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

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

**Основные свойства**

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

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

**свойства сервиса**

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

**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).

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

Основной принцип сервисной шины — концентрация **обмена сообщениями** между различными системами **через единую точку**, **в которой** обеспечивается **транзакционный контроль**, **преобразование данных**, **сохранность сообщений**.

Все **настройки** обработки и передачи **сообщений** предполагаются также сконцентрированными **в единой точке**, и формируются в терминах служб, таким образом, **при замене** какой-либо информационной системы, подключённой к шине, **нет** необходимости в **перенастройке** остальных систем.

**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.

**СОА реестр (SOA Registry):** реестр сервисов, информация о сервисах и их интерфейсах.

СОА реестр это своего рода **электронный каталог**, где хранится **информация** о каждом **компоненте**, составляющем корпоративную **информационную систему**, и об **интерфейсах**, которые эти компоненты **используют** для обеспечения **связи** между собой.

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

**Cервисный брокер (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.

**СОА супервизор (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.

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

Обратный вызов – функция, которая вызывается позже.

Способы клиент серверного взаимодействия:











Основные стандарты W3С: **XML, SOAP, WSDL, UDDI.**

*World Wide Web Consortium,* ***W3C*** — организация, разрабатывающая и внедряющая технологические стандарты для Всемирной паутины.

**XML** — расширяемый язык разметки. Рекомендован Консорциумом Всемирной паутины (W3C).

**SOAP** (от англ. Simple Object Access Protocol — простой протокол доступа к объектам) — протокол обмена структурированными сообщениями в распределённой вычислительной среде. Первоначально SOAP предназначался в основном для реализации удалённого вызова процедур (RPC). Сейчас протокол используется для обмена произвольными сообщениями в формате XML, а не только для вызова процедур.

SOAP может использоваться с любым протоколом прикладного уровня: SMTP, FTP, HTTP, HTTPS и др. Однако его взаимодействие с каждым из этих протоколов имеет свои особенности, которые должны быть определены отдельно. Чаще всего SOAP используется поверх HTTP.

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

**UDDI** (англ. Universal Description Discovery & Integration) — инструмент для расположения описаний веб-сервисов (WSDL) для последующего их поиска другими организациями и интеграции в свои системы. UDDI является проектом, который позволяет организациям публиковать описания веб-сервисов.

**Платформы - JavaEE, .Net, Node.js, PHP** (Software AG webMethods OneData, Oracle SOA Suite 12C, Microsoft WCF).

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

**REST API – Representational** State Transfer **–** архитектурный стиль взаимодействия компонентов распределенного приложения в сети.

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

**Представление** – это URI (идентификатор ресурса), управление ресурсом с помощью глаголов (GET, POST, PUT, DELETE).

В отличие от веб-сервисов на основе SOAP, не существует «официального» стандарта для RESTful веб-API. Дело в том, что REST является архитектурным стилем, в то время как SOAP является протоколом. Несмотря на то, что REST не является стандартом сам по себе, но REST использует стандарты HTTP, URL/URI, XML, JSON. Форматы передачи данных Передача данных в XML, JSON или в обоих форматах.

Два основных типа ресурса – коллекции и элемент коллекции: /api/users, /api/users/288.

Шесть обязательных ограничений:

- модель клиент-сервер;

- отсутствие состояния на стороне сервера, сохранение состояния допускается на стороне клиента, допускается сохранение состояния 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);

- для клиента сервер должен представляться конечным;

- код по требованию: допускается (необязательно) выгрузка на клиент апплетов или сценариев для расширения его функциональности.

Унифицированный Идентификатор Ресурса, в простонародье — **URI**

**URI – Uniform Resource Identificator** — **последовательность** символов, **идентифицирующая** физический или абстрактный **ресурс**, который **не обязательно** должен быть доступен **через сеть Интернет**, причем, тип ресурса, к которому будет получен доступ, определяется контекстом и/или механизмом.

В современном интернете, чаще всего используется две разновидности URI — URL и URN.

URL – Uniform Resource Locator — это URI, который, помимо идентификации ресурса, предоставляет ещё и информацию о местонахождении этого ресурса. А URN – Uniform Resource Name — это URI, который только идентифицирует ресурс в определённом пространстве имён, но не указывает его местонахождение.

Общепринятые правила:

* **Общий префикс** для всех ресурсов сервиса … /API/…, http://API.BSTU.BY/...
* Два типа ресурсов: **коллекция**, **элемент коллекции** /api/users/238, /api/students/ef3d26.
* **Иерархическая** связь …/api/users/238/cars/aah4899
* **Ограничить количество HTTP-статусов**, сопроводить сообщение дополнительным **кодом** (например 20003, 404001,…), сделать **отдельный ресурс** **для пояснения ошибок** (HATEOAS link) http://ccc/api/errors/20003.
* **Подавление статуса ответа** …/api/students/ef3d26?status\_code=200.
* **Версионность** /api/students/ef3d26?v=7.
* **Постраничное** получение данных: параметры limit, offset.
* **Сортировка**: параметр sort.
* Все **фильтры** вынести **за знак вопроса**: …/api/students?minbday=1998101&maxbday=20001231&gender=m.
* Пользователь получает **только то, что хочет**: …/api/students?field=bday,surname,gender.
* Обозначать в запросе **формат** сообщений: …/api/students.json?field=bday,surname,gender;
* **Глобальный** поиск: ../api/search?q=19600107+Иванов.
* **Документация**.

Недостатки:

* нет общепризнанного стандарта RESTful API;
* не все браузеры поддерживают полный словарь REST-методов (PUT, DELETE); на практике часто используется только GET и POSТ
* не однозначны коды состояний.

HATEOAS (Hypermedia as the Engine of Application State) - это правило архитектуры REST приложения – гипермедиа в качестве управления состоянием.

С HATEOAS клиент взаимодействует с сетевым приложением, сервер которого обеспечивает динамический доступ через гипермедиа. Клиент REST не нуждается в предварительных знаниях о том, как взаимодействовать с приложением или сервером, кроме общего понимания гипермедиа.

REST-клиент обращается к фиксированному URL, а все последующие действия клиента становятся известными из возвращаемых с сервера ресурсов. Типы ресурсов, представления и их связи стандартизированы. Клиент проходит по ресурсам, выбирая ссылки или взаимодействуя любым другим способом, возможным для этого типа ресурса. Таким образом RESTful-взаимодействия работают через гипермедиа, а не через заранее указанный интерфейс

Смысл в том, что работа с API осуществляется посредством ссылок

Пользователь видит, что ему доступно при определенных условиях.

Гипермедиа: технология обработки, структурирования информации и произвольного доступа к ее элементам с помощью гиперсвязей, WWW – реализация гипермедиа.

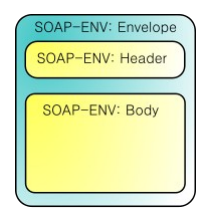
HATEOAS позволяет создать более независимую клиентскую компоненту. Вместе с ответом от сервера присылается ссылки помеченные с помощью специального компонента rel.

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

Simple Object Access Protocol – простой протокол доступа к объекту. Это не просто протокол, но и платформа на основе которой можно разрабатывать свои собственные протоколы. Протокол обмена структурированными сообщениями в распределённой вычислительной среде.

Первоначально - реализация RPC, расширение протокола XML-RPC, позже - произвольный обмен XML-сообщениями. спецификация поддерживается W3C, последняя версия 1.2, в версии 1.2. аббревиатура SOAP не расшифровывается.

формат сообщений: SOAP Envelope, SOAP Header (необязательный), SOAP Body, SOAP Fault



Сообщение SOAP – это простой XML-документ, который состоит из следующих элементов:

* **Envelope** Обязательный элемент, который определяет начало и конец сообщения.
* **Header** Опциональный элемент, который может включать в себя любые опциональные (не обязательные) атрибуты сообщения.
* **Body** Обязательный элемент, который включает в себя данные XML, которые определяют сообщение, которые должно быть отправлено.
* **Fault** Опциональный элемент, который предоставляет информацию об ошибках, которые возникают в процессе обработки сообщения.

Элемент SOAP сообщения **ENVELOPE**определяет начало и конец сообщения. Именно благодаря этому элементу, клиент, принимающий сообщение знает, когда сообщение полностью получено.

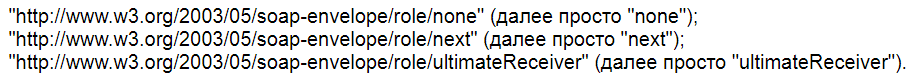
При работе с SOAP важно учитывать следующие моменты:

* Элемент Envelope является обязательной частью SOAP сообщения.
* Внутри каждого элемента Envelope должен быть один элемент Body.
* Элемент Envelope определяется с помощью пространства имён ENV и элемента Enveloper.

Элемент Header даёт нам возможность определять дополнительные свойства приложения. Например, мы можем использовать его для определения сигнатуры сервисов, защищённых паролем.

**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).



**Роль** может быть указана во всех блоках и указывает, кому предназначена информация блока. Обычно, обработанный блок извлекается из конверта, если не указано другого.

**Узел –** компонент SOA архитектуры, который может получать и отправлять сообщения.

**SOAP:** стандартные пространства имен:

**env**-описание самого SOAP,

**enc**-кодировка нашего сообщения,

**rpc**-для работы с рпс,

**rep**- Этот документ описывает семантику и сериализацию блока заголовка SOAP для переноса представлений ресурсов в сообщениях SOAP,

**xop-** механизм, определенный для сериализации информационных наборов XML (информационных наборов), содержащих двоичные данные, а также десериализации обратно в информационный набор XML,

**xmime**

**Пространства имён:** [**https://www.informit.com/articles/article.aspx?p=169106&seqNum=2**](https://www.informit.com/articles/article.aspx?p=169106&seqNum=2)

*Пространства* имен *XML* играют важную роль в сообщениях *SOAP*.  *SOAP* сообщение может включать в себя несколько различных *XML* - элементов в Header и Body, и, чтобы избежать столкновения имен, каждый из этих элементов должно быть идентифицировано уникальным пространством имен.

Использование пространств имен *XML* делает *SOAP* гибким и расширяемым протоколом.

Envelope , определяет пространство имен стандартных элементов *SOAP -*Envelope , Header и Body.

Каждый блок заголовка в элементе Header должен иметь собственное пространство имен. Это особенно важно, поскольку пространства имен помогают приложениям *SOAP* идентифицировать блоки заголовков и обрабатывать их отдельно.

Реальная сила пространств имен *XML* выходит за рамки простого предотвращения конфликтов имен, а также правильного управления версиями и обработкой. Использование полных имен для *SOAP* и данных приложения сообщает получателю *SOAP,* как обрабатывать сообщение, и какие схемы *XML* применять для проверки его содержимого. Например, различия в конкретной версии блока заголовка могут повлиять на то, как получатель обрабатывает сообщения, поэтому определение версии блока заголовка по его пространству имен позволяет получателю переключать модели обработки или отклонять сообщения, если он не поддерживает указанная версия.

**Ключевые аспекты:**

При работе с SOAP стоит учитывать следующие моменты

* Элемент Header является не обязательным.
* Этот элемент используется для добавления новых возможностей и функциональности.
* Элемент header можно использовать несколько раз в файле.
* Header является первым к прочтению файлом внутри элемента envelope.

**Аттрибуты элемента Header**

Элемент Header может иметь следующие два атрибута:

* **Actor** Протокол SOAP определяет расположение сообщения, как список сервисов SOAP. Каждый из этих промежуточных сервисов может выполнять определённую работу, а затем передавать сообщение следующему сервису по цепочке. С помощью атрибута Actor, клиент может определить получателя элемента header.
* **MustUnderstand** Этот аттрибут показывает, является ли элемент обязательным или опциональным (**true –**обязательный, **false**– нет). Если элемент обязательный, то получатель ведомляется об этом и обрабатывает элемент **header** соответственно указанной семантике, либо возвращает ошибку.

Элемент Body опредёлен содержит обязательную информацию, которая предназначена для конечного получателя SOAP сообщения.

Элемент SOAP сообщения **fault**обрабатывается в случае какой-либо ошибки. Ошибка возвращается отправителю сообщения.

атрибут **env: mustUnderstand = true** требует обязательную обработку; если обработчик не разобрал блок, то обработка прерывается и формируется сообщение с блоком Fault.

атрибут **env: mustUnderstand = false**, если обработчик не разобрал блок, то ошибка может быть проигнорирована.

атрибут **env: relay = true**, если обработчик не разобрал блок, то блок передается дальше.

атрибут **env: relay = false**, если обработчик не разобрал блок, то блок не передается дальше

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

WSDL (Web Services Description Language) — язык описания [веб-сервисов](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B8%D1%81) и доступа к ним, основанный на языке [XML](https://ru.wikipedia.org/wiki/XML). Последняя официальная спецификация на момент написания статьи версия 2.0, которая имеет статус рекомендации, и версия 1.1, которая имеет статус заметки (note). *Содержит не только формальную часть, описанную с хмл, но и содержит множество комментов, что превращает это в документацию, описывающую не только формально сервис, но и документацию.*

Каждый документ WSDL 1.1 можно разбить на следующие логические части:

1. определение типов данных (types) — определение вида отправляемых и получаемых сервисом XML-сообщений
2. элементы данных (message) — сообщения, используемые web-сервисом. Их может быть несколько и они могут состоять из нескольких частей
3. абстрактные операции (portType) — список операций, которые могут быть выполнены с сообщениями
4. связывание сервисов (binding) — протоколы свзи, используемые веб-сервисом, форматы сообщений и детали протоколов для каждого порта. Способ, которым сообщение будет доставлено

WSDL 1.1

Компоненты описания. В документах WSDL 1.1 используется фиксированный корневой элемент с удобным названием <wsdl:definitions>. В пределах этого корневого элемента в пространстве имен WSDL 1.1 определены один «пассивный» дочерний элемент (просто ссылка на отдельные документы WSDL 1.1) и пять «активных» дочерних элементов (которые собственно и составляют описание сервиса):

* <wsdl:import> ссылается на отдельный документ WSDL 1.1 с описаниями, подлежащими включению в этот документ;
* <wsdl:types> определяет типы XML или элементы, используемые для обмена сообщениями;
* <wsdl:message> определяет фактическое сообщение с точки зрения типов или элементов XML;
* <wsdl:portType> определяет абстрактный набор операций, осуществленных сервисом;
* <wsdl:binding> определяет фактическую реализацию <wsdl:portType> с помощью конкретных протоколов и форматов;
* <wsdl:service> определяет сервис в целом, как правило, включая один или несколько элементов <wsdl:port> с информацией доступа для элементов <wsdl:binding>.

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**.

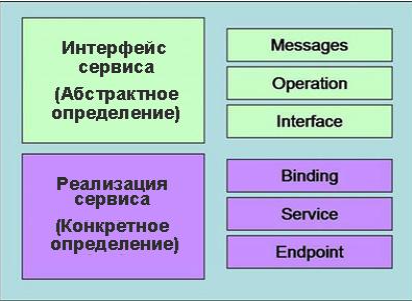
!Стандарт WSIL построен на основе XML. Он представляет собой **модель** распределенного поиска **информации** о службах в точках предложения поставщиков служб. Для организации такого поиска **поставщики** служб **документируют** **способ поиска** предоставляемых ими служб. В документах WSIL хранится **информация** о том, в каких **областях** сайта следует искать **описания веб-служб**. Поскольку в стандарте WSIL применяется механизм распределенного поиска, этот стандарт расширяет возможности UDDI, позволяя находить на сайтах информацию о службах, которых нет в реестре UDDI.

Язык WSDL применяется для описания интерфейсов и экземпляров веб-служб на основе XML. Это расширяемый язык, позволяющий описывать службы как конечные точки обработки сообщений. Эти конечные точки не зависят от конкретных форматов сообщений и сетевых протоколов, применяемых для передачи данных. Документы WSDL позволяют разработчикам публиковать информацию о своих службах по протоколам UDDI и WSIL, а также путем массовой пересылки адресов служб в документы WSDL по электронной почте или через Internet. !

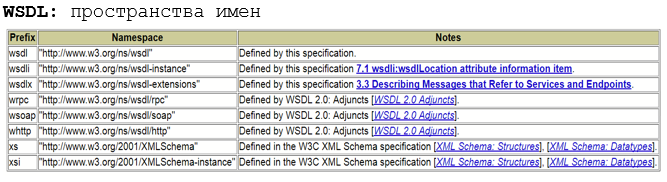
Протокол SOAP позволяет организовать привязку обнаруженных веб-служб и работу с ними путем определения путей для маршрутизации сообщений. Протокол SOAP может применяться для передачи запросов UDDI при поиске веб-служб.

Службы хранятся в поставщике служб, который обеспечивает доступ к ним по протоколам SOAP/HTTP и SOAP/JMS. Документы WSDL с описанием веб-служб хранятся на сервере поставщика служб или в специальном реестре. Ссылки на документы WSDL могут храниться как в реестре UDDI, так и в документах WSIL. Эти ссылки указывают на файлы WSDL с описаниями веб-служб.

**Концептуальная модель WSDL 2.0**

Описание Web-сервиса можно разделить на две части. В абстрактной части описания Web-сервис описывается в языке WSDL с помощью системы типов, обычно W3C XML-схемы, в терминах сообщений, которые этот сервис отправляет и получает. Шаблоны обмена сообщениями определяют последовательность и количество сообщений. Элемент **operation** связывает шаблоны обмена сообщениями с одним или несколькими сообщениями. Элемент **interface** группирует операции (элементы **operation**) независимо от транспорта и способа доставки.

В конкретной части описания элементы **binding** задают транспорт и формат доставки для интерфейсов (элементов **interface**). Элемент сервиса (элемента **service**) **endpoint** связывает сетевой адрес в соответствие со связыванием (элементом **binding**). Наконец, элемент **service** группирует конечные точки (элементы **endpoint**), которые реализуют общий интерфейс (элемент **interface**). На рисунке 1 изображена концептуальная модель компонентов WSDL.

****

<!-- 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>

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

Active Server Method Extended; технология Microsoft для разработки web-сервисов(2002, 2007), основанная на XML, WSDL, SOAP, (UDDI); официальное название в MSDN «XML Web Services»; 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.

**Плюсы ASMX:**

* Легкость в разработке
* Легкость в изучении
* Нет «ада» конфигурирования

**Плюсы WCF:**

* Очень разнообразные и гибкие возможности транспорта
* Актуальная и развивающаяся технология
* Различные варианты хостинга
* Возможность реализации большого множества стандартов WS-\*

Веб-сервис представлен одним обычным классом с одной лишь обязательной особенностью – некоторые его методы помечены специальным атрибутом **[WebMethod]**. Такие методы класса становятся веб-методами веб-сервиса с соответствующей сигнатурой вызова. Этот класс должен обладать конструктором по умолчанию. При каждом новом запросе IIS его инстанциирует дефолтным конструктором и вызывает соответствующий метод.

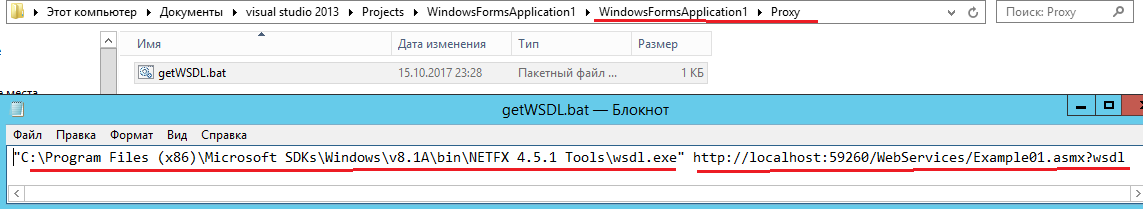
Вторая обязательная часть минимальной конструкции – это файл с расширением asmx, внутри которого необходимо указать этот класс.

**Прокси-класс с помощью wsdl.exe**

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.

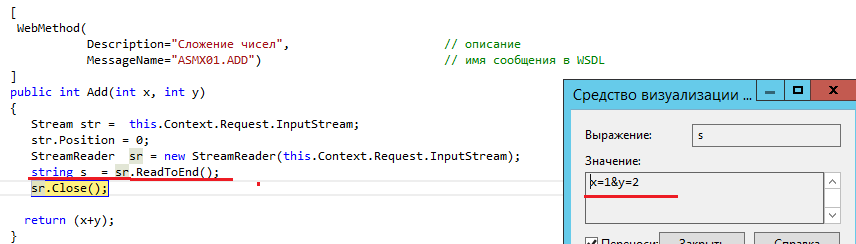
Утилита расположена в папке C:\Program Files (x86)\Microsoft SDKs\Windows, более того, она там представлена в разных версиях, в зависимости от версии .net, разрядности, версии windows и visual studio.

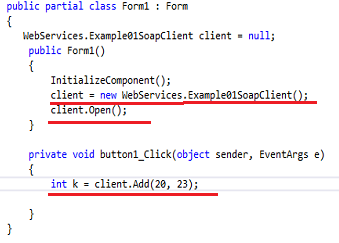
Как известно, wsdl описание веб-сервиса в технологии ASMX генерируется автоматически. Однако иногда возникает обратная задача: по данному wsdl файлу разработать соответствующий ему веб-сервис. Решается она с помощью той же утилиты wsdl.exe. Она может создать необходимый скелет из классов и вам останется только реализовать программную логику веб-методов.

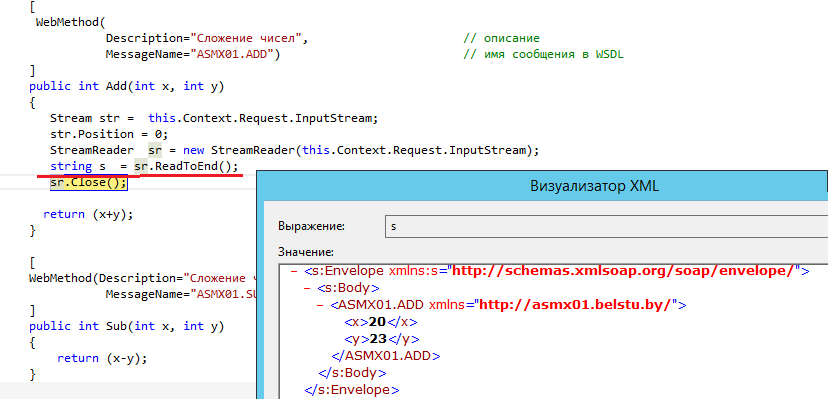
****

В Visual Studio, к любому проекту добавить ASMХ-службу.

Написать методы:



****

****

**Деплой (публикация)**

С помощью команды **Publish** и метода File System Visual Studio создает в указанной папке набор файлов, который необходимо предоставить для IIS

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.

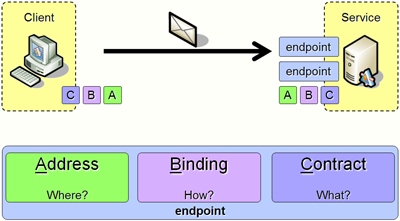
**Windows Communication Foundation** – технология основанная на .NET FRAMEWORK для разработки приложений SOA-архитектуры, первая версия 2006 (Indigo).

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

Основные **принципы**:

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

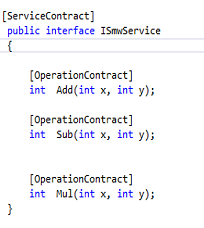
Коммуникация модель:



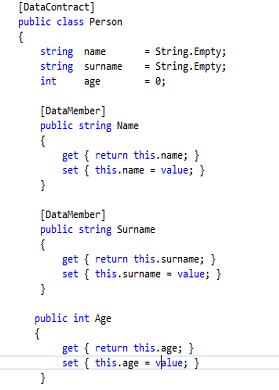
Контракты – это набор операций, который определяет, какую функциональность конечная точка предоставляет клиенту. Обычно состоит из имени интерфейса.

* контракт службы;
* контракт данных;
* контракт сообщений

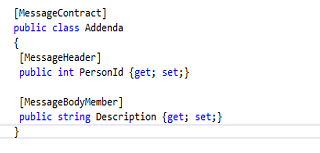
**WCF:** контракт службыэтот контракт предоставляет клиенту, а также внешнему миру информацию о предложениях конечной точки и протоколах, которые будут использоваться в процессе связи.

****

**WCF:** контракт данных – указывает каким образом данные будут сериализованы. Данные, которыми обменивается сервис, определяются контрактом данных. И клиент, и сервис должны быть согласованы с договором на данные.

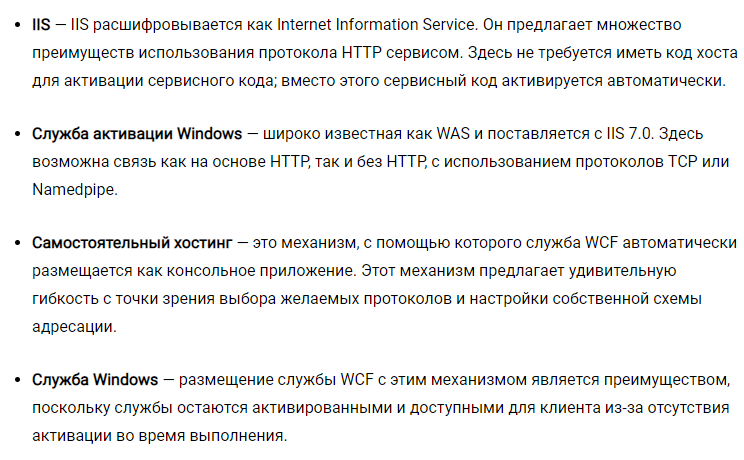


**WCF:** контракт сообщений – описывает формат сообщений –контракт данных контролируется контрактом сообщения. Он в первую очередь выполняет настройку форматирования типов параметров сообщений SOAP. Здесь следует отметить, что WCF использует формат SOAP для связи. SOAP означает простой протокол доступа к объектам.

****

**Хостинг:**

хост **–** контейнер для сервиса (любое С#-приложение) (Приложение которое создаёт процесс нашего веб-приложения в операционной системы). Основное назначение хоста: присоединить WCF-инфраструктуру, создать список конечных точек.

****



**Конечная точка** – адрес, привязка, контракт.

Все взаимодействия со службой Windows Communication Foundation (WCF) осуществляется через **конечные точки** службы.

Конечные точки предоставляют клиентам доступ к функциональным возможностям, предоставляемым службой WCF.

Каждая конечная точка состоит из четырех свойств:

* адрес, показывающий, где можно найти конечную точку;
* привязку, показывающую, как клиент может связаться с конечной точкой;
* контракт, определяющий доступные операции;
* набор поведений, задающих сведения о локальной реализации конечной точки.

Привязки:

**безопасность**:WS-ReliableMessaging; WS-Security

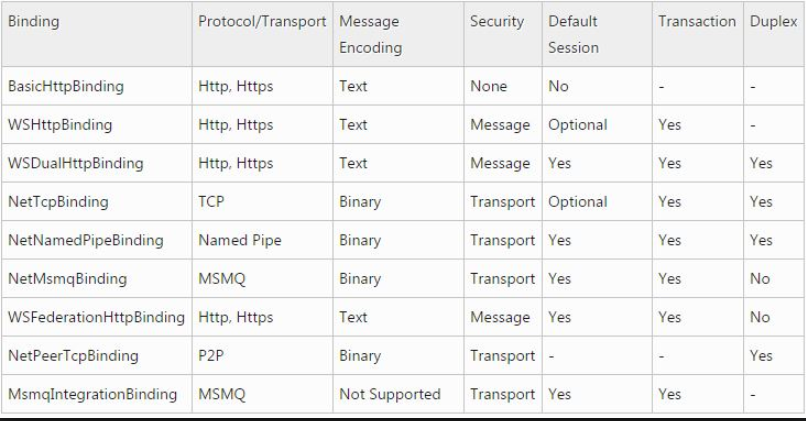
**кодирование:**

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

**транспорт:**

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

**стандартные:**

****

* [BasicHttpBinding](https://msdn.microsoft.com/ru-ru/library/system.servicemodel.basichttpbinding(v=vs.110).aspx): привязка протокола HTTP, которая подходит для подключения к веб-службам, соответствующим спецификации WS-I Basic Profile (например, службы на основе веб-службы ASP.NET).
* [WSHttpBinding](https://msdn.microsoft.com/ru-ru/library/system.servicemodel.wshttpbinding(v=vs.110).aspx): привязка с возможностью взаимодействия, которая подходит для подключения к конечным точкам, соответствующим протоколам WS-\*.
* [NetNamedPipeBinding](https://msdn.microsoft.com/ru-ru/library/system.servicemodel.netnamedpipebinding(v=vs.110).aspx): использует .NET Framework для подключения к другим конечным точкам WCF на том же компьютере.
* [NetMsmqBinding](https://msdn.microsoft.com/ru-ru/library/system.servicemodel.netmsmqbinding(v=vs.110).aspx): использует .NET Framework для создания подключений очередей сообщений к другим конечным точкам WCF.

**ASMX&WCF**

**Плюсы ASMX:**

* Легкость в разработке
* Легкость в изучении
* Нет «ада» конфигурирования

**Плюсы WCF:**

* Очень разнообразные и гибкие возможности транспорта
* Актуальная и развивающаяся технология
* Различные варианты хостинга
* Возможность реализации большого множества стандартов WS-\*

1. 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.

Служба WCF может похвастаться надежной системой безопасности с двумя режимами или уровнями безопасности, так что только предполагаемый клиент может получить доступ к службам.

Безопасность в WCF обеспечивается следующими функциями:

**Аутентификация** — здесь аутентификация не ограничивается идентификацией отправителя сообщения, а является взаимной, т. Е. Аутентификация получателя сообщения требуется для исключения возможности любого рода атак со стороны посредника.

**Авторизация** — это следующий шаг, предпринятый службой WCF для обеспечения безопасности, и здесь определяется, должна ли служба авторизовать вызывающую программу для продолжения или нет. Хотя авторизация не зависит от аутентификации, она обычно следует за аутентификацией.

**Конфиденциальность** — обмен информацией между вызывающим абонентом и службой является конфиденциальным, чтобы ограничить его толкование другими лицами, для которых сообщение не предназначено. Чтобы сделать это возможным, шифрование используется вместе с множеством других механизмов.

**Целостность** . Последняя ключевая концепция — сохранение целостности, т. **Е.** Предоставление гарантии того, что сообщение не было подделано кем-либо на пути от отправителя к получателю.

Режимы безопасности передачи:

**Нет.** Этот режим не гарантирует никакой безопасности сообщений, и служба не получает учетные данные о клиенте. Этот режим очень рискованный, поскольку может допускать подделку сообщений и, следовательно, не рекомендуется.

**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.

**WCF RPC:**

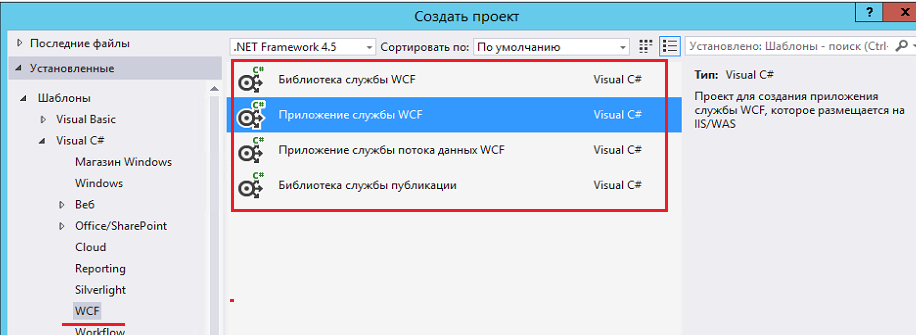
****

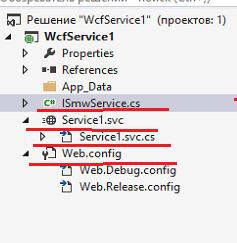
**WCF REST:** [**https://www.c-sharpcorner.com/article/wcf-restful-service/**](https://www.c-sharpcorner.com/article/wcf-restful-service/)

**Несколько конечных точек:**

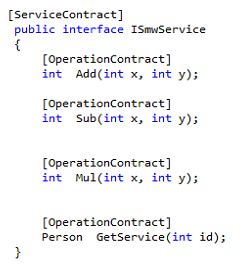
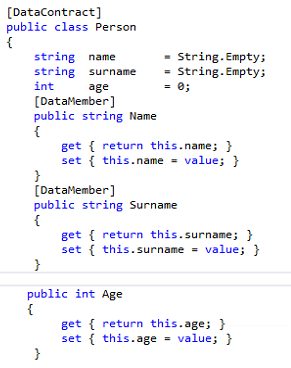
[**https://docs.microsoft.com/ru-ru/dotnet/framework/wcf/samples/multiple-endpoints**](https://docs.microsoft.com/ru-ru/dotnet/framework/wcf/samples/multiple-endpoints)

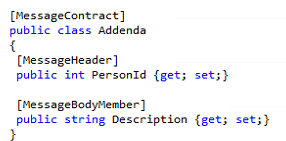
**Разработка WCF RPC:**

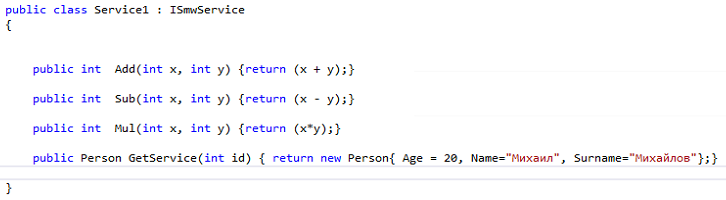
****

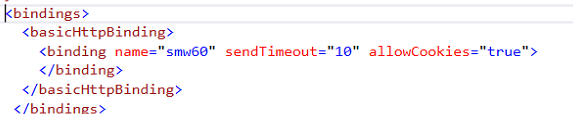
****

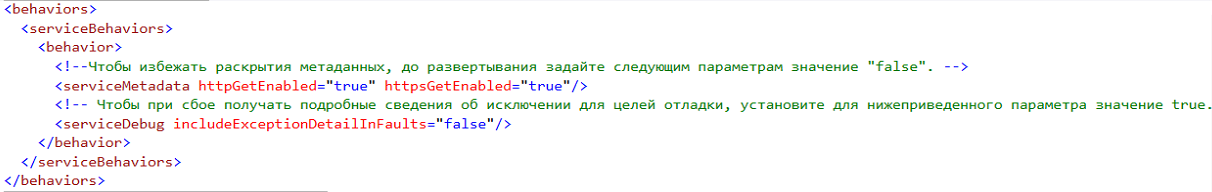


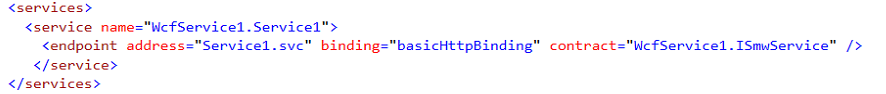
** **

****

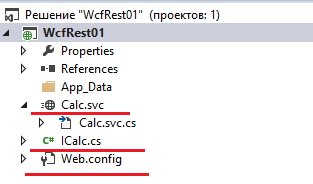
****

****

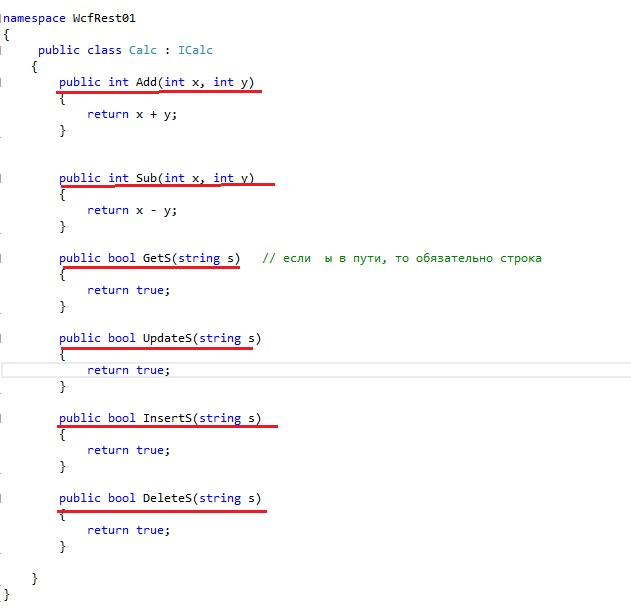
****

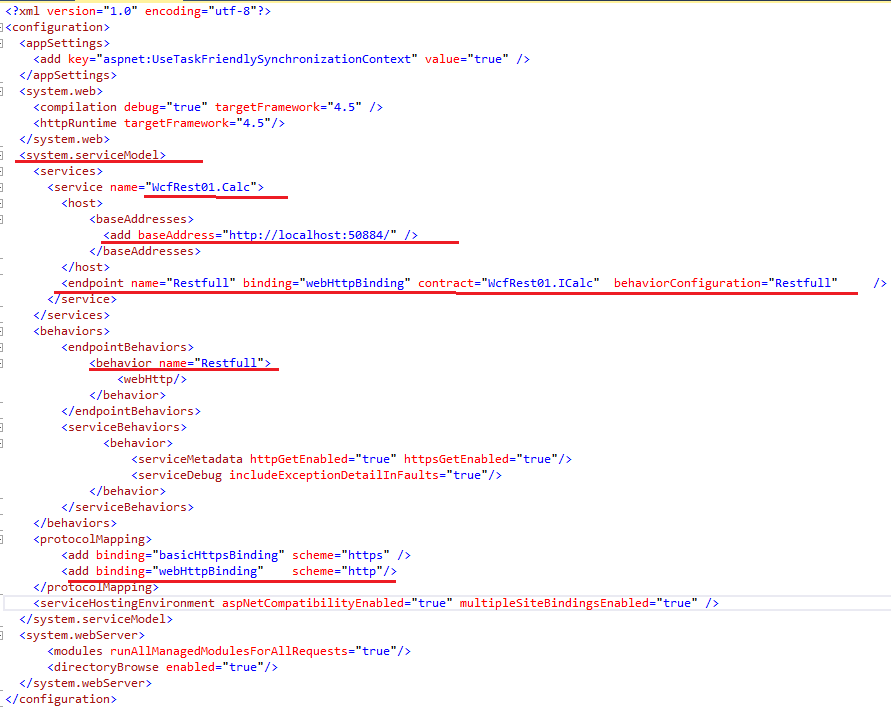
****

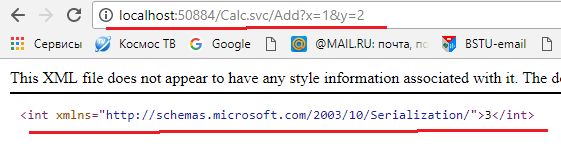
**WCF/REST:** разработка WCF-службы с REST-интерфейсом











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.

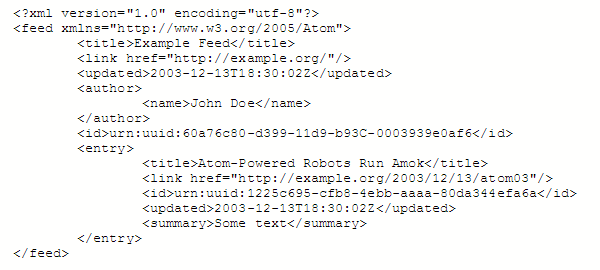
Синдикация – это механизм интеграции приложения, в котором сервер предоставляет некоторые данные в совместимом формате, известным как канал.

Канал – набор данных приложения, состоящий из некоторых метаданных уровня канала (заголовок, автор, урл и др.).

Atom – формат описания web-ресурсов и протокола для их публикации (2005г.).

Формат синдикации Atom основан на XML и позволяет описывать наборы веб-ресурсов.

MIME: application/atom+xml

****

**Дополнительные возможности Atom**

**Модель содержания**

Atom представляет возможность собирать и представлять информацию в различных форматах, таких как обычный текст, HTML, XHTML, XML, двоичные данные в формате Base64 и ссылки на внешнее содержание документов (например, .pdf), видеофайлов, аудиопотоков и еще ряда других.

**Представление даты и времени**

Для представления даты и времени Atom использует международный стандарт, описанный в [RFC 3339](http://tools.ietf.org/html/rfc3339) (из состава стандартов ISO 8601).

**Интернационализация**

Atom имеет возможность объединить в одном канале информацию на различных языках. Это обеспечивается стандартным атрибутом xml:lang и позволяет делать человеко-читаемыми различные части фида.

**Модульность**

Спецификация Atom изначально разрабатывалась как открытая и модульная. Это позволяет расширять формат путем подключения внешних модулей, с одной стороны, а с другой — экспортировать возможности Atom во внешние приложения и другие форматы. К примеру, модули RSS 1.0 или RSS 2.0 можно использовать в канале Atom.

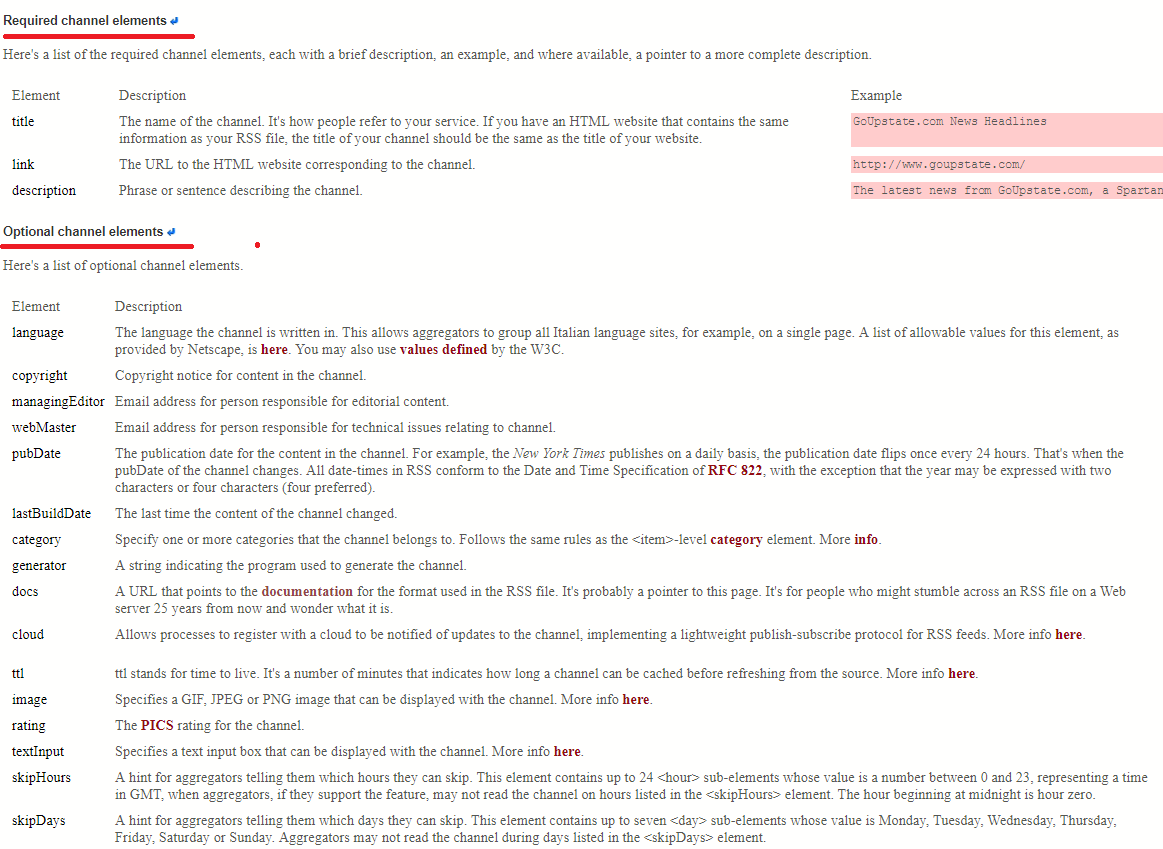
**RSS 2.0 –** **Rich Site Summary** – семейство XML-форматов, предназначенного для описания новостных лент, анонсов и статей

– основанный на XML формат описания лент новостей, анонсов статей, изменений в блогах и форумах и т. п.

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 (например, заголовки новостей, блогов и пр.). Feedly, Яндекс.Лента, My Yahoo.

****

****

**Структура и элементы RSS 2.0**

RSS — диалект XML, поэтому описание канала RSS обязано соответствовать [спецификации XML](http://www.w3.org/TR/REC-xml)

Корневым элементом документа RSS является элемент <rss> с обязательным атрибутом version, указывающим версию формата канала. Для формата RSS 2.0 значение этого атрибута должно быть 2.0. Единственный дочерний элемент первого уровня, <channel>, включает метаинформацию о канале (описание канала новостей) и его содержимое — обязательные и необязательные дочерние элементы разной вложенности. Элементы могут содержать обязательные и необязательные атрибуты, определяющие свойства и поведение элемента.

**ATOM/RSS:**

ATOM более универсальный и чаще применяется; следует использовать его при разработке новых приложений.

1. RSS пользуется двумя протоколами передачи информации, Atom имеет один стандартизированный канал. Таким образом Atom решает проблему совместимости протоколов Blogger и MetaWeblog (используются в RSS).  
   **Выбор: Atom.**
2. Atom требует большего количества данных, RSS более демократичен. На самом деле мне нравится, что в Atom требуется метка «последнее обновление». В то же время я люблю RSS за простоту пользования.  
   **Выбор: оба приложения имеют свои сильные стороны.**
3. В RSS невозможно отсылать html страницу или форматированный текст, что позволяет делать Atom, поэтому подписка по Atom лучше для восприятия, несмотря на большую сложность в подготовке (требуется больше данных data).  
   **Выбор: RSS (простота и автоматизация процесса получения информации на первом месте).**
4. Atom дает отчетливое представление о полноте публикации в подписке, сразу понятно, анонс это или полный текст. В RSS это не ясно, поэтому приходится загружать сайт-источник.  
   **Выбор: однозначно Atom**
5. Автопоиск стандартизирован в спецификации Atom, хотя в RSS тоже существует несколько способов поиска фидов.  
   **Выбор: Atom, хотя это не принципиальный критерий.**
6. Обработка контента — одна из важнейших функций служб подписки. И здесь Atom получает явное преимущество. RSS публикует контент только внутри монолитного документа, включающего несколько записей и распознаваемого как документ.  
   Atom 1.0 позволяет формировать автономные документы ленты, которые могут быть переданы с помощью любого сетевого протокола, например, XMPP. Atom также имеет поддержку для агрегированных фидов, позволяющих в случае включения записи в другие фиды, давать в ней ссылку на фид, где она была первоначально опубликована.  
   **Выбор: Atom.**

**Дополнительные возможности RSS 2.0**

Согласно спецификации, версия 2.0 поддерживает следующие схемы в элементах <link> и <url>: http://, https://, news://, mailto: и ftp://. В предыдущих версиях поддерживались только http:// и ftp://.

Порядок разработки: <https://professorweb.ru/my/csharp/web/level10/10_3.php>

**Atom:**

**Элемент <feed>**

**Обязательные элементы**

<id> — Идентификатор фида, постоянный URI.

<title> — Название канала. Не может быть пустым.

<updated> — Дата последнего обновления в формате [RFC 3339](http://www.faqs.org/rfcs/rfc3339.html).

**Рекомендуемые элементы**

<author>

<link> — Адрес связанного сайта. Тип связи определяется в атрибуте rel.

**Необязательные элементы**

<category> — Задает категории, к которым относится канал. Элемент feed может включать несколько элементов category.

<generator> — Название программы, с помощью которой собран канал.

<icon> — Маленькое изображение, пиктограмма канала.

<logo> — Большое изображение, логотип канала.

<rights> — Информация об авторском и смежных правах.

<subtitle> — Человеко-читаемое описание или подзаголовок канала.

**RSS:**

**Элемент <channel>** определяет канал как таковой и содержит всю информацию.

**Обязательные элементы канала**

title — Название канала, по которому люди будут ссылаться на сервис. Если канал связан с веб-сайтом, то значение title должно совпадать с заголовком стартовой страницы сайта.

link — URL веб-сайта, связанного с каналом.

description — Краткое описание канала.

**Необязательные элементы канала**

language — Язык канала, как [определено W3C](http://www.w3.org/TR/REC-html40/struct/dirlang.html#langpres).

pubDate — Дата публикации канала как определено в спецификации [RFC 822](http://asg.web.cmu.edu/rfc/rfc822.html).

lastBuildDate — Время последнего изменения содержимого канала.

category — Указывает одну и более категорию, к которой относится канал

ttl — Время жизни: количество минут, на которые канал может кешироваться перед обновлением с ресурса.

image — Изображение GIF, JPEG или PNG, которое может отображаться с каналом. Например, логотип компании.

**Атрибуты элементов**

В качестве примера элемента с атрибутами рассмотрим <image>. Это — дочерний элемент <channel>, содержащий три обязательных и три дополнительных элемента:

url — URL изображения GIF, JPEG или PNG, представляющего канал

title — Название изображения, которое будет использовано в атрибуте alt при отображении канала в виде гипертекста.

link — URL сайта; изображение канала будет служить ссылкой на этот сайт.

width, height — Необязательные атрибуты, задающие ширину и высоту изображения в пикселях. Максимальные размеры изображения — 144 х 400px, по умолчанию — 88 x 31px. Например: <image url="http://4stud.info/img/logo.gif"

title="Учебные материалы для студентов АСОИУ"

link="http://4stud.info"

width="200px" height="60px" />

description — Необязательное описание изображения.

**Элемент <item>** Канал может содержать любое количество элементов <item>, описывающих отслеживаемую информацию. Каждый <item> может содержать заголовок сообщения, его содержание, ссылку на источник, информацию об авторе.

**Дочерние элементы <item>**

title — Заголовок сообщения.

link — URL сообщения.

description — Краткий обзор сообщения.

author — Адрес электронной почты автора сообщения.

category — Включает сообщение в одну или более категорий.

enclosure — Описывает медиа-объект, прикрепленный к сообщению, например:

pubDate — Показывает, когда сообщение было опубликовано.

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

**Open Data Protocol (OData)** — это открытый веб-протокол для запроса и обновления данных. Протокол позволяет выполнять операции с ресурсами, используя в качестве запросов HTTP-команды, и получать ответы в форматах [XML](https://ru.wikipedia.org/wiki/XML) или [JSON](https://ru.wikipedia.org/wiki/JSON).

Начиная с версии 4.0, OData — открытый стандарт, одобренный [OASIS](https://ru.wikipedia.org/wiki/OASIS).

Компонент платформы .NET Framework Службы данных WCF (известный ранее как службы данных ADO.NET) позволяет создавать службы, использующие Open Data Protocol (OData) для представления и получения данных по сети или интранету с помощью семантики протокола [REST](http://go.microsoft.com/fwlink/?LinkId=113919). OData представляет данные в качестве источников, к которым можно обращаться по URI. Обращаться к данным и изменять их можно с помощью стандартных HTTP-заголовков GET, PUT, POST и DELETE. OData использует связи и сущности [Модель EDM](https://msdn.microsoft.com/ru-ru/library/ee382825(v=vs.110).aspx) для представления источников в качестве наборов сущностей, связанных с помощью ассоциаций.

В основном применяется совместно с реляционными базами данных и есть куча продуктов которая трансформирует язык запросов, которые делаются с помощью параметров запросов и по своей мощности практически равен sql запросам.

Средства Visual Studio упрощают создание служб на основе OData с использованием модели данных ADO.NET платформы Entity Framework. Можно также создать каналы OData на основе классов среды CLR и даже данных с поздним связыванием или нетипизированных данных.

Создание службы данных

1. В **Обозревателе решений** щелкните правой кнопкой мыши имя проекта ASP.NET и выберите **Добавить новый элемент**.
2. В диалоговом окне **Добавление нового элемента** выберите пункт **Служба данных WCF**.
3. В качестве имени службы укажите **Northwind**. Visual StudioВ Visual Studio для новой службы создаются файлы разметки и кодов XML. По умолчанию открывается окно редактора кода. В **обозревателе решений** для службы будет отображаться имя Northwind с расширением .svc.cs или .svc.vb.

В коде службы данных замените комментарий /\* TODO: put your data source class name here \*/ в определении класса, задающего службу данных, типом контейнера сущностей модели данных, который в данном случае равен NorthwindEntities. Определение класса должно выглядеть следующим образом.

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

**JSON-RPC** (*JavaScript Object Notation Remote Procedure Call* ) — [протокол](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB_%D0%BF%D0%B5%D1%80%D0%B5%D0%B4%D0%B0%D1%87%D0%B8_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) [удалённого вызова процедур](https://ru.wikipedia.org/wiki/%D0%A3%D0%B4%D0%B0%D0%BB%D1%91%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80), использующий [JSON](https://ru.wikipedia.org/wiki/JSON) для кодирования сообщений. Это очень простой протокол (очень похожий на [XML-RPC](https://ru.wikipedia.org/wiki/XML-RPC)), определяющий только несколько типов данных и команд. JSON-RPC поддерживает уведомления (информация, отправляемая на сервер, не требует ответа) и множественные вызовы. Это частный случай JSON, смесь JSON и REST.

Последняя версия 2.0.

JSON-RPC работает, отсылая запросы к серверу, реализующему протокол. Клиентом обычно является программа, которой нужно вызвать метод на удалённой системе. Множество входных параметров может быть передано удалённому методу, как массив или объект. Метод также может вернуть множество выходных данных (это зависит от реализации). Удалённый метод вызывается отправлением запроса на удалённый сервер посредством [HTTP](https://ru.wikipedia.org/wiki/HTTP) или [TCP/IP](https://ru.wikipedia.org/wiki/TCP/IP) [сокета](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BA%D0%B5%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D1%8B%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81)) (начиная с версии 2.0). При использовании HTTP, [заголовок](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B3%D0%BE%D0%BB%D0%BE%D0%B2%D0%BA%D0%B8_HTTP) Content-Type определяется как application/json

Все передаваемые данные — простые объекты, [сериализованные](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) в JSON.

Запрос — вызов определённого метода, предоставляемого удалённой системой. Он должен содержать три обязательных свойства:

* method — Строка с именем вызываемого метода.
* params — Массив объектов, которые должны быть переданы методу, как параметры.
* id — Значение любого типа, которое используется для установки соответствия между запросом и ответом.

Сервер должен отослать правильный ответ на каждый полученный запрос. Ответ должен содержать следующие свойства:

* result — Данные, которые вернул метод. Если произошла ошибка во время выполнения метода, это свойство должно быть установлено в null.
* error — Код ошибки, если произошла ошибка во время выполнения метода, иначе null.
* id — То же значение, что и в запросе, к которому относится данный ответ.

Для ситуаций, когда ответ не требуется, были введены уведомления. Уведомление отличается от запроса отсутствием свойства id, которое не требуется, так как не будет передан ответ.

Обзор

JSON-RPC - это легкий протокол удаленного вызова процедур (RPC) без сохранения состояния. В первую очередь эта спецификация определяет несколько структур данных и правила их обработки. Он независим от транспорта, так как концепции могут использоваться в одном и том же процессе, через сокеты, через http или во многих различных средах передачи сообщений. Он использует JSON в качестве формата данных.

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). Термин «Примитив» в данной спецификации относится к любому из этих четырех примитивных типов JSON.

Термин «структурированный» относится к любому из структурированных типов JSON. Всякий раз, когда этот документ ссылается на какой-либо тип JSON, первая буква всегда пишется с большой буквы: Object, Array, String, Number, Boolean, Null. True и False также пишутся с большой буквы.

Все имена членов, которыми обмениваются между Клиентом и Сервером и которые рассматриваются для сопоставления любого рода, должны рассматриваться с учетом регистра. Термины функция, метод и процедура могут считаться взаимозаменяемыми.

Клиент определяется как источник объектов Request и обработчик объектов Response.

Сервер определяется как источник объектов Response и обработчик объектов Request.

Одна реализация этой спецификации может легко заполнить обе эти роли, даже в одно и то же время, для других разных клиентов или одного и того же клиента. Эта спецификация не затрагивает этот уровень сложности.

Запрос объекта

Вызов rpc представляется путем отправки объекта запроса на сервер. Объект Request имеет следующие члены:

Jsonrpc Строка, указывающая версию протокола JSON-RPC. ДОЛЖЕН быть точно "2.0".

Method Строка, содержащая имя вызываемого метода. Имена методов, которые начинаются со слова rpc, за которым следует символ точки (U + 002E или ASCII 46), зарезервированы для внутренних методов и расширений rpc и НЕ ДОЛЖНЫ использоваться ни для чего другого.

Params Структурированное значение, которое содержит значения параметров, которые будут использоваться во время вызова метода. Этот член МОЖЕТ быть опущен.

Id Идентификатор, установленный Клиентом, который ДОЛЖЕН содержать значение String, Number или NULL, если оно включено. Если он не включен, предполагается, что это уведомление.

Сервер ДОЛЖЕН ответить тем же значением в объекте Response, если он включен. Этот член используется для корреляции контекста между двумя объектами.

Использование Null в качестве значения для элемента id в объекте Request не рекомендуется, поскольку в данной спецификации используется значение Null для ответов с неизвестным идентификатором. Кроме того, поскольку JSON-RPC 1.0 использует значение идентификатора Null для уведомлений, это может вызвать путаницу при обработке

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

OWIN связывает хост, вебсервер и собственно ваше приложение. Позволяет сделать свой собственный хост. Компоненты являются независимыми и связываются с помощью OWIN.

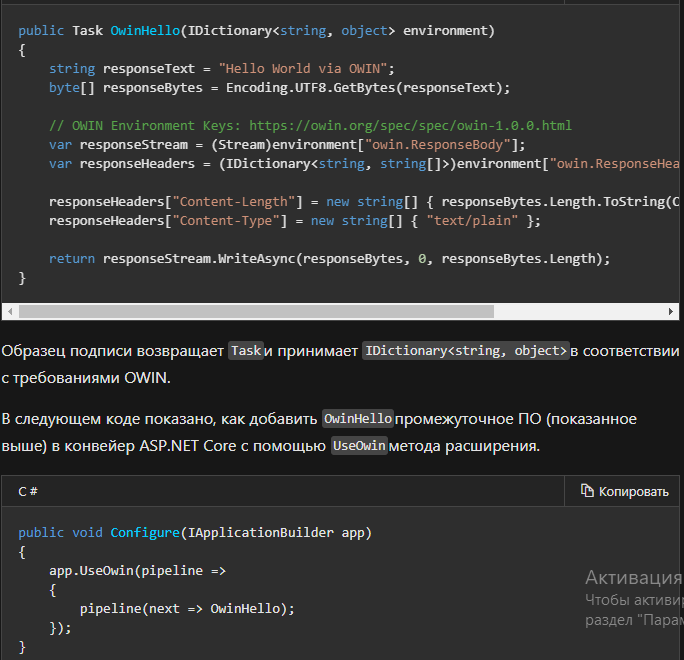
У приложения есть хост, веб-сервер и Application. Мы говорим о фреймворке, который позволяет эффективно разработать часть application – Nancy.

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.

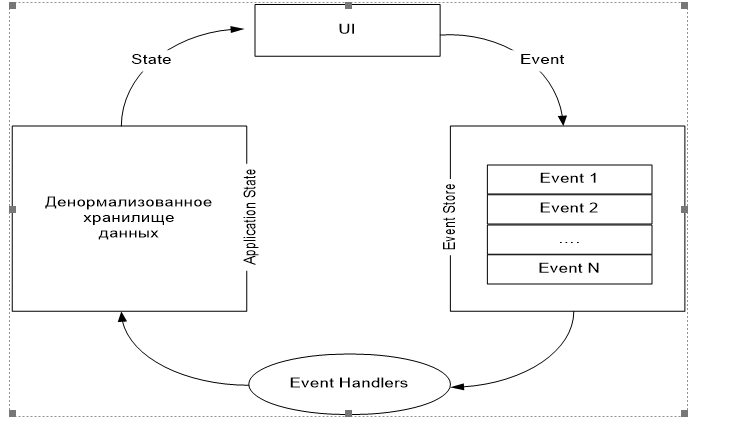
1. **Event Storing:** назначение, принципы применения, примеры реализации.

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

Условно говоря если девушка выходит замуж и меняет фамилию мы заходим в систему, находим этого человека и меняем ему фамилию. Если мы говорим о EventStoring, то в этом случае записывается информация что с такого времени такая информация была изменена на такую. 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.

* 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

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. В дополнение к возможности независимого развертывания и масштабирования каждый сервис также получает четкую физическую границу, которая позволяет разным сервисам быть написанными на разных языках программирования. Они также могут разрабатываться разными командами.

**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:** правильное распределение функциональности на микросервисы в соответствием с ***потребностями бизнеса***. Каждый микросервис включает в себя полный набор технологий, но обеспечивает одну технологию. Организация разрабатывающая приложения с микросервисной архитектурой должны иметь ***кросс-функциональные*** команды.

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

**Микросервисы:** применение микросервисной архитектуры, как правило, приводит к необходимости поддержки системы разработчиками (правило «вы разработали, вам и поддерживать», Amazon),

**Микросервисы:** тенденция, передача разработчикам – поддержку микросервисов (Amazon, Netflix).

**Микросервисы:** популярные паттерны ***Tolerant Reader***, ***Consumer Driver***.

**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;

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

**Микросервисы:** автоматизация, продукты DevOps: ***Continuous Delivery***, ***Continuous Integration***.

В большинстве случаев **непрерывная доставка** — это серия практик, направленных на то, чтобы обновления программного обеспечения происходили практически постоянно. Данные методы гарантируют быстрое развёртывание на продакшене не меняя существующий функционал. Непрерывная доставка осуществима благодаря различным оптимизациям на ранних этапах процесса разработки.

Разработчик, сделав какую-либо фичу, отправляет её QA-инженерам для тестирования. Тестировщикам легче досконально оттестировать небольшой новый функционал и написать к нему тест-кейсы. Как только все проверки – прошли, новая фича попадает на дальнейшее тестирование авто-тестами и потом уже в релизный брэнч в системе контроля версий.

**Continuous delivery** поставляет бизнесу каждый функционал постепенно. Это позволяет получить сразу отклик от клиента и, при необходимости, сделать некоторые изменения.

#### Другие преимущества Continuous delivery:

1. Внесение нового функционала в back-end для проверки совместимости с системой;
2. Быстрое реагирование на потребности рынка;
3. Возможность подстраивания под изменение бизнес-стратегии;
4. Низкое количество потенциальных ошибок.

**Непрерывная интеграция** является ключевым компонентом практики Agile Development. Основой данной практики является постоянное попадание кода в центральный репозиторий после успешного запуска тестов. Основные цели **continuous integration** – поиск и устранение потенциальных проблем как можно быстрее, улучшение качества ПО и сокращение время для выпуска обновлений.

До того, как непрерывная интеграция стала широко распространённой, разработчики обычно работали изолировано, а только по окончанию работы объединяли свои наработки. Порой это был очень трудоёмкий и длительный процесс.

При непрерывной интеграции разработчики часто заливают свои изменения в центральный репозиторий, выполняя до этого unit – тесты. Затем система контроля версий автоматически проверяет код на возможно безопасной интеграции с существующим в репозитории. При этом идёт постоянное поступление кода, что облегчает тестирование и сводит к минимуму возможные риски.

#### Идеальный процесс выглядит примерно так:

* разработчик отправляет код в центральный репозиторий;
* на сервере непрерывной интеграции изменения объединяются с основным кодом, выполняются юнит – тесты и всё заливается на стэйжинг среду;
* в стэйжинг среде QA инженеры тестируют приложение;
* дальше всё проверяется для попадания на продакшен;
* развёртывание на продакшене.

В конце концов, все эти «непрерывные» штуки способствуют устранению накладных расходов процесса разработки. Однако, не стоит забывать о целесообразности всех этих процессов. Возможно, для вашего бизнеса это будет излишним.

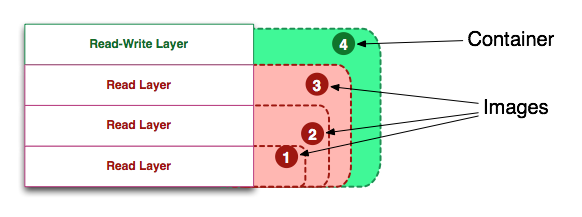
1. **Docker**: назначение, архитектура, основные команды

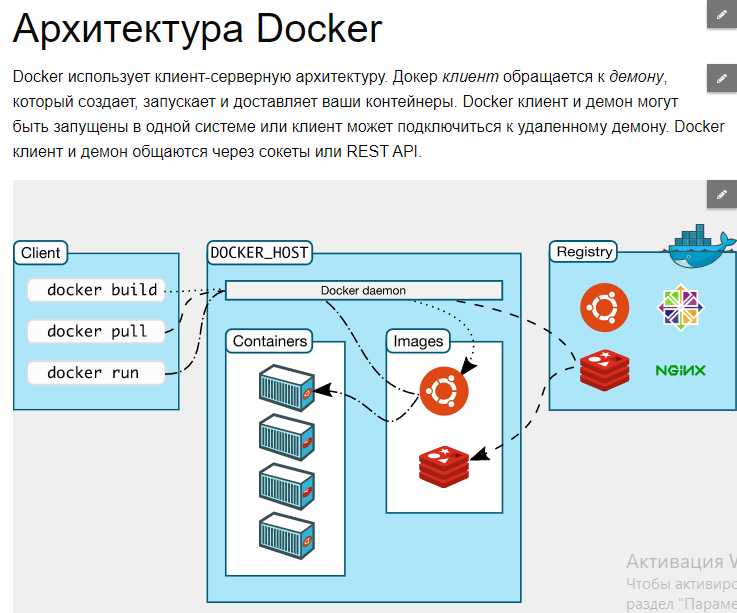
Docker - платформа для разработки, доставки и эксплуатации приложений. Основное назначение – упростить развертывание приложения.

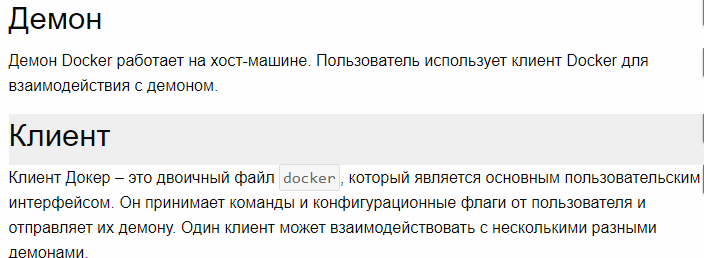
**Docker** = Docker Desktop(Engine) + Docker Hub

**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.

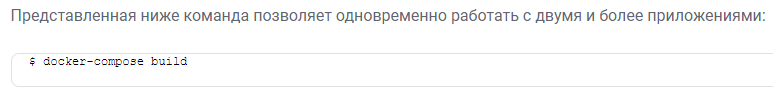
Compose превосходен для разработки, тестирования и настройки среды, а также непрерывной интеграции. Вы этом разделе вы можете узнать более подробно о [решаемых задачах](https://dker.ru/docs/docker-compose/overview-of-docker-compose/#common-use-cases).

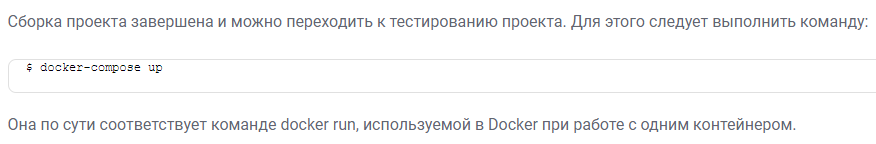
Использование Compose обычно разделяется на три этапа:

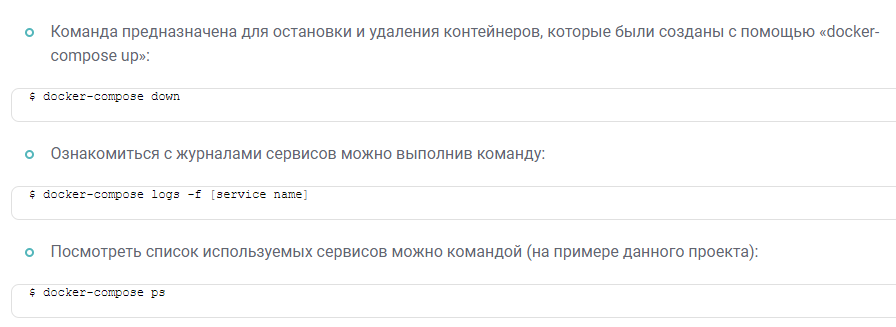
1. Определение окружения вашего приложения в Dockerfile, это можно сделать в любом месте.
2. Определение сервисов из которых будет состоять ваше приложение в docker-compose.yml, в последствии они смогут быть запущены все вместе в изолированном окружении.
3. И наконец, выполнение команды docker-compose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

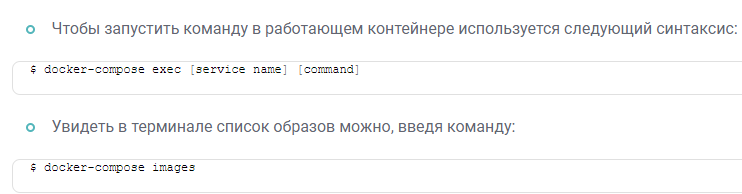
Compose имеет команды для управления всем жизненным циклом приложения:

* 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.