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

**SOA: Service-oriented architecture** – парадигма разработки программного обеспечения, основанная на применении распределенных слабосвязанных компонентов, обеспечивающих стандартные интерфейсы. Компоненты распределенной системы SOA – узлы – сервисы.

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

**SOA: основные свойства**

* независимость от аппаратной реализации узлов;
* независимость от ОС в узлах;
* независимость от языка программирования разработки сервиса;
* масштабируемость.

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

**SOA:** 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-MakeConnection (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).

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

* SOA Registry;
* Workflow Engine;
* Service Broker;
* SOA Supervisor.

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

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



**Workflow Engine –** программный компонент, предназначенный для построения модели бизнес процесса и выполнения бизнес процесса на основе группы сервисов. Другими словами, разработать новый сервис на основе последовательного выполнения нескольких сервисов.



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

Сервисным брокером является служба, соединяющая различные сервисы вместе. Он получает всю необходимую информацию от СОА реестра (SOA Registry), что означает, что реестр и брокер должны работать координировано.



**SOA Supervisor** – служебный сервис, предназначенный для управления и мониторинга других служебных сервисов.

**SOA:** интерфейсы – REST, SOAP, JSON-RPC

**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.







**SOA:** Software AG webMethods OneData, Oracle SOA Suite 12C, Microsoft WCF.

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

**REST**: Архитектурный стиль взаимодействия компонентов распределенного приложения.

**REST:** альтернатива RPC.

**REST**: форматы передачи данных – xml и json

**REST**: шесть обязательных ограничений:

* модель клиент-сервер;
* отсутствие состояния на стороне сервера, допускается сохранение состояния на клиенте или в другом сервисе (например, в БД);
* кэширование на стороне клиента, сервер явно управляет кэшированием;
* единообразие интерфейсов (идентификация ресурсов, манипуляция ресурсами через представления, самодостаточные сообщения, HATEOAS);
* код по требованию: допускается (необязательно) выгрузка на клиент апплетов или сценариев для расширения его функциональности.

**HATEOAS:** гипермедиа в качестве управления состоянием.

**Гипермедиа**: технология обработки, структурирования информации и произвольного доступа к ее элементам с помощью гиперсвязей.

**REST:** **общепринятые правила**

* Общий префикс для всех ресурсов сервиса
* Два типа ресурсов: коллекция, элемент коллекции
* Иерархическая связь
* Ограничить количество HTTP-статусов
* Версионность
* Постраничное получение данных
* Сортировка: параметр sort
* Все фильтры вынести за знак вопроса
* Обозначать в запросе формат сообщений (желательна поддержка нескольких форматов)

Платформы для разработки сервисов:

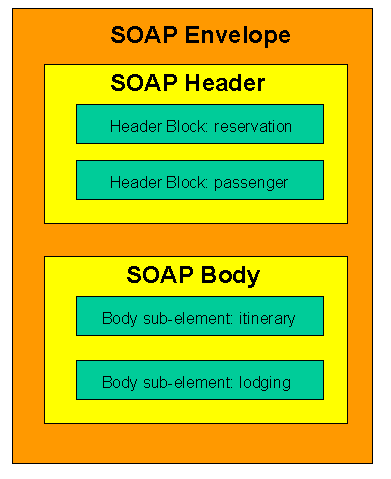
* на базе ASP.NET HTTP Handler
* на базе ASP.NET MVC WEB API

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

**SOAP:** протокол обмена структурированными сообщениями в распределённой вычислительной среде.

* Simple Object Access Protocol — простой протокол доступа к объектам;
* основывается на XML (предназначен для пересылки XML-сообщений);
* транспорт: HTTP, SMTP, FTP;
* не зависит о платформы.

**SOAP:** формат сообщений:



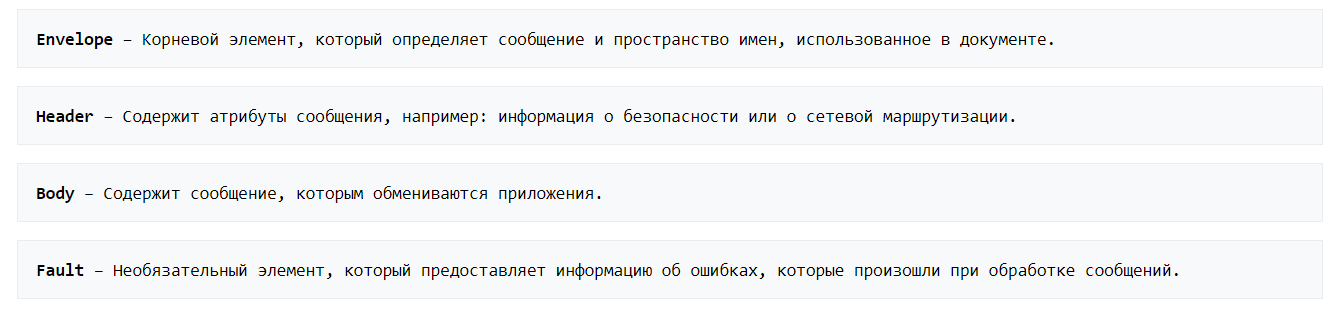
**SOAP:** стандартные пространства имен: env, enc, rpc, rep, xop, xmime

**SOAP:** узел – компонент SOA-архитектуры, который может отправлять или получать SOAP-сообщения.

**SOAP:** роль - набор правил, определяющих поведение узла, задается атрибутом env:role.

**SOAP:** тристандартные роли для узла (нет роли, промежуточный узел, конечный получатель), роль может быть пользовательской (задаваться собственным URI)

**SOAP:** структура



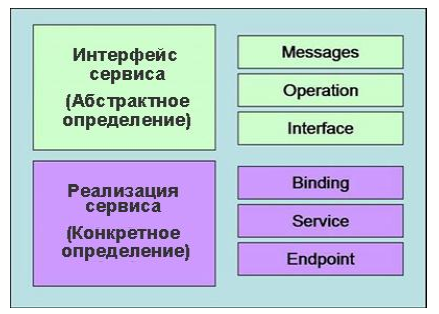
# **WSDL:** определение, стандарты и версии, концептуальная модель, пространства имен, структура WSDL-документа.

**WSDL:** средство XML-описания интерфейса для доступа к web-службе; платформонезависимое описание.

**WSDL** (Web Services Description Language) — язык описания веб-сервисов и доступа к ним, основанный на языке XML

**Стандарты и версии:** W3C *wsdl 1.1*, текущая версия 2.0

**WSDL: концептуальная модель WSDL 2.0**

****

**WSDL:** пространства имен: wsdl, wsdi, wsdx, wrpc, wsoap, whttp, xs, xsi

<!-- структура wsdl-документа -->

<definition>

<types>

<!-- оперделение типов, используемых web-сервисом -->

<!-- для пл.-независимого WSDL используется синтаксис XML Schema -->

</types>

<message>

<!-- сообщения, используемые web-сервисом -->

<!-- сообщений, может быть несколько -->

<!-- каждое сообщение может состоять из нескольких частей -->

</message>

<portType>

<!-- методы, предоставляемые web-сервисом -->

<!-- может быть несколько портов -->

<!-- определены операции web-сервиса и используемые сообщения -->

</portType>

<binding>

<!-- протоколы связи, используемые web-сервисом -->

<!-- форматы сообщений и детали протокола для каждого порта -->

</binding>

<service>

<!— набор портов связанных с сервисом -->

</service>

</definition>

# **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;

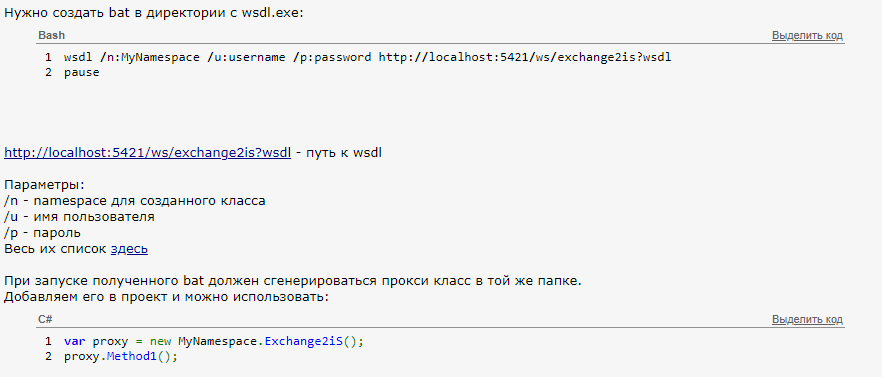
* В качестве хоста только IIS
* Модель взаимодействия только полудуплекс
* Имеет только 1 конечную точку
* Транспорт только HTTP

**Веб-сервис** — это веб-приложение, предоставляющее открытый (значит опубликованный) интерфейс, пригодный для использования другими приложениями и интерфейсами.

**Разработка ASMX сервиса:**

1. В Visual Studio добавить к проекту ASMX-службу.
2. В WebService(): Добавить пространство имен(Namespace) и описание(Description).
3. В WebMethod(): описание методов(Description), описание возвращаемого результата, добавить имя сообщения в WSDL(MessageName).
4. Добавить proxy-класс на клиент.

**Утилита WSDL.exe:**



# **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.

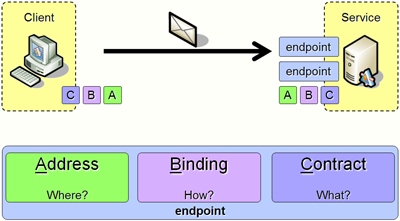
**WCF:** Windows Communication Foundation; технология основанная на .NET FRAMEWORK для разработки приложений SOA-архитектуры.

**Веб-сервис** — это веб-приложение, предоставляющее открытый (значит опубликованный) интерфейс, пригодный для использования другими приложениями и интерфейсами.

**WCF:** основные принципы

* разработка сервиса должна быть простой и иметь способность к расширению;
* один API для всех коммуникационных протоколов;
* сервис должен функционировать по отрытым телекоммуникационным стандартам;
* сервис должен поддерживать стандарты WS-\*;
* сервис должен поддерживать REST, RPC и др. архитектуры;

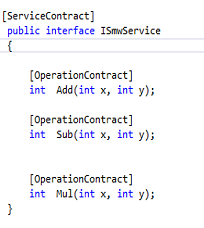
**WCF:** коммуникация модель



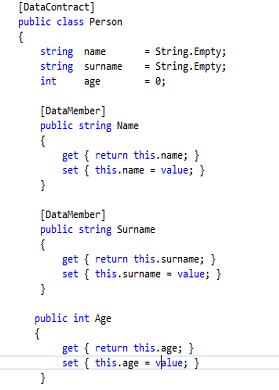
**WCF:** контракты

* контракт службы;
* контракт данных;
* контракт сообщений.

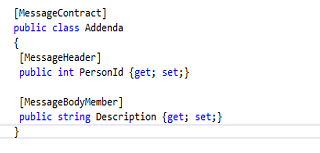
**WCF:** контракт службы

****

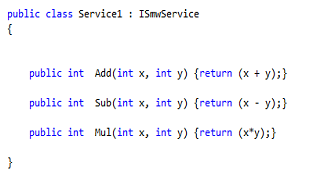
**WCF:** контракт данных – указывает каким образом данные будут сериализованы.



**WCF:** контракт сообщений

****

**WCF:** реализация сервиса

****

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

**WCF:** конечная точка – адрес, привязка, контракт.

**WCF:** привязка – инкапсулирует все технологии, позволяющие переправить сообщение от одной конечной точке к другой.



**WCF:** привязка: безопасность:

* WS-ReliableMessaging
* WS-Security

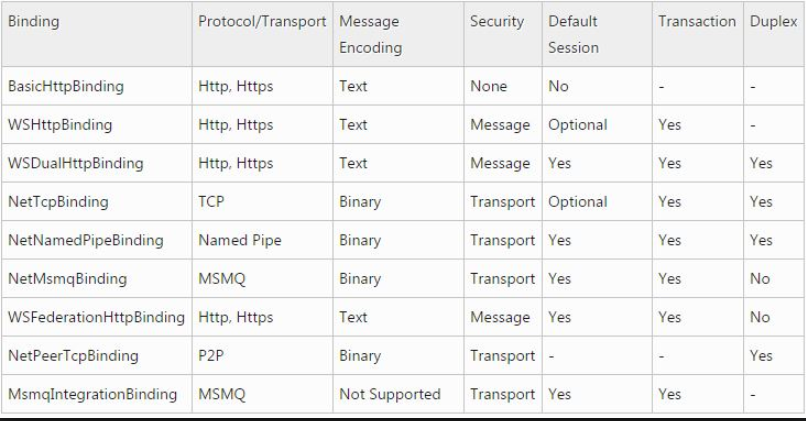
**WCF:** привязка: кодирование:

* текст (ASCII, UTF-8, UTF-16);
* двоичная (проприетарные алгоритмы);
* МТОМ.

**WCF:** привязка: транспорт:

* HTTP;
* TCP;
* IPC Named Pipes;
* MSMQ;
* Custom.

**WCF:** привязка: стандартные:

****

**Отличия asmx-сервиса и wcf-сервиса:**

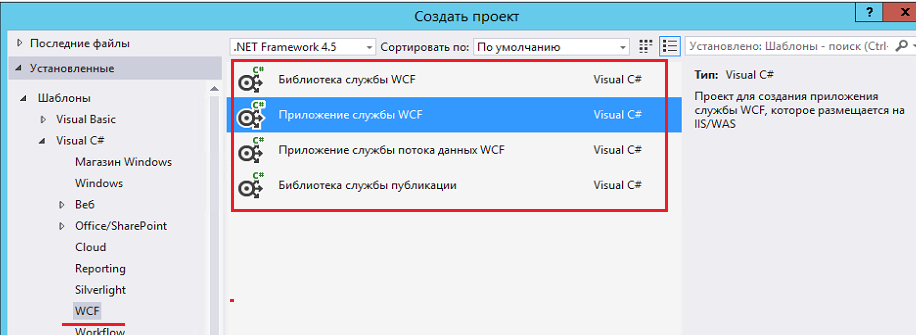
1. asmx-сервис работает только по протоколу http в качестве транспорта, wcf может работать как по http, так и по tcp, named pipe, msmq.

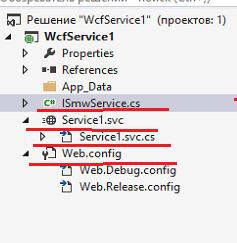
2. asmx-сервис можно размещать только на iis-сервере, у wcf в качестве хоста может выступать любое консольное или графическое .NET-приложение, Windows-служба или IIS

3. asmx-сервис может использовать только одну конечную точку, wcf может использовать по одной конечной точке на каждый используемый протокол передачи данных.

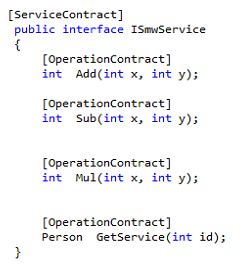
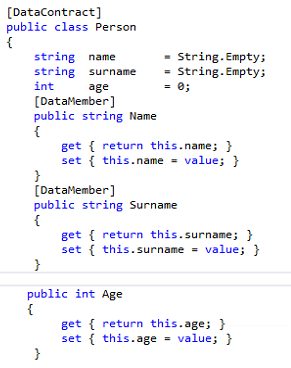
4. asmx-сервис работает только по полудуплексной модели взаимодействия, wcf – по полудуплексной, дуплексной, однонаправленной, потоковой и издатель-подписчик.

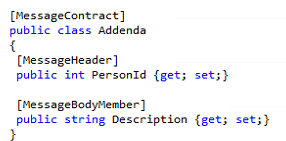
**Разработка WCF RPC:**

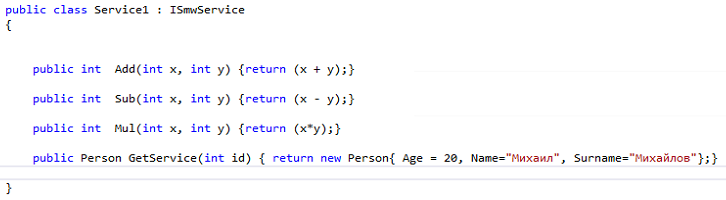
****

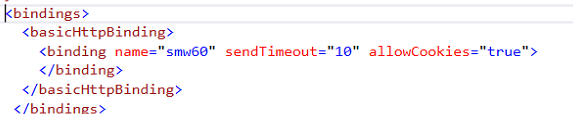
****

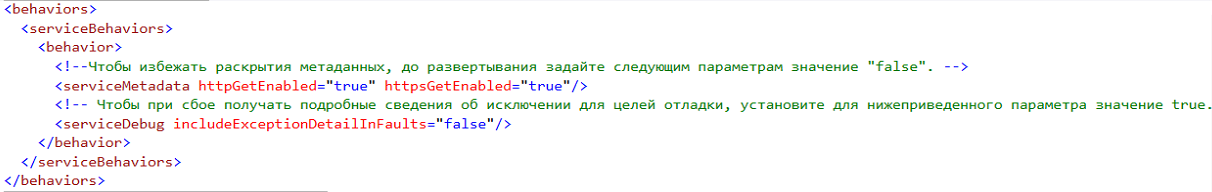


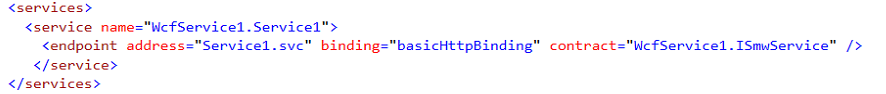
** **

****

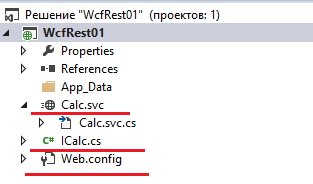
****

****

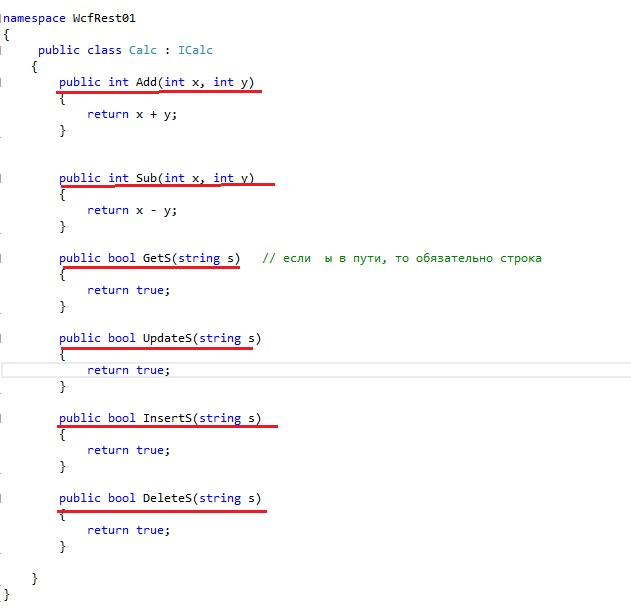
****

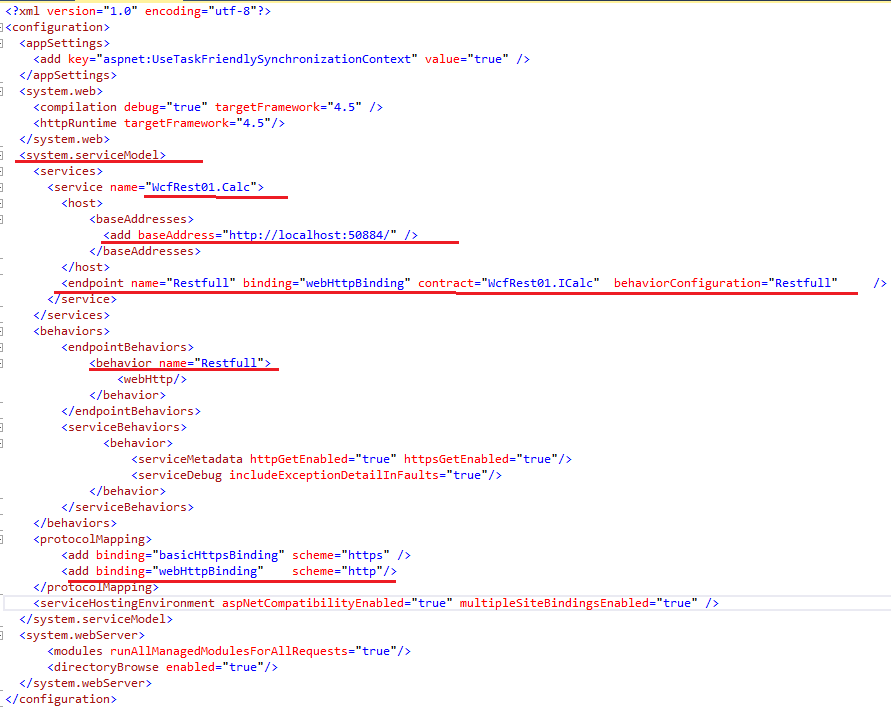
****

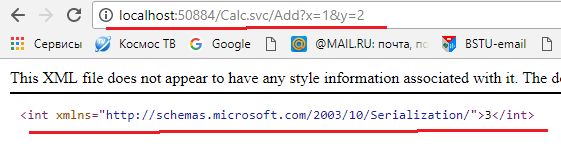
**WCF/REST:** разработка WCF-службы с REST-интерфейсом





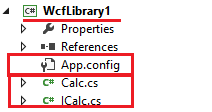




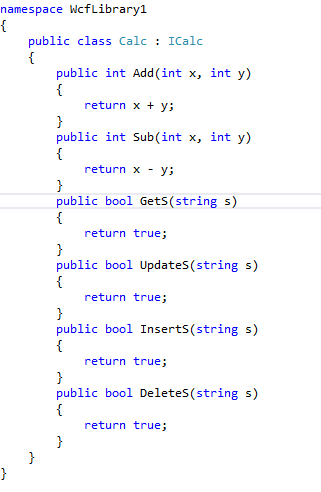


**WCF-сервис с несколькими конечными точками**

**WCF:** создать приложение ***библиотека службы WCF***

****

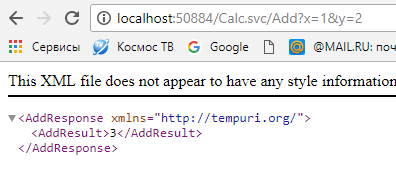
**WCF:** контракт и реализация сервиса

****

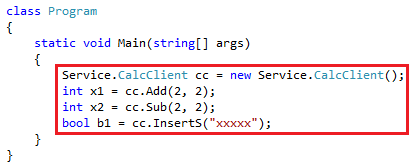
**WCF:** конфигурационный файл сервера

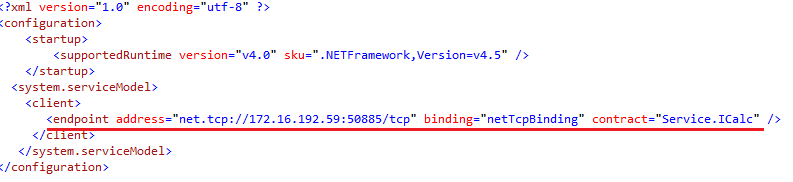


**WCF:** REST-клиент

****

**WCF:** WCF/TCP-клиент

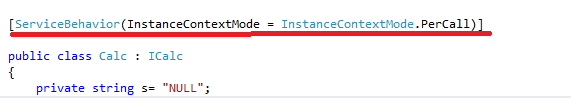
****

****

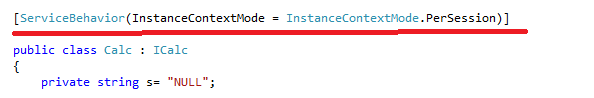
**Поведение WCF-сервиса**

**WCF: Service Behavior/Instance Context Mode –** определяет когда объект сервиса будет создан

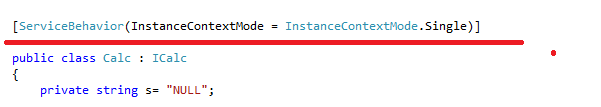
**WCF: Per Call**

****

**WCF: Per Session**

****

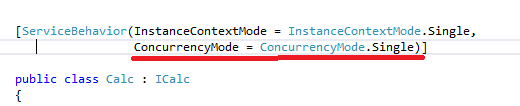
**WCF: Single**

****

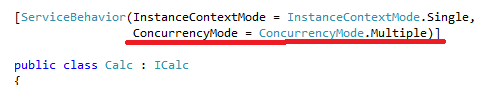
**WCF:** поведение сервиса Per Session необходимо сопоставлять с binding

**WCF: Service Behavior/Concurrency Mode -** получает или задает, поддерживает ли служба один поток, несколько потоков или повторные входящие вызовы.

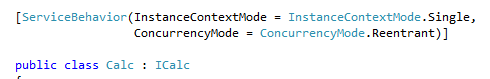
**WCF: Single**

****

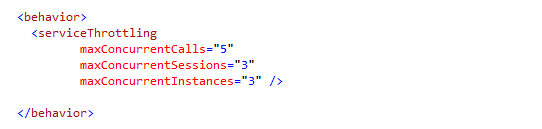
**WCF: Multiple**

****

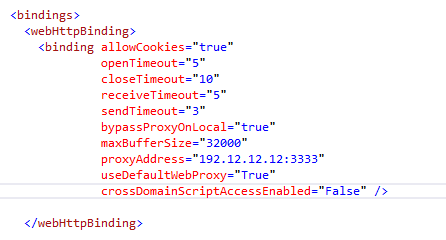
**WCF: Reentrant**

****

**WCF: Service Behavior/serviceThrottling -** Определяет механизм регулирования службы Windows Communication Foundation (WCF)

****

**WCF: Service Behavior/ binding**

****

**Безопасность WCF-сервиса**

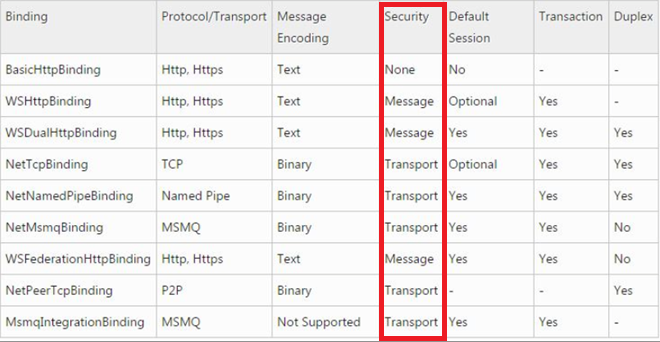
**WCF:** Аутентификация, авторизация, целостность сообщений, конфиденциальность сообщений

**WCF:** Безопасность транспорта (TLS/SSL)

**WCF:** Безопасность сообщений, спецификация WS-Security (цифровая подпись и шифрование фрагментов конверта)

**WCF:** Гибридная модель безопасности.

**WCF: binding/Security**

****

****

**WCF: message-аутентификация:** None**,** Windows (NTLM, Kerberos), Username(имя/пароль), Certificate (X509), IssueToken (служба токенов третьей стороны).

**WCF: transport-аутентификация:** None, Windows (NTLM, Kerberos), Certificate (X509),Basic (протокол Http Basic Authentication RFC 7615), Digest (Http Digest Access Authentication RFC 7616),NTLM (NT LAN Manager, Microsoft Windows NT)

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

**RSS:** семейство XML-форматов, предназначенного для описания новостных лент, анонсов и статей (199).

**RSS:** MIME: application/rss+xml.

**RSS:** пример

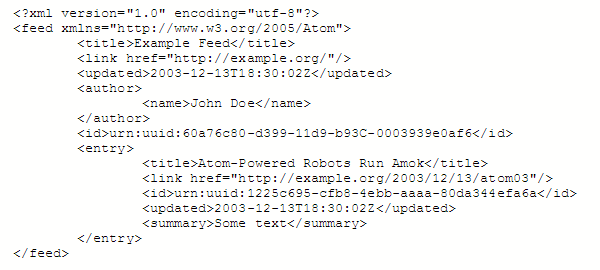
****

**ATOM**: формат описания web-ресурсов и протокола для их публикации (2005г.).

**ATOM**: последняя версия 1.0

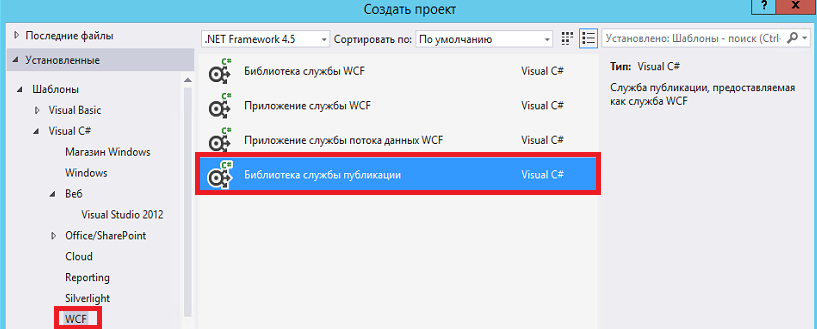
**ATOM**: MIME: application/atom+xml.

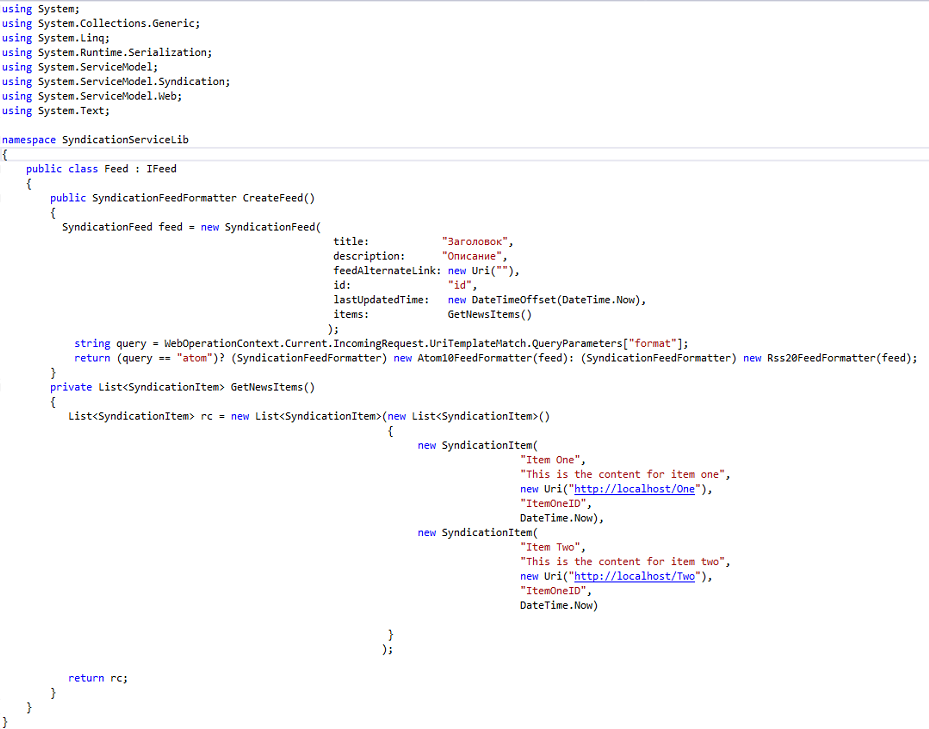
**ATOM**: примеры

****

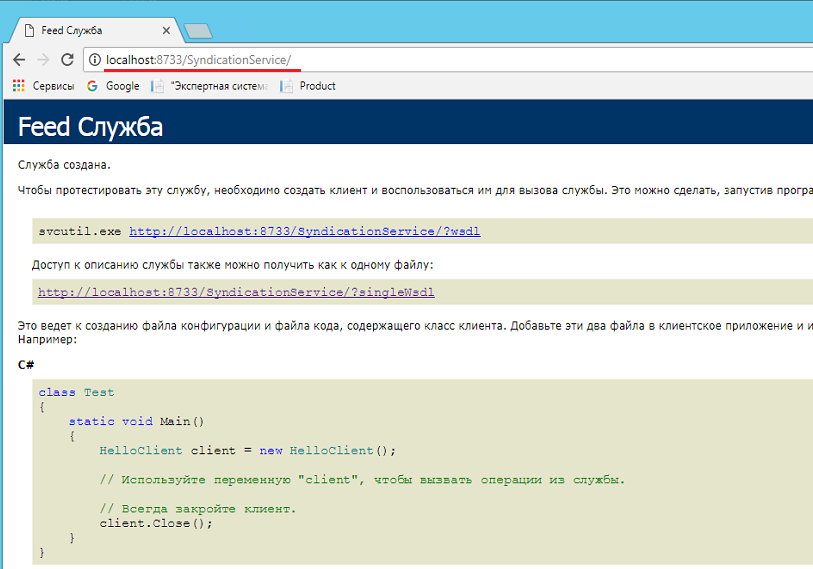
**ATOM/RSS:** ATOM более универсальный и чаще применяется; следует использовать его при разработке новых приложений.

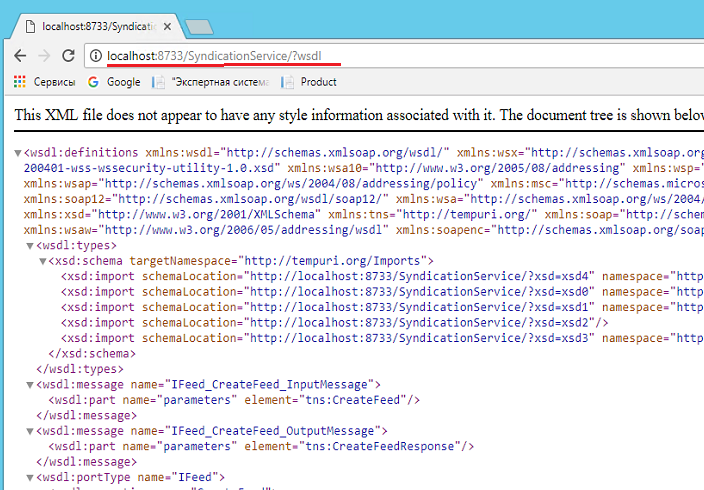
**ATOM/RSS:** реализация на платформе WCF .NET

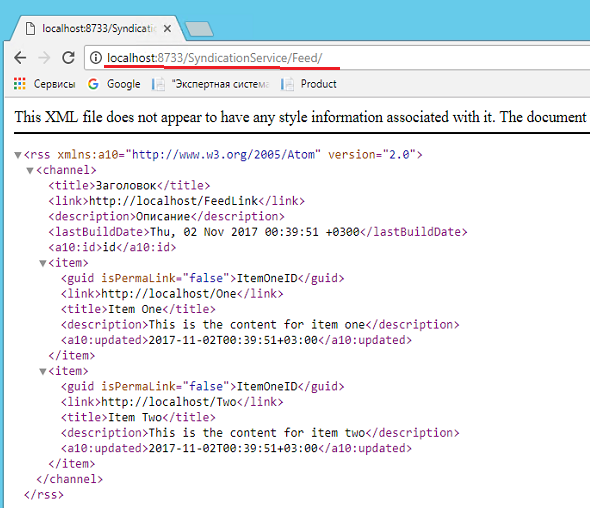
****

****

****

****

****

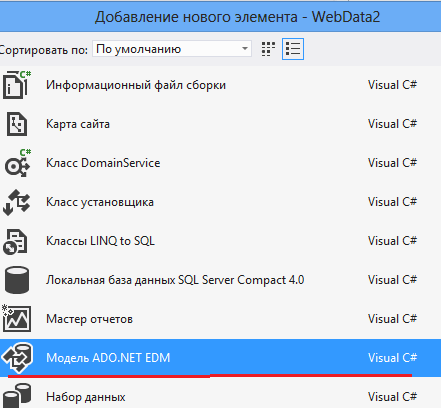
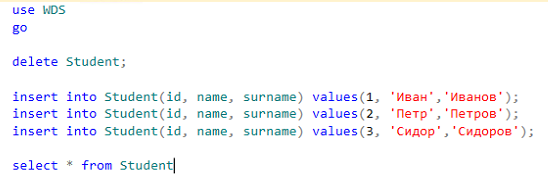
****

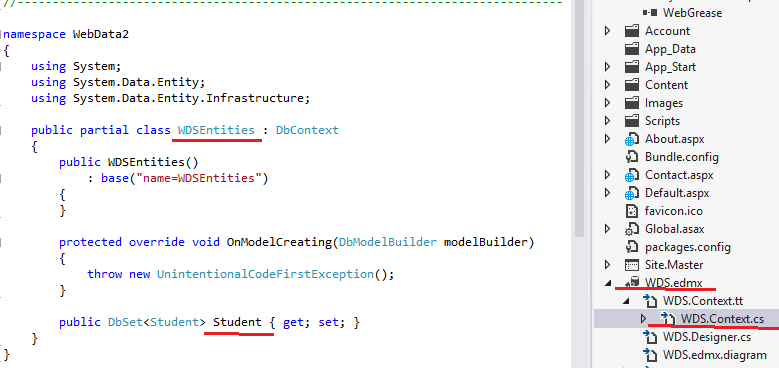
# **WCF Data Services:** протокол Open Data Protocol, возможности предоставляемые OData-интерфейсом, порядок разработки Data Services, применение Data Services.

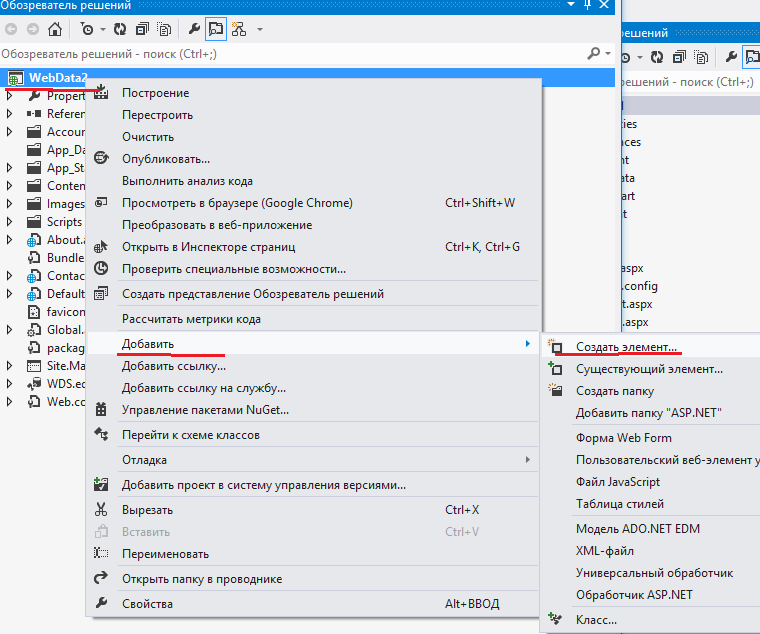
**OData: Open Data Protocol –** открытый web-протокол; позволяет выполнять операции с ресурсами и получать ответы в форматах XML, JSON.

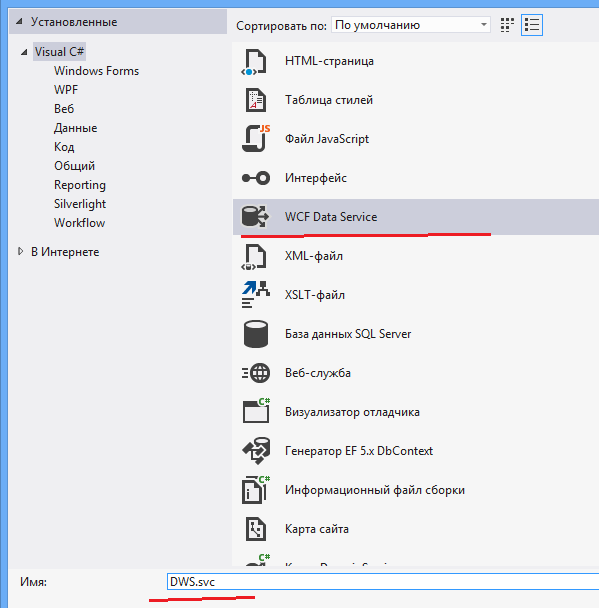
**OData:** поддерживается OASIS.

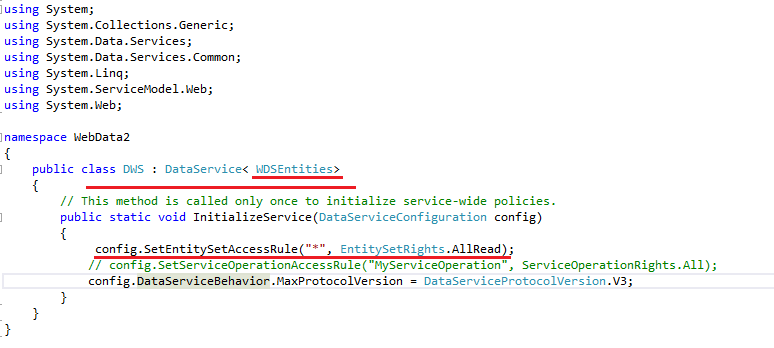
**WCF DS: ASP.NET**

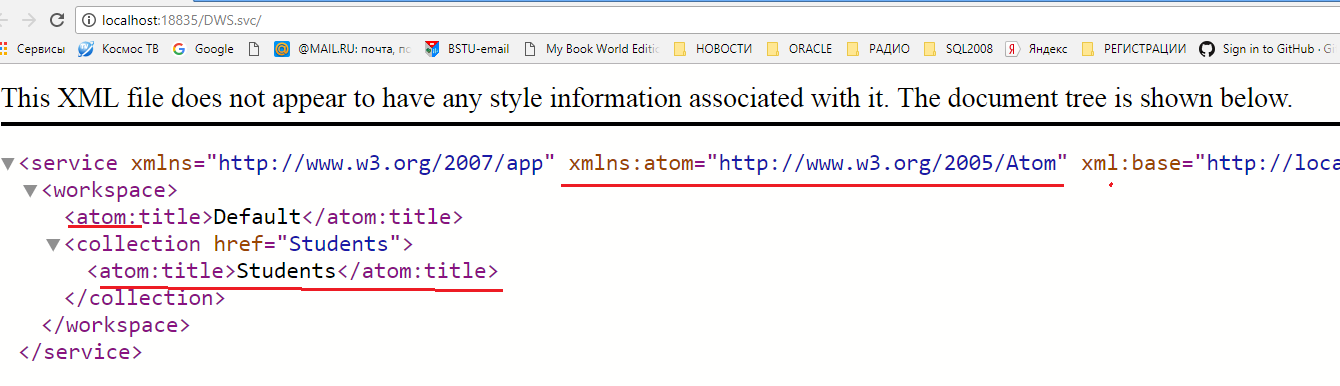
****

****

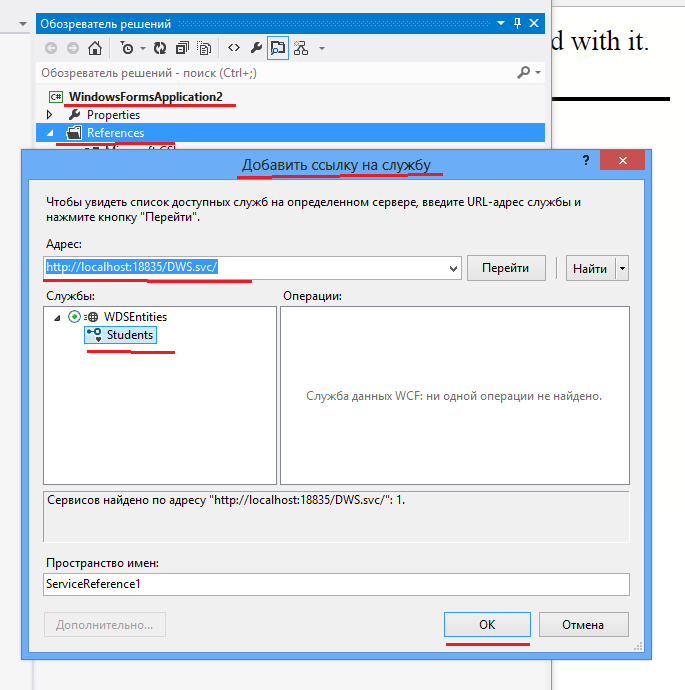
****

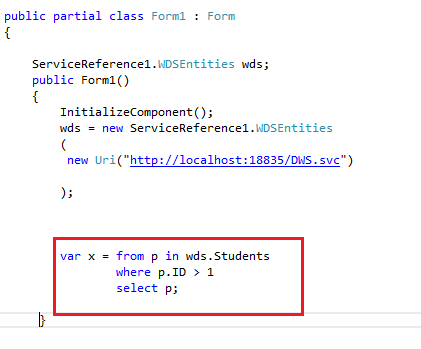
****

****

****

**WCF DS: клиент**

****

****

# **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.

**JSON-RPC:** последняя версия 2.0.

**JSON-RPC:** объект запроса

{“jsonrpc”, “method”, “params”, “id”}

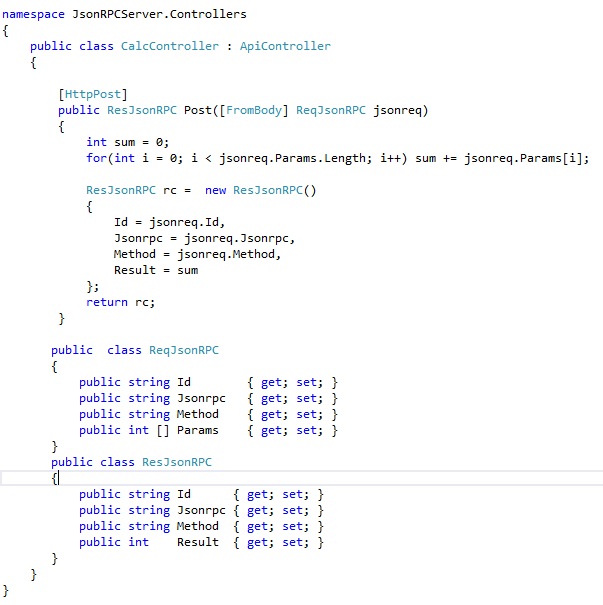
**JSON-RPC:** объект ответа

{“jsonrpc”, “result”/“error”, “id”}

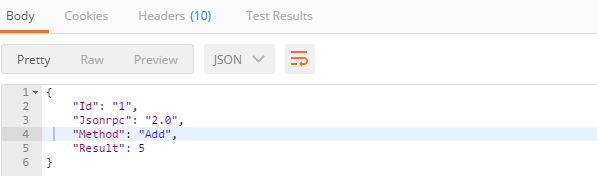
**JSON-RPC:** объект(ошибка)

{ “message”, “data”, “code”}

**JSON-RPC:** начало примера

****

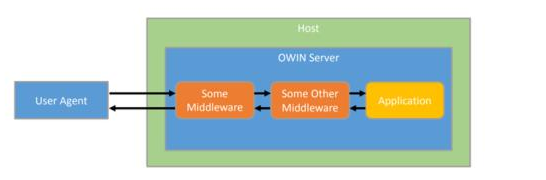
****

****

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

**Nancy:** Framework для разработки self-hosting приложений в .NET и Core.

**OWIN** – интерфейс между net-web-сервером и серверным приложением (обработчиками запросов). Основная цель OWIN отделить web-сервер от серверного приложения. Можно разрабатывать отдельно сервер и приложение.



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

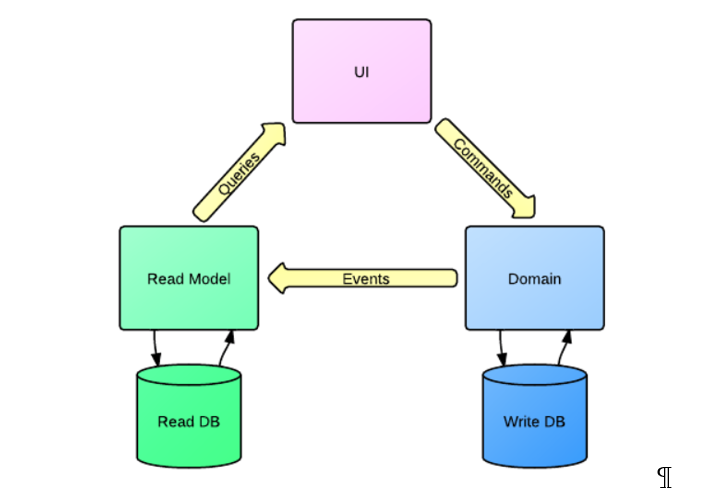
**OWIN Server** - http-сервер, реализующий интерфейс OWIN.

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

**Middleware** – подключенные компоненты (модули), предназначенные для обработки запросов

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

*(Сказать о новом подходе который существует сейчас связанном с микросервисами каждый сервис должен быть изолированный и иметь свою. Сервисы которые заполняют бд и которые читают)*



DDD: **Domain-Driven Design** - набор принципов направленных на создание систем объектов со сложной предметной областью.

цель – ускорить процесс проектирование программного обеспечения.

DDD: система имеет 4 уровня:

1. interface (пользовательское взаимодействие, взаимодействие с другими системами)
2. application (реализация, технические аспекты приложения, хранение данных, создание данных)
3. business (реализация доменного языка)
4. infrastructure (платформа и дополнительное программное обеспечение).

**Command-Query responsibility segregation(CQRS)**

CQRS: разделение ответственности на команды и запросы;

обычно 2 базы данных: для записи и для чтения. Базы данных синхронизируются.

**Event Sourcing (ES)**

ES: подход к хранению данных, при котором вместо конечного результата хранится череда событий произошедших с некоторой сущностью.

ES: каждое событие имеет имя.

ES: произошедшие события неизменны.

ES: проекция - вычисленные результаты для UI (аналог view).

ES: ориентация на процесс (workflow driven), а не на данные (data-input based).

ES: обычно применяется в архитектуре CQRS для реализации WriteDB.

EventStore: функциональная база данных с комплексной обработкой событий в JavaScript.

# Микросервисы:микросервисная архитектура, определение микросервиса, основные принципы разработки микросервиса, паттерны разработки, DevOps для микросервисов, оркестровка и хореография микросервисов.

**Микросервисы:** один из подходов к разработке SOA-приложений.

**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,…).

*Основные принципы разработки микросервиса:*

* Легкость сервисов (сервисы маленькие и протоколы легковесные). Легкий сервис – сервис, который может быть переписан за 2 недели (Real Estate).
* Принцип разбиения – изменение сервиса не затрагивает другие сервисы.
* Правильное распределение функциональности
* ESB – не является приемлемым решением. Обычно применяется HTTP/REST, HTTP+HATEOAS, XML/JSON.
* Обеспечение устойчивости к сбоям, большего понимания кода, облегчения работы программиста.
* Распределенные процессы - каждый модуль в отдельном удаленном процессе.
* Применения микросервисной архитектуры.
* Не применяются синхронные вызовы.
* Распределенное управление данными.

*Популярные паттерны разработки:*

* Tolerant Reader - приложение будет получать информацию и обрабатывать только то, что будет использоваться, независимо от неожиданной дополнительной информации.
* Consumer Driver - позволяет предупредить коллизии и улучшить понимание при взаимодействии двух связанных компонентов(сервисов).

*DevOps для микросервисов:*

Требуется DevOps (Development & Operation)- набор технологий, нацеленных на интеграцию процессов разработки и информационно-техническое обслуживание.

*Цели DevOps*:

* сокращение 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.

 DevOps – следствие увеличения релизов (Agile-технологии) и усложнения инфраструктуры.

Автоматизация, продукты DevOps: Continuous Delivery, Continuous Integration.

*Оркестровка и хореография микросервисов:*

клиенты работающие с микросервисом могут быть реализованы с помощью оркестрового (прямые вызовы сервисов) или хореографического (сервисы подписываются на события клиента) принципов.

# **Docker**: назначение, архитектура, основные команды

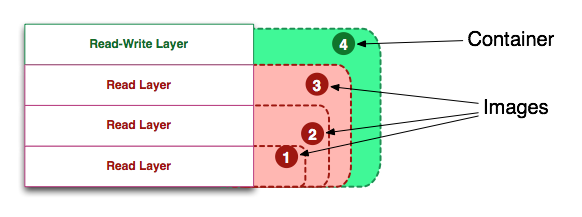
<https://dker.ru/docs/docker-engine/docker-overview/>

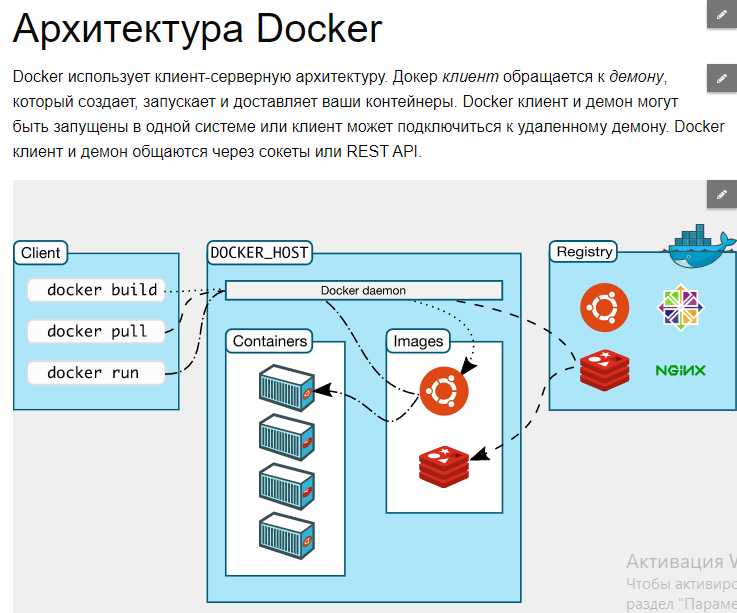
Docker - платформа для разработки, доставки и эксплуатации приложений. Основное назначение – упростить развертывание приложения.

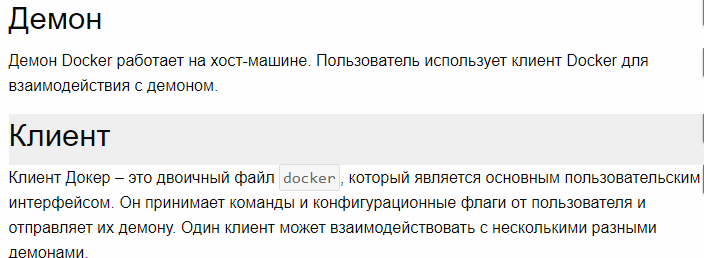
**Docker** = Docker Desktop(Engine) + Docker Hub

**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

# **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:

Определение окружения вашего приложения в Dockerfile, это можно сделать в любом месте.

Определение сервисов из которых будет состоять ваше приложение в docker-compose.yml, в последствии они смогут быть запущены все вместе в изолированном окружении.

И наконец, выполнение команды docker-compose up которая запустит все ваше приложение.

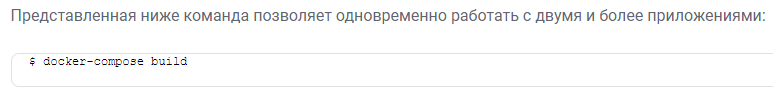
Compose имеет команды для управления всем жизненным циклом приложения:

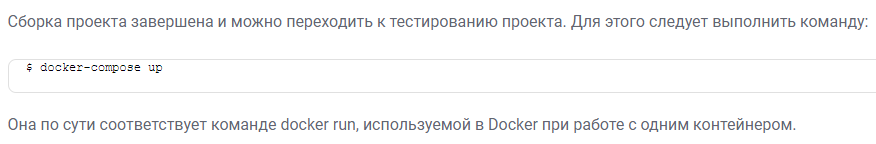
Запуск, остановка и пересоздание сервисов

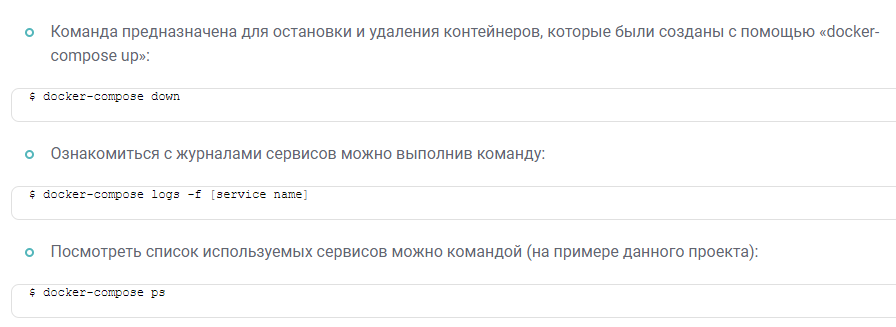
Просмотр статус запущенных сервисов

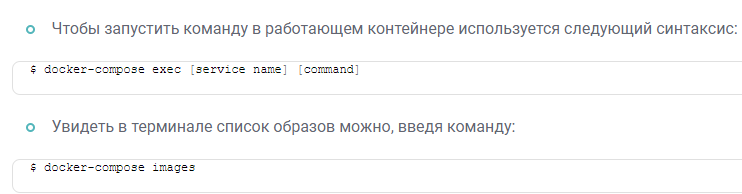
Поток вывода журнала запущенных служб

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.