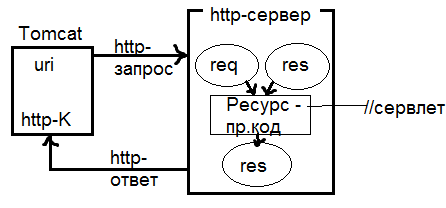
**ЛК 1. Программирование в Internet**



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

**Tomcat** – контейнер сервлетов, это С, обесп работу спецификаций JEE  
JEE – документация, сервер внутри webapps  
Tomcat вып в кач-ве службы Windows

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

\* web.xml – (в папке code) – опис. конфиг по умолч, sessionTimeout, web.xml есть (мб) в каждом приложении. Если его нет, смотрим общий, т.е. в папке code томката (по умолч).  
welcome-file-list – список стартовых страниц в порядке их перечисления – м. размещ в web.xml прил-я (приоритетнее)

\* tomcat-users.xml – юзеры, группы, роли. Поддерж basic, forms-аутент. Ресурсы Са мб защищены ролью (ресурс – uri, например)

□ **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

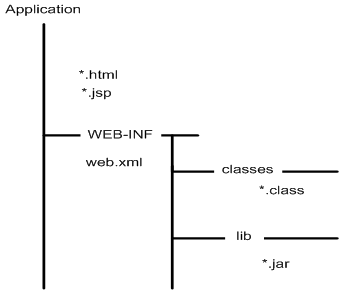
□ **webapps** – то что мы будем разраб

**Простейшее приложение:**  
webapps – SimpleApp – index.html  
localhost:48080/SimpleApp/  
будет наш html

**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

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

**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)

**Способы взаимодействия Сервлетом др. с др. и др. компонентами:**

|  |  |
| --- | --- |
| 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 |

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

**JAVA SERVER PAGES**

**JSP** – и есть сервлет в др. виде, определение то же  
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)  
- include (вкл др. страницу)  
- agrig (для библ тегов)

**Формы записи динам. контекста:**  
- объявления  
- скриплеты  
- выражения

<%!...%> - на уровне класса, потом исп для др. динам эл-тов  
**Скриплет** – место, кот. превращ в метод класса сервлета (мб service)  
req, res нигде не опред, это авто- объекы в JSP, вверху есть Import  
**Выражение** – элем JSP, позв вычислить что-то и подставить значение

**ЛЕКЦИЯ 2. СЕРВЛЕТЫ**

**3 базовые конструкции:**  
1) есть 1 сервлет и он встречает запросы, работает с БД и формир ответ  
2) примен и сервлет и 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. Если много динамики – лучше Ss

в JSP доступен ряд неявных объектов: ServletResponce, service, pageContext, out, page…

Есть **библиотека тегов JSP** – спец механизм, с пом к-го м разрабатывать собс теги, кот можем применять в JSP-странице. Цели: разделить работу верстки и разработки  
\* создать дескриптор библиотеки  
\* создать xml файл  
\* написать обработки библиотеки тегов (тег-хэндлеры), кот выполняют подмену тегов библиотеки на необх разметку  
\* использ директиву taglib  
\* Добавить новые разработанные теги на странице

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.

….если версия веб сферы равна версии galssfish, и совместимость JVM. В рамках этого контейнера работает наше прил, т.е. обращаемся к функциям из контейнера. Конт предлагает ряд **системных объектов**, кот мы м исп в прцоессе работы. В томкат всего 4 типа таких объектов:

1) **контекст прил** – объект, кот создается при старте сервера (С стартует, ищет web.xml и формир контекст), сущ пока работает С, явл общим объектом для всего приложения. Самый долго играющий объект. В рамказ этого контекста сущ коллекция, кот создает память, в к-. м запис пары имя:значение (атриб) и извлек, м сделать setAttribute, removeAttribute. В кач значенпия часто использ ссылку на обхект. М создать объект к-то класса а ссылку его записать в контекст. Ест живая ссылка на объект и она будет хранится и garbage-мусорщик его не уберет. Те. контекст предоставляет нек память, кот хранится, пока работает С, если сами не удалим. Это первый уровень памяти  
В Web.xml также м поместить параметр инициализации

2) **session** – системный объект, созд для серии запросов, t м/которами не превыш sessionTimeout. Эта серия создается с пом куки, там session id. Каждый запрос помечается к какой сессии он относ. Если С не может определить сессию к к-й относ запрос, он создает новую сессию. Т.о. любой запрос пришедший на С, помечается к какой сесссии он относ. У сессии есть id (128 бит число) + память + коллекция (атрибуты имя:знач анал.) и эта коллекция сохр в рамках жизни сесии. СЕссия живет, пока жива серия запрсов. И для этой серии запросов м созранять нек данные (сохр состояние). Сессия – механизм сохранения состояния. Но когда сессия удалена, ссылка на объект пропадет и garbage cleaner его сожрет. Это второй уровень памяти (контекст для всех сессий общий => межсессионный обмен данных. Если хотим обмен между запросами => исп сессию). Крайние случаи – оч редкие запросы, для каждого своя сесиия, или очень частые запросы – одна сессия

3) **request** – живет в рамках запроса, тоже содрежит атрибуты и память. Это третий уровень памяти. Если хотим сохранить данные в рамках обработки 1 запрсоа, лучше в рамках объекта рекуест. В виде setAttribute запис данные, когда зарпос 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.

**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.   
- init() – при иниц фильтра, чтобы фильтр мог восст нек данные, кот он где-то сохранил (в БД, в файле…)  
- destroy() – чтобы мог данные записать, 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) и они будут посл-но выполняться. Обычно фильтр завершается вызовом doFilter след. фильтра в цепочке. В конце цепочки последний фильтр вызывает service() cервлета, к-му передает req, res.

**Фильтр** – серверные компоенены для предварит обработки запросов прежде чем они попадут на конкр ресурс, созд при старсте сервреа

Они описаны в *web.xml*, там же мэппинг фильтров. Паттерн связывает фильтр с конкр ресурсом. Один и тот же класс фильтра мб исп в разных цепочках. Фактически это тупо middleware.   
Один фильтр вызывает другой, обрабатывая req, потом доходит до ресурса. Отрабатывает jsp/…, формир res и этот стек разворач в другую сторону. doFilter() - по стеку в глубь иду, на ресурсе делает петлю. На обратном пути можем обрабаывать уже res.

**Слушатели событий** – спец механизм обработки событий на стороне С. М создать классы, кот рализуют этот ин-с и записать инфу об этих классах в web.xml. Тогда C при старте загрузит эти листенеры и мы можем обраб различ событий, связ с этими фильтрами. Есть события, связ с созданием/разруш контекста, измен атрибутов… Те же события можем подловить и обработать у сессии (создание, измен ее атрибутов). Чтобы создать обработчик таких событий, м создать класс , кот реализует 1/неск интерфейсов (смотря на какие события срабатывает). Чтобы С эти объекты мог поднять и зарегистрировать в контексте. Для этого пропишу их в web.xml C сразу поднимает фильтры и листенеры.

jstl – готовая реализация нек компонент.

схема клетка как-то

М поместить собс данные в контекст. М записать параметры контекста кот потом м извлекать в программе с пом InitParam => обесп работу фреймворка

**JDBC** – java db community – 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: одна – стандартная, общая для всего ПО, кот использ ин-с (.NET, Java пофиг), 2я – 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 – 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.

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

ссхема ромб

в след раз коллоквиум -5 до 5 (0 – не влияет на оценку)

**JNDI** (java name and directory interface) – прилож работате с неск источниками д-х, к к. из них надо знать параметры соединения, мы не можем их зашивать в код, что с ними делать? М в property file (конфиг файл). Есть спец бд, позв хранить в себе такую инфу (в mssql – “Контроллер сети”, в oracle – “Active Directory”)

Эти с-мы наз **с-мами наименований**. В unix наз Directory. Протокол для таких бд – **L-DAV**.

В составе JEE входит **JNDI** – спец технология с пом кот 1) м написать ин-с для с-м наименований уже готовых, 2) JNDI дает набор ин-сов и спец. классов, с пом кот мы м разработать свою с-му наименований. В простейшем случае она выглядит как xml-файл, все к нему м обращаться и извлекать ту инфу, необх для соединений с другими источниками д-х. Там мб любая метаинфа по распределнной сети: имена юезров, сетевые пароли, коннекты. JNDI – та технология кот позволяет создавать такую службу наименования, кот м. расположить в своей сети (локальной). Обычно эти службы намиенований – Серввер и какой-то заместитель у к-го есть копия этой службы наименования.

**лекция 29.10 – Смелов не пришел**

**ЛЕКЦИЯ 05.11 (Протокол WEBDAV)**

**WEBDAV** - расширенеи протокола JDB, м. пригодиться при разработке приложений.   
Как он мб использован в программах Java, если мы работаем с облачными с-мами.

**Diskstation** - фактически (application) сервер, кот. используется как файловый сервер, кот. мы можем конфигурировать, редактировать права доступа отдельных пользователей и тд

С DropBox, OneDrive, Яндекс.Диск, Google Drive тоже можно работать через WebDAV  
В Томкат тоже есть отдельные пакеты, обесп. 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)

Он **потокобезопасен**: можем одновременно в неск. соединений взаимод. с этим файловым хранилищем  
+ можем **блокировать**, чтобы предотвратить конфликты совместной работы  
В основном, используется в с-мах документооборота

WebDAV - фактически альтернатива протоколов **FTP**, **SNB** - протоколы, использ-ся чтобы представить ФС. Apple тоже поддерживает webDAV, есть у них iDisk

**ИТОГО**: Протокол WebDAV нужен, чтобы мы могли доступиться к файловому хранилищу, кот. находится на удаленном сервере, и использовать формат сообщений HTTP.

WebDAV фактически является ***расширением протокола HTTP***, и он добавляет (помимо 8 основных методов) свои дополнительные методы, которые расширяют возможности этого протокола.  
В общем сложности, WebDAV расширен до 34 методов, в то же время он унаследовал осн. методы, кот. есть у HTTP.  
WebDAV описан в неск. **RFC-документах**: RFC 2291, 4918, 3648, 3744, 3253 (их много, т.к. протокол долго совершенствовался)

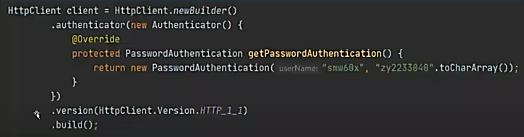
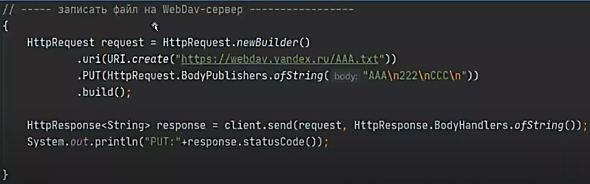
**Унаследованные методы:**  
\* *GET* - скачать файл  
\* *PUT* - загрузить файл на сервер  
\* *DELETE* - удалить серверный объект

Адрессация осущ. с помощью uri

**Версионность**: в нашем хранилище может находиться один и тот же файл разных версий, можем доступаться к этим версиям с пом webdav (как в git)

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 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 добавляем имя папки/файла  
\* аналогично: …*Http.****Get*** - скачать файл  
\* удалить: request.Method = “*DELETE*”

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


тут метод PUT (вход в API), создаю request, указываю имя файла, кот хочу создать. После аутентификации Яндекс нас выведет на свою собственную папку. Запишем AAA.txt. Дальше дела 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.

**ЛЕКЦИЯ 12.11 (маленькие вопросы, теор на экзамене)**

HTTP2 будет на экзамене!!!!!

**HTTP3** – еще в черновом варианте но уже опубликована.

1) ПОВТОРИМ HTTP 2:  
**осн новые возможности HTTP2:**  
(HTTP 3 тоже эти возможности сохранил, но уже на новом уровне)  
\* ***tcp*** устан соединения:  
+ м обесп надежную передачу данных  
+ есть обмен спец сообщениями в рамках tcp, м проверить живо ли соединение  
+ надежная передача д-х  
- надо устан соединение для к. чиф(?)  
- установка коннекта дорогостоящая, потом они еще сохраняются keepAlive  
Http2 обесп бинарную сериализацию д-х: данные отпр в *бинарном виде* (они сжаты). И осущ мультиплексирование д-х:

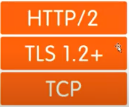


Есть Http1, какая-то страница (http, css, картинка) - и для каждого из них устанавливается соединение. Когда мы говорим о *мультиплексировании*, то появляется понятие потока (кот. введен в HTTP2) и в рамках одного соединения tcp, если всё находится по одному адресу, организовывал несколько потоков, и эти потоки работали независимо друг от друга, и фактически эмиитировали *отдельное соединение*.

+ В HTTP2 можно установить *приоритетность для потоков* в рамках мультиплексированного соединения => могли обеспечить более раннюю загрузку картинки или css.

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

Также в http2 появилась возможность **Server-push**: сервер может делать *уведомления* в рамках соединения. Т.е. сервер может посылать сообщение клиенту, чего раньше не было (было 1 req = 1 res).

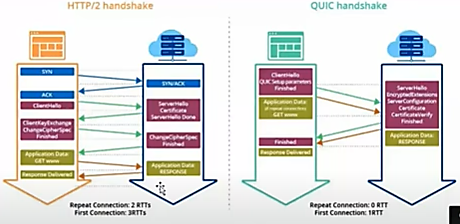
Http2 обесп шифрование с пом. протокола TLS версии 1.2+  
и использует TCP в кач-ве транспорта  
Уже на этапе разработки HTTP2 задумывались о том, чтобы заменить протокол TCP => google разработал *Speedy*-протокол. Но полноценной альтернативы не удалось.

Замена TCP назрела давно, не обесп. всех потребностей. Но это очень сложно: все ОС включено трансп. уровня TCP, UDP в кач-ве базового. На уровне железа знают о TCP (коммутаторы, шлюзы…).

2) ТЕПЕРЬ ПРО HTTP3:

Но HTTP3 все-таки уже несколько отошел от TCP.  
Другое название - **HTTP-over-QUIC** - это название произошло от протокола гугла *gQUIC* - фактически протокол, кот. пришел на смену Speedy. Но с 2018 название HTTP/3.  
Сегодня 7% сайта уже поддерживают HTTP3.

**Стек протоколов:**  
HTTP3 - это просто новый синтаксис HTTP2 (семантика та же) == они овместимы. 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:**  


Процедура установки соединения (**handshake**) в Http 2 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.

**2я особенность QUIC:**  
Он стал работать на уровне 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 HTTP 3: 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 искаение  
\* включает в себя внутьр 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. Возможно, появятся новые возможности. Незамено для нас Google уже работает с HTTP3. Если посмотреть реим отлажки хрома, будет заголовок alt-svc (указ. инфу клиенту о том, как он может переключиться на HTTP3 и какая его версия уже поддерж).

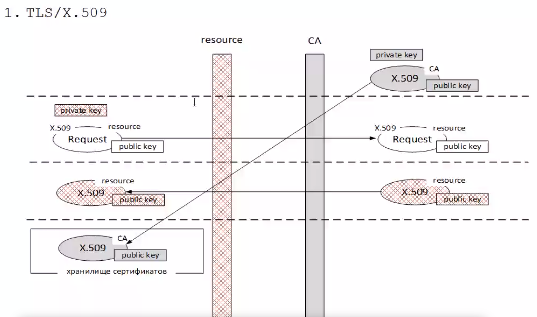
**ПОВТОР:**  
Http 3 использует в кач-ве транспорта QUIC  
QUIC использует в кач-ве транспорта UDP  
обесп. более легкие потоки над UDP, мультиплексирование, быстрое соединение  
уменьшает кол-во обменов информации  
QUIC пока не включен в ОС, работает на уровне приложения  
проблема с NAT-ом  
есть уже реализации QUIC-a  
В Http3 абсолютно все соединения зашифрованы!  
незаметно для нас мы уже работаем с Http3  
если пользуемся утилитой curl, то она уже поддерживает ключ --http3

**лекция 19.11.20 - ServerPush. Сертификаты**

**ServerPush** - эта возможность нарушает правила http (на к. запрос 1 ответ)  
м отправить на 1 запрос неск ответов, но с нек. ограничениями  
HTTP/2 протокол по соглашению всех разработчиков браузеров не работает без https  
=> остается установить https на application server, в котором собираемся работать на Java

Как установить https на томкат? расскажу

выдем connection upgrade до H2 (http/2 протокола)  
сервер отвечает 101 код – согласен на апргрейд  
Если говорим о хроме, то хром делает это всегда

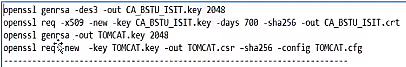


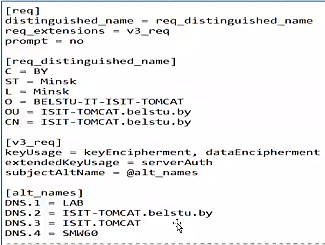
1. изгот приватный ключ
2. изгот на К запрос
3. отправляем запрос на сторону Центра сертиф
4. ЦС генерит X.509 сертификат, кот мы получаем
5. также получаем сам сертификат ЦС с откр ключом (публ ключ проверяет сертификат)

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

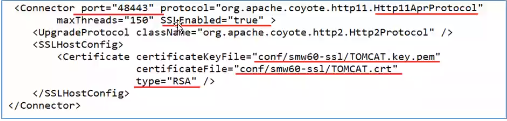
Для С дб собс сертификат + сертификат удост. центра  
Если эти 2 компонента есть, мы м настроить свой сервер чтобы он мог поддерж https-соединение.

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

OpenSSL команды:  


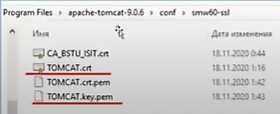


есть **ген. формат** - используется, чтобы представить ключи.  
\* залезть в файл Tomcat/conf/server.xml – там прописать соединение *Connector*  
***redirect*** – если сделаем например HTTPS на 48080, выдаст редирект (302). Выдаст на тот же адрес, но с портом 48443 (redirectPort).

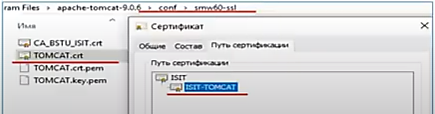


Предалагаю этот *порт 48083* использ для https.  
*Http1Appr* - внутренний java-вский протокол абстр. уровня, кот. строится над HTTPS  
*SSLEnabled* - разрешаем SSL  
*maxThreads* - макс кол-во соединений по порту  
*UpgradeProtocol* – м апргрейдить соединение до http2-протокола  
Дальше указ. сертификаты,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

**Реально происходит:**  
Мы соединяемся по протоколу Http1, после рукопожатия и т.д… Когда идет первая посылка «отправить», в этой посылке идет 2 тега: connection upgrade и upgrade до htt2. Если сервер готов, то он апргрейдится до http2 (chrome, firefox, safari точно это делают)

Затем надо реально изготовить эти файлы:  


Tomcat требует нативную библиотеку, чтоб обеспечить работу с сертификатами. Поэтому надо скачать *tcnative-1.dll* в Tomcat/bin в любом директории, кот. есть в PATH.

На сервере Tomcat надо установить сертификат в хранилище (с пом. mmc) и проверить его: (должен залиться в конфиг. файл?)  


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

Потом проверяем соединение: *isit-tomcat:48443*   
должен стоять замочек слева. (Прописали в dns). По умолчанию root-приложение запускается.

**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.

**лекция 03.12.20 - Система обработки сообщений Kafka**

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.

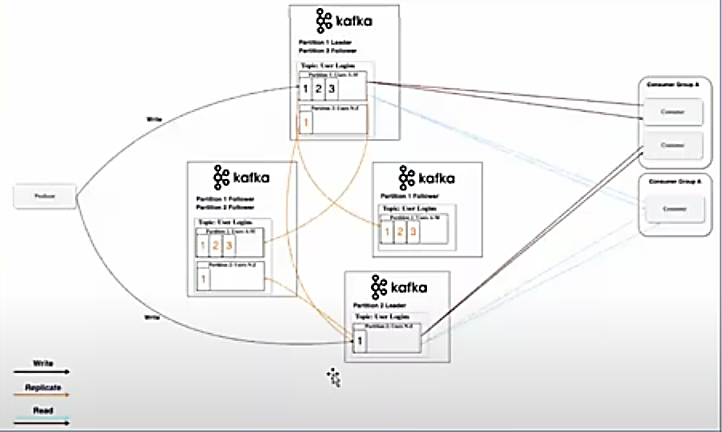
- *commit* – фиксация чтения: когда подписчик прочитал сообщение, он говорит об этом, кафка сдвигает offset раздела и будет в очереди выбираться след. элемент.

- *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.

**лекция 10.12.20 - ПОВТОРЕНИЕ (последняя лк)**

1) Сеть Интернет – в основе TCP/IP. Наиб. развитие приобрели веб-приложения, осн. на протоколе HTTP. Они предст из себя КС-приложения, где К и С взаимод. по HTTP.

Сам интернет предст. собой новую среду программирования. Есть среда - ОС. У интернета есть свои службы, как и у ОС, сервисы, есть тоже свои ср-ва коммуникации (http). При желании можно сделать сопоставления интернет-ОС. Интернет – среда, в кот. можно разрабатывать приложения.   
Java представлен несколькими стандартами: EE… Java EE - спецификация, кот м иметь разные реализации: ibm, oracle, community…   
**Application server** - сервер, кот соответствует технологиям JEE: glassfish (образцовый, все серверы тестируются на совместимоость с glassfish, сод. в себе все сразу технологии), tomcatб jboss, websphera.

SMTP позвол. отправить сообщение  
POP3 позвол. получить сообщение

*DDD*: Domain Drive Design – доменная разработка – 1 из подходов  
*CORS* – command-query responsibility segregation – разделение ответственности – эти подходы применяются для разработки сервисов и микросервисов

*EventSourcing* - подход, позвол. записывать данные в виде событий и потом эти события преобразуются в данные, кот. ложатся в статические БД. (в кафке об этом напомним)

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