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

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.
4. **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).

**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. Сервер + протокол. Стандартные серверы, которые прослушивают стандартные порты (от 0 до 1024).

другое название Internet-сервис, один из видов Internet-ресурса, имеющий специальное назначение (DNS, WWW, E-mail, FTP, ICQ, Telnet).

1. Основные организации управления сетью Интернет.
2. **ISOC: Internet Society** – международная организация (офисы США, Швейцария), занимающаяся развитием сети Internet. ISOC владелец RFC-стандартов. ISOC обеспечивает правовую поддержку и финансирует все другие организации, связанные с деятельностью Internet (IETF, IAB,…).
3. **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.
4. **IAB**: **Internet Architecture Board** - совет по архитектуре Internet, одна из комиссий IETF, имеет консультативный статус при ISOC.
5. **ICANN: Internet Corporation for Assigned Names and Numbers** – корпорация по управлению доменными именами и IP-адресами.
6. **IANA: Internet Assigned Numbers Authority –** Администрация адресного пространства Internet. Под контролем ICANN. Кроме того регистрирует типы данных **MIME**.
7. **W3C: World Wide Web Consortium –** организация разрабатывающая и внедряющая web-стандарты (HTTP, HTML, URI/URL, CSS, DOM, XML, PNG, SVG,…).
8. Протокол HTTP. Основные свойства, версии и их особенности.

HTTP - **полудуплексный протокол прикладного уровня**.

передачи данных, изначально — в виде гипертекстовых документов в формате HTML, в настоящее время используется для передачи произвольных данных

Является **ассиметричным** (т.е. сообщения от клиента к серверу отличаются от сообщений сервера клиенту)

Образует **полудуплексный канал** – в каждый момент времени сообщение передается или в одну, или в другую сторону.

*TCP-дуплексный  
Web-Socket – дуплексный*

Cтруктура:

* 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

**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.

читать инфу в конце в вопросах про методы статус коды переадресацию.

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 ресурса, описывает множество URI.

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

**Структура и компоненты URI:**

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

**2)Authority.** 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, 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).

**3)Path (Путь).** 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.

**4)Query (Запрос).** Компонента запроса содержит данные, организованные в неиерархической форме, которые, вместе с данными в иерархическом компоненте пути (Path), служит, чтобы идентифицировать ресурс в рамках схемы URI и authority (если таковая компонента указана).

**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.

**Стуруктура URN:**

*В п.2 RFC3305 сообщается об устаревании такого термина* ***как URL****, применимо к ссылкам, и что отныне верным будет именование URI, с того момента, во всех документах W3C использует термин URI.*

*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.*

***15. 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.

является стандартом, который описывает природу и формат документа, файла или набора байтов.

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

Изначально есть 2 типа заголовков:

1)Определены в HTTP

2)Кастомные

Заголовки делятся на 4 типа:

1. **General**: общие заголовки, используются в запросах и ответах;
2. **Request**: используются только в запросах;
3. **Response**: используются только в ответах;
4. **Entity:** для сущности в ответах и запросах.

**Date** основной 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 просто "HTTP over TLS", т.е. "HTTP/TLS". Это все тот же старый HTTP, просто он обернут в TLS, который является своего рода защитным слоем сверху, обеспечивающим зашифрованную связь.

Здесь связь между браузером и сервером отличается тем, что она начинается с **TLS Handshake**.

Короче рассказать что безопаснее и про рукопожатие и TLS

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

протокола безопасности транспортного уровня (TLS). TLS позволяет приложениям клиент-сервер взаимодействовать через Интернет таким образом, чтобы предотвратить подслушивание, вмешательство и подделку сообщений.

новое название **Transport Layer Security**

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

Cipher Suites – шифронаборы; в шифронабор входит: сказал смелов хеш-функция, алгоритм шифрования, и как вырабатывать ключ

* **Сообщение "hello" клиента:** Клиент инициирует рукопожатие, посылая серверу сообщение "hello". В сообщении будет указано, какую версию TLS поддерживает клиент, поддерживаемые наборы шифров и строка случайных байтов, известная как "random клиента".
* **Сообщение "hello" сервера:** В ответ на сообщение "hello" клиента сервер отправляет сообщение, содержащее [SSL-сертификат](https://www.cloudflare.com/learning/ssl/what-is-an-ssl-certificate/) сервера, выбранный сервером набор шифров и "random сервера" – еще одну строку случайных байтов, генерируемую сервером.
* **Аутентификация:** Клиент проверяет SSL-сертификат сервера в центре сертификации, который его выдал. Он подтверждает, что сервер является тем, за кого себя выдает, и что клиент взаимодействует с реальным владельцем домена.
* **Premaster secret:** Клиент посылает еще одну строку случайных байтов, которая называется "premaster secret". Premaster secret шифруется открытым ключом и может быть расшифрован сервером только с помощью закрытого ключа. (Клиент берет [открытый ключ](https://www.cloudflare.com/learning/ssl/how-does-public-key-encryption-work/) из SSL-сертификата сервера).
* **Используемый закрытый ключ:** Сервер расшифровывает "premaster secret".
* **Создание сеансовых ключей:** Клиент и сервер генерируют сеансовые ключи из random'а клиента, random'а сервера и premaster secret. Они должны прийти к одинаковым результатам.
* **Клиент готов:** Клиент отправляет сообщение "готово", зашифрованное сеансовым ключом.
* **Сервер готов:** Сервер отправляет сообщение "готово", зашифрованное сеансовым ключом.
* **Безопасность симметричного шифрования достигнута:** Рукопожатие завершено, и связь продолжается с использованием сеансовых ключей.

Смелов

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

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

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

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

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

**Сервер** выбирает подходящую длину ключа.

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

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

1. Протокол TLS: X509-сертификаты, назначение.

**Сертификат X.509:** стандартныйформат хранения и транспортировки атрибутов безопасности; главное – открытый ключ; сертификаты выдают центры сертификации

сертификат содержит: имя держателя, адрес, серийный номер сертификата, даты проверки, открытый ключ держателя.

**Сертификат X.509:** Выдает центр сертификации (СА).

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](https://en.wikipedia.org/wiki/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.

**1. Клиент делает запрос к защищенному ресурсу**: Клиент пытается получить доступ 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==**.

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.

**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** предлагает клиенту пройти аутентификацию с использованием дайджест-аутентификации и отправляет клиенту необходимую информацию. 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**: Одноразовая строка, выданная сервером для предотвращения повторных атак.
3. **Клиент** вычисляет значение ответа и отправляет его вместе с именем пользователя, областью, URI, nonce, opaque, qop, nc и cnonce. Много всего. Или Клиент генерирует хеш из различных элементов, включая имя пользователя, пароль, **nonce**, URI и некоторые другие параметры.
4. **Сервер** вычисляет хэш самостоятельно и сравнивает их. Если они совпадают, он передает клиенту запрошенные данные.

**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.

Forms (нет стандарта, но применяется)

Алгоритм работы аутентификации через веб форму входа примерно следующий:

* пользователь вводит данные на веб-страницу и они передаются, обычно, методом POST на веб-сервер
* веб-сервер передаёт принятые данные веб-приложению
* веб-приложение сверяет присланные учётные данные с хранящимися в базе данных
* если имя пользователь и пароль верны, то пользователю присылается маркер (токен) любого вида, который позволяет отличить пользователя, и даётся указание веб-браузеру сохранить его в кукиз
* веб-брауезр сохраняет этот токен в cookie
* сайт также запоминает токен и пользователя, которому он назначен
* при каждом последующем запросе веб-браузер отправляет, среди прочих HTTP заголовков, и токен из cookie
* 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

**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.

1. HTTP: принципы кэширование на стороне клиента.

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

HTTPS не кэшируется

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.**

Идея заключается в том, что 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.

Таким образом, используя Last-modified мы экономим на загрузке большого файла, отделываясь пустым быстрым ответом от сервера.

**Принцип работы Etag** очень 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, в случае, 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.

Сохранить состояние можно в URI и в Cookie

path=/mypath

URL-префикс пути, куки будут доступны для страниц под этим путём. Должен быть абсолютным. По умолчанию используется текущий путь.

domain=site.com

Домен определяет, где доступен файл куки. Однако на практике существуют определённые ограничения. Мы не можем указать здесь какой угодно домен.

Expires и max=age: по умолчанию, если куки не имеют ни одного из этих параметров, то они удалятся при закрытии браузера. Такие куки называются сессионными («session cookies»). Чтобы помочь куки «пережить» закрытие браузера, мы можем установить значение опций expires или max-age.

Смелов:

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.

**IndexedDB**:

* **Описание**: Встроенная база данных в браузере, которая позволяет сохранять большие объемы структурированных данных.
* **Использование**: Хранение сложных данных, таких как объекты, файлы и другие ресурсы.

**LocalStorage**:

* **Описание**: Встроенное хранилище в браузере, которое позволяет сохранять данные на стороне клиента без ограничения времени.
* **Использование**: Хранение данных, которые должны быть доступны между сессиями браузера, например, пользовательские настройки.

**SessionStorage**:

* **Описание**: Встроенное хранилище в браузере, которое сохраняет данные только на время текущей сессии браузера (пока вкладка открыта).
* **Использование**: Хранение данных, которые должны быть доступны только в течение текущей сессии, например, временные настройки или состояния форм.

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

SGML главный от него все пошли и XML HTML

**Первая и** базовая версия HTML была разработана для поддержки основных функций, таких как текстовые элементы управления или размещение изображений.

**HTML 2.0** - это версия, имевшая все возможности HTML 1.0 с некоторыми новыми функциями для веб-дизайна. HTML 2.0 считался стандартной версией HTML до 1995 года. В этой версии были улучшены теги разметки, текстовых полей, кнопок и т. д. Браузеры, разработанные для этой версии HTML, также ввели концепцию наличия тегов и слоев.

**HTML 3.0** предоставила авторам HTML и веб-мастерам больший контроль и более широкий спектр способов разметки текста и повышения качества и внешнего вида веб-сайтов.

Новая версия началась **с HTML 4.0,** известной как Cougar. Но со временем некоторые улучшения были изменены и вошли в версию HTML 4.01.

В 1999 году была представлена новая версия HTML 4.01, которая была значительно усовершенствована и предназначалась для достижения более масштабных целей. **Расширенные версии HTML 4.01 поддерживали каскадные таблицы стилей (CSS).** Представленная концепция таблиц стилей решила проблему наличия CSS на каждой веб-странице и убирала повторяющийся код.

Большинство разработчиков ожидали, что после HTML 4.0 и HTML 4.01 следующим будет HTML 5.0. Но следующим в линейке стандартов стал **XHTML.** Новый XHTML сопровождался появлением улучшенных тегов и опций, которые очень понравились разработчикам. Аббревиатура XHTML расшифровывается как Расширяемый язык гипертекстовой разметки (англ. Extensible HyperText Markup Llanguage). При этом цель запуска XHTML заключалась вовсе не в улучшении тегов. Основная причина выхода этого стандарта состояла в том, чтобы улучшить взаимодействие с новыми браузерами, которые постоянно меняют динамику просмотра.

**Самая последняя и современная версия - HTML 5.** В ней поддерживаются все теги и другие элементы, такие как элементы ввода различных типов, теги поддержки, геолокацию и пр. Основная цель внедрения HTML 5 состояла в том, чтобы удовлетворить две вещи - улучшить язык и соответствовать новейшим разработкам в области мультимедиа.

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

*Спецификация CSSOM*

*Описывает файлы стилей, правила написания стилей и манипуляций с ними, а также то, как это всё связано*

**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

*По сути, уровень DOM просто представляет собой что-то вроде релизной версии. Это может быть связано с его эволюцией: последующие уровни (спецификации) строятся на предыдущих, добавляются какие-то новые модули, методы и поведения... до тех пор, пока он не остановился на DOM 4-го уровня. Стандарт больше не описывается как набор слоев, а как моментальный снимок живой спецификации.*

содержит описание JavaScript (ECMAScript)

**DOM** – это объектное представление исходного HTML-документа, попытка преобразовать его структуру и содержимое в объектную модель, с которой смогли бы работать различные программы.

**В ближайшем приближении DOM** – это «дерево узлов» (node tree). У него единый корень, который разветвляется на множество дочерних ветвей, каждая из которых может ветвиться сама и заканчивается «листьями». Корень – это элемент html, а ветви – вложенные элементы.

DOM всегда валидна, поэтому в процессе создания браузер может поправить некоторые ошибки исходного кода, например, добавить пропущенный tbody или закрыть какой-нибудь незакрытый тег.

DOM позволяет не только просматривать содержимое страницы, но и взаимодействовать с ним, изменять. Это не статичное отображение, а живой ресурс.

**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).

**CSS(Cascading Style Sheets)** – каскадные таблицы стилей, CSS1, CSS2, CSS3 (текущий уровень), CSS4(в разработке с 2011).

**CSS-фреймворки**: Bootstrap, Kube, Foundation, Semantic UI, PureCSS.

**CSS-расширения**: Sass, SCSS, LESS

CSS стандартизируется организацией World Wide Web Consortium (W3C), которая устанавливает рекомендации по использованию CSS

**1. Селекторы:** Селекторы определяют, к какому элементу или группе элементов на странице применяются стили. Существуют различные типы селекторов, включая селекторы типов, классов, идентификаторов, атрибутов, псевдоклассов и псевдоэлементов.

**2. Декларации:** Декларации состоят из свойства и его значения, разделенных двоеточием. Набор деклараций, заключенных в фигурные скобки, формирует блок деклараций.

**3. Свойства:** Свойства — это способы, с помощью которых можно стилизовать элементы. Например, color, font-size, margin, border и background являются свойствами в CSS.

**4. Значения свойств:** Каждое свойство имеет значение, которое определяет, как оно будет применено к элементу. Например, color: red; устанавливает цвет текста элемента в красный.

**5. Каскадность:** Каскадность означает, что при конфликте нескольких правил стилей применяется то, которое имеет наибольший приоритет. Приоритет определяется на основе специфичности селекторов, важности (!important) и порядка их объявления.

**6. Наследование:** Некоторые свойства CSS наследуются потомками от их родительских элементов, что означает, что если вы установите свойство для родителя, оно может быть применено и к дочерним элементам.

**7. Медиа-запросы:** Медиа-запросы позволяют применять различные стили в зависимости от типа устройства, его характеристик или размера экрана, что является ключевым аспектом адаптивного веб-дизайна.

**8. Фреймворки и препроцессоры:** Существуют CSS-фреймворки (например, Bootstrap) и препроцессоры (например, Sass или Less), которые предоставляют дополнительные возможности и упрощают написание и поддержку CSS.

1. AJAX: асинхронный запрос, объекты XMLHTTPREQUEST и FETCH, форматы передачи данных.

**Асинхронные запросы**: запрос, при котором поток, выдавший http-запрос, не блокируется до поступления запроса; для обработки ответа применяется функция обратного вызова.

**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.

**XMLHttpRequest** умеет делать запросы на сервер асинхронно, то есть без блокировки выполнения других операций веб-страницы. Это означает что отправка запроса на сервер не задерживает выполнение остальных операций на странице.

Вместо остановки и блокировки остальных операций на странице, асинхронные запросы работают в фоновом режиме. Это означает, что страница может продолжать свою работу и не останавливается в ожидании ответа от сервера.

Объект XMLHttpRequest встроен в браузер.

// 1. Создаём новый XMLHttpRequest объект new XMLHttpRequest();

// 2. Настраиваем его: GET-запрос по URL /users xhr.open('GET', 'https://hexlet.io/users');

// 3. Отсылаем запрос на получение пользователей xhr.send();

// 4. Этот код сработает после того, как мы получим ответ сервера xhr.onload

**Метод fetch** позволяет асинхронно отправлять сетевые запросы на сервер и получать ответы. Этот подход не блокирует основной поток выполнения, что означает, что во время отправки запроса и ожидания ответа приложение может продолжать работу без задержек. Fetch предоставляет более современный и гибкий способ выполнения HTTP-запросов и обработки полученных данных.

Например для выполнение простейшего запроса с использованием XMLHttpRequest нам необходимо написать минимум десять строчек кода, в то время как с fetch всего несколько.

Метод fetch предоставляет более простой и лаконичный синтаксис для выполнения HTTP запросов. Короче просто синтаксис удобнее и быстрее писать.

fetch('https://hexlet.io/users') (1)

.then(response => response.json()) // (2)

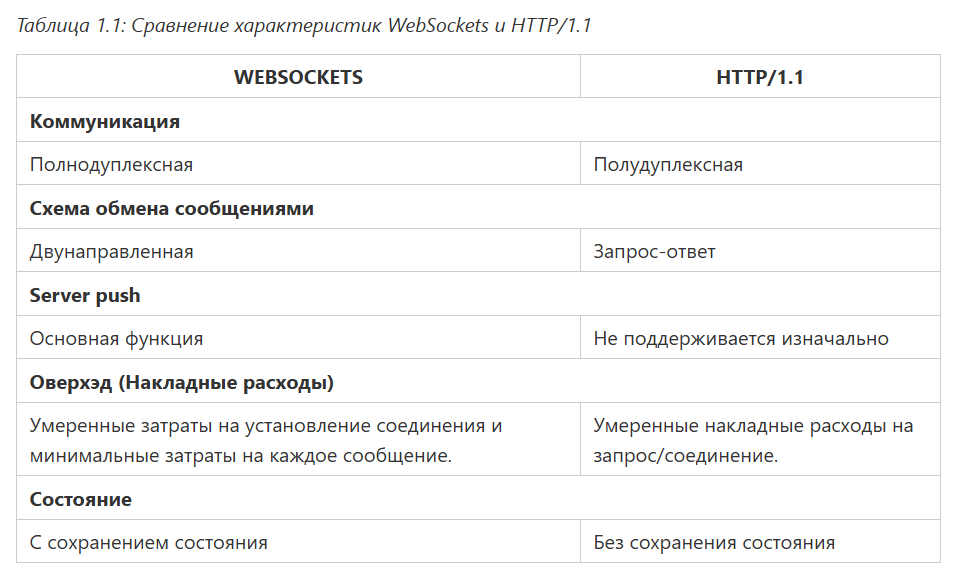
.then(users => console.log(users)) // (3)

.catch(error => console.log('Произошла ошибка:', error)); //(4)

1. Протокол WebSocket: назначение, применение.

**WebSocket** — протокол двунаправленной связи между браузером и веб-сервером. Протокол включает в себя описание запроса клиента и ответа сервера на установление соединения, а также базовое оформление сообщений, передаваемых поверх TCP-соединения.

* **Протокол WebSocket**. Обеспечивает связь между клиентами и серверами через Интернет и поддерживает передачу двоичных данных и текстовых строк.
* **API WebSocket**. Позволяет выполнять необходимые действия, такие как управление соединением WebSocket, отправка и получение сообщений, а также прослушивание событий, инициируемых сервером



Как работает протокол WebSocket

**Установление соединения:** Чтобы установить соединение WebSocket, клиент отправляет на сервер HTTP-запрос, содержащий заголовок "Upgrade", с просьбой переключиться с традиционного протокола HTTP на протокол WebSocket. Если сервер поддерживает протокол WebSocket, он отвечает кодом состояния HTTP 101, подтверждающим переключение протокола.

**Кадрирование данных:** После того, как соединение установлено, данные могут быть отправлены в обоих направлениях в виде «кадров» без необходимости повторного установления соединения. Протокол WebSocket определяет два типа фреймов данных: текстовые и двоичные. Текстовые фреймы содержат текстовые данные в кодировке UTF-8, в то время как двоичные фреймы могут содержать любые произвольные двоичные данные.

**Обработка сообщений:** Каждый кадр, отправленный через соединение WebSocket, может быть либо полным сообщением, либо частью более крупного сообщения. Протокол определяет управляющие кадры, такие как "ping" и "pong", для поддержания соединения и проверки его состояния.

**Завершение соединения:** Соединение WebSocket остается открытым до тех пор, пока клиент или сервер не отправят контрольный кадр "close" или не прервут базовое TCP-соединение. Это обеспечивает эффективную коммуникацию без накладных расходов, связанных с многократным открытием и закрытием соединений

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

**Web-программирование:** разработка клиент-серверных приложений, компоненты которого взаимодействуют по протоколу HTTP-протокола; web-программирование – частный случай программирования в Internet

**Клиент – Сервер**

**Клиент – Посредник Сервер – Сервер**

**Клиенты – Сервер**

**Клиент – Сервера**

**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).

* web-ресурсы приложения;
* запросы и ответы;
* соединение(сессия);
* конфигурационный файл приложения;
* контекст приложения;
* фильтры;
* кэш (данных и вывода);
* слушатели событий;
* принципы безопасности.

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

1. Web-сервер: основные объекты.

Про все что ниже рассказать

1. Web-сервер: объекты Request и Response.

**Запрос (Request)** - серверный объект, 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.

**Ответ (Response)** - серверный объект, 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. Описывает соединение на сеансовом уровне.

*TCP – помнит свое состояние(помнит номер пакета)*

*UDP – нет*

Cоздается при первом обращении к серверу.

Как сервер определяет: 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.

**Время жизни: timeout** (системный параметр, обычно равен 10 – 30 минутам) – максимальное время между запросами клиента.  Session timeout – скользящее время.

**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 Request принадлежит, 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.

У каждого соединения своя сессия.

1. Web-сервер: сохранение состояния.

По смелову проверка на сохранение состояния:  
Объявить переменную на уровне программного объекта и инициализировать ее в 1 запросе, если при 2 запросе значение то же, то сохраняет.

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

Есть всего 3 хранилища.

**В Request**(меньше всего, инфа хранится в рамках 1 реквеста)

В Session (информация между реквестами в рамках 1 сессии)

В Context (информация между сессиями)

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

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

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.

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

**Контекст** создается при загрузке сервера из конфигурационного файла, то чего нет ставится по дефолту или из переменных окружения

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

Контекст – 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.

Как правило, формируется сразу при загрузке web-сервера.

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

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. Кэширование вывода – кэширование объекта Response.

**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 хранилище. Запоминает по ключу (самый простой вариант – URL).

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).

**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.

Слушатели событий (Lister): серверные объекты – для обработки событий жизненного цикла web-приложения

**1. Загрузка конфигурации:** При запуске приложения конфигурационный контекст Context инициализируется значениями из конфигурационного файла. Это может включать настройки для веб-сервера, баз данных, логирования и других сервисов.

**2. Инициализация слушателей**: На основе данных из Context инициализируются объекты Listener. Например, если Context содержит информацию о портах и протоколах, которые должны слушать слушатели, то эти данные используются для настройки Listener-ов.

**3. Запуск слушателей:** После инициализации Listeners запускаются и начинают прослушивание входящих сетевых запросов в соответствии с конфигурацией, полученной из Context.

**4. Жизненный цикл слушателей:** Listeners продолжают работать во время выполнения приложения, принимая запросы и передавая их для обработки в соответствующие компоненты системы.

**5. Остановка слушателей**: При завершении работы приложения или при необходимости изменения конфигурации слушатели могут быть остановлены и повторно инициализированы с новыми параметрами из Context.

*В контексте, где `Listener` обрабатывает события жизненного цикла веб-приложения, он не обязательно связан с прослушиванием сетевых запросов, как это было описано ранее. Вместо этого, `Listener` может относиться к компоненту, который "слушает" или отслеживает события жизненного цикла приложения, такие как его запуск, остановка и так далее.*

*В таком случае, жизненный цикл `Listener` может выглядеть следующим образом:*

*1****. \*\*Инициализация:\*\**** *При старте приложения инициализируются `Listener`-ы, которые могут быть настроены для отслеживания определенных событий жизненного цикла приложения.*

*2.* ***\*\*Подписка на события:\*\**** *`Listener` подписывается на интересующие его события жизненного цикла приложения, используя, например, `IHostApplicationLifetime` в ASP.NET Core или аналогичные механизмы в других фреймворках.*

*3****. \*\*Обработка событий:\*\**** *Когда происходит событие, на которое подписан `Listener`, он выполняет определенные действия, такие как логирование, запуск или остановка вспомогательных сервисов, оповещение других компонентов системы и т.д.*

*4.* ***\*\*Остановка и отписка:\*\**** *При завершении работы приложения или при его перезапуске `Listener` может быть корректно остановлен, отписаться от событий и освободить ресурсы.*

*5.* ***\*\*Завершение работы:\*\**** *После остановки приложения `Listener` завершает свою работу, и его экземпляр уничтожается.*

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.

Части:

**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 перенсти).

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).

**IndexedDB:** стандарт W3C.

indexDB.open();

CreateObjectStore(); DeleteObjectStore();

Запись: инициализировать objectStore, к объекту add. Удаление так же, только delete

1. Протокол WebDAV: назначение, принцип применения.

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

WebDAV: альтернатива FTP, SMB.

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

WebDAV: GET – скачать файл.

WebDAV: PUT – загрузить файл на сервер.

WebDAV: DELETE – удалить серверный объект.  
12 методов

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

**WebRTC (Web Real-Time Communication)** — это набор технологий, позволяющих 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:**

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

-   нельзя скрыть IP.

1. Сервер NGINX: назначение и применение.

**это HTTP-сервер общего назначения**

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

поддержка SSL/TLS

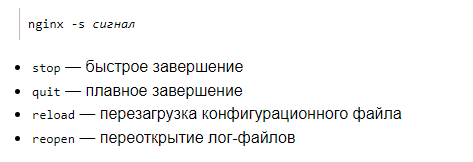
обслуживание статических запросов

обратное проксирование

поддержка http2 и 3

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

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

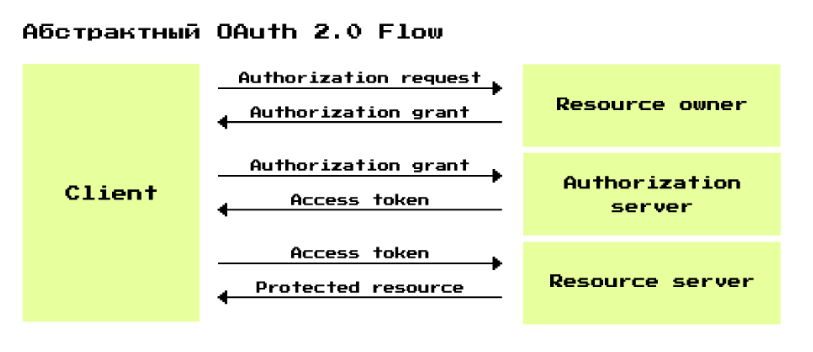
****

1. Протокол OAuth 2.0: назначение, принципы применения.

**OAuth2.0** — это фреймворк авторизации, позволяющий сторонним разработчикам подключаться к API от имени клиента или владельца ресурса

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.



**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.

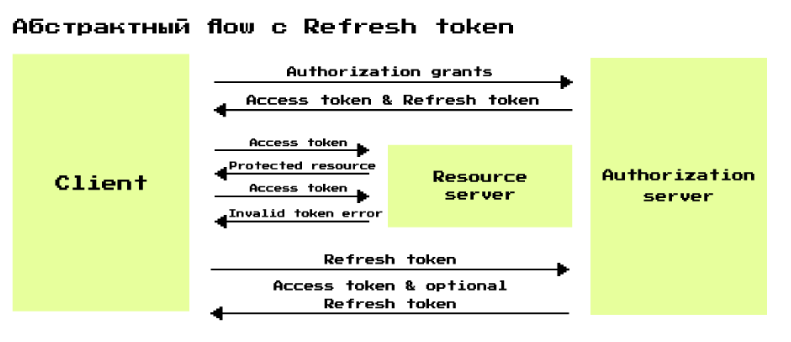
**Client отправляет authorization grant, полученный** в предыдущем шаге authorization server, ожидая от него Access token для доступа к защищённому ресурсу.

**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.



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

Происходит**: клиент отправляет запрос серверу -> сервер отвечает клиенту на этот запрос, устанавливая статус код класса 3xx и заголовок Location, где указывает новую ссылку -> клиент видя, что статус-код 3xx отправляет исходный свой запрос, заменяя там старый URI на URI из Location**

**Код состояния 301** (перемещено навсегда) указывает на то, что [целевому ресурсу](https://httpwg.org/specs/rfc9110.html#target.resource) назначен новый постоянный URI и любые будущие ссылки на этот ресурс должны использовать один из вложенных URI. Исторически сложилось так, что агент пользователя МОЖЕТ изменить метод запроса с POST на GET для последующего запроса. Если такое поведение нежелательно, вместо него можно использовать код состояния [308](https://httpwg.org/specs/rfc9110.html#status.308)

**Код состояния 302** (Найдено) указывает на то, что целевой ресурс временно находится под другим универсальным кодом ресурса (URI). Так как перенаправление может быть изменено в некоторых случаях, клиент должен продолжать использовать целевой URI для будущих запросов. Исторически сложилось так, что агент пользователя МОЖЕТ изменить метод запроса с POST на GET для последующего запроса. Если такое поведение нежелательно, вместо него можно использовать код состояния [307](https://httpwg.org/specs/rfc9110.html#status.307)

**Код состояния 307** (временное перенаправление) указывает на то, что целевой ресурс временно находится под другим универсальным кодом ресурса (URI) и агент пользователя НЕ ДОЛЖЕН изменять метод запроса, если он выполняет автоматическое перенаправление на этот URI. Так как перенаправление может меняться со временем, клиент должен продолжать использовать исходный целевой URI для будущих запросов. **Сервер ДОЛЖЕН сгенерировать поле заголовка Location** в ответе, содержащее ссылку URI для другого URI. Агент пользователя МОЖЕТ использовать значение поля Location для автоматического перенаправления.

**Код состояния 308** (постоянное перенаправление) указывает на то, что целевому ресурсу назначен новый постоянный URI и любые будущие ссылки на этот ресурс должны использовать один из вложенных URI. Сервер предполагает, что агент пользователя с возможностью редактирования ссылок может навсегда заменить ссылки на целевой URI одной из новых ссылок, отправленных сервером. Однако это предложение обычно игнорируется, если только агент пользователя активно не редактирует ссылки (например, не участвует в создании контента), соединение защищено, а сервер-источник не является доверенным центром для редактируемого контента.

1. Основные методы HTTP-запроса.

Маркер метода запроса является основным источником семантики запроса; В нем **указывается цель, с которой клиент сделал этот запрос**, и то, что клиент ожидает в качестве успешного результата.

Все серверы общего назначения ДОЛЖНЫ поддерживать методы GET и HEAD. Все остальные методы являются НЕОБЯЗАТЕЛЬНЫМИ.

Набор методов, разрешенных целевым ресурсом, может быть перечислен в поле заголовка [Allow](https://httpwg.org/specs/rfc9110.html" \l "field.allow)

Сервер-источник, получающий метод запроса, который распознается и реализован, но не разрешен для целевого ресурса, должен ответить кодом состояния [405 (метод не разрешен).](https://httpwg.org/specs/rfc9110.html#status.405)

**╔═════════╦═════════════════════════════════════════════╗**

**║ Method ║ Description ║**

**╠═════════╬═════════════════════════════════════════════╣**

**║ GET ║ Передача текущего представления целевого ║**

**║ ║ ресурса ║**

**║ HEAD ║ То же, что и GET, но не передавайте ║**

**║ ║ содержимое ответа. ║**

**║ POST ║ Выполните обработку содержимого запроса ║**

**║ ║ для конкретного ресурса. ║**

**║ PUT ║ Замените все текущие представления целевого ║**

**║ ║ ресурса содержимым запроса ║**

**║ DELETE ║ Удалите все текущие представления ║**

**║ ║ целевого ресурса ║**

**║ CONNECT ║ Установите туннель к серверу, определенному ║**

**║ ║ целевым ресурсом. ║**

**║ OPTIONS ║ Опишите варианты коммуникации для ║**

**║ ║ целевого ресурса. ║**

**║ TRACE ║ Выполните проверку зацикливания сообщений ║**

**║ ║ на пути к целевому ресурсу. ║**

**╚═════════╩═════════════════════════════════════════════╝**

Методы запросов **считаются безопасными**, если их определенная семантика по существу доступна только для чтения; Т. е. клиент не запрашивает и не ожидает изменения состояния на сервере-источнике в результате применения безопасного метода к целевому ресурсу.

Из методов запроса, определенных в этой спецификации, методы [GET,](https://httpwg.org/specs/rfc9110.html#GET) [HEAD,](https://httpwg.org/specs/rfc9110.html#HEAD) [OPTIONS](https://httpwg.org/specs/rfc9110.html#OPTIONS) и [TRACE](https://httpwg.org/specs/rfc9110.html#TRACE) определены как безопасные.

**Метод GET** запрашивает передачу текущего [выбранного представления](https://httpwg.org/specs/rfc9110.html#selected.representation) для [целевого ресурса](https://httpwg.org/specs/rfc9110.html#target.resource).

**Метод HEAD** идентичен методу GET, за исключением того, что сервер НЕ ДОЛЖЕН отправлять содержимое в ответе. HEAD используется для получения метаданных о [выбранном представлении](https://httpwg.org/specs/rfc9110.html#selected.representation) без передачи его данных представления, часто ради тестирования гипертекстовых ссылок или поиска последних модификаций.

**Метод POST** требует, чтобы [целевой ресурс](https://httpwg.org/specs/rfc9110.html#target.resource) обработал представление, заключенное в запросе, в соответствии с собственной семантикой ресурса. Ответы на запросы POST можно кэшировать. Например, POST используется для следующих функций (среди прочих):

* Предоставление блока данных, таких как поля, введенные в HTML-форму, процессу обработки данных;
* Публикация сообщения на доске объявлений, в группе новостей, списке рассылки, блоге или аналогичной группе статей;
* Создание нового ресурса, который еще не идентифицирован сервером-источником; и
* Добавление данных к существующим представлениям ресурса.

**Метод PUT** запрашивает создание или замену состояния [целевого ресурса](https://httpwg.org/specs/rfc9110.html#target.resource) состоянием, определенным представлением, заключенным в содержимое сообщения запроса. Однако нет никакой гарантии, что такое изменение состояния будет наблюдаемым, так как целевой ресурс может подвергаться параллельным действиям со стороны других агентов пользователей или может подвергаться динамической обработке сервером-источником до получения любого последующего запроса GET. Успешный ответ означает только то, что намерение агента пользователя было достигнуто в момент его обработки сервером-источником.

Если целевой ресурс не имеет текущего представления и PUT успешно его создает, то сервер-источник *ДОЛЖЕН* сообщить об этом агенту пользователя, отправив ответ [201 (Created).](https://httpwg.org/specs/rfc9110.html#status.201) Если целевой ресурс имеет текущее представление и это представление успешно изменено в соответствии с состоянием вложенного представления, то сервер-источник *ДОЛЖЕН* отправить ответ [200 (OK)](https://httpwg.org/specs/rfc9110.html#status.200) или [204 (No Content),](https://httpwg.org/specs/rfc9110.html#status.204) чтобы сообщить об успешном завершении запроса. Ответы на метод PUT не кэшируются.

**Метод DELETE** требует, чтобы сервер-источник удалил связь между [целевым ресурсом](https://httpwg.org/specs/rfc9110.html#target.resource) и его текущей функциональностью. Ответы на метод DELETE не кэшируются.

**Метод CONNECT** требует, чтобы получатель установил туннель к целевому серверу-источнику, определенному целевым объектом запроса, и в случае успеха затем ограничил его поведение слепой пересылкой данных в обоих направлениях до тех пор, пока туннель не будет закрыт. Туннели обычно используются для создания сквозного виртуального соединения через один или несколько прокси-серверов, которые затем могут быть защищены с помощью TLS. CONNECT использует специальную форму цели запроса, уникальную для этого метода, состоящую только из номера хоста и порта назначения туннеля, разделенных двоеточием. Порт по умолчанию отсутствует; клиент ДОЛЖЕН отправить номер порта, даже если запрос CONNECT основан на ссылке URI, содержащей компонент полномочий с опущенным портом.

**Метод OPTIONS** запрашивает сведения о параметрах связи, доступных для целевого ресурса, либо на сервере-источнике, либо на промежуточном сервере. Этот метод позволяет клиенту определять параметры и/или требования, связанные с ресурсом или возможностями сервера, не подразумевая действия ресурса. Он не делает ничего, кроме того, что позволяет клиенту протестировать возможности сервера. Например, это может быть использовано для проверки прокси на соответствие (или его отсутствие) HTTP/1.1.

**Метод TRACE** запрашивает удаленный цикл на уровне приложения для сообщения запроса. Конечный получатель запроса ДОЛЖЕН отразить полученное сообщение, за исключением некоторых полей, описанных ниже, обратно клиенту в виде содержимого ответа [200 (OK](https://httpwg.org/specs/rfc9110.html#status.200))

Зав. каф. ИСиТ В.В. Смелов