«**УТВЕРЖДАЮ»**

**Зав. каф.ИСиТ В.В. Смелов**

**Экзаменационные вопросы по курсу**

**«Программирование в Internet»(ПОИБМС, 7сем.)**

1. **SOA:** определение, свойства, стандарты, спецификации, интерфейсы, специальные компоненты, способы клиент-серверного взаимодействия, платформы для разработки.
2. **REST API:** определение, стандарт форматы передачи данных, HATEOAS, общепринятые правила REST API, платформы для разработки сервисов.
3. **SOAP:** определение, структура Envelop-сообщений, пространства имен, роли, принципы расширения.
4. **WSDL:** определение, стандарты и версии, концептуальная модель, пространства имен, структура WSDL-документа.
5. **ASMX:** определение ASMX-сервиса, порядок разработки, принципы применения, утилита WSDL.EXE.
6. **WCF:** определение WCF-сервиса, коммуникационная модель, WCF-контракты, WCF-хостинг, конечные точки, стандартные привязки, основные отличия от ASMX-сервисов, поведение и безопасность WCF-сервиса, порядок разработки WCF/RPC и WCF/REST-сервисов, разработка WCF-сервиса с несколькими конечными точками.
7. **WCF Syndication Services:** стандарты ATOM, RSS, порядок разработки WCF Syndication Service.
8. **WCF Data Services:** протокол Open Data Protocol, возможности предоставляемые OData-интерфейсом, порядок разработки Data Services, применение Data Services.
9. **JSON-RPC:**  определение JSON-RPC-сервиса, форматы запросов и ответов, обработка ошибок, пакеты запросов, реализация JSON-RPC на платформе Web API.
10. **ASP.NET CORE Nancy:** интерфейс OWIN,архитектура приложения, принцип разработки сервиса.
11. **Event Storing:** назначение, принципы применения, примеры реализации.
12. **Микросервисы:** микросервисная архитектура, определение микросервиса, основные принципы разработки микросервиса, паттерны разработки, DevOps для микросервисов.
13. **Docker**: назначение, архитектура, основные команды
14. **Docker-Compose**: назначение, основные команды.

**Сеть Интернет – 4 компонента:**

1) сеть на основе ст.пр. TCP/IP

2) набор служб (протокол + сервер, все № портов до 1024) : DNS, FTP, TelNet, POP3

3) стандарты (там прописаны протоколы) : RFC, STD

4) организации, упр-щие этой сетью : IAB, IANA, ICANN, ISOC

**Web-сервис** – прил, предост откр интерфейс для других приложений.Его юзеры – другие приложения. Мы говорим здесь о WEB.API приложениях.

**WEB API** – обычное MVC-прилож (веб-сервис), кот обесп REST-интерфейс – набор uri (get, post, put, delete), опис доступ к удаленной коллекции объектов, кот мы м считывать, удалять, корректировать.   
у к-ра другой базовый класс ( : ApiController)  
другая маршрутизация

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

***КС* прил** – прил, сост из К и С. которые взаимод. в соотв. с заданными (*протоколами*)

**HTTP** – прот.перед.гипертекста – прот.прикл.ур перед.данных Основа – технол.«К-С».

**HTTP-протокол –** формат передачи д-х по TCP.

**URI** (Uniform Resourse ID) – унифицир. id ресурса (док, изобр, файл, служба, email)– имя ресурса, не содержащее месторасп. и способа доступа к ресурсу (можем опр. месторасп. ресурсов в глобальной БД)

**URL** (Uniform Resourse Location) – унифиц. локатор ресурса

**Сокет –** объект ОС, кот. в своих св-ва хранит (IP, порт, сем-во проток). Мы можем с ним работать через сист. вызовы, используя *дескриптор* **(**32-бит. число – id сист.объекта ядра ОС)

**HttpHandler** – простейшее ASP.NET-приложение, предназн. для получения запроса, его обработки, формирования ответа и его отправки.   
**REST:** Representational State Transfer – передача сост. предст-я  
предст – uri, упр. ресурсом с пом глаголов (GET, POST, PUT, DELETE)

**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)

!

**SOA** - **подход к разработке** программного обеспечения, базирующийся на обеспечении удаленного по стандартизированным протоколам использования распределенных, слабо связанных легко заменяемых компонентов со стандартизированными интерфейсам

**REST -** **архитектурный стиль** взаимодействия компонентов распределённого приложения в сети

**SOAP - протокол**, основанный на xml для обеспечения доступа к веб-сервисам

**WSDL - язык** **описания веб сервисов и доступа** к ним, основанный на xml.

**ASMX - технология** для разработки веб-сервисов на основе WSDL,SOAP,XML.

**WCF - технология** для разработки приложений SOA архитектуры основанная на .net

**ATOM - формат описания web-ресурсов** и протокола для их публикации

**RSS - формат** описания новостных лент, статей, основанный на XML

**OData –** открытый **web-протокол**; позволяет выполнять операции с ресурсами и получать ответы в форматах XML, JSON.

**JSON-RPC - протокол** удаленного вызова процедур, использующий формат JSON для передачи сообщ.

**Nancy – фреймворк** для разраб приложений микро сервисной арх.

**OWIN – интерфейс** между net-web-сервером, серверным приложением и хостом.  
**Event Storing - подход**, который заключается не в сохранении данных, а событий, которые происходят с д

**Микросервисная архитектура — это подход,** при котором приложение строится как набор сервисов

**Docker - инструмент** для разработки, контейнеризации и развертывания приложений.

1. **SOA:** определение, свойства, стандарты, спецификации, интерфейсы, специальные компоненты, способы клиент-серверного взаимодействия, платформы для разработки.

# SOA: Service-oriented architecture, сервис-ориентированная архитектура – подход разработки программного обеспечения, основанная на применении распределенных слабосвязанных компонентов, обеспечивающих стандартные интерфейсы.

**SOA:** компоненты распределенной системы SOA – узлы - сервисы.

**SOA: основные свойства**

* + независимость от аппаратной реализации узлов;
  + независимость от операционной системы в узлах;
  + независимость от языка программирования разработки сервиса; - масштабируемость.

**SOA:** **сервис** – это видимый ресурс, выполняющий повторяющуюся задачу и описанный внешней инструкцией.

**SOA:** свойства сервиса

* + сервис ориентирован на бизнес;
  + сервис автономен;
  + повторное использование;
  + четко описанная инструкция в терминах интерфейса;
  + сервер виден (доступен).



основные стандарты W3С: XML**, SOAP, WSDL, UDDI**

**ИНТЕРФЕЙСЫ – REST, SOAP, JSON-RPC**

спецификации второго уровня W3C (WS\*):

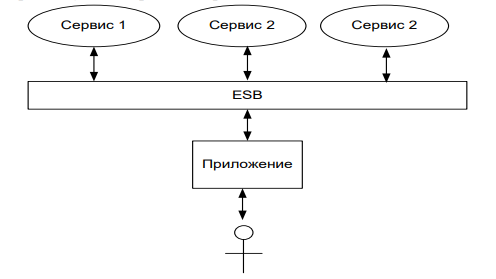
* WS-Policy, WS-PolicyAttachment, WS-PolicyAssertion (описание политик web-сервиса);
* WS-Addressing (механизм адресации web-cthdbcf);
* WS-Security (целостность и конфиденциальность web-сервисов);
* WS-Trust (механизм получения маркеров защиты);
* WS-SecureConversion (создание безопасной сессии обмена сообщениями);
* WS-SecurityPolicy (определяет набор утверждений политики безопасности);
* WS-Federation (объединение защищенных доменов);
* WS-Transfer (механизм обновления, создания и удаления ресурсов);
* WS-ResourceTransfer, WS-Fragment (обеспечивает частичный доступ к ресурсам);
* WS-MetadataExchange (механизм получения метаданных);
* WS-Enumeration (механизм получения данных больших размеров);
* WS-Eventing (механизм уведомления о событиях web-сервисов);
* WS-Management (SOAP-управление системами);
* WS-Discovery (механизмы публикации и поиска web-сервисов);
* WS-ReliableMessaging, WS-ReliableMessagingPolicy (механизмы надежной передачи сообщений между web-сервисами);
* WS-MakeCpnnection (установка соединения с сервисом не имеющего доступный адрес);
* WS-Coordination (механизмы взаимодействия web-сервисов);
* WS-AtomicTransaction (поддержка транзакций web-сервисов);
* WS-BusinessActivity (координация бизнес-взаимодействия web-сервисов).

Компоненты

**SOA: ESB – Enterprise Service Bus:** программный компонент обеспечивающий обмен сообщениями между различными информационными системами, имеющих сервис-ориентированную структуру.

**SOA: ESB**

* + синхронный и асинхронный вызов сервисов;
  + гарантированная доставка сообщений;
  + поддержка транзакций;
  + маршрутизация сообщений;
  + мониторинг, аудит и протоколирование.



**SOA: специальные компоненты** (как правило часть ESB)

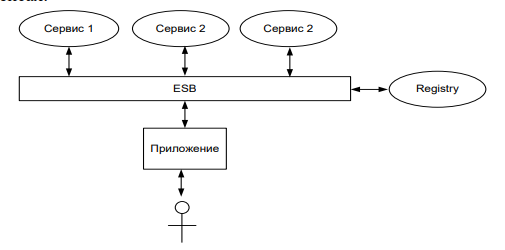
- SOA Registry;

- Workflow Engine;

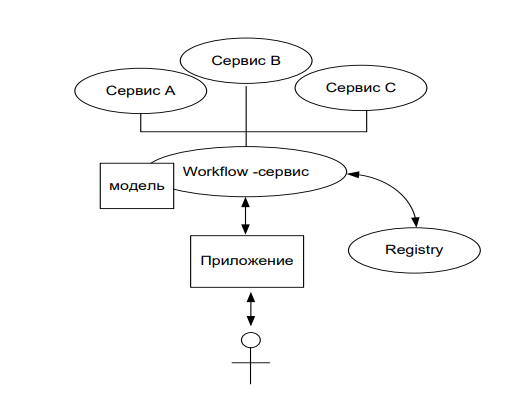
- Service Broker;

- SOA Supervisor.

**SOA:** **SOA Registry –** реестр сервисов, информация о сервисах и их интерфейсах.



**SOA:** **Workflow Engine –** программный компонент, предназначенный для построить (построить модель бизнес процесса) и выполнить бизнес процесс на основе группы сервисов. Другими словами разработать новый сервис на основе последовательного выполнения нескольких сервисов.

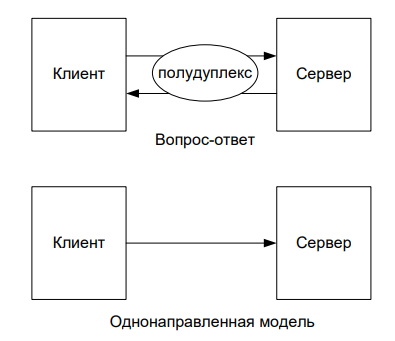
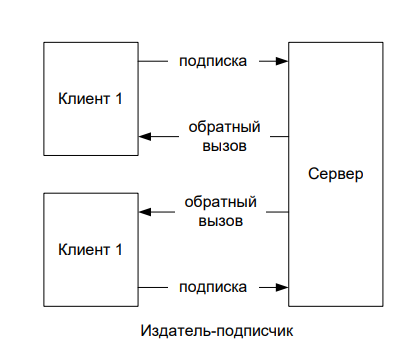
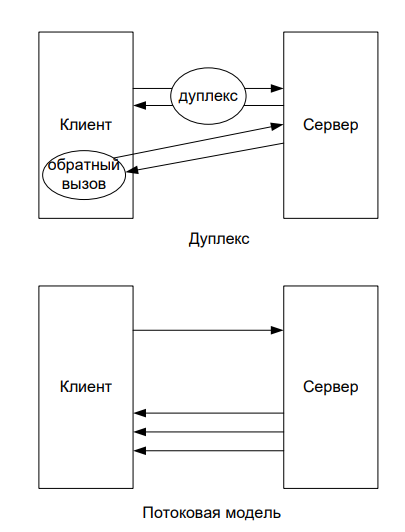


**SOA: Service broker** –программный компонент, позволяющий трансформировать запрос пользователя в системе в запуск и координированную работу.



**SOA: SOA Supervisor** – служебный сервис, предназначенный для управления и мониторинга других служебных сервисов. Одна из основных задач СОА супервизора это отслеживать работу различных компонентов внутри СОА системы, оценивать корректность их функционирования, а также отслеживать запросы, посланные во внешние системы.

**SOA: способы клиент-серверного взаимодействия**

+ Полудуплекс? – режим, когда два компонента могут отпр и получать данные с разделением по времени. А дуплекс могут одновременно.

+Обратный вызов – функция, которая вызывается позже.

1. **REST API:** определение, стандарт форматы передачи данных, HATEOAS, общепринятые правила REST API, платформы для разработки сервисов.

# REST- Архитектурный стиль взаимодействия компонентов распределенного приложения. Для обращения взаимодействия стандарты HTTP URL XML JSON Форматы XML JSON HTML.

REST: Representational State Transfer - передача состояния представления.   
REST: представление – это URI (идентификатор ресурса), управление ресурсом с помощью глаголов (GET, POST, PUT, DELETE).   
REST: Архитектурный стиль взаимодействия компонентов распределенного приложения.   
REST: реализация (альтернатива) RPC.  
RESTful: описание web-службы, поддерживающей REST-интерфейс в полном объеме (со строгим соблюдением правил).  
REST: нет официального стандарта, но REST использует стандарты HTTP, URL/URI, XML, JSON.

**REST**: форматы передачи данных – xml и json  
REST: два основных типа ресурса – коллекции и элемент коллекции: /api/users, /api/users/288.   
HATEOAS: Hypermedia As The Engine Of Application State – гипермедиа в качестве управления состоянием.  
Гипермедиа: технология обработки, структурирования информации и произвольного доступа к ее элементам с помощью гиперсвязей (Тед Нильсон, 1965), WWW – реализация гипермедиа.   
**REST: общепринятые правила**1. Общий префикс для всех ресурсов сервиса … /API/…,   
[http://API.BSTU.BY/](http://api.bstu.by/)...  
2. Два типа ресурсов: коллекция (users, students, …), элемент коллекции /api/users/238, /api/students/ef3d26.  
3. Иерархическая связь …/api/users/238/cars/aah4899  
4. Доп коды ошибок (например 20003, 404001,…),сделать отдельный ресурс (HATEOAS link) для пояснения ошибок http://ccc/api/errors/20003.  
5. Подавление статуса ответа …/api/students/ef3d26?status\_code=200.  
6. Версионность /api/students/ef3d26?v=7.  
7.Постраничное получение данных: параметры limit, offset.  
8.Сортировка: параметр sort.  
9.Все фильтры вынести за знак вопроса: …/api/students?minbday=1998101&maxbday=20001231&gender=m.  
10.Пользователь получает только то, что хочет: …/api/students?field=bday,surname,gender.  
11.Обозначать в запросе формат сообщений (желательна поддержка нескольких форматов):  
…/api/students.json?field=bday,surname,gender; один из форматов должен быть по умолчанию; могут применяться заголовки Accept и Content-Type со значениями application/xml и application/json для запроса или обозначения в ответе формата.  
12.Глобальный поиск: ../api/search?q=19600107+Иванов.  
13.Документация.   
REST: недостатки  
- нет общепризнанного стандарта RESTful API; - не все браузеры поддерживают полный словарь REST-методов (PUT, DELETE); на практике часто используется только GET и POSТ(insert, delete, update);   
- не однозначны коды состояний.

**REST: шесть обязательных ограничений:**  
- модель клиент-сервер;  
- отсутствие состояния на стороне сервера, допускается на стороне клиента  
- кэширование на стороне клиента, сервер явно управляет кэшированием;  
- единообразие интерфейсов (идентификация ресурсов, манипуляция ресурсами через представления, самодостаточные сообщения, HATEOAS);  
- для клиента сервер должен представляться конечным;  
- код по требованию: допускается (необязательно) выгрузка на клиент апплетов или сценариев для расширения его функциональности.   
REST: GET (select), POST(insert), PUT (update), DELETE (delete).   
Платформы: Http-handler webapi servlet

1. **SOAP:** определение, структура Envelop-сообщений, пространства имен, роли, принципы расширения.

# SOAP — это основанный на XML протокол для доступа к веб-сервисам. У него есть некоторая спецификация, которая может быть использована во всех приложениях.

SOAP известен как простой протокол доступа к объектам, но в более поздние времена был сокращен до SOAP v1.2. SOAP — это протокол, или, другими словами, это определение того, как веб-сервисы взаимодействуют друг с другом или взаимодействуют с клиентскими приложениями, которые их вызывают.  
SOAP-сообщение — это обычный XML-документ, содержащий следующие элементы:  
Конверт — определяет начало и конец сообщения. Это обязательный элемент.  
Заголовок — содержит любые необязательные атрибуты сообщения, используемые при обработке сообщения, либо в промежуточной точке, либо в конечной конечной точке. Это необязательный элемент. Содержит теги actor определяет получателя хедера и mustUnderstand= true требует обязательную обработку  
Тело — содержит данные XML, содержащие отправляемое сообщение. Это обязательный элемент.  
Неисправность — необязательный элемент неисправности, который предоставляет информацию об ошибках, возникающих при обработке сообщения.

SOAP: роль - набор правил, определяющих поведение узла,  
задается атрибутом env:role.  
SOAP: три стандартные роли для узла (нет роли none,  
промежуточный узел next, конечный получатель ultimateReciever), роль может быть  
пользовательской (задаваться собственным URI)  
  
SOAP: роль может быть указана во всех блоках и указывает  
кому предназначена информация блока. Обычно, обработанный  
блок извлекается из конверта, если не указано другого  
  
Определены три стандартизованные роли (см. SOAP Часть 1 Раздел 2.2):  
  
"<http://www.w3.org/2003/05/soap-envelope/role/none>" (далее просто "none");  
  
"<http://www.w3.org/2003/05/soap-envelope/role/next>" (далее просто "next");  
  
"<http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver>" (далее просто "ultimateReceiver").

Неймспейсы:   
soap- encoding, soap-envelope, soap-rpc, representation, xor, xmlmime

Является расширением протокола XML-RPC

Расширение за счёт неймспейсов, создаём свои неймспейсов на основе соап и расширяем до бесконечности

1. **WSDL:** определение, стандарты и версии, концептуальная модель, пространства имен, структура WSDL-документа.

# WSDL (англ. Web Services Description Language /ˈwɪz dəl/) — язык описания веб-сервисов и доступа к ним, основанный на языке XML.

версия 1.1(2001), версия 2.0(2007, статус рекомендации).

При разработке веб-служб наиболее активно используются следующие стандарты: UDDI (Universal Description, Discovery and Integration), WSDL (Web Services Description Language), WSIL (Web Services Inspection Language) и WS-I (Web Services Interoperability).

UDDI - это открытый и независимый от платформ **стандарт** публикации **описаний веб-служб** в глобальных реестрах, **поиска** веб-служб в этих реестрах и определения **способов** взаимодействия между службами в сети **Internet**.

Структура: definitions  
- types служит для определения базовых типов данных, которые далее используются для определения сообщений.  
- message — для определения абстрактных сообщений, каждое из которых состоит из логических частей, определяемых с использованием какой-либо системы типизации.  
- portType — набор абстрактных операций, для каждой из которых определены типы входящих и исходящих сообщений.  
- binding — устанавливает транспортный протокол и формат данных для операций и сообщений конкретного portType.  
- port — задает адрес для конкретного связывания, тем самым определяет единичную конечную точку.  
- service используется для группировки логически связанных портов, образующих службу.  
Концептуальная модель:   
делится на два уровня: интерфейс( message, operation, interface) и реализация (binding, service, endpoint)  
Namespaces: wsdl, wsdl-instance, wsdl-extencions, wsdl-http, wsdl-soap, wsdl- rpc

1. **ASMX:** определение ASMX-сервиса, порядок разработки, принципы применения, утилита WSDL.EXE.

# ASMX: Active Server Method Extended; технология Microsoft для разработки web-сервисов), основанная на XML, WSDL, SOAP; легкая технология: для работы с ней не обязательно знать XML, SOAP и WSDL.

WSE(Web Services Enhancements – улучшение)- дополнение к ASMX основанное на спецификациях WS-\*.

ASMX сервисы это частный случай WCF сервисов. Всегда используются в качестве транспорта (поэтому HTTP и полудуплекс). ASMX хостятся на IIS и сейчас в составе ASP.NET и WCF.

Веб-служба, веб-сервис — идентифицируемая уникальным веб-адресом программная система со стандартизированными интерфейсами. Веб-службы могут взаимодействовать друг с другом и со сторонними приложениями посредством сообщений, основанных на определённых протоколах и соглашениях  
  
Прокси-сервер является заменой для кодов веб-службы. Перед использованием веб-службы необходимо создать прокси. Прокси-сервер зарегистрирован в клиентском приложении. Затем клиентское приложение выполняет вызовы веб-службы, как если бы они использовали локальный метод.  
  
Прокси принимает вызовы, упаковывает их в надлежащий формат и отправляет на сервер в виде SOAP-запроса. SOAP означает простой протокол доступа к объектам. Этот протокол используется для обмена данными веб-службы.  
  
Когда сервер возвращает пакет SOAP клиенту, прокси-сервер декодирует все и представляет его клиентскому приложению.  
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 Можно с генерировать прокси, стабы, клиент

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  
Утилита wsdl.exe является соответствующей для asmx техникой потребления SOAP веб-сервисов. По wsdl файлу или ссылке она генерирует прокси-класс – специальной класс, максимально упрощающий обращение к данному веб-сервису. Разумеется, не важно на какой технологии реализован сам веб-сервис, это может быть что угодно — ASMX, WCF, JAX-WS или NuSOAP. Кстати, у WCF аналогичная утилита называется SvcUtil.exe.

ASMX предоставляет возможность создавать веб-службы, отправляющие сообщения с помощью протокола SOAP Потребители службы ASMX не должны знать ничего о платформе, объектной модели или языке программирования, используемом для реализации службы. Им нужно только понять, как отправлять и получать сообщения SOAP

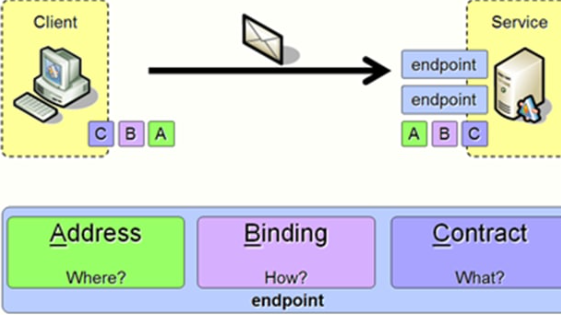
1. **WCF:** определение WCF-сервиса, коммуникационная модель, WCF-контракты, WCF-хостинг, конечные точки, стандартные привязки, основные отличия от ASMX-сервисов, поведение и безопасность WCF-сервиса, порядок разработки WCF/RPC и WCF/REST-сервисов, разработка WCF-сервиса с несколькими конечными точками.

# WCF: Windows Communication Foundation; технология основанная на .NET FRAMEWORK для разработки приложений SOA-архитектуры, первая версия 2006 (Indigo).

**Windows Communication Foundation** (WCF) ― это унифицированная интегрированная среда для создания защищенных, надежных, транзакционных и интероперабельных распределенных приложений. (Это технология для создания SOA приложений).

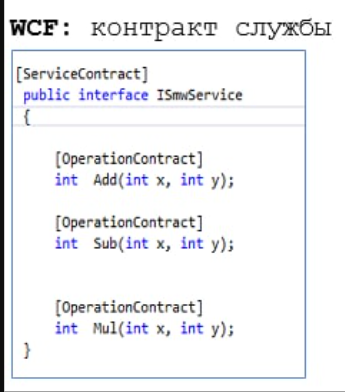
Soa - подход к разработке программного обеспечения, базирующийся на обеспечении удаленного использования распределённых, слабо связанных легко заменяемых компонентов (сервисов) со стандартизированными интерфейсами  
WCF: основные принципы  
- разработка сервиса должна быть простой и иметь способность к расширению его функциональных возможностей;

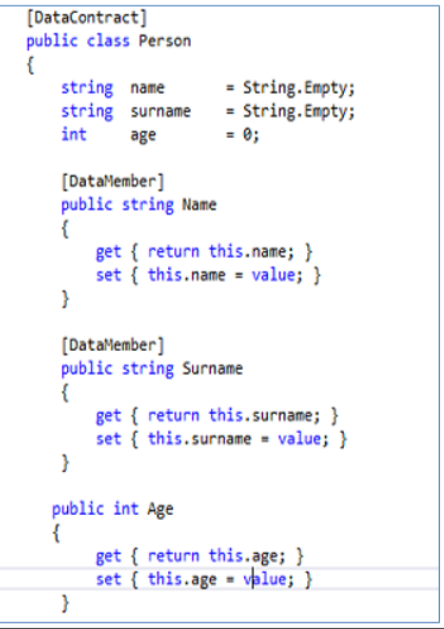
- один API для всех коммуникационных протоколов(например tcp-ip);  
- сервис должен поддерживать стандарты WS-\*(веб сервисов);  
- сервис должен поддерживать REST, RPC и др. архитектуры;

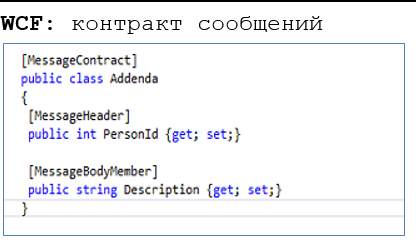


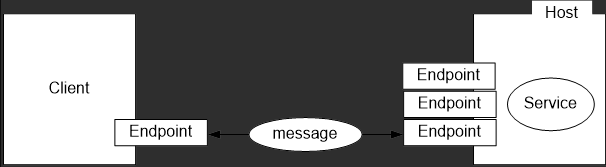
Модель коммуникации

WCF: контракты  
- контракт службы;  
- контракт данных;  
- контракт сообщений.  
Контракт на обслуживание – этот контракт предоставляет клиенту, а также внешнему миру информацию о предложениях конечной точки и протоколах, которые будут использоваться в процессе связи.  
  
Контракт данных – данные, которыми обменивается сервис, определяются контрактом данных. И клиент, и сервис должны быть согласованы с договором на данные.  
  
Контракт сообщения – контракт данных контролируется контрактом сообщения. Он в первую очередь выполняет настройку форматирования типов параметров сообщений SOAP. Здесь следует отметить, что WCF использует формат SOAP для связи. SOAP означает простой протокол доступа к объектам.

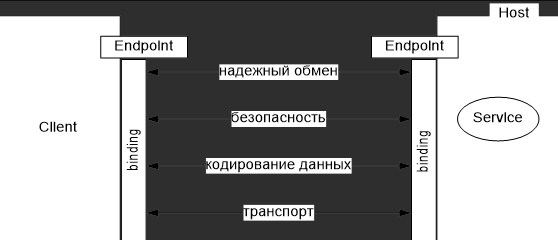


Контакт данных, указывает каким образом данные будут сериализованы  




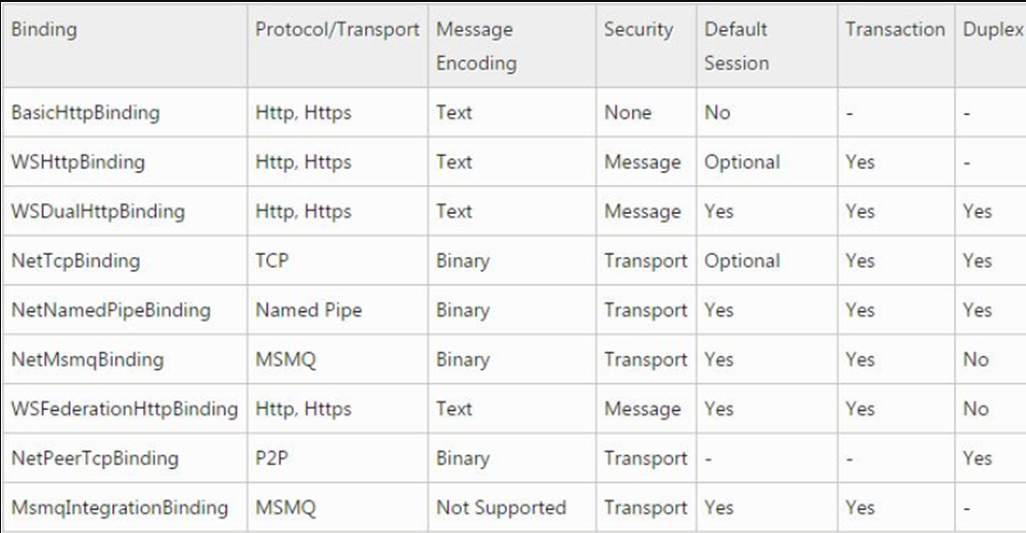


9. WCF: хост – контейнер для сервиса (любое С#-приложение). Основное назначение хоста: присоединить WCF-инфраструктуру, создать список конечных точек.



10. WCF: конечная точка – адрес, привязка, контракт.  
11. WCF: привязка – инкапсулирует все технологии, позволяющие переправить сообщение от одной конечной точке к другой (транспорт, кодировка, протоколы (безопасность,сессия), правила обмена сообщениями (размер, timeout)).

Стандартные привязки



1. Отличия:  
   ASMX веб сервисы могут быть размещены только в IIS, в то время как WCF сервис имеет следующие варианты хостинга:
   * IIS
   * WAS (Windows Process Activation Services)
   * Console Application
   * Windows NT Services
   * WCF provided Host
2. ASMX веб сервисы ограничены в поддержке одного лишь HTTP,  в то время как WCF поддерживает HTTP, TCP, MSMQ, Named Pipes.
3. Безопасность ASMX ограничена. Обычно аутентификация и авторизация производится используя конфигурацию безопасности IIS и ASP.NET и безопасность транспортного слоя. WCF предоставляет согласованную программную модель безопасности для любого протокола и поддерживает много таких же возможностей как IIS и WS-\* протоколы безопасности.
4. ASMX веб сервисы используют для сериализации класс  XmlSerializer, в то время как WCF использует DataContractSerializer, который намного лучше по производительности чем XmlSerializer.
5. WCF имеет встроенный механизм ведения журналов, тогда как в других технологиях необходимо выполнять необходимое кодирование.
6. WCF интегрировал AJAX и поддерживает JSON (нотация объектов JavaScript).
7. Он предлагает масштабируемость и поддержку новых стандартов веб-сервисов.

WCF: привязка: безопасность: WS-ReliableMessaging;  
WCF: привязка: безопасность: WS-Security

**порядок разработки WCF/REST-сервисов:**

**создаем приложение wcf, в файл конфигурации создаем эндпоинт и определяем биндинги, дальше создаем интерфейс, помечен атрибутом [ServiceContract], а его методы атрибутами [OperationContract**].

Создаем если нужно классы, сдержащие messageContract или dataContract, **далее интерфейс реализуем в классе сервиса.**

Также можно заметить, что каждый метод помечен атрибутом [WebGet] или [WebInvoke]. Данные атрибуты являются одна из составляющих механизма REST в WCF.  
Для операции GET используется атрибут [WebGet]. Поскольку в этой операции данные передаются на сервер в строке запроса, необходимо указать такой параметр как UriTemplate – это текстовая константа, которая обозначает, каким образом параметры строки запроса будут мапиться на параметры метода. Стоит отметить, что параметры могут быть только строкового типа!  
Для всех остальных операций кроме GET используется атрибут [WebInvoke]. У него есть параметр Method, где необходимо указать название операции (POST, PUT, DELETE). Почему-то этот параметр сделали строкой, а не перечислением.  
Также, у обоих атрибутов есть такие параметры, как RequestFormat и ResponseFormat, которые определяют, в каком формате сервис будет принимать, и отдавать данные.





Безопасность 4 принципа:  
Аутентификация, авторизация, конфиденциальность, целостность сообщений,  
конфиденциальность сообщений  
WCF предлагает следующие режимы безопасности передачи, чтобы обеспечить защищенную связь между клиентом и сервером  
<security mode = "None"/>  
None,Transport,message,cмешанный

**Транспорт.** Этот режим является наиболее простым способом обеспечения безопасной передачи сообщений с использованием протоколов связи, таких как TCP, IPC, Https и MSMQ. Этот режим более эффективен, когда передача является двухточечной и используется в основном в контролируемой среде, то есть в приложениях интрасети.

**Сообщение** — режим безопасности обеспечивает взаимную аутентификацию и в значительной степени обеспечивает конфиденциальность, поскольку сообщения зашифрованы и могут передаваться через http, который не считается безопасным протоколом. Здесь безопасность обеспечивается сквозной без учета того, сколько посредников вовлечено в передачу сообщения и есть ли защищенный транспорт или нет. Режим обычно используется интернет-приложениями.

**Смешанный** — этот режим безопасности используется не часто, и аутентификация клиента предлагается только на уровне клиента.

**И то и другое.** Этот режим безопасности включает в себя как транспортную безопасность, так и безопасность сообщений, чтобы обеспечить надежную защиту, но часто приводит к перегрузке общей производительности. Этот поддерживается только MSMQ.

Все привязки WCF, кроме BasicHttpBinding, имеют некоторую степень безопасности передачи по умолчанию.

Уровень безопасности сообщений  
<message clientCredentialType = "None"/>  
Windows ,Имя пользователя,Сертификат

1. **WCF Syndication Services:** стандарты ATOM, RSS, порядок разработки WCF Syndication Service.

# Syndication Service Library – для создания синдицированного сервиса типа новостной ленты RSS или ATOM – это сервисы которые предназначены обычно для новостных полос, обычно используют формат atom/rss, это частный случай wcf сервисов.

ATOM:  
- формат описания web-ресурсов и протокола для их публикации;  
- последняя версия 1.0;  
- MIME: application/atom+xml;  
- RFC-4287 - The ATOM Syndication Format;  
- RFC-5023 - The ATOM Publishing Protocol.  
основные теги:

Xml, feed, entry

RSS (Rich Site Summary):  
- семейство XML-форматов, предназначенного для описания новостных лент, анонсов и статей;  
- последняя версия 2.0 (2007);  
- MIME: application/rss+xml;  
- интернет-ресурс в формате RSS называется RSS-каналом, RSS-лентой, RSS-фидом.  
  
RSS-агрегатор - web-приложение, которое автоматически с заданным интервалом времени проверяют, указанные в настройках ресурсы   
и экспортируют в формате RSS или ATOM информацию.  
Основные теги:

- Rss, channel, item  
Обязательные RSS-элементы:  
- title link description  
Опциональные RSS-элементы:  
- language category docs image

Порядок разработки:

создается edm модель, создается интерфейс, в котором размещается контракт службы WCF. В интерфейсе содержится метод CreateFeed(), который возвращает SyndicationFeedFormatter. Поскольку SyndicationFeedFormatter представляет собой абстрактный класс, а в реальности возвращается класс либо типа Atom10FeedFormatter, либо типа Rss20FeedFormatter, эти типы перечисляются в ServiceKnownTypeAttribute, чтобы тип был известен и могла выполняться сериализация. Далее мы реализовавыем интерфейс для форматирования данныхь в необходимом формате.

1. **WCF Data Services:** протокол Open Data Protocol, возможности предоставляемые OData-интерфейсом, порядок разработки Data Services, применение Data Services.

# **OData: Open Data Protocol – открытый web-протокол; позволяет выполнять операции с ресурсами и получать ответы в форматах XML, JSON.**

http://localhost:58061/StudentNotes.svc/Students(2)/Notes?$format=json&$select=Note1,Subject&$orderby=Note1%20desc&$filter=Note1%20ge%203  
Top skip мы не использовали,но они есть

Select=

Orderby =

Filter= Subject ge 3

=startswith

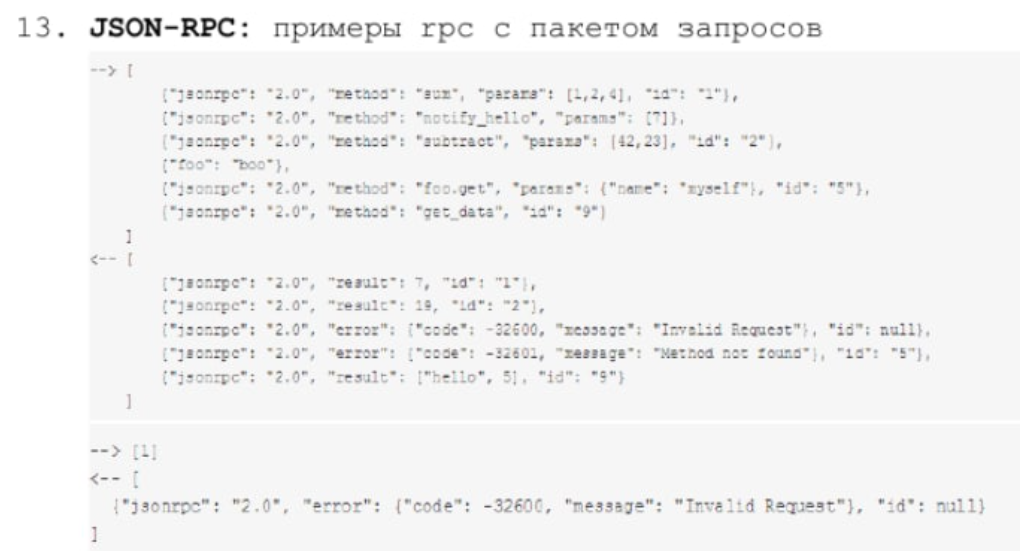
Возможности: интеграция со сторонними ресурсами, реализация доп функций без именения программы, загрузка данных, получение данных с фильтрами, создание новых или изменение документов.

Порядок разработки:  
1)cоздаем бд  
2)создаем edm model и подключаемся к нашей бд

3)создаем интерфейс сервиса с атрибутами сервис контракт и внутри операция контракт  
4)создаем файл WCF Data Services (расширение svc), который реализует интерфейс

WCF Data Services — это реализация стандарта Open Data Protocol (OData). В WCF Data Services можно предоставлять табличные данные в виде набора REST API, что позволяет возвращать данные с помощью стандартных HTTP-команд, таких как GET, POST, PUT и DELETE.

1. **JSON-RPC:**  определение JSON-RPC-сервиса, форматы запросов и ответов, обработка ошибок, пакеты запросов, реализация JSON-RPC на платформе Web API.



# JSON-RPC:протокол удаленного вызова процедур, использующий формат JSON для передачи сообщений;

- последняя версия 2.0;

- транспорт: HTTP/HTTPS или TCP Socket.

Запросы бывают Позиционные, именованые, с уведомлением(без ответа), с ошибкой, пакетный

Объект запроса:

- jsonrpc: строка, указывающая версию JSON-RPC протокола. Должна быть 2.0;

- method: строка, содержащая метод, который должен быть исполнен;

- params: параметры метода. Не обязательный;

- id: идентификатор установленный клиентом. Может содержать строку, число или null. Если не включен, то предполагается запрос без ответа (уведомления).

Объект ответа:

- jsonrpc: строка, указывающая версию JSON-RPC протокола. Должна быть 2.0;

- result: содержится в случае успешного выполнения. Содержит результат выполненного метода;

- error: содержится в случае ошибки (см. объект ошибка);

- id: обязательный параметр. Должен быть тот же, что и в объекте запроса.

Объект ошибки:

- code: код ошибки;

- message: описание ошибки;

- data: дополнительная информация об ошибке. Не обязательный.

Кода ошибок:

- Кода от -32768 до -32000 зарезервированы для предопределенных ошибок. Кода не описанные ниже, но входящие в заданный интервал нужны для будущего использования:

-32700 - ошибка парсинга

-32600 - неверный запрос

-32601 - метод не найден

-32602 - неверные параметры

-32603 - внутрення ошибка JSON-RPC

от -32000 до -32099 - ошибка сервера

Конвенции

Поскольку JSON-RPC использует JSON, он имеет ту же систему типов. JSON может представлять четыре примитивных типа (Strings, Numbers, Booleans и Null) и два структурированных типа (Objects и Arrays).

1. **ASP.NET CORE Nancy:** интерфейс OWIN,архитектура приложения, принцип разработки сервиса.

# ASP.NET CORE – Nancy Это фреймворк для создания микросервисов на платформе .NET. OWIN –интерфейс, который связывает независимые компоненты арх core приложения: хост, веб-сервер и наше прил.

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 добавлять 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.

В NANCY сть 2 типа разметки: html, sshtlm (super simple), но м также применять razor 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.

**OWIN** **(Open Web INterface for NET)**  
– интерфейс между net-web-сервером, серверным приложением (обработчиками запросов) и хостом

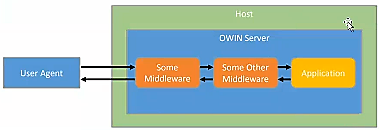
OWIN позволяет ослабить зависимость веб-приложений от веб-сервера. Он определяет стандартный способ использования ПО промежуточного в конвейере для обработки запросов и связанных с ними откликов. Приложения ASP.NET Core и ПО промежуточного слоя могут взаимодействовать с приложениями, серверами и ПО промежуточного слоя, основанными на OWIN.

OWIN обеспечивает разделительный уровень, позволяющий совместно использовать две платформы с разнородными объектными моделями. Пакет Microsoft.AspNetCore.Owin содержит две реализации адаптера:

* ASP.NET Core в OWIN
* OWIN в ASP.NET Core

Это позволяет размещать ASP.NET Core поверх сервера или узла, совместимого с OWIN, а также запускать другие совместимые с OWIN компоненты поверх ASP.NET Core.

подход к разработке веб-сервером

архитектура:  
  
есть OWIN-сервер (веб-сервер) и есть внутри наше приложение (оранж и желт). OWIN опис как они м/собой взаимод. User Agent – http-клиент, взаимод с нашим приложением. Хост нужен чтобы создать процесс ОС, кот потом м включить в себя объект сервера. Сервер приним запросы, обраб битовую послед-сть, формир респонс и отпр его. OWIN – набор ин-сов, кот. должны реализовать объекты сервера, объекты хоста и объекты приложения. Когда разраб прилож, мы фактически реализуем методы, опис в этих ин-сах.

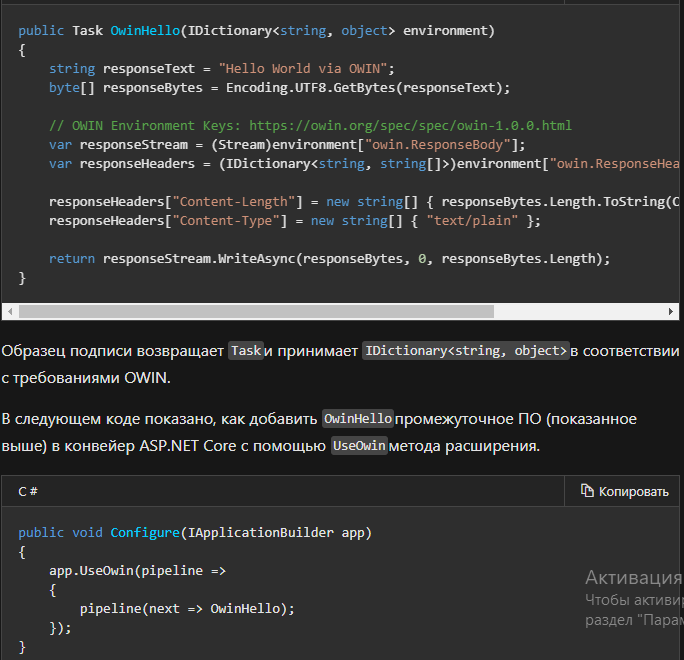
Хост, веб-сервер, appl – части OWINOWIN (Open Web Interface for .NET) связывает хост, вебсервер и собственно ваше приложение. Позволяет сделать свой собственный хост. Компоненты являются независимыми и связываются с помощью OWIN.

ASP.NET Core поддерживает открытый веб-интерфейс для .NET (OWIN). OWIN позволяет отделить веб-приложения от веб-серверов. Он определяет стандартный способ использования промежуточного программного обеспечения в конвейере для обработки запросов и связанных ответов. Приложения и промежуточное ПО ASP.NET Core могут взаимодействовать с приложениями, серверами и промежуточным ПО на основе OWIN.

OWIN обеспечивает уровень разделения, который позволяет использовать вместе две инфраструктуры с разными объектными моделями.

Это позволяет размещать ASP.NET Core поверх OWIN-совместимого сервера / хоста или запускать другие OWIN-совместимые компоненты поверх ASP.NET Core.

Поддержка OWIN в ASP.NET Core развернута как часть Microsoft.AspNetCore.Owin пакета.



Используя OWIN, мы вольны подключать только те компоненты, которые нам нужны прямо здесь и сейчас, будь то модуль авторизации, тот же SignalR, статические страницы и т.п. И в отличие от IIS, например, наш сервер не будет перегружен ненужным функционалом, а значит будет производительнее.  
  
Katana — это OWIN-совместимый хост написанный Microsoft. Помимо реализации спецификации OWIN, Katana содержит различные вспомогательные классы и обертки для упрощения разработки, содержащиеся в библиотеке Owin.Types. Например два класса-адаптера упрощающие работу со словарём: OwinRequest и OwinResponse, реализация WebSocket для Owin — OwinWebSocket, вспомогательный класс для работы с заголовками и другими параметрами запросов — OwinHelpers.

Шаг 1: Создаем консольное приложение на C#  
Разумеется, это можно сделать и на другом языке для платформы .NET. Даже на VB.  
  
Шаг 2: Импортируем Nancy  
Nancy предоставляется в виде пакетов Nuget, так что рекомендую воспользоваться менеджером пакетов Visual Studio – самым удобным инструментом для импорта двоичных файлов в проект и ссылки на них. В данном случае нам понадобится пакет Nancy.Hosting.Self, зависящий от основного пакета Nancy.  
  
Шаг 3: Создаем хост Nancy  
В методе Main (или эквивалентной входной точке программы) напишите:

using (var host = new NancyHost(new Uri("http://localhost:1234"))

{

Console.ReadKey();

}  
Вы уже создали консольное приложение, которое слушает HTTP на порте 1234. На самом деле, нам нравится так делать, это простая реализация обратных прокси, передающих внешние HTTP-запросы управляемому процессу. Однако, Nancy поддерживает и OWIN, традиционный IIS-хостинг.  
  
Шаг 4: Создаем маршрут к ресурсу  
Создаем наш первый маршрут, наследуя класс Module из Nancy. Напишите вот это в файле с новым проектом:

class Dinosaur

{

public string Name { get; set; }

public int HeightInFeet { get; set; }

public string Status { get; set; }

}

class DinosaurModule : NancyModule

{

private static Dinosaur dinosaur = new Dinosaur()

{

Name = "Kierkegaard",

HeightInFeet = 0,

Status = "Deflated"

};

public DinosaurModule()

{

Get["/dinosaur"] = parameters => dinosaur;

}

}

При запуске хоста Nancy просматривает вашу сборку и ищет в ней классы, наследующие NancyModule. Они будут инстанцироваться всякий раз при поступлении запроса, обеспечивать маршрутизацию и действия. В данном случае мы создаем простой маршрут GET в конструкторе модуля и пользуемся лямбда-выражением, при помощи которого возвращаем определенный нами объект модели. Если вызвать [localhost](http://localhost/):1234/dinosaur без заголовка с типом содержимого, то модель динозавра придет нам в формате JSON.

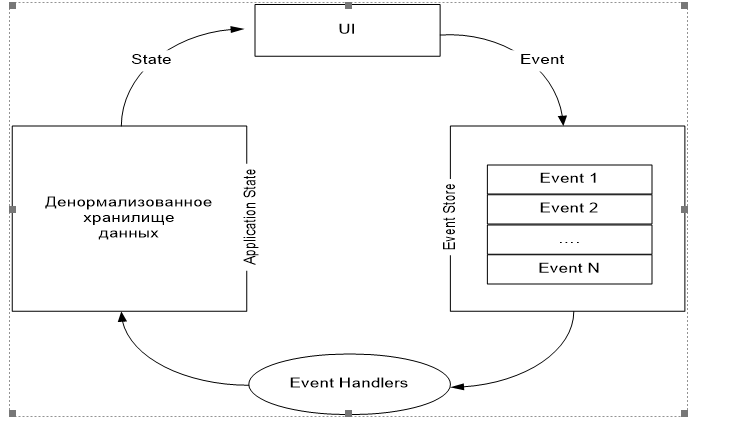
Nancy определенно создан для разработчиков MVC и может разрабатывать чрезвычайно быстрые веб-приложения. Nancy - это легкий фреймворк или микро-веб-фреймворк для создания сервисов на основе HTTP в .Net (также Mono). Здесь службы на основе HTTP означают, что эта структура может обрабатывать все стандартные методы HTTP, такие как GET, POST, PUT, DELETE, HEAD и т.д ... Все в Нэнси - это "HOST". Хост выступает в качестве платформы или адаптера для среды размещения и позволяет Nancy работать на существующих технологиях, таких как ASP.NET, WCF и т. Д.

1. **Event Storing:** назначение, принципы применения, примеры реализации.

# Event Storing рассказать что это такое, развитие подхода разделения ответственности, есть сервисы которые предназначены для выполнения изменений в бд , обычно используются event store, которые сохраняют не данные, а события которые возникают на стороне клиента и эти события сохраняются в специальных базах данных event storing, обычно это очень скоростные базы данных которые быстро позволяют перекачать эти данные в обычную бд, откуда эти данные используются на чтение

Когда мы используем бд в обыкновенном режиме у нас база данных как правило это некоторый слепок того состояния системы в котором она находится, вносим изменения в бд и она отражает в каждый момент времени данные о какой-то системе (слепок).

Выделяют такой подход Event Storing (он не новый) - заключается в том, что мы регистрируем не данные, а события, есть какая-то система и мы регистрируем не результат какой-то операции а какое-то событие. Сохраняем событие, что мы изменяем бд, регистрируем не само изменение а событие. БД которые ориентированы на регистрацию событий изменения бд называются Event Storing DB, бд для хранения событий. Каждое изменение в приложение данных генерирует событие а база данных регистрирует его, после того как событие зарегистрировано, возникает репликация, которая вносит изменение в бд, которая отражает текущее состояние бд.



2 базы данных, одно отражает текущее состояние системы, а другое регистрирует события изменения бд.

Рекомендуют NoSql бд только для чтения, а справа специальные бд, для хранения событий.

Есть такие базы данных

Событие представляет собой запись у которого есть имя, дата, информация об этом событии, такие записи организуются в виде очередей, есть всякие аналоги и прибамбасы, проекции (view - позволяет вычислять промежуточные результаты в самой бд для событий)

Можно написать триггеры - event hadlers, можно их там обрабатывать.

Что в подходе хорошего:

* храним историю событий системы
* Есть такие бд, в котором мы можем сделать select и указать время состояния. (Временные бд)

Минусы:

* большие ресурсы требуются

Есть база данных Event Store - есть для кора, через нугет мы её устанавливаем и запускаем, можно скачать и интерфейс для работы с бд, можем использовать интерфейс для создания событий, обработки событий, настройки обработчиков.

1. **Микросервисы:** микросервисная архитектура, определение микросервиса, основные принципы разработки микросервиса, паттерны разработки, DevOps для микросервисов.

# Микросервисы: один из подходов к разработке SOA-приложений, основной принцип – сервисы должны быть легкими. Легкий сервис – сервис, который может быть переписан за 2 недели.

Архитектурный стиль микросервисов — это подход, при котором единое приложение строится как набор небольших сервисов, каждый из которых работает в собственном процессе и коммуницирует с остальными используя легковесные механизмы, как правило HTTP. Эти сервисы построены вокруг бизнес-потребностей и развертываются независимо с использованием полностью автоматизированной среды. каждый сервис также получает четкую физическую границу, которая позволяет разным сервисам быть написанными на разных языках программирования.

**Микросервисы:** цель:

1) устойчивость к сбоям;

2) облегчить понимание и поддержку кода;

3) усилить работу команды программистов.

**Микросервисы:** микросервис – сервис, выполняющий одну элементарную функцию; основной принцип разбиения – изменение сервиса не затрагивает другие сервисы.

**Микросервисы:** микросервисная архитектура - набор принципов, которым должны соответствовать сервисы. Задается или в форме правил или основывается на применении готовых фреймворков (например, Karyon, Dropwiard,…).

**Микросервисы:** HTTP+HATEOAS, XML/JSON.

**Микросервисы:** распределенные процессы, монолит - несколько модулей в одном процессе; микросервисы – каждый модуль в отдельном удаленном процессе.

Принципы:

* отвечает за одну функциональную возможность (бизнес-возможность или техническая возможность);
* можно разворачивать по отдельности;
* состоит из одного или нескольких процессов (каждый микросервис работает в отдельных от других микросервисах процессах);
* имеет собственное хранилище данных;
* легко заменить;
* может просто поддерживаться

**Микросервисы:** популярные паттерны ***Tolerant Reader* (получает доступ только к тем данным которые будет использовать, без сериализации всего остального*)***, ***Consumer Driver* (вроде как суть в том что у нас появаляется дополнительный обьект, который преобразовывает контракты, так что у клиентов одного сервиса могут быть разные форматы контрактов, но сервис все равно все почитает*)***.

**Микросервисы:** проблема – транзакции. Часто без применения транзакций. Микросервисы в основном на чтение, а пишет отдельный сервис.

**Микросервисы:** клиенты работающие с микросервисом могут реализованы с помощью ***оркестрового*** (прямые вызовы сервисов) или ***хореографического*** (сервисы подписываются на события клиента) принципов;

***DevOps*** (Development & Operation)- набор технологий нацеленных на интеграцию процессов разработки и информационно-техническому обслуживанию. Цели DevOps: сокращение времени выхода продукта на рынок, снижение частоты отказов релизов, сокращение времени на изменения, сокращение времени на восстановление. Задача DevOps сделать согласованным процесс разработки и эксплуатации приложений. Основные задачи DevOps: контроль версий, непрерывная сборка, непрерывное тестирование, поддержка репозиториев артефактов, конфигурация инфраструктуры, мониторинг работоспособности и производительности.

**Микросервисы: *Continuous Delivery***, ***Continuous Integration – одновременно работают***.

**Непрерывная доставка** — это серия практик, направленных на то, чтобы обновления программного обеспечения происходили практически постоянно, без ожидания работы других специалистов. Разработчик, сделав какую-либо фичу, отправляет её *QA-инженерам* для тестирования и потом уже в релизный брэнч в системе контроля версий.

**Непрерывная интеграция** является ключевым компонентом практики *Agile Development*. Основой данной практики является постоянное попадание кода в центральный репозиторий после успешного запуска тестов. Таким образом все разработчики пилят свои фичи сразу в общий репозиторий и потом не приходится все сводить отдельно сделанное.

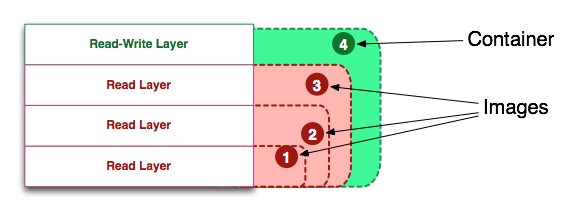
1. **Docker**: назначение, архитектура, основные команды

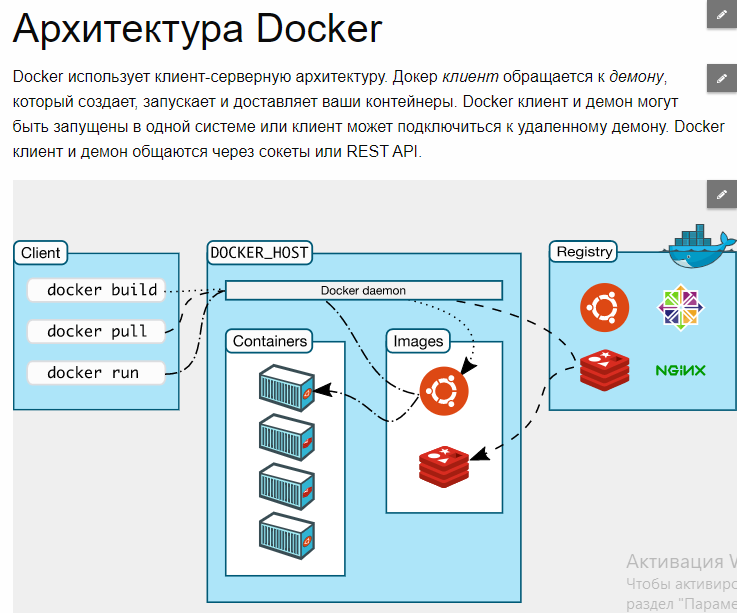
# Docker - платформа для разработки, доставки и эксплуатации приложений. Основное назначение – упростить развертывание приложения.

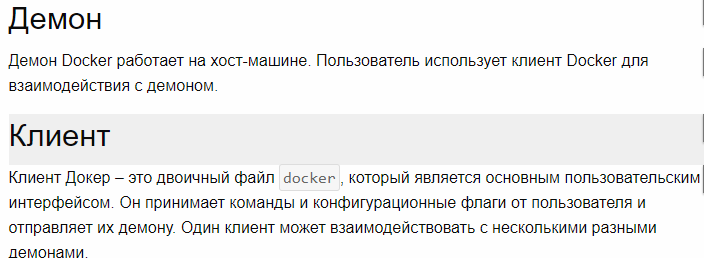
**Docker** = Docker Desktop(Engine) + Docker Hub

**Docker Hub** – облачный сервис для распространения **контейнеров**.

**Docker Engine** – механизм (сервис и приложение) для создания и функционирования **контейнеров**.







Команды:

* docker version – узнать версию докера
* docker run – запустить контейнер на основе указанного образа
* docker ps – список запущенных контейнеров
* docker inspect – вывод подробной информации о контейнере
* docker rm – удалить контейнер
* docker commit – Создать новый образ из изменений контейнера. Может быть полезно зафиксировать изменения или настройки файла контейнера в новом образе.
* docker build – Данная команда собирает образ Docker из файла докера (dockerfile) и контекста сборки. Контекст сборки — это набор файлов, расположенных по определенному пути. Для задания имени образа используйте параметр -t, например, «docker build -t my.».
* docker push – отправка образа в удалённый реестр.
* docker stop – Используется для «мягкой» остановки контейнера. Пример: docker stop my\_cont. Можно остановить не конкретный контейнер, а все запущенные — docker stop $(docker ps -a -q).
* docker pull – загрузка образа. Как правило, образы создаются на основе базового — из Docker Hub, где есть множество уже готовых образов и которые ты можешь использовать, а не тратить время на создание собственного. Для загрузки образа используется команда docker pull.
* docker logs – Позволяет просмотреть логи указанного контейнера. Можно использовать флаг -follow, чтобы следить за логами работающего контейнера, например, docker logs -follow my
* docker kill – «убивает» контейнер
* docker rmi – удаляет образ
* docker volume ls – показывает список томов, которые являются основным механизмом для хранения данных, генерируемых контейнерами Docker

1. **Docker-Compose**: назначение, основные команды.

Docker применяется для управления отдельными контейнерами (сервисами), из которых состоит приложение.

# **Docker Compose используется для одновременного управления несколькими контейнерами, входящими в состав приложения. Этот инструмент предлагает те же возможности, что и Docker, но позволяет работать с более сложными приложениями.**

Compose инструмент для создания и запуска многоконтейнерных Docker приложений. В Compose, вы используете специальный файл для конфигурирования ваших сервисов приложения. Затем, используется простая команда, для создания и запуска всех сервисов из конфигурационного файла.

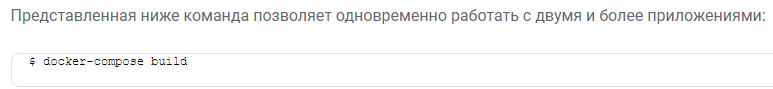
Compose превосходен для разработки, тестирования и настройки среды, а также непрерывной интеграции. Вы этом разделе вы можете узнать более подробно о [решаемых задачах](https://dker.ru/docs/docker-compose/overview-of-docker-compose/#common-use-cases).

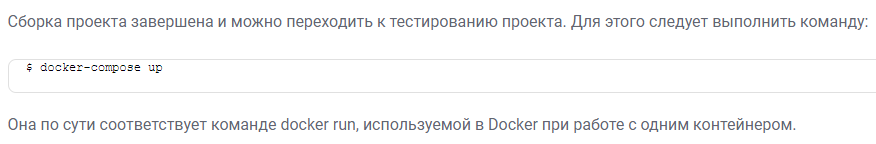
Использование Compose обычно разделяется на три этапа:

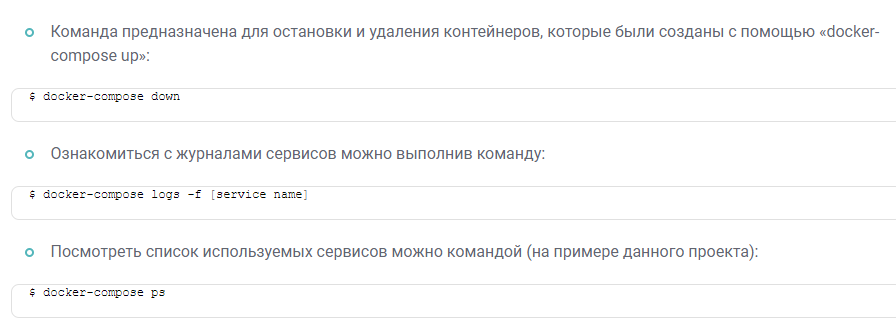
1. Определение окружения вашего приложения в Dockerfile, это можно сделать в любом месте.
2. Определение сервисов из которых будет состоять ваше приложение в docker-compose.yml, в последствии они смогут быть запущены все вместе в изолированном окружении.
3. И наконец, выполнение команды docker-compose up которая запустит все ваше приложение.

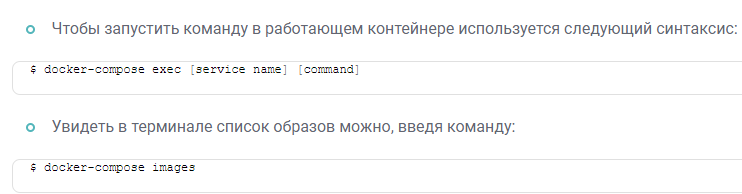
Compose имеет команды для управления всем жизненным циклом приложения:

* Запуск, остановка и пересоздание сервисов
* Просмотр статус запущенных сервисов
* Поток вывода журнала запущенных служб
* Выполнение одноразовых команд в сервисах









docker swarm, docker stack cuberneites

Kubernetes и Docker - два основных игрока в оркестровке контейнеров. Они создали себе достойные ниши и укрепили свои позиции в Docker и экосистеме контейнеров. Оба этих инструмента позволяют управлять кластером серверов, на которых запущена одна или несколько служб. Итак, прежде чем перейти к сравнительной части, давайте рассмотрим эти два инструмента.

## Kubernetes

Kubernetes - это платформа с открытым исходным кодом, созданная Google для операций развертывания контейнеров, увеличения и уменьшения масштабирования и автоматизации в кластерах хостов. Эта готовая к работе платформа корпоративного уровня с самовосстановлением (автоматическое масштабирование, автоматическая репликация, автоматический перезапуск, автоматическое размещение) является модульной, поэтому ее можно использовать для развертывания любой архитектуры.

Kubernetes также распределяет нагрузку между контейнерами. Он направлен на то, чтобы избавить инструменты и компоненты от проблемы, возникающей из-за запуска приложений в частных и публичных облаках, путем размещения контейнеров в группах и именования их как логических единиц. Их сила заключается в простом масштабировании, переносимости вне зависимости от среды и гибкости роста.

## Докер Рой

Как платформа, Docker произвел революцию в способах упаковки программного обеспечения. Docker Swarm или просто Swarm - это платформа оркестровки контейнеров с открытым исходным кодом, которая является собственным механизмом кластеризации для Docker и от него. Любое программное обеспечение, службы или инструменты, которые работают с контейнерами Docker, одинаково хорошо работают в Swarm. Кроме того, Swarm использует ту же командную строку из Docker.

Swarm превращает пул хостов Docker в единый виртуальный хост. Swarm особенно полезен для людей, которые пытаются освоиться с оркестрованной средой или которым необходимо придерживаться простой техники развертывания, но при этом иметь более одной облачной среды или одной конкретной платформы для ее запуска.

## Kubernetes против Docker Swarm

Хотя обе платформы оркестровки с открытым исходным кодом предоставляют в основном одни и те же функциональные возможности, между тем, как они работают, есть некоторые фундаментальные различия. Ниже приведены некоторые из примечательных моментов. В этом разделе сравниваются функции Docker Swarm и Kubernetes и слабые / сильные стороны выбора одной платформы над другой.

### Определение приложения

**Kubernetes:**  приложение может быть развернуто в Kubernetes с использованием комбинации сервисов (или микросервисов), развертываний и модулей.

**Docker Swarm:**  приложения могут быть развернуты как микросервисы или сервисы в кластере Swarm в Docker Swarm. Файлы YAML (YAML Ain't Markup Language) могут использоваться для идентификации нескольких контейнеров. Более того, Docker compose может установить приложение.

### Сети

**Kubernetes:**  сетевая модель - это плоская сеть, позволяющая всем модулям взаимодействовать друг с другом. Сетевая политика определяет, как модули взаимодействуют друг с другом. Плоская сеть обычно реализуется в виде наложения. Для модели требуются два CIDR: один для сервисов, а другой, из которого поды получают IP-адрес.

**Docker Swarm:** узел, присоединяющийся к кластеру swarm, создает оверлейную сеть для сервисов, которые охватывают каждый хост в рою докеров, и сеть моста докеров только для хоста для контейнеров. Пользователи могут самостоятельно шифровать трафик данных контейнера при создании оверлейной сети в Docker Swarm.

### Масштабируемость

**Kubernetes:**  для распределенных систем Kubernetes - это скорее универсальный фреймворк. Это сложная система, поскольку она обеспечивает надежные гарантии состояния кластера и унифицированный набор API. Это замедляет масштабирование и развертывание контейнера.

**Docker Swarm:**  Docker Swarm, по сравнению с Kubernetes, может развертывать контейнер намного быстрее, и это позволяет быстрее масштабировать время реакции по запросу.

### Высокая доступность

**Kubernetes:** все поды в кубернетах распределены между узлами, и это обеспечивает высокую доступность, выдерживая сбой приложения. Сервисы балансировки нагрузки в Kubernetes обнаруживают нездоровые поды и избавляются от них. Итак, это поддерживает высокую доступность.

**Docker Swarm:**  поскольку сервисы могут быть реплицированы в узлах Swarm, Docker Swarm также обеспечивает высокую доступность. Узлы диспетчера Swarm в Docker Swarm отвечают за весь кластер и обрабатывают ресурсы рабочих узлов.

### Настройка контейнера

**Kubernetes:**  Kubernetes использует свои собственные определения YAML, API и клиента, и каждое из них отличается от стандартных эквивалентов докеров. То есть вы не можете использовать Docker Compose или Docker CLI для определения контейнеров. При переключении платформ определения и команды YAML необходимо переписать.

**Docker Swarm:**  API Docker Swarm не полностью охватывает все команды Docker, но предлагает большую часть знакомых функций из Docker. Он поддерживает большинство инструментов, работающих с Docker. Тем не менее, если в Docker API отсутствует какая-либо конкретная операция, не существует простого способа обойти это с помощью Swarm.

### Балансировки нагрузки

**Kubernetes: модули** доступны через службу, которая может использоваться в качестве балансировщика нагрузки в кластере. Как правило, для балансировки нагрузки используется входящий трафик.

**Docker Swarm:** режим Swarm состоит из элемента DNS, который можно использовать для распределения входящих запросов на имя службы. Службы могут быть назначены автоматически или могут работать на портах, указанных пользователем.

## Заключение

Kubernetes поддерживает более высокие требования с большей сложностью, в то время как Docker Swarm предлагает простое решение, с которым можно быстро начать работу. Docker Swarm пользуется большой популярностью среди разработчиков, предпочитающих быстрое развертывание и простоту. Одновременно Kubernetes используется в производственной среде различными известными интернет-компаниями, работающими с популярными сервисами.

И Kubernetes, и Docker Swarm могут запускать многие из одних и тех же сервисов, но могут потребоваться несколько разные подходы к определенным деталям. Итак, [изучив Kubernetes](https://mindmajix.com/kubernetes-training) и Docker и сравнив их по различным функциям, вы можете принять решение о выборе правильного инструмента для оркестрации вашего контейнера.