекста

Умирает при выключении сервера

1. Web-браузер: структура, принципы функционирования, основные объекты.

Браузер - прикладное программное средство, предназначенное:

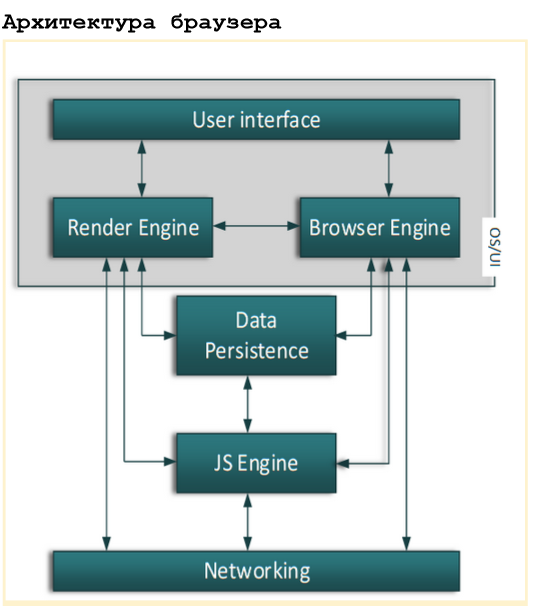
- формирования и выполнения http-запросов;

- 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

**Browser API**: Geolocation/HTML5 – программный интерфейс позволяющий определить географические координаты месторасположения пользователя

**Browser API**: Web Storage/HTML5 – программный интерфейс позволяющий сохранять данные домена (URL) в хранилище данных (Data Persistence) браузера (Local Storage, Session Storage).

**Browser API**: Drag&Drop/HTML5 – программный интерфейс позволяющий пользователю захватить мышью элемент и перенести).

Основные объекты **Browser Object Model**: window, navigator,  history, location, screen, document.

<https://itchief.ru/javascript/bom#osnovnye-obekty-bom>

1. Web-браузер: назначение и возможности IndexDB API.

**IndexDB/HTML5** - программный интерфейс для работы с встроенной в  браузер NoSQL БД.

**IndexedDB:** СУБД, встроенная в web-browser, NoSQL, система индекированных хранилищ, формат данных JSON, JavaScript API, асинхронная работа, транзакционная модель, поддерживается курсор, использует события DOM об уведомлении javascrpt-приложения (error/succes).

В отличие от других механизмов хранения, таких как cookies, localStorage и sessionStorage, IndexedDB предназначен для работы с большим объемом данных и поддерживает более сложные запросы и индексацию.

**Объектное хранилище** (Object Store)– аналог таблицы в реляционной базе данных. Объектные хранилища используются для хранения записей и создаются при инициализации базы данных.

**Итого**

* СУБД, встроенная в веб-браузер
* NoSQL база данных
* IndexedDB организует данные в **объектные хранилища** (object stores), которые можно сравнить с таблицами в реляционных базах данных.
* Каждое хранилище может иметь индексы, которые можно использовать как фильтр
* Данные в IndexedDB хранятся в формате JSON
* Доступ к IndexedDB осуществляется через JavaScript API
* Все операции в IndexedDB выполняются асинхронно
* поддерживает транзакции
* поддерживает курсоры
* IndexedDB использует события DOM для уведомления JavaScript-приложений об успешном завершении операций или возникновении ошибок. Основные события включают onsuccess и onerror
* предназначен для работы с большим объемом данных
* у IndexedDB есть версии и изменения структуры бд (например, добавить новое объектное хранилище или индекс) приводят к изменению версии. При открытии базы данных с новой версией запускается событие **onupgradeneeded**,

let request = **indexedDB.open**("myDatabase", 1);

request.onupgradeneeded = function(event){

…

// Создаем объектное хранилище

let objectStore = **db.createObjectStore**("myObjectStore", { keyPath: "id" });

// Создаем индекс по полю name

**objectStore.createIndex**("name", "name", { unique: false });

}

request.onsuccess = function(event) {

…

// Открываем транзакцию для записи данных

let transaction = db.**transaction**(["myObjectStore"], "readwrite");

// Получаем объектное хранилище

let objectStore = **transaction.objectStore**("myObjectStore");

// Добавляем данные

let addRequest = **objectStore.add**({ id: 1, name: "John Doe", age: 30 });

// Получаем данные по ключу

let getRequest = **readObjectStore.get**(1);

addRequest.onsuccess = function(event) {

console.log("Данные успешно добавлены");

};

};

**Возможности**:  
 JS API, проверка браузера

* создание и модификация БД
* запись в БД
* читать из БД по идентификатору
* удалить из БД по идентификатору
* читать из БД с помощью курсора без индекса
* читать из БД по индексу с помощью курсора, режимы  (IDBKeyRange)

1.в Chrome можно увидеть в режиме ***Просмотреть код/Application/Storage/IndexedDB***.

1. Протокол OAuth 2.0: назначение, принципы применения.

**OAuth 2.0** — это открытый стандарт для авторизации, который позволяет предоставлять сторонним приложениям ограниченный доступ к ресурсам пользователя без необходимости предоставления учетных данных (логина и пароля). OAuth 2.0 широко используется для интеграции различных веб-приложений и сервисов.

Стандарт OAuth 2.0 определяет следующие четыре роли:

* **владелец ресурса** - сущность, обладающая правом на выдачу доступа к защищенным ресурсам. В случае, если владелец является человеком, его называют конечным пользователем;
* **сервер ресурсов** - сервер, содержащий защищаемые ресурсы и обладающий возможностью получения и формирования ответа на запросы к защищаемым ресурсам посредством использования маркера доступа;
* **клиент** - приложение, осуществляющее доступ к защищенным ресурсам от имени Владельца. Термин "клиент" явно не определяет какое-либо конкретное исполнение (будь то сервер, персональный компьютер или мобильное приложение);
* **сервер авторизации** - сервер, осуществляющий выпуск маркеров доступа для клиентских приложений после успешной аутентификации и авторизации Владельца ресурсов.



**OAuth 2.0: Authorization Code (используется с серверными приложениями), Implicit (используется мобильными или веб-приложениями), Resource Owner Password Credentials (используются доверенными приложениями), Client Credentials (используются при доступе приложения к API).**

**Client отправляет запрос** на доступ к требуемому ресурсу resource owner.

**Resource owner передаёт** обратно клиенту authorization grant, который подтверждает личность resource owner и его права на ресурс, доступ к которому запрашивает client. В зависимости от flow это может быть токен или учётные данные.

**Client отправляет authorization grant, полученный** в предыдущем шаге authorization server, ожидая от него Access token для доступа к защищённому ресурсу.

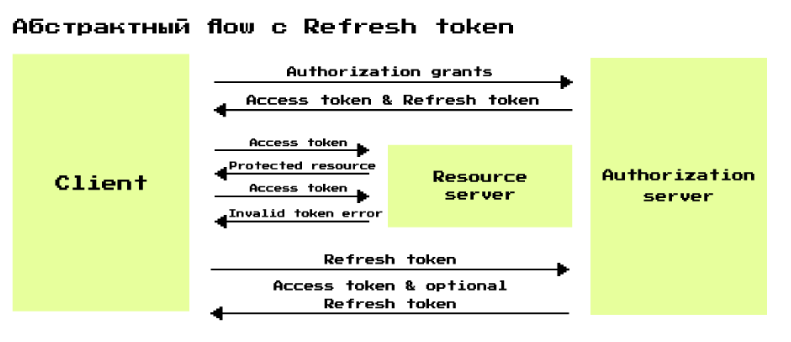
**authorization server убеждается** в валидности authorization grant, после чего отсылает access token клиенту в ответ.

**Получив access token**, клиент запрашивает защищённый ресурс у resource server.

**Resource server убеждается** в корректности access token, после чего предоставляет доступ к защищённому ресурсу.

А будет ли так же просто, если мы добавим в эту схему работу с **refresh token**? Первый и второй шаги опущены из данной схемы — они ничем не отличаются от схемы абстрактного flow выше.

<https://selectel.ru/blog/oauth-2/>



**Client приходит c authorization grant** к authorization server и просит предоставить ему access token и refresh token.

**Authorization server** убеждается, что с authorization grant всё нормально и возвращает клиенту запрошенные access token и refresh token.

**Client с access token** запрашивает защищённый ресурс, пока не получит первую ошибку доступа к ресурсу — invalid token error.

После получения ошибки доступа, клиент идет к authorization server с refresh token и просит заменить просроченный access token на новый.

**В ответ клиент получает новый access token**, а также новый refresh token, либо продлевается время жизни старого refresh token.

**Grant** — это данные, которые представляют из себя успешную авторизацию клиента владельцем ресурса, используемые клиентом для получения access token.

Есть 4 этих гранта-способа:

1. Authorization Code Flow
2. Implicit Flow
3. Resource Owner Password Credentials Flow
4. Client Credentials Flow
5. 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).

WebDAV: альтернатива FTP, SMB.

**Всего 39 метода, но уже 40**

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

WebDAV: в Windows WebDAV API C++.

WebDAV: унаследованные HTTP-методы

GET – скачать файл.

PUT – загрузить файл на сервер.

DELETE – удалить серверный объект.  
12 методов

PROPFIND – получить свойства серверного объекта(ресурса) в XML-формате

PROPPATCH – изменить свойства серверного объекта.

MKCOL – создать папку на сервере

COPY – копировать на сервере

MOVE – переместить на сервере.

LOCK – заблокировать серверный объект

UNLOCK – разблокировать серверный объект.

**Назначение**

**Управление документами**: WebDAV позволяет пользователям создавать, изменять и удалять документы на сервере, обеспечивая удаленный доступ к файлам.

1. WebRTC: назначение, протоколы и их назначение, API.

**В конспекте норм написано**

**протокол прикладного уровня, для передачи потоковых данных (видео, аудио) P2P.**

**WebRTC (Web Real-Time Communication)** — это технология, позволяющая реализовывать потоковую передачу аудио, видео и данных в реальном времени между браузерами и другими устройствами через Интернет без использования плагинов или сторонних программ.

WebRTC - основан на 2 протоколах RTP и RTCP.

**Назначение WebRTC:**

1. **Видеосвязь и аудиосвязь**: Позволяет организовывать видео- и аудио-конференции прямо в браузере без дополнительных программ.

2. **Потоковая передача данных**: Позволяет обмениваться данными между браузерами в реальном времени для различных сценариев, таких как игры, стриминг и другие веб-приложения.

**Перечень протоколов, используемых в WebRTC:**

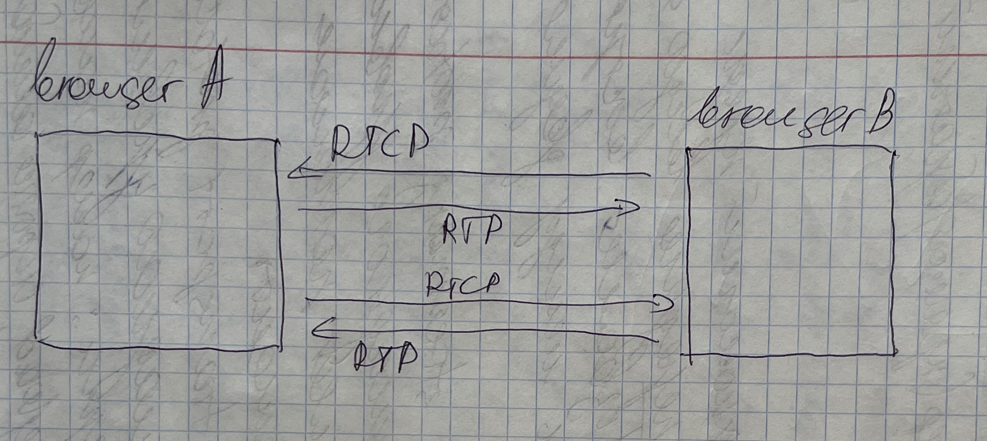
1. **ICE (Interactive Connectivity Establishment)**: Протокол, используемый для обнаружения и выбора наиболее эффективного пути для установления соединения между устройствами.

2. **STUN (Session Traversal Utilities for NAT)**: Протокол, позволяющий определить публичный IP-адрес и порт клиента, находящегося за NAT.

3. **TURN (Traversal Using Relays around NAT)**: Протокол, используемый для обеспечения связности передачи данных между клиентами, находящимися за различными типами NAT.

4. **DTLS (Datagram Transport Layer Security)**: Протокол, обеспечивающий безопасную передачу данных в реальном времени посредством шифрования и аутентификации.

5. **SRTP (Secure Real-time Transport Protocol)**: Протокол, обеспечивающий безопасную передачу аудио- и видео-данных в реальном времени.



**WEBRTC:** технология передачи потоковых данных:

-   JS API, W3C;

-   **RTSP:** протокол прикладного уровня, только управление серверным потоком данных; методы:   play(старт вещания), приостановка(pause) проигрывания мультимедийного содержимого,  запись на содержимого на сервере(record), …. Команды посылаются отдельно от мультимедийного потока, использует порт 554.

-   **RTP**: протокол прикладного уровня, в заголовке: номер пакета, временная метка,  кодировка (JPEG, MPEG, …),  формат полезной нагрузки, алгоритмы скрытия ошибок,  динамический порт (четный), следующий нечетный RTCP, над UDP.

-   **RTCP**: передача информации о задержках, передача метрик качества;

-   SDP: сигнальный протокол;

-   STUN: прокотол прикладного уровня, позволяет определить внешний IP клиента, скрытого за NAT, необходим для установки соединения между двумя точками, скрытыми за NAT.

-   TURN: протокол, позволяющий обеспечивать обмен потоковыми данными между клиентами, скрытыми за NAT.

-   аудиокодеки:G.711, Opus; видеокодеки:VP8, H.264.

**WEBRTC:**

-   WEBRTC-решения не совместимы, условно нельзя позвонить из Skype на Google Meet;

-   нельзя скрыть IP.

1. Сервер NGINX: назначение и применение.

В конспекте норм

NGINX — это высокопроизводительный HTTP-сервер и обратный прокси-сервер, а также почтовый (IMAP/POP3) прокси-сервер. Он был создан для решения проблем, связанных с масштабированием и производительностью, и стал одним из самых популярных серверов в мире.

**Назначение:**

HTTP-сервер: NGINX может работать как полноценный веб-сервер, обслуживая статические файлы, обрабатывая динамические запросы через CGI, FastCGI, SCGI, uWSGI и другие протоколы

forward proxy(прямой прокси, кэширование, скрыть ip-адрес клиента), reverse proxy (обратный прокси, балансировка нагрузки

  обслуживание статических запросов

  обратное проксирование

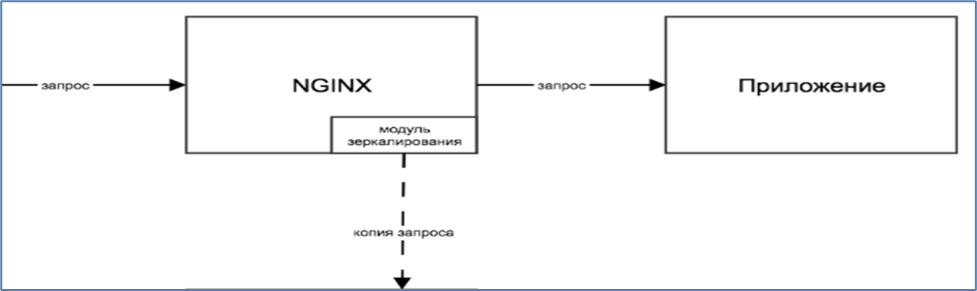
**Применение NGINX**

* **Обслуживание статического контента**: NGINX отлично справляется с доставкой статических файлов, таких как изображения, видео и другие медиафайлы.
* **Обратное проксирование и балансировка нагрузки**: Обеспечивает равномерное распределение запросов между несколькими серверами для повышения производительности и отказоустойчивости.
* **Кэширование**: Уменьшает нагрузку на серверы приложений и ускоряет доставку контента пользователям.
* **Безопасность**: Позволяет использовать SSL/TLS терминаторы для шифрования трафика, а также настраивать защиту от DDoS-атак и других угроз.
* **Мультипротокольная поддержка**: Может работать с различными протоколами и модулями для расширения функциональности.

Статику нужно держать отдельно.

Nginx: forward proxy, reverse proxy, кеширование балансировка нагрузки.

Зеркалирование: отправить запрос в два места.



обслуживаниестатических запросов

обратное проксирование

поддержка SSL/TLS

HTTP/2 с приоритизацией на основе весов и зависимостей.

**процессы**

-  главный процесс (один): чтение конфигурационного файла, управление рабочими процессами;

-  рабочие процессы (конфигурация, по умолчанию количество равно числу ядер): обработка http-запросов.

Зав. каф. ИСиТ В.В. Смелов