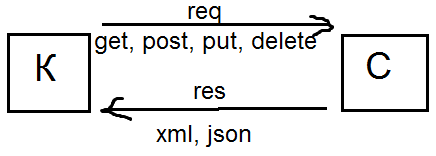
**REST API**

**REST:** Representational State Transfer – передача сост. предст-я  
предст – uri, упр. ресурсом с пом глаголов (GET, POST, PUT, DELETE)  
Альтернитива RESTу – RPC (набор удал процедур), оба стиля развив и начинают сливаться

**REST-FULL** – описание веб-службы, поддерж rest-ин-с в полном объеме. Сегодня нет офиц. стандарта REST (это минус). У RPC – SOAP (на практике он исп. и для REST)

**HTTP, URI, XML, JSON** – на эти стандарты REST опир-ся  
\* модель КС  
\* нет состояния на С (жц «запрос-ответ»)  
\* сохр. сост допуск на К или в рамках др. С (кот. обнов БД)  
\* кеширование т. на стор К и упр-е кешем с пом Са (он явно управляет сохр-ем сост на К: cookie, control-cash)   
**WEB-API** – м. закешировать, но это не сочет с треб-ями REST API

\* единообразие ин-сов, какой-то принцип д/разраб URI, чтобы юзеры не путались  
\* для К, С должен казаться конечным, мы не д. знать, что он к кому-то обращ (специф ATIL как организ IT-предпр) к др. сервисам  
\* допуск, что на сторону К мб выгружен код

   
Сервис дб предст Клиенту как храниище, где м. доб, уд…

**Общепринятые правила (HATEOAS):**  
\* общ. префикс /api/… лучше цел. домен api.belstu.by  
\* 2 типа рес: коллекц (users), эл-т коллекции (users/238)  
\* иерарх связь ../api/users/238/cards/AB1998  запрос исп. 2 коллекции  
\* ограничить кол-во http-статусов, сопроводить сообщ доп кодом (напр. 2003)  
сделать отд. ресурс (HATEOAS link) для пояснения ошибок http://ccc/api/errors/20003  
\* подавление статуса ответа ../api/students/ef3d26?status\_code=200  
\* версионность /api/students/ef3d26?v=7

\* 2 параметра (постраничное получ. д-х): limit – огранич ск. вывести, offset – смещ отн начала  
\* сортировка sort: фильтры после ‘?’ gender=m  
\* json или xml дб по умолч, 2й – по параметру, обознач в запросе формат сообщ-й  
\* обозначать в запросе формат сообщ: Accept  
\* глоб поиск: ../api/search?q=196+Иван. Берется коллекция, все эл-ты конкатен в 1 строку и выбир т. те элем коллекции, в кот. есть строка ..find. Если нах -> соотв. элем выбир

**Недостатки:**  
\* нет общепризн. стандарта RESTful API  
\* не все браузеры поддерж словарь REST-методов (PUT, DELETE). На практике исп. т. GET, POST (insert, delete, update)  
\* не однозначны коды сост

**ЛЕКЦИЯ 2. ASMX**

asmx – частный случай wcf  
мы разраб прокси клиент, кот генерировала исх файл прокси исходя из wsdl (кот мы разработали)

WSDL.exe – утилита, вызывается вижлой для генерации прокси, с ее пом можно генер сервис .Имея wsdl, м сгенеировать сервис (до этого мы генер только клиент)  
Откуда wsdl знает о логике работы?  
формально знает параметры, что возвр, и генер абстрактный класс, кот содержит абстр методы, кот формально имеют те параметры, кот описаны в wsdl, но нет реализации  
=> сгенерив абстр класс, мы можем взять его в качестве базового для нашего сервиса и просто реализовать эти методы

(на лабе будем исп ч/cmd. Есть 2 варика работать с абстр классом: 1) созд класс кот наследует => придется перетаскивать атрибуты в свой класс; 2) тупо переделать сгенерир класс в свой сервис: поубирать слово abstract методов и подсунуть реализацию = это все и для WCF, ASMX)

Как использовать службы wcf для работы с ajax?

**WCF**

это отд платформа для разработки приложений КС архитектуры на основе .NET-фреймворка  
Между клиентом и сервисом разраб хорошо продуманная архитектура взаимоотношений. Она состоит из **3 компонентов**:  
a) address – где нах компонент  
b) binding – как можно добраться  
c) contract – что эта конечная точка м делать (какие методы)

Особенность WCF – м поддерж неск точек, кот отлич транспортом. Напр, подписчик-издатель полностью по http не реализовать

Когда разраб точки, если они имеют разный транспорт, м для каждой точки разраб собс контракт того, что эта точка умеет делать

Коммуникация м/КС, взаимодействие во многом опис контрактом, на него ссылается точка доступа, он описывает что умеет делать эта точка. Если еще:  
- контракт данных  
- контракт сообщений (не затрагиваем их)

**Контракт** – соглашение м/КС о принципах взаимоотношений м/2 абонентами  
**Контракт службы** – обычный ин-с C#, кот содерж спец атрибут, и наш сервис фактически – реализация этого контракта. Если создадим класс, кот реализцет этот ин-с – это и етсь сервис

Когда создаем службу и точку доступа для нее, мы прописываем 1 контракт для нее => м разнести функциональность по разным точкам доступа с пом этих контрактов

**Контракт данных:** все типы данных – простые и композитные (класс). Если у нас сложная структурный тип данных с разными св-вами, мы должны создать *data contract*, пометить свойства как data member. На основании этого описания wsdl внесено описание этих данных

Если я все методы пометил как operation contract, на самом деле я могу создать ин-с которые не помечены так, но они будут не доступны извне, не видны в точке доступа, будет доступен толко изнутри приложения, то же самое с контрактом данных и data member

**Контракт сообщений:** сервис обесп методы, которые прописаны сервис-контрактом, понятно что между КС ходят сообщения. Мы как разработчики выс. уровня не знаем что в них находится. Не можем управлять на этом уровне содержимым этих сообщений. А message contract позволяет влезть в этот уровень, на уровне протокола soap добавить свою инфу, кот будет перемещаться вместе с обычной инфой. Контракт сообще – это тоже класс, декорир аннотациями messageHeader, messageBody. SOAP сообщ сост из envelope (header,body,…). Мы также должны обесп соотв проргаммный код, кот будет эти сообщения заполнять и их обраб

Хост – это…..

Рассм взаимоотношения точки доступа на стороне К и сервиса подробнее. На wcf м разраб собс протоколы. Есть неск характеристик, опис взаимоотношения КС:  
\* обесп надежный обмен  
\* безопасность  
\* кодирование  
\* транспорт

Есть набор **привязок**, готовых для использ – указ для конечных точек (либо собственную): какой протокол исп, как будет кодироваться, на каком уровне обесп security, поддерж ли сессия, транзакции, дуплекс канал связи?

Когда будем создавать сервис с неск точакми, етсь еще точка доступа для получения метаданных (MEXT) по точкам доступа выношу в web.config, вместо этоо м с пом классов описать внутри кода, не исп-я web.config

На базе WCF есть много полезных продуктов – OData (Open Data) – веб-протокол, позв вып операции с ресурсами и получить данные формата xml, json: у нас есть источник д-х, к к-му м организовать доступ и получать рез-т в формате xml/json. Все запросы (и insert… даже ddl ! ) к этому ресурсу явл GET http-запросами и всё что мы хотим получить, описывается с пом параметров (с пом спец языка – по мощности равен SQL). Разработан OASIS-ом (они уже разработали кучу стандартов передачи д-х, не надо придумывать ничего нового). Чаще у нас удаленная sql-бд. Фактически он берет наш запрос на языке OData, конвертирует его в sql и делает запрос.

Есть 2 версии технологии data services: я делал на более старой, либо можно на базе MVC. Если вижла новая, надо доустапн элемент (wcf data services), все просто

Есть БД, для нее посмотрим EDMX-модель, для нее построил контекст :DbContext, построил OdataServices, в этом классе указал этот DbContext => у нас готовая служба  
м работать как с обычным REST-сервисом, исп-я язык запросов

Есть парам, указ формат в к-м хотим вернуть данные

**02.10 Микросервисы**

**Микросервисы** – компания Амазон. Надо было представить клиентам Амазона сервис для работы.  
**Осн. задача:** быстрое реагирование на потребности клиента  
1) быстро разраб. новые сервисы  
2) быстро переделывать сервисы, которые есть

**Микросервисы** – арх. подход, осн задача – увел. скорость разработки новых сервисов и переделки старых. Важна именно скорость

Неотрывная часть – **BIOS**. Дб написаны сразу все тесты, построена инфраструктура. Микросервисы, BIOS и команда – неразрывные части

**Микросервис** – средство продажи сервиса, а не программы. Он не явл прогр продуктом. Микросервисы – один из подходов к разработке. Должен быть легким! Маленькие и легковесные.   
**Микросервис** – прогр продукт, кот заново мб написан за 2 недели

Осн цель – продать, сделать устойчивым к сбоям  
Чтобы запустить на неск серверах одновременно, есть балансировщик, кот раскидывает нагрузку, микросервисы дб **автономные**. Отд сервис кот тоже может размножатьс и их хранилище реплицируют др в друга.   
Сервис дб автономный и независимый!

Игнорируется, на каком коде написано. Вычисления – лучше С++, обработка данных – лучше Jango. Позволяют сделать работу программистов эффективнее. «Спринты» программистов и за 2 недели выпускают продукт

Микросервис – это сервис, кот вып 1 элементарную ф-ю. В сервисе скрывается и бэк и фронт, обычно фронт это single page прил. Например, микросервис кот принимает заявку на кредит. 1 функция. Его изменение не д затрагивать другие сервисы – независимость

Есть готовые фреймворки – **нэти** – позв создавать модульную стуктуру, кот м наращиваться (.NET)

На самом деле работает не само наше приложение, а егок копия (iis). Если делаем новую публикацию, то если есть еще подключенный клиент, он еще со старой версией работает. Остальные уже с новой. => с пом iis м менять наше приложение не останавливая работоспос прил-я. В core так делать нельзя

Для микросервисов разработан спец док **CLA (agreement)** – соглашение, от берет на себя та организация, кот предоставляет сервис. Там гарантии, как он работает и т.д.

**ЛК 09.10.20 - NANCY**

Этот фреймворк раб в ASP.NET CORE – Nancy. OWIN – арх core приложения – связ компоненты: хост, веб-сервер и наше прил. Эти компоненты явл независимыми и связ с пом ин-са OWIN.

Katana: берем готовую спецификаицю и дорабатываем только application – для этих апл. есть спец. фреймворки (Nancy, напр.)

Хост, веб-сервер, appl – части OWIN  
Nancy – фреймворк позв разраб прил

Nancy фреймворк скачивается с пом Nuget  
разраб обычое консольное прил – и буду инсталировать package (SelfHosting?...). Внутри main запускаем хост. Есть готовый хост и сервер. И теперь я могу с пом этого фрейморвка обавлять свою фун-сть – та часть, кот наз application

Все что будем добавлять – наз. модули – это класс, явл наследником баз. класса (отд файл .cs) NancyModule. Внутри класса только конструктор – запис. обработчик запроса – выглядит как подобие обработчика маршрутизации.

Nancy в авто- режиме сериализует наш объект в xml/json в завис от того какой Accept приходит от К.

QUERY-парам: мб  
**BIND-модель**: спец механизм, позв связывать параметры, кот есть в запросе, с параметрами вашего модуля.

Генерит класс – указ что те параметры кот мб в запросе, будут преобразованы станд. образом в одноименные переменным структуры/класса, кот мы указ в кач-ве типа

Если есть тип parm – у него есть 2 св-ва: x,y. То такой бинд будет в запросе отыскивать параметры x,y и пытаться их преобразовать

Мб еще св-ва, не предназначенные для binding-а – м их внести в black list, кот мы потом при связывании не используем вот такого типа имена для свзывания. Это чтобы если параметры каким-то образом появится параметр с именем кот. есть в моем классе, а я хочу чтобы он не сработал. Могу указать 2 строки с именами параметров, кот не подлежат связыванию. Теперь если в запросе будет параметр с именем R, Не будет свзяывать эти паарметры со значениями кот есть в parm. Это защита чтобы не было испорчено значение, кот есть в классе.

Это get-модель – часть кот сильно развита в NANCU и позв много чего делать: связывать параметры со св-вами объектов, также позвол валидировать: есть с-ма с пом кот мы можем сразу валидировать параметры кот пришли в запросе. Отд пакет – IsValidate

Если делается с браузера запрос, было бы удобно прилагать инструкцию по использованию этого сервиса. АНалогичный подход здесь – позв создавать какую-то разметку если запрос идет на AccpetHtml. Nancy поддреж 3 разметки: собственную sshtml, html, cshtml – позв в простом режиме вывести к-то текст: написать инструкцию по использованию этого сервиса.

Надо написать view – html или cshtml, у кот. имя как у класса (parms): В html доступан разметка sshtml – super simple html. Accept на html если с браузера – нанси вызываем view с именем parms – в этом view есть модель, в кот доступны эти св-ва : x, y.

Есть класс c именем parms, html c именем parms.

**Обработка POST-параметров:**еще один модуль  
через форму могу передавать параметры – они будут нормально восприниматься если аккуратно расставлен accpet и content-type.

Если accept = content-type = json, То json передаю в кач-ве параметры и ответ получаю тоже json. В зависим от conetnt-type по-разному интерпретируется body. Если даные с формы нормально их обрабаытывает в завис от content-type. Мы пишем один и тот же код и он по-разному работает в завис от тогок акой content-type и accpet

Сонтент-тайп: форма, Json…  
Один код, а формат д-х которые приходят мб различным – и в любом случае nancy их нормально обрабатывате

BodyOnly – учит только параметры из тела  
Если заданы и query И body параметры, мы м. управлять какие из них учитывать: bodyonly – query Игнрируются. М сделать get, post put delete в одном модуле – обычно так и делают. Обрабывают все 4 типа запроса кот предумсотрены в rest

М исп шаблоны auery-параметров: констрейны (как в ПИС). Тоже есть атрибутные способы маршрутизации.   
указ **dynamic** – динамич метод (Пацей): как js-объект – м динамически создавать свойства. Например в ViewBag.type = ; (авто- появляются свойства). Но свойства не имеют опред. типа, надо ставить впееди приведение к типу. Потому что dynamic Не знает о типе объекта, кот ему присвоен.

В NANCY сть 2 типа разметки: html, sshtlm (super simple), но м также применять razer view engine cshtml. М тоже использ ViewBag при передаче параметров во view. Можно тоже использовать partial – частичные представления. Есть такие же section . М применять шаблон и razor разметку.

**ЛК 30.10. Services Gateway Initiative**

С-ма не останавливается. В БД было горячее резервирование, тоже непрерывная работа была. С-ма обычно сост. из сервера, кот обслуж запросы + БД

Обычно задается время, сколько требуется на восст с-мы при различ сбоях. Первыми такие механизмы появились в СУБД: SQL Server 2014 - первый непрерывную работоспос обеспечил.

Когда на IIS устанавливаем прил, оно устан в спец папку (узел), но на самом деле при запуске IIS использ копию этой папки, кот находится у него внутри (скрытая). М публиковать прилож прямо на раб-щий сервер и IIS обеспечит горячий переход с-мы на новую версию. Все старые запросы дообраб на старой версии, остальные обработаются новой версией сервера.

Еще один подход осн на платформк **Docker**: прил упаков в контейнер Docker и с пом Cubernate с-мы запуска неск таких приложений одновременно и потомнаши запросы раскидываются м/этими приложениями. К это приложение работает автономно и неск копий одного и того же прил. Там есть механизм кот позволяет обновить верси. приложения ч/Hibernate - он по очереди меняет версию (в одном сервисе, перезапуск, на другом.....). Делает все аккуратно, плавненько все копии одного сервиса по-тихоньку переустановятся (горячий способ замены компонентов приложения) - на след. лк

Сейчас все с-мы (даже ОС) направлены на обесп непрерывной работы 4/7, чтобы в то же время мы могли изменять софт, не останавливая работу всего приложения. Платформа ALS - тоже основана на Docker-е.

Платформа - непрерывность работы java-приложений, задача: дать разработчику разработать сервер, кот сост из модулей, и не отключая с-му позв отключат модули и изменять их, вся с-ма остается работоспособной. Подход как в JEE: есть консорциум OSGI, суть его в том, чтобы согласовать стандарты компаний (???)

Эта организация, кот использует java, заинтересованы в едином стандарте на такого рода модульную с-му. **OSGI** Разрабатывает только спецификацию, стандарт этой с-мы. А потом др. организации берут этот стандарт и разраб с-мы на его основании. Первыми использовали коомпании Simens. В основе этой спецификации лежит понятие "процесс OSGI". **Суть:** вначале разраб модуль (bundle), кот потом воплощается в jar, кот потом м выполняться. Этот модуль bundleмб динамически присоединен к с-ме модулей, кот разраб в рамках этой OSGI. М создать модуль и внедрить на лету в эту уже раб-щую с-му. Модуль также мб динамически отсоединен (предварительно приостановлен модуль). С-ма конечно частично потеряет свою ф-сть.

В основе ЖЦ osgi-модуля лежит интерфейс BundleActivator. Этот ин-с позволяет фиксировать состояния, кот есть у данного модуля: мб инсталирован, готов к работе, ему доступны все java-классы, бандлы мб зависимы др от друга, есть состояние starting, active - стартинг завершился, stopping - процесс останова, uninstalled - убрать бандл с с-мы (платформы).

схема граф: сплошные линии - явный переход, штриховая - неявный переход

сущ команды управления ЖЦ с-мы. У этой с-мы (плфтаормы), когда мы ее запускаем, у нее есть спец cmd, кот мы запускаем и м вводить команды, позвол выполнять эти install, refresh...

Есть файл META-INF/MANIFEST: там симв имя, версия, экспортирумые пакеты, импортируемые пакеты (из др. бандлов)....

**Все бандлы делят на 3 части:**  
1) **ресурсные** - когда создаем модели (EF), мы создаем нек. модель д-х, кот потом отражается в БД. Смысл тот же: все д-е, кот хотим применяьт, лучше сделать в форме ресурсных бандлов (предназн для хран, записи, извлч данных). Но они не обязательно мэпятся в БД

2) **сервисные** - классы, кот мы м вызывать. М к ним обращаться, это мб бандлы, организ в виде http-сервера, к нему тоже обращаться, м сделать бандл кот вызыв из других бандлов и т.д. Короче тот, кот. можно вызывать. Он сам себя регистрирует в реестре платформы и потом доступ к нему мб организ по этому способу

3) **издатель-подписчи**к - не будем рассм: одни бандлы генер события, др - обрабатывают

Организовать сервисные бандлы в виде http-сервера не очень рационально. Лучше сделать отд http-сервер, а уже эти бандлы реализуют ф-сть этого сервера. Это своеобразный appl-cервер

**Интерфейс** - это контракт с платформой. Сама платформа предст баз классы, абстр и ин--сы. Также использоуются как маркеры. Класс будет помечен как реализует ин-с (ладе если там нет методов)

**06.11 DOCKER**

быстрый вывод микросервисов на боевой сервер, чтобы их м использ конечные юзеры и этот сервис приносил реальную прибыль.

Докер - платформа для разработки, доставки и эксплуатации прил. Приспособлена для разработки прил типа микросеврисаов (прил дост изолированы чтобы их отдельно распространтяь и отд запускать) Если собираемся делать болеее сложные приложения, сост из неск компонентов взаимоде, это тоже можно сдлетаь в докере но это сложнее.

Docker - Это набор программ, кот сост из **2 компонент**: Docker Desktop (engine) + docker hub

docker hub - облачный сервис, репозиторий, там м укладывать компонент, кот мы будем разраб(docker image)  
docker desktop - 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 Namedpace, 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. Чтобы исп докер, надо на комп устан эту платформу - докер десктоп. Сам по себе тот сост из 2 компонентов: сервис (в линукс Демон, в виндоус служба) + прога, с пом к-й м создавать приложения, упакованные в докер контейнер.

Докер енджин нужен для созд и ф-рования контейнеров на нашем ПК. Там есть опр уровни совместимости.

1) устан докер десктоп на свой пк == м создавать контейнер (та единица, кот работает на платформе дкоер. Эта единица осн на понятии image - это образ нашего контейнера. Образ контейнера лежит на диске, м нах в docker hub и в момент запуска он превращ в контейнер. Разница между контейнером и image как между БД и инстансом.

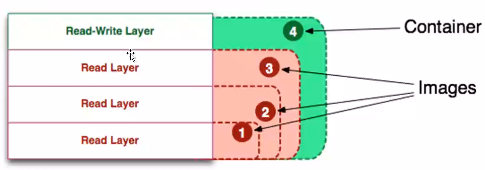
Контейнер - это инстанс какого-то image. Image - файл, содержит в себе данные т. для чтения. Сам по себе контейнер предст собой многослойную структуру. Там есть 3 из 4 уровней т. для чтения - все эти 3 уровня лежат в image. Эти уровни мб вложены друг в друга. Когда создается контейнер - создается новый уровень для чтения-записи. Контейнеру тоже соотв какой-то файл на диске. Ему тоже надо нек. пространство куда он может что-то записывать.   


Image м иметь много слоев, внутри к-х другие Images, Самый верхний слой образуется при создании контейнера. Union file system - опред ФС, которая обесп расположение д-х в контейнере.

В линкус ядра все одинаковые. В виндоус если утсановим докер платформу, есть 2 режима работы с ней: режим виндовс-контейнеров либо в режиме линукс-контейнеров. Если мы говорим что работае м влинук реиме, то наши контенейры будут работать на динуксе, кот. входит в состав виндовса (линуксовое ядро). М брать линуксовые имэжди, устан на своем пк и они будут работать. Но с виндосовскими контейнерами: сам по себе виндовс не поддерж эти name-spac-ы. Поэтому виндоус фактически эмулируют докер, потому что туда заталкивается виртуальная машина. Сам имэдж занимает в 4-5 раз больше места чем под линуксом. Лучше всегда делать контейнеры под линукс.

Когда работаем с докерами, докер хаб - удаленный репозиторий, куда м загружать свои имэджи. Кроме того есть свой лок. репозиторий, кот нах на нашем ПК.