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

**Веб-приложение**

**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.

**Сервис**(версия смелова на лк) – приложение, которое предоставляет интерфейс или сетевой апи для других прилаг

**свойства сервиса**

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

**ИНТЕРФЕЙС - ПОИМЕННОВАНЫЙ НАБОР СИГНАТУР МЕТОДОВ** (в том числе событий, свойств и индексаторов, т.к. всё это по сути синтаксический сахар для методов)

**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, при необходимости, 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;  нескольких запросов, которые трактуются как единый запрос.
* 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 построить (построить модель бизнес процесса) и выполнить бизнес процесс на основе группы сервисов. Другими словами разработать новый сервис на основе последовательного выполнения нескольких сервисов.

Workflow engine это программный продукт, позволяющий соединить весь бизнес процесс в корпоративной информационной системе от начала до его завершения, система для воспроизведения потока работ по имеющейся модели.

При этом обработка данных на отдельных этапах осуществляется в различных независимых друг от друга приложениях, а функции организации процесса и связи различных подсистем реализует специализированная Workflow система.

**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):**

СОА супервизор это, можно сказать, главный служебный сервис, функционирующий все время работы системы и контролирующий и координирующий работу всех остальных, прежде все 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.

Трудно переоценить важность этого компонента.

Не секретом является тот факт, что для достижения определенного уровня перфоманса гораздо проще не использовать принцип низкой связанности, поскольку его реализация ведет к необходимости создания определенной инфраструктуры, которая несомненно налагает свой отпечаток на скорость выполнения.

Поэтому реализуя принцип СОА необходим своего рода контролирующий компонент, который своевременно будет оповещен в случае возникновения каких-либо проблем в ходе выполнения, чтобы можно было своевременно принять меры и продолжать обеспечивать клиентам достаточный уровень сервиса.

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

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

1. **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-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).

**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. Каждый документ WSDL 1.1 можно разбить на следующие логические части:

* определение типов данных (types) — определение вида отправляемых и получаемых сервисом XML-сообщений

- элементы данных (message) — сообщения, используемые web-сервисом

- абстрактные операции (portType) — список операций, которые могут быть выполнены с сообщениями

- связывание сервисов (binding) — способ, которым сообщение будет доставлено

**UDDI** (англ. Universal Description Discovery & Integration, произносится [юди́]) — инструмент для расположения описаний веб-сервисов (WSDL) для последующего их поиска другими организациями и интеграции в свои системы. UDDI это кроссплатформенное программное обеспечение, основанное на XML. UDDI является открытым проектом, спонсируемым OASIS, который позволяет организациям публиковать описания веб-сервисов.

**UDDI** — это основанный на XML стандарт для описания, публикации и поиска веб-сервисов.

**Платформы-JavaEE, .Net, Node.js и т.д**

[Симплексная, полудуплексная и дуплексная связь | LAZY SMART](http://lazysmart.ru/osnovy-avtomatiki/klassifikatsiya-kanalov-svyazi-simplek/)

Способы клиент серверного взаимодействия:











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

REST (сокр. от англ. Representational State Transfer — «передача состояния представления») — архитектурный стиль взаимодействия компонентов распределённого приложения в сети. Это альтернатива rpc.

[REST, что же ты такое? Понятное введение в технологию для ИТ-аналитиков / Хабр (habr.com)](https://habr.com/ru/post/590679/?)

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

В отличие от веб-сервисов (веб-служб) 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, JavaScript Object Notation (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** — 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.

Основное различие между ними — в задачах:

* **URL** — Uniform Resource Locator, помогает найти какой либо ресурс
* **URN** — Uniform Resource Name, помогает этот ресурс идентифицировать

Общепринятые правила:

* Общий префикс для всех ресурсов сервиса … /API/…, http://API.BSTU.BY/...
* Два типа ресурсов: коллекция (users, students, …), 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 (HATEOAS link) для пояснения ошибок http://ccc/api/errors/20003.
* Подавление статуса ответа …/api/students/ef3d26?status\_code=200.
* Версионность /api/students/ef3d26?v=7.
* 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 (желательна поддержка нескольких форматов): …/api/students.json?field=bday,surname,gender; один из форматов должен быть по умолчанию; могут применяться заголовки Accept и Content-Type со значениями application/xml и application/json для запроса или обозначения в ответе формата.
* 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(insert, delete, update);
* не однозначны коды состояний.

HATEOAS (Hypermedia as the Engine of Application State) - это правило архитектуры REST приложения.

С HATEOAS клиент взаимодействует с сетевым приложением, чьи серверы приложений предоставляют информацию динамически через гипермедиа . Клиент REST не нуждается в предварительных знаниях о том, как взаимодействовать с приложением или сервером, кроме общего понимания гипермедиа.

Смысл в том, что работа с API осуществляется посредством ссылок

Пользователь видит что ему доступно при определенных условиях.

механизм, который может в запросе отправить еще доп ссылки на другие сервисы

Гипермедиа: технология обработки, структурирования информации и произвольного доступа к ее элементам с помощью гиперсвязей (Тед Нильсон, 1965), WWW – реализация гипермедиа.

Платформы для разработки: сервлет, http-handler, web-api

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

Simple Object Access Protocol – простой протокол доступа к объекту.

Сетевой протокол прикладного уровня

Может перемещаться на http, tcp, smtp

* Чтобы в нттр перемещать, что нужно указать?  
  Контент-тайп: соап+хмл
* Как вписать в тср?
* Как придумали так и пишем  
  Надо придумать самим, после какого сообщения будет идти соап  
  Ибо нет там никаких заголовках или тип того
* **Надо самому придумать протокол**  
  Надо самому все разрабатывать в тср  
  Самому придумать протокол прикладного уровня или тип того...
* ***Т..е надо придумать какую-то последовательность данных, при которой сервер и клиент смогут договориться об обмене соап сообщений***

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.

SAOP: сайт W3C:

- основы (primer);

- каркас сообщения (messaging framework);

- дополнение (adjuncts);

- спецификация утверждений и коллекция тестов (specification assertions and test collection).

SOAP: primer (part 0)

- XML 1.0;

- XOP (XML-binary Optimized Packaging) – формат передачи двоичных вложений;

- SOAP MTOM (Message Transmission Optimization Mechanism) – формат;

- SOAP-node – участник обмена SOAP-сообщений (отправитель, получатель, посредник);

- SOAP-sender – отправитель SOAP-сообщения;

- SOAP-receiver – получатель SOAP-сообщения;

- конвенция обозначений

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 любый опциональные (не обязательные) аттрибуты сообщения.
* **Body** Обязательный элемент, который влючает в себя данные XML, которые определяют сообещние, которые должно быть отправлено.
* **Fault** Опцилнальный элемент, который предоставляет информацию об ошибках, которые возникают в процессе обработки сообещения.

Элемент SOAP сообщения **ENVELOPE**опредляет 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 Enveloper 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 определния сигнатуры сервисов защищённых паролем.

**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 слудующие два аттрибута:

* **Actor** Протокол SOAP определяет расположение сообщения, как список сервисов SOAP. Каждый из этих промежуточных сервисов может выполнять опредлеённую работу, а затем передавать сообщение следующему сервису по цепочке. С помощью аттрибута 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, илбо взвращает ошибку.

Элемент SOAP сообщения Bodyявляется обязательным и содержит предназначенные для передачи XML-данные определяемые приложением. Этот элемент должен быть помещён внутри элемента Envelopeи следовать после элемента Header, который может быть определён в SOAP сообщении.

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.

Данный механизм SOAP возвращает определённую информацию об ошибке, в которую входят:

* Код ошибки;
* Описание;
* Адрес обработчика SOAP, который сгенерировал ошибку.

**Под-элементы Fault**

Элемент SOAP сообщения  Fault имеет следующие под-элементы:

* **<faultCode>** Текстовый код, который указывает на тип ошибки (примеры укзаны ниже).
* **<faultString>** Текстовое сообщение с объяснением ошибки.
* **<faultActor>** Текстовое сообщение, которое указывает, кто именно является причиной ошибки. Крайне полезно, когда SOAP сообщение проходит через несколько элементов (nodes) и клиенту необходимо узнать, в каком имеено из этих узлов произошла ошибка. Узел, который не является конечным получателем должен включать в себя элемент *faultActor*.
* **<detail>** Этот элемент используется для хранения специального сообщения об ошибке, которое характерно для конкретного приложения. Этот элемент может включать под-элементы, которые называются **detail entries**.

**Код ошибок SOAP**

Значения, приведённые ниже должны использоваться в элементе faultCode для описаня ошибки.

* **SOAP-ENV:MustUnderstand** Промежуточный под-элемент элемента Header с аттрибутом mustUnderstand, который установлен на 1 не был понят.
* **SOAP-ENV:VersionMismatch** Находит неверное пространство имён в элементе Envelope.
* **SOAP-ENV:Server** Проблема произошла на сервере, из-за чего сообщение не может быть передано.
* **SOAP-ENV:Client** Сообщение было некорректно сформировано, либо содержало некорректную информацию.

Определены три стандартизованные роли (см. SOAP Часть 1 Раздел 2.2):

* "http://www.w3.org/2003/05/soap-envelope/role/none" (далее просто "none");
* "http://www.w3.org/2003/05/soap-envelope/role/next" (далее просто "next");
* "http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver" (далее просто "ultimateReceiver").

SOAP: Processing Model

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

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

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

После того, как SOAP-узел с помощью атрибута env:role верно идентифицирует все адресованные ему заголовочные блоки (и, возможно, также тело сообщения), дальнейшие действия по обработке, которые должны быть предприняты, определяются дополнительным атрибутом  env:mustUnderstand в заголовочных элементах.

Этот необязательный атрибут может включаться в заголовочные блоки SOAP-сообщения для обеспечения уверенности в том, что SOAP-узлы не проигнорируют важные для приложения в целом заголовочные блоки.

Если атрибут env:mustUnderstand имеет значение "true", это означает, что SOAP-узел, которому адресовано сообщение, **должен** обработать блок согласно спецификации этого блока. Фактически обработка SOAP-сообщения даже не должна начинаться, пока узел не идентифицирует и не "поймет" все обязательные блоки, адресованные ему.

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.

# SoapExtension Класс

Базовый класс расширений SOAP для веб-служб XML, создаваемых с помощью ASP.NET.

**!Принципы расширения**:

* добавить собственный неймспейс,
* описать множество тегов.
* Перечислить в соап в заголовке свой неймспейс.
* Обеспечить необходимые xml-схемы.
* **Пространства имён:** [**https://www.informit.com/articles/article.aspx?p=169106&seqNum=2**](https://www.informit.com/articles/article.aspx?p=169106&seqNum=2)

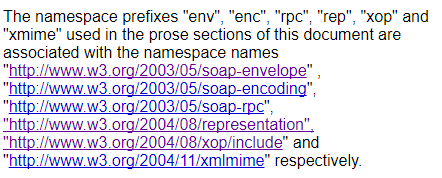
*Пространства* имен *XML* играют важную роль в сообщениях *SOAP*.  *SOAP* сообщение может включать в себя несколько различных *XML* - элементов в Header и Body, и, чтобы избежать столкновения имен, каждый из этих элементов должно быть идентифицировано уникальным пространством имен.

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.



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).

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
3. абстрактные операции (portType) — список операций, которые могут быть выполнены с сообщениями
4. связывание сервисов (binding) — 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.

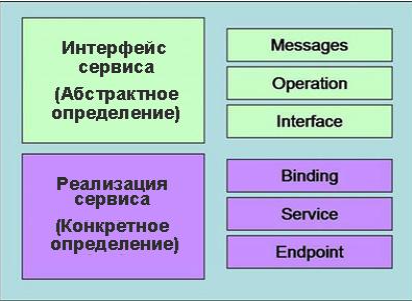
Стандарт WSIL построен на основе XML. Он представляет собой модель распределенного поиска информации о службах в точках предложения поставщиков служб. Для организации такого поиска поставщики служб документируют способ поиска предоставляемых ими служб. В документах WSIL хранится информация о том, в каких 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 адресов служб в документы WSDL по электронной почте или через Internet.

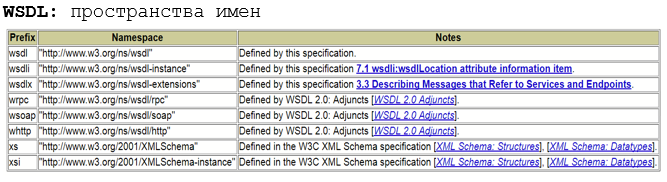
Протокол SOAP позволяет организовать привязку обнаруженных веб-служб и работу с ними путем определения путей для маршрутизации сообщений. Протокол SOAP может применяться для передачи запросов UDDI при поиске веб-служб.

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.

****

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; официальное название в MSDN «XML Web Services»; легкая технология: для работы с ней не обязательно знать XML, SOAP и WSDL.

Це RPC-сервис, веб-сервис

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

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

**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.

**Веб**-**сервис** - это открытая конечная точка, которая **обычно** используется как API, или, другими словами, ее конечный пользователь **обычно** является другим приложением, а не пользовательским интерфейсом. **Веб**-**метод** - это особый метод, который отображается через веб-службу.

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

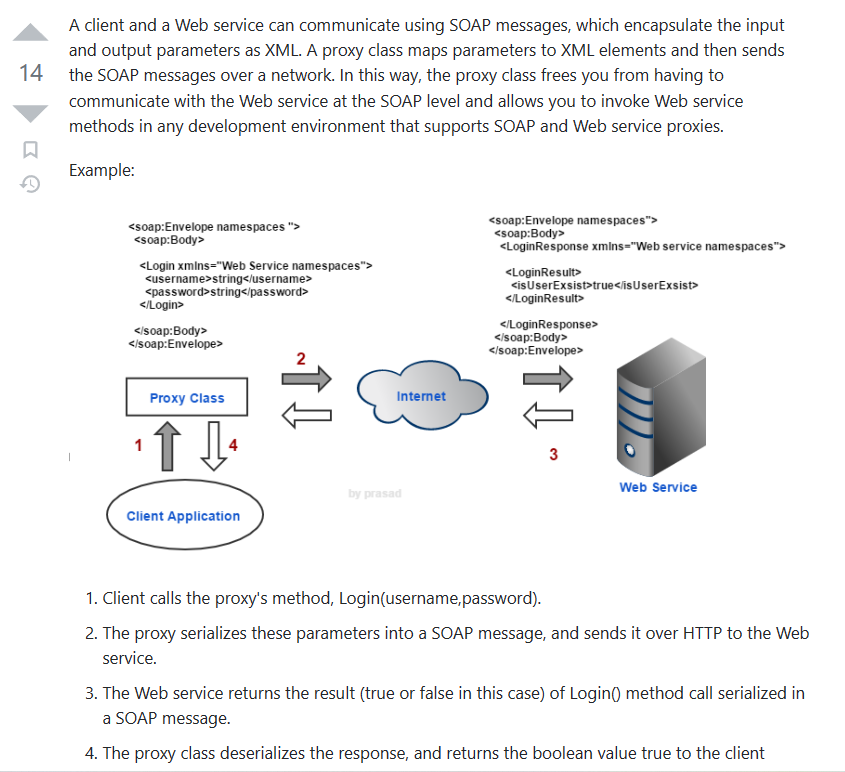
**Прокси-класс с помощью wsdl.exe**

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.

Нахуя нужен прокси



By definition, Web services can be communicated with over a network using industry standard protocols, including SOAP. That is, a client and a Web service communicate using SOAP messages, which encapsulate the in and out parameters as XML. Fortunately, for Web service clients, the proxy class handles the work of mapping parameters to XML elements and then sending the SOAP message over the network.

As long as a service description exists, a proxy class can be generated if the service description conforms to the Web Services Description Language (WSDL). A service description defines how to communicate with a Web service. With a service description, a proxy class can be created with the Wsdl.exe tool. In turn, a Web service client can then invoke methods of the proxy class, which communicate with a Web service over the network by processing the SOAP messages sent to and from the Web service. Because the proxy class communicates with the Web service across the Internet, it is a good idea to verify that the [Url](https://msdn.microsoft.com/en-us/library/w2dk0twy(v=vs.100)) property of the proxy class references a trusted destination.

By default, the proxy class uses SOAP over HTTP to communicate with the Web service. However, Wsdl.exe can generate proxy classes to communicate with a Web service, using either the HTTP-GET protocol or HTTP-POST protocol. To specify that the proxy class should use HTTP-GET or HTTP-POST, provide the **/protocol** switch to the Wsdl.exe tool, as described in the following table.

**SoapExtension** – это мощная техника, которая позволяет вручную видоизменять запросы и ответы, причем как со стороны веб-сервиса, так и со стороны клиента.

Возможности SoapExtension:

* Чтение контента запроса или ответа (просмотр soap конвертов в виде MemoryStream)
* Модификация контента запроса или ответа
* Чтение и обработка заголовков (soap header)
* 1 способ привязки: к веб-методу (веб-сервиса или клиента) через кастомный атрибут
* 2 способ привязки: к веб-сервису (или клиенту) с помощью web.config (app.config) без перекомпиляции!
* Взаимодействие с самим классом веб-сервиса или прокси-классом
* Создание цепочек из различных SoapExtension

Что можно сделать с помощью SoapExtension

* полное низкоуровневое логирование
* аутентификация
* наложение и проверка ЭЦП
* шифрование
* сжатие
* обработка soap header
* любое преобразование контента

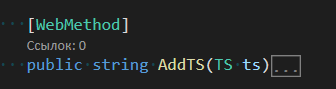
**Деплой (публикация)**

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

Порядок разработки:

1. Создать веб-сервис:

* создаём приложение типа веб-служба ASP.Net
* прописываем методы сервиса и с помощью атрибута WebMethod, помечаем, что это метод сервиса;



Атрибут WebMethod может иметь параметр EnableSession = true, который устанавливается, если веб-методу нужен сеанс.

* перед классом, где прописаны методы необходимо указать пространство имён, спецификацию собственности веб-служб и управление панелью элементов VS ToolBox

Атрибут [WebService имеет всего три свойства:

Namespace – дефолтный ХМЛ нэймспейс – указывать обязательно Description – описание веб-сервиса, отображаемое в браузере Name – имя веб-сервиса (по дефолту берется имя класса)

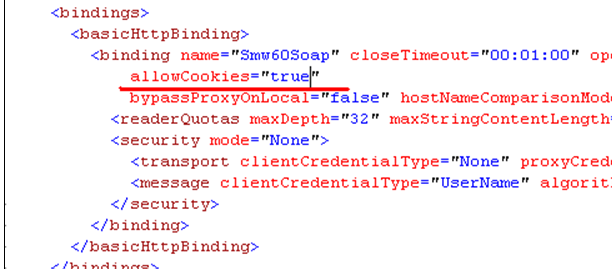
Namespace = http://tempuri.org/ - отладочное пространтсво имен для службы. Имена, используемые в файле WSDL службы используют это пространтство. Разработчик сам должен опеределить пространство имен для своей службы.

1. Создание клиента:

* создать приложение любого типа, например WinForms
* добавить в созданное приложение ссылку на службу с помощью ПКМ->Добавить ссылку на службу
* создать объект клиента и через клиента вызывать методы



* если использовался параметр EnableSession = true в вебметоде, то на клиенте должен быть установлен параметр allowCookies=”true” в теге <binding>



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** (**WCF**) — программный [~~фреймворк~~](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA)**ПЛАТФОРМА(**так сказал смелов**)**, используемый для обмена данными между приложениями, входящий в состав [.NET Framework](https://ru.wikipedia.org/wiki/.NET_Framework). До своего выпуска в декабре 2006 года в составе .NET Framework 3.0, WCF был известен под кодовым именем **Indigo**. Заложенные в неё принципы [интероперабельности](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D0%B1%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C) позволяют организовать работу с другими платформами, для чего используются технологии взаимодействия платформ, например WSIT, разрабатываемые на базе открытого исходного кода. Класс службы WCF не может существовать самостоятельно. **Каждая служба WCF должна находиться под управлением некоторого** [**процесса**](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0))[**Windows**](https://ru.wikipedia.org/wiki/Microsoft_Windows)**, называемого хостовым процессом.**

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;

Существуют несколько вариантов хостинга:

* Автохостинг (то есть хост-процессом является, к примеру, консольное или графическое Windows приложение) может сам себя запускать, но хз это вариант автохостинга или не
* Хостинг в одной из служб Windows
* Хостинг с использованием IIS
* **WCF:** 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;

Windows Communication Foundation (WCF) ― это унифицированная интегрированная среда для создания защищенных, надежных, транзакционных и интероперабельных распределенных приложений. В более ранних версиях Visual Studio имелись некоторые технологии, которые можно было использовать для взаимодействия между приложениями.

Однако если требовалось обеспечить совместный доступ к информации с любых платформ, необходимо было использовать веб-службу (также известную как веб-служба ASMX). Если достаточно было только обеспечить передачу данных между клиентом и сервером, использующим ОС Windows, то использовалось удаленное взаимодействие .NET Remoting. Если требовалось обеспечить транзактные коммуникации, то нужно было использовать Enterprise Services (DCOM), а если требовалось реализовать модель с организацией очереди, то нужно было использовать Message Queuing (или MSMQ).

WCF сводит воедино функциональные возможности всех этих технологий в унифицированную модель программирования. Это упрощает взаимодействие между разработанными распределенными приложениями.

Службы Службы данных WCF взаимодействуют с базой данных напрямую, что позволяет возвращать данные с помощью стандартных операций HTTP: GET, POST, PUT или DELETE.

Служба WCF основана на интерфейсе, задающем контракт между службой и клиентом. Она помечается атрибутом ServiceContractAttribute

Методы и функции, которые предоставляются службой WCF, задаются путем пометки их атрибутом OperationContractAttribute. Кроме того, можно предоставить сериализованные данные, пометив составной тип атрибутом DataContractAttribute. Это делает возможным привязку данных на клиенте.

После того как интерфейс и его методы были определены, они инкапсулируются в класс, реализующий этот интерфейс. Множественные контракты службы WCF может реализовать единственный ее класс.

Служба WCF предоставляется для использования посредством того, что называется конечной точкой. Конечная точка ― это единственный способ сообщения с этой службой; невозможно получить к ней доступ посредством прямой ссылки, в отличие от других классов.

Конечная точка состоит из адреса, привязки и контракта. Адрес задает расположение сервера; это может быть URL-адрес, FTP-адрес, а также сетевой или локальный путь. Привязка задает способ сообщения с этой службой.

Клиент WCF состоит из прокси, позволяющего приложению сообщаться со службой (WCF), и конечной точки. Прокси создается на стороне клиента в файле app.config и включает информацию о типах и методах, предоставляемых службой. В случае служб, предоставляющих несколько конечных точек, клиент может выбрать одну, лучше всего соответствующую его требованиям; например, клиент может выбрать способ сообщения посредством протокола HTTP и проверку подлинности Windows.

После создания клиента WCF можно обращаться к его службе в коде точно так же, как это делается с любым другим объектом

Служба WCF поддерживает несколько шаблонов обмена сообщениями, включая запрос-ответ, одностороннюю и дуплексную связь. Разные транспорты поддерживают разные шаблоны обмена сообщениями и таким образом влияют на типы поддерживаемых взаимодействий.

**Binding** – задает способ связи конечной точки с внешним миром. Она состоит из набора компонентов, называемых элементами привязки, которые компонуются в "стек", один над другим, образуя инфраструктуру связи. Как минимум, привязка определяет используемые транспорт (например HTTP или TCP) и кодирование (например, текстовое или двоичное).Привязка может содержать элементы привязки, задающие такие сведения, как механизмы безопасности, используемые для защиты сообщений, или шаблон сообщений, используемый конечной точкой.

**Behaviors** – компонент, управляющий различными аспектами работы службы, конечной точки, определенной операции или клиента во время выполнения. Расширения функциональности группируются в соответствии с областью действия: общие расширения функциональности влияют глобально на все конечные точки, расширения функциональности служб влияют только на аспекты, относящиеся к службам, расширения функциональности конечных точек влияют только на свойства, относящиеся к конечным точкам, а расширения функциональности уровня операции влияют только на конкретные операции. Например, одно расширение функциональности службы это регулирование, которое определяет реакцию службы при избытке сообщений, превосходящем возможности обработки. С другой стороны, поведение конечной точки управляет только аспектами, относящимися к конечным точкам, например способом поиска учетных данных безопасности и расположением для поиска.

**контракт службы - о**бъединяет несколько связанных операций в один функциональный модуль. Контракт может определять параметры уровня службы, такие как пространство имен службы, соответствующий контракт обратного вызова и другие подобные параметры. В большинстве случаев контракт задается путем создания интерфейса на выбранном языке программирования и применения атрибута ServiceContractAttribute к этому интерфейсу.

**контракт операции**

Контракт операции определяет параметры операции и тип возвращаемых ею данных. При создании интерфейса, определяющего контракт службы, контракт операции задается применением атрибута OperationContractAttribute к определению каждого метода, входящего в контракт. Операции могут задаваться как получающие одно сообщение и возвращающие одно сообщение или как получающие набор типов и возвращающие тип. В последнем случае формат сообщений, обмен которыми происходит при выполнении данной операции, определяется системой.

**контракт сообщения**

Описывает формат сообщения. Например, в нем описывается, должны ли элементы сообщения размещаться в заголовках или в тексте, какой уровень безопасности должен применяться к определенным элементам сообщения и т. д.

**контракт сбоя**

Может связываться с операцией службы для обозначения ошибок, которые могут возвращаться вызывающему объекту. С операцией могут быть связаны ноль или более сбоев. Эти ошибки представляют собой сбои протокола SOAP, которые моделируются в модели программирования как исключения.

**контракт данных**

Хранящееся в метаданных описание типов данных, используемых службой. Это описание позволяет другим объектам работать со службой. Типы данных могут использоваться в любой части сообщения, например в виде типов параметров или возвращаемых значений. Если в службе используются только простые типы, явное использование контрактов данных не требуется.

Безопасность в WCF включает обеспечение конфиденциальности (шифрование сообщений для исключения перехвата), целостности (средства обнаружения подделки сообщения), проверку подлинности (средства проверки серверов и клиентов) и авторизацию (управление доступом к ресурсам).Эти функции предоставляются либо путем использования существующих механизмов безопасности, таких как TLS по HTTP (также называемый HTTPS), либо путем реализации одной или нескольких различных спецификаций безопасности WS-\*.

КОНЕЧНЫЕ ТОЧКИ

Вся связь со службой Windows Communication Foundation (WCF) осуществляется через конечные точки службы. Конечные точки обеспечивают доступ клиентов к функциональным возможностям службы WCF.

Каждая конечная точка состоит из четырех свойств:

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

Каждая конечная точка состоит из следующего:

Адрес: адрес однозначно определяет конечную точку и указывает потенциальным потребителям на место расположения службы.В объектной модели WCF адрес представлен классом [EndpointAddress](https://msdn.microsoft.com/ru-ru/library/system.servicemodel.endpointaddress(v=vs.110).aspx).Класс [EndpointAddress](https://msdn.microsoft.com/ru-ru/library/system.servicemodel.endpointaddress(v=vs.110).aspx) содержит следующее.

Свойство [Uri](https://msdn.microsoft.com/ru-ru/library/system.servicemodel.endpointaddress.uri(v=vs.110).aspx), представляющее адрес службы.

Свойство [Identity](https://msdn.microsoft.com/ru-ru/library/system.servicemodel.endpointaddress.identity(v=vs.110).aspx), представляющее удостоверение безопасности службы и коллекцию необязательных заголовков сообщений.Необязательные заголовки сообщений используются для вывода дополнительной и более подробной информации, необходимой для идентификации конечной точки или взаимодействия с ней.

Привязка.

Служба WCF содержит ряд привязок, предоставляемых системой. Они являются коллекциями элементов привязки, оптимизированными для конкретных сценариев. Например, класс [WSHttpBinding](https://msdn.microsoft.com/ru-ru/library/system.servicemodel.wshttpbinding(v=vs.110).aspx) предназначен для взаимодействия со службами, реализующими различные спецификации WS-\*. Эти заранее определенные привязки экономят время, предоставляя только те параметры, которые могут быть правильно применены для конкретного сценария. Если заранее определенная привязка не удовлетворяет необходимым требованиям, можно создать собственную настраиваемую привязку.

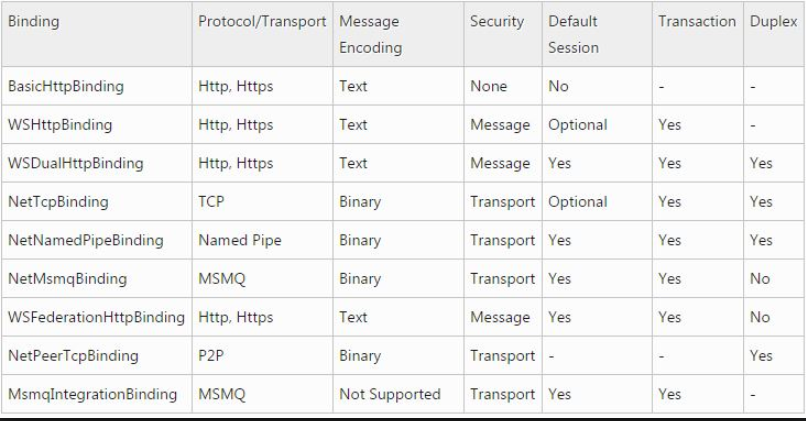
Привязка задает способ связи клиента с конечной точкой. В том числе следующее:

* + используемый транспортный протокол (например, TCP или HTTP);
  + используемую в сообщениях кодировку (например, текст или двоичное кодирование);
  + необходимые требования безопасности (например, безопасность сообщений SSL или SOAP).

Привязки - это объекты, используемые для задания сведений о связи, необходимых для подключения к конечной точке службы Windows Communication Foundation (WCF). Каждая конечная точка службы WCF требует точного задания привязки.

Информация в привязке может быть очень простой или очень сложной. Самые простые привязки указывают только транспортный протокол (такой как HTTP), который должен использоваться для подключения к конечной точке. Как правило, привязка содержит сведения о подключении к конечной точке, которые попадают в одну из следующих категорий.

Стандартные привязки

****

Контракты. Контракты показывают, какие функциональные возможности дает клиенту конечная точка. В контракте задается следующее:

* операции, которые могут быть вызваны клиентом;
* форма сообщения;
* тип входных параметров или данных, требуемых для вызова операции;
* тип обработки или ответного сообщения, который может ожидать клиент.
* Конечную точку для службы можно указать либо императивным методом (с помощью кода), либо декларативным (через настройки).

Что определяет привязка

Протоколы

Определяют используемый механизм безопасности: возможность надежного обмена сообщениями или параметры потока контекста транзакций.

рые параметры могут быть несовместимыми с другими. Поэтому WCF содержит набор привязок, предоставляемых системой. Эти привязки удовлетворяют большинство требований приложения. В следующих классах представлены некоторые примеры привязок, предоставляемых системой.

* [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.

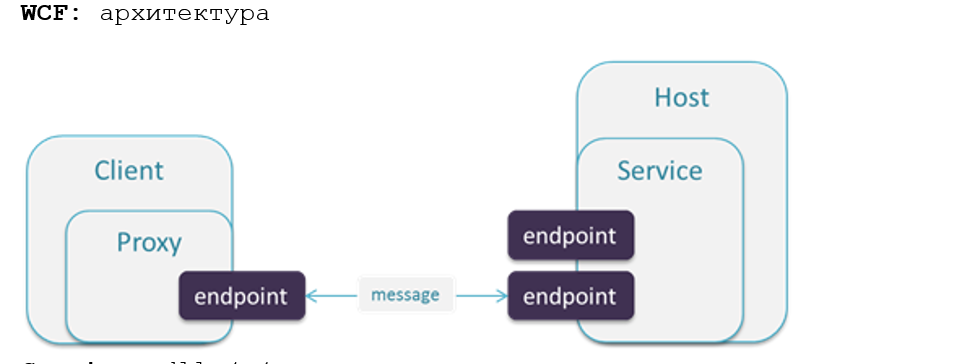
Если ни одна из предоставляемых системой привязок не имеет требуемого приложением службы сочетания функций, можно создать собственную привязку. Это можно сделать двумя способами. Можно создать новую привязку из уже имеющихся элементов привязки с помощью объекта [CustomBinding](https://msdn.microsoft.com/ru-ru/library/system.servicemodel.channels.custombinding(v=vs.110).aspx) или создать полностью пользовательскую привязку, наследуемую от привязки [Binding](https://msdn.microsoft.com/ru-ru/library/system.servicemodel.channels.binding(v=vs.110).aspx).

РАЗНИЦА ASMX vs WCF

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. Для слоя сообщений может быть использовано WSE. WCF предоставляет согласованную программную модель безопасности для любого протокола и поддерживает много таких же возможностей как IIS и WS-\* протоколы безопасности. Дополнительно, WCF предоставляет поддержку claims-based авторизации которая позволяет контролировать ресурсы более четко чем безопасность на основе ролей (role-based security). WCF безопасность остается согласованной не зависимо от того какой хост был использован для реализации WCF сервиса.
4. Еще одним важным отличием является то, что ASMX веб сервисы используют для сериализации класс  xmlserializer, в то время как WCF использует datacontractserializer, который намного лучше по производительности чем xmlserializer. Ключевыми недостатками xmlserializer при сериализации .NET типов в XML являются:
   * Только публичные поля или свойства .NET типов могут быть переведены в XML
   * Только классы, которые реализовывают интерфейс ienumerable могут быть переведены.
   * Классы, которые реализовывают idictionary, такие как Hashtable, не могут быть сериализованы.

Каждый элемент **behavior** определяется атрибутом **name** и обеспечивает либо предоставленное системой поведение (например <**throttling**>), либо пользовательское поведение. Если имя не задано, то элемент behavior будет соответствовать поведению по умолчанию для службы или конечной точки. Данный элемент определяет две дочерние коллекции с именами **endpointBehaviors** и **serviceBehaviors**. Каждая коллекция определяет элементы поведений, используемые конечными точками и службами соответственно. Каждый элемент поведения идентифицируется по уникальному атрибуту **name**. Начиная с версии .NET Framework 4 для привязок и поведений необязательно задавать имена.

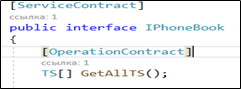
Поведения безопасности позволяют управлять учетными данными, проверкой подлинности, авторизацией и журналами аудита.Поведения можно использовать путем программирования или через конфигурацию.



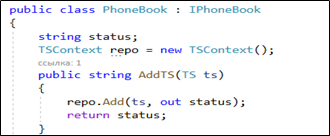
Порядок разработки:

создаём веб-сервис (приложение типа библиотека классов)

создаём интерфейс, атрибут [ServiceContract]показывает, что интерфейс определяет контракт службы в приложении WCF, [OperationContract] указывает, что метод определяет операцию, которая является частью контракта службы WCF



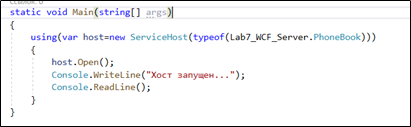
создаем класс, который наследуется от интерфейса и прописываем методы сервиса



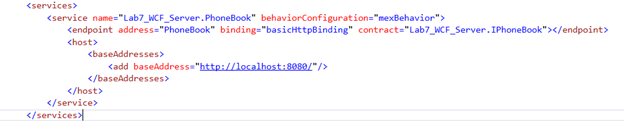
в файле app.config указываем имя сервиса, его конечные точки, адрес



создаём хост (в данном случае консольное приложение)



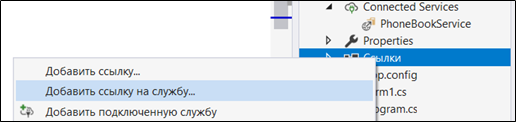
в файле App.config указываем имя сервиса, конечные точки и адрес хоста



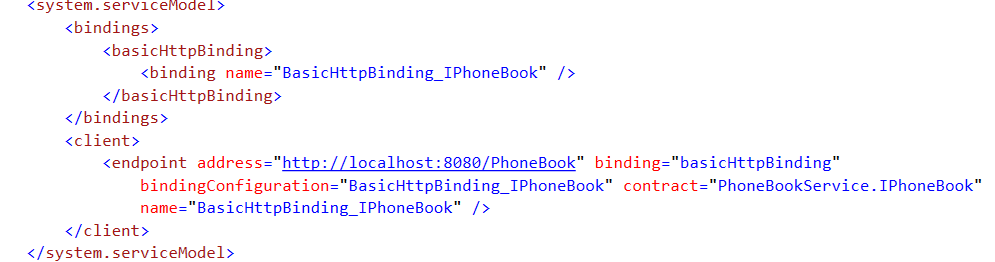
создаём клиент и используем прокси класс PhoneBookClient



подключаем сервис к клиенту (появятся Connected Services и подключённый сервис в них)



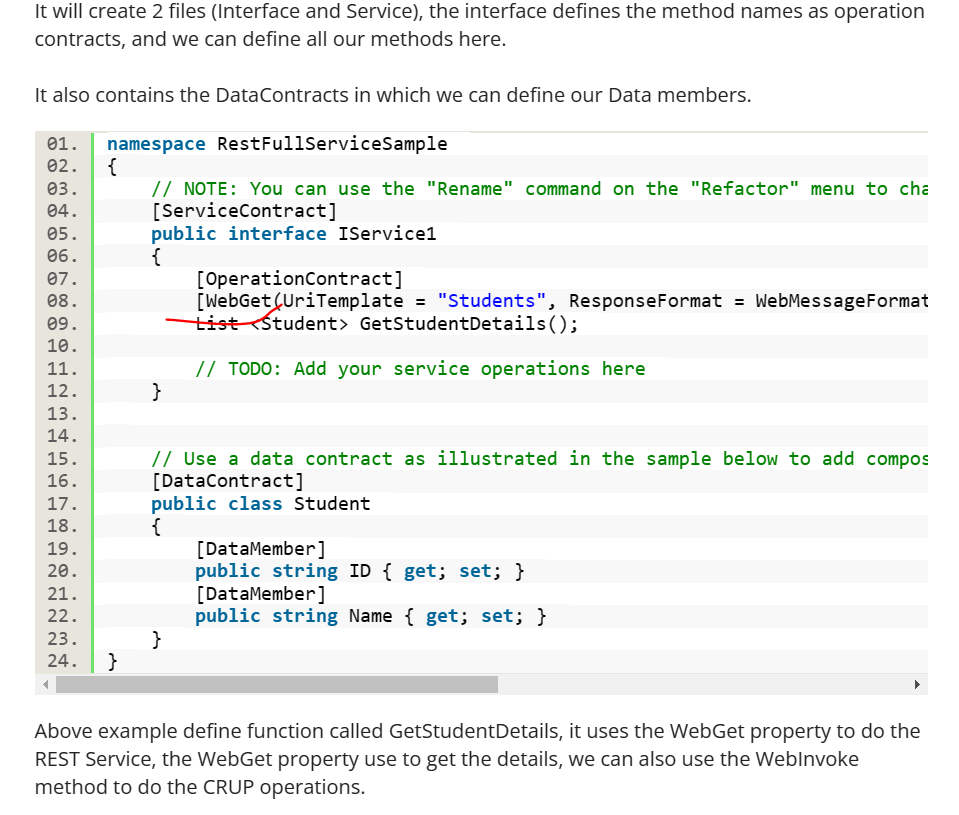
В файле app.config указываем биндинг и конечные точки для клиента

Для запуска проекта необходимо запустить вижлу в режиме администратора

**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)

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.

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.).

Название **Atom** применяется к двум связанным веб-стандартам: Atom Syndication Format и Atom Publishing Protocol.

1. **Atom Syndication Format** — это основанный на XML язык, используемый для описания каналов новостей;
2. **Atom Publishing Protocol** (AtomPub, APP) — это простой, основанный на HTTP протокол создания и обновления веб-ресурсов.

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

Протокол публикации Atom (также AtomPub, от англ. Atom Publishing Protocol) основан на HTTP и позволяет создавать, изменять и удалять ресурсы, собранные в коллекции на веб-сайте.

**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.

**Дополнительные возможности Atom**

**Модель содержания**

Atom представляет возможность собирать и представлять информацию в различных форматах, таких как обчный текст, HTML, XHTML, XML, двоичные данные в формате Base64 и ссылки на внешнее содержание в 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.

**Публикация информации**

Atom Publishing Protocol (AtomPub) — протокол прикладного уровня для публикации и редактирования веб-ресурсов. Он базируется на представлении данных в формате Atom, в качестве транспорта использует протокол HTTP. AtomPub может использоваться в rich-клиентах (например, в системах управления контентом) или самостоятельных приложениях для управления информацией на веб-сайтах.

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

В разных версиях аббревиатура RSS имела разные расшифровки:

* **Rich Site Summary** (RSS 0.9x) — расширенная сводка сайта;
* **RDF Site Summary** (RSS 0.9 и 1.0) — сводка сайта с применением инфраструктуры описания ресурсов (RDF — Resource Description Framework);
* **Really Simple Syndication** (RSS 2.x) — очень простое приобретение информации.

Исторически, разработкой формата занимались по меньшей мере четыре различные организации, в результате сложилась такая ситуация, что различные версии формата слабо совместимы, а то и совсем не совместимы, между собой.

**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.

**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.

**Дополнительные возможности RSS 2.0**

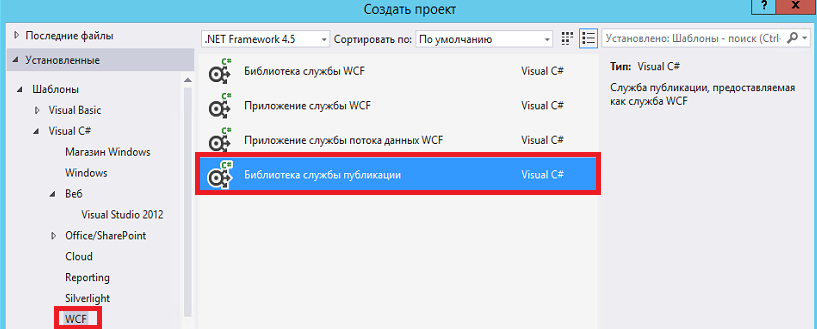
Согласно спецификации, версия 2.0 поддерживает следующие схемы в элементах <link> и <url>: http://, https://, news://, mailto: и ftp://. В предыдущих версиях поддерживались только http:// и ftp://.

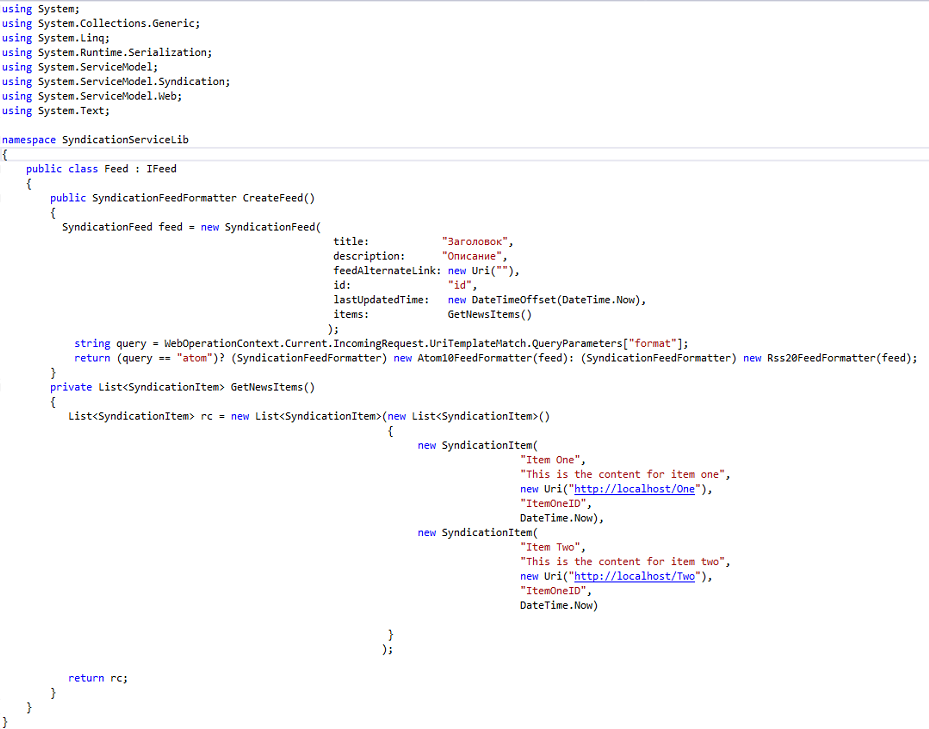
В RSS 0.91 различные элементы ограничивались до 500 или 100 символов. Кроме того, в <channel> не могло быть более 15 сообщений <item>. Ограничения на длины строк отсутствуют, начиная с RSS 0.92. Агрегаторы могут налагать свои ограничения по чтению, а генераторы могут иметь опции, ограничивающие количество <item> в канале, либо длины строк.

Заявленная в спецификации, но редко используемая возможность RSS 2.0 - отправка данных из ленты. Для этого стандарт представляет специальный элемент — <textInput>, атрибуты которого определяют текст отправляемого сообщения, адрес скрипта-обработчика, надпись на кнопке Submit, заголовок.

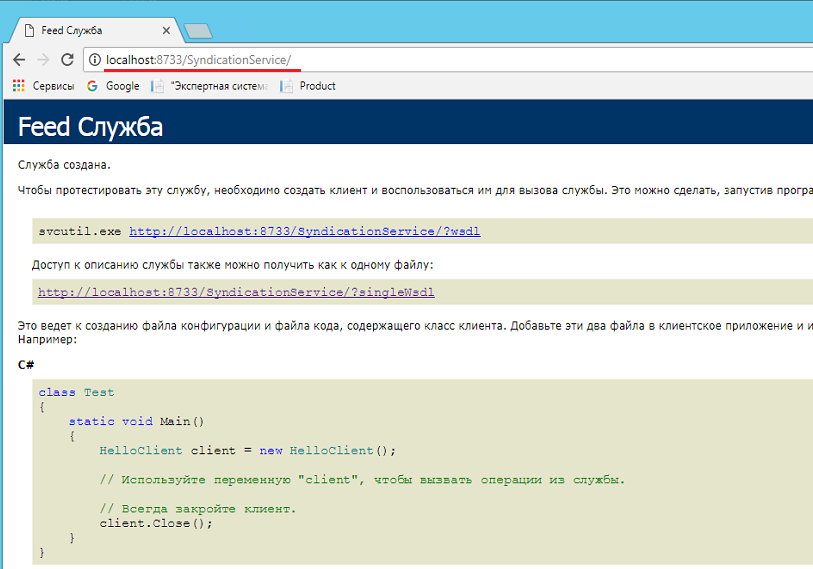
Расширение функциональности реализовано через поддержку модулей, определенных в [пространствах имен XML](http://www.w3.org/TR/REC-xml-names/). Канал RSS 2.0 может содержать элементы и атрибуты, не объявленные в спецификации, только если они определены в пространствах имен. В листинге 3 использована эта возможность — подключено пространство имен <xmlns:dc>:

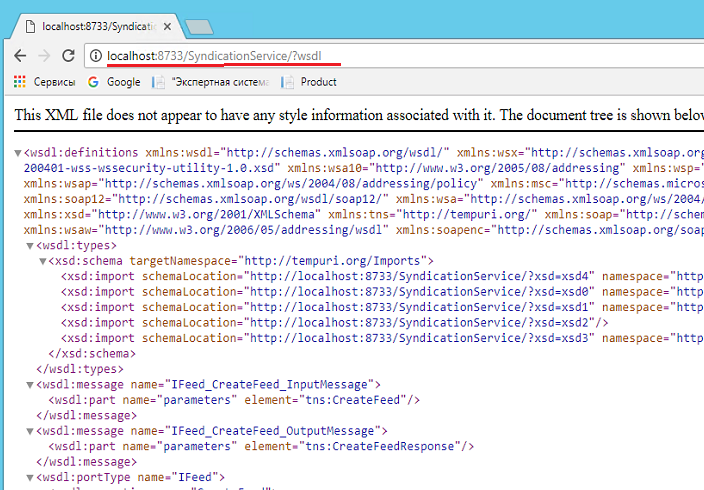
**ATOM/RSS:** реализация на платформе WCF .NET

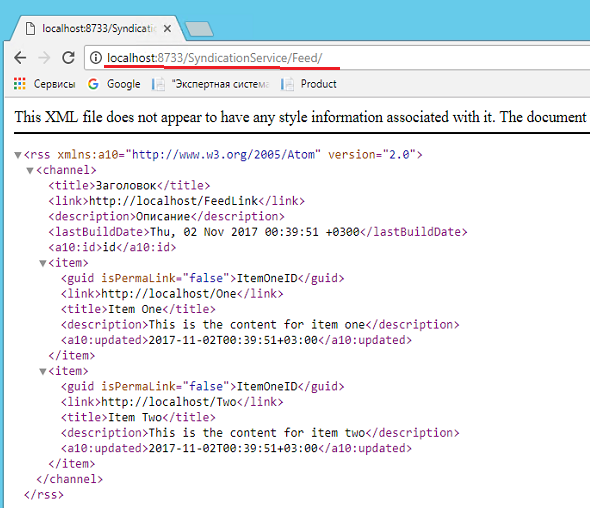
****

****

****

****

****

****

**8.WCF Data Services: протокол Open Data Protocol, возможности предоставляемые OData-интерфейсом, порядок разработки WCF Data Services, применение WCF 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 [REST](http://go.microsoft.com/fwlink/?LinkId=113919)-интерфейс. 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.

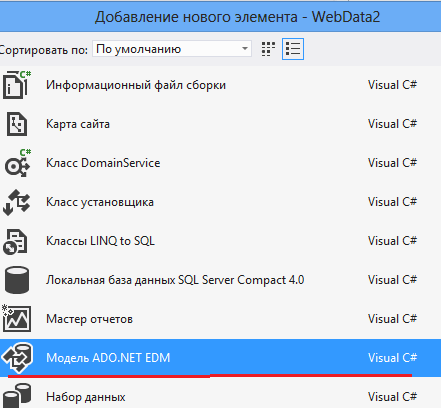
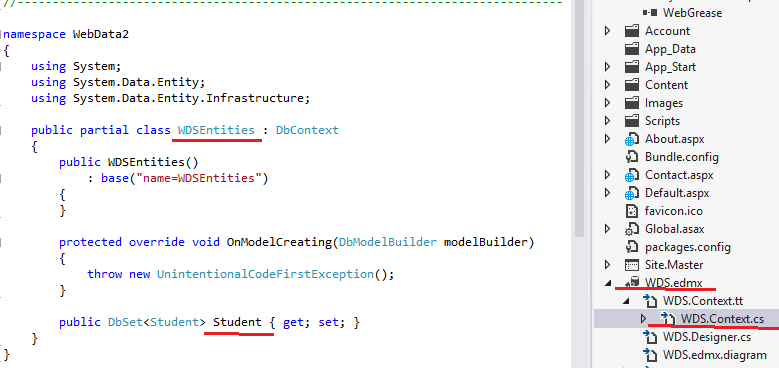
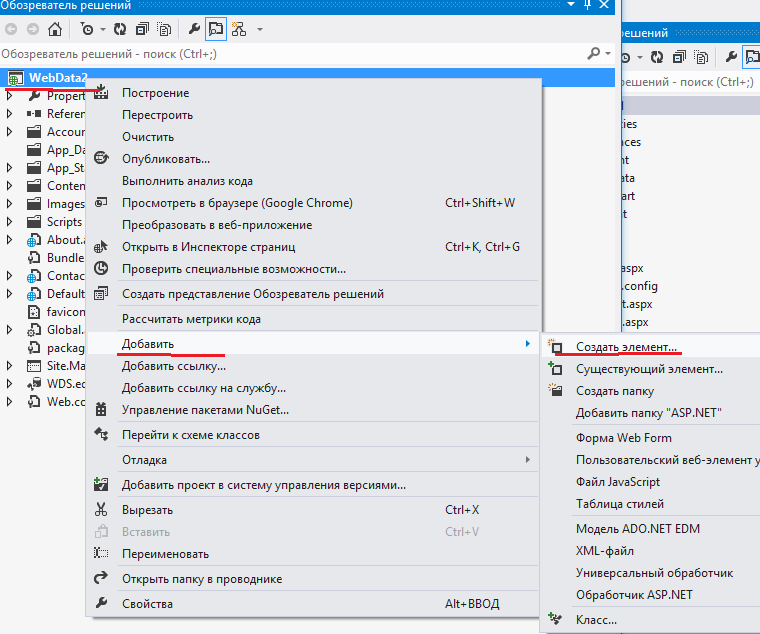
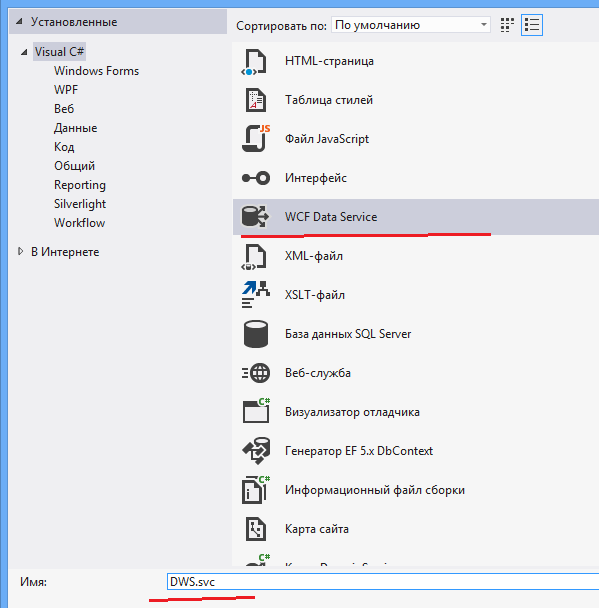
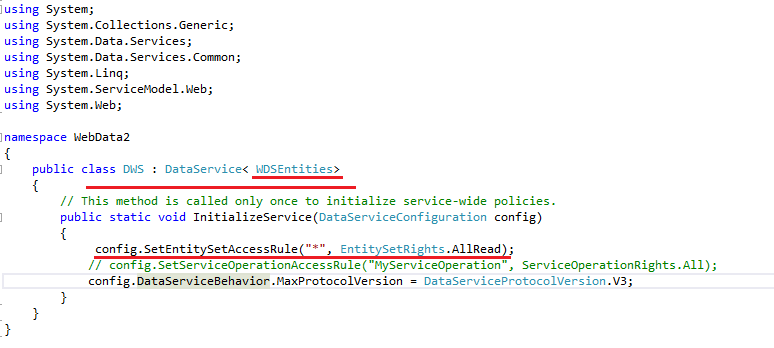
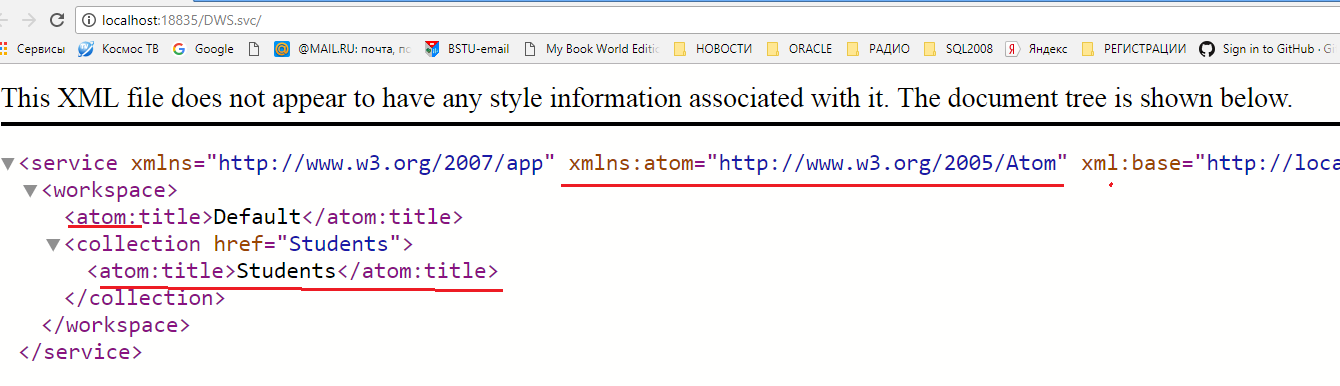
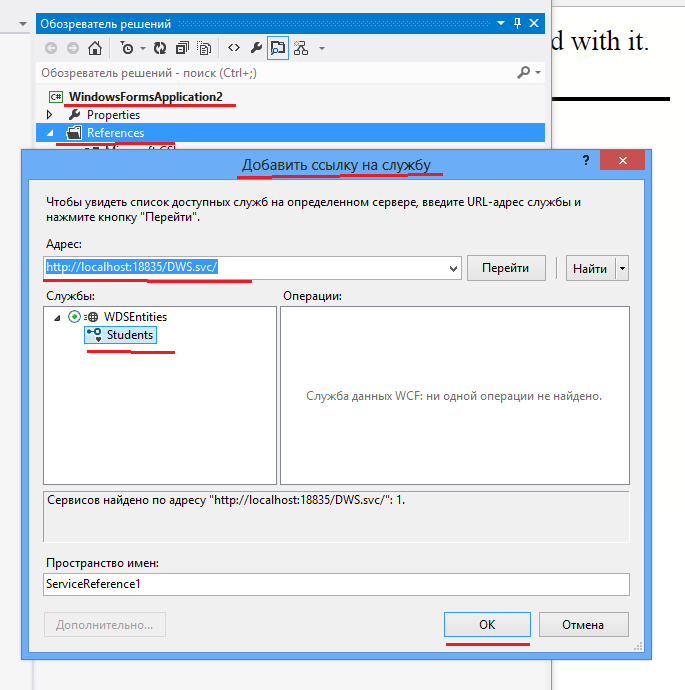
Средства Visual Studio упрощают создание служб на основе OData с использованием модели данных ADO.NET платформы Entity Framework. Можно также создать каналы OData на основе классов среды CLR и даже данных с поздним связыванием или нетипизированных данных.

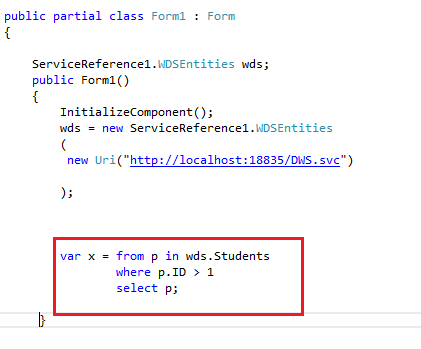
Датасервисес это одна из реализацией всф которая позволяет использовать протокол одата  
С помощью этого протокола можно реализовать доступ к реляционной бд через рест-интерфецс

Есть рест-интерфецс, есть правила, есть спец.язык запросов через рест-интерфецс и он равен по мощи с гуери

Делите, апдейт, и тд через опендата  
И можно dll делать  
Dml и dll через одата есть и можно из джава срикпта работать напрямую в бд

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.
4. 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.
5. 
6. 
7. 
8. 
9. 
10. 
11. **WCF DS: клиент**
12. 



Порядок:

1. Создать приложение ASP.NET
2. Создать модель EDMX
3. Добавить службу WCFDATAService
4. Написать в этой службе в кавычки название класса из EDMX
5. **JSON-RPC: определение JSON-RPC-сервиса, форматы запросов и ответов, обработка ошибок, пакеты запросов, реализация JSON-RPC на платформе Web API.**

**JSON-RPC** (сокр. от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *JavaScript Object Notation Remote Procedure Call* — JSON-вызов удалённых процедур) — [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.

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.

**Суть подхода достаточно примитивна**  
Запрос включает в себя 4 поля:

* jsonrpc — всегда будет “2.0”, указывает версию протокола.
* method — название метода (функции), который нужно вызвать.
* params — опциональное поле, нагрузка к вызову (аргументы функции).
* id — опциональное поле, уникальный идентификатор вызова. Если вы хотите получить значение от вызванной функции, то вы должны сгенерировать id на стороне клиента и при ответе вы сможете понять, на какой именно вызов пришел ответ, сопоставив id ответа.

Если вы не отправили id, то это означает, что ответ вас не интересует и от сервера вы ничего не получите. Такой вызов называется нотификацией.

**Ответ может иметь следующие поля**:

* jsonrpc — всегда будет “2.0”, указывает версию протокола.
* result — тело ответа (возвращаемое значение функции).
* id — уникальный идентификатор ответа. Он нужен для того, чтобы клиент мог сопоставить, на какой запрос он получил ответ.
* error — в случае ошибки вместо result, вы получите поле error, содержащее в себе code (код ответа: по протоколу их может быть шесть) и message (человекопонятное описание ошибки).

## Вишенка на торте — это Batch-запрос!

Это означает, что мы можем слепить несколько отдельных ajax запросов в один и отдать его серверу в виде массива запросов. А при получении по заданным id мы поймем, на какой запрос получили ответ.

## Небольшая тонкость!

Учтите, что по протоколу сервер не гарантирует последовательность элементов в ответе, например:

> [{id:1}, {id:2}, {id:3}]  
<-[{id:2}, {id:1}, {id:3},]

Будет считаться вполне нормальным ответом, т.к. время выполнения функций разное, и сервер заполняет массив ответами по мере их асинхронного выполнения.

Поэтому клиент сам должен промапить ответ и сопоставить id каждого элемента в ответе с запросом.

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

**SOA:** сервис-ориентированная архитектура; программное обеспечение представляет собой набор независимых компонентов, оснащенных стандартными интерфейсами (REST, RPC) для взаимодействия по стандартным протоколам. Технологии: DCOM, ASMX, WCF, CORBA, Java EE EJB. Альтернатива большим монолитным пиложениям.

**Микросервисы:** один из подходов к разработке SOA-приложений, основной принцип – сервисы должны быть легкими (сервисы маленькие и протоколы легковесные(быстрая передача)). Легкий сервис – сервис, который может быть переписан за 2 недели (Real Estate).

**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).

**Микросервисы:**  умные приемники и глупые каналы. Enterprise Service Bus (ESB) – не является приемлемым решением. Обычно применяется простой HTTP/REST, но необязательно. Есть легковесные шины для обеспечения асинхронности (MOM: RabbitMQ, ZeroMQ).

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

**Микросервисы:** ошибки проектирования, ***закон*** ***Мелвина Конвея*** «Любая организация, которая проектирует какую-то систему, получит решение, структура которого копирует структуру коммуникаций этой организации».

**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 подходящего инструментария для разработки микросервисов (дает возможность привлекать разные команды).

**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:** синхронные вызовы сервисов считаются опасными; как правило, синхронные вызовы не применяются или «один синхронный запрос к сервису на один пользовательский запрос».

**Микросервисы:** распределенное управление данными. Подход ***Polyglot Persistence***.

Основной принцип работы микрослужб заключается в том, **что каждая служба управляет собственными данными. Две службы не могут использовать одно и то же хранилище данных. Наоборот, каждая служба отвечает за собственное хранилище данных, которое напрямую недоступно для других служб**.

Это **позволяет устранить непреднамеренную взаимозависимость между службами**, которая может возникнуть, когда службы совместно используют одни и те же базовые схемы данных. Если в схему данных вносятся изменения, их нужно согласовать в каждой службе, которая зависит от этой базы данных. Изолируя хранилище данных каждой службы, мы можем ограничить область изменений и сохранить гибкость полностью независимых развертываний. Кроме того, у каждой микрослужбы могут быть собственные модели данных, запросы или шаблоны операций чтения и записи. При использовании общего хранилища данных для каждой команды ограничивается возможность оптимизировать хранилище данных для какой-либо службы.

Этот подход естественным образом приводит к [Polyglot Persistence](https://martinfowler.com/bliki/PolyglotPersistence.html) — использованию нескольких технологий хранения данных в одном приложении. Для работы одной службы могут требоваться возможности схемы при чтении, свойственные базе данных документов. Для другой — целостность данных, предоставляемая реляционной СУБД. Каждая команда может выбрать подходящий вариант для своей службы.

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

**Микросервисы:** проектирование под отказ; разработка систем мониторинга работоспособности сервисов; разработка систем протоколирования работы сервисов; консоль сервиса: статус, тестирование, получение различных метрик.

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

**Микросервисы:** требуется ***DevOps*** (Development & Operation)- набор технологий нацеленных на интеграцию процессов разработки и информационно-техническому обслуживанию. 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.

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

***Непрерывная интеграция (Continuous Integration)****— это практика разработки программного обеспечения, которая заключается в слиянии рабочих копий в общую основную ветвь разработки несколько раз в день и выполнении частых автоматизированных сборок проекта для скорейшего выявления и решения интеграционных проблем*

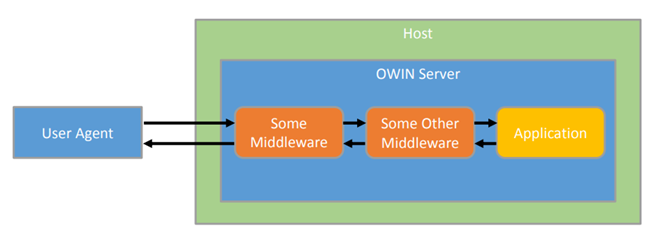
Бизнес значение

* Ускорение поставки (Accelerate Delivery) – достигается тем, что мы сразу же узнаем об ошибке сборки и, соответственно, можем быстрее начать её исправлять.
* Повторяемость (Repeatability) — весь процесс повторяем, то есть если никаких изменений не произошло, то и сборка будет так же успешна (или не успешна). Нет такой проблемы как то, что у одного разработчика все собирается, а у другого – нет.
* Оптимизация ресурсов (Optimized Resources) — нет необходимости вручную запускать сборку, на компьютере человека или билд сервере, нет необходимости готовить сборку – выкачивать исходники из source control и т.п.

**Непрерывная доставка (CD)** - это процесс сборки, тестирования, настройки и развертывания из сборки в производственную среду.

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

OWIN  
Open Web Interface for NET. OWIN – интерфейс взаимодействия сервера, хоста и самого приложения. Основная цель OWIN отделить web-сервер от серверного приложения. Можно разрабатывать отдельно сервер и приложение.  
Nancy служит для разработки самого приложения. И оно взаимодействуют с хостом и сервером с помощью OWIN

Nancy - фреймворк с помощью которого можно разработать ту часть архитектуры, которая называется приложение и он взаимодействует с сервером и хостом с помощью OWIN  
OWIN: архитектура  
  
OWIN: Host – приложение-процесс операционной системы, управляющий жизненным циклом OWIN: 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  
OWIN: middleware – подключенные компоненты (модули), предназначенные для обработки запросов

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.

**Шаг 1: Создаем консольное приложение на C#**Разумеется, это можно сделать и на другом языке для платформы .NET.

**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. 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"

};

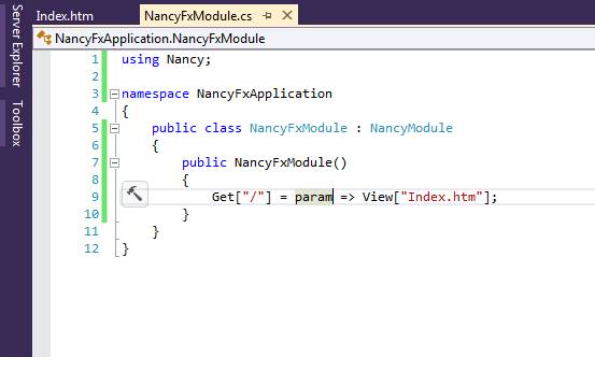
public DinosaurModule()

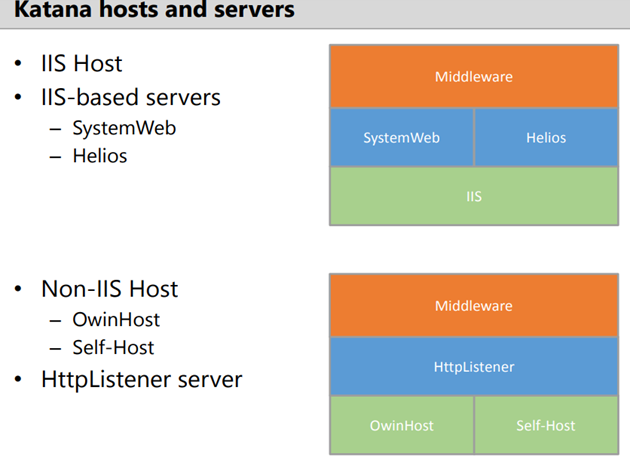
{

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

}

}

- В конструкторе создаем делегат, при помощи которого будем хэндлить запросы к приложению  


Katana  
Katana: Microsoft-реализация OWIN сервера (говорят проект Katana). В качестве Host можно использовать IIS или self-hosting. Кроме того, позволять подключать модули middleware; предоставляет набор классов для работы с сервером и механизм подключения приложения к серверу  
Katana: виды конфигурации  
  
Katana: фреймворки приложений поддерживающие OWIN/Katana  
 -WebAPI;  
 -SignalR;  
 -Nancy, Nancyfx;  
 -FubuMVC;  
 -Simple.Web;  
 -RevwnDB;  
 -Thinktecture IdentityServer v3.  
  
  
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.

***По словам смелова:***

Ненси это один из способов реализации сервисов на основе АСП.нет

Позволяет создавать отдельные модули, которые независимы друг от друга  
Аналог оэсджэай(?)

Меняются модули через остановку сервера

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

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

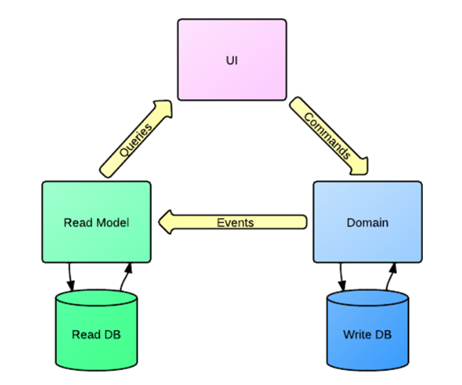
**ES:** каждое событие имеет имя.

**ES:** произошедшие события неизменны.

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

**ES:** обычно применяется в архитектуре CQRS для реализации WriteDB

**CQRS**: разделение ответственности на команды и запросы ; CQS – command-query separation(Б. Майер) = принцип императивного программирования (команда-память-команда-память, ассемблер – типичный императивный язык). Основной принцип – команда-чтение не вызывает побочного эффекта(не изменяет состояние объекта).

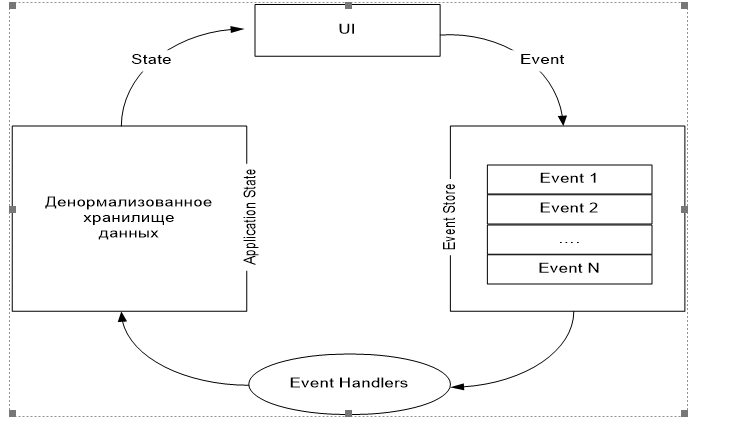


Event Storing рассказать что это такое, развитие подхода разделения ответственности, есть сервисы которые предназначены для выполнения изменений в бд , обычно используются event store, которые сохраняют не данные, а события которые возникают на стороне клиента и эти события сохраняются в специальных 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.

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.

**EventStore**: функциональная база данных с комплексной обработкой событий в JavaScript.Собственно говоря Event Store это БД которая предназначена для хранения событий.

Пример реализации:

* + - 1. Качаем .Net Core
      2. Запускаем exe EventStore при запуске можно посмотреть какие юзеры создались и на каком порту работает это приложение.
      3. Скачиваем новый нугет архив, в котором будут лежать новые для на классы для работы с евенсторем
      4. Далее класс, который имплементит NancyModule, в котором

В красной рамке:

1) Просто создаем параметры для сохранения

2) Устанавливаем соединение с EventStore, могут использоватся разные протоколы, в данном случае tcp. Можно http, admin:changeit - логин и пароль от eventstore стандартные.

3)Устанавливаем соединение

4) Создаем новый ивент и заполняем инфой(id,eventtype-любое имя, отправлять джсоном-true, какая-то информация )

5) Отправляем и закрываем соединение

***По словам смелова:***

Это подход(!) Который применяется чаще всего в соа-приложениях  
И чаще ещё в микросервисах  
Когда запоминается в бд- ивентсторинг не изменение, а само событие

Когда запоминается в бд- ..  
Есть спецбд для этого, в которых запоминается н еизменение данных в бд, а само событие  
Происходит серверное событие,оно фиксируется и потом уже в Асинхронном режиме это серверное событие преобразовывается в запрос к бд и проводится изменение в этой бд  
Есть 2 бд; одна запоминает события, а другая только для чтения  
Это и есть этот подход

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

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

[Что такое Docker: для чего он нужен и где используется - Блог компании Селектел (selectel.ru)](https://selectel.ru/blog/what-is-docker/)

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**.

Контейнеры — хорошая альтернатива аппаратной виртуализации. Они **позволяют запускать приложения в изолированном окружении**, но при этом потребляют намного меньше ресурсов.

**Контейнеры** — это способ стандартизации развертки приложения и отделения его от общей инфраструктуры. Экземпляр приложения запускается в изолированной среде, не влияющей на основную операционную систему.

**Docker** — это платформа для разработки, доставки и запуска контейнерных приложений. Docker позволяет создавать контейнеры, автоматизировать их запуск и развертывание, управляет жизненным циклом. Он позволяет запускать множество контейнеров на одной хост-машине.

Контейнеризация похоже на виртуализацию, но это не одно и то же. Виртуализация запускает полноценный хост на гипервизоре со своим виртуальным оборудованием и операционной системой. При этом внутри одной ОС можно запустить другую ОС. В случае контейнеризации процесс запускается прямо из ядра основной операционной системы и не виртуализирует оборудование. Это означает, что контейнеризованное приложение может работать только в той же ОС, что и основная. Контейнеры не виртуализируют оборудование, поэтому потребляют меньше ресурсов.

Преимущества использования контейнеров

* Docker решает проблемы зависимостей и рабочего окружения

Контейнеры позволяют упаковать в единый образ приложение и все его зависимости: библиотеки, системные утилиты и файлы настройки. Это упрощает перенос приложения на другую инфраструктуру.

Например, разработчики создают приложение в системе разработки — там все настроено, приложение работает. Когда оно готово, его нужно перенести в систему тестирования, а затем в продуктивную среду. Если в одной из них нет нужной зависимости, приложение не будет работать. Программистам придется отвлечься от разработки и совместно с командой поддержки разобраться в ситуации.

* Изоляция и безопасность

Контейнер — это набор процессов, изолированных от основной операционной системы. **Приложения работают только внутри контейнеров и не имеют доступа к основной операционной системе.** Это повышает безопасность приложений:они не смогут случайно или умышленно навредить основной системе. Если приложение в контейнере завершится с ошибкой или зависнет, это никак не затронет основную ОС.

* Ускорение и автоматизация развертывания прилаг и масштабируемость

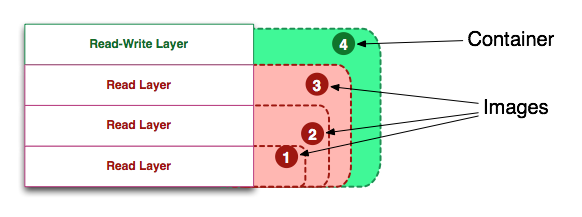
Контейнеры упрощают развертывание приложений. *В классическом подходе* для установки программы нужно совершить несколько действий: выполнить скрипт, изменить файлы настроек и так далее. В этом процессе не исключена вероятность человеческой ошибки: пользователь запустит скрипт два раза, перепутает последовательность или что-то не поймет. **Контейнеры позволяют полностью автоматизировать этот процесс, так как включают в себя все нужные зависимости и порядок выполнения действий.**

Также контейнеры упрощают развертывание на нескольких серверах. В классическом подходе для того, чтобы развернуть одно и то же приложение на нескольких машинах, нужно будет повторять одни и те же действия. Контейнеры избавляют от этой рутинной работы и позволяют автоматизировать развертывание.

* Контейнеры приближают к микросервисной архитектуре

Контейнеры хорошо вписываются в микросервисную архитектуру. **Это подход к разработке, при котором приложение разбивается на небольшие компоненты, по возможности независимые.** Обычно противопоставляется монолитной архитектуре, где все части системы сильно связаны друг с другом.

Это позволяет разрабатывать новую функциональность быстрее, ведь в случае с монолитной архитектурой изменение какой-то части может затронуть всю остальную систему.



АРХИТЕКТУРА

### Docker daemon

Это некоторый резидентный **процесс, который запущен на хост-машине постоянно**. Он владеет всей инфраструктурой, а также **предоставляет интерфейс взаимодействия с контейнерами**, *включающего создание и удаление, запуск и остановку.*

### Docker client (клиент)

Это **интерфейс командной строки для управления Docker daemon**. Мы пользуемся этим клиентом, когда создаем и разворачиваем контейнеры, а *клиент отправляет эти запросы в Docker daemon*.

### Docker image (образ)

Это **неизменяемый файл (образ), из которого разворачиваются контейнеры.** Приложения упаковываются именно в образы, из которых потом уже создаются контейнеры. В технической литературе можно также встретить описание image как шаблона запуска процесса.

Приведем аналогию на примере установки операционной системы. В дистрибутиве (образе) ОС есть все, что необходимо для ее установки. Но этот образ нельзя запустить, для начала его нужно «развернуть» в готовую ОС. Так вот, дистрибутив для установки ОС — это образ, а установленная и работающая ОС — это контейнер. Но контейнеры обычно разворачиваются одной командой — это намного проще и быстрее, чем установка ОС.

### Docker container (контейнер)

Это уже **развернутое** из образа **и работающее приложение.**

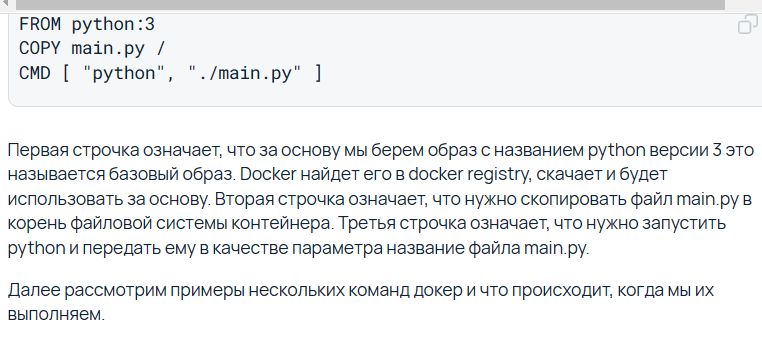
### Docker Registry

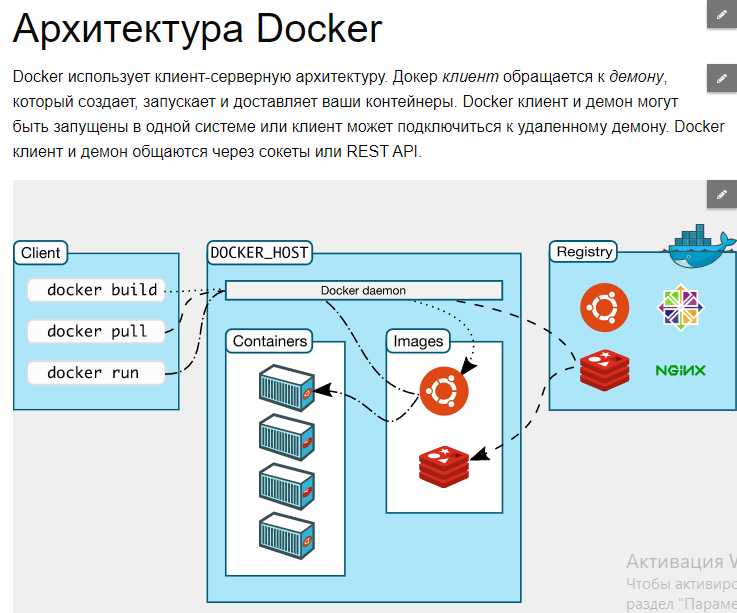
Это **репозиторий с образами**. Разработчики создают образы своих программ и выкладывают их в репозиторий, чтобы их можно было скачать и воспользоваться ими. Распространенный публичный репозиторий — [Docker Hub](https://hub.docker.com/" \t "_blank). В нем собраны образы множества популярных программ или платформ: базы данных, веб-серверы, компиляторы, операционные системы и так далее. Также можно создать свой приватный репозиторий, например внутри компании. Разработчики будут размещать там образы, которые будут использоваться всей компанией.

### Dockerfile

Dockerfile — **это инструкция для сборки образа.** Это **простой текстовый файл**, *содержащий по одной команде в каждой строке.* В нем указываются все программы, зависимости и образы, которые нужны для разворачивания образа.

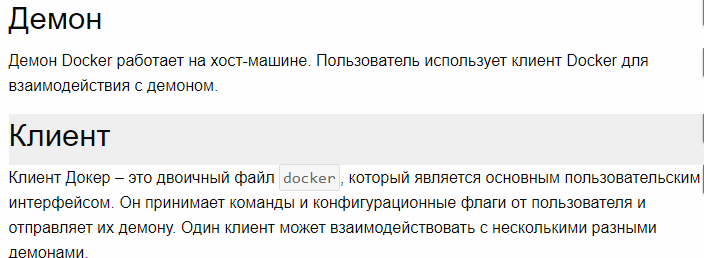
Для примера рассмотрим Dockerfile, который мы будем использовать далее в этой статье чтобы развернуть собственное приложение:





Все эти команды выполняются в Docker client, который отправляет их в Docker daemon:

* Команда **docker build** (зеленая стрелка) читает dockerfile и собирает образ.
* Команда **docker pull** (красная стрелка) скачивает образ из docker registry. По умолчанию docker скачивает образы из публичного репозитория Docker Hub. Но можно создать свой репозиторий и настроить докер, чтобы он работал с ним.
* Команда **docker run** (черная стрелка) берет образ и запускает из него контейнер.



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

**14.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.

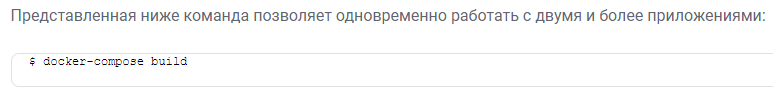
Compose превосходен для разработки, тестирования и настройки среды, а также непрерывной интеграции. Вы этом разделе вы можете узнать более подробно о [решаемых задачах](https://dker.ru/docs/docker-compose/overview-of-docker-compose/#common-use-cases).

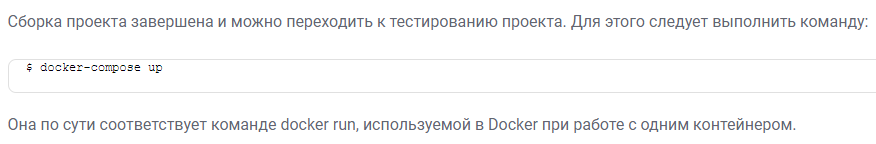
Использование Compose обычно разделяется на три этапа:

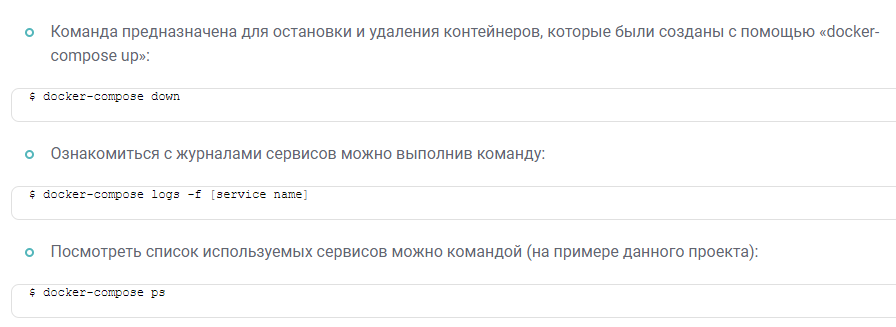
1. Определение окружения вашего приложения в Dockerfile, это можно сделать в любом месте.
2. Определение сервисов из которых будет состоять ваше приложение в docker-compose.yml, в последствии они смогут быть запущены все вместе в изолированном окружении.
3. И наконец, выполнение команды docker-compose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

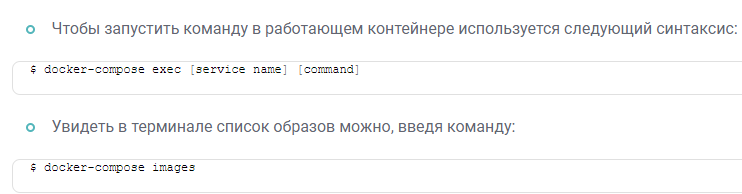
Compose имеет команды для управления всем жизненным циклом приложения:

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









docker swarm, docker stack cuberneites