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

**HTTP** (англ. HyperText Transfer Protocol — «протокол передачи гипертекста») протокол прикладного уровня передачи данных, изначально — в виде гипертекстовых документов в формате HTML, в настоящее время используется для передачи произвольных данных.

**Основные свойства**

- версии HTTP/1.1 – действующий (текстовый), HTTP/2 – черновой (не распрост, 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 (HTTPS);

- для адресации используется URI или URN;

- поддерживается W3C, описан в нескольких RFC.

**Консо́рциум Всеми́рной паути́ны (англ. World Wide Web Consortium, W3C)** — организация, разрабатывающая и внедряющая технологические стандарты для Всемирной паутины.

**RFC** - Рабочее предложение (англ. Request for Comments, RFC) — документ из серии пронумерованных информационных документов Интернета, содержащих технические спецификации и стандарты, широко применяемые во всемирной сети.

**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).

**Типы сообщений:** request и response;

**Структура HTTP-запроса**

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

**Структура ответа**

- версия протокола (HTTP/1.1);

- код состояния (1xx, 2xx, 3xx, 4xx, 5xx);

- пояснение к коду состояния;

- заголовки (пары: имя/заголовок);

- расширение.

**Response: Код состояния:**

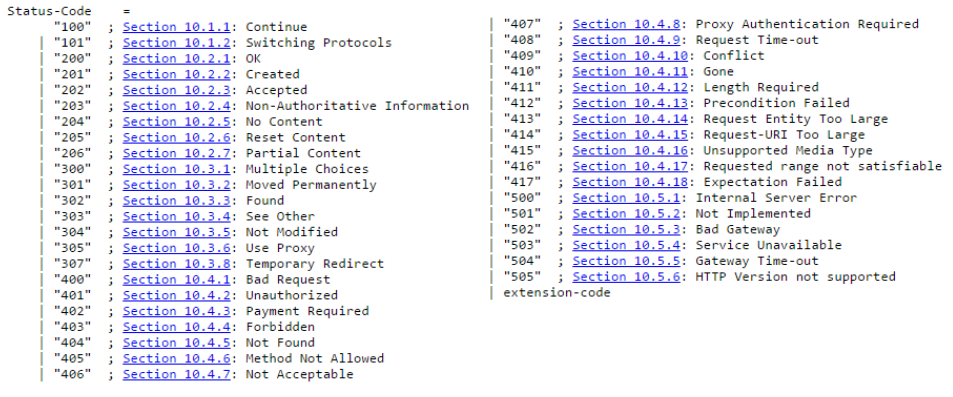
**1xx**: информационные сообщения;

**2xx**: успешный ответ;

**3xx**: переадресация;

**4xx**: ошибка клиента;

**5xx**: ошибка сервера.



**Методы**:

**get** – запросы могут только извлекать данные;

**post** – исп для отправки сущностей к ресурсу;

**patch(или put)** – исп для частичного изменения ресурса;

**head** – запрашивает, как и get, но без тела ответа;

**delete** – удаляет указанный ресурс;

**connect** – устанавливает соединение к серверу, определенному ресурсу;

**options** – исп для описания параметров соединения с ресурсом;

**trace** – выполняет вызов возвращаемого тестового сообщения с ресурса.

**Заголовки**:

Заголовок – пара ключ значение, которые содержат служебную инфу о запросе или ответе.

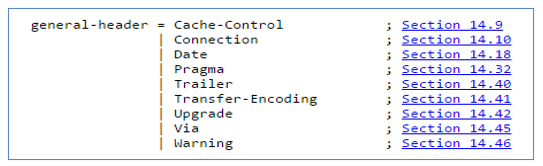
Изначально есть 2 типа заголовков:

1) Стандартные (Определены в HTTP)

2)Кастомные

Заголовки делятся на 4 типа:

- General: общие заголовки, используются в запросах и ответах;



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

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

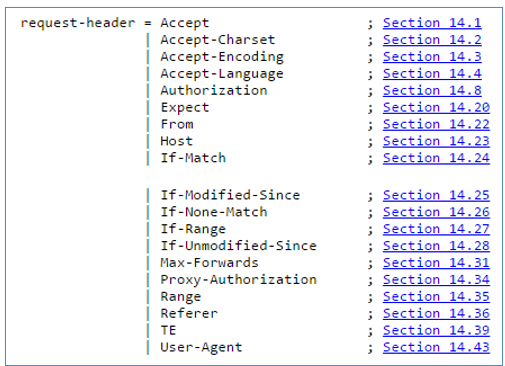
**Transfer-Encoding** указывает форму кодирования, используемую для безопасной передачи тела полезных данных пользователю.

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 подключения!!!!, 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.

**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))

- Request: используются только в запросах;



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.

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).

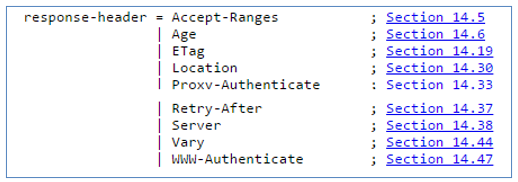
Заголовок запроса **Host** указывает хост и номер порта сервера, на который отправляется запрос. Если порт не указан, подразумевается порт по умолчанию для запрошенной службы (например, 443 для URL-адреса HTTPS и 80 для URL-адреса HTTP).

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** – 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.

- Response: используются только в ответах;

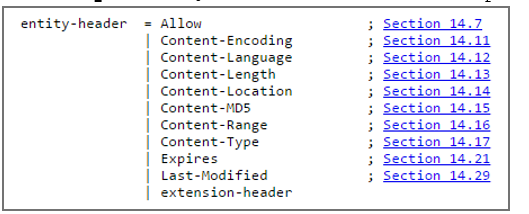


Заголовок **Server** описывает программное обеспечение, используемое исходным сервером, который обработал запрос, то есть сервером, сгенерировавшим ответ.

**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)

**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)

- Entity: для сущности в ответах и запросах.



**Content-Type** используется для указания исходного типа мультимедиа ресурса (до любого кодирования контента, примененного для отправки). описывает MIME-тип содержимого, передаваемого в теле сообщения.

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

**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

-Пользовательские: для управления каналом связи между клиентом и сервером. Начинаются с символа Х.

**Протокол без сохранения состояния** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Stateless protocol*) — это [протокол передачи данных](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB_%D0%BF%D0%B5%D1%80%D0%B5%D0%B4%D0%B0%D1%87%D0%B8_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), который относит каждый запрос к независимой транзакции, которая не связана с предыдущим запросом, то есть общение с сервером состоит из независимых пар [запрос-ответ](https://ru.wikipedia.org/w/index.php?title=%D0%97%D0%B0%D0%BF%D1%80%D0%BE%D1%81-%D0%BE%D1%82%D0%B2%D0%B5%D1%82&action=edit&redlink=1). Протокол не помнит своё состояние. Каждый запрос открывает новый цикл и отрабатывает независимо от остальных.

Протокол без сохранения состояния не нуждается в сохранении информации о сессии на сервере или статусе о каждом клиенте во время множественных запросов. В противовес этому, протокол, которому необходим учёт о внутреннем состоянии сервера, называется протоколом с сохранением [состояния](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D0%B5_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *stateful*).

Примерами протоколов без сохранения состояния являются [Internet Protocol](https://ru.wikipedia.org/wiki/Internet_Protocol) (IP), фундаментальный протокол сети [Интернет](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82), и [Hypertext Transfer Protocol](https://ru.wikipedia.org/wiki/Hypertext_Transfer_Protocol) (HTTP), это означает, что каждое сообщение запроса может быть понято в изоляции от других запросов.

[**HTTP не имеет состояния, но имеет сессию**](https://developer.mozilla.org/ru/docs/Web/HTTP/Overview#http_не_имеет_состояния_но_имеет_сессию)

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

<https://developer.mozilla.org/ru/docs/Web/HTTP/Overview>

Версии HTTP:

* 1. 0.9 (methods: GET only, hypertext only, no HTTP headers (cannot transfer other content type files), no status/error codes, no URLs, no versioning)
  2. 1.0 (недействующий, нет постоянного соединения, methods: GET , HEAD , POST , имеет `Content-Type`)
  3. 1.1 (действующий, имеет постоянное соединение, has `Keep-alive`)
  4. 2.0 (бинарный, не распространен)
  5. 3.0 (не распространен)

*2. Протокол HTTPS: TLS, шифронаборы, сертификаты, процедура рукопожатия.*

**HTTPS** (Hypertext Transport Protocol Secure) – расширение протокола HTTP, который обеспечивает конфиденциальность обмена данными между сайтом и пользовательским устройством.

Главные возможности HTTPS (со слов смелова)

1. Аутентификация (обеспечивается через TLS)

2. Шифрование канала

TLS – надстройка над TCP, имеет свой API

HTTPS не является отдельным протоколом. Это обычный HTTP, работающий через шифрованные транспортные механизмы SSL и TLS. Он обеспечивает защиту от атак, основанных на прослушивании сетевого соединения — от снифферских атак и атак типа man-in-the-middle, при условии, что будут использоваться шифрующие средства и сертификат сервера проверен и ему доверяют.

**Transport Layer Security**

**1) обеспечивает конфиденциальность;**

**2) обеспечивает целостность (обнаружение подмены);**

**3) аутентификация узлов (проверка подлинности источника сообщений);**

**SSL** (Secure Sockets Layer) и TLS (Transport Level Security) — криптографические протоколы, обеспечивающие защищенную передачу данных в компьютерной сети.

Соединение, защищенное протоколом TLS, обладает одним или несколькими следующими свойствами:

* Безопасность: симметричное шифрование защищает передаваемую информацию от прочтения посторонними лицами.
* Аутентификация: "личность" участника соединения можно проверить с помощью асимметричного шифрования.
* Целостность: каждое сообщение содержит код (Message Authentication Code, MAC), с помощью которого можно проверить, что данные не были изменены или потеряны в процессе передачи.

«Рукопожатие SSL/TLS» — это название этапа установки HTTPS-соединения. Большая часть работы, связанной с протоколом SSL/TLS, выполняется именно на этом этапе.

**Цель рукопожатия: (почти со слов смелова)**

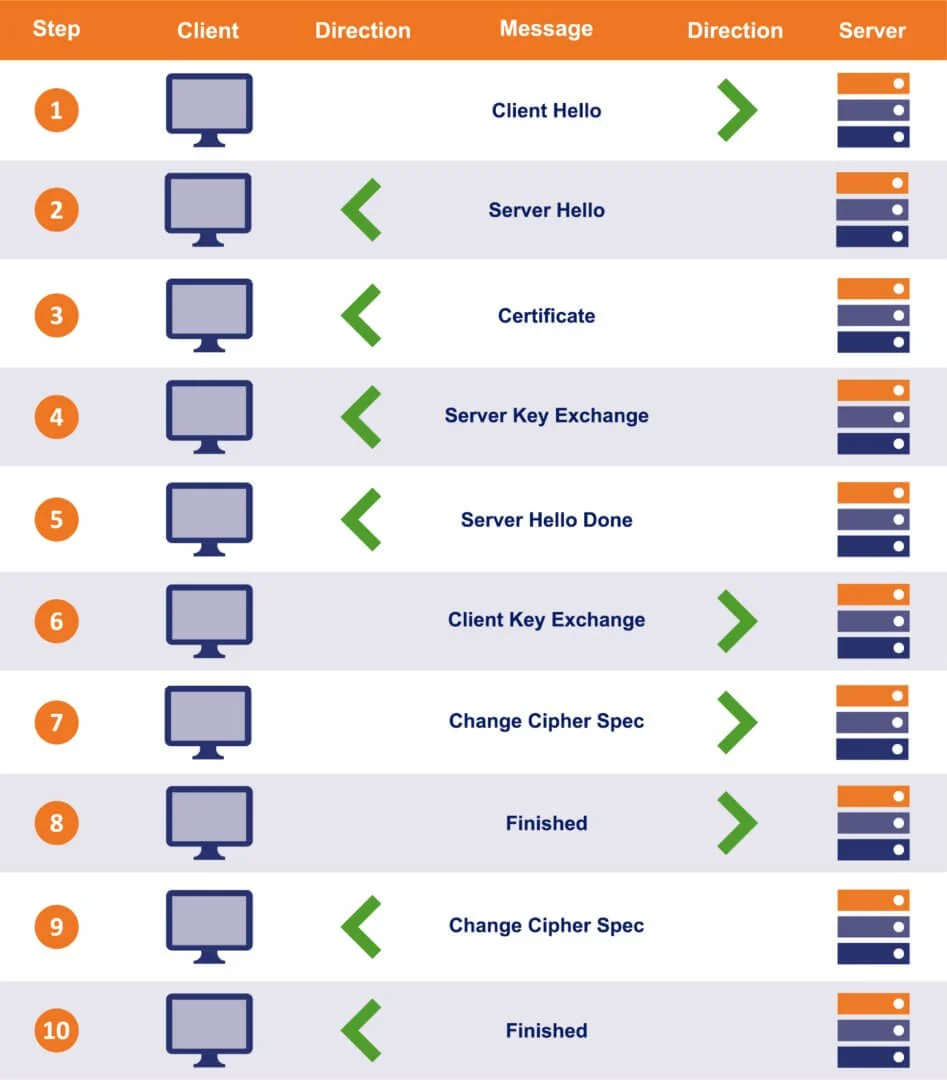
1. что-то про соединение между клиентом и сервером

2. договорится о ключе шифрования

**Процедура рукопожатия:**

Перед тем, как начать обмен данными через TLS, клиент и сервер должны согласовать параметры соединения, а именно: версия используемого протокола, способ шифрования данных, а также проверить сертификаты, если это необходимо.

Ниже описан детальный процесс рукопожатия для **TLS версии 1.2**



1. Первое сообщение называется «Client Hello». В этом сообщении перечислены возможности клиента, чтобы сервер мог выбрать **шифронабор**, который будет использовать для связи. Также сообщение включает в себя большое случайно выбранное простое число, называемое «случайным числом клиента».
2. Сервер вежливо отвечает сообщением «Server Hello». Там он сообщает клиенту, какие параметры соединения были выбраны, и возвращает своё случайно выбранное простое число, называемое «случайным числом сервера». Если клиент и сервер не имеют общих шифронаборов, то соединение завершается неудачно.
3. В сообщении «Certificate» сервер отправляет клиенту свою цепочку SSL-сертификатов, включающую в себя листовой и промежуточные сертификаты. Получив их, клиент выполняет несколько проверок для верификации сертификата. Клиент также должен убедиться, что сервер обладает закрытым ключом сертификата, что происходит в процессе обмена/генерации ключей.

Все доверенные SSL-сертификаты выпускаются центром сертификации (ЦС). ЦС должен следовать строгим правилам выдачи и проверки сертификатов, чтобы ему доверяли. Вы можете считать ЦС кем-то вроде нотариуса — его подпись значит, что данные в сертификате реальны.

1. Это необязательное сообщение, необходимое только для определённых методов обмена ключами (например для алгоритма Диффи-Хеллмана), которые требуют от сервера дополнительные данные.
2. Сообщение «Server Hello Done» уведомляет клиента, что сервер закончил передачу данных.
3. Затем клиент участвует в создании сеансового ключа. Особенности этого шага зависят от метода обмена ключами, который был выбран в исходных сообщениях «Hello». Так как мы рассматриваем RSA, клиент сгенерирует случайную строку байтов, называемую секретом (pre-master secret), зашифрует её с помощью открытого ключа сервера и передаст обратно.
4. Сообщение «Change Cipher Spec» позволяет другой стороне узнать, что сеансовый ключ сгенерирован и можно переключиться на зашифрованное соединение.
5. Затем отправляется сообщение «Finished», означающее, что на стороне клиента рукопожатие завершено. С этого момента соединение защищено сессионным ключом. Сообщение содержит данные (MAC), с помощью которых можно убедиться, что рукопожатие не было подделано.
6. Теперь сервер расшифровывает pre-master secret и вычисляет сеансовый ключ. Затем отправляет сообщение «Change Cipher Spec», чтобы уведомить, что он переключается на зашифрованное соединение.
7. Сервер также отправляет сообщение «Finished», используя только что сгенерированный симметричный сеансовый ключ, и проверяет контрольную сумму для проверки целостности всего рукопожатия.

После этих шагов SSL-рукопожатие завершено. У обеих сторон теперь есть сеансовый ключ, и они могут взаимодействовать через зашифрованное и аутентифицированное соединение.

На этом этапе могут быть отправлены первые байты веб-приложения (данные, относящиеся к фактическому сервису, — HTML, Javascript и т. д.).

Рукопожатие **TLS 1.3** значительно короче, чем его предшественник.



1. Как и в случае TLS 1.2, сообщение «Client Hello» запускает рукопожатие, но на этот раз оно содержит гораздо больше информации. TLS 1.3 сократил число поддерживаемых шифров с 37 до 5. Это значит, что клиент может угадать, какое соглашение о ключах или протокол обмена будет использоваться, поэтому в дополнение к сообщению отправляет свою часть общего ключа из предполагаемого протокола.
2. Сервер ответит сообщением «Server Hello». Как и в рукопожатии 1.2, на это м этапе отправляется сертификат. Если клиент правильно угадал протокол шифрования с присоединёнными данными и сервер на него согласился, последний отправляет свою часть общего ключа, вычисляет сеансовый ключ и завершает передачу сообщением «Server Finished».
3. Теперь, когда у клиента есть вся необходимая информация, он верифицирует SSL-сертификат и использует два общих ключа для вычисления своей копии сеансового ключа. Когда это сделано, он отправляет сообщение «Client Finished».

Смелов + лк

**Клиент** выдает запрос серверу в котором определен его список шифрнабора (Client Hello).

**Сервер** подписывает свой сертификат и высылает клиенту поддерживаемый шифрнабор + свой сертификат (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.

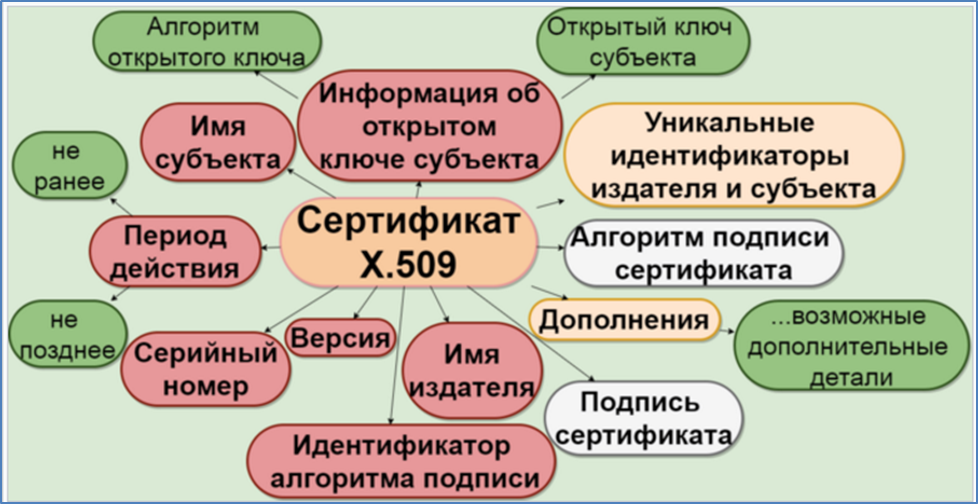
**Клиент** генерирует симметричный ключ, шифрует его открытым ключом.

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

**SSL-сертификат**

SSL-сертификат – это своего рода уникальная цифровая подпись вашего сайта. Такой сертификат нужен, в первую очередь, банкам, платежным системам и другим организациям, работающим с персональными данными, – для защиты транзакций и предотвращения несанкционированного доступа к информации.

**SSL-сертификат содержит следующую информацию:**

****

* доменное имя, на которое оформлен SSL-сертификат;
* юридическое лицо, которое владеет сертификатом;
* физическое местонахождение владельца сертификата (город, страна);
* срок действия сертификата;
* реквизиты компании-поставщика SSL-сертификатa

**Существует несколько типов SSL-сертификатов:**

**SSL-сертификаты с проверкой домена (Domain Validation**) – это сертификаты начального уровня, являются самыми распространенными в мире, а скорость выдачи таких сертификатов варьируется от 2 до 10 минут, зависит от бренда. Чтобы получить такой сертификат, не требуется специальных документов. Такие сертификаты подойдут для небольших сайтов и маленьких проектов, когда нет необходимости в большом доверии со стороны клиентов и посетителей сайта.

**SSL-сертификаты с проверкой компании (Business Validation)** – такие сертификаты актуальны для тех, кто думает о доверии к своим продуктам, компании и сервисам, так как центр сертификации проводит более тщательную проверку.

**SSL-сертификаты с расширенной проверкой (Extended validation)** – только EV сертификаты обеспечат сайт зеленой адресной строкой в браузере. Чаще всего такие сертификаты можно встретить у банков, онлайн-систем с большим количеством посетителей.

**SSL-сертификаты c поддержкой субдоменов (Wildcard)** – это очень удобный сертификат, когда речь идет о защите большого количества субдоменов в рамках одного домена.

**SAN SSL-сертификаты** – единые сертификаты связи, которые способны защищать множество доменов, субдоменов, локальных доменов и серверов.

**SGC SSL-сертификаты** – устаревающие сертификаты, которые принудительно увеличивают уровень шифрования для старых браузеров с 40 бит до полноценных 256 бит.

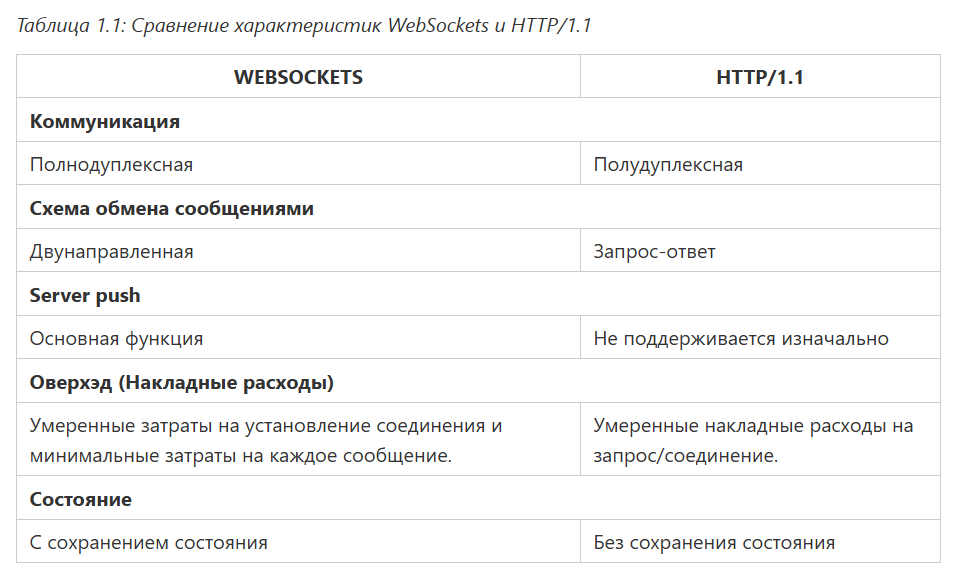
**SSL-cертификаты для ПО (CodeSigning SSL)** – такие сертификаты могут помочь, когда ваши пользователи получают предупреждения и ошибки при скачивании программного кода с ваших ресурсов. Это идеальный продукт для разработчиков программного обеспечения, он используется для защиты программных продуктов распространяемых в сети.

*4. Протокол WebSockets: принципы работы и применения. Пример.*

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

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

**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.



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

Из другого файла

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).

Из книжки смелова

Web Sockets API – новый программный интерфейс, основанный на протоколе полнодуплексной связи WebSocket. Протокол работает над TCP-соединением и предназначен для обмена сообщениями между браузером и web-сервером. Протокол WebSocket описан в RFC 6465 [19].

Для установки соединения (процедура websocket handshake) между клиентом и сервером WebSocket использует запрос и ответ в формате протокола HTTP. При этом протокол определяет две URL-схемы: ws – для незашифрованного соединения и wss – для зашифрованного.

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

Для разработки серверной компоненты web-приложения, работающей по протоколу WebSocket, программисты могут применять программные интерфейсы, входящие в состав различных технологий разработки web-приложений. Поддержка WebSocket осуществляется технологиями корпорации Microsoft: IIS, ASP.NET и WCF. В состав технологий Java EE тоже входит пакет (javax.websocket), обеспечивающий взаимодействие клиента и Application Server по протоколу WebSocket.

три обработчика событий соединения: onopen (соединение открыто), onclose (соединение закрыто), onmessage (от сервера поступило сообщение)

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

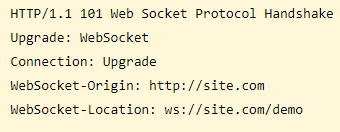
Изначально синхронный HTTP протокол, построенный по модели «запрос — ответ», становится **полностью асинхронным и симметричным**. Теперь уже нет клиента и сервера с фиксированными ролями, а есть два равноправных участника обмена данными. Каждый работает сам по себе, и когда надо отправляет данные другому. 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.

**Скорость и эффективность:**

Высокую скорость и эффективность передачи обеспечивает малый размер передаваемых данных. Так же соединение уже готово — не надо тратить время и трафик на его установление.

**Время жизни канала:**

Веб-сокеты не имеют ограничений на время жизни в неактивном состоянии. Соединение может висеть в неактивном виде и не требовать ресурсов.

**Количество:**

Открываете столько, сколько вам нужно. А сколько использовать — одно (и через него все мультиплексировать) или же наоборот — на каждый блок свое соединение — решать вам. Исходите из удобства разработки, нагрузки на сервер и клиент.

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

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

Из книжки смелова

Для построения эффективных web-приложений до появления HTML5 применялась технология AJAX (Asynchronous JavaScript and XML), которая основывалась на возможности доступного из JavaScript объекта XMLHTTPRequest асинхронно отправлять http-запросы серверу и обрабатывать его ответы. В качестве формата передачи данных в AJAX, как правило, используется XML и JSON. Основное достоинство AJAX-приложения заключается в его динамичности: применение AJAX позволяет сократить сетевой трафик и избежать лишних перезагрузок html-страниц.

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:

**HTML–** гипертекстовый язык разметки. Интерпретируется браузером. В Ajax динамически изменяется содержимое html-документа.

**JavaScript**– скриптовый язык, предназначенный для создания сценариев поведения браузера. Интерпретируется браузером. В Ajax html-документ динамически изменяется на стороне клиента с помощью сценариев написанных на языке JavaScript.

**DOM –** объектная модель, позволяющая сценариям JavaScript получить доступ (читать и изменять содержимое) к элементам html-документа (к атрибутам и содержимому тегов). В Ajax ответ сервера ―встраивается‖ с помощью JavaScript-сценария в загруженную ранее браузером страницу. При этом доступ к элементам html-документа осуществляется в соответствии с моделью DOM.

**HTTP –**сетевой протокол передачи гипертекста. Используется для обмена данными между двумя приложениями (клиентом и сервером). В Ajax обмен данными между JavaScript-сценарием на клиенте и серверным приложением (например, сервлетом) осуществляется по правилам HTTP.

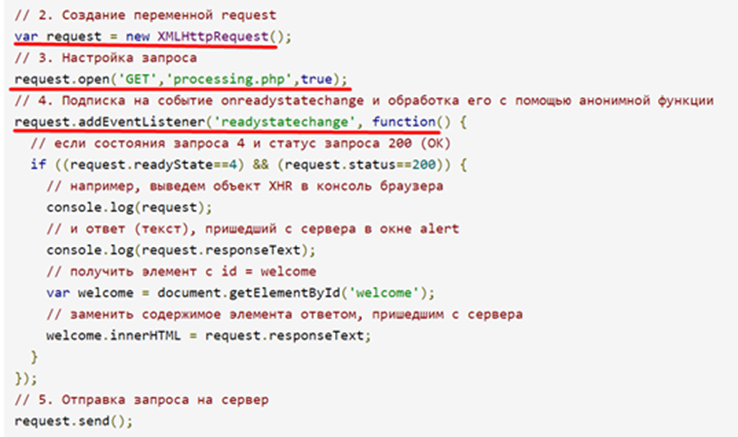
**XML –** расширяемый язык разметки данных. Предназначен для структуризации данных с целью хранения или/и передачи. В Ajax язык XML является одним из форматов, который используется для структуризации данных пересылаемых между JavaScript-сценарием и серверным приложением.

**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).



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).

Прежде чем, мы с помощью объекта XMLHttpRequest выполним метод Send(), который отправляет асинхронный запрос к серверу, мы этому объекту скажем: а вот когда придет ответ, ты вызови эту функцию, которую мы укажем Т.е. мы ему заранее подсказываем: вызови такую функцию, когда придет ответ и мы с вами продолжаем работать: отправили запрос, продолжаем дальше работать и когда приходит ответ, он вызывает эту функцию и она выполняется, в ней обрабатывается ответ. Причем, эта функция вызывается 5(хммм смелов говорил 4 раза) раз в разные этапы выполнения запроса и только на 4 этапе (нумерация с 0, по счету 5ый раз получается) мы получаем статус ответа, где мы можем выяснить, нормально или ненормально закончился ответ. Это то, что лежит в глубине ajax.

**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** или **onchange** — 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:

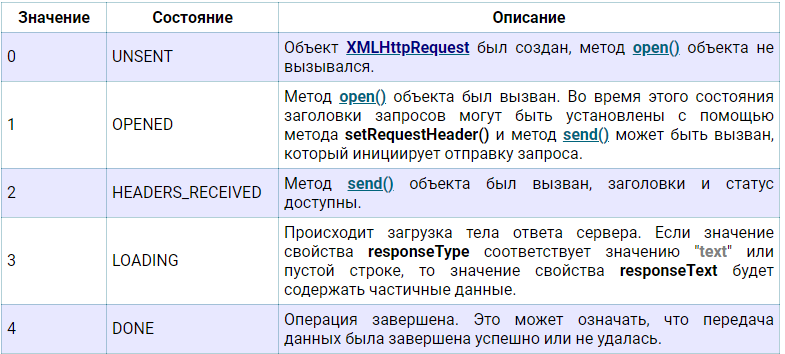
* 0 – начальное состояние
* 1 – вызван open
* 2 – получены заголовки
* 3 – загружается тело
* 4 – запрос завершён

Сейчас существует множество js-библиотек: JQuery, Angular, React, которые скрывают от нас этот механизм, и мы с ним не работаем.

**AJAX в ASP.net представлен в виде:**

* UpdatePanel
* Trigger
* Timer
* UpdateProgress

Когда мы пишем cкрипт, браузер сам подставляет объявление всех его объектов, например: window, [XMLHttpRequest](https://ru.wikipedia.org/wiki/XMLHttpRequest), document.



**AJAX** — не самостоятельная технология, а концепция использования нескольких смежных технологий. AJAX базируется на двух основных принципах:

1. использование технологии динамического обращения к [серверу](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5)) «на лету», без перезагрузки всей страницы полностью, например с использованием [XMLHttpRequest](https://ru.wikipedia.org/wiki/XMLHttpRequest) (основной объект);

* через динамическое создание дочерних фреймов[[1]](https://ru.wikipedia.org/wiki/AJAX#cite_note-1);
* через динамическое создание тега <script>[[2]](https://ru.wikipedia.org/wiki/AJAX#cite_note-JsHttpRequest-2).
* через динамическое создание тега <img>, как это реализовано в Google Analytics.

1. использование [DHTML](https://ru.wikipedia.org/wiki/DHTML) для динамического изменения содержания страницы;

ПРИМЕНЕНИЕ:

**Экономия трафика:** загружает определенную часть страницы или просто получает/передает данные в формате JSON или XML, и затем изменяет содержимое страницы с помощью JS.

**Уменьшение нагрузки на сервер:** говорится, что для загрузки страницы требуется обращение к разным файлам, время на обработку скриптов (иногда запросы к БД) и все это можно заменить загрузкой и генерацией лишь части страницы.

**Ускорение реакции интерфейса:** Поскольку загрузка изменившейся части значительно быстрее, то пользователь видит результат своих действий быстрее и без мерцания страницы (возникающего при полной перезагрузке).

**Возможности для интерактивной обработки:**

Например, при вводе поискового запроса в [Google](https://ru.wikipedia.org/wiki/Google_(%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%BE%D0%B2%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0)) выводится подсказка с возможными вариантами запроса. На многих сайтах при регистрации пользователь вводит имя, и сразу же видит, доступно это имя или нет. AJAX удобен для программирования [чатов](https://ru.wikipedia.org/wiki/%D0%A7%D0%B0%D1%82), [административных панелей](https://ru.wikipedia.org/w/index.php?title=%D0%90%D0%B4%D0%BC%D0%B8%D0%BD%D0%B8%D1%81%D1%82%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BD%D0%B5%D0%BB%D1%8C&action=edit&redlink=1) и других инструментов, которые выводят меняющиеся со временем данные.

[**Мультимедиа**](https://ru.wikipedia.org/wiki/%D0%9C%D1%83%D0%BB%D1%8C%D1%82%D0%B8%D0%BC%D0%B5%D0%B4%D0%B8%D0%B0) **не останавливается :**

Страница не перезагружается, плеер продолжает работать. Потому AJAX ценен на аудио- и видеохос

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.

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

**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.

Веб-сервер представляет собой: HTTP-сервер + файлы. Именно HTTP-сервер взаимодействует с клиентом. Asp.net framework должен быть установлен только на сервере, на клиенте - его нет. Веб-приложение включает все: клиент + сервер.

**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. Одним из таких Web-серверов является Internet Information Services (IIS) компании Microsoft. Это единственный Web-сервер, который способен выполнять Web-приложения, созданные с использованием технологии ASP.NET.

**Клиент-серверная архитектура** – это архитектура, которая подразумевает две компоненты: клиент и сервер. Клиент является инициатором соединения.

Одним из преимуществ такого подхода является тот факт, что клиенты не зависят от конкретной операционной системы пользователя, поэтому веб-приложения являются межплатформенными службами.

**Архитектура web-приложения(везде протокол http):**

Есть клиент и есть сервер – две части одной и той же программы, которые взаимодействуют между собой по протоколу HTTP.

Бывают случаи, когда один клиент работает с несколькими серверами. Может быть, когда несколько клиентов работают с одним сервером. Может быть, когда сервер обращается к другому серверу и первый сервер выступает в качестве клиента по отношению к другому серверу.

Между клиентом и сервером ходит 2 типа сообщений:

* от клиента к серверу – **request**
* от сервера к клиенту – **response**

**Особенности реализации web-приложения:**

1. При проведении работ, которые требуют остановку веб сервера, необходимо иметь запасной, на который можно перенаправить трафик (**отказоустойчивость**)
2. Т.к. к веб-серверу происходит множество запросов от различных пользователей, то возможности ответа веб-сервера упираются в пропускную способность канала и, чтобы не загружать этот канал слишком сильно, используются максимально короткие сеансы подключения и кэширование ответов на стороне клиента, а кэширование на стороне сервера позволяет быстрее сформировать ответ. (**Быстродействие**)
3. Использование https для устранения возможности злоумышленникам подделывать запросы/ответы и просматривать их. (**Шифрование**)

**Web-сервер:**

1. Программа, которая обрабатывает Http запросы от веб-клиента и отправляет ответ
2. Примером веб-сервера может служить IIS, Apache Tomcat

**Web-клиент:** можно рассматривать веб-браузер. Это программа, которая переносит большую часть функционала по обработке информации на сервер.

1. Для отображения контента используется HTML5.
2. Веб-клиент инициирует соединение(отправляет запрос) с помощью протокола http
3. Использует DOM модель для представления html документа, а javascript для доступа к содержимому HTML документа
4. Использует CSS (Cascading Style Sheets) для добавления стилей к html документу
5. Web-engine - веб движок, который преобразует html страницу во внутреннее представление веб-браузера в соответствии с моделью DOM. В хроме это Blink, написан на C++

**Клиент: веб-браузер**

Умеет интерпретировать программу, которую ему присылает сервер.

Веб-браузер умеет генерировать HTTP-запросы:

– если есть адресная строка, и он делает запрос

– есть ряд HTML-тегов, при интерпретации которых браузер делает запрос (form, a, img, script, link, audio, video)

– объект веб-браузера: XMLHTTPRequest

– JS API

Любой веб-браузер характеризуется моделью DOM. Браузер в себе содержит web engine (ядро браузера). На сегодняшний момент существует набор веб-движков, на базе которых работают браузеры. Стандарты веб-браузеров: HTML5, CSS3, SVG, JavaScript (ECMAScript5, ECMA-262-6, ECMA-262-7).

**Общие принципы построения веб-приложений**

– веб-ресурсы приложения

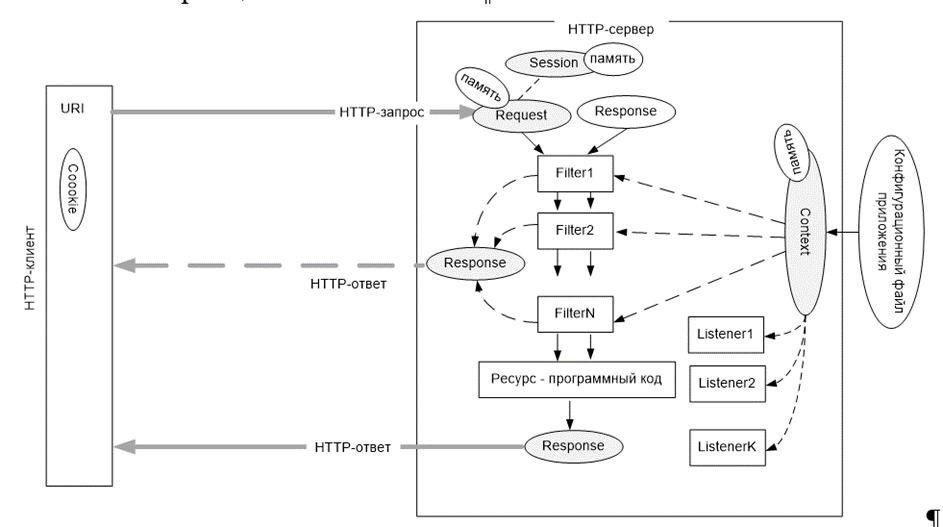
– запросы и ответы

– фильтры

– кэш (данных и вывода)

– слушатели событий

– принципы безопасности



**HTTP–сервер** – программа, которая слушает некоторый порт на своем компьютере.

**Порт** – номер приложения, которому адресован этот запрос. Понятие порт существует на уровне tsp-протокола. Порт слушает входящие сообщения.

**Веб-приложение** принимает заполненный request ответ, пустой ответ response, заполняет response и отдает его серверу.

**Сессия** – серверный объект, хранящий информацию о соединении с клиентом, создается при первом обращении. Время жизни: **timeout** (системный параметр, обычно равен 10-30 минут) – максимальное время между запросами клиента.

Если timeout превышен, то session разрушается и при последующем запросе создается новый экземпляр. Каждая сессия имеет собственный идентификатор (Session ID, 128 или больше бит (16 байт)). Каждый request принадлежит какой-то сессии. Сессия характеризуется двумя параметрами: timeout и session ID.

Обычно в request сервер записывает либо идентификатор этой сессии, либо просто программную ссылку на этот объект сессии.

*Если с одним и тем же сервером работает много клиентов, у каждого из них своя сессия.*

**Куки** – это порция информации, которая может быть сохранена на стороне клиента по инициативе сервера. Когда клиент делает первый запрос, сервер проверяет, есть ли у него заголовок с именем куки. Если этого заголовка нет, то он считает, что это первый запрос и для него создается сессия.

Клиент должен хранить информацию о том, в рамках какой сессии мы отправляем эти запросы. Браузер в себе сохраняет либо файл куки, но чаще сохраняет **localstorage** (поддерживается на стороне браузера).

**LocalStorage** **это** свойство объекта window, предназначенное для хранения пар ключ/значение в браузере. В зависимости от браузера, мы можем сохранять до 5 мб данных.

**IndexedDB** – **это** встроенная база данных, более мощная, чем localStorage . Хранилище ключей/значений: доступны несколько типов ключей, а значения могут быть (почти) любыми. Поддерживает транзакции для надёжности.

**Протокол RFC 6265** – стандарт, описывающий механизм куки.

**Конфигурационный файл** (обычно xml) содержит в себе некоторые статические характеристики приложения. Он служит для создания контекста веб-приложения.

**Контекст веб-приложения** – системный объект общий для всех сессий. Предназначен для хранений информации об одном веб-приложений, общий для всех сессий. Как правило, формируется сразу при загрузке веб-сервера.

**Фильтр** – серверный объект, препроцессор запроса, предназначенный для предварительно обработки объекта request. К одному ресурсу может быть построена цепочка фильтров, последний в цепочке – ресурс. Фильтр может прервать цепочку и сам сформировать ответ клиенту.

Информация о цепочке фильтров и привязке этих фильтров к ресурсу хранится в контексте.

Фильтр может не пустить запрос к ресурсу. Он может сам ответить, не пуская его дальше: к следующему фильтру или ресурсу.

**Слушатели событий (Listener)** – серверные объекты для обработки событий жизненного цикла веб-приложения. С помощью Listener можно врезаться в ход выполнения приложения и что-то там изменит.

*Пример – событие создания контекста. Можно написать listener, который выполнится, когда создастся контекст. Это необходимо, например, когда надо записать динамические данные в контексте. Или может быть listener на создание сессии.*

**Кэш** – это системный объект, предназначенный для хранения данных в оперативной памяти с целью ускорения работы веб-приложения. Часто в кэше запоминаются response.

Для каждого запроса создается новый экземпляр приложения. Общим остается контекст. В рамках серии запросов остается сессия. На каждый новый запрос создается новый экземпляр приложения.

**Фильтры** загружены всегда, вместе с контекстом. Фильтры привязываются к URI.

**Постоянное соединение**. Использование одного TCP-соединения для многократных пар запрос-сервер вместо последовательного открытия новых соединений для каждой пары запрос-ответ. Клиент может запросить постоянное соединение с помощью заголовка *Connection: Keep-Alive*, сервер подтверждает заголовком Connection: Keep-Alive.

**Обычный запрос проходит несколько стадий**:

1. Открытие соединения.

2. Отправка запроса.

3. Получение ответа.

4. Закрытие соединения.

**Конвейерная обработка** (**HTTP pipelining**). Редко поддерживаются серверами.

**Пул соединение 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 ставиться в очередь.

**Пул соединение** – два постоянно открытых коннекта в базе данных. К этому пулу устанавливается очередь запросов. Можно регулировать количество этих соединений в зависимости от нагрузки.

Создается несколько соединений, к пулу отправляется sql-запрос, который пул перенаправляет к БД.

**Кэширование на стороне браузера**: управление заголовками. Кеширование на стороне браузера описано в протоколе 7234. Если запрос обыкновенный, без кэширования, то при каждом новом запросе эта картинка будет скачиваться. Если картинка кэшируется, то все сохраняется на стороне браузера и запрос получается легковесным.

**Краткий вариант объяснения**

У нас есть объекты **response**, **request**. Они всегда ходят парой. Время их жизни совпадает. У нас есть **ресурс** - это наш **программный код**, который генерирует нам ответ. В объекте request у нас есть **память**, которой программист может воспользоваться. Также у нас есть объект **session**. В нем тоже есть память. Есть **ID** сессии и **timeout** - время жизни. Сессия живет один сеанс. Очень важно, чтобы были **cookies**, иначе работать не будет. Сессия связана с нашим запросом.

У нас есть **конфигурационный файл веб-приложения**, который содержит все параметры приложения. На базе этого файла создается контекст приложения.

**Контекст веб-приложения** - это тоже системный объект (как сессия, запрос или ответ). Он общий для всех сессий! Он хранит информацию о веб-приложении. В нем соответственно своя память, которую программист может тоже использовать.

**Фильтр** - еще один системный объект, он нужен для предварительной обработки запросов и ответов. Он решает куда отдать твой запрос дальше. Может быть несколько фильтров и они будут друг другу их "перефутболивать", как говорил Смелов. Фильтр может вообще не отдать твой запрос на ресурс и, грубо говоря, сформировать ответ сам. Например, это используется для шифрования или расшифрования информации. Информация об этих всех фильтрах (**цепочке**) содержится в контексте.

**Слушатели событий** (листенер) - еще одни специальный объект на стороне сервера, используются для обработки событий жизненного цикла приложения. То есть, мы можем врезаться в ход выполнения нашего приложения. (например, мы это можем делать в файле global.asax). Жизненные циклы это: инит, лоад, прелоадер, анлоад, диспоуз.

**Кэш** - еще один объект. Есть специальная программа кэш-менеджер, которая управляет объектом кэш.

**Как это работает**

Мы отправляем http-запрос от клиента к серверу. Общий вид запроса - битовая последовательность. Http-сервер состоит из http-драйвера и ресурс-программный код. Драйвер первый принимает http-запрос. Он формирует 2 серверных объекта: реквест и респонс (пустой). Далее проверяется это новая сессия или нет. Дальше кэш-менеджер смотрит, надо ли отправлять на ресурс или нет. Допустим нет, тогда из кэша берет данные и назад их отсылает. А если в кэше нет данных, то идет на фильтр. Фильтр может отослать на другой фильтр (тогда будет цепочка) или на ресурс, а может решить, что не будет никуда отсылать.

Так же у нас есть контекст. Он хранит информацию. Используем его, если нужно что-то сохранить во всех сессиях! Он общий для всех сессий. Листенеры - они нужны, если нам нужная какая-то обработка дополнительная в жизненном цикле нашего приложения.

Дальше request + response переходят на программу http-сервера и в этой программе есть http-обработчики (хэндлеры) и эти обработчики принимают request и response. Из request программа, которая находится на сервере понимает, что именно клиент хочет получить от нее, эта программа заполняет response.

Обработчик формирует нам response и шлет обратно на http-драйвер. Http-драйвер преобразовывает в битовую последовательность и отправляет клиенту.

Клиент получает ответ в виде битовой последовательности, преобразовывает в ответ, который понятен для него - html-разметка. Действия повторяются несколько раз. На 1 запрос - 1 ответ. Так как много клиентов можно быть, а серверов меньше, то чтобы не перегружать сервер он удаляет все экземпляры, которые создаются при запросе.

**Таким образом у нас объекты**

1. Реквест

2. Респонс

3. Сессион

4. Куки с указателем на сессию на СЕРВЕРЕ

5. Контекст (кот. создаётся на базе конфигурационного файла)

6. Фильтр

7. Кэш с программой кэш менеджер

8. Http-драйвер

1. ASP.NET Core (на основе лекции и конспекта)

.NET Core – это платформа для разработки ПО (Фреймворк ОС)

Свойства: кроссплатформенность и открытый код

ASP.NET Core – программная платформа для разработки web-приложений на основе .NET Core

Платформа – это набор библиотек и инструментов для разработки

.NET Core: альтернатива .NET, своя реализация CoreCLR и библиотеки CoreFX.

.NET Core = CoreCLR + библиотеки CoreFX.

ASP.NET Core: программная платформа, разработанная Microsoft и предназначается для разработки web-приложений. Является развитием технологии OWIN (The Open Web Interface for.NET). Katana – OWIN-совместимый хост, разработанный Microsoft.

!!!Главная особенность – кроссплатформенность. Она достигается за счёт существования разных версий этой платформы для разных ОС. Она состоит из CoreCLR + библиотеки CoreFX и эти 2 компонента машинозависимы и существуют для разных ОС.

**ASP.NET Core:**

**-** является фреймворк с открытым кодом

- может работать над .NET Core и над полной .NET.

- приложения могут работать под IIS (Windows) или под web-сервером **Kastrel** (кроссплатформенный вариант).

- благодаря модульности, все отдельные компоненты загружаются через Nuget, нет необходимости применять библиотеку System.Web.dll (основная библиотека для приложений ASP.NET).

**-NuGet** – программный менеджер для установки компонентов приложения оформленных в виде пакетов

- модульность – …

- pay-for-play – подтягиваем только то, что используем

- CLR и JIT также устанавливаются как компонент

**-** типы приложений MVC UI, Web API, Web Pages. Web Forms не поддерживается.

- все приложения имеют одну и ту же структуру

- есть встроенное внедрение зависимостей

- есть интерфейс командной строки CLI, с помощью которого можно создавать и запускать приложения, там есть команда dotnet

- в основе asp.net core лежит интерфейс OWIN

- последняя версия 8.0

Развертывание приложений:

1. FDD – Зависящее от платформы развертывание (Framework-Dependent Deployment, FDD) — это метод развертывания .NET приложений, при котором приложение зависит от уже установленной на целевой системе платформы .NET. В этом случае приложение использует общий runtime, установленный на машине.

2. SCD – Самодостаточное развертывание (Self-Contained Deployment, SCD) — это метод развертывания .NET приложений, при котором приложение включает в себя все необходимые компоненты, включая .NET runtime. Таким образом, приложение не зависит от установленной на целевой системе платформы .NET. Приложение включает все необходимые зависимости и может быть запущено на любой системе без необходимости устанавливать .NET runtime.

.NET runtime – набор приложений и библиотек необъодимых для работы приложения

dodnet.exe точка входа для любого asp приложения

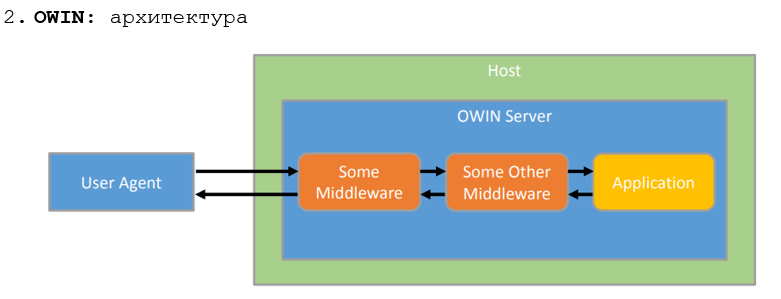
Фреймворк находится в с/програм файлс/дотнет/shared

Там на фотке ещё какая-то муть была, но тут без нее

2. OWIN

Open Web Interface OWIN – описывает взаимодействие компонентов **серверной** части.

OWIN: Open Web Interface for NET. OWIN – интерфейс между net-web-сервером и серверным приложением (обработчиками запросов). Основная цель OWIN отделить web-сервер от серверного приложения. Можно разрабатывать отдельно сервер и приложение.



Приложение состоит из 4 компонентов

1. Host – приложение-процесс операционной системы, управляющий жизненным циклом OWIN Server.

2. OWINServer - http-сервер, реализующий интерфейс OWIN. Превращает побитовые запросы клиента в Request, а Response преобразует в побитовый ответ клиенту

3. Application (обработчик запросов)

4. middleware – конвейер обработки запросов и ответов; подключенные компоненты (модули), предназначенные для обработки запросов.

Есть 5 компонент – внедрение зависимостей

**OWIN:** обеспечивает интерфейсы: между приложением и http-сервером, между http-сервером и Host, между http-сервером и middleware.

**Katana:** Microsoft-реализация OWIN сервера (говорят проект Katana). В качестве Host можно использовать IIS или self-hosting. Кроме того, позволять подключать модули middleware; предоставляет набор классов для работы с сервером и механизм подключения приложения к серверу.

3. Структура проекта

ASP.NET Core: Startup.cs – класс Startup, содержащий логику обработки входящих запросов, выполняет инициалзацию, сейчас заменён на Program.cs

**wwwroot** – папка для статического контента (html, css, js,…), **статический контент не отображается по умолчанию**.

отображение статического контента, необходимо подключить дополнительные nuget-пакеты; при создании Core-проекта автоматически подключен мегапакет Microsoft.AspNetCore.All в котором все есть.

**ASP.NET Core:** файлы для скачивания \

ASP.NET Core: добавление папок для статических файлов

4. Основы веба

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

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

**Программирование в Internet:** разработка распределенного (сетевого) приложения, представляющего собой один или несколько связанных Internet-ресурсов. Сайт – частный случай приложения, Вычислительное облако – частный случай приложения.

Технологии программирования в Internet:

- LAMP (Linux, Apache, MySQL, PHP/Perl, кроссплатформенная технология);

- Java EE (кроссплатформенная технология на основе Java);

- ASP.NET (ASP.NET CORE – кроссплатформенная версия);

- NODE.JS (кроссплатформенная технология, на основе JavaScript);

- Ruby on Rails(кроссплатформенная технология на основе Ruby и фреймворка MVC для web-разработки);

- Python Django (кроссплатформенная технология на основе Python и фреймворка MVC для web-разработки).

Программирование в Internet: в большинстве случаев для связи между компонентами приложения применяются протоколы HTTP, WebSocket, SMTP, POP3, IMAP, WebDav.

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