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

**HTTP** – прот.перед.гипертекста – прот.прикл.ур-ня перед.д-х (изнач.– в виде гипертекст.док-ов).

Основа – технол.«К-С», т.е. предполаг.сущ-ние К, кот. инициир. соед. и посыл.запрос, и С, кот. ожид. соед. для получ.запроса, вып.необх.д-вия и возвращ. обратно сообщ.с рез-том.

**Спецификация HTTP**

* Название: HyperText Transfer Protocol
* Уровень (по модели OSI): прикладной
* Семейство: TCP/IP
* Порт/ID: 80/TCP
* Назначение протокола: доступ к гипертексту, ныне стал универсальным
* Спецификация: RFC 1945, RFC 2616

|  |  |
| --- | --- |
| Достоинства | Недостатки |
| \*простота – леко созд.К прилож  \*расширяемость  \*распространенность | \*большой р-р сообщ  \*нет поддержки распределнности |

**!Осн. свойства HTTP:**

- версии http/1.1 (текстовый), http/2

- 2 вида сообщений: req, res => асинхр (неравноцен К и С, разный формат сообщ)

Сообщ, отправленные клиентом – ***запросы***; сообщ, отправленные сервером - ***ответы***.

- 2 типа абонентов: К –req–> <–res– С

- 1 req : 1 res и наоборот (иначе ошибка)

- TCP-порты: **80** (для серверов, не поддерж. шифр) и **443** (поддерж шифр; HTTPS)

- для адресации исп. **URI** (часть без http://имя\_пк)

- поддерж. W3C, описан в неск RFC (2616)

|  |  |
| --- | --- |
| **!Структура Request:**  - метод (метка get/post)  - URI  - версия протокола (http/1.1)  - заголовки (имя+знач)  - параметры (имя+знач)  - расширение (тело) | **!Структура Response:**  - версия протокола  - код сост (1хх, 2хх, 3хх, 4хх, 5хх)  - пояснение к коду сост  - заголовки (имя+знач)  - расширение |

**Request-методы:**

\*OPTIONS – исп.для опред-ния возм-стей веб-С или пар-ров соед для рес-са.

\*GET – узнать содержимое ресурса; м. начать какой-то процесс и передать инфу

\*HEAD – анал.get. Отличие – отсутствует тело в ответе С.

\*POST – для передачи пользовательских д-х задан.ресурсу.

\*PUT – загрузки содержимого на указ.URI (не было – созд; есть – измен.)

\*DELETE – удаляет указ.рес-с

\*TRACE – вовзращ.запрос так, что К м.увидеть какую инфу промеж.С доб/измен в запросе

connect, extension-method

|  |  |
| --- | --- |
| **Заголовки:**  \* general (д.ключ.в люб.сообщ К/С)  \* request (т. в запросах)  \* response (исп т. в ответах)  \* entity (опис содержимое тела – ContentType) | **Код сост. Response:**  \* 1хх: информ. сообщение  \* 2хх: успешный ответ  \* 3хх: переадресация  \* 4хх: ошибка клиента (404)  \* 5хх: ошибка сервера |

**Протокол без сохранения состояния** (Stateless protocol) – протокол передачи данных, который относит каждый запрос к независимой транзакции, которая не связана с предыдущим запросом, то есть общения с сервером состоит из независимых пар запрос-ответ. Примерами таких протоколов являются протоколы IP, HTTP и др.

1. Протокол TLS. Принципы работы протокола HTTPS.

на К дб сертификат удост центра для проверки сертификата Сервера

HTTPS - нельзя кешировать, ограничения в сессии

**HTTPS** (Hypertext Transport Protocol Secure) – это протокол, который обеспечивает конфиденциальность обмена данными между сайтом и пользовательским устройством. HTTP – как транспорт.

**Протокол TLS** основан на ассиметричном криптовании, на этапе соединения они договариваются как криптуются. Сначала договариваются, а потом передают.

Безопасность информации обеспечивается за счет использования криптографических протоколов SSL/TLS, имеющих 3 уровня защиты:

1. Шифрование данных (Позволяет избежать их перехвата)

2. Сохранность данных (Любое изменение данных фиксируется)

3. Аутентификация (Защищает от перенаправления пользователя)

**Что такое SSL?** SSL является аббревиатурой для Secure Sockets Layer. Это тип цифровой безопасности, которая позволяет зашифровать связь между веб-сайтом и веб-браузером. Технология в настоящее время устарела и полностью заменена TLS.

**Что такое TLS?** Transport Layer Security - обеспечивает конфиденциальность данных так же, как и SSL.

Поскольку SSL фактически больше не используется, это правильный термин, который люди должны начать использовать.

**Что такое HTTPS?** Это безопасное расширение HTTP. Веб-сайты, устанавливающие и настраивающие SSL/TLS-сертификат, могут использовать протокол HTTPS для установления безопасного соединения с сервером.

Цель SSL/TLS — сделать соединение безопасным для передачи конфиденциальной информации, включая личные данные, информацию о платеже или регистрации.

Это альтернатива простой передаче текстовых данных, в которой ваше соединение с сервером не зашифровано, и это затрудняет мошенникам и хакерам отслеживание соединения и кражу ваших данных.

**Как установить HTTPS на Tomcat:**

1. изгот приватный ключ

2. изгот запрос наосн. прив. ключа (+отвеч кто мы) и отправляем на ЦС

3. ЦС выдает нам (серверу):

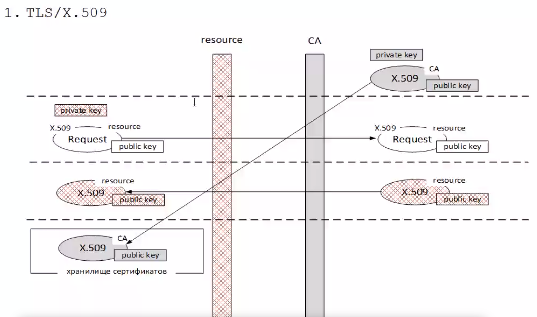
*\* Сертификат X.509* – эл. док, выдается серверу ЦС-ом. Содержит:

- имя держателя, адрес, серийный номер сертификата, даты проверки

- ЭЦП цс

- публ. ключ держателя (прочесть ЭЦП)

*\* Приватный ключ* ЦС – зашифр. ЭЦП

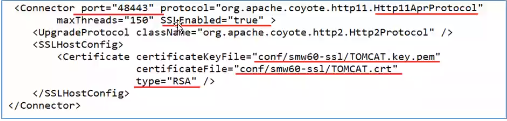


Есть К и С, кот верят одному ЦС. В их хранилище дб установлены эти сертификаты этого ЦС в разделе Доверенные сертификаты корневых удостов. центров.

Для С дб собс сертификат + сертификат удост. центра  
Если эти 2 компонента есть, мы м настроить свой сервер чтобы он мог поддерж https-соединение.

* приватный ключ
* сертификат X.509 с публ ключ ом
* сертификат удост. центра (сразу устан в хранилище)

есть **ген. формат** - используется, чтобы представить ключи.  
\* залезть в файл Tomcat/conf/server.xml – там прописать соединение *Connector*  
***redirect*** – если сделаем например HTTPS на 48080, выдаст редирект (302). Выдаст на тот же адрес, но с портом 48443 (redirectPort).



Предалагаю этот *порт 48083* использ для https.  
*Http1Appr* - внутренний java-вский протокол абстр. уровня, кот. строится над HTTPS  
*SSLEnabled* - разрешаем SSL  
*maxThreads* - макс кол-во соединений по порту  
*UpgradeProtocol* – м апргрейдить соединение до http2-протокола  
Дальше указ. сертификаты, *type=RSA* – для криптографии.

**Реально происходит:**

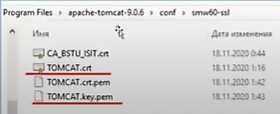
1) соединияемся по http1

2) рукопожатие

3) в первой посылке «отправить» посылаем 2 тега: *connection upgrade* и *upgrade* до http2.

4) сервер готов (код 101) = он апгрейдится до http2 (chrome, ff, safari точно это делают)

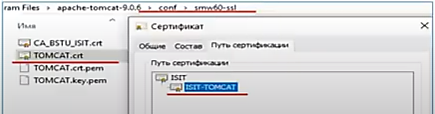
5) надо реально изготовить файлы:



6) скачать *tcnative-1.dll* в tomcat / bin в любом директории, кот. есть в PATH

т.к. tomcat требует нативную библиотеку, чтоб обесп. работу с сертификатами

7) на сервере tomcat надо устан. сертификат в хранилище (mmc) и проверить его (должен залиться в конфиг. файл)



8) при первом старте зайти в */logs* и проверить такие записи: https 48443, убедиться, что он его апргрейдил

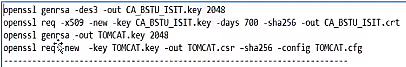
9) Потом проверяем соединение: *isit-tomcat:48443*

должен стоять замочек слева. (Прописали в dns). По умолчанию root-приложение запускается.

**TLS** - *прослойка* между http–tcp. Эта аутентификация на основе сертификата X.509.

мы не замечаем что есть шифрование, работаем как с обычным HTTP, скрыто на более низком уровне стека протоколов

**Изгот. сертификат - OpenSSL**



***Схема работы TLS:***



**Схема работы TLS:**

1) К лет запрос на соед. + список шифронаборов

2) С выбир ифронабор + шлет сертификат (откр. ключ)  
3) К проверяет сертификат на ЦС

4) К генер. сеансовый ключ (по ДиффиХ) для симм шифрования + шлет сеанс. ключ, зашифр. откр. ключом

5) С расшифр. сеанс. ключ своим закр. ключом

6) Далее данные будут перед. по защищ. каналу, шифр и расшифр с пом. симм ключа, пока соединение не завершено

**Из вопросов лк:**

* TLS - прослойка между TCP, HTTP.
* Сертификат хранит в себе публичный ключ, информация о издателе, владельце.
* Рукопожатие - шифронаборы, клиент отправляет возможные свои поддерживаемые шифронаборы, сервер выбирает один, отправляет его обратно вместе с сертификатом. Получается аутентификация сервера.
* На стороне клиента должен быть сертификат того центра, который выдал сертификат серверу, чтобы он мог с помощью ключа проверил сертификат сервера.
* Https поверх TLS

1. Протоколы HTTP2 и HTTP3. Push-Server.

НОВЫЕ ВОЗМОЖНОСТИ HTTP2:  
(HTTP 3 тоже эти возможности сохранил, но уже на новом уровне)  
\* ***tcp*** устан соединения:  
+ надежная передача д-х  
+ есть обмен спец сообщениями в рамках tcp, м проверить живо ли соединение  
- надо устан соединение для к. чиф(?)  
- установка коннекта дорогостоящая, потом они еще сохраняются keepAlive  
+ бинарная сериализация д-х: данные отпр в *бинарном виде* (они сжаты)

**+ мультиплексирование:**

В Http1: есть какая-то страница (http, css, img) - и для каждого из них устан соединение.  
В Http2: при *мультиплексировании* ­– в рамках одного соед tcp , если всё нах. по одному адресу, организуется неск. потоков (раб. независимо друг от друга), кот. фактически эмитировали *отдельное соединение*. + В HTTP2 можно установить *приоритетность для потоков* в рамках мультиплексированного соединения => могли обеспечить более раннюю загрузку картинки или css.



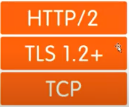
**+ псевдо-заголовки:**

Кол-во заголовков HTTP2 увеличилось. Раньше то, что мы передавали в полях http-запросов и ответов (метод, uri, статус), теперь передается в *псевдо-заголовках*: у них в имени впереди стоит “:”  
Эти заголовки сильно сжимаются => быстрее

**+ server-push:**

эта возможность нарушает правила http (1 req = 1 res). Можно отпр на 1 запрос неск ответов, но с нек. ограничениями == сервер может делать *уведомления* в рамках соединения.  
HTTP/2 протокол по соглашению всех разработчиков браузеров не работает без https  
=> остается установить https на application server, в котором собираемся работать на Java

**+ шифрование:**

Http2 обесп шифрование с пом. *TLS* 1.2+ и исп. *TCP* в кач-ве транспорта  
Уже на этапе разработки HTTP2 хотели заменить TCP (не обесп. всех потреб) => google разработал *Speedy*-протокол. Но это сложно: все ОС включено трансп. уровня TCP, UDP в кач-ве базового. На уровне железа знают о TCP (коммутаторы, шлюзы…).

2) ТЕПЕРЬ ПРО HTTP3:

**HTTP3** – еще в черновом варианте но уже опубликована. Уже несколько отошел от TCP. Другое название - **HTTP-over-QUIC** - это название произошло от протокола гугла *gQUIC* - фактически протокол, кот. пришел на смену Speedy.  
Сегодня 7% сайта уже поддерживают HTTP3.

**Стек протоколов:**  
HTTP3 - это просто новый синтакис HTTP2 (семантика та же) == они совместимы. Под ним находится протокол *QUIC*, кот. включает в себя TLS 1.3 (в нем есть все его ср-ва шифрования, handshake). Сохранился транспорт для QUIC – UDP. UDP - фактически слабая оболочка над IP. Все требования TCP (правильность пакета, правильная последовательность..) теперь берет на себя QUIC. QUIC фактически оборачивает UDP и создает новую реальность над UDP, кот. раньше обеспечивалась протоколом TCP.

**Явные преимущества QUIC:**  
+ **handshake (уст. соед.)**:

в Http2 состояла из 3ёх обменов сообщениями: К 🡪С, потом C 🡪K, К 🡪С, при повторном соединении он обеспечивал за две пересылки повторное соединение.

QUIC handhake: за один обмен осущ. полностью соединение, при повторном соединении ничего не надо: сразу за 0 посылок будем работать в рамках этого соединения.

Сама процедура установки соединения сильно упростилась и ускорилась в рамках этого протокола.

+ **работа на ур. приложений**:

Если HTTP и TLS находились в рамках ОС, то теперь всё это вынесено в отдельных библиотеках. Т.е. если хотим работать с HTTP 3, то мы должны отдельно скачивать QUIC, либо например в ASP.NET уже включен этот QUIC. Он уже включается *не на уровне ОС*, а *на уровне приложений*. Не стали ждать обновления ОС, который включит в себя QUIC. Пока его включили на уровне тех приложений, которые собираются поддерживать HTTP3: ASP.NET – IIS, ASP.NET CORE – kestrel-сервер или OWIN, Apache – Apache Server

Большой перечень библиотек, обеспеч. работу QUIC, кот. поддерживают HTTP 3. Среди них  
Производители: Mozilla, Facebook. Google, MS. Полный спектр языков, ведуще компании. Уже поддерживается Chrom-ом с 2019 (canar версия), Firefox-ом и Safari.

**Основные сведения QUIC:**  
\* протокол поверх *UDP*, исп. как транспорт  
\* поддерж все возможности *TCP*, обесп. безопасность, правильную последовательность д-х в рамках потоков, проверку на искажение  
\* включает в себя внутрь протокол *TLS*  
\* прототип его - протокол *Speedy* (трансформир. в googleQUIC 🡪 QUIC)  
\* обесп. connection от udp  
\* обесп несколько потоков  
\* ошибки передачи в одном потоке не влияют на другой поток  
\* в рамках 1 QUIC соедиенния работает несколько потоков, каждый работает независимо друг от друга, идет мультиплексирование над UDP.  
\* потоки легки: это является основой для создания бесшумных WiFi зон  
\* если у нас есть нек. пр-во с подпростр-вом, каждое из кот. обесп. своей Wi-Fi-точкой, то в процессе перемещения по пр-ву, мы к. раз начнем переключения на новый IP-адрес. Каждая точка Wi-FI обслуж своим роутером, у каждой свой IP. В HTTP/2 надо новый коннект на новый IP (процедура в TCP достаточно сложная).   
В UDP же все пакеты передаются независимо друг от друга => переключение в QUIC становится более легковесным, простым. Обеспечение работы бесшумного вай-фая становистя намного проще.  
\* шифрование вутри, протокола TLS не нужно устанавливать  
\* реализован QUIC на уровне приложения

**Компрессия заголовков:**HTTP2 ориентирован на соединение, на гарантированно правильную передачу пакетов. UDP такого не обеспечивает. Сжатие в HTTP2 настроено т.о., что пакеты д. идти в правильном порядке. Поэтому компрессия заголовков тоже есть в QUIC, но она отличается в силу особенностей UDP.

**Проблема с NAT:**  
*NAT* – спец. прогр. механизм, с пом. к-го можем обеспечить выход компа в локальной сети через 1 белый IP (установив его ч/шлюз). Можем через локальную сеть работать с интернетом, используя 1 ip-адрес.  
*Проблема:* NAT Не поддерживает HTTP3, QUIC, потому что он ориентирован на TCP-соединение.

**Есть несколько реализаций QUIC:** - MsQUIC (использ. IIS-сервер, написал драйвера для К и С)  
- NGINX (написал драйвера только для С)  
- AppleQIUC (для К и С)

ИТОГО:  
Для программистов всё равно: писать на HTTP2 или HTTP3. Есть приложения, кот написаны на HTTP1, переключение на HTTP3 безболезненно. Если работаем в Asp.NET – переключение произойдет на уровне IIS-сервера. Семантика протокола останется такая же: uri, заголовки, параметры – всё останется на месте. Для нас HTTP3 – это просто новый синтаксис. Всё изменилось на уровне транспорта. Незаметно для нас Google уже работает с HTTP3. Если посмотреть режим отладки хрома, будет заголовок alt-svc (указ. инфу клиенту о том, как он может переключиться на HTTP3 и какая его версия уже поддерж).

если пользуемся утилитой curl, то она уже поддерживает ключ --http3

1. Протокол WebDav.

**WebDAV** - *Web Distributed Authoring and Versioning* - это протокол, позвол. доступиться к удаленной ФС, используя формат данных HTTP и поддерж. совместную работу (неск. юзеров могут работать в общем хранилище)

+ **потокобезопасен**: можем одновременно в неск. соединений взаимод. с этим файл.хранилищем  
+ можем **блокировать**, чтобы предотвратить конфликты совместной работы

\* исп. в с-мах документооборота

\* альтернатива протоколов **FTP**, **SNB** (они для представления ФС)

\* работает с: DropBox, OneDrive, Яндекс.Диск, Google Drive, iDisk

\* расширяет возможности **HTTP** унасл. 8 --> 34 методов

\* **RFC**: 2291, 4918, 3648, 3744, 3253 (их много, т.к. протокол долго совершенствовался)  
\* **версионность**: в хранилище мб один файл разных версий, м. доступаться к этим версиям (как в git)

**Diskstation** - фактически (application) сервер, кот. используется как файловый сервер, кот. мы можем конфигурировать, редактировать права доступа отдельных пользователей и тд   
В Томкат тоже есть отдельные пакеты, обесп. протокол WebDAV

**Унаследованные методы:**  
\* *GET* - скачать файл  
\* *PUT* - загрузить файл на сервер  
\* *DELETE* - удалить серверный объект

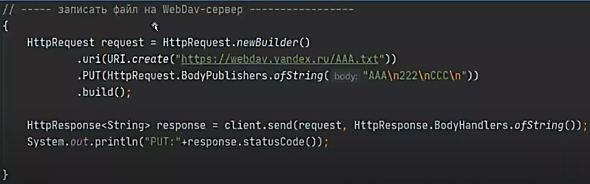
Адрессация осущ. с помощью uri

Будем рассматривать работу *Клиента*. Есть API, с пом которого м создавать свой собственный сервер. Сам сервер предст из себя прокладку между веб интерфейсом, кот. обеспечивает этот сервер, и существующей ФС. Он фактически принимает запросы по webdav протоколу в формате http и работает с той ФС, кот есть на данном компе.  
Поддержка календаря, всяких систем поиска.

**1 пример: Яндекс.Диск, используя ASP.NET**  
\* зарегаться на Я.Диск и получить имя и пароль  
\* создать папку: обычный запрос: *req.Method = WebRequestMethods.Http.****MkCol***  
в WebResuestMethod уже поддерж методы типа MkCol авто- (встроенный http-клиент)

\* в url в конце доб. имя папки/файла

|  |
| --- |
| string *url = “https://webdav.yandex.ru/ имя\_папки ”;*  HttpWebRequest req = HttpWebRequest.Create(***url***) as HttpWebRequest;  req.Credentials = new NetworkCredential (“**логин**”, “**пароль**”);  HttpWebResponse res = (HttpWebResponse) req.GetResponse();  HttpStatusCode code = res.StatusCode; |

**2 пример: Яндекс.Диск, используя Java sdk 11**  
в состав sdk уже входит клиент, кот. позволяет в том числе работать с webdav-протоколом  
\* создать клиент (HttpClient.newBuilder()…), указ. имя, пароль, версия http  
 используя этот объект, выполнять действия с удал. хранилищем:  
**Запись файла на сервер:**  


\* создать запрос (HttpRequest.build() ) :

- .uri : в конце имя файла, кот. хочу создать

- .PUT : входит в API

После аутентификации Яндекс нас выведет на свою собственную папку. Запишем AAA.txt. Дальше делаю GET-запрос, где в body указываю что записать в файл.  
Проверяю, чтобы статус был из серии 200.

**Создать папку:**  
  
метод MkCol позволяет создавать папку (xxx)  
опять жду ответ со string (там ничего нету) и проверяю статус ответа.  
*MkCOL* - расширение протокола HTTP (в стандартный протокол не входит)  
*COPY* - тоже расширяющий метод + надо указать заголовок *Destination* (а откуда указ. в uri)  
*MOVE* - тоже заголовок *Destination  
PROPFIND* - позвол. прочитать св-ва объекта, в uri пропис. этот объект, жду в ответ body string. Что за файл, какой размер, кто владелец

Метод ***POST*** тоже можно использовать в webDAV сервере, но он для экзотических случаев, если мы собираемся самостоятельно обрабатывать эти команды на стороне webdav-сервера.

1. Протокол HTTP: HTTP-аутентификация.

**Oauth:**протокол **авторизации** (выдать права!), позвол. выдать одному сервису (приложению) права на доступ к ресурсам юзера на другом сервисе. Избавляет от необходимости доверять приложению логин и пароль + позвол. выдавать ограниченный набор прав, а не все сразу.

**Идентификация** – заявление пользователя о себе.

**Аутентификация** – процедура проверки подлинности идентификатора юзера (логин и пароль, как правило). На основании какого-то секрета.

**Авторизация** – после аутентиф; проверка есть ли права на выполнение тех или иных действий.

Исп-ся **1 код возврата** (401 (и 200 само собой))

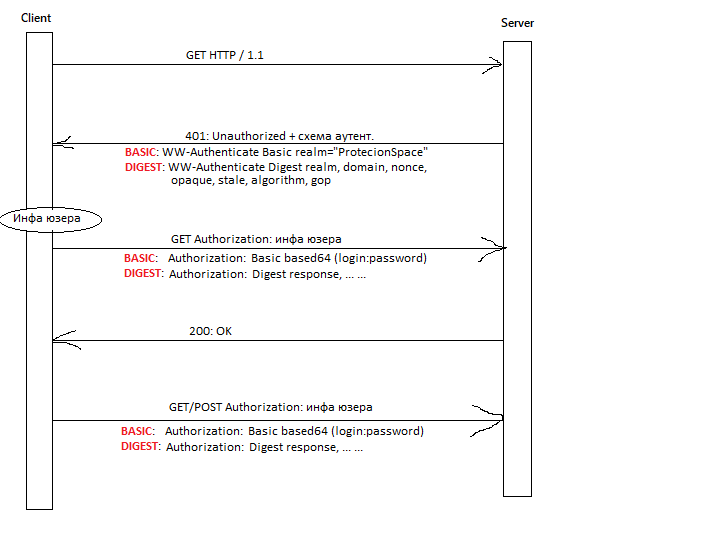
401 код – Unauthorized – ваш запрос является не авторизированным.   
407 код – для прокси серверов.

**2 заголовка**, кот.мы будем использовать:

\*authorization

\*www-authenticate

Все открыто, злоумышленники могут работать вместо клиента, надо шифровать.



**Способы аутентификации:**

1. BASIC
2. 7617 протокол

**Realm** – То, что знает клиент (твоя аутентификация действует в рамках этого сектора), т.е. определяет область действия аутентификации.

Если Клиент – браузер, то 401 код заставляет зажечь окошко логина пароля

Base64 строка – сцепленные через «:» имя и пароль в заголовке authorization.

С получает, проверяет валидность, если все хорошо – 200, если нет – 401, все сначала

1. Дайджест - DIGEST

7616 протокол

Особенность в том, что указывается не имя и пароль, а хэш имени и пароля.

У С есть БД, где существует хеш и остается проверить валидный хеш или нет.

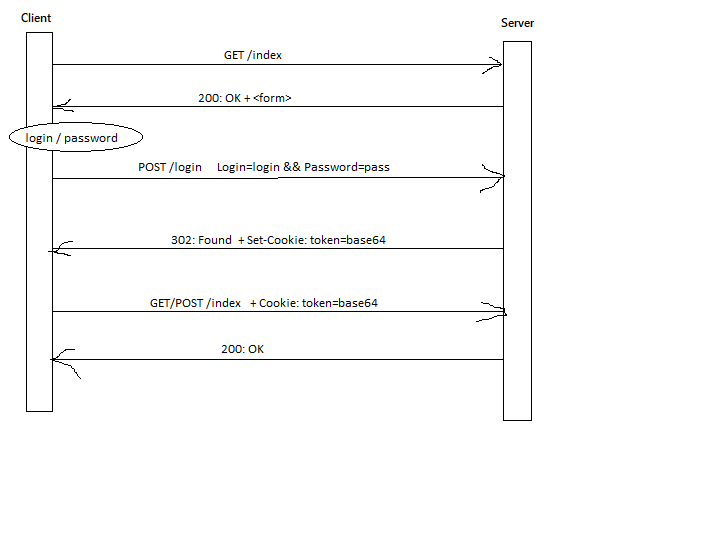
Является большей защитой, чем basic.

С т.зр. криптографии – также беззащитен как basic.

Исп-ся реже, т.к. не защищает (ур-нь защиты почти как у basic), но больше мороки => исп-ют в основном basic.

1. FORMS

Нет стандарт (нигде не описан = народный способ)



Инфа о пользователе и пароле передается в куки

Процедура аутентификации:

Идет запрос

С проверет есть ли куки:

- есть – проверяет проверяет содержимое, есть инфа, кот.интересует и она валидна – отвечаем ресурсом;

- нет – отпр. страницу с формой.

Юзер вводит и жмет сабмит, эта инфа отправляется на С. С обрабатывает запрос и формирует на основании имени и пароля токен (бит.посл-сть, кот. С м.проверить на валидность; имеет время жизни). Отправляется на К SetCookie И указывает токен, который должен отправляться клиентом в каждом запросе.

Используется чаще всего, т.к.простой и понятный.

Все 3 способа – слабые. Если не исп-ть HTTPS – данные легко перехватить и модифицировать.

1. HTML. Структура HTML-страницы. Каскадные таблицы стилей (CSS). Модель DOM. Жизненный цикл HTML-страницы.

**HTML** (HyperText Markup Language) — это язык гипертекстовой разметки страницы. Он используется для того, чтобы дать браузеру понять, как нужно отображать загруженный сайт.

|  |  |
| --- | --- |
| <!DOCTYPE html>  <html>  <head>  <meta>  <title>  </head>  <body>  </body>  </html> | \* *Doctype* – элемент, для указания типа текущего документы – DTD (document type definition – описание типа документа). Это необх чтобы браузер понимал, как нужно интерпретировать текущую веб-страницу.  \* *Html* – начало html документа, внутри которого находят head и body  \* *Head* – голова страницы – содержимое не показ. на странице, кроме title  *Meta* – можно изменить кодировку страницы, добавить ключевые слова, описание документа и др.  \* *Body* – тело документа – для размещения элементов и содержимого стр  6 типов заголовков – h1-h6  <!—комментарий -->  <p> - абзац текста (блочный, всегда начинается с новой строки) |

Почему мы заботимся о валидации HTML-документа?

* **Правильность**: валидный документ корректно отображается в браузере.
* **Отладка:** некорректный код HTML может вызвать ошибки, сложные для выявления.
* **Поддержка**: валидный документ легче обновлять позже, даже кому-то другому.

**Стили** представляют собой набор свойств, управляющих видом и положением элементов веб-страницы.

Где можно прописать CSS?

* Css как атрибут

Можно прописать в элементах html с помощью атрибута style

<p style="color: red;">Это важный текст.</p>

* Css в <head>

Использовать тег <style> внутри <head>

<head>

<style>

p { color: red;}

</style>

* Css в отдельном файле

Записать css в отдельный файл .css, а затем связать его с html с помощью тега link

<link rel="stylesheet" href="style.css">

Третий метод более предпочтительный. Это нужно для того, чтобы отделить содержимое от представления. Это делает обслуживание проще: тот же css-файл мб использован на всем сайте. А также обеспечивается гибкость: содержание страницы обеспечивается с одной стороны, а стилизация с другой.

Браузер, когда запрашивает страницу и получает в ответе от сервера её исходный HTML-код, должен сначала его разобрать. В процессе анализа и разбора HTML-кода браузер строит на основе него **DOM-дерево.** После выполнения этого действия и ряда других браузер приступает к отрисовке страницы. В этом процессе он, конечно, уже использует созданное им DOM-дерево, а не исходный HTML-код. **DOM – это объектная модель документа, которую браузер создаёт в памяти компьютера на основании HTML-кода, полученного им от сервера.** Если сказать по-простому, то HTML-код – это текст страницы, а DOM – это набор связанных объектов, созданных браузером при парсинге её текста.

Есть 3 цровня DOM: DOM1, DOM2, DOM3; современные браузеры уровня 2 с элементами ур 3.

*В Chrome исходный код страницы, который получает браузер, можно посмотреть во вкладке «Source» на панели «Инструменты веб-разработчика».*

*В Chrome нет инструмента, с помощью которого можно было бы посмотреть созданное им DOM-дерево нет. Но есть представление этого DOM-дерева в виде HTML-кода, оно доступно на вкладке «Elements». С таким представлением DOM веб-разработчику, конечно, намного удобнее работать. Поэтому инструмента, который DOM представлял бы в виде древовидной структуры нет.*

DOM документ после его формирования можно изменять. При изменении DOM браузер практически мгновенно перерисовывает изображение страницы. В результате у нас отрисовка страницы всегда соответствует DOM.

Для чтения и изменения DOM программно браузер предоставляет нам **DOM API**.

**DOM API** – это набор огромного количества различных объектов, их свойств и методов, которые мы можем использовать для чтения и изменения DOM.

Зачем нам нужен DOM API? Он нам нужен для того, чтобы мы могли с помощью JavaScript изменять страницу на «лету», т.е. делать её динамической и интерактивной. Т.е. данный программный интерфейс позволяет нам создавать сложные интерфейсы, формы, выполнять обработку действий пользователей, добавлять и удалять различные элементы на странице, изменять их содержимое, свойства (атрибуты), и многое другое.

Жизненный цикл **HTML-страницы** состоит из трех важных событий:

* **DOMContentLoaded** — браузер полностью загрузил **HTML-код** страницы и построил дерево **DOM**. Но внешние ресурсы, такие как изображения <img> и таблицы стилей, могут все еще загружаться;
* **onload JavaScript** — браузер загрузил все ресурсы (*изображения, стили и т. д*.);
* **beforeunload/unload** — когда пользователь покидает страницу.

Каждое событие может использоваться для различных целей:

* Событие **DOMContentLoaded** — **DOM** готов, поэтому обработчик может искать узлы **DOM**, инициализировать интерфейс.
* Событие **load** — загружаются дополнительные ресурсы, и можно получать размеры изображений (*если это не указано в****HTML / CSS***) и т. д.
* Событие **beforeunload/unload** — пользователь покидает страницу. Можно проверить, сохранил ли пользователь изменения, и спросить его, действительно ли он хочет уйти.

1. JavaScript. Основные стандарты. Типы данных. Программные структуры. Принцип применения. Понятие DHTML.

**JS –** то, что делает живым веб-страницы.

**JS** – (объектно-ориентированный) язык программирования, разработанный для записи "сценариев", последовательностей операций, которые пользователь может выполнять на компьютере.

Подключение js на страницу делается в <script> (либо в head, либо в body)

Скрипты – это программы, которые не требует компиляции.

**ОСНОВНЫЕ СТАНДАРТЫ**

**ECMAScript**, или ES, это название стандарта, которым руководствуются разработчики JavaScript-движков. Самой свежей версией стандарта на момент публикации этого материала является [ES9](http://www.ecma-international.org/ecma-262/9.0/index.html), принятая в июне 2018 года.

Проще говоря, ECMAScript — стандарт, а JavaScript — самая популярная реализация этого стандарта. Существует 11 версий этого стандарта.

**ES1** была выпущена в июне 1997 года, **ES2** — в июне 1998 года, **ES3** — в декабре 1999 года, **ES4** — так и не была принята.

**ES5** был выпущен в декабре 2009 года, спустя 10 лет после выхода третьего издания. Среди изменений можно отметить:

* поддержку строгого режима (strict mode);
* аксессоры getters и setters;
* возможность использовать зарезервированные слова в качестве ключей свойств и ставить запятые в конце массива;
* поддержку JSON.

Версия **ES6/ES2015** вышла в июне 2015 года. С выходом этого пакета обновлений комитет принял решение перейти к ежегодным обновлениям. Поэтому издание было переименовано в ES2015, чтобы отражать год релиза. добавлено [деструктурирующее присваивание](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) ([a, b] = [1, 2];);

* добавлены [стрелочные функции](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/Arrow_functions) ((singleParam) => { statements });
* let и const — альтернативы var для объявления переменных. Добавлена «временная мертвая зона»;
* добавлены промисы.

**ES2016 (ES7)**, **ES2017 (ES8), ES2018 (ES9), ES2019 (ES10), ES2020 (ES11)**

**ТИПЫ ДАННЫХ**

JavaScript является слабо *типизированным* или *динамическим языком*. Это значит, что вам не нужно определять тип переменной заранее. Тип определится автоматически во время выполнения программы.

Стандарт ECMAScript определяет ***9(8) типов***:

* 6 типов данных являющихся *примитивами*:
  + [Undefined](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/undefined) (Неопределенный тип)
  + [Boolean](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/Boolean) (Булев, Логический тип): У него всего два значения: true (истина) и false (ложь).
  + [Number](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/Number) (Число): используется как для целых, так и для дробных чисел.
  + [String](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%B0) (Строка)
  + [BigInt](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/BigInt): способ представления целых чисел
  + [Symbol](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/Symbol): новый примитивный тип данных, который служит для создания уникальных идентификаторов
* [Null](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/Null) (Null) отдельный тип, состоящий из единственного значения null. Это просто специальное значение, которое имеет смысл «ничего» или «значение неизвестно»;
* [Object](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/Object) (Объект): Простая структура, используемая не только для хранения данных, но и для создания других структур, где любая структура создаётся с использованием ключевого слова [new](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/new): new [Object](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object), new [Array](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array), new [Map](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Map), new [Set](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Set), new [WeakMap](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/WeakMap), new [WeakSet](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/WeakSet), new [Date](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date) и множество других структур;
* (?)и [Function](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F): Специальный случай, упрощающий определение типа для Функций, несмотря на то, что все функции конструктивно унаследованы от Object.

**ПРОГРАММНЫЕ СТРУКТУРЫ**

**Функции**

JS поддерживает функции. Есть встроенные функции: alert(message). А также можно писать свои собственные. Вначале идет ключевое слово function, после него имя функции, затем список параметров в скобках и тело функции – код, который выполняется при её вызове.

**Циклы**

Для многократного повторения одного участка кода – предусмотрены циклы.

**while** – проверка условия перед каждым выполнением. Пока условие верно – выполняется код из тела цикла: while (условие) { код, тело цикла}

**do…while** – проверка условия после каждого выполнения. do { тело цикла } while (условие); – сначала выполняет тело, а затем проверяет условие.

**for** – проверка условия перед каждым выполнением, а также дополнительные настройки. Применятся чаще всего. for (начало; условие; шаг) { тело цикла }

Прерывание цикла: **break**.

Директива **continue** прекращает выполнение текущей итерации цикла (прерывает только текущее выполнение его тела, как будто оно закончилось).

**Условные операторы**

**if** («если»). Получает условие, вычисляет его, и если результат – true, то выполняет команду. if (условие) { код }

**else** («иначе»). Необязательный блок выполняется, если условие неверно. if (условие) { код } else { код }

**else if**. Когда необходимо проверить несколько условий. else if (условие) { код } else if (условие) { код } else { код }

Оператор вопросительный знак '**?**'. Состоит из трех частей:

условие ? значение1 : значение2. Проверяется условие, затем если оно верно – возвращается значение1, если неверно – значение2.

**DHTML**

**Dynamic HTML** или **DHTML** — это способ (подход) создания интерактивного [веб-сайта](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B0%D0%B9%D1%82), использующий сочетание статичного языка разметки [HTML](https://ru.wikipedia.org/wiki/HTML), встраиваемого (и выполняемого на стороне клиента) [языка](https://ru.wikipedia.org/wiki/%D0%A1%D0%BA%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B2%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) [JavaScript](https://ru.wikipedia.org/wiki/JavaScript), [CSS](https://ru.wikipedia.org/wiki/CSS) и [DOM](https://ru.wikipedia.org/wiki/Document_Object_Model) без увеличения загрузки сервера.

Т.е., DHTML — не особый язык, а совокупность *клиентских* технологий веб-разработки, которую можно представить в виде такой условной «формулы»:

DHTML = HTML + CSS + DOM + JavaScript.

Другими словами, определенные действия посетителя ведут к изменениям внешнего вида и содержания страницы без обращения к серверу.

На WEB-страницах можно обеспечить реакцию на определенные действия посетителя или изменения состояния документа или окна, которые вызывают определенные события.

Посетитель генерирует события при передвижении мыши, нажатии кнопок мыши и клавиатуры. Изменения состояния документа генерируют события при загрузке документа, изображений или объектов, при появлении ошибки на странице или переходе фокуса от одного элемента к другому.

**Особенности:**

- регистрозависим

- слабая типизация (переменные могут динам. менять тип):

var xNumber;  // тип undefined

xNumber = 45;  // тип number

xNumber = "45"; // тип string

- есть преобраз. д-х:

ParseInt(), ParseFloat()

если преобр. не удалось = *NaN* (Not a Number)

- есть массивы: new Array()

- мат. операции: + - \* / % ++ - - (префикс. и постфиксный)

- typeof - получ. тд

- alert - выскак. уведомление

promt - диал. окно с предложением ввести значение

function - и вызов функции обязателен

- м. подкл js-файл с пом <script src=…>

к. команда заканч на ‘;’

**Переменные** предназн.для хранения данных. Создать переменную – var / let;

**Константы –** const – нельзя изменять.

**Инициализация** (присвоение нач.знач) - с пом =

Когда браузер получает веб-страницу с кодом html и javascript, то он ее интерпретирует. Результат интерпретации в виде различных элементов - кнопок, полей ввода, текстовых блоков и т.д., мы видим в браузере. Интерпретация веб-страницы происходит последовательно сверху вниз.

Когда браузер встречает на веб-странице элемент <script> с кодом javascript, то вступает в действие встроенный интерпретатор js. И пока он не закончит свою работу, дальше интерпретация веб-страницы не идет.

JS практически всегда вставляется перед закрытием body когда практически вся страница загружена – повыш.произв.

***Возможные вопросы от Смелика:***

* 1. **Создание объекта**

Пустой **объект** («пустой ящик») можно создать, используя один из двух вариантов синтаксиса:

let user = new Object(); // синтаксис "конструктор объекта"

let user = {}; // синтаксис "литерал объекта"

* 1. **конструкторы**

**Функции-конструкторы** являются обычными функциями. Но есть два соглашения:

1. Имя функции-конструктора должно начинаться с большой буквы.
2. Функция-конструктор должна вызываться при помощи оператора "new".

*Например:*

function User(name) {

this.name = name;

this.isAdmin = false;

}

let user = new User("Вася");

alert(user.name); // Вася

alert(user.isAdmin); // false

Когда функция вызывается как new User(...), происходит следующее:

1. Создаётся новый пустой объект, и он присваивается this.
2. Выполняется код функции. Обычно он модифицирует this, добавляет туда новые свойства.
3. Возвращается значение this.

**3) prototype**

Кроме непосредственного определения свойств и методов в конструкторе мы также можем использовать свойство prototype. Каждая функция имеет свойство prototype, представляющее прототип функции. То есть свойство User.prototype представляет прототип объектов User. И любые свойства и методы, которые будут определены в User.prototype, будут общими для всех объектов User.

User.prototype.hello = function(){

document.write(this.name + " говорит: 'Привет!'<br/>");

};

User.prototype.maxAge = 110;

**4) “use strict”**

На протяжении долгого времени JavaScript развивался без проблем с обратной совместимостью. Новые функции добавлялись в язык, в то время как старая функциональность не менялась.

Преимуществом данного подхода было то, что существующий код продолжал работать. А недостатком – что любая ошибка или несовершенное решение, принятое создателями JavaScript, застревали в языке навсегда.

Так было до 2009 года, когда появился ECMAScript 5 (ES5). Он добавил новые возможности в язык и изменил некоторые из существующих. Чтобы устаревший код работал, как и раньше, по умолчанию подобные изменения не применяются. Поэтому нам нужно явно их активировать с помощью специальной директивы: "use strict".

Современный JavaScript поддерживает «классы» и «модули» — продвинутые структуры языка, которые автоматически включают строгий режим. Поэтому в них нет нужды добавлять директиву "use strict".

**5) promise**

**Promise** – это специальный объект, который содержит своё состояние. Вначале pending («ожидание»), затем – одно из: fulfilled («выполнено успешно») или rejected («выполнено с ошибкой»).

***Способ использования***, в общих чертах, такой:

1. Код, которому надо сделать что-то асинхронно, создаёт объект promise и возвращает его.
2. Внешний код, получив promise, навешивает на него обработчики.
3. По завершении процесса асинхронный код переводит promise в состояние fulfilled (с результатом) или rejected (с ошибкой). При этом автоматически вызываются соответствующие обработчики во внешнем коде.
4. Методология Ajax. Структура Ajax-приложения, принципы разработки и применения. Объект XMLHTTPREQUEST, промис FETCH.

**AJAX** (Asynchronous JS and XML) – методология, основ.на возм-сти XMLTHHPRequest асинхронно отправлять запросы С и обрабатывать ответы.

**AJAX** – технология обращения к С без перезагрузки страницы. Уменьшается время отклика.

\* асинх: пока запрос не выполнен, || выполн. остальной js-код

\* js: использ. для создания запроса ajax, парсинга ответа и обновл dom

\* xml/json: формат передачи данных

В основе AJAX:

* Html
* JS
* XML (eXtensible Markup Language) — расширяемый язык разметки данных.
* DOM
* HTTP
* JSON (JavaScript Object Notation) — текстовый формат обмена данными, применяемый обычно в сценариях JavaScript.
* XMLHttpRequest
* Callback – функция обратного вызова – функция, которая вызывается после того, как пришел ответ от сервера, для его обработки

**Структура AJAX** - для каждой части страницы должен создавать свой объект xmlHttpRequest (он живёт в браузере).

**Особенности:**

+ динамичность: сокращ сетевой трафик

+ нет перегрузок html-страниц.

– основан на http == требует работу в режиме «запрос-ответ», чтобы получить порцию данных от С необх.сделать запрос. В прилож., задача кот реагировать на соб, кот происх.на С, клиентской части приходится период.выполнять запросы к С, чтобы увидеть эти события. Для того, чтобы не пропустить серверное событие необх.увеличить частоту запросов.

**Объект XMLHTTPRequest**

var req = new XMLHttpRequest(); //создаем новый объект XMLHttpRequest

req.open("GET", "<http://localhost/hello.txt>", false); //конфигур его: гет запрос на ури, синх

\* тип запроса

\* адрес запроса

\* t - асинх, f - синх

\* мб + логин пароль, если для вып. нужна аутент

req.send(); //отсылаем запрос

Если код ответа сервера не 200, то это ошибка.

if (xhr.status != 200) { // обработать ошибку

} else { // вывести результат}

**Свойства XMLHttpRequest** позвол.контролить вып-ние запроса

* status – статусный код ответа http от С
* statusText – текст статуса
* responseType – тип ответа (text, json, document, blob, arraybuffer, “”)
* response – ответ С
* responceText – текст ответа
* responceXML – xml, если формат ответа от С – хмл

**Асин. запросы**

Работы сложнее чем с синхр., т.к.надо обраб.событие redystatechange объекта xmlhttprequest.

При ас.запросе xmlhttprequest исп св-во readystate для хранения сост.запроса

* 0: объект создан, open() не было
* 1: open() был, send() не было
* 2: запрос отправлен, заголовки и статус ответа получены
* 3: ответ получен от сервера
* 4: выполнение запроса полностью завершено (даже если ошибка)



**Что можно делать с помощью AJAX:**

* Элементы интерфейса – полезен для форм, кот связаны с действиями – добавить в корзину
* Динамическая подгрузка страницы
* Живой поиск

С пом AJAX К (браузер) общается с С и запрашивает у него д-е. Полученный ответ обрабатывается, в страницу вносятся изменения без полной перезагрузки, обновляется лишь часть DOM.

**Как на С определить ajax?**

есть заголовок xhr (или это значение в каком-то заголовке)

можно увидеть в Dev Tools

Метод **fetch**() — современный и очень мощный, поэтому начнём с него. Он не поддерживается старыми (можно использовать полифил), но поддерживается всеми современными браузерами.

Запрос с помощью **fetch** возвращает Promise (обещание — которое будет выполнено или отклонено). Для того чтобы получить данные из промиса, нам нужно включить в цепочку два колбека с помощью метода then.

* Делаем запрос, получаем промис.
* После того как промис выполнен успешно, получаем специальный объект ответа. Выполним метод объекта ответа json(), чтобы получить данные.
* Метод json() возвращает промис, так что далее снова вызываем then и в колбеке выводим данные в консоль.

**Базовый синтаксис:**

let promise = fetch(url, [options])

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

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

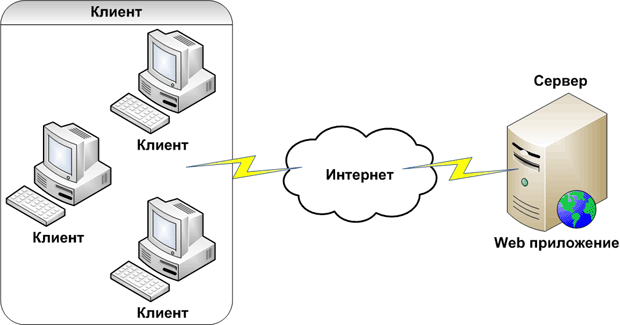
1. Web-приложение. Архитектура web-приложения. Особенности реализации web-приложения. Web-сервер и web-клиент.

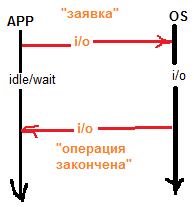
***Web-прил*** – прил с КС архитектурой, где К и С взаимод. по протоколу HTTP (прикл.ур)

***КС прил*** – прил, сост из 2 компонент: К и С, кот. взаимод. в соотв. с правилами (*протоколами*)

***Клиент*** – часть прил, явл. инициатором соед. Между К и С дб устан *соединение*

**Клиентская часть** реализует [пользовательский интерфейс](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81), формирует запросы к серверу и обрабатывает ответы от него. **Серверная часть** получает запрос от клиента, выполняет вычисления, после этого формирует [веб-страницу](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%86%D0%B0) и отправляет её клиенту по сети с использованием [протокола](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D0%B5%D0%B2%D0%BE%D0%B9_%D0%BF%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB) [HTTP](https://ru.wikipedia.org/wiki/HTTP).



**Web-сервер:**

принимает запросы, обраб их и отпр ответы К

чтение http-запроса i/o

парсинг http-запроса cpu

запросы к БД i/o

запросы к др. серверам i/o

вычисления cpu

форматиров. ответа cpu

отправка http-ответа i/o

В основном, web-сервер делает в(ы)вод, мало использует процессор

Под всеми к-дами в(ы)вода скрыт системный вызов функции ядра ОС!

Пока ждем ответ ядра ОС, это время надо рационально исп.

*“проблема блокирующего в(ы)вода”* – если ничего в это время не делаем – **синхр. операция** (процессор простаивает)

**Решение:**

1) многопоточность (огран кол-во потоков, т.к. каждый требует доп. памяти) – в Apache

2) асинхр прогр – паттерн Reactor – в Nginx

**Типы веб-приложений:**

1. Backend (бэкенд или серверная часть приложения) работает на удаленном компьютере, который может находиться где угодно. Она может быть написана на разных языках программирования: PHP, Python, Ruby, C# и других. Если создавать приложение используя только серверную часть, то в результате любых переходов между разделами, отправок форм, обновления данных, сервером будет генерироваться новый HTML-файл и страница в браузере будет перезагружаться.
2. Frontend (фронтенд или клиентская часть приложения) выполняется в браузере пользователя. Эта часть написана на языке программирования Javascript. Приложение может состоять только из клиентской части, если не требуется хранить данные пользователя дольше одной сессии. Это могут быть, например, фоторедакторы или простые игрушки.
3. Single page application (SPA или одностраничное приложение). Более интересный вариант, когда используются и бэкенд и фронтенд. С помощью их взаимодействия можно создать приложение, которое будет работать совсем без перезагрузок страницы в браузере. Или в упрощенном варианте, когда переходы между разделами вызывают перезагрузки, но любые действия в разделе обходятся без них.



**Основными особенностями web-приложения являются среда его выполнения и многокомпонентная структура.** **Среда выполнения web-приложения** – это web-контейнер, который, как правило, встраивается в web-сервер или в сервер приложений. Web-контейнер является набором классов и интерфейсов, имеющих то же предназначение, что и API (Application program interface) для операционных систем. Web-приложение ничего не «знает» о среде, в которой работает web-контейнер и тем более сервер, – это является одним из необходимых условий обеспечения независимости web-приложения от платформы. В состав web-приложения могут входить объекты различного назначения: java-классы, html-файлы, xml-файлы, графические файлы и т. п.

**Разработка web-приложения сводится к разработке его компонентов**, размещению их в определяемой спецификацией Java Servlet структуре директорий web-сервера и подготовке специального xml-файла, называемого дескриптором развертывания web-приложения (web.xml). HTML-файлы, Java Servlet и Java Server Pages являются тремя основными типами компонентов, входящих в состав web-приложения.Дескриптор развертывания web-приложения содержит информацию, необходимую web-контейнеру для взаимодействия с приложением.

**Для вызова web-приложения, как правило, используется web-браузер.** Для вызова web-приложения необходимо в адресной строке набрать его адрес (Uniform Resource Locator, URL – унифицированный локатор ресурса). В простейшем случае адрес представляет собой строку Hhttp://cccc:8080/pppH, где http – протокол связи, ссс – разрешаемое символическое имя или ip-адрес компьютера, 8080 – номер tcp-порта (этот номер, как правило, используется в java-серверах по умолчанию) и ppp – имя web-приложения (оно совпадает с именем директории web-приложения).

1. Web-приложение: принципы сохранение состояния на сервере.

м. на стороне С и К

на С: с пом сессий, как правило

Преимущества хранения данных на стороне сервера:

* Более высокий уровень секретности
* Уменьшение количества передаваемых данных

Имеется два способа хранения общей информации для набора web-страниц без отправки их клиенту

* Cостояние приложения (Application state)
* Состояние сессии (Session state)

Информация из состояния приложения доступна всем страницам для всех пользователей. Информация из состояния сессии доступна для всех страниц открытых одним пользователем в течении одного его посещения сайта. Оба состояния теряются при перезапуске приложения (application restart). Для сохранения состояния между запусками приложения его нужно сохранять в профиле пользователя (profile properties).

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

Для получения объекта сессии в сервлете у объекта **HttpServletRequest** определен метод **getSession()**. Он возвращает объект **HttpSession**.

***HttpSession session = request.getSession();***

**Сессия** - это сеанс работы одного пользователя, в ходе которого запросы пользователя не поступают через интервал времени больший, чем задано в установках сервера. Он позволяет сохранить информацию на одной странице и использовать ее на другой.

Каждый пользователь приложения имеет разные сессии и разные коллекции информации. Session state идеально подходит для хранения такой информации, как «корзина для покупок», когда пользователь переходит между страницами сайта.

*Управление состоянием сесс*ии (Session state) является трудоемким способом, который требует дополнительных ресурсов процессора и памяти сервера. Это не часть HTTP стандарта. ASP.Net присваивает каждой сессии 120 битный идентификатор. Только этот id передается между web- сервером и пользователем.

Методы для управления сесиией:

* ***setAttribute(String name, Object o)***: сохраняет в сессии под ключом name
* ***getAttribute(String name)***: возвращает из сессии объект с ключом name. Если ключа name в сессии неопределено, то возвращается null
* ***removeAttribute(String name)***: удаляет из сессии объект с ключом name
* **setMaxInactiveInterval(int seconds)**: устанавливает интервал неактивности в секундах. И если в течение этого интервала клиент был неактивен, то данные сессии данные удаляются. По умолчанию максимальный интервал неактивности 1800 секунд. Значение -1 указывает, что сессия удаляется только тогда, когда пользователь закрыл вкладку в браузере.
* **invalidate(**): удаляет из сессии все объекты

**Контекст web-приложения** - серверный объект, предназначенный для хранения информации об одном web-приложении, общий для всех серверных объектов. 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.

Для получения контекста приложения может использоваться метод **getServletContext()** объекта **HttpServlet**. Он возвращает объект **ServletContext.**

Для работы с атрибутами контекста используются следующие методы.

**getAttribute(String name)**- получает атрибут с именем name

**getAttributeNames()** - получает имена всех атрибутов

**setAttribute(String name, Object object)** - сохраняет в контексте под ключом name

**removeAttribute(String name)** - удаляет из контекста атрибут с именем name

1. Web-приложение: принципы кэширования на стороне клиента.

с пом LocalStorage или исп. куки

м. гонять инфу в uri (там запис. длинные ссылки)

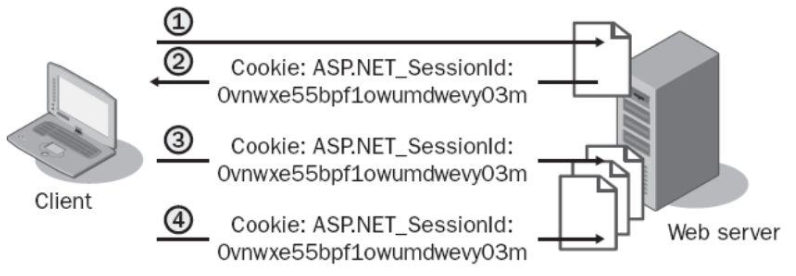
преимущества хранения данных на стороне клиента:

* Лучшая масштабируемость – на сервере не используется память. При большом количестве пользователей могут возникать проблемы с памятью на сервере
* Поддержка множества веб-серверов – можно распределять входящие запросы между мн-вом одновременно работающих серверов

**Cookies**

* Web-приложение может хранить небольшие части данных на компьютере клиента используя cookies.
* Cookies это небольшое текстовые файлы, которые хранятся в файловой системе пользователя или в оперативной памяти в сессии работы браузера клиента (если cookies временные).
* Web-приложение создает cookie отправляя его в заголовке HTTP ответа.
* Web-browser запоминает этот cookie и отправляет его на данный сервер в каждом новом запросе.
* Наиболее гибкий и надежный способ хранения данных на стороне клиента.
* Однако пользователь может удалить cookies на своем компьютере в любое время.
* Могут легко использоваться любой страницей приложения и сохраняться между посещениями сайта.

**Использование cookies для отслеживания web-клиента**



* Наиболее частое cookies используются для идентификации пользователя при его посещении нескольких страниц.
* В cookies могут храниться информация о состоянии, предпочтения пользователей или зашифрованные элементы, которые показывают, что пользователь был успешно идентифицирован.

1. Утилита ANT: назначение, принцип работы, применение.

назначение и принципы работы

какие есть команды, таски

в чем отличие от мавена

С-ма сборки **Ant** опис. XML-ом и сост. из target (цель), все цели именованы. Вначале в <project> указ. <default-copy>, т.е. его цель – вып. copy  
Когда дойдет в <target-copy> увидит depend? По стеку до init будет подним… Надо знать структуру папок  
А если делаем **Maven** (Смелову не нравится), все там по умолч, скрыто

***Apache Ant*** – средство автоматизации, созданное на базе Java и XML, которое представляет собой инструмент, позволяющий автоматизировать процесс компоновки java-проекта. Сюда относится компиляция java-файлов, создание java-библиотек (jar-файлы) и специальных файлов для размещения приложения на сервере (war-файлы), а также выполнение задач, связанных с файловой системой (создание, перемещение, копирование, удаление файлов и директорий). Важной особенностью утилиты Ant является ее независимость от платформы и возможность расширения.

Необходимым условием для инсталляции утилиты **Ant** является предварительная установка **Java Development Kit**.

Порядок компоновки web-приложения описывается в файле сборки, который содержит входную информацию для утилиты **Ant**. Файл сборки – это xml-файл, описывающий с помощью xml-элементов и атрибутов последовательность действий, которые должна исполнить утилита **Ant**. По умолчанию имя файла сборки – **build.xml**. Если в параметрах вызова команды **ant** не указан путь, то предполагается, что он находится в текущей директории.

Файл сборки имеет единственный корневой элемент **<project>**. Внутри него может находиться один или несколько элементов **<target>** (эти элементы называют ***целями***), которые объединяют группу элементов ***задач***. Каждой задаче соответствует свой тег. Например, задача, выполняющая компиляцию java-файлов, описывается с помощью тега **<javac>** , задача копирования файлов – с помощью тега **<copy>** , задача создания jar-файла – с помощью тега **<jar>** и т. д.

**Пример**

|  |
| --- |
| **<project name="ANaive" default = "copy" >**  **<target name="init">**  **<echo message="Building ANaive Project" />**  **</target>**  **<target name="create" depends ="init">**  **<delete dir= "C:\Program Files\Apache Software Foundation\Tomcat**  **6.0\webapps\ANaive\" />**  **<mkdir dir= "C:\Program Files\Apache Software Foundation\Tomcat**  **6.0\webapps\ANaive\" />**  **<mkdir dir= "C:\Program Files\Apache Software Foundation\Tomcat**  **6.0\webapps\ANaive\WEB-INF" /> </target>**  **<target name="copy" depends ="create">**  **<copy file= "D:\Workspace\_ANaive\WEB-INF\web.xml"**  **todir="C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\ANaive\WEB-INF\" />**  **<copy file= "D:\Workspace\_ANaive\html\index.html" todir="C:\Program Files\Apache Software Foundation\Tomcat**  **6.0\webapps\ANaive\" /> </target>**  **</project>** |

**<echo>** (задача echo) предназначен для вывода на консоль сообщения **«Building ANaive Project»**, указанного атрибутом **message**.

Цель **create** включает три задачи. Первая удаляет указанную атрибутом **dir** директорию вместе ее содержимым. Если указанной директории нет, то задача не выполняется и ошибки не возникает. Две другие задачи **mkdir** создают директории, имена которых также указаны атрибутом **dir**.

Цель **copy** включает две задачи типа **copy**. Каждая из них копирует файл, указанный атрибутом **file**, в директорию, указанную атрибутом **todir**.

Два тега **<target>** содержат атрибут с именем **depends**, который указывает имена одной или нескольких целей, которые должны быть успешно выполнены предварительно.

В теге **<project>** атрибут **default** указывает цель, задачи которой должны быть выполнены.

Процесс создания внутри структуры директорий web-сервера структуры директорий web-приложения и заполнение их файлами web-приложения называется компоновкой web-приложения. Для выполнения компоновки, как правило, используются специальные программные средства, например утилиты Apache Ant и Apache Maven.

1. Спецификация Java Platform Enterprise Edition (Java EE): структура, состав технологий. Понятие Application Server (сервер приложений).

это стандарт прежде всего, кот был разработан САНом, перехвачен ораклом

опис. набор технологий, к-ми должен обладать Application Server

кто ее поддерживает: про comminuty рассказать, (оракл отказался)

**JEE** – не набор библ, а их описание! (это док), а реализацией заним-ся др. компании

На сегодняшний день имеется **три семейства Java-технологий:**

− Java Platform, Standard Edition (Java SE);

− Java Platform, Enterprise Edition (Java EE);

− Java Platform, Micro Edition Specification (Java ME).

В основе технологии Java EE лежат четыре основных документа:

− Java EE Platform Specification (спецификация платформы Java EE);

− Java EE Reference Implementation (образцовые реализации платформы Java EE);

− Java EE Blueprints (модель приложений Java EE);

− Java Compatibility Test Suite (набор тестов на совместимость платформы Java EE).

**Спецификация Java EE Platform** определяет компонентную структуру Java EE-приложения и содержит минимальный набор свойств, которыми должен обладать сервер приложений (Application server), поддерживающий эту платформу.

**Сервер приложений** – это сервер, умеющий исполнять прикладные программы, специальным образом установленные на нем. Если говорят о Java EE-сервере приложений, то подразумевается, что он соответствует некоторой версии спецификации Java EE и может исполнять Java EE-приложения. Java EE-сервера: Sun GlassFish Enterprise Server, IBM WebSphere Application Server, Oracle Application Server, JBOSS, BEA WebLogic и т. д.

Составной частью любого сервера приложений является web-сервер (его часто называют web-контейнером). В некоторых случаях это может быть отдельный продукт, который встраивается в сервер приложений, в других случаях web-сервер может являться неотделимой составной частью сервера приложений (например, GlassFish) или вообще могут использоваться как несколько различных web-серверов, так и собственный встроенный (WebSphere).

1. Java EE: структура Java EE, Web-Profile: состав технологий и их назначение.

часть технологий из JEE предназначены для разработки веб-прил

какой состав этих технологий

как они описаны (jsr или jrs)

входят: сервлет, jsp, сокеты, tag library…

для чего все эти технологии нужны

**JEE** – не набор библ, а их описание! (это док), а реализацией заним-ся др. компании

На сегодняшний день имеется **три семейства Java-технологий:**

− Java Platform, Standard Edition (Java SE);

− Java Platform, Enterprise Edition (Java EE);

− Java Platform, Micro Edition Specification (Java ME).

В основе технологии Java EE лежат четыре основных документа:

− Java EE Platform Specification (спецификация платформы Java EE);

− Java EE Reference Implementation (образцовые реализации платформы Java EE);

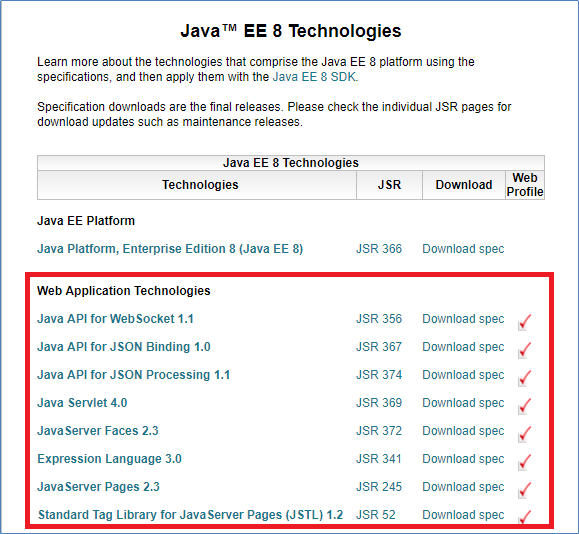
− Java EE Blueprints (модель приложений Java EE);

− Java Compatibility Test Suite (набор тестов на совместимость платформы Java EE).

**Спецификация Java EE Platform** определяет компонентную структуру Java EE-приложения и содержит минимальный набор свойств, которыми должен обладать сервер приложений (Application server), поддерживающий эту платформу.

Часть Java EE, необходимая для создания веб-приложений (servlet, jsp, web socket, taglib), называется **Web Profile’ом.**

Технологии Web Profile, позволяющие создавать web-прилаги:

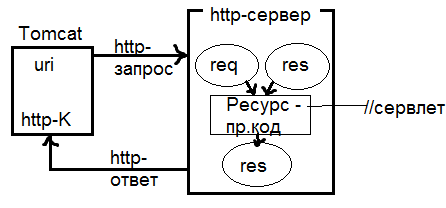


Технологии Web Profile, позволяющие создавать web-сервисы:

****

1. Java EE: сервер Tomcat, назначение, архитектура, конфигурация.

что умееет



**Tomcat** – сервер., напис. в соотв JEE (опис. appl.server), но это урезанный вариант appl-server-а (под ним сервлет)

**Tomcat** – контейнер сервлетов, это С, обесп работу спецификаций JEE

Цель контейнера сервлетов – поддержка сервлетов и обеспечение ЖЦ.  
JEE – документация, сервер внутри webapps  
Tomcat вып в кач-ве службы Windows

**Функции контейнера сервлетов:**

* Обмен данными между К и С
* Организация клиентскиз сессий
* Создание программной среды для функционирования сервлета
* Идентификация и авторизация клиентов
* Управление метаданными (заголовки, методы и т.д.)

**Структура папок Tomcat (архитектура):**

□ **conf** – Для конфигур С-а, там ряд файлов, позв настроить С:  
 *server.xml* – порт, протокол, redirectPort, connectTimeout [мс] – допуск на подключ

*web.xml* – опис. конфиг по умолч. Если нет web.xml прил-я, смотрим этот

*tomcat-users.xml* – юзеры, группы, роли. Поддерж basic, forms-аутент

□ **logs** – журналы – видим дату старта:  
 *catalina.log* – процесс загрузки сервера  
 *localhost\_access.log* – все, что кас. запросов  
 *tomcat9\_stderr(stdout).log* – все, что пишем

□ **bin** – исполняемые файлы (батники)  
 *tomcat9w.exe* – м. упр. работой Са

□ **webapps** – то что мы будем разраб

□ **common** – для хранения доступных для всех веб-приложений и ядра томкат jar-файлов и классов

□ **lib** – jar-файлы, доступные для всех веб-приложений и недоступных для ядра томкат

□ **temp / work** – используются для хранения промежуточных и временных файлов.

В папке webapps есть езе встроенные папки:

* Docs – документация томкат
* Examples – примеры веб приложений, которые используются для проверки инсталяции томкат
* Host-manager и manager – используется специальным web-приложением Tomcat Web Application Manager для выполнения операций, необходимых для установки и настройки веб-приложений.
* Root – для веб-приложения томкат, которое будет выполняться по умолчанию

Файл web.xml – конфигурационный файл web-приложения ANaive, который принято называть дескриптором развертывания web-приложения.

**Конфигурация:**

Файл **server.xml** — главный файл настроек Tomcat.

# Там имеется множество атрибутов, которые позволяют точно определить функциональные возможности. Чаще всего если необходимо внести какие-то существенные изменения в установку Tomcat, как, например, изменить число портов.

например, тэг «Connector»:



Файл **tomcat-users.xml** – настройка списка ролей, пользователей и паролей, которые томкат использует для аутентификации



**Что умеет Tomcat:**

\* запускать веб-приложения

\* откр порт для взаимод-вия с клиентом

\* настройка сессий

\* разворачивать свои веб-прил. в контейнере

\* реализ. servlet API, jsp

\* прослуш. http-запросы от К на порту, отпр эти д-е приложению, возвр юзерам ответы

1. Java EE: конфигурация приложение (web.xml)

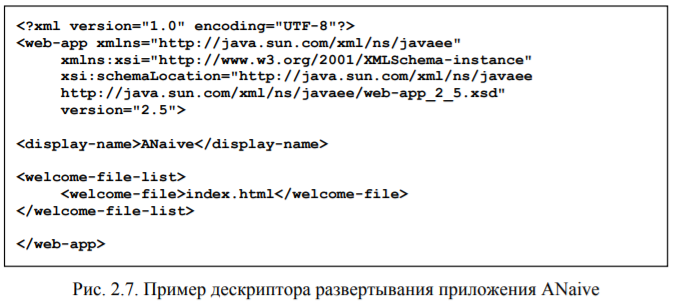
все-все теги, кот находится в web.xml

что там можно сделать: мэппинг, велком-файл-лист, листенеры, фльтры, констрейны для JAAS, параметры инициализации и тд

Файл web.xml – конфигурационный файл web-приложения ANaive, который принято называть дескриптором развертывания web-приложения.

Файл **web.xml** хранит информацию о конфигурации приложения. Он не является обязательной частью приложения Java EE.

Данный файл должен располагаться в папке **WEB-INF** приложения. При запуске Tomcat считывает его содержимое и использует считанную конфигурацию. Если же файл содержит ошибки, то Tomcat отображает ошибку.



Т.е. это xml-файл, корневым эл-том которого является <web-app>. Порядок элементов внутри и их синтаксис определяется схемой XML, которая, как это видно из параметров тега <web-app>, находится на http://java.sun.com/xml/ns/j2ee/web-app\_2\_5.xsd.

В самом простом случае дескриптор развертывания состоит только из одного тега – <web-app> , внутри которого ничего нет.

В нашем случае:

* <display-name> не является обязательным, но если есть, то не может повторяться более одного раза. Он предназначен для указания имени web-приложения, которое потом может быть использовано в графическом интерфейсе. Для этого имени не требуется уникальность, и его значение не оказывает влияния на работу приложения.
* <welcome-file-list> тоже не является обязательным и предназначен для указания списка стартовых страниц web-приложения
* <welcome-file> - имена файлов страниц внутри предыдущего тега

Дескриптор развертывания web-приложения содержит информацию, необходимую web-контейнеру для взаимодействия с приложением.

\* welcome-file-list: список стартовых страниц в порядке перечисления – м. размещ в web.xml прил-я (приоритетнее) или общем conf/web.xml

\* session-TimoOut

\* параметры иниц

1. Java EE: спецификация Servlet, назначение, основные возможности, принципы применения. Структура Servlet. Жизненный цикл Servlet.

как обесп совместимость сервлетов

структура сервлета

осн. методы из ин-са

ЖЦ сервлета

не гарантирует состояния

**Servlet** – серверный web-компонент JEE-приложения предн. для обработки запроса, формир и отправки ответа  
**Servlet** – базовая единица, элементарное веб-прилож – java-класс, реализ. ин-с Servlet. Любой запрос к томкату встречает сервлет (явный или нет)  
\* есть в составе tomcat  
\* есть кот. сами разработали

Количество сервлетов на web-сервере не ограничено. Один сервлет может отвечать на один или несколько типов запросов клиента. Можно говорить о двух крайних случаях реализации сервлетов: 1) для каждого типа запроса реализуется свой сервлет; 2) все запросы обрабатываются одним общим сервлетом.

Следует подчеркнуть, что сервлет управляется web-контейнером, т. е. имеет API, управляемый событиями (even-driven API). Загрузку, выгрузку сервлетов, вызов методов, предварительную обработку запросов и ответов клиента, а также многое другое выполняет **web-контейнер**.

В спецификации Java Servlet предусмотрена следующая **функциональность сервлета:**

1) прием и чтение данных, посылаемых клиентом в качестве запроса;

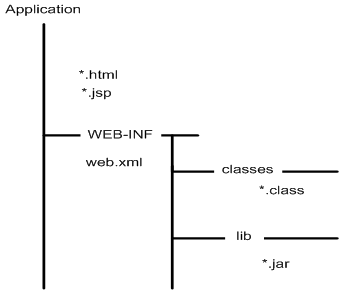
2) получение любой информации о запросе (свойства запроса, имя хоста-отправителя, свойства браузера и т. п.);

3) генерация и форматирование ответа на запрос; установка необходимых параметров ответа;

4) в процессе обработки запроса сервлет может обращаться к базе данных, использовать файловую систему, осуществлять необходимые вычисления и вызывать другие приложения

5) отсылка сформированного ответа клиенту. В дальнейшем изложение в основном будет ориентировано на сервлеты, обрабатывающие http-запросы.

**Структура папок Servlet:**

  
В папке classes откомпилированные классы объектов. Java-классы не компонуются, нах. по-отдельности. Мб папка WEB-INF – там ф., манифест с доп инфой

**Сервлет** – класс, реализ. ин-с Servlet (опосредованно : extends HttpServlet – это класс, кот реализ. ин-с Servlet)  
- есть к-р для иниц памяти и 3 метода:  
 \* **init** – 1 раз при созд объекта сервлета  
 \* **destroy** – вызыв сборщиком мусора  
 \* service

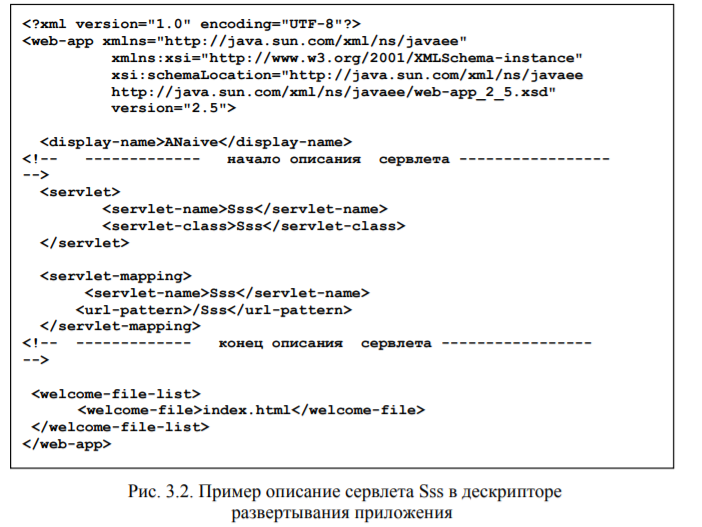
Init() и destroy() обесп **персистентность** (нек. св-ва S-a м. проиниц при его создании и при разруш S-a м. сохранить нек. св-ва), но сам S не гарант сохр. своего сост (stateless)

При обраб запроса, поступ на С, сраб сервлет. Если запрос 1-й, то new() servlet, init, service… (ему передаст req, res. Если серлвет уже создан, мб исп тот экземпляр или создан новый => stateless

1) Сервлет подразумев. переносимость на ур. байткода (пис. на Windows, получ. байткод, в Linux раб. без предкомпиляции)

2) Сервлет д. лежать в папках опр. структуры. Совместимость обесп и на ур. папок. Во всех appl-serv структура папок один. Перенесем папки WebSphere в Tomcat => все будет раб

Абсолютно **ЛЮБОЙ** запрос обраб сервлетом (встроен в Tomcat). Если запросим html, вызыв встроен. Ss, тот счит. html-файл и выводит на сторону К. (в любом appl-сервере в JEE)



<servlet> - информация о сервлете: имя <serlet-name> и класс <servlet-class>. Откомпилированный класс будет назодится в стандартной директории tomcat/webapps/WEB-INF/classes. Имя, задаваемое в <servlet-name>, не обязательно должно совпадать с именем класса сервлета.

<servlet-mapping> - указывает на суффикс URL (<url-pattern>), с помощью которого можно вызвать сервлет. В данном случае, для вызова сервлета в браузере нужно вбить: http://xxx:8080/ANaive/Sss

1. Java EE: переопределение http-запроса, переадресация запроса, формирование http-запроса.

|  |  |
| --- | --- |
| переопред запроса (forward) | вызыв. service() др. сервлета \* взаимод. осущ в рамках 1 прил! (передает req, res ч/строку кот. он получил от др. Ss \* возврата нет! ответ отпр тот компонент, на кот мы сделали forward |
| переадрес запроса (redirect) | отпр res на К, там код 3хх и в заголовке ‘Location’ адрес нового рес, на кот. К обязан сделать новый запрос метод res.SendRedirect() \* м. нах в др. прил, сервере, хосте, где угодно \* если код не 307 и 308, повторный запрос всегда GET! (даже если был POST изначально), иначе тот же метод |
| непосред. запрос | м непоср отпр запрос к др. ресурсу (Ss в рамках service() исп Http-клиент. Делаем http-запрос на др. ресурс (в любом месте) и получаем сюда же ответ, обраб в рамках этого service() метода |

**ИТОГИ:**

***Переопределение*** можно сделать, применив метод forward класса Request Dispatcher. Объект этого класса создается сервером, а ссылка на него становится доступной сервлету при помощи метода getRequestDispatcher класса HttpServletRequest.

***Переадресация*** выполняется с помощью метода sendRedirect. В качестве параметра метода sendRedirect указывается URI нового ресурса. Ресурсом, на который осуществляется переадресация, в общем случае, может быть любой компонент приложения, допускающий запрос типа GET.

Принципиальное отличие переадресации от переопределения запроса заключается в том, что передача управления от сервлета А к сервлету В осуществляется через посредника – клиента сервлета A.

***Для генерации http-запросов*** будем использовать пакет org.appache.commons.httpclient. В процессе своей работы классы httpclient вызывают методы еще двух пакетов – org.appache.commons.logging и org.appache.commons.codec. Для того чтобы пакеты были доступны для всех приложений web-сервера, необходимо поместить их в директорию Tomcat6.0/common/lib.

1. Java EE: спецификация Java Server Page (JSP), назначение, основные возможности, принципы применения. Структура JSP. Компоненты JSP. Жизненный цикл JSP.

перечислить, что есть в сервлете

ЖЦ

**JSP** – и есть сервлет в др. виде, определение то же

использ., чтобы проще написать сервлет, у кот. много статич. вывода  
JSP-стр = html-стр + директива ‘page’  
При 1-ом запросе JSP преобр в сервлет. (То же самое как Razor Engine: cshtml -> c# в 1 раз) => 1й запрос вып. долго

- Если много статич текста -> лучше JSP  
- Если много динамики -> лучше Сервлет  
- Если MVC -> лучше JSP (=view, формир содерж) + Сервлет (=к-р, обраб запрос + forward на JSP)

**JSP:** мб директивы, выражения, объявления, действия

**Директивы:**  
- page (старт страницы) **<%@ page language="java" contentType="text/html; charset=ISO-8859-1" %>**

- include (встав текст и код в процессе трансляции jsp-страницы, вкл др. страницу)  
- taglib (указ, что на jsp будет использ. библ тегов)

**Формы записи динам. контекста:**  
- объявления <%! .. %> (на уровне класса, потом использ для др. динам эл-тов)  
- скриплеты <% .. %> (место, кот. превращ в метод класса сервлета (мб service))  
- выражения <%= .. %> (элем JSP, позв вычислить что-то и подставить значение)   
req, res нигде не опред, это авто- объекы в JSP, вверху есть Import

**ЖЦ:** компиляция, init, service, destroy

1. Java EE: библиотека JSP-тегов, компоненты, назначение, основные возможности, порядок разработки.

что в дескрипторе, как создать

как создать класс обработчика

что они делают

какие бывают теги: с телом, без

как разместить, для чего надо?

Есть **библиотека тегов JSP** – спец механизм, с пом к-го м разрабатывать собс tdl-теги, кот можем применять в JSP-странице. Цели: разделить работу верстки и разработки  
1) создать дескриптор библиотеки (Tag library descriptor, TDL) в директории прил.  
2) создать обработчики тегов (Tag handler) - java-классы, генерирующие html-текст, замещают tdl-теги в выходном потоке jsp-страницы

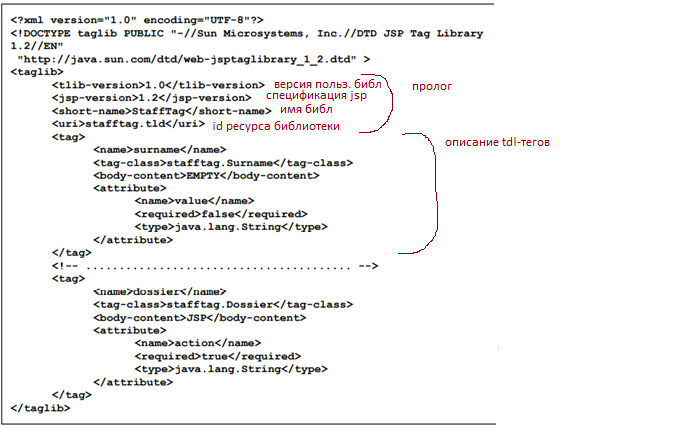
3) поместить на jsp-странице директиву taglib, указ. на месторасполож дескриптора библиотеки тегов и задающую префикс (пр-во имен) для имен tdl-тегов на данной странице

4) добавить tdl-теги на jsp-страницу

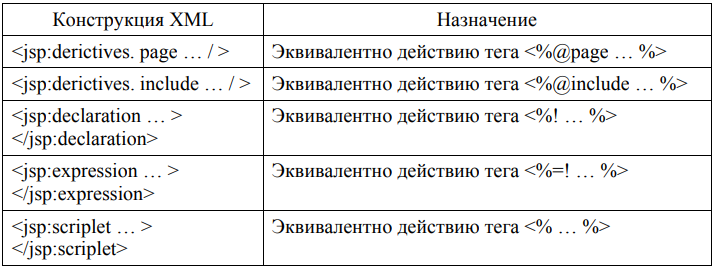
Все теги: кот содержат тело внутри (jsp), и не содержат (empty)

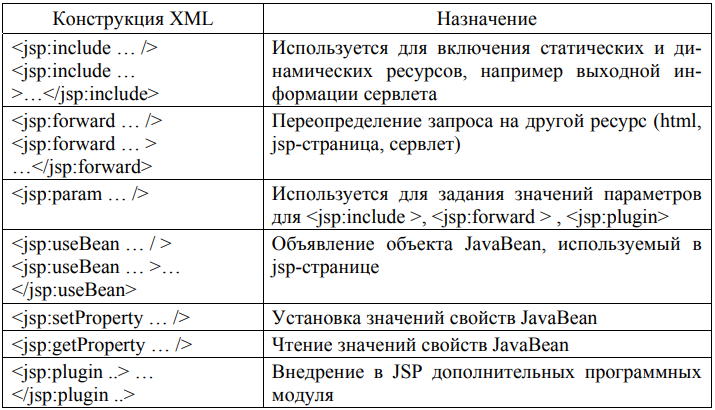
WEB несколько расширяет понятие сервлет, т.к. там мб еще вебсокеты и др. компоненты.

**Пример TLD (это xml-файл):**



Есть стандартные теги, поддерж. любым сервером:





Также есть готовая станд. библиотека тегов - JavaServer Pages Standart Tag Library (JSTL). Там широкая функц-сть, позвол. сократить время на разработку jsp-страницы. Позвол устан атрибуты объектов, выводить текст на jsp-страницах, организовать циклы обработки д-х, форматировать числа и даты, преобр xml-конструкции, применять sql для работы с БД.

Также есть специализир. библиотеки тегов: Session Tag Library (для работы с HttpSession), Mailer Tag Library (для работы с эл. почтой), Regexp Tag Library.

**Класс обработчик:**

extends TagSupport

@Override public int doStartTag(), также мб doEndTag (не обяз)

1. Java EE: основные модели web-приложений на основе технологий Servlet и JSP.

1) *servlet*: он встречает запросы, работает с БД и формир ответ. Используем это, если много динамики  
2) *servlet + jsp*: Встрчает сервлет, обраб и делает forward на jsp-страницу, а та формирует ответ на сторону К. JSP – фактически view, SS – контроллер. Самая частая схема.

3) *JSP*: мб самодостаточна, м сама принимать запросы, обраб их и возвр резульат  
JSP примен если в основе статический вывод. JSP - это html-страница с вкраплениями java-кода. Для ее исполнения нужен спец. контейнер (JSP Engine), кот. отвечает за парсинг страницы и преобрз. ее в сервлет, кот. при исполнении генерирует html-код.

1. Java EE: основные системные объекты (контекст, сессия, запрос, ответ), назначение и жизненный цикл объектов. Атрибуты системных объектов и принципы их применения.

В рамках контейнера работает наше прил, т.е. обращаемся к функциям из контейнера. Контейнер предлагает **4 системных объекта**, кот мы м исп в процессе работы:

1) **контекст прил**:

- созд. при старте сервера

- сущ. пока работает С (самый долгий)

- это общий объект для всего прилож

- предост. память, кот. хранится пока работает С, пока мы сами не удалим, garbage-мусорщик бессилен (*1 ур. памяти*) = межсессионный обмен

- в эту память помещ. коллекция *атрибутов* (имя : знач) + то, что в web.xml

м. сделать SetAttr, removeAttr, значение = часто ссылка на объект

м. создать объект класса, его ссылку записать в контекст

2) **session**:

– Это механизм сохр. состояния между запросами, t м/которыми не превыш sessionTimeout

- живет, пока жива серия запросов

- когда сессия удалена, ссылка на объект пропадает, garbage-cleaner его уберет (*2 ур. памяти*)

- эта серия созд. с пом cookie, там session id (128-число)

- к. запрос помечен к какой сессии он относ

- Если С не может определить сессию к к-й относ запрос, он создает новую сессию.

- у сессии есть *id* + *память* + *коллекция* (атрибуты имя : знач), кот. сохр в рамках жизни сесии

- Крайние случаи: оч редкие запросы (для каждого своя сесиия), или очень частые (одна сессия)

3) **request** – живет в рамках запроса, тоже содрежит атрибуты и память. (3 ур. памяти). Если хотим сохранить данные в рамках обработки 1 запрсоа, лучше в рамках объекта request. В виде setAttribute запис данные, когда запрос умрет, освободится ссылка и очищ.

4) **response** – *не содержит атрибуты*! живет столько же как и req, атрибуты не нужны. Когда приходит req, существует конвеейр его обрабоки – фильтр. Если надо данные передавать между фильтрами, лучше с пом атрибутов req.

1. Java EE: дескриптор развертывания web-приложения. Параметры инициализации web-приложения: назначение, принципы применения.

= 16 ВОПРОС

1. Java EE: фильтры (Java Filter), назначение и принципы применения.

**Объект ФИЛЬТР:**  
\* похож на middleware  
\* это серверный компонент, предн для предварит обработки запроса, для организации контейнера обработки запроса, «препроцессор запроса»  
\* обычный класс, реализ интерфейс Filter (тот оч похож на сервлет: метод init, destry, doFilter)  
\* фильтры созд при загрузке сервера. Они записаны *в web.xml* => вся инфа о фильтре хранится в контексте прил. Там же *мэппинг* фильтров с ресурсом.  
- init() – при иниц фильтра, чтобы фильтр мог восст нек данные, кот он где-то сохранил (в БД, файле)  
- destroy() – чтобы записать д-е, чтобы потом использ при восст  
init, destry – способы сохр состояния объектов между перезагрузками сервера  
- doFilter() – метод, вызывается контейнером сервлетов при поступлении запросов позв вызвать doFilter и передать парам: запрос, ответ, filterChain (Для образования цепочки фильтров)

**Цепочка фи** – фильтры, привяз к 1 ресурсу (ресурс: сервлет, jsp)

- фильтры послед-но выполняются

- к. фильтр завершается вызовом doFilter() след. фильтра в цепочке, обрабатывая req

- последний вызывает service() сервлета, формир. res

- стек разворачивается в другую сторону и уже обраб. res

! один класс фильтра мб в разных цепочках

1. Java EE: слушатели (Java Event Listener), назначение и принципы применения.

если есть собс фреймворки, м. предварительную обработку

**Слушатели событий** – спец механизм обработки событий на стороне С.

М создать класс, кот рализует 1/неск интерфейсов (смотря на какие события срабатывает) и записать инфу этом классе в *web.xml*. С зарегает эти объекты в контексте. Тогда C при старте сразу поднимет фильтры и листенеры.

События (контекст и сессия):

- созд/разруш контекста, сесии

- созд/измен атрибутов контекста, сесии

1. Java EE: применение заголовков запросов и ответов.

есть заголовки стандартные http (классиф)

есть заголовки самих разработчиков

как можно передавать запросы, обраб, применять

Все заголовки на **4 осн. группы**:

- general - исп. в запросах и ответах (cache-control, connection, date, warning, pragma)

- request - в запросах (accept, authorization, from (эл. адрес), host (сет. адрес сервера))

- response - в ответах (location (адрес перенаправления), server (инфа о по сервера), proxy-authenticate (сообщ о статусе 407))

- entity - сопровожд. каждую сущность сообщения. в запросах и ответах (allow (список методов), content-type, -length, -language, -encoding, expires, last-modified)

Запрос и ответ также можно дополнить собс. заголовками, кот. потом мб получены и обработаны на клиенте. Заголовку с одним именем может соотв. неск. значений.

*addHeader(‘jjj’, ‘abc’)* – если заголовок jjj уже есть, то просто добавляется значение ‘abc’, иначе создается новый.

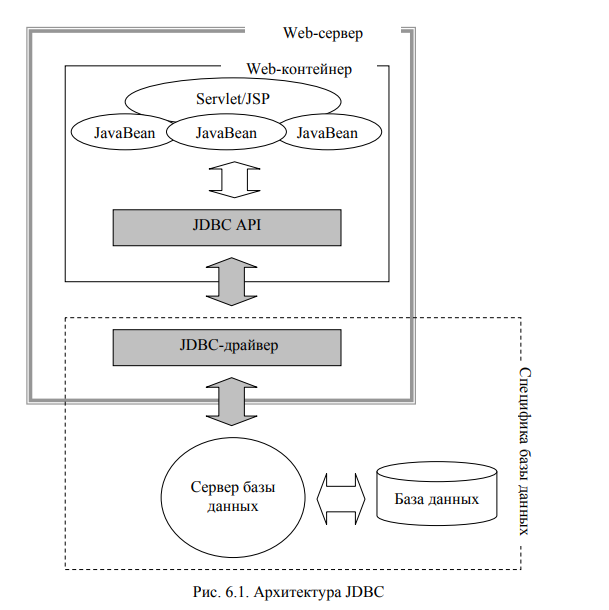
*setHeader* – тоже создает новый заголовок или заменяет значение

Доступ к заголовкам ответа осущ. так же, как к заголовком запроса: с пом. getHeaderNames, getHeader.

1. Java EE: JDBC, структура, принципы использования

что, из чего сост, архитеута, классы, осн. методы

**JDBC** – java db connectivity – прогр ин-с, обесп взаимод нашего прогр кода на Java с реляц БД. ODBC – самая быстрая, прародитель всхе таких технологий. Если прогр код на С++, его используем (либо OleDB).



Больш-во JDBC-драйверов предоставляют авто- **connection pool** – спец. механизм для экономии подключений. Позволяет создавать очередь подключений, обесп. более гибкую работу с БД.

Надо разделить ин-с на 2 компонента:

\* *JDBC API* – стандартная, общая для всего ПО, кот использ ин-с (.NET, Java пофиг)

\* *JDBC-драйвер* – учит особености каждой СУБД (Драйвер бд – составная часть какой-то субд чаще, ее разрабатывают разработчики субд). В ADO.NET это называется ***провайдер***, но фактически это драйвер.

JDBC API – набор программных ин-сов (поименованный набор сигнатур). А драйвер – классы, кот реализуют этот объект

**Типы драйверов:**

1. Первый тип jdbc-драйверов является мостом между JDBC API и драйвером ODBC. Все вызовы JDBC-драйвера первого типа транслируются в вызовы odbc-драйвера, с помощью которого и выполняется доступ к базе данных.
2. Второй тип драйверов только частично реализован на языке Java и является зависимым от платформы.
3. Третий тип драйверов полностью реализован на языке Java, но взаимодействует с сервером через клиентскую часть СУБД.
4. Четвертый тип драйверов тоже полностью реализован на языке Java и не требует никакого посредника для связи с сервером базы данных (тонкий клиент).

JDBC – jar-файл, ODBC – это dll-файл. Главный недостаток odbc – он не кроссплатформенный, в отличии от jdbc, кот написан на джаву, и работает на всех платформах, где работает jvm. Можно смело переностиь на уровне байт-кода.

**Connection** – Объект, предост нам драйвер, но этот объект реализует ин-с Connection, кот есть в jsbc. **Драйвер** – реализация этих ин-сов. ПОграничным явл объекь Connection, кот мы извлекаем из этого драйвера и начинаем с ним работать.

**Чтобы соединиться с сервером СУБД:**  
\* ip, где сервер или символический адрес (dns, netBIOS, hosts, gethostbyname помогут)  
\* порты 1521 oracle, 1433 mssql  
\* user name, password  
\* имя инстанса, к к-му мы коннектимся  
\* тип драйвера: тонкий или толстый (для oracle)

Все эти параметры в объекте Connection

Дальше м делать статические, динамические запросы и вызывать удал.процедуры, кот есть на стороне Сервера  
В оракл процедура м возвр только 1 строку, м еще здесь вызывать ф-ю в оракл

**update** – м формир рез набор (появился недавно) с пом секции output  
preparedstatement – то же самое executequery – в рез объекта класса resultset, кот мы м опреационным способом перебирать (это коллекция строк) с пом цикла, и обрабаывать их. Получаем как будто курсор для работы с рез набором

1. Java EE: принципы разработки фреймворка.

удобно использ листенеры и фильтры

интерфейсы!!! (wtf)

1. Java EE: HTTP-аутентификация, JAAS,назначение, применение.

использ бэсик или формс

томкат users. - роли и юзеры (файл)

***вопрос 5 !!***

***JAAS (Java Authentication and Authorization Service)*** – Java-сервис по аутентификации и авторизации. Входит в томкат. Предполагает 2 формы аутент: *BASIC*, *FORM*.

- *basic* - стандарт. форма ввода данных для аутен (логин пароль)

- *form* - собс форма (по опред. правилам jsp-страницы) для аутен

Нельзя использ. 2 вида аутент одновременно, потому что джава скажет, что web.xml невалидный

Защищенный ресурс (html –страница, jsp-страница, сервлет) привязывается к ролям. Все пользователи, содержащие в своем списке роль, обеспечивающую доступ (привязанную роль) к некоторому ресурсу, имеют возможность доступа к данному ресурсу.

**Применение JAAS в томкат:**

- изменить *tomcat-users.xml* - для хранения инфы о ролях, именах, паролях юезров в сервере Apache Tomcat Каждый пользователь характеризуется: именем, паролем и списком ролей, приписываемых ему.

- внести инфу в *web.xml*: указ. защищенный ресурс, адрес страницы, защищенные методы обращения к ней, название роли и метод аутент. В forms также надо указать адрес страницы логина и страницы с ошибкой.

- разраб. форму для ввода + сообщ об ошибке (для form), там использ. заранее-определенные имена полей: action = j\_security\_check, j\_username, j\_password

1. Java EE: принципы разработки download/upload-приложения.

низкоуровневые принципы

какие заголовки

в какой части запроса файл

Чтобы сохранить файл с клиента, надо разраб. сервлет, отвеч. на запрос с установленными заголовками:

- *Content-Type*: опред. тип файла, кот. пересылается клиенту

- *Content-Length*: кол-во байт пересылаемых в потоке

- *Content-Disposition*: указ. мета-инфу файла (напр, имя файла)

**Пример:**

File file = new File(**path + "\\" + fileName**);

response.**setContentType**("application/msword");

response.addHeader("**Content-Disposition**", "attachment; filename=" + file.getName());

response.**setContentLength**((int) file.length());

BufferedInputStream buf = new BufferedInputStream(new FileInputStream(file));

int readBytes;

while ((**readBytes = buf.read()**) != -1) {

**response.getOutputStream().write(readBytes);**

}

*Объяснения:*

тут создается экземпляр File с указ. физ. пути на сервере. Устан 3 заголовка. Дальше потоковое чтение из файла и потоковая запись в http-ответ.

**UPLOAD:**

создается форма с указанием атрибута enctype=”multipart/form-data”, в кот. есть input с типом type=”file”. На сервер, на сервлет нам приходит запрос с огромным телом, внутри к-го инфа о файле, разделитель и потом само содержимое файла. Если файлов несколько, они разделяются разделителем. Если не используем библиотеку для парсинга, то чтобы сохранить файл на сервере, надо распарсить тело, извлечь из него название файла с расширением, содержание фалй, и потом записать это всё в новый файл. Но есть уже куча готовых библиотек.

1. Java Jetty Framework: архитектура, назначение, применение.

2 слова: embated системы, что это что умеет делать

тоже контейнер сервлетов - упрощенный tomcat

**Jetty** -веб сервер, такой же контейнер сервлетов как tomcat; но полностью написан на Java; Он ищет папку webapps и регистрирует все приложения оттуда. Также есть другие папки.

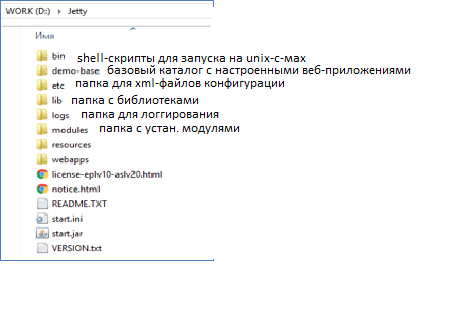
**Технологии из JEE, кот. поддерживают Jetty:**

Сервлеты, JSP, JSTL, WebSocket, JAAS, JMX, JNDI JavaServer Pages Standard Tag Library, JSTL) — расширение спецификации JSP, добавляющее библиотеку JSP тегов для общих нужд

JSP (JavaServer Pages) — технология, позволяющая веб-разработчикам создавать содержимое, кот. имеет как статич, так и динам компоненты

**Контейнер сервлетов** - программа, предст. собой сервер, кот. занимается системной поддержкой сервлетов и обесп их ЖЦ в соотв. с правилами спецификаций. М работать как самост. веб-сервер.

Обесп обмен д-ми между сервлетом и клиентами, создает прогр. среду для функц-щего сервлета, идент и автор клиентов, организ. сессии.



1. OSCi: назначение, архитектура, реализация.

*OSGI* - технологии, кот. позв. отключить и заменить отд. модули?   
набор для разработки + набор утилит для изменения этих модулей, подменять и тд

Часто нужна модульность приложений: разраб. прилож. из отд. модулей. Каждый модуль д. реализовать опред. функционал и мб использован в других прил. Модули дб слабозависимы, а завимодействовать с ними лишь через простые ин-сы.

М. отдельно обновлять модули = улучшать ПО не всё целиком, а отд. части.

Также у прилож. дб непрерывная работа, особенно серверные. Но иногда надо остановить прил для обновления или замены модуля. Часто такое прерывание недопустимо.

OSGI справляется с этой задачей.

*OSGi (Open Services Gateway Initiative)* - спецификация динам. модульной с-мы для Java-приложений, разрабатываемую консорциумом OSGi Alliance. Она определяет модель построения приложения из компонентов, кот. динамически может изменяться во время выполнения прил (runtime). Взаимод между компонентами осущ. с пом. сервисов, кот. зареганы в регистре сервисов (Service Register)

Изначально спецификация разрабатывалась для встроенных систем (BMW, Siemens), затем распространилась на многофункциональные приложения

OSGi оперирует *модулями (бандлами).* Каждый bundle - логически завершенный прогр. компонент (обычный jar-архив), реализ. опред. функционал. Возможности:

- подключ модулей в с-му: подключ bundle м во время работы прил

- разреш. зависимостей с учетом версий: у к. модуля номер версии == другие модули м. заявлять зависимость от конкр. версии модуля

- управл юзерами (user admin service): ин-сы для аутент и автор юзеров в приоржении

- протоколирование событий (log service)

- механизм конфигурирования модулей (preferences service)

- сервис свойств модулей (preferencies service)

- сервис http: м обраб web-запросы, т.е. можно зарегать модуль в кач-ве обработчика http-запросов

Согласно спецификации OSGi для bundle, реализующего интерфейс BundleActivator, определен следующий жизненный цикл (Lifecycle) :

INSTALLED успешно установлен;

RESOLVED bundle готов к старту и ему доступны все Java-классы и бандлы, от которых он зависит;

STARTING bundle стартует, т.е. выполняется метод BundleActivator.start();

ACTIVE bundle стартовал, т.е метод BundleActivator.start() завершился;

STOPPING bundle останавливается, т.е. выполняется метод BundleActivator.stop();

UNINSTALLED bundle не установлен (удален), т.е. жизненный цикл бандла завершен.

1. Maven: архитектура, репозитории, артефакт, POM, Parent POM.

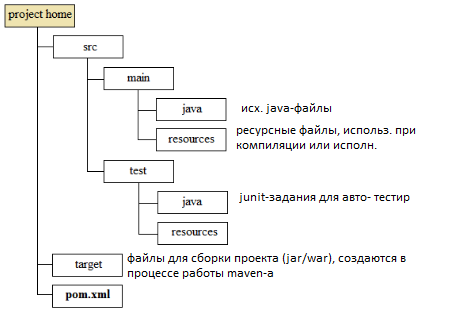
неуверено с ним себя чувствую

непредсказуем

**Maven** - для автоматизации сборки проектов на основе POM-файла (project object model - подмн-во xml). Выполн компиляцию кодов, созд. дистрибутивы программы, jar/war файлы и генерирует документацию.

В отличии от ANT (императивная сборка), в файле описания проекта не отд. команды, а спецификация проекта (декларативн. сборка)

**Архитектура:**



**Репозиторий:** это место, где хран. jar, pom, javadoc, исход. файлы

- внешний центр. репозиторий Maven, в кот. собраны популярные библиотеки

- удаленный: указывается разработчиком

- локальный, в кот. хранятся копии использ-мых ранее библиотек

доп. репозитории для сборки проекта перечисл. в <repositories> в pom.xml

**Артефакт:** это скомпилированный jar/war-файл, кот. развертывается в репозитории maven, и содержит в себе gav.

**Зависимость:** это связь, кот. говорит что для нек. фаз ЖЦ maven-проекта нужны нек. артефакты. Они описаны в <dependencies> в pom.xml. Для каждого артефакта надо указать GAV:

\* *groupId* - id производителя объекта (имя с обратным доменом: com.example.foo)

\* *artifactId* - обычно имя создаваемого модуля

\* *version* - версия объекта

**POM.XML:**

там описана структура проекта. Этот файл дб в корневой папке. Там секции:

- gav-параметры описания проекта (обяз.)

- свойства properties

- репозитории repositories

- зависимости dependencies

- сборка build: имя файла сборки, source и ouput директории, плагины и т.д.

**ParentPom** - файл, от кот. наследуются все pom-файлы. Там указаны значения по умолчанию

**ЖЦ мавена:**

- clean - удаление директории сборки

- validate - подтвержд. корректности проекта и доступности инфы для сборки

- compile - компилирует исх. код

- test - запуск тестов

- package - преобраз. скомпилир. код в пакет (jar/war)

- verify- проверка пригодности пакета

- install - установка пакета в лок. реп, чтобы его мб использ как зависимость в др. проектах

- deploy - копирование пакета в удал. реп

1. Система обмена сообщениями Kafka: назначение, принципы применения, reduce/map

Был интерфейс JMS. Писали лабу обмена сообщениями.  
JMS - просто интерфейс, за ним скрываются реальные с-мы обмена сообщениями.

С-мы обмена сообщениями (**СОС**) относятся к т.н. mom-системам. Интерес к ним появился с появлением «Больших данных».   
**Большие данные** – группа технологий и матем. методов, направленных на обработку неограниченного объема д-х за огранич. время.  
В основе больших данных лежат процедуры:  
\* *Map* – разделение данных по к-то принципу на неск. потоков для паралл.обработки;  
\* *Reduce* – данные обработали, преобразовали и свёртываем обратно в 1 общий поток.

Примеры таких с-м: Hadoop, Apache. В их основе лежит *надежная очередь сообщений* – пока потребитель данных из очереди их не прочитает, данные не продадут. Прочитал – сделал коммит «я прочел» == данные из очереди будут пропадать.

**СОС:**  
\* *p2p* – между 2 приложениями канал в виде этой очереди. От одной точки до другой идут сообщения. Всегда есть 1 продюсер, 1 очередь, 1 consumer + прочитанное сообщение удаляется из очереди.  
\* *pub-sub* – producer (производит сообщение) и consumer (потребитель). 1 producer, несколько очередей (тема), неск. consumer-ов, кот. подписываются на очередь. Независимо друг от друга считывают сообщения из этих очередей. Данные вообще не удаляются из очереди, т.к. другим consumer-ам тоже могут понадобиться эти данные.

**MOM: Message Orriented Middleware** – middleware, ориент. на обмен сообщений: RabbitMQ, Microsoft MQ, Apache Kafka.  
Эта с-ма может иметь централиз. и распределенный вид. Также она может конфигурироваться в завис от потребностей.

**Apache Kafka (AK):** **программный брокер сообщений** – распределенная гориз-масштабируемая с-ма: за счётболее детального распределения софта можно увел. кол-во серверов, потоков и т.д.   
- применяется чаще всего в приложениях с *микросервисной архитектурой*.  
- для обработки неогранич. массивов данных  
- осущ. централизованную координацию сообщений  
- надежная очередь  
- модель pub-sub  
- потоковая обработку  
- кластерная архитектура: м увел кол-во компонентов в кластере. Есть много компонентов, а для конечного юзера представл. как одно целое (кластер)  
- обесп. репликация в кластере, для надежности  
- транзакции   
- протоколирование работы  
- есть механизм обеспечения надежности (за счет репликации)

В 2016 кафку скупил MS за 26 млрд $.

**AK**: диспетчер сообщений на Java-платформе. понятия: *издатели* (пищут сообщ), *тема* (в нее издатели пишут сообщ, = имя очереди), *подписчики* (подпис на тему, читают сообщ). АК хранит записи заданное время.  
- *тема* – именованный поток сообщений  
- *producer* – издатель сообщений  
- *consumer* – подписчики  
- *partition* – раздел темы: логич. точка подключения consumer, для распараллеливания обработки сообщений темы. Подписчики объед в группы, т.е. они могут обрабатывать общий поток. Эти группы считаются одним целым.

- *offset* – все сообщения в теме нумеруются. С одной темой могут работать столько подписчиков, сколько есть разделов. Если подписчиков будет меньше, то одному подписчику будет приписано несколько разделов сразу. Идеал: кол-во разделов = кол-во подписчиков. Если подписчиков больше, лишние будут ждать. В рамках раздела идет индексация сообщений. Когда читаем сообщение как consumer, мы получаем его № (offset?). Для каждой группы свой offset. Несколько подписчиков в группе == 1 offset.

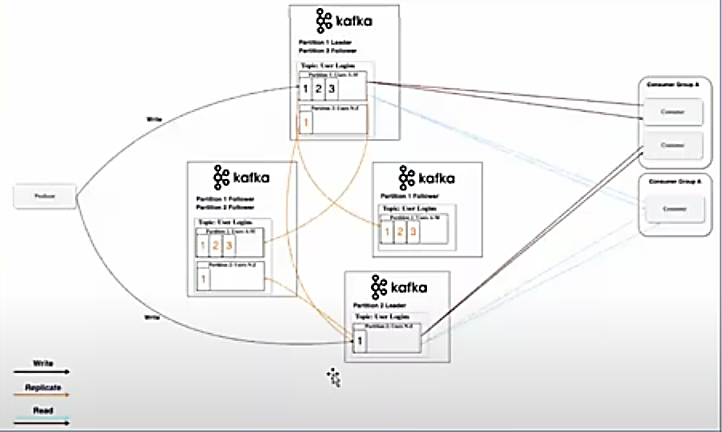
- *commit* – фиксация чтения: когда подписчик прочитал сообщение, он говорит об этом, кафка сдвигает offset раздела и будет в очереди выбираться след. элемент.

- *streams* – до сих пор мы говорили об одиночных сообщениях (проходят сквозняком от продюсера к консьюмеру) – механизм обработки потока сообщений, включает доп. де-вия, кот происходят в рамках кафки: *фильтрация*, *преобразование*, *агрегация*.

С понятием streams тесно связана «таблица», кот. к нему всегда привязана. Streams на лету может что-то вычислять, например агрегации, что-то считать: суммы, среднее. Например, если мы собираем данные о местоположении наших юзеров. В вашей с-ме обработки сообщений с телефона поступает сообщение о их месторасположении. Мы будет собирать инфу, сколько каждый наш клиент находился в Минске, Москве. Для этого сущ-ет спец. таблица, кот. связана с этим потоком: там id юзера; город; число. Вся эта инфа агрегируется в нашем streams. Сама таблица организована в виде очереди.

**ИТОГО Кафка**: обрабатывает сообщения, распределяет их и организует стримы: которые на лету выполняют предварительную обработку этих данных.

**Системная организация:**  
Сама кафка представляется в виде т.н. *брокеров* – узел (с-ма будет состоять из неск. узлов), кот. может предоставить опубликованные данные.



На схеме эти узлы. Справа консьюмеры, слева один продюсер, кот. постоянно пишет данные. У него есть узлы, кот. делятся на 2 типа: лидеры и фоловеры. Фоловеры просто для репликации данные чтоб их сохранить. Самый верхний узел (брокер) - он лидер для партишиал 1 и фоловер для партишиала 2. А внизу еще один лидер для партиции 2. Есть два последних - фоловеры для всех партиций, чтоб сохранять данные в с-ме. Для надежности. Всё что есть в с-ме делится на такие брокеры (узел с-мы), их можно создавать, растаскивать по разным ПК для гориз. масштабирования == увел произв-сть с-мы.   
На сайте можем найти Producer API (для описания продюссеров), Kafka Concumer API для написания консьюмеров, для стрмов и для администрирования. Для нас как прогеров, представляется Kafka Server, Kafka-утилиты, ZooKeeper-сервер (координатор, хранит служеб. инфу, чтобы обеспечить скомбинир. работу консьюмеров и продюсеров). Они между собой дружат.

**КАК УСТАНОВИТЬ И РАБОТАТЬ:**

*Kafka* - это сервер, кот. на самом деле состоит из 2 серверов: сервер кафки и сервер zookeeper (дружит с кафкой и предн. для координации работы кафки. хранит служ инфу, кот использует в процессе обработки сообщений).

1) Сначала надо установить ZooKeeper. Обычное java-приложение. Скачать, развернуть файл. В папке conf переименовать конф. файл в zoo.config и указать там, где zookeper будет хранить свои снапшоты данных (ФС путь).  
2) Добавить zookeper\_home, + в path ..\bin  
3) Запустить zookeper-сервер - проверка.   
4) Установить Kafka: она заточена на линукс. Кафка не любит отд. директориев (не русское и без пробелов). В config перестроить 2 файла: *serverproperties* - указ. директории для логов; *zookeper-properties* - указ. куда zookerper будет писать (как в п.1)  
5) Запустить kafka. Там есть какие-то батники, кот. сами запускают сначала zookeper, в кот. я указываю properties кафки, а во втором указ. properties сервер кафки.  
== у нас готовая с-ма обработки сообщений

Набор **утилит** для проверки работоспособности с-мы:  
\* kafka-topics --create --topic (к. консьюмер читает с какого-то топика)  
\* kafka-topics --list (выведет все топики)  
\* kafka-topics --describe -topic (задать детальную инфу о топике: кто у него лидер)  
\* kafka-console-producer -topic smw (имитируем продюсера для проверки)  
\* kafka-console-consumer -topic smw (имитируем консьюмеры в другой консоли)

Spring дружит с кафкой.

ОБЯЗАТЕЛЬНО ВОПРОС ПРО WEBSOCKET

как переключать, как его использовать и тд

LONG POOL повторить!!

сети, протоколы, веб-прил, ISO OSI

**WEB SERVER COMMINUCATION**

Это новая возможность, кот. появилась в HTTP2. Можем предварительно загружать на браузер (=любой http-клиент) некоторые ответы заранее, кот. будут сохраняться в кеше браузера.

**Пример:** есть index.html, на кот. link к css. В этом случаем делаем запрос index.html на Сервер, тот делает нам response с index.html. Дальше браузер делает новый get-запрос, чтобы получить styles.css.   
Http2 делает это в рамках 1 соединения (мультиплексирование). Также можно заранее подкачать при первом запросе этот css на сторону браузера. Т.е. есть запрос на index.html, а мы можем сразу тут же подкачать туда css: **ServerPush** - можем выполнить вторую отправку, кот. предварительно подкачивает необх. файлы, кот. пригодятся для первоначального файла. Мы должны заранее знать, что надо предварительно подкачать. И браузер их сохранит в кеше. И дальше request на css не нужен (есть в кеш).

**Код:**Есть сервлет, в нем doGet : появляется класс **newPushBuilder**. Этот builder строю от request-а (3 строка). У этого builder-а есть метод .*Push()* - отправляю картинки (11 строка). Push подкачивает на сторону браузера эти картинки.

**WEBSOCKETS**

**WebSocket** – \*новый протокол

\*формат передачи д-х

\*надстройка над TCP для обмена сообщ в режиме реал.времени

(подробнее в устан.соед.)

\*стандарт RFC 6455

\*дуплексный (2 канала связи)

\*пакет require(‘ws’)

\*широковещ.сообщ (всем узлам в сети)

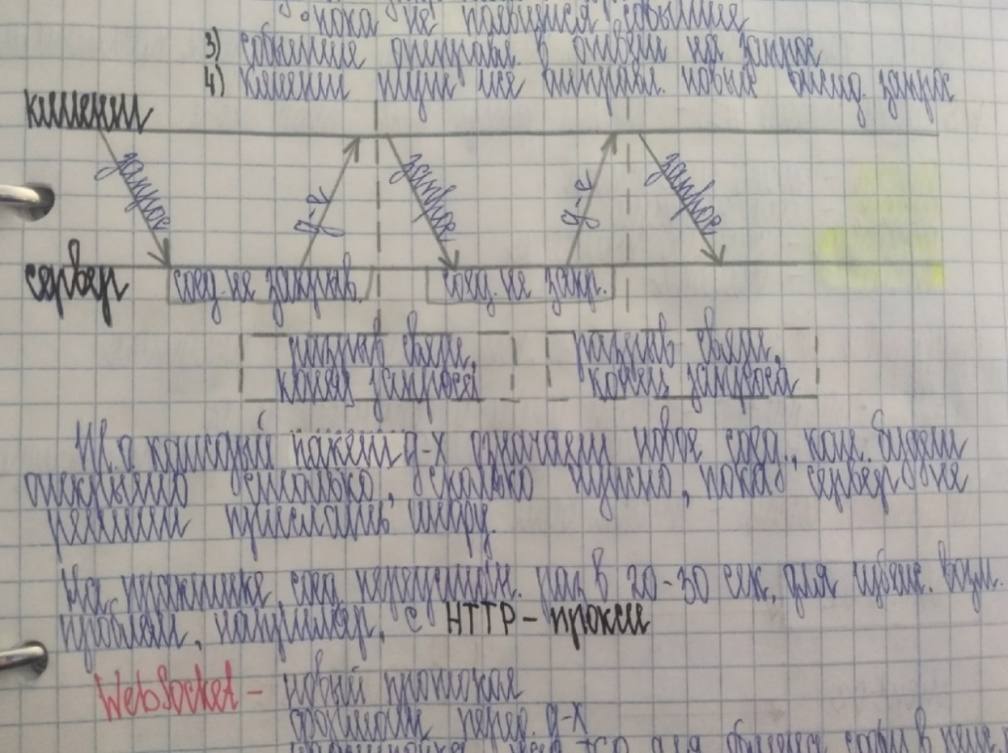
\*каналы – *потоки*

\*механизм ping/pong

\*JSON, XML и др.форматы передачи сообщ.

\*RPC

**Схема:**



каждый пакет д-х означает новое соед., кот.будет открыто столько, сколько нужно, пока С не решит прислать инфу.

**Процесс установки соед:**

*Upgrade*– К просит С переключится на новый протокол (WebSocket)

*Процесс «рукопожатия»* - процесс установки соед по новому протоколу:

К посыл обычный HTTP-запрос на С с заголовком **upgrade** кот сообщ.С, что К хочет устан.WS-соед.

**лонг пул** – **это** технология, которая позволяет получать данные о новых событиях с помощью «длинных запросов». Сервер получает запрос, но отправляет ответ на него не сразу, а лишь тогда, когда произойдет какое-либо событие (например, придёт новое сообщение), либо истечет заданное время ожидания. Если за это время поступило неск. запросов, то отдает в одном ответе.

*Пул соединений:*

Есть min и max кол-во соед. При старте соп. пула, он авто- открывает min соед-й. Когда они все занят, пул откр еще одно… до max. Никогда не закр соед!

*Пул соед* – набор соед. с БД, кот. по мере нагрузки увел. с min->max + очередь

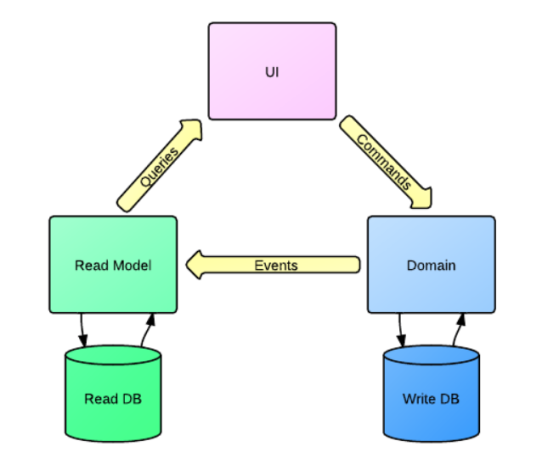
**CQRS (command query separation)** - принцип императивного программирования (Б. Мейер), кот. гласит, что *метод* должен быть либо *командой*, выполняющей действие, либо *запросом*, возвращающим данные, но не одновременно. Т.е., это стиль архитектуры, где операции чтения отделены от операций записи.

+ т.к. запрос не меняет состояние объекта, то методы типа Запрос можно распараллелить

+ команда чтения не вызывает побочного эффекта (не измен сост объекта)

+ обычно 2 бд: для записи и для чтения. Базы данных синхронизируются

Перенос осн. принципов CQS на архитектуру (М. Фаулер)



\* контрактное программирование

\* спецификация методов (ин-сы)

\* предусловие

\* постусловие

\* инвариант (обяз. соглашения в методе)

\* исключения

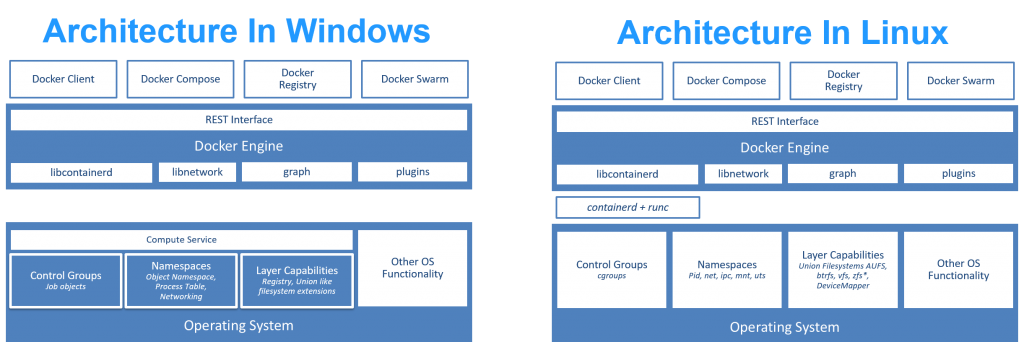
*CORS* – command-query responsibility segregation – разделение ответственности – эти подходы применяются для разработки сервисов и микросервисов

**Docker** - ПО, кот. позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, кот. мб перенесен на любую Linux-систему, а также предоставляет среду по управлению контейнерами.

**Docker** - Это набор программ, кот сост из **2 компонент**:

*- docker hub* - облачный сервис, репозиторий, там м укладывать docker-image  
*- docker desktop (engine)* - для формир. контейнеров на нашем ПК. его структ зависит от ОС, на к-ю он устан. Изначально все было ориент на Линукс. В конце концов Микрософт подхватил инициативу и тоже поддерж эту платформу (с 10 версии).

Внутри ядра линукса появилась возможность создавать своеобразные Namespaces, в рамках кот м запускаться наше прил и для нашего прил м создать Namespace, с кот связываются все объекты СС, необх упр-нием приложения в рамках ОС, и эти компоненты сущ в отд пр-ве имен. Разработав неск приложений, к. раб своем namespac-е (это условное название) == есть неск приложений, изолир др от друга. Вся эта изоляция осущ ровно до польз. ин-са. Если потом создать файл, в кот будет располагаться эта среда, его м переносить.



2. Java Spring Boot Framework: архитектура, назначение, применение  
2. Архитектура CQRS.

2. Фреймворк и Spring Boot, Jetty, Undertow: назначение, применение.

Платформа Docker: назначение, применение.