Лекция “Архитектура ИС”

Оглавление

Что такое Архитектура?

Определения

- *архитектура ИС* - концепция, определяющая модель, структуру, выполняемые функции и взаимосвязь компонентов информационной системы;

- *архитектура ИС* - базовая организация системы, воплощенная в ее компонентах, их отношениях между собой и окружением, а также принципы, определяющие проектирование и развитие системы; Согласно ГОСТ Р 57100-2016/ISO/IEC/IEEE 42010:2011:

- *архитектура ИС* - набор значимых решений по поводу организации системы программного обеспечения, набор структурных элементов и их интерфейсов, при помощи которых компонуется система вместе с их поведением, определяемым во взаимодействии между этими элементами, компоновка элементов в постепенно укрупняющиеся подсистемы, а также стиль архитектуры, который направляет эту организацию (элементы и их интерфейсы, взаимодействие и компоновка).

Все определения архитектуры ИС могут быть разделены на два класса - «концептуальные» и «технологические».

Концептуально ***архитектура ИС -*** набор основных решений, неизменных при изменении бизнес-технологии в рамках бизнес- видения и оказывающих определяющее влияние на совокупную стоимость владения системой.

Технологически архитектура обычно определяется следующими параметрами: назначением системы, составными частями системы и их размещением, связями между частями системы, принципами взаимодействия составных частей.
Таким образом, технологически ***архитектура ИС*** - набор основных решений по выбору средств реализации, а именно, аппаратной платформы, операционной системы, телекоммуникационных средств, СУБД, и т.д., влияющих на совокупную стоимость владения системой.

В рамках разработки архитектуры определяются:

- что будет делать система;
- из каких компонентов (частей, модулей) она будет состоять;
- где именно компоненты будут располагаться;
- каким образом компоненты будут взаимодействовать.

Для архитектуры определяются и описываются:

- Базовые параметры и характеристики архитектуры.
- Логическая и физическая структура.
- Взаимодействие системных компонентов (подсистемы и модули, синхронность и асинхронность их взаимодействия, каналы коммуникации и их характеристики, протоколы и интерфейсы, тип программного обеспечения промежуточного слоя, форматы файлов, которыми система будет оперировать, и другие особенности).

- Необходимые элементы ИТ-инфраструктуры для реализации выстраиваемой архитектуры ИС – платформа (среда), аппаратный комплекс, СУБД, инструментарий прикладное ПО.
- Возможные риски, ограничения, стоимость владения, экономическая обоснованность.

Классификация

Архитектуры ИС

- настольные *(Desktop),* или локальные ИС, в которых все компоненты (базы данных (БД), системы управления базами данных (СУБД), клиентские приложения) находятся на одном компьютере;
- • распределенные *(Distributed)* ИС, в которых компоненты распределены по нескольким компьютерам.

Существуют следующие виды архитектур ИС: Локальная; Файл-серверная; Клиент-серверная; Трехслойная. Трёхслойная или многозвенная сейчас разделяется на Монолитную, Микро-сервисную и Сервисно-ориентированную (SOA).

Далее рассмотрим историю развития архитектур, а так же характеристики, достоинства и недостатки каждой из них.

История

Основные события, которые повлияли на создание новых типов Архитектур

Локальные информационные системы

широко использовались до появления компьютерных сетей. В этом случае все компоненты ИС располагаются на одном компьютере. Очевидным недостатком этой архитектуры является возможность работать в ИС только одному пользователю. Другие пользователи не имеют возможности получить доступ к данным даже для чтения.

Данные в локальных ИС хранятся и обрабатываются на одном и том же компьютере. До появления компьютерных сетей все ИС вынужденно являлись локальными. В те времена для работы с одной базой данных каждому пользователю приходилось создавать локальную копию этой базы данных на своем компьютере. С некоторой периодичностью данные нужно было синхронизировать и объединять. Конечно, это порождало ошибки и сложности. В современных

условиях этот подход считается устаревшим, а многопользовательский режим работы является одним из основных требований к ИС.

Однако, существует немногочисленный класс программ, где не нужен многопользовательский доступ к данным. К ним относятся мобильные приложения, однопользовательские игры, организация хранения, поиска и просмотра баз переписки почтовых клиентов и некоторых мессенджеров, плейлистов медиапроигрывателей, уменьшенных копий изображений и т.д. Программы, выполняющие подобные задачи, являются локальными ИС.

Для работы локальной ИС необходимо организовать на пользовательском компьютере хранение данных и доступ к ним. Это можно сделать тремя способами:

- С использованием полной версии любой СУБД;
- С использованием встроенной СУБД;
- Без использования СУБД.

Использование полной версии СУБД.
На локальный компьютер устанавливается полная версия СУБД, а приложение подключается к ней. Фактически в этом случае все организовано так же, как и в любой многопользовательской архитектуре. Просто СУБД не вынесена на отдельный сервер. Этот способ удобен в том случае, если память компьютера позволяет разместить полную версию СУБД и она будет использоваться для дальнейших модификаций ИС. Часто таким методом пользуются программисты в процессе разработки с последующим переносом СУБД на сервер и переходом к многопользовательскому режиму.

Встаиваемые СУБД обычно имеют ограниченные возможности в сравнении с обычными СУБД. Для доступа к данным используется либо усеченная версия sql, либо специальный язык запросов. Права пользователей никак не поддерживаются и не контролируется. Не поддерживаются архивация и репликация данных, поэтому надежность хранения данных зависит от надежности самой библиотеки и файловой системы компьютера. При такой технологии работы приложения, как правило, не компилируются, а интерпретируются, что несколько замедляет работу всей системы.

Работа без использования СУБД
В этом случае данные все равно нужно где-то хранить. С этой целью используются текстовые файлы. Это могут быть файлы известных стандартных форматов: XML; CSV; JSON; YAML. Могут использоваться другие «авторские форматы». ИС полностью закрыта для модификаций, что существенно повышает уровень защиты приложения и скорость его работы.

Файл-серверная архитектура

С появлением компьютерных сетей возникла возможность хранить данные в файлах на выделенном специально для этой цели компьютере. Такой компьютер называется файловым сервером или просто сервером. Компьютеры пользователей соединены с сервером сетью, поэтому доступ к данным, могут получить несколько пользователей одновременно. Однако, кроме функции хранения данных и обеспечения доступа к ним, сервер никаких функций не выполняет. Приложения, обрабатывающие данные, находятся на пользовательских компьютерах.

ПРИМЕР

Предположим, что в базе данных на сервере хранится список сотрудников крупного предприятия. На предприятии 1500 сотрудников и 10 подразделений. Пользователю нужно получить число сотрудников, работающих в каждом подразделении. Для решения этой задачи пользователь должен запросить данные всех 1500 сотрудников с сервера по сети, после чего на пользовательском компьютере выполнится процедура, которая осуществит подсчет сотрудников в каждом подразделении. Результатом процедуры будет 10 строк. Таким образом, чтобы получить 10 строк придется передать по сети 1500 строк.

Обработка данных на пользовательском компьютере всегда сопровождается передачей по сети большого количества «лишней» информации. Основными недостатками файл-серверной архитектуры являются: Высокая загруженность сети и, как следствие, низкая скорость работы; Сложность поддержания непротиворечивости данных, из-за их несогласованной обработки разными пользователями.

Клиент-серверная архитектура

До определенного момента на СУБД возлагались лишь задачи хранения данных и организации доступа к ним. С развитием технологий в состав СУБД разработчики стали включать новый компонент – процедурный язык программирования. С его помощью в СУБД стало возможным создавать процедуры для обработки данных, которые можно вызывать повторно. Такие процедуры называются хранимыми процедурами. Наличие хранимых процедур дало возможность осуществлять некоторую часть обработки данных на сервере.

ПРИМЕР
Рассмотрим задачу из примера 1 в условиях клиент-серверной архитектуры. Пользователь отправит на сервер запрос, который запустит процедуру. Процедура выполнится непосредственно на сервере. Она подсчитает количество сотрудников в каждом подразделении и отправит полученные 10 строк по сети на клиентский компьютер. Таким образом, произойдет существенная экономия трафика: вместо 1500 строк будет передано по сети всего 10.

Клиент-серверная архитектура позволяет разгрузить сеть и поддерживать непротиворечивость данных за счет их централизованной обработки. Однако, языки хранимых процедур не приспособлены для полноценной реализации бизнес-логики. Поэтому бизнес-логика в клиент-серверных ИС по-прежнему реализуется на клиентских компьютерах. Такой подход имеет следующие недостатки: Любые изменения в бизнес-логике требуют обновления на клиентском компьютере; Клиентские компьютеры должны быть достаточно производительными; Слабая защита данных от взломов.

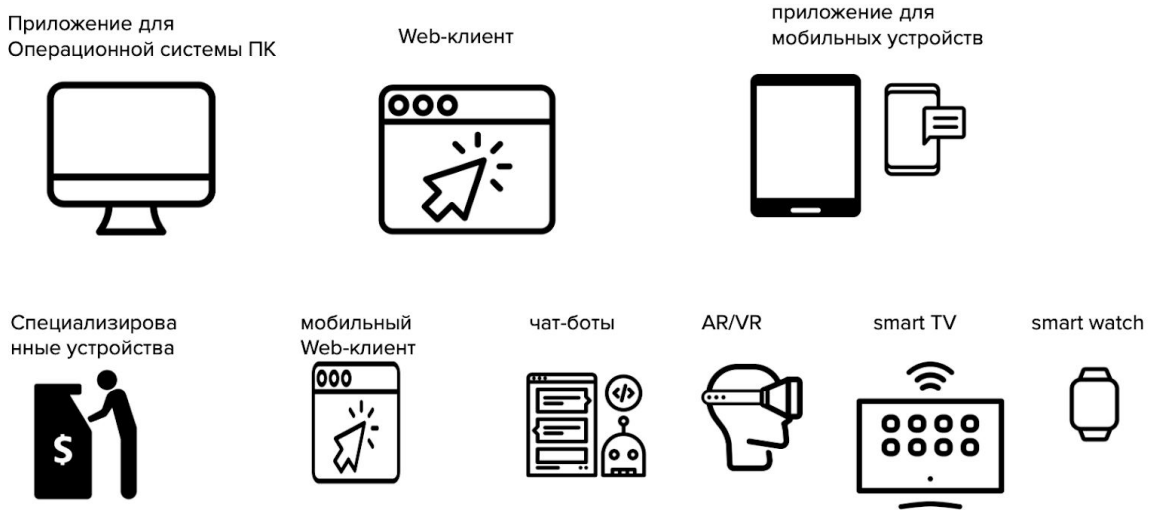
Трехуровневая архитектура

Все недостатки клиент-серверной архитектуры связаны с тем, что на клиентском компьютере лежит слишком большая нагрузка, которую можно было бы перенести на сервер. Поэтому дальнейшее развитие технологий двигалось в направлении переноса нагрузки с клиентских компьютеров на сервер. В дополнение к хранимым процедурам разработчики стали

использовать серверные языки программирования. Это дало возможность создавать в ИС промежуточный уровень - сервер приложений.
Использование сервера приложений позволяет максимально разгрузить клиентские компьютеры и сделать обработку данных еще более централизованной, что повышает скорость и надежность ИС.

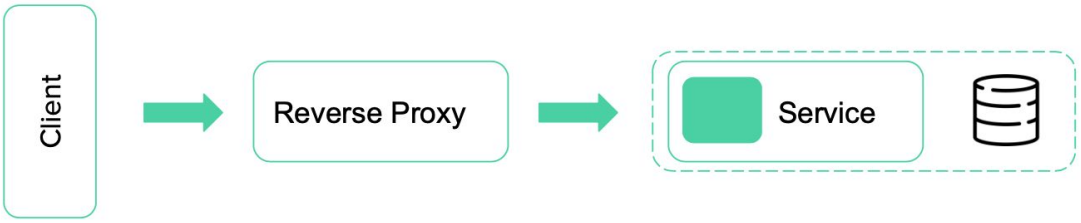
В файл-серверных ИС база данных находится на файловом сервере, а СУБД и клиентские приложения находятся на рабочих станциях. В клиент-серверных ИС база данных и СУБД находятся на сервере, а на рабочих станциях находятся клиентские приложения. В свою очередь, клиент-серверные ИС разделяют на двухзвенные и многозвенные. В двухзвенных (англ, *two-tier*) ИС всего два типа «звеньев»: сервер баз данных, на котором находятся БД и СУБД (англ, *back-end*), и рабочие станции, на которых находятся клиентские приложения (англ, *front-end').* Клиентские приложения обращаются к СУБД напрямую. В многозвенных (англ. *multi-tier*) ИС добавляются промежуточные «звенья»: серверы приложений *(application servers*). Пользовательские клиентские приложения не обращаются к СУБД напрямую, они взаимодействуют с промежуточными звеньями. Типичный пример применения многозвенности — современные веб-приложения, использующие базы данных. В таких приложениях помимо звена СУБД и клиентского звена, выполняющегося в веб-браузере, имеется как минимум одно промежуточное звено — веб-сервер с соответствующим серверным программным обеспечением

Типы клиентов



Монолитный бекенд

Простейший и популярный вариант архитектуры – монолитная. Каждый начинал с неё, и здесь нет никакой изоляции и распределённости: один монолит обрабатывает все запросы.



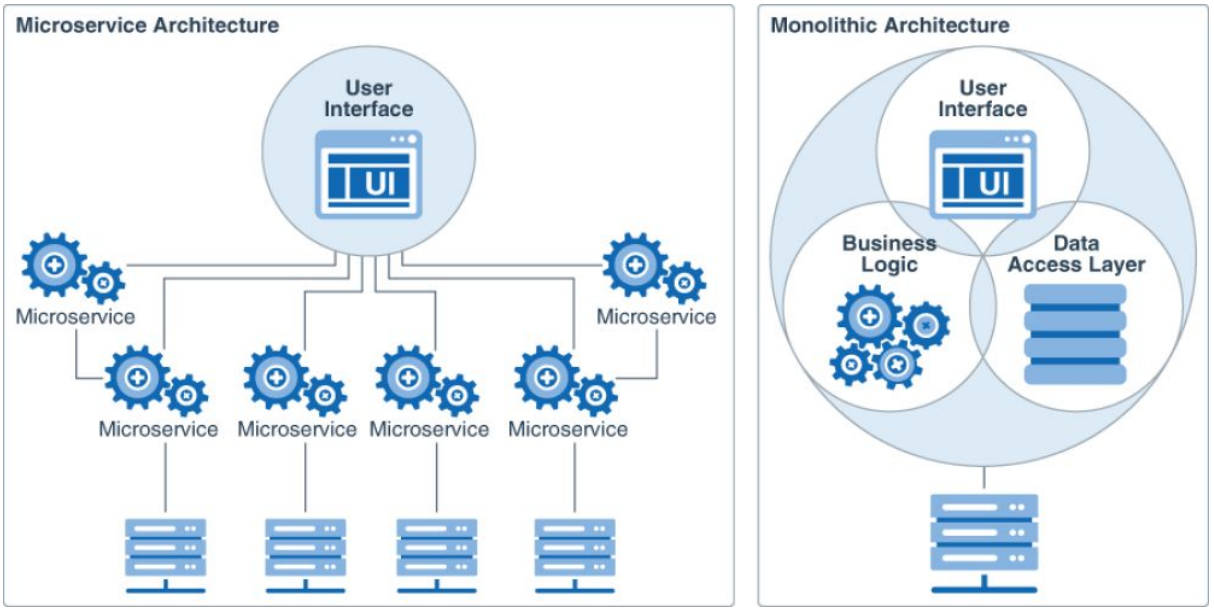
Что такое монолитная архитектура?

Монолитное приложение (назовем его монолит) представляет собой приложение, доставляемое через единое развертывание. Таким является приложение, доставленное в виде одной WAR (файл исполняемого кода) или приложение Node с одной точкой входа.

> ***Обратный прокси-сервер*** *(англ. reverse proxy) — тип прокси-сервера, который ретранслирует запросы клиентов из внешней сети на один или несколько серверов, логически расположенных во внутренней сети. При этом для клиента это выглядит так, будто запрашиваемые ресурсы находятся непосредственно на прокси-сервере. В отличие от классического прокси, который перенаправляет запросы клиентов к любым серверам в Интернете и возвращает им результат, обратный прокси непосредственно взаимодействует лишь с ассоциированными с ним серверами и возвращает ответ только от них.*

ПРИМЕР
Давайте представим классический интернет-магазин. Стандартные модули: UI, бизнес-логика и дата-слой. Возможны способы взаимодействия с сервисом: API REST и веб-интерфейс.

При построении монолита все эти вещи будут управляться внутри одного и того же модуля.



Достоинства

Большим преимуществом монолита является то, что его легче реализовать. В монолитной архитектуре вы можете быстро начать реализовывать свою бизнес-логику, вместо того чтобы тратить время на размышления о межпроцессном взаимодействие. Идеально для реализации MVP

Говоря об операциях, важно сказать, что монолит прост в развертывании и легко масштабируется. Для развертывания вы можете использовать скрипт, загружающий ваш модуль и запускающий приложение. Масштабирование достигается путем размещения Loadbalancer перед несколькими экземплярами вашего приложения. Как вы можете видеть, монолит довольно прост в эксплуатации.

Теперь давайте рассмотрим негативный аспект монолитной архитектуры.

Недостатки

Монолиты, как правило, перерождаются из своего чистого состояния в так называемый «большой шарик грязи». Вкратце это описывается как состояние, возникшее, потому что архитектурные правила были нарушены и со временем компоненты срослись.

Это перерождение замедляет процесс разработки: каждую будущую функцию будет сложнее развивать. Из-за того что компоненты растут вместе, их также необходимо менять вместе. Создание новой функции может означать прикосновение к 5 различным местам: 5 мест, в которых вам нужно написать тесты; 5 мест, которые могут иметь нежелательные побочные эффекты для существующих функций.

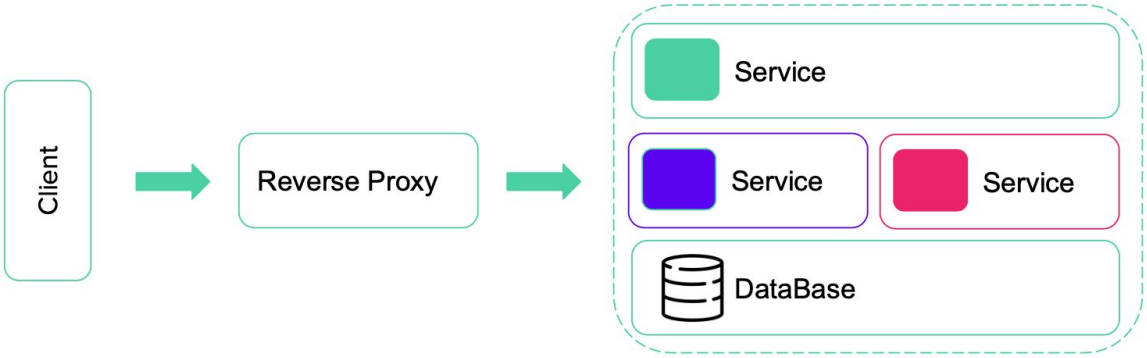
В монолите легко масштабировать. Это действительно так до тех пор, пока он не перерастёт в «большой шарик грязи», как упоминалось ранее. Масштабирование может быть проблематичным, когда только одной части системы требуются дополнительные ресурсы, ведь в монолитной архитектуре вы не можете масштабировать отдельные части вашей системы.

В монолите практически нет изоляции. Проблема или ошибка в модуле может замедлить или разрушить все приложение.

Строительство монолита часто протекает с помощью выбора основы. Отключение или обновление вашего первоначального выбора может быть затруднительным, потому что это должно быть сделано сразу и для всех частей вашей системы.

- низкая отказоустойчивость;
- отсутствие горизонтального масштабирования;
- применение одной технологии или языка и высокая трудоёмкость поддержки;
- сложность рефакторинга из-за хранения кода в одном месте и большое количество “старого кода”;
- трудности работы в команде разработчиков;
- чтобы использовать какой-то функционал повторно, придётся делать рефакторинг.

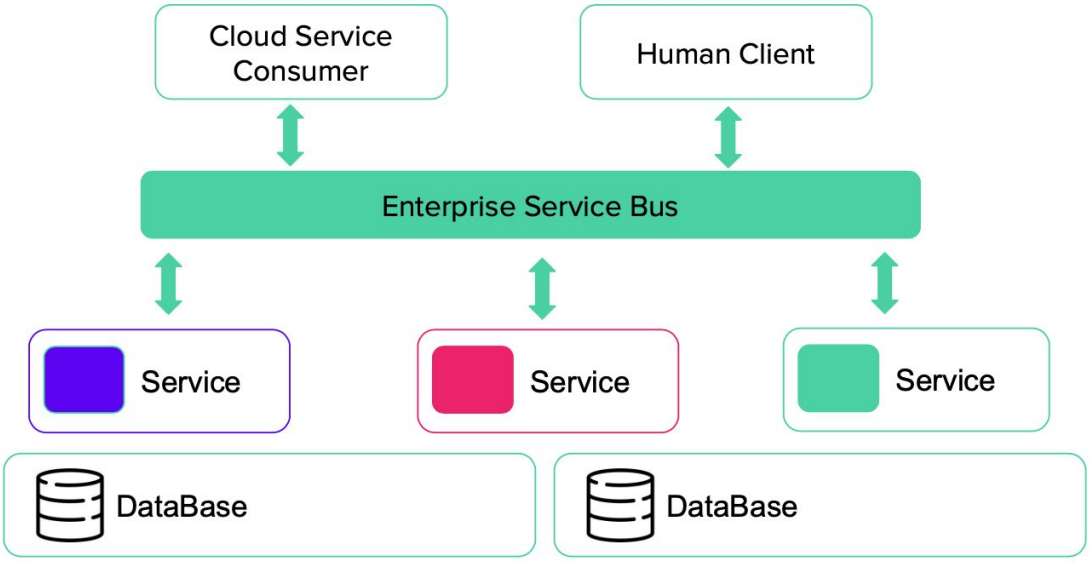
Второй по популярности вид архитектуры – пара монолитов, микс из монолита и сервисов или даже микросервисов. То есть вы сохраняете монолит, а доработки выполняете с использованием современных технологий.



Это частично решает проблемы отказоустойчивости, масштабируемости и одного стека технологий.

SOA

Сервис-ориентированная архитектура предусматривает модульность разработки и слабую связанность компонентов, поэтому получаем изолированную и распределенную систему.



Главный минус – общая шина данных Enterprise Service Bus с огромными спецификациями и сложностями работы с абстракциями и фасадами

> ***Сервисная шина предприятия*** *(англ. enterprise service bus, ESB) — связующее программное обеспечение, обеспечивающее централизованный и унифицированный событийно-ориентированный обмен сообщениями между различными информационными системами на принципах сервис-ориентированной архитектуры.*

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.

> ***Транза́кция*** *(англ. transaction) — группа последовательных операций с базой данных, которая представляет собой логическую единицу работы с данными.*

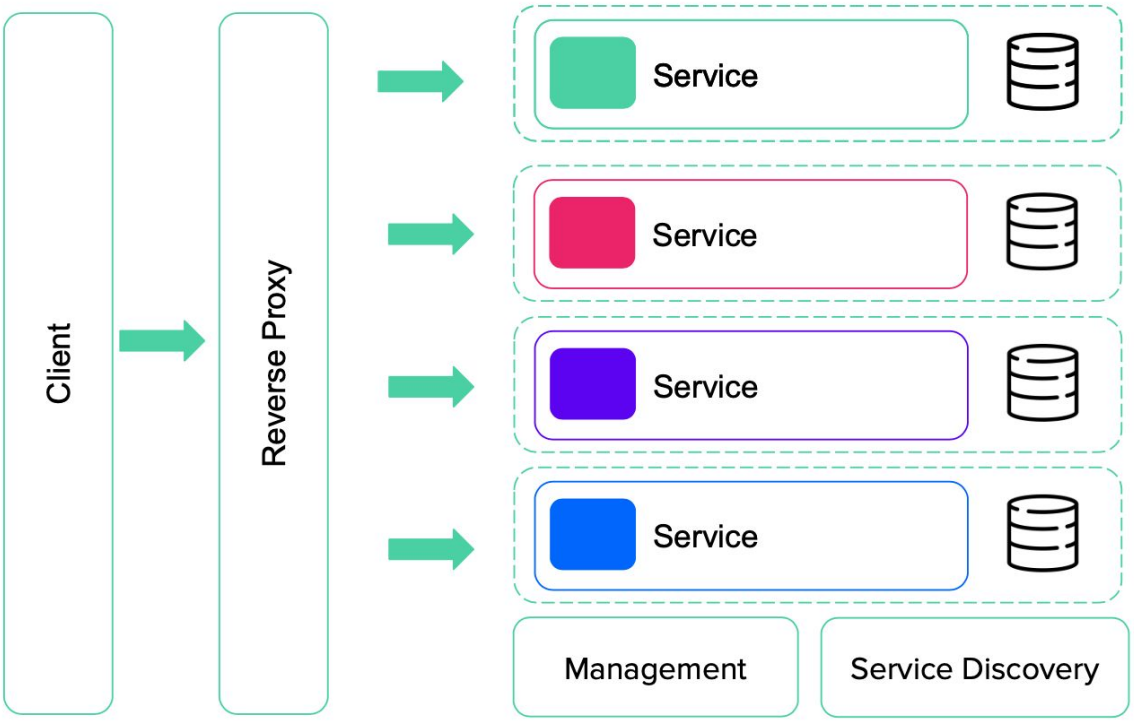
Наименование подобрано по аналогии с системной шиной компьютера, позволяющей подключать несколько устройств и передавать данные между ними по одному набору проводников.

«Сервисная шина предприятия» является зонтичным термином для набора возможностей, которые в разных реализациях трактуются несколько различными способами. Как правило, выделяются следующие ключевые возможности:

- поддержка синхронного и асинхронного способа вызова служб;
- использование защищённого транспорта, с гарантированной доставкой сообщений, поддерживающего транзакционную модель;
- статическая и алгоритмическая маршрутизация сообщений;
- доступ к данным из сторонних информационных систем с помощью готовых или специально разработанных адаптеров;
- обработка и преобразование сообщений;
- оркестровка и хореография служб;
- разнообразные механизмы контроля и управления (аудиты, протоколирование).

Конкретные программные продукты обычно также содержат готовые адаптеры для соединения с конкретным прикладным программным обеспечением, а также могут включать API для создания таких адаптеров.

Микро-сервисы



Что такое микросервисная архитектура?

В микросервисной архитектуре слабо связанные сервисы взаимодействуют друг с другом для выполнения задач, относящихся к их бизнес-возможностям.

Микросервисы в значительной степени получили свое название из-за того, что сервисы здесь меньше, чем в монолитной среде. Тем не менее, микро — о бизнес-возможностях, а не о размере.

По сравнению с монолитом в микросервисах есть несколько единиц развертывания. Каждый сервис развертывается самостоятельно.

ПРИМЕР
Давайте вновь рассмотрим в качестве примера Интернет-магазин.
Как и раньше, у нас есть: UI, бизнес-логика и дата-слой.
Здесь отличие от монолита состоит в том, что у всех вышеперечисленных есть свой сервис и своя база данных. Они слабо связаны и могут взаимодействовать с различными протоколами (например, REST, gRPC, обмен сообщениями) через свои границы.
На следующем рисунке показан тот же пример, что и раньше, но с разложением на микроуслуги.

Каковы преимущества и недостатки этого варианта?

Достоинства

Микросервисы легче держать модульными. Технически это обеспечивается жесткими границами между отдельными сервисами.

В больших компаниях разные сервисы могут принадлежать разным командам. Услуги могут быть повторно использованы всей компанией. Это также позволяет командам работать над услугами в основном самостоятельно. Нет необходимости координировать развертывание между командами. Развивать сервисы лучше с увеличением количества команд.

Микросервисы меньше, и благодаря этому их легче понять и проверить.

Меньшие размеры помогают, когда речь идет о времени компиляции, времени запуска и времени, необходимом для выполнения тестов. Все эти факторы влияют на производительность разработчика, так как позволяют затрачивать меньше времени на ожидание на каждом этапе разработки.

Более короткое время запуска и возможность развертывания микросервисов независимо друг от друга действительно выгодны для CI / CD. По сравнению с обычным монолитом он намного плавнее.

> ***Непрерывная интеграция*** *(CI, англ. Continuous Integration) — практика разработки программного обеспечения, которая заключается в постоянном слиянии рабочих копий в общую основную ветвь разработки (до нескольких раз в день) и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем. В обычном проекте, где над разными частями системы разработчики трудятся независимо, стадия интеграции является заключительной. Она может непредсказуемо задержать окончание работ. Переход к непрерывной интеграции позволяет снизить трудоёмкость интеграции и сделать её более предсказуемой за счёт наиболее раннего обнаружения и устранения ошибок и противоречий, но основным преимуществом является сокращение стоимости исправления дефекта, за счёт раннего его выявления.*
> ***Непрерывная доставка*** *(англ.* ***Continuous delivery*** *или* ***CD****, или* ***CDE****) — это подход к разработке программного обеспечения, при котором программное обеспечение производится короткими итерациями, гарантируя, что ПО является стабильным и может быть передано в эксплуатацию в любое время, а передача его происходит вручную. Целью является сборка, тестирование и релиз программного обеспечения с большей скоростью и частотой. Подход позволяет уменьшить стоимость, время и риски внесения изменений путём более частных мелких обновлений в продакшн-приложение.*

Микросервисы не привязаны к технологии, используемой в других сервисах. Значит мы можем использовать лучшие технологии. Старые сервисы могут быть быстро переписаны для использования новых технологий.

Недостатки

Все звучит довольно хорошо, но есть и недостатки.

Распределенная система имеет свою сложность: в ней вам приходится иметь дело с частичным отказом, более затруднительным взаимодействием при тестировании, а также с более высокой сложностью при реализации взаимодействия между сервисами.

Транзакции легче проводить в монолите, так как одна транзакция в монолите управляется в рамках одного сервиса, а в рамках микро-сервисной архитектуры одна транзация может затрагиват сразу несколько сервисов.

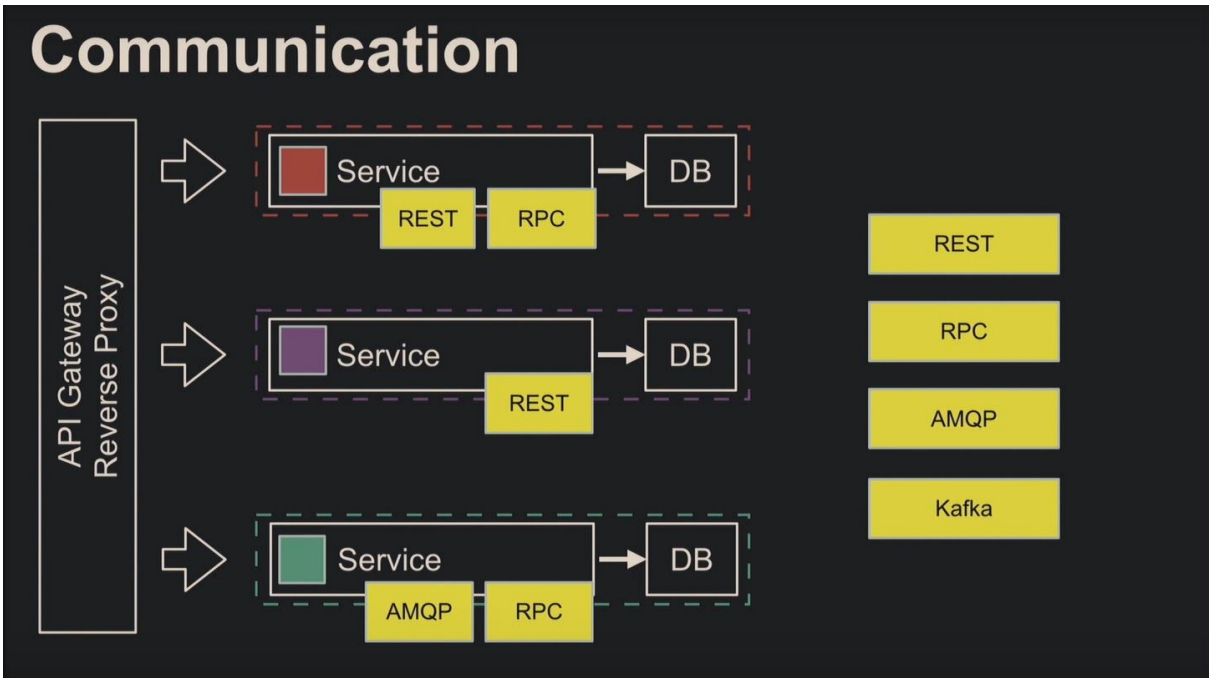
Существуют эксплуатационные накладные расходы, а множество микросервисов сложнее в эксплуатации, чем несколько экземпляров сигнального монолита.

Помимо вышеперечисленных сложностей, для микросервисов также может потребоваться больше оборудования, чем для традиционных монолитов. Иногда микросервисы могут превзойти один монолит, если есть его части, которые требуют масштабирования до предела. В тех случаях, когда бизнес-прцессы системы не позволяют изолировать достаточно маленькие сервисы, возникают отдельные сервисы “гиганты”, на которые может переходить много нагрузки. “Слабое звено цепи”

Изменения, затрагивающие несколько сервисов, должны координироваться между несколькими командами, а это может быть сложно, если команды еще не имели контактов.

Протоколы взаимодействия между сервисами

Выбор протоколов общения зависит от программиста. Например, вы используете REST для публичных запросов и RPC через AMQP для внутренних либо один общий протокол для всех.



Разделяют микросервисы с точки зрения либо бизнеса, либо программиста для переиспользования. Но мешают этому две вещи:

- внутренние связи – при тесном взаимодействии микросервисы объединяют;
- транзакции – у разных микросервисов базы данных изолированы, а нужна одна общая.

Что лучше? Монолит, SOA или Микросервисы

Не существует ответа на данный вопрос. В каждом отдельном случае этот вопрос необходимо решать отдельно.

Необходимо проанализировать организационную структуру и объема проекта. Если проект достаточно большой (есть 6 команд, которые будут работать над одним продуктом, но над разными отдельными функциями и сервисами), то подойдут микросервисы или SOA.

Есть проект не большой (например, команда из 3 разработчиков), то они будут хорошо строить и поддерживать монолит.

Другими факторами являются скорость изменения и сложность. Высокие темпы изменений и высокая сложность могут быть факторами, которые заставляют выбрать архитектуру микросервиса, даже при маленькой команде разработки.

Напротив, когда не очень хорошо знакомы с предметной областью, начать с монолита может быть полезно. Просто необходимо сохранить модульность. Это облегчит задачу, если вы когда-нибудь решите разделить свой монолит на несколько сервисов.

Интеграция. Брокеры

Взаимодействие интегрированных приложений

Для взаимодействия приложений обычно используются такие методы, как обмен файлами, общая база данных, удаленный вызов и асинхронный обмен сообщениями. В этом списке нет прямого обмена данными между базами данных приложений: этот метод ближе не к интеграции приложений, а не к перемещению данных. С точки зрения интеграции приложений важна возможность в процессе обмена данными выполнять какую-то содержательную обработку (например, при загрузке накладных пересчитывать товарные остатки). Прямой обмен данными, который обычно выполняется средствами класса ETL (extract, transfer, load) или самодельными утилитами, обычно такой возможности не предоставляет, так как предназначен исключительно для задач миграции данных из одной системы в другую, или для хранения данных в других местах, для улучшения уровня безопасности хранения данных и сокращения рисков их потери.

Обмен файлами

Обмен файлами пожалуй, самый распространенный подход к организации взаимодействия. Это связано с относительной простотой реализации, а также существованием стандартных (или «почти» стандартных) форматов обмена. Например, большая часть корпоративных информационных систем позволяет загружать и выгружать файлы, например, в формате CSV (Comma-Separated Values — «поля, разделенные запятыми»). Но у этого подхода есть и недостатки; если необходимо оперировать сложными структурами, то простые форматы обмена уже не пригодны. Возникающие в таких случаях специализированные форматы файлов должны «понимать» взаимодействующие системы, что ведет к жесткой зависимости систем друг от друга. Этот недостаток обычно преодолевают всевозможными утилитами конвертации данных. Кроме того, обычно обмен файлами подразумевает участие человека — кто-то должен выгрузить файл, скопировать его на другой компьютер, загрузить. Однако если интегрируемые методом обмена файлами системы имеют возможность автоматической загрузки/выгрузки (например, по расписанию), то данный подход позволяет построить полностью автоматизированное решение, которое вследствие своей простоты обладает высокой надежностью и пропускной способностью.

Общая база данных

Данный подход концептуально очень прост — несколько информационных систем или приложений используют одну базу данных. Главный его недостаток — связь между интегрированными приложениями настолько тесная, что иногда невозможно заметить границу между ними (обычно так интегрируются продукты одного производителя). Примером такого подхода могут служить большинство ERP-систем, где различные модули системы используют одну базу. Однако слишком тесная связь превращает конгломерат интегрированных приложений в монолит, в «суперсистему», отдельные части которой с трудом поддаются самостоятельной модернизации и замене. С этим борются, используя механизмы серверов баз данных (представления данных, промежуточные таблицы и т.п.), но далеко не всегда эффективно.

Удаленный вызов

Стандарты на удаленный вызов процедур возникли два десятка лет назад, позволяя программному коду, который выполняется на одном компьютере, вызывать код на другом. Стандарты появлялись, развивались и угасали: RPC, CORBA, DCOM, RMI… последним в этом

ряду стал протокол SOAP, основа современных Web-сервисов. Собственно в подходе к интеграции с использованием удаленных вызовов за эти годы ничего принципиально не изменилось — если приложению А что-то нужно от приложения Б, то А одним из перечисленных способов вызывает функцию приложения Б.
Основной недостаток удаленного вызова — требование работоспособности всех задействованных приложений в момент взаимодействия. Представьте себе систему ведения справочников, изменения из которой каждую ночь распространяются в десятки корпоративных систем. Вероятность того, что, скажем, в два часа ночи все корпоративные системы находятся в состоянии полной боеготовности, невелика. На этом «погорели» и мы, реализовав с помощью технологий Web-сервисов распространение справочников по корпоративным системам; все пришлось переписать.
Опыт показывает, что подход, основанный на удаленном вызове, приемлем только в тех случаях, когда взаимодействие приложений инициируется пользователем, который сам контролирует результат. Для автоматического взаимодействия без участия человека данный подход практически неприменим. В этом нет ничего удивительного: удаленные вызовы изначально были придуманы не для интеграции разных приложений, а для создания распределенных систем, когда компоненты одной системы могут работать на разных компьютерах.

Асинхронный обмен сообщениями

Это, пожалуй, единственный из перечисленных подходов, который создавался специально для интеграции информационных систем. Идея концептуально проста и напоминает работу электронной почты. Когда приложению А необходимо вызвать какое-то действие в приложении Б, оно формирует соответствующее сообщение с данными и инструкциями и отправляет его посредством системы доставки сообщений. Слово «асинхронный» означает, что приложение А не должно ждать, пока сообщение дойдет до Б, будет обработано, сформирован ответ и т.п. Сообщение гарантированно доставляется благодаря механизму очередей сообщений, которые снимают с взаимодействующих систем заботу о надежности сети передачи данных, работоспособности взаимодействующих систем в конкретные моменты времени и т.д.
Недостаток данного подхода — высокая цена разработки.

Брокер сообщений

> *Брокер сообщений - (англ. message broker, integration broker, interface engine) — архитектурный паттерн в распределённых системах; приложение, которое преобразует сообщение по одному протоколу от приложения-источника в сообщение протокола приложения-приёмника, тем самым выступая между ними посредником.*

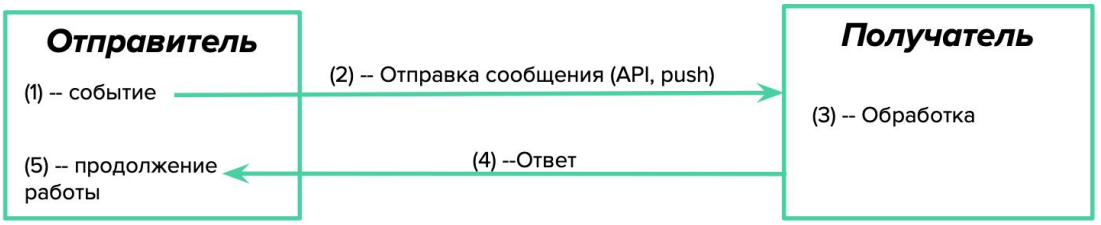
Кроме преобразования сообщений из одного формата в другой, в задачи брокера сообщений также входит:

- проверка сообщения на ошибки;
- маршрутизация конкретному приемнику(ам);
- разбиение сообщения на несколько маленьких, а затем агрегирование ответов приёмников и отправка результата источнику;
- сохранение сообщений в базе данных;
- вызов веб-сервисов;

- распространение сообщений подписчикам, если используются шаблоны типа издатель-подписчик.

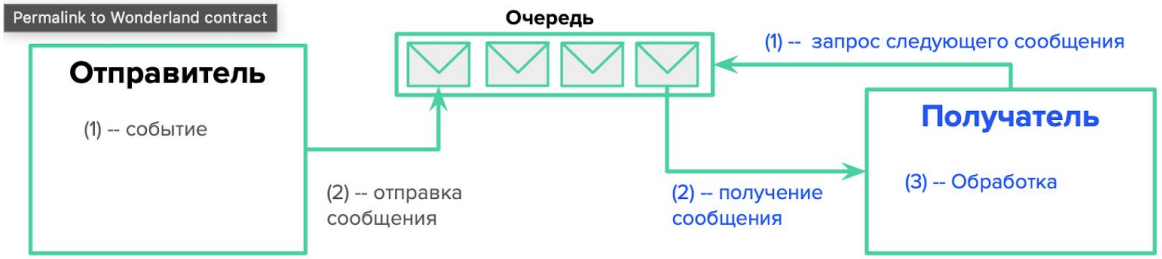
Использование брокеров сообщений позволяет разгрузить веб-сервисы в распределённой системе, так как при отправке сообщений им не нужно тратить время на некоторые ресурсоёмкие операции типа маршрутизации и поиска приёмников. Кроме того, брокер сообщений для повышения эффективности может реализовывать стратегии упорядоченной рассылки и определение приоритетности, балансировать нагрузку и прочее.

Прямое взаимодействие



- Взаимодействие через API
- Отправитель вынужден ждать завершения обработки получателем
- **Получатель может быть занят или недоступен**

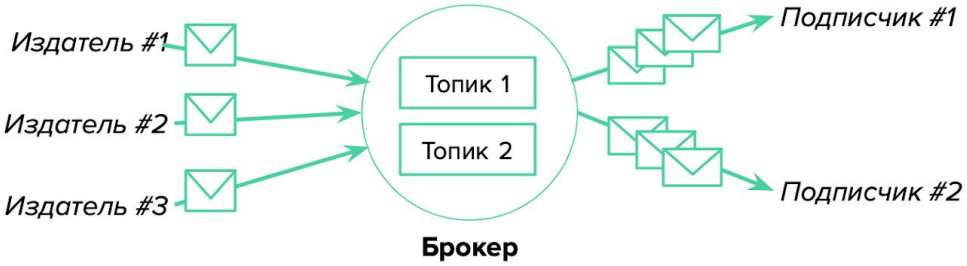
Очередь



- Отдельный сервис, отвечающий за доставку сообщений
- Отправитель может продолжить работу сразу после отправки сообщения
- Получатель обрабатывает сообщения асинхронно в порядке их поступлений (LIFO) или в произвольном порядке

Издатели-подписчики

- **Издатель (producer)** - процесс, отправляющий сообщения при наступлении события
- **Подписчик (consumer)** - процесс, получающий и обрабатывающий сообщения



- Одно и тоже сообщение могут получать одновременно несколько подписчиков
- **Топик** - отдельный “канал” или тема сообщений, позволяющая разделить сообщения по темам и отправлять подписчикам только те сообщения, которые им интересны

Нотации описания Архитектуры

Блок-схемы (схемы алгоритмов)

Данный тип схем (графических моделей) визуализирует разрабатываемые алгоритмы или процессы. В описании блок-схем принято отдельные стадии процессов изображать в виде блоков (отсюда и соответствующее название данной нотации). Блоки различаются в зависимости от семантики, содержащейся в представляемой ими форме. Отображаемые блоки соединены между собой линиями, имеющими направление и определенную алгоритмами последовательность.
В широком значении термина «блок-схемы» он является метанотацией. Различные спецификации «блок-схем» получили свое определенное назначение и форму представления необходимой информации, в зависимости от специфики каждой конкретной нотации, но при этом общие правила построения моделей наследуются от правил построения «блок-схем».

DFD-диаграмма (диаграмма потоков данных)

DFD – это нотация структурного анализа.
В данной нотации принято описывать внешние, по отношению к разрабатываемому продукту:

- Источники данных;
- Адресаты данных;
- Логические функции;
- Потоки данных;
- Хранилища данных.

Диаграмма потоков данных - это один из первых и основных инструментов структурного анализа и проектирования верхнеуровневой архитектуры программных продуктов, существовавших до последующего широкого распространения UML.

В основе данной нотации находится методология проектирования и метод построения модели потоков данных проектируемой информационной системы. В соответствии с соответствующей методологией проектирования модель системы идентифицируется как иерархия диаграмм потоков данных, которые представляют собой последовательный процесс преобразования данных, с момента их поступления в систему, до момента представления информации конечному (или псевдоконечному) пользователю.
Диаграммы потоков данных разработаны для определения основных процессов или модулей программного продукта. Далее, в целях комплексного представления разрабатываемой архитектуры, они детализируются диаграммами нижних уровней представления информации (DFD) и процессов (IDEF). Подобный принцип отображения данных продолжается, реализуя многоуровневую иерархию подчиненных и взаимосвязанных диаграмм. В результате будет достигнут нужный уровень декомпозиции, на котором процессы становятся абсолютно понятным и прозрачным.
В современных процессах разработки программного обеспечения подобный подход к проектированию программных продуктов практически не применяется, по причине смещения акцентов создания информационных систем от структурного к объектно-ориентированному подходу.

ER-диаграмма (диаграмма сущность-связь)

ER – это тип нотации, задача которой состоит в создании формальной конструкции, описывающей и визуализирующей верхнеуровневое представление бизнес объектов будущей системы, их атрибуты и связь между ними, в виде основных элементов, которые будут использоваться в качестве фундамента для проектирования таблиц будущей базы данных программного продукта. Таким образом, ER-диаграмма представляется как концептуальная модель создаваемой информационной системы, в терминах конкретной предметной области. Данный тип диаграмм помогает выделить ключевые сущности и определить связи между ними. На сегодняшний день существует достаточно мощный инструментарий, который может позволить выполнить преобразование созданной ER-диаграммы в конкретную схему базы данных на основе выбранной модели данных.

UML- диаграммы

UML (Unified Modeling Language) - не просто нотация или группа нотаций, а язык графического описания для объектного моделирования.
UML - это язык широкого профиля, который создан для формирования серии нотаций, поддерживающих полный цикл разработки, как архитектуры, так и функциональности программных продуктов, реализуемых по принципам объектно-ориентируемого программирования.
Широкая распространенность и повсеместное применение данного языка привели к тому, что он стал открытым стандартом, использующим графические обозначения для создания моделей систем, которые принято называть UML-моделями. Специфика UML не предполагает ориентацию на описание архитектуры, так как основное его назначение заключается в определении, визуализации, проектировании, документировании, преимущественно, программных продуктов, бизнес процессов и структур данных.
Язык UML получил популярность за счет жесткой связки с популярной и распространенной методологии разработки и внедрения программных продуктов RUP, в соответствии с которой была организована работа многих консалтинговых и производственных компаний отрасли информационных технологий. Не смотря на множество преимуществ, UML не является

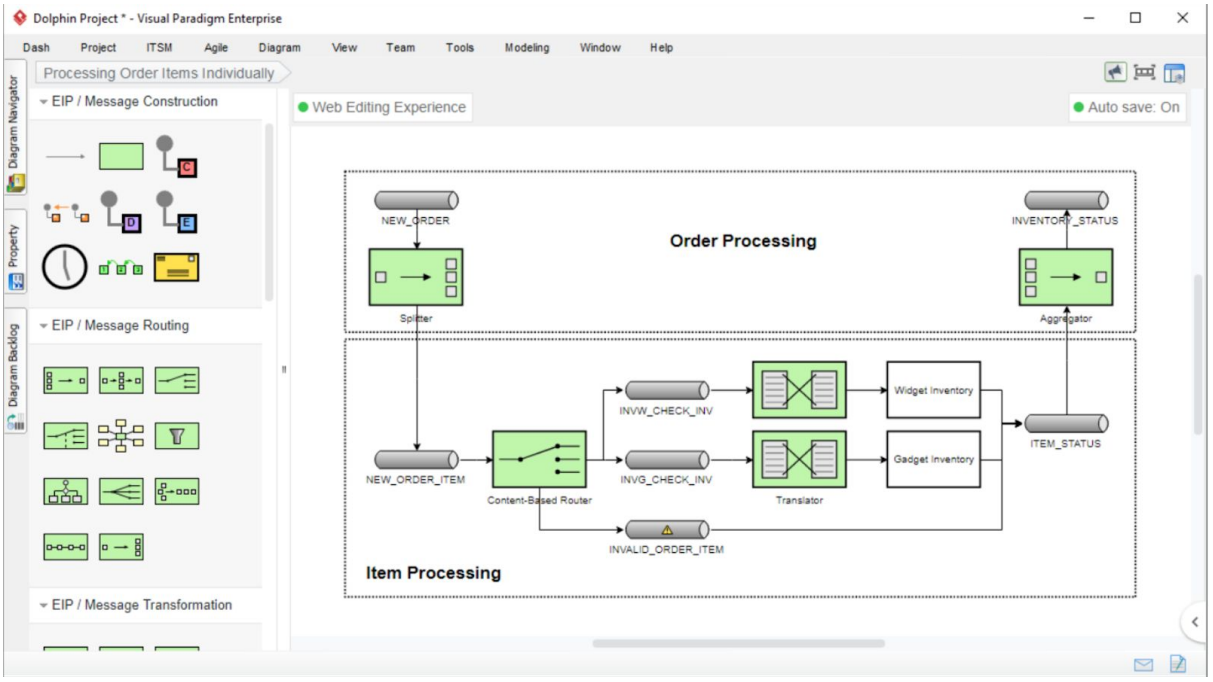
панацей и абсолютным универсумом в процессах проектирования архитектур и программных продуктов.
Применять и разбираться в диаграммах UML могут только узкоспециализированные специалисты области проектирования программных систем. Вовлечь в обсуждение и согласование разрабатываемой архитектуры информационного продукта, задокументированного на UML, стэйкхолдеров, представителей не направления информационных технологий, будет достаточно сложно.
UML, в отличие от ER и BPMN диаграмм, поддерживает генерацию не просто отдельной функциональности, привязанной к специфичным типам информационных систем, а целостного программного кода на основании сформированных UML-моделей.
Концепция использования UML для построения Архитектур Информационных систем “4+1” https://en.wikipedia.org/wiki/4%2B1_architectural_view_model.

EIP - диаграммы

Самая специфичная и наименее распространенная нотация из всех рассматриваемых в сегодня.

EIP (Enterprise Integration Patterns) диаграммы - это набор из 65 шаблонов диаграмм, которые предназначены для описания специализированных процессов взаимодействия между программными продуктами. Специфика данной нотации состоит в том, что она разработана для описания конкретных процессов, обеспечивающих интеграционное взаимодействие между приложениями и функциональность программных продуктов, ориентированных на обмен сообщениями между информационными системами.

Шаблоны интеграции приложений и необходимая функциональность, описание которых поддерживают EIP диаграммы, встроены в множество современных специализированных открытых программных продуктов и доступны для применения в специализированных интеграционных процессах.



ПРИМЕР
tool и примеры - шаблоны
https://online.visual-paradigm.com/diagrams/features/enterprise-integration-patterns-diagram-tool/

Модель - Прототип - MVP - и тд

Модели компонентов, программных продуктов и функциональных процессов

Модели, как таковые, не являются нотациями, а представляют собой набор необходимых для реализаций процессов/подпроцессов, представленных в виде нескольких взаимодополняющих друг друга нотаций, с целью моделирования конечного результата. Говоря о модели, со структурной точки зрения, корректно будет сказать, что это «метанотация», включающая в себе несколько основных нотаций, реализация диаграмм которых запланированы для качественной разработки конкретной архитектуры.
Таким образом, модель – это абстракция, которая фокусирует субъекта, работающего с ней на основных, критичных для реализации, характеристиках конечного продукта.

Прототипы

Когда мы говорили о моделях, то отметили, что модель это следующий, качественный шаг вперед к реализации программного обеспечения, по сравнению с диаграммами, реализуемыми нотациями.
После того, как реализованы несколько моделей требований, можно приступать к реализации прототипов, если это предусмотрено и спланировано в проекте разработки программного продукта.
Прототип представляет собой «черновой» вариант конечной реализации программного продукта или его компонента, разработанного с учетом определенных ограничений для демонстрации заинтересованным сторонам на определенной стадии разработки только архитектуры или программного продукта в целом.
Цель разработки прототипа – подтвердить ожидания стэйкхолдеров от реализации конечного продукта и согласовать её функциональность. Прототипы принято реализовывать в специализированном программном коде, поэтому их следует воспринимать как часть процесса разработки, но при этом не рассчитывать на то, что прототип будет являться частью будущей архитектуры или информационной системы.
Прототипы, как правило, разрабатываются достаточно быстро с помощью специализированного инструментария, освоение которого требует гораздо меньше времени и сил, чем освоение определенного языка программирования.
Жизненный цикл прототипа заканчивается в тот момент, когда ожидания стэйкхолдеров подтверждены и разработанный облик (не более) будущего продукта запланирован к реализации;

MVP

В случае если бизнес-процессы изучены не глубоко и функционала Прототипа не достаточно, разрабатывается MVP. Чаще всего это ограниченное в функциональности приложение с Монолитной архитектурой.

Каждый крупный программный продукт или его модуль, который состоит из совокупности компонентов и связей между ними, должен быть детально описан в соответствующий момент процесса архитектурного проектирования. Описание должно быть выполнено по стандартизированным правилам организации или проекта, с использованием необходимых нотаций, для данного конкретного случая.

По принципам программной инженерий и здравой логики, которая находится в основе процессов разработки информационных систем, любая подсистема или компонент в дальнейшем может выступать в качестве составного элемента более масштабной системы. Поэтому, в архитектурном проектировании должно обязательно содержаться подробное описание укрупнённых частей системы, выполненных с помощью CASE средств и нотаций, согласованных друг с другом и обеспечивающих последующую преемственность.
Важно, чтобы выбранная нотация или серия нотаций поддерживали существующий процесс архитектурного проектирования, способствовали его развитию и совершенствованию и были «в силах» поддержать соответствующую фиксацию функциональных, нефункциональных и прочих требований к разрабатываемой архитектуре и программному продукту в целом.
Необходимо осознавать, что нет единой нотации или языка проектирования, который мог бы быть решением всех поставленных задач процесса проектирования. Сегодня существует достаточное количество универсальных инструментов и средств, но ни один из них не может обеспечить весь спектр возникающих задач проектирования.
Для того, чтобы создать достаточно комплексную и взаимосвязанную среду проектирования архитектур, моделей и функциональности программных продуктов нужно использовать инструменты и диаграммы, которые смогут поддержать процессы разработки, в зависимости от специфики конкретной задачи, но при этом должен существовать минимальный набор необходимых диаграмм, который должен присутствовать в каждом проекте по разработке программного продукта не смотря на его масштабы и значимость.

Облака

Something as a Service

Хорошая статья, я лучше не напишу =)
https://www.ispsystem.ru/news/xaas

> ***Виртуализа́ция*** *— предоставление набора вычислительных ресурсов или их логического объединения, абстрагированное от аппаратной реализации, и обеспечивающее при этом логическую изоляцию друг от*

друга вычислительных процессов, выполняемых на одном физическом ресурсе.

Виртуализация — это когда вместо физической версии создается имитированная или виртуальная вычислительная среда. Виртуализация часто включает в себя созданные компьютером версии аппаратных средств, операционных систем, устройств хранения и многое другое. Это позволяет организациям разделить один компьютер или сервер на несколько виртуальных машин. Таким образом, каждая виртуальная машина работает независимо и выполняет разные операционные системы или приложения, при этом совместно используя ресурсы одного хост-компьютера.

Виртуализация оптимизирует масштабируемость и рабочие нагрузки за счет нескольких ресурсов, созданных на одном компьютере или сервере, что позволяет уменьшить совокупное количество серверов, понизить энергопотребление и сократить стоимость инфраструктуры и ее обслуживания. Существует четыре основных типа виртуализации. Первый — это виртуализация настольных систем, которая позволяет разместить несколько отдельных рабочих столов и управлять ими на одном централизованном сервере. Второй тип — это виртуализация сети, предназначенная для разделения пропускной способности сети на отдельные каналы и их последующего назначения определенным серверам или устройствам. Третьим типом является виртуализация программного обеспечения, которая отделяет приложения от аппаратных средств и операционной системы. Наконец, четвертый тип — это виртуализация хранилища, с помощью которой несколько сетевых хранилищ объединяются в одно устройство хранения, к которому могут получить доступ несколько пользователей.

12 главных принципов построения cloudnative архитектуры

https://12factor.net/ru/

Риски

Совокупная стоимость владения системой включает плановые затраты на создание и эксплуатацию системы и стоимость рисков, а именно, бизнес-рисков и вероятностями технических рисков а также соответствием между ними. Рассмотрим ряд наиболее значимых типов рисков:

- риски проектирования при разработке системы;
- технические риски, которые реализуются в виде отказов, простоев, потерь данных;
- риски бизнес-потерь, возникающие при эксплуатацией системы (бизнес-риски). Они обусловлены, в свою очередь, техническими рисками. Например, отсутствие реакции или слишком большое время реакции на действия пользователя приводит к его отказу от использования данной системы;
- риски бизнес-потерь, связанные с многообразием бизнес- процессов (так называемые неопределенности). Бизнес-процессы систематически изменяются, а это, в свою очередь, требует своевременного изменения и ИС. При этом бизнес какое-то время может функционировать неоптимально, что приводит к потерям. Кроме того, неизбежно

возникают затраты за модификацию системы. Бизнес-риск может быть нейтрализован соответствующей организацией процесса и/или архитектурным решением.

Риски проектирования

Риски, возникающие на этапе проектирования, играют важную роль в процессе разработки программного обеспечения и требуют особого внимания, так как основные технические вопросы решаются именно на этом этапе.

1. Сложность архитектуры программного обеспечения

Архитекторы программного обеспечения не всегда придерживаются правила «Не порождайте сущностей сверх необходимого». Но ведь именно простота, понятность и единая концептуальная целостность обеспечивают эффективную реализацию системы.

2. Неудобный пользовательский интерфейс

Разработка пользовательского интерфейса является частью любого проекта, связанного с созданием программного обеспечения. Интерфейс пользователя является точкой взаимодействия человека и программы, зачастую имеющей сложную функциональность. Именно через интерфейс пользователь судит о программе в целом; более того, часто решение об использовании программного обеспечения пользователь принимает по тому, насколько ему удобен и понятен пользовательский интерфейс. Следовательно, от того насколько удобным будет разработанный интерфейс пользователя будет зависеть и успех продукта.

3. Неправильная структура базы данных

При разработке программного обеспечения проектирование базы данных требует особого внимания и ответственности, так как стоимость допущенных на этом этапе ошибок особенно велика. Проблемы, которые могут произойти на этапе проектирования базы данных:

- некорректность схемы базы данных по отношению к предметной области;
- несоответствие аппаратным ограничениям;
- сложность и неудобная работа с базой данных;
- невозможность к поддержке и сопровождению.

4. Неоптимальный выбор структур данных

Структуры данных влияют на эффективность алгоритмов и, следовательно, на производительность программного обеспечения.

5. Неоптимальный выбор языка программирования

При разработке программного обеспечения имеется огромный выбор языков программирования, в лабиринтах которых можно легко заблудиться. Для того чтобы выбор языка программирования оказал благоприятное влияние на реализацию системы, необходимо учитывать следующие факторы:

- целевая платформа;
- гибкость языка;
- время реализации;

- производительность;
- сопровождение программного обеспечения;
- предметная область разрабатываемой системы;
- необходимость в использовании библиотек;
- опыт разработчиков.

Риски разработки

Основные риски на этапе программирования описаны ниже.

1. Изобретение «велосипеда»

Написание кода без использования возможностей языка и существующих библиотек.

2. Нечитаемый код

Последствие этого риска – трудность в изменении и сопровождении программного обеспечения.

3. Создание программных закладок

Программная закладка – это внесенные в программное обеспечение функциональные объекты, которые при определенных условиях (входных данных) инициируют выполнение не описанных в документации функций, позволяющих осуществлять несанкционированные воздействия на информацию.
На этапе разработки программистам достаточно легко внедрить в код системы программную закладку. В настоящее время известно 30 публичных случаев обнаружения программных закладок. В большинстве случаев закладки успешно использовались злоумышленниками и были обнаружены уже после активации.

4. Нерегулярное резервное копирование кода

Последствие этого риска очень масштабно.

Термины и определения

Термин	**Определение**
Архитектура ИС	набор значимых решений по поводу организации системы программного обеспечения, набор структурных элементов и их интерфейсов, при помощи которых компонуется система вместе с их поведением, определяемым во взаимодействии между этими элементами, компоновка элементов в постепенно укрупняющиеся подсистемы, а также стиль архитектуры, который направляет эту

	организацию (элементы и их интерфейсы, взаимодействие и компоновка).
Распределенная система	система, которая работает сразу на множестве машин, образующих цельный кластер (набор компьютеров/серверов, объединенных сетью и взаимодействующих между собой). система, которая работает сразу на множестве машин, образующих цельный кластер (набор компьютеров/серверов, объединенных сетью и взаимодействующих между собой).
Сервисная шина предприятия	связующее программное обеспечение, обеспечивающее централизованный и унифицированный событийно-ориентированный обмен сообщениями между различными информационными системами на принципах сервис-ориентированной архитектуры.
Микросервисная архитектура	вариант сервис-ориентированной архитектуры программного обеспечения, направленный на взаимодействие насколько это возможно небольших, слабо связанных и легко изменяемых модулей — микросервисов
Обратный прокси-сервер (англ. reverse proxy)	тип прокси-сервера, который ретранслирует запросы клиентов из внешней сети на один или несколько серверов, логически расположенных во внутренней сети. При этом для клиента это выглядит так, будто запрашиваемые ресурсы находятся непосредственно на прокси-сервере.
Модель	это метанотация, которая фокусирует на основных, критичных для реализации, характеристиках конечного продукта.
Прототип	это черновой вариант конечной реализации программного продукта или его компонента.
MVP	обладающий минимальными, но достаточными для удовлетворения первых потребителей функциями.
Вертикальная масштабируемость	это наращивание ресурсов, то есть увеличение количества ядер, оперативной памяти и т.д. на одном сервере.
Горизонтальная масштабируемость	это добавление новых серверов с более или менее любыми характеристиками в вычислительный кластер.
Отказоустойчивая система	это система, где отсутствует единая точка отказа, их конфигурацию можно корректировать, подстраиваясь под случающиеся отказы. Единая точка отказа характерная для нераспределенных систем. Например, если один сервер откажет, то вся система отключается. Увеличение количества точек отказа приводит к повышению отказоустойчивости
Сервисная шина предприятия (англ. enterprise service bus, ESB)	связующее программное обеспечение, обеспечивающее централизованный и унифицированный событийно-ориентированный обмен сообщениями между различными информационными системами на принципах сервис-ориентированной архитектуры.

Транза́кция (англ. transaction)	группа последовательных операций с базой данных, которая представляет собой логическую единицу работы с данными.
Непрерывная интеграция (CI, англ. Continuous Integration)	практика разработки программного обеспечения, которая заключается в постоянном слиянии рабочих копий в общую основную ветвь разработки (до нескольких раз в день) и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем. В обычном проекте, где над разными частями системы разработчики трудятся независимо, стадия интеграции является заключительной. Она может непредсказуемо задержать окончание работ. Переход к непрерывной интеграции позволяет снизить трудоёмкость интеграции и сделать её более предсказуемой за счёт наиболее раннего обнаружения и устранения ошибок и противоречий, но основным преимуществом является сокращение стоимости исправления дефекта, за счёт раннего его выявления.
Непрерывная доставка (англ. Continuous delivery или CD, или CDE)	это подход к разработке программного обеспечения, при котором программное обеспечение производится короткими итерациями, гарантируя, что ПО является стабильным и может быть передано в эксплуатацию в любое время, а передача его происходит вручную. Целью является сборка, тестирование и релиз программного обеспечения с большей скоростью и частотой. Подход позволяет уменьшить стоимость, время и риски внесения изменений путём более частных мелких обновлений в продакшн-приложение. CD отличается от непрерывного развёртывания, в котором ПО также производится короткими циклами через автоматическое развёртывание, в противоположность ручному.
Брокер сообщений (англ. message broker, integration broker, interface engine)	архитектурный паттерн в распределённых системах; приложение, которое преобразует сообщение по одному протоколу от приложения-источника в сообщение протокола приложения-приёмника, тем самым выступая между ними посредником.
Виртуализа́ция	предоставление набора вычислительных ресурсов или их логического объединения, абстрагированное от аппаратной реализации, и обеспечивающее при этом логическую изоляцию друг от друга вычислительных процессов, выполняемых на одном физическом ресурсе.

Источники

- Максим Смирнов https://mxsmirnov.com
  Канал «Архитектура ИС» в Telegram: https://t.me/it_arch

- https://proglib.io/p/po-stopam-luchshih-mikroservisnaya-arhitektura-v-razreze-2019-11-07
- https://spravochnick.ru/bazy_dannyh/bazy_dannyh_vvedenie/arhitektura_informacionnoy_sistemy/#trehurovnevaya-arhitektura
- http://it-claim.ru/Education/Course/ISDevelopment/Lecture_3.pdf
- http://it-claim.ru/Education/Course/ISDevelopment/L_3.pdf
- https://studme.org/154671/informatika/arhitektura_informatsionnyh_sistemhttps://warren2lynch.medium.com/enterprise-integration-patterns-eip-tutorial-f6d7134f67ae
- https://www.osp.ru/os/2006/09/3776464
- https://habr.com/ru/post/261171/
- ГОСТ Р 51275-99. Защита информации. Объект информатизации. Факторы, воздействующие на информацию. — Москва: Изд-во стандартов, 2003. – 12 с.
- ДеМарко Т., Листер Т. Вальсируя с медведями: управление рисками в проектах по разработке программного обеспечения. — М.: Компания p.m.Office, 2005. – 208 с
- https://www.enterpriseintegrationpatterns.com
- https://pubs.opengroup.org/architecture/o-aa-standard/#_evolutionary_architecture