**Оглавление**

[1. SOA: определение, свойства, стандарты, спецификации, интерфейсы, специальные компоненты, способы клиент-серверного взаимодействия, платформы для разработки. 2](#_Toc155188986)

[2. REST API: определение, форматы передачи данных, HATEOAS, общепринятые правила REST API, платформы для разработки сервисов. 6](#_Toc155188987)

[3. SOAP: определение, структура Envelop-сообщений, пространства имен, роли, принципы расширения. 7](#_Toc155188988)

[4. WSDL: определение, стандарты и версии, концептуальная модель, пространства имен, структура WSDL-документа. 9](#_Toc155188989)

[5. ASMX: определение ASMX-сервиса, порядок разработки, принципы применения, утилита WSDL.EXE. 11](#_Toc155188990)

[6. WCF: определение WCF-сервиса, коммуникационная модель, WCF-контракты, WCF-хостинг, конечные точки, стандартные привязки, основные отличия от ASMX-сервисов, поведение и безопасность WCF-сервиса, порядок разработки WCF/RPC и WCF/REST-сервисов, разработка WCF-сервиса с несколькими конечными точками.???? 13](#_Toc155188991)

[7. WCF Syndication Services: стандарты ATOM, RSS, порядок разработки WCF Syndication Service. 16](#_Toc155188992)

[8. WCF Data Services: протокол Open Data Protocol, возможности предоставляемые OData-интерфейсом, порядок разработки Data Services, применение Data Services. 17](#_Toc155188993)

[9. JSON-RPC: определение JSON-RPC-сервиса, форматы запросов и ответов, обработка ошибок, пакеты запросов, реализация JSON-RPC на платформе Web API. 18](#_Toc155188994)

[10. ASP.NET CORE Nancy: интерфейс OWIN, архитектура приложения, принцип разработки сервиса. 19](#_Toc155188995)

[13. Event Storing: назначение, принципы применения, примеры реализации. 28](#_Toc155188996)

[14. Микросервисы: микросервисная архитектура, определение микросервиса, основные принципы разработки микросервиса, паттерны разработки, DevOps для микросервисов, оркестровка и хореография микросервисов. 30](#_Toc155188997)

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

**SOA (Service-oriented architecture, сервис-ориентированная архитектура)** – архитектура приложения, которая состоит из отдельных компонентов, каждый компонент называется сервис. Сервис – приложение, которое предоставляет интерфейс или API, сетевой интерфейс для других приложений. Если наше приложение состоит из таких вот компонентов, каждый из которых предоставляет интерфейс, то это и называется сервис-ориентированная архитектура, то есть не обязательно это веб-интерфейс, это могут быть и TCP-интерфейсы, а если у нас только веб-интерфейсы, то это веб-приложение (веб-сервис или веб-сервисная архитектура).

Ну и там есть такие стандартные части: шина, брокер, реестр сервисов, вот всё что мы с вами проходили, какие там есть дополнительные компоненты, они фактически в той или иной степени повторяют эти вот, у всех есть там какие-то реестры, есть очереди, можно называть их шинами, фактически разное воплощение одного и того же. Только если мы возьмём с вами какие-нибудь облачные системы, там вот реально есть реестры, так называется **Enterprise Service Bus**, шины, такие интерфейсы есть. Если же когда мы строим с вами какое-то приложение такой архитектуры, есть компоненты, которые можно применять и различные там есть фреймворки, есть различные приложения, с помощью которых мы можем с вами организовать своего рода такие стандартные компоненты.

SOA: стиль архитектуры информационной системы, который позволяет создавать приложения путем комбинации слабосвязанных распределенных компонент.

SOA: основные свойства

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

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

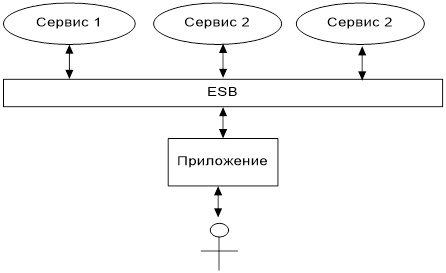
SOA: спецификации второго уровня W3C (WS\*):

* WS-Policy, WS-PolicyAttachment, WS-PolicyAssertion (описание политик web-сервиса);
* WS-Addressing (механизм адресации web-сервиса);
* 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).

SOA: интерфейсы – REST, SOAP, JSON-RPC

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

SOA: ESB

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

протоколирование.

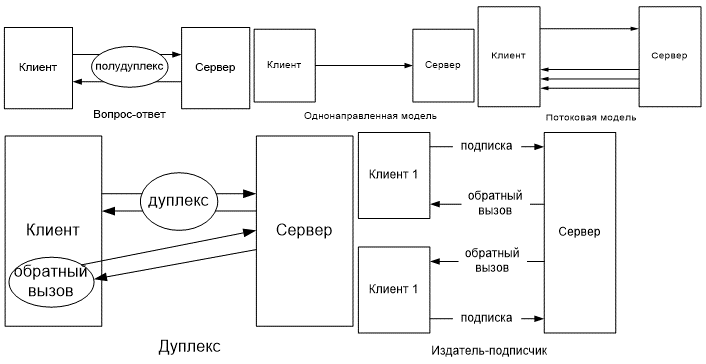
SOA: специальные компоненты (как правило часть ESB)

* SOA Registry – реестр сервисов, информация о сервисах и их интерфейсах.;
* Workflow Engine ­­­­­– позволяет разработать новый сервис на основе последовательного выполнения нескольких сервисов.;
* Service Broker – позволяет трансформировать запрос пользователя в системе в запуск и координированную работу.;
* SOA Supervisor – управления и мониторинг других служебных сервисов.

SOA: Identity Service - сервис аутентификации, предназначен для реализации SSO (Single Sign On), протоколы токен-авторизации (JWT), OpenID Connect, OAuth2. IDaaS. (AWS IAM).

Платформы - JavaEE, .Net, Node.js

Способы клиент-серверного взаимодействия



RPC означает удаленный вызов процедур (Remote Procedure Call). Это протокол взаимодействия между компьютерами в сети, который позволяет одному компьютеру вызывать процедуры (функции, методы) на удаленном компьютере, как если бы они выполнялись локально. Иными словами, RPC позволяет программам работать с процедурами на удаленных серверах, скрывая сложности взаимодействия по сети.

Процесс выглядит примерно так:

1. Клиент отправляет запрос на сервер, чтобы выполнить определенную процедуру.
2. Сервер получает запрос, выполняет процедуру и отправляет результат обратно клиенту.

Существует несколько протоколов для реализации RPC, таких как XML-RPC, JSON-RPC, и протоколы, основанные на HTTP, такие как SOAP (Simple Object Access Protocol) и REST (Representational State Transfer). RPC широко используется в распределенных системах и клиент-серверных приложениях для упрощения взаимодействия между компонентами, даже если они распределены по сети.

Основные отличия между REST и RPC:

1. **Архитектурный стиль:**
   * **REST:** REST является архитектурным стилем, который базируется на ограничениях, таких как использование ресурсов (например, URL), представление данных (например, JSON или XML) и стандартных методов HTTP (например, GET, POST, PUT, DELETE).
   * **RPC:** RPC, с другой стороны, является протоколом, ориентированным на вызов удаленных процедур. Он фокусируется на вызове функций или методов, которые находятся на удаленном сервере.
2. **Транспорт протокола:**
   * **REST:** REST часто использует HTTP как транспортный протокол, что делает его простым для взаимодействия с веб-сервисами.
   * **RPC:** RPC может использовать различные транспортные протоколы, такие как HTTP, JSON-RPC, XML-RPC, gRPC и другие.
3. **Уровень абстракции:**
   * **REST:** REST ориентирован на ресурсы и операции с ними. Клиенты манипулируют ресурсами, представленными URL, и используют стандартные методы HTTP для выполнения операций (например, получение, создание, обновление, удаление).
   * **RPC:** RPC скрывает детали передачи данных и позволяет вызывать функции или методы удаленного сервиса так, как если бы они были локальными.
4. **Состояние:**
   * **REST:** RESTful сервисы часто стремятся быть безсостоятельными (stateless), что означает, что каждый запрос от клиента содержит всю необходимую информацию для выполнения операции.
   * **RPC:** RPC может быть состоятельным или безсостоятельным в зависимости от конкретной реализации.
5. **Кэширование:**
   * **REST:** REST поддерживает кэширование данных, что может повысить производительность и эффективность.
   * **RPC:** RPC не предоставляет встроенной поддержки для кэширования.
6. **Метаданные:**
   * **REST:** RESTful сервисы часто включают метаданные в ответах (например, заголовки HTTP), что упрощает взаимодействие с клиентами.
   * **RPC:** Метаданные в RPC могут быть включены в самом сообщении.

Отличия сервиса от микросервиса:

**Размер и сложность:**

* + **Сервисы:** Могут быть крупными и монолитными, предоставляя различные функциональности в рамках одного приложения.
  + **Микросервисы:** Маленькие, специализированные и автономные, часто выполняющие ограниченный набор функций.

1. **Зависимости:**
   * **Сервисы:** Могут быть тесно связанными и зависеть друг от друга.
   * **Микросервисы:** Строятся с учетом минимизации зависимостей между собой.
2. **Управление данными:**
   * **Сервисы:** Часто используют общие базы данных и ресурсы.
   * **Микросервисы:** Могут иметь собственные базы данных, что способствует изоляции данных и уменьшению влияния на другие микросервисы.
3. **Разворачивание и масштабирование:**
   * **Сервисы:** Могут разворачиваться как единое приложение и масштабироваться как целое.
   * **Микросервисы:** Могут быть независимо развернуты и масштабированы, что обеспечивает гибкость в управлении ресурсами.
4. **Сбои и отказоустойчивость:**
   * **Сервисы:** Отказ одного компонента может повлиять на всю систему.
   * **Микросервисы:** Изоляция микросервисов позволяет уменьшить влияние сбоев на всю систему.
5. REST API: определение, форматы передачи данных, HATEOAS, общепринятые правила REST API, платформы для разработки сервисов.

(02:20)

REST API – архитектурный стиль. Веб-приложение с REST API – это веб-приложение с набором конечных точек (в виде URI) и правилами работы с этими точками. Форматы для передачи: JSON и XML. Как передаются параметры: либо в строке (query), либо через тело в виде JSONа или XMLа.

HATEOS – это механизм, с помощью которого мы можем в ответе отправлять ещё дополнительные ссылки на другие сервисы, чтобы можно было уточнить ответ. Он нужен для того, чтобы клиента отвязать от месторасположения отдельных средств … настраивается на стороне клиента на необходимые URI.

На каких платформах можем разрабатывать REST API: Web API, Servlet, на обыкновенном http-handler, с помощью всего этого мы можем с вами сделать REST API, *но есть стандарт…. Web API* *(по сути повторил тоже самое)* Web API, или интерфейс программирования приложений для веба, представляет собой набор соглашений и протоколов, которые позволяют различным программам взаимодействовать друг с другом через интернет. Он предоставляет способ коммуникации между разными программами, позволяя им обмениваться данными и функциональностью.

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

Основные методы управления ресурсами: GET, POST, PUT, DELETE.

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 клиенте апплетов или сценариев для расширения его функциональности.

Общепринятые правила

* + Общий префикс для всех ресурсов сервиса … /API/…
  + 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.

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

(04:55)

Что такое SOAP? – сетевой **протокол** прикладного уровня.

Какие могут использоваться транспорты для перемещения? – HTTP, TCP, SMTP.

Если спрошу, как Envelop в http передавать (перемещать)? – в теле.

Как указать, что у вас в теле находится Envelop? – с помощью заголовков, content-type: SOAP + XML.

Как его вписать в TCP? – как придумаете, так и впишите, должны сами придумать какое-то первое сообщение, в котором вы укажите, что теперь идёт SOAP-сообщение, по-другому там никак не сделать, т.е. нет там никаких заголовков, ничего такого нету. Все заголовки что там есть, все сообщения заняты информацией. Поэтому самому нужно придумать протокол (протокол прикладного уровня), который будет говорить, что у вас идёт SOAP-сообщение. Т.е. надо придумать какую-то последовательность данных, при котором сервер и клиент смогут договориться, что будут SOAP-сообщениями меняться. Другого там способа нет.

Какая информация передаётся в SOAPе? – заголовки есть, есть там тело (body), сообщения об ошибках.

Какое основное замечательное свойство есть у SOAPа? – может расширяться (с помощью namespace(ов) вы можете расширять его до бесконечности). На базе SOAP вы можете строить собственные протоколы: взять SOAP за основу, добавить свои namespace(ы) и на основе его делать свои собственные протоколы, можете делать всё, что хотите, можете перемещать в рамках SOAP-сообщений. Но на конечных точках вы должны предусматривать соответствующую обработку своих собственных namespace(ов).

Протокол обмена структурированными сообщениями в распределённой вычислительной среде.

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

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

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

При работе с SOAP важно учитывать следующие моменты:

* Элемент Envelope является обязательной частью SOAP сообщения.
* Внутри каждого элемента Envelope должен быть один элемент Body.
* Элемент Envelope определяется с помощью пространства имён ENV и элемента Envelope.
* Элемент Header даёт нам возможность определять дополнительные свойства приложения. Например, мы можем использовать его для определения сигнатуры сервисов, защищённых паролем.

**Роль** – набор правил, определяющих поведение узла, задается атрибутом env:role.

Тристандартные роли для узла (нет роли, промежуточный узел, конечный получатель), роль может быть пользовательской (задаваться собственным URI).

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

Атрибут env:mustUnderstand:

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

= false, если обработчик не разобрал блок, то ошибка может быть проигнорирована.

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

Атрибут env:relay:

= true, если обработчик не разобрал блок, то блок передается дальше.

= false, если обработчик не разобрал блок, то блок не передается дальше.

**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.

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

(08:25)

Что такое WSDL? – язык (L – language).

Для чего он? – для описания сервисов.

Для чего он используется? – для того, чтобы каким-то формальным способом описать сервис; для того, чтобы можно было в автоматическом режиме сгенерировать клиента.

Что входит в описание WSDL?

Какая последняя версия? – 2.0. Но пока в основном используется версия 1, они фактически взаимозаменимы эти версии.

Какая информация лежит? – типы данных, типы сообщений, порты (… методы) и предаётся binding (как связываться с транспортом).

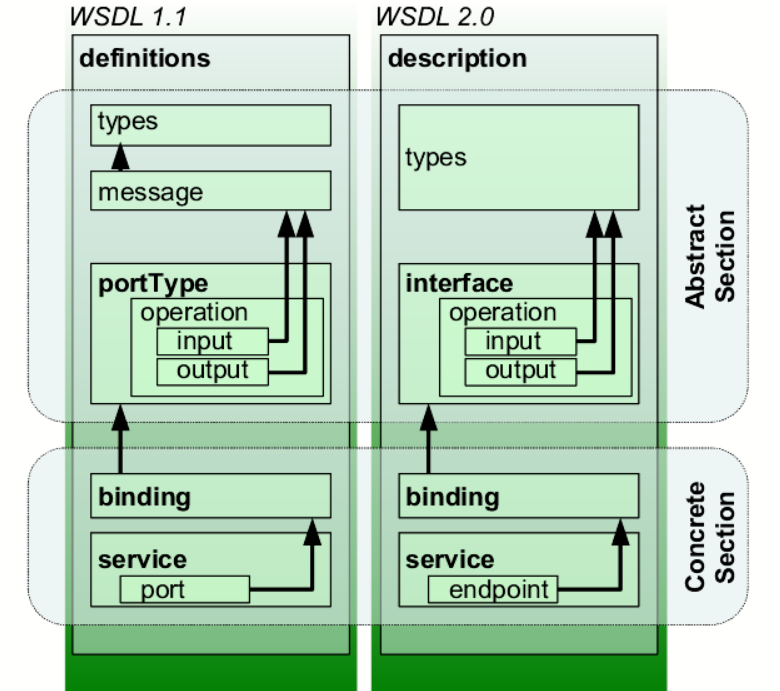
На основе WSDLможно генерировать proxy и stap (специальный класс, который занимается обменом сообщений с proxy, он находится на стороне сервера, т.е., по сути, это proxy на стороне сервера)

Вы генерировали его с помощью wsdl.exe утилиты, вы его генерировали и там получали абстрактный класс, на основе которого вы можете потом генерировать сервис.

WSDL (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:

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

****

**WSDL 1.1**

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

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

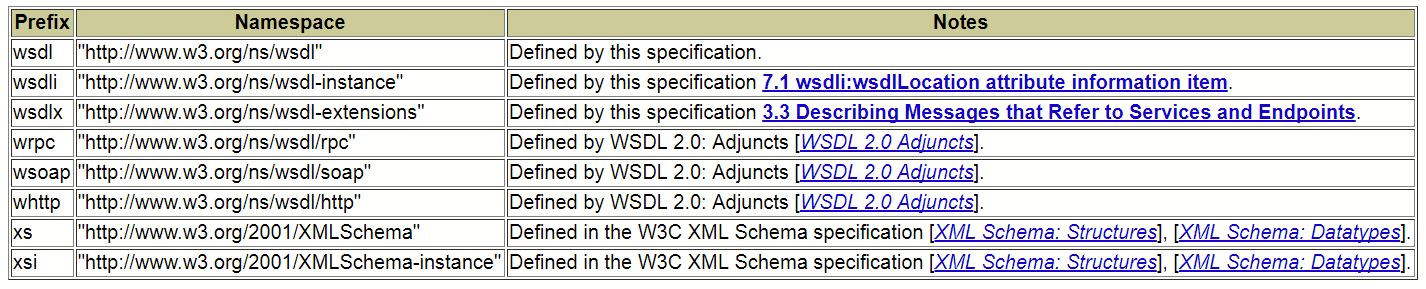
**Концептуальная модель WSDL 2.0**

Описание Web-сервиса можно разделить на две части: интерфейс сервиса (Абстрактное отделение: Messages, Operation, Interface) и реализация сервиса (Конкретное отделение: Binding, Service, Endpoint).

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).

**Пространства имён:**

****

**Структура wsdl-документа**

|  |
| --- |
| **<definition>**  **<types>**  **<!-- оперделение типов, используемых web-сервисом -->**  **<!-- для пл.-независимого WSDL используется синтаксис XML Schema -->**  **</types>**  **<message>**  **<!-- сообщения, используемые web-сервисом -->**  **<!-- сообщений, может быть несколько -->**  **<!-- каждое сообщение может состоять из нескольких частей -->**  **</message>**  **<portType>**  **<!-- методы, предоставляемые web-сервисом -->**  **<!-- может быть несколько портов -->**  **<!-- определены операции web-сервиса и используемые сообщения -->**  **</portType>**  **<binding>**  **<!-- протоколы связи, используемые web-сервисом -->**  **<!-- форматы сообщений и детали протокола для каждого порта -->**  **</binding>**  **<service>**  **<!— набор портов связанных с сервисом -->**  **</service>**  **</definition>** |

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

(10:20)

ASMX – это RPC-сервис (что такое RPС надо рассказать), web-сервис, канал у него полудуплекс он использует (раз это веб-сервис), частный случай WCF.

SOAP передаёт в качестве формата данных, вернее SOAP-протокол Envelop передаёт.

В качестве транспорта использует HTTP.

Встраивается в ASP.NET приложения, может требоваться IIS.

Есть wsdl.exe утилита, с помощь который можем генерировать proxy и с помощью которого можем генерировать star на стороне сервера. На входе wsdl.exe находится wsdl-файл, там вы укажите что сгенерирова… сервер или сгенерирова… клиент

ASMX (Active Server Method Extended) – технология Microsoft для разработки web-сервисов (2002, 2007), основанная на XML, WSDL, SOAP.

Плюсы ASMX:

• Легкость в разработке

• Легкость в изучении

• Нет «ада» конфигурирования

Плюсы WCF:

• Очень разнообразные и гибкие возможности транспорта

• Актуальная и развивающаяся технология

• Различные варианты хостинга

• Возможность реализации большого множества стандартов WS-\*

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

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

**Прокси-класс с помощью wsdl.exe**

Утилита wsdl.exe является соответствующей для asmx техникой потребления SOAP веб-сервисов. По wsdl файлу или ссылке она генерирует прокси-класс – специальной класс, максимально упрощающий обращение к данному веб-сервису. Кстати, у WCF аналогичная утилита называется SvcUtil.exe.

Утилита расположена в папке C:\Program Files (x86)\Microsoft SDKs\Windows.

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.

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.????

(11:48)

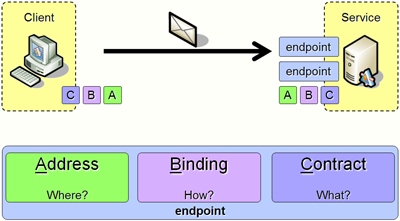
WCF (Windows Communication Foundation) – это платформа для разработки SOA-приложений (приложений сервис-ориентированной архитектуры). В основе его лежит протокол SOAP, в основе её лежит .NET.

WCF поддерживает не только веб-сервисы, но и сервисы других протоколов. Главная его особенность: один сервис может иметь много точек, сервис хост, сам себя запускает (может сам себя запускать как обыкновенное приложение). Может себя встраивать и в IIS, и в обыкновенное консольное приложение, куда угодно (в любое .NETовское приложение). Кроме того, там поддерживаются такие протоколы, как TCP, MSMQ, поддерживаются NamePipe(ы), всё что угодно, в том числе и HTTP.

Есть там понятие привязки, которые стандартную комбинацию протоколов позволяют нам использовать.

На самом деле, с помощью WCF можно соpдавать не только PRC-сервис, но и REST-сервис, но всё равно в качестве формата передачи данных используются Envelop(ы) SOAP протокола.

WCF:коммуникация модель



WCF: контракты

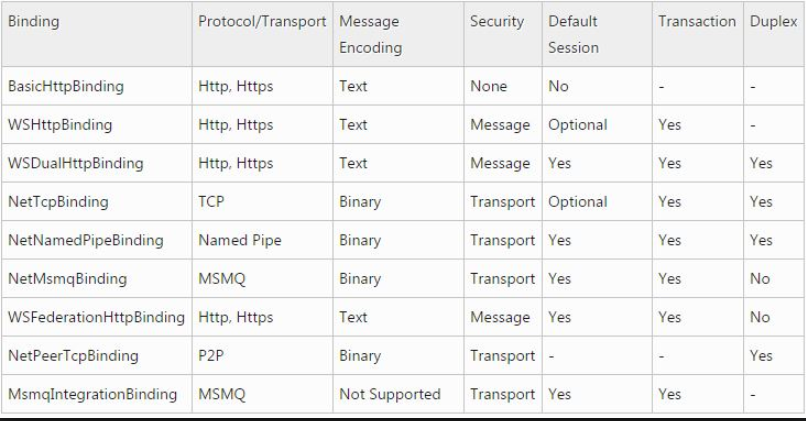
* контракт службы – этот контракт предоставляет клиенту, а также внешнему миру информацию о предложениях конечной точки и протоколах, которые будут использоваться в процессе связи.;
* контракт данных – указывает каким образом данные будут сериализованы;
* контракт сообщений – описывает формат сообщений – контракт данных контролируется контрактом сообщения. Он в первую очередь выполняет настройку форматирования типов параметров сообщений SOAP.

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

WCF: конечная точка – адрес, привязка, контракт.

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

Стандартные:

****

Основные отличия от ASMX-сервисов:

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

**Поведение**:

[ServiceBehavior(InstanceContextMode = InstanceContextMode. …)]

* PerSession – 0 – для каждого сеанса создается новый объект InstanceContext. Если канал не создает сеанс, это значение ведет себя так, как если бы оно было PerCall;
* PerCall – 1 – Новый объект InstanceContext создается перед каждым вызовом и удаляется после него;
* Single – 2 – Для всех входящих вызовов используется только один объект InstanceContext, который не удаляется после этих вызовов. Если объект службы отсутствует, он создается.

[ServiceBehavior(ConcurrencyMode = ConcurrencyMode. …)]

* Single – 0 – Экземпляр службы является однопотоковым и не допускает повторные входящие вызовы. Если свойство InstanceContextMode имеет значение Single и во время обслуживания вызова экземпляром поступают дополнительные сообщения, эти сообщения должны ждать освобождения службы или истечения времени ожидания;
* Reentrant – 1 – Экземпляр службы является однопотоковым и допускает повторные входящие вызовы. Служба, допускающая повторные входящие вызовы, принимает вызовы при вызове другой службы; следовательно, разработчик должен обеспечить сохранение согласованного состояния объекта до вызовов и подтвердить допустимость данных локальной операции после вызовов. Обратите внимание, что экземпляр службы разблокируется только путем вызова другой службы по каналу WCF. В этом случае вызванная служба может повторно войти в первую службу через обратный вызов. Если первая служба не допускает повторные входящие вызовы, последовательность вызовов приводит к взаимоблокировке. Дополнительные сведения см. в разделе ConcurrencyMode;
* Multiple – 2 – Экземпляр службы является многопотоковым. Гарантии синхронизации не предоставляются. Поскольку другие потоки могут в любое время изменить объект службы, необходимо всегда обеспечивать синхронизацию и согласованность состояний.

**Безопасность**:

Безопасность в WCF обеспечивается следующими функциями:

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

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

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

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

Режимы безопасности передачи:

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

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

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

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

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

При разработке RESTа обязательно перед методом указать:

[WebGet(UriTemplate = ”Add?a={a}&b={b}”)], либо

[WebInvoke(Method = “POST”, UriTemplate = ”Add?a={a}&b={b}”)]

И указать binding = “webHttpBinding”

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

(13:58)

Это просто на просто реализация WCF-сервиса для создания новостных каналов. Здесь применяется ATOM либо RSS (стандартные форматы, которые используются для передачи новостных сообщений в ленты в веб приложениях)

**RSS:**

Интернет-ресурс в формате RSS называется RSS-каналом, RSS-лентой, RSS-фидом.

MIME: application/rss+xml

RSS-агрегатор – web-приложение, которое автоматически с заданным интервалом времени проверяют, указанные в настройках ресурсы и экспортируют в формате RSS или ATOM информацию (например, заголовки новостей, блогов и пр.). Feedly, Яндекс.Лента, My Yahoo.

Структура:

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.

pubDate — Дата публикации канала как определено в спецификации RFC 822.

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

**ATOM:**

формат описания web-ресурсов и протокола для их публикации.

MIME: application/atom+xml.

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

Структура:

Элемент <feed>

Обязательные элементы

<id> – Идентификатор фида, постоянный URI.

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

<updated> – Дата последнего обновления в формате RFC 3339.

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.

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

(14:46)

Это тоже одна из реализаций WCF? Которая позволяет вам использовать протокол OData. С помощью этого протокола вы можете реализовать доступ к реляционной базе данных через REST-интерфейс. То есть у нас есть этот REST-интерфейс, есть правила использования этого REST-интерфейса, и эти правила позволяют (то есть там описан язык запросов) создавать запросы, равные по мощности SQL-запросам. И можем делать полноценные все селекты, все DML-операции через Open Data. Более того, можем делать DDL.

То есть DML и DDL реализованы в полной мере через OData, и вы OData можете вызывать из JavaScriptа, просто из JavaScriptа на прямую работать с реляционной базой данных и ничего нам больше не надо. Это очень удобный механизм, который используется достаточно часто

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

Поддерживается практически всеми системами программирования, в т.ч. JavaScript.

Для получения JSON, необходимо обеспечить в заголовке заголовок ACCEPT:application/json или $format=json.

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.

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

(16:28)

JSON-RPC – это тоже протокол, который позволяет вам реализовать RPC-интерфейс, но в качестве формата использует json. В информации запроса передаётся: версия RPC, команда(метод), параметры и id (id не передаётся только в случае уведомления, то есть одностороннего отправления); в ответе тоже: версия, сам ответ или сообщение об ошибке и id. Кроме того, в этом протоколе оговаривается какие ошибки может возвращать, формат ошибок.

Позволяет реализовать пакетные запросы (сделать пакет запросов). Оправлять пакет запросов в json формате и получать пакет ответов.

Где он применяется чаще всего? – в сети эфириум

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

Последняя версия 2.0.

JSON-RPC работает, отсылая запросы к серверу, реализующему протокол. Клиентом обычно является программа, которой нужно вызвать метод на удалённой системе. Множество входных параметров может быть передано удалённому методу, как массив или объект. Метод также может вернуть множество выходных данных (это зависит от реализации). Удалённый метод вызывается отправлением запроса на удалённый сервер посредством [HTTP](https://ru.wikipedia.org/wiki/HTTP)/HTTPS 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:

* id – Значение любого типа, которое используется для установки соответствия между запросом и ответом.
* Jsonrpc Строка, указывающая версию протокола JSON-RPC. ДОЛЖЕН быть точно "2.0".
* params – Массив объектов, которые должны быть переданы методу, как параметры.
* method – Строка с именем вызываемого метода.

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

* id – То же значение, что и в запросе, к которому относится данный ответ.
* Jsonrpc Строка, указывающая версию протокола JSON-RPC. ДОЛЖЕН быть точно "2.0".
* 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.
* code
* message
* data

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.

От смелова: Если вам когда-нибудь в жизни придется выбирать какой-то интерфейс для удалённого сервиса – выбирайте json rpc. Поддерживает сессию, что противоречит сервисам.

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

(18:05)

Один из способов реализации сервисов на основе ASP.NET, Core Nance позволяет создавать отдельные модули, которые независимы друг от друга. Мы с вами проходили OS GI, такой … технологий … на джаве, вот это аналог OS GI. OS GI позволяет в горящем режиме изменять модули, *а тут только меняется через остановку сервера*, но тем не менее модуль получается изолированный, который можно извлекать и вставлять туда. Вот такая значит есть технология.

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

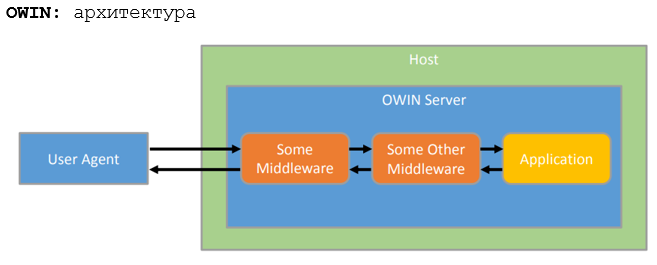
OWIN: Host – приложение-процесс операционной системы, управляющий жизненным циклом OWIN Server.

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.

middleware – подключенные компоненты (модули), предназначенные для обработки запросов.

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.

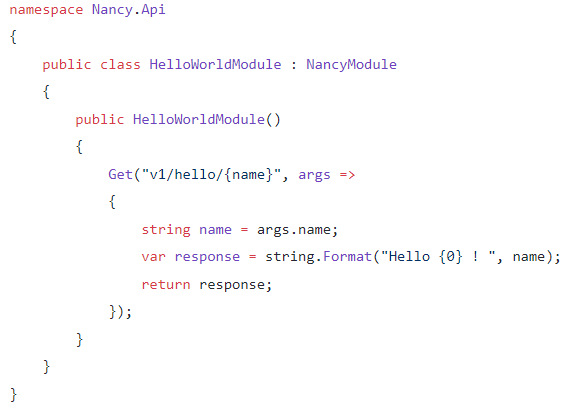
****

**Nancy** — это легкий, мощный, с открытым исходным кодом, очень простой веб-фреймворк, предназначенный для реализации сервисов на основе HTTP.Позволяет создавать независимые друг от друга модули

* простая конфигурация
* простой путь

**Принципы разработки:**

1. create an API .Net Core projects
2. скачать пакеты из NuGet: Nancy, Nancy.Owin
3. в методе Configure() Startup.cs конфигурация Nancy: app.UseOwin(x => x.UseNancy());
4. унаследовать NancyModule и в конструкторе



OSGi: спецификация динамической модульной системы и сервисной платформы для Java-приложений. Позволяет построить приложение из компонентов, которые можно динамически переинсталлировать компоненты (модули).

OSGi**:** процесс

* разрабатывается ***модуль*** (bundle, jar);
* модуль может быть ***динамически подсоединен*** к системе модулей, работающей на платформе OSGi;
* модуль может быть ***динамически отсоединен*** от системы модулей, работающей на платформе OSGi.

OSGi: жизненный цикл OSGi bundle (интерфейс BundleActivator).



1. **DDD/CQRS:** назначение, принципы применения.

DDD: Domain-Driven Design - предметно(проблемно)-ориентированное проектирование: набор принципов направленных на создание систем объектов со сложной предметной областью.

DDD (Domain-Driven Design) — это методология проектирования программного обеспечения, которая сфокусирована на области предметной области (домене) при разработке системы. Она была предложена Эриком Эвансом и предоставляет набор принципов и практик для улучшения процесса проектирования сложных систем.

DDD: цель – ускорить процесс проектирование программного обеспечения.\*\*\*\*

DDD: Domain – предметная область. Эксперт домена – специалист в предметной области.

DDD: 1) создание единого языка домена (Ubiquitous Language).Язык - набор однозначно определенных терминов. Важно, чтобы все участники команды использовали общий язык, который отражает основные понятия предметной области. Общение должно быть ясным и понятным для всех сторон.

DDD: 2) выделение ограниченных контекстов (имена) (Bounded Context). Домен разделяется на контексты, в каждом из которых определены свои термины, правила и модели. Контексты представляют собой границы, в пределах которых термины и модели имеют конкретное значение.

DDD: 3) определение контрактов (имена) (интерфейсов взаимодействия контекстов); контракты описывают команды (приводит к изменению состояния контекста, могут быть отклонены) и события (сообщение системы об изменении состояния, на них могут подписываться другие контексты, могут быть проигнорированы, но не могут быть отклонены).

DDD: 4) описание контекстов.

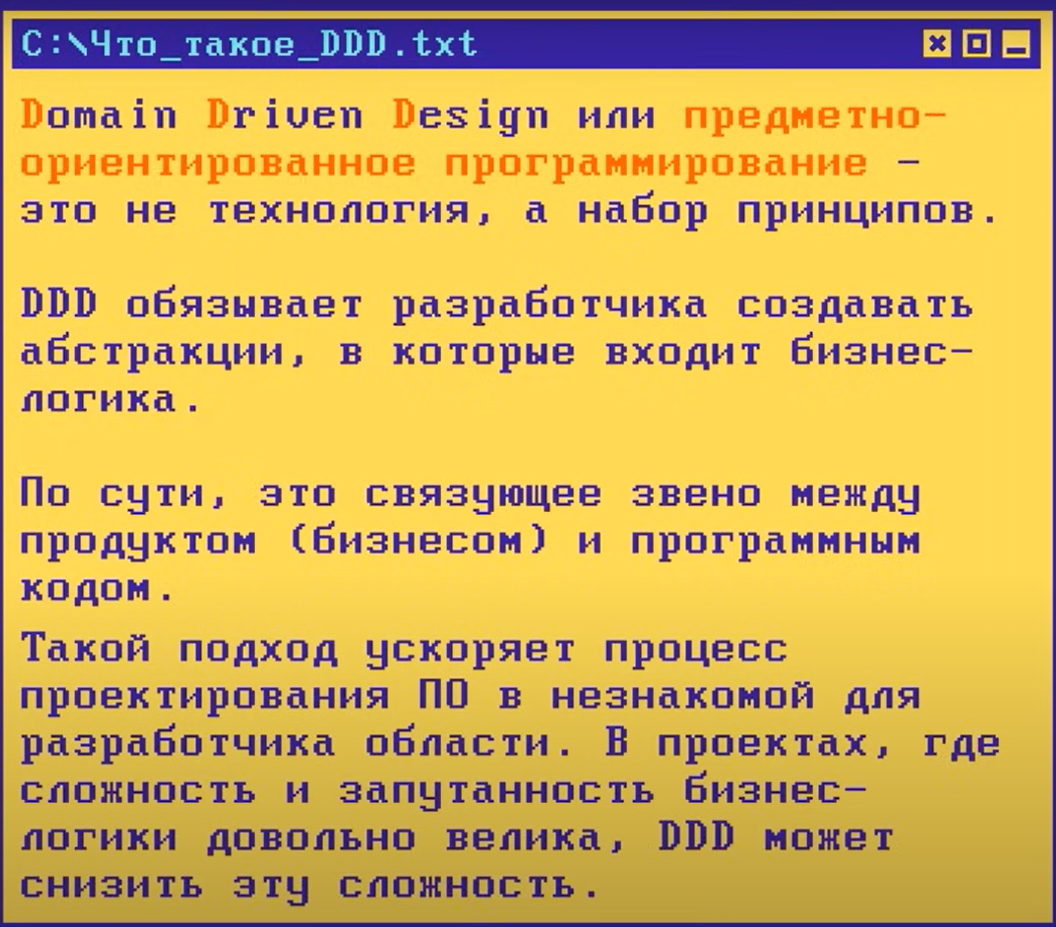
DDD: система имеет 4 уровня: interface (пользовательское взаимодействие, взаимодействие с другими системами), application (реализация, технические аспекты приложения, хранение данных, создание данных), business (реализация доменного языка), infrastructure (платформа и дополнительное программное обеспечение). \*\*\*\*

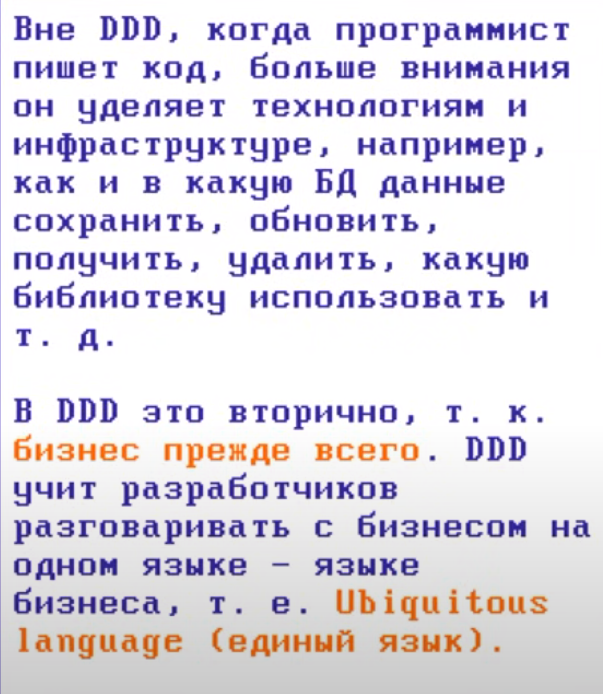
DDD: рекомендует паттерны: Value Object, Entity Object, Aggregate (граница транзакции, граница Entity), Repository, Factory,

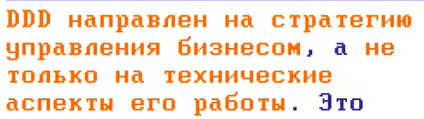
DDD: модули (высокая степень независимости).

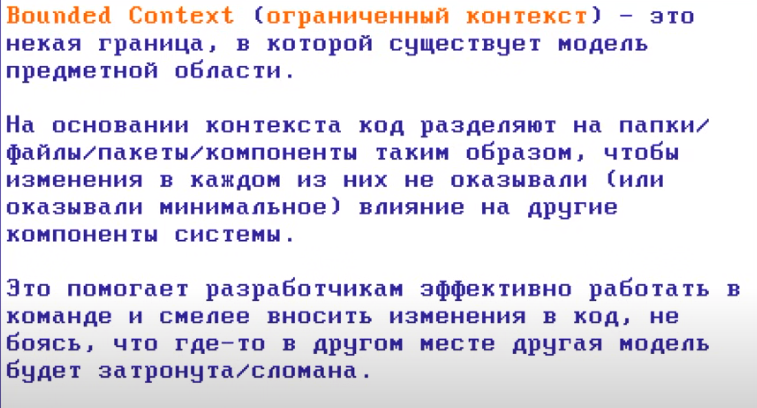
DDD: документация на языке

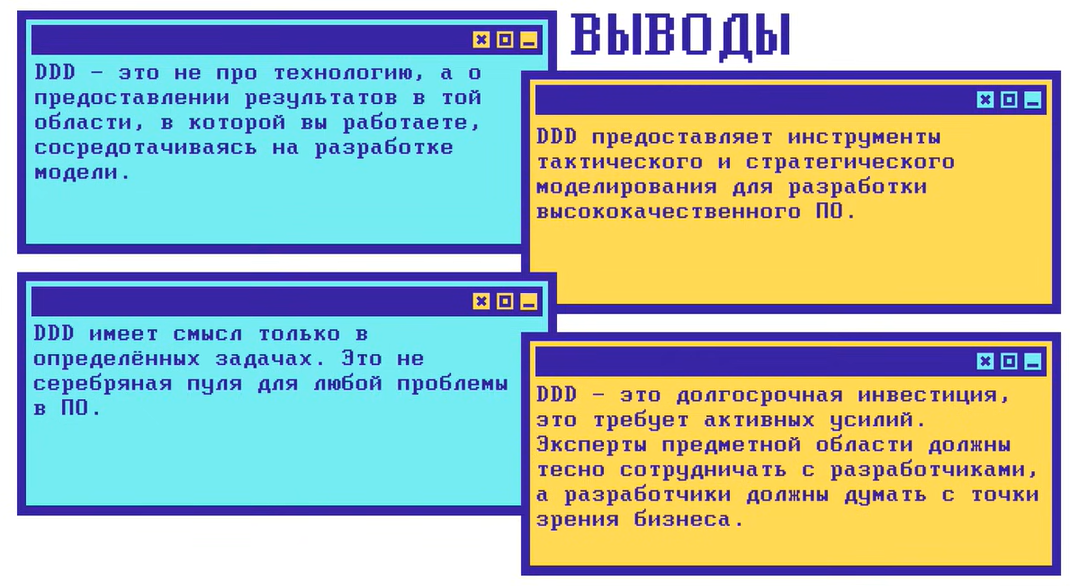
DDD: используется для больших проектов, длительностью более 6 месяцев.











**CQRS: Command-Query responsibility segregation**

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). \*\*\*\*

CQRS: контрактное программирование (Б.Майер): спецификация методов (интерфейсы), предусловие, постусловие, инвариант (обязательные соглашения в методе), исключения.

CQRS: перенос основных принципов CQS на архитектуру (М.Фаулер)

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

CQRS (Command Query Responsibility Segregation) - это паттерн проектирования, который предлагает разделение операций записи (команд) и операций чтения (запросов) в системе. Этот подход позволяет улучшить масштабируемость, гибкость и обслуживаемость системы. Вот основные принципы и назначение CQRS:

**Назначение CQRS:**

1. **Разделение ответственности:** CQRS разделяет операции записи и операции чтения, что позволяет легче оптимизировать и масштабировать обе части системы.
2. **Масштабируемость:** Поскольку операции записи и операции чтения часто могут быть масштабированы независимо друг от друга, CQRS упрощает горизонтальное масштабирование конкретных компонентов системы.
3. **Гибкость:** CQRS делает возможным применение различных моделей данных и логики для операций записи и операций чтения, что дает большую гибкость в проектировании системы.
4. **Улучшенная производительность:** Поскольку операции записи и операции чтения могут использовать оптимизированные хранилища данных, система может достичь лучшей производительности.

**Принципы применения CQRS:**

1. **Явное разделение команд и запросов:** Команды (операции записи) и запросы (операции чтения) должны быть явно разделены. Команды обрабатывают изменение состояния системы, а запросы - предоставление информации.
2. **Моделирование команд и запросов отдельно:** Лучше создавать разные модели для команд и запросов, чтобы облегчить разработку и обслуживание кода. Модель команд обычно ориентирована на изменение состояния, в то время как модель запросов на предоставление данных.
3. **Асинхронная коммуникация:** Использование асинхронных механизмов для команд и запросов может улучшить производительность и уменьшить зависимости между частями системы.
4. **Оптимизация для чтения и записи:** Поскольку операции чтения и записи могут быть оптимизированы по-разному, их следует рассматривать отдельно при выборе хранилищ данных и других технологий.
5. **Событийно-ориентированная архитектура (Event Sourcing):** Использование событий для хранения изменений в системе может облегчить отслеживание изменений и восстановление состояния системы.
6. **Внимание к согласованности:** Поскольку CQRS может включать в себя асинхронную обработку команд, необходимо уделять внимание согласованности данных между разными частями системы.
7. **Тестирование:** Важно тщательно тестировать как операции записи, так и операции чтения, чтобы удостовериться в правильности их работы.

В микросервисной архитектуре данные распределены по разным микросервисам, у каждого микросервиса своя база данных, а для того чтобы отрисовать пользовательский интерфейс нам как правило требуются агрегированные данные из нескольких микросервисов. Можно попробовать ходить по сети во все микросервисы, собирать с них нужные данные, объединять и выдавать клиенту. Но в реальности это не работает. Такое решение имеет низкий Availability, при разделении сети, когда один сервис перестает видеть другой приложение фактически не работает. Кроме того сервисы могут медленно отвечать, не успевать укладываться в таймаут ответа и т.п.

CQRS позволяет решить проблему чтения в микросервисной архитектуре. Теперь мы ничего напрямую не читаем из микросервисов, а подписываемся на все события которые генерируются микросервисами и по этим событиям обновляем нашу базу данных на чтение. Таким образом любые запросы на чтение становятся элементарным действием, так как достаточно сходить в базу данных на чтение и получить все необходимые данные.

1. **OSGi:** назначение, принципы применения.

OSGi разрабатывают не конкретный продукт, а спецификации вечно работающих сервисов.

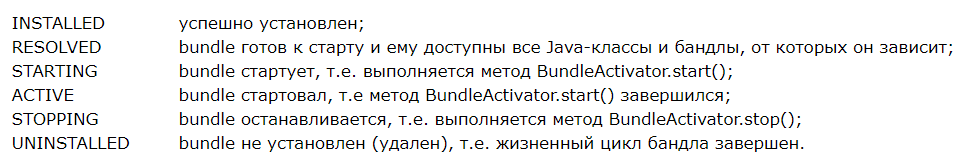
OSGi: спецификация динамической модульной системы и сервисной платформы для Java-приложений. Позволяет построить приложение из компонентов, которые можно динамически переинсталлировать компоненты (модули).

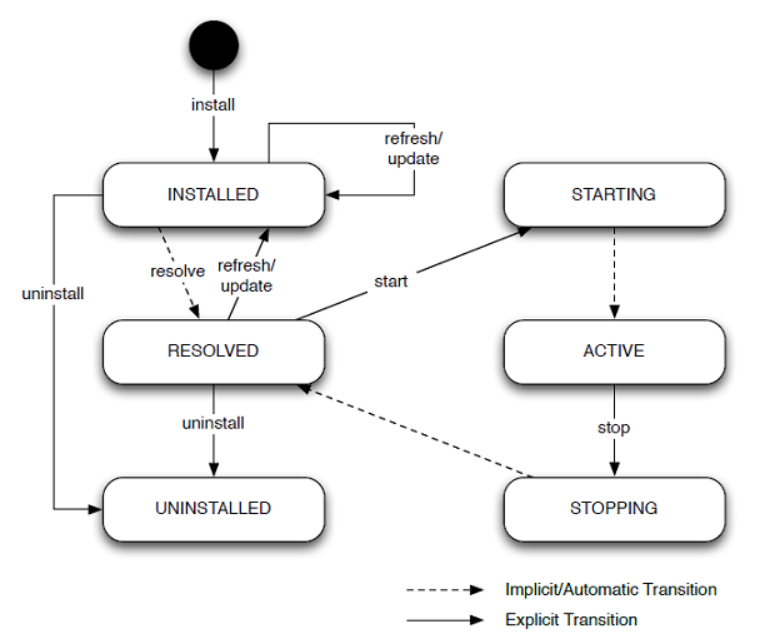
OSGi:

* + разрабатывается ***модуль*** (bundle, jar);
  + модуль может быть ***динамически подсоединен*** к системе модулей, работающей на платформе OSGi;
  + модуль может быть ***динамически отсоединен*** от системы модулей, работающей на платформе OSGi.

OSGi: OSGi-bundle – это jar-архив, включающий дополнительные данные.

**OSGi:** жизненный цикл OSGi bundle (интерфейс **BundleActivator**).

****

****

**OSGi:** взаимодействие OSGi-модулей:

* ***ресурсные bundle*** – централизованное хранилище, предназначенное для хранения данных;
* ***сервисные bundle*** – классы, с методами, которые можно вызывать (должен быть зарегистрирован в реестре сервисов, позднее связывание);
* ***издатель/подписчик***.

OSGi (Open Service Gateway initiative) - это стандарт для динамического модульного программного обеспечения в языке Java. Его основной целью является создание гибких, расширяемых и модульных приложений. Вот основные назначение и принципы применения OSGi:

**Назначение OSGi:**

1. **Модульность:** OSGi позволяет разрабатывать приложения в виде небольших, независимых модулей, называемых "бандлами". Эти бандлы могут быть динамически установлены, обновлены и удалены без перезапуска всего приложения.
2. **Динамическое управление:** OSGi предоставляет механизмы для динамического управления жизненным циклом бандлов. Это позволяет добавлять, обновлять и удалять компоненты приложения в реальном времени, что особенно полезно для систем с высокой доступностью.
3. **Сервисная модель:** OSGi включает сервисную модель, которая позволяет бандлам предоставлять и потреблять услуги. Эти услуги могут динамически регистрироваться и извлекаться, обеспечивая слабую связанность компонентов.
4. **Изоляция классов и пространств имен:** OSGi обеспечивает изоляцию классов и пространств имен между бандлами, что предотвращает конфликты имен и обеспечивает чистоту окружения выполнения.
5. **Управление зависимостями:** OSGi автоматически управляет зависимостями между бандлами, что упрощает развертывание и обеспечивает согласованность приложений.

**Принципы применения OSGi:**

1. **Модульность приложения:** Разрабатывайте приложение с использованием модульной структуры, разбивая его на бандлы. Каждый бандл должен предоставлять конкретный функционал и зависеть только от тех ресурсов, которые ему необходимы.
2. **Динамическое обновление:** Воспользуйтесь возможностью динамического обновления бандлов. Это облегчит развертывание изменений и уменьшит необходимость перезапуска всего приложения.
3. **Использование сервисной модели:** Используйте сервисы для реализации слабосвязанных компонентов. Это позволит легко заменять или добавлять функциональность без изменения других частей приложения.
4. **Управление зависимостями:** Внимательно управляйте зависимостями между бандлами. Проверьте, чтобы каждый бандл явно указывал свои зависимости, чтобы избежать конфликтов и обеспечить совместимость.
5. **Изоляция ресурсов:** Используйте изоляцию классов и пространств имен для предотвращения конфликтов имен и обеспечения чистого окружения выполнения для каждого бандла.
6. Event Storing: назначение, принципы применения, примеры реализации.

**Event Sourcing (Ивент-сорсинг):** Это паттерн проектирования, при котором все изменения состояния системы представлены в виде событий.  
**Event Storing (Хранение событий):**

* **Определение:** Это относится к механизму сохранения событий в системе.

Здесь правильно сказать не Event Storing, а Event Sourcing, ну Event Storing так называется такие базы данных, вернее подход, который применяется чаще всего в сервис-ориентированной архитектуре приложения (приложениях с сервис-ориентированной архитектурой), ещё чаще применяется в микросервисах. Когда запоминается в базе данных, есть специальные базы данных, Event Storing базы данных, в которых запоминается не изменение данных, а запоминается само событие, все эти события фиксируются, а потом уже в асинхронном режиме это серверное событие преобразовывается в запрос реляционной или нереляционной базе данных и проводятся изменения в этой базе данных. И есть две базы данных, одна Event Storing, которая запоминает события, а вторая база данных используется для чтения, вот между ними есть репликация … это вот как раз и есть подход Event Storing

**Event Sourcing** (ES)

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

ES: каждое событие имеет имя.

ES: произошедшие события неизменны.

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

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

ES: обычно применяется в архитектуре CQRS для реализации WriteDB.



**EventStore**

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

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

Ну, что такое микросервисы надо рассказать, в чём их особенность, какие архитектурные решения там могут быть, хореография, что это такое, что такое оркестровка микросервисов, что такое DevOps надо рассказать, и как это связано с микросервисами, дать определение микросервисам, которое вы знаете

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

Микросервис – сервис, выполняющий одну элементарную функцию; основной принцип разбиения – изменение сервиса не затрагивает другие сервисы.

Микросервисная архитектура – набор принципов, которым должны соответствовать сервисы. Задается или в форме правил или основывается на применении готовых фреймворков (например, Karyon, Dropwiard, …).

Цель:

1) устойчивость к сбоям;

2) облегчить понимание и поддержку кода;

3) усилить работу команды программистов.

Популярные паттерны Tolerant Reader, Consumer Driver.

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

Требуется DevOps – 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.

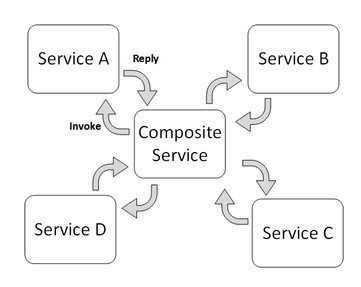
ci = пуш в репозиторий -> мерж -> тесты

cd = деплой на платформу

**Оркестровка**:

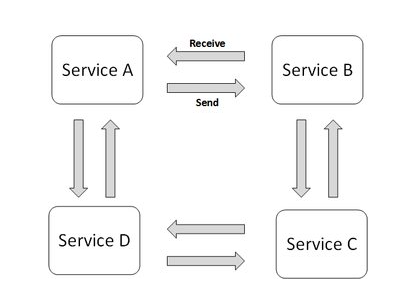
Оркестровка сервисов представляет собой единый централизованный исполняемый бизнес-процесс (оркестратор), который координирует взаимодействие между различными сервисами. Оркестр отвечает за вызов и объединение сервисов.

Отношения между всеми участвующими услугами описываются одной конечной точкой (то есть, составной услугой). Оркестровка включает управление транзакциями между отдельными службами. Оркестровка использует централизованный подход к составу услуг.



**Хореография** описывает взаимодействие между несколькими службами, в то время как оркестровка представляет контроль с точки зрения одной из сторон. Хореография отличается от оркестровки расположением логики, которая контролирует взаимодействие между службами.

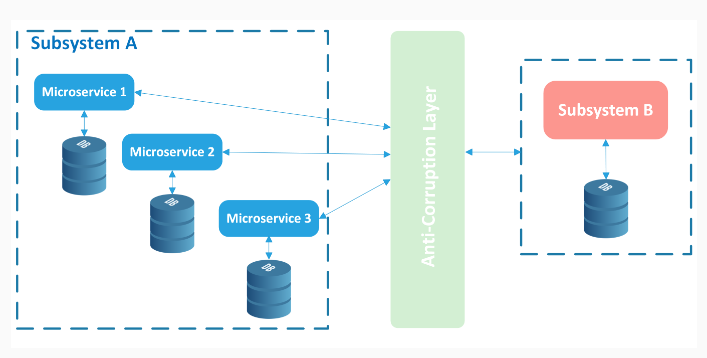
Хореография является глобальным описанием участвующих услуг, который определяется путем обмена сообщениями, правил взаимодействия и соглашений между двумя или более услугами, и использует децентрализованный подход к составу услуг.



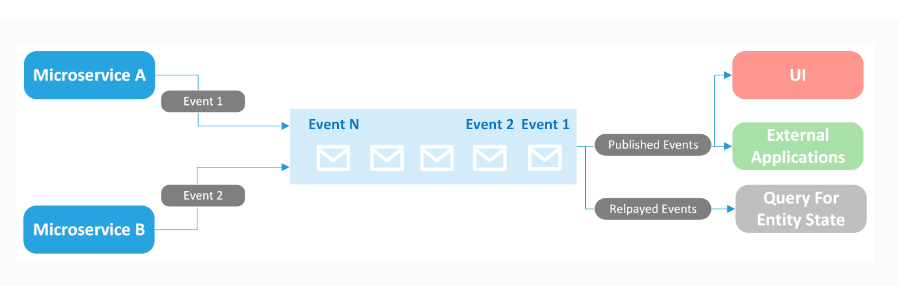
* Хореография. Децентрализованная координация, при которой каждый микросервис прослушивает события/сообщения другого микросервиса и решает, следует предпринять действие или нет.
* Оркестровка. Централизованная координация, при которой отдельный компонент (оркестратор) сообщает микросервисам, какое действие необходимо выполнить далее.

1. **Паттерны микросервисной архитектуры: Strangler,** **Anti-Corruption Layer, Command Query Responsibility,** **Event Sourcing, API Gateway,** Server-Side Service Discovery, Circuit Breaker, **Log Aggregation, Health Check.**

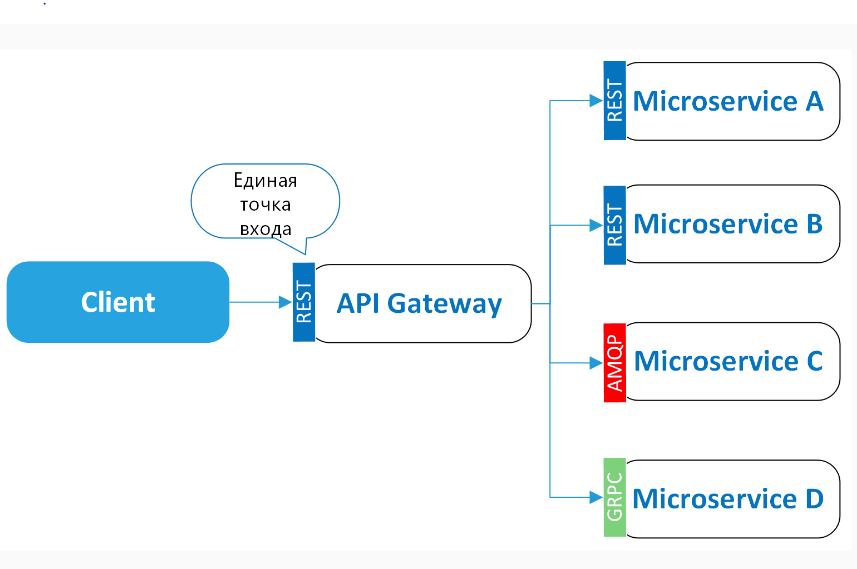
**Паттерны рефакторинга для перехода на микросервисы** (эта группа шаблонов предназначена для организации взаимодействия с Legacy-приложениями и/или их постепенного перевода на микросервисную архитектуру.)

1. Шаблон «Душитель» (Strangler) – Паттерн Strangler помогает поэтапно переходить от монолитной архитектуры к микросервисам. Новые функции или обновления реализуются как микросервисы, постепенно замещая части монолита. Таким образом, старый монолит "удушается" новыми микросервисами, и в конечном итоге монолит может быть полностью вытеснен.
2. Шаблон «Уровень защиты от повреждений» (Anti-Corruption Layer) – уровень защиты от повреждений, паттерн рефакторинга, Anti-Corruption Layer - отдельный сервис для взаимодействия с Legacy-подсистемами (Subsystem B). Этот паттерн используется для изоляции микросервисов друг от друга. Слой против коррупции предоставляет прослойку, которая переводит сложные внутренние структуры данных одного сервиса в более простые и понятные для другого. Это помогает избежать "заражения" одного сервиса внутренней логикой другого. 

**Паттерны управления данными в микросервисной архитектуре**

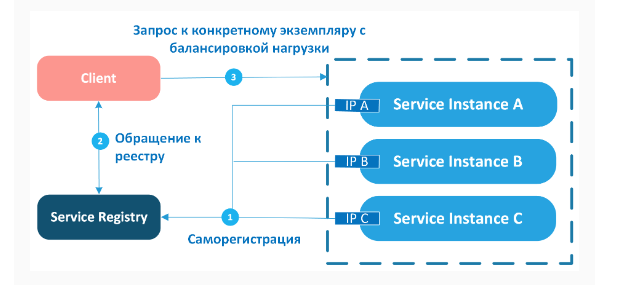
1. Шаблон «Разделение команд и запросов» (Command Query Responsibility Segregation, CQRS) – хранить данные и их процессы изменения либо в одной бд, либо в разных. Событийное ведение подразумевает сохранение изменений в системе в виде событий. Эти события используются для воссоздания состояния системы. Этот подход полезен для слежения за изменениями в системе, отладки и восстановления после сбоев.
2. Шаблон «Поиск событий» (Event Sourcing) – хранение только событий. Событийное ведение подразумевает сохранение изменений в системе в виде событий. Эти события используются для воссоздания состояния системы. Этот подход полезен для слежения за изменениями в системе, отладки и восстановления после сбоев. 

**Паттерны коммуникации микросервисов** (этот блок шаблонов охватывает способы внешних взаимодействий микросервисов: с клиентскими приложениями, удаленными сервисами и так далее.)

1. Шаблон «API-шлюз» (API Gateway) – промежуточное звено между клиентами и микросервисами. Шлюз API предоставляет единую точку входа для клиентов, объединяя запросы и предоставляя унифицированный интерфейс для взаимодействия с множеством микросервисов. Это упрощает управление запросами, обеспечивает безопасность и улучшает производительность. 

**Паттерны обнаружения сервисов в микросервисной архитектуре**

1. Шаблон «Обнаружение сервисов на стороне сервера» (Server-Side Service Discovery) – всё происходит на стороне сервера. Этот паттерн автоматически обнаруживает и регистрирует микросервисы в системе, облегчая процесс поиска и взаимодействия между сервисами в распределенной среде. **Client-Side Service Discovery:** обнаружение сервисов на стороне клиента, Registry.
2. сервис регистри­рует свое сетевое местоположение в реестре; может предоставить URL-адрес для проверки своей работоспособности (Health Check);
3. клиент обращается к реестру, чтобы получить список сервисов; клиент может кэшировать список;
4. клиент использует алгоритм балансировки нагрузки.



1. **Server-Side Service Discovery:** обнаружение сервисов на стороне сервера, Router, Registry, Registrator.
2. регистратор(часть платформы развертывания) прописывает все сервисы в реестре сервисов (DNS-имя, виртуальный IP-адрес);
3. по DNS-имени клиент обращается в маршрутизатор (часть платформы развертывания);
4. маршрутизатор обращается к реестру для получения сетевого расположения сервиса;
5. маршрутизатор балансирует нагрузку выбором конкретного экземпляра сервиса для отправки запроса.



**Паттерны повышения отказоустойчивости** (эта группа шаблонов предназначена для повышения надежности приложений с микросервисной архитектурой.)

1. Шаблон «Автоматический выключатель» (Circuit Breaker) – есть прокси сервис, который блокирует доступ к микросервисам, на время, если было выполнено много неудачных (безответных) запросов. Автоматический выключатель предотвращает запросы к сервису, который временно недоступен или испытывает проблемы. Это помогает предотвратить перегрузку системы и ускоряет восстановление после сбоев.

**Паттерны мониторинга микросервисов** (этот блок шаблонов охватывает возможные варианты построения мониторинга работы микросервисов.)

1. Шаблон «Агрегация логов» (Log Aggregation) – Этот паттерн собирает и агрегирует журналы из различных микросервисов в централизованный репозиторий. Это упрощает мониторинг, отладку и анализ системы, т.е. **агрегация журналов, централизованный журнал для всех сервисов.**
2. Шаблон «Проверки здоровья» (Health Check) – Health Check предоставляет механизм для мониторинга состояния каждого микросервиса. Это помогает выявлять проблемы и автоматически управлять недоступными сервисами.