1. Какими качествами должны обладать требования

Полнота

Непротиворечивость

Последовательность изложены

Понятность

2. Виды взаимодействия (синхронное и асинхронное)

Синхронное отправляем запрос и система блокируется, пока не получим запрос от сервера. Регситрация или отображение товаров, пока загружается главная страница, остальное не появляется

Асинхронное - когда пользователь не ждет, сайт отправил запрос и сервер долго обрабатывает запрос, что портит uix, например отправляет документ на подписание, чтобы не держать на экране задача ставится в очередь и он может пользоваться другими частями системы

3. Первичный уникально идентифицирует строку. Вторичный ключ – устанавливает связь с соседней таблицей.

4. Бизнес требования - требования. Которые помогают бизнесу добиться свои цели (заработать деньги или сократить расходы) путем привлечения пользователей, всегда связаны с ценностью бизнесу

   Функциональные требования - как именно будем реализовывать бизнес требования функционально, описывают поведения, функции системы

5. Документирование требований – текст, диаграммы, таблицы, прототип, регламенты для спецификации userStory, RSS, RAP, UseCase, поведенческие диаграммы, потоки данных. Смотрим что лучше ложится под структуры, которые сейчас в работе, чтобы процесс был четко описан. Иногда сложно воспринимать алгоритм в виде текста, поэтому используем диаграммы.

6.

   Rest что это и SOAP. Rest архитектурный стиль, содержит набор 7 правил. Это HTTP. SOAP – протокол. Они не в одной линейки, но с точки зрения интеграции. Это XML. Это формат обмена сообщений. Очень стандартизированный и очень избыточен. Для строгих правил его используют, и по безопасности в банковских он использует.

7. Зачем API – интерфейс для связи с системой, чтобы общаться с системой. Набор правил, что присылают на вход и что получают в ответ.

8. JSON – формат данных в виде ключ значение, есть типы int, string, numeric, массивы, bool

9. Диаграмма последовательности — sequence diagram - подвид диаграмм взаимодействия, который позволяет описать взаимодействие между объектами в системе в виде последовательности сообщений, действий и операций, отображая порядок выполнения действий и обмена информацией между объектами во времени. Диаграмма состоит из вертикальных линий жизни и горизонтальных стрелок. Линии представляют отдельные объекты, а горизонтальные стрелки — сообщения и операции, передаваемые между объектами или участниками. В

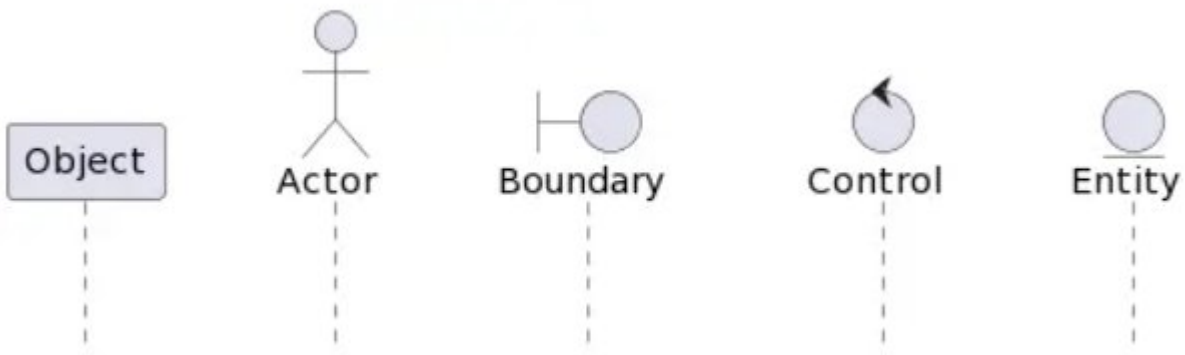
диаграмме объектами выступают участники системы, такие как акторы (Actor), границы (Boundary), контроллеры (Control) и сущности (Entity). Они называются участниками (Participants) и отображаются в виде пиктограмм или прямоугольника с названием.

Акторы (Actor) представляют пользователей или другие системы, взаимодействующие с системой, которая описывается на диаграмме. Они могут вызывать действия, которые система выполняет в ответ на их запросы. Например, актор может запросить у системы информацию и получить ответ.

Границы (Boundary) определяют внешние границы системы и представляют собой точки входа или выхода, через которые система взаимодействует с внешним миром. Например, граница может представлять интерфейс пользователя, через который пользователь взаимодействует с системой.

Контроллеры (Control) обрабатывают запросы и управляют потоком данных в системе. Они представляют собой узлы, через которые проходят данные и управляющие выполнением операций в системе. Например, контроллер может обрабатывать запросы, которые поступают от акторов и возвращать результаты.

Сущности (Entity) представляют данные и хранят состояние системы. Они могут быть представлены как базы данных или другие системы хранения данных. Например, сущность может обрабатывать запросы на чтение или запись данных.



https://itonboard.ru/analysis/394-diagramma_posledovatelnosti_sequence_diagrams_uml/

10. CTE это Common Table Expression — общие табличные выражения, их еще называют конструкциями с WITH. Фактически это создание временных таблиц, но существующих только для одного запроса, а не для сессии. К ним можно обращаться внутри этого запроса. Такой запрос хорошо читается, он понятен, его легко видоизменять, если потребуется. Это очень востребованная вещь, и она в PostgreSQL давно. WITH Aeroflot_trips AS (SELECT TRIP.* FROM Company INNER JOIN Trip ON Trip.company = Company.id WHERE name = "Aeroflot")SELECT plane, COUNT(plane) AS amount FROM Aeroflot_trips GROUP BY plane;

Данные передаются в виде XML-сообщений, которые состоят из следующих частей:

- Envelope (конверт) — корневой элемент, обязательный элемент;
- Header (заголовок) — содержит параметры (атрибуты) для обработки сообщения, необязательный элемент;
- Body (тело) — содержит сообщение, обязательный элемент;
- Fault (ошибка) — содержит информацию об ошибках, необязательный элемент.

11. Индексы предоставляют путь для быстрого поиска данных на основе значений в этих столбцах. Например, если вы создадите индекс по первичному ключу, а затем будете искать строку с данными, используя значения первичного ключа, то *SQL Server* сначала найдет значение индекса, а затем использует индекс для быстрого нахождения всей строки с данными. Без индекса будет выполнен полный просмотр (сканирование) всех строк таблицы, что может оказать значительное влияние на производительность.

Кластеризованный индекс

Кластеризованный индекс хранит реальные строки данных в листьях индекса. Возвращаясь к предыдущему примеру, это означает что строка данных, связанная со значение ключа, равного 123 будет храниться в самом индексе. Важной характеристикой кластеризованного индекса является то, что все значения отсортированы в определенном порядке либо возрастания, либо убывания. Таким образом, таблица или представление может иметь только один кластеризованный индекс. В дополнение следует отметить, что данные в таблице хранятся в отсортированном виде только в случае если создан кластеризованный индекс у этой таблицы.
Таблица не имеющая кластеризованного индекса называется кучей.

Некластеризованный индекс

В отличие от кластеризованного индекса, листья некластеризованного индекса содержат только те столбцы (*ключевые*), по которым определен данный индекс, а также содержит указатель на строки с реальными данными в таблице. Это означает, что системе подзапросов необходима дополнительная операция для обнаружения и получения требуемых данных.

- 12. Отличие бизнес аналитика от системного - Бизнес-аналитик анализирует процессы компании, чтобы сделать бизнес более прибыльным и удобным для клиентов. Он исследует действия сотрудников, выявляет проблемы и разрабатывает схемы процессов для общего понимания ситуации. **Системный аналитик, в свою очередь, проектирует функции системы, которые помогут улучшить бизнес-процессы. Основное отличие между этими ролями в том, с кем они общаются**: бизнес-аналитик чаще взаимодействует с бизнес-заказчиками, а системный аналитик работает с техническими специалистами: разработчиками и

тестировщиками. **Владелец продукта проекта (Product Owner)** — отвечает за развитие продукта, а также планирование, координацию и контроль выполнения проекта, управляет сроками и ресурсами. Ставит задачу аналитику: доработка продукта (я совмещаю роли бизнес-аналитика и системного).

- **Бизнес-аналитик (Business Analyst)** — выясняет требования клиентов и анализирует бизнес-процессы для определения потребностей разработки.
- **Системный аналитик (System Analyst)** — работает с техническими специалистами для проработки архитектуры системы и технических требований. Ставит задачи разработчикам.
- **Frontend-разработчик** — занимается разработкой пользовательского интерфейса.
- **Backend-разработчик** — отвечает за серверную часть и логику приложения.
- **UI/UX-дизайнер (UI/UX Designer)** — разрабатывает удобный и привлекательный интерфейс, отвечает за опыт пользователя.
- **Тестировщик (QA Engineer)** — проводит тестирование программного обеспечения для выявления ошибок и обеспечения качества продукта. Проверяет, чтобы все требования, описанные в техническом задании, были реализованы.
- **DevOps-инженер** — отвечает за автоматизацию процессов разработки и развёртывания, а также за инфраструктуру и системное администрирование.
- **Архитектор программного обеспечения (Software Architect)** — проектирует архитектуру системы, выбирает технологии и платформы, определяет стандарты разработки.

13. BPMN-диаграмму (Нотация BPMN (Business Process Modeling Notation), Нотация моделирования бизнес-процессов) — это метод составления блок-схем, отображающий этапы выполнения бизнес-процесса от начала до конца. BPMN-схемы наглядно и подробно демонстрируют последовательность рабочих действий и перемещение информационных потоков, необходимых для выполнения процесса. Также могу использовать диаграмму активностей (activity diagram) https://www.comindware.ru/blog/нотация-bpmn-2-0-элементы-и-описание/

14. Диаграмма активности UML позволяет более детально визуализировать конкретный случай использования. Это поведенческая диаграмма, которая иллюстрирует поток деятельности через систему.

Диаграммы активности UML также могут быть использованы для отображения потока событий в бизнес-процессе. Они могут быть использованы для изучения бизнес-процессов с целью определения их потока и требований. https://creately.com/blog/ru/uncategorized-ru/учебник-по-диаграмме-активности/

https://habr.com/ru/articles/843942/

15. Зачем нам UML – для плнирования и моделирования ведь вносить изменения в диаграммы легче, чем в код. Унифицированный язык моделирования. Какие диаграммы есть – диаграмма посдеовательностей, деятельности, классов.

Состояний – поведение отдельно взятого объекта при определенных условиях.

Классов – рассказать о коде без коды, показввает классы интерфейсы и взаимодействия между ними (паттерны программмрования можно ими описать). Дает представление о программном коде. пРимер класс посетитель.

Деятельности – логику процедур, бизнес-процессы и потоки работ.

BPMN – ссхемы моделирования бизнес-процесслв. Все сценарии взаимоджействия пользователей и системы. Это сиситема условных обозначений обозначающая процесс в виде блок-схем.

Нотация система условных обозначений. Набор элементов и символов, который комбинируется к схему.

Как соединить микросервисы между собой

МА микросервисы запускаются отдельно. Сервисы в облаках развернуты, общаются по сети. Какой сейчас рабоатет какой упал, принял или нет пакет сервис. (Задача двух генералов нужно скоорденировать их указы, надо отправить гонца). Еще проблема нагрузка на трафик. Общение надо оптимизировать. Если использовать асинхронные системы, надо хранить информацию какое-то время, что требует утилизацию памяти или диска. Решение должно быть отказоустойчивым, чтобы по цепочки серверы не упали один за другим.

Решения разные:

Иделаьного варианта есть. Синхронные вызовы – делаем запрос и ждем ответ. Синхронные REST-like

gRPC- RPC удаленные вызов процедур на бинарном формате поверх HTTP2 от гугл

SOAP- RPC с форматом xml. Синх

Асинхронные – отправляем соообщение, а ответ придет когда-нибудь потом или он вообще не предусмотрен.

Мессенджинг rabbirMq, ZeroMq, ActiveMq

Стриминг – Kafka

REST API

Легкий старт, обращаются по ендпоинту, легко дебажить и логировать, легко прочитать json, синхронная апрадигма запрос-ответ

Син-если сервис недоступен нельзя отложить задачу на позднее время

ПИр то пир- нужна обращаться напрямую к искомому сервису, нет бродкастинга

Текстовый формат потребление доп трафика

Нет схемы данных, ес ть сваггер

Концепция «все есть ресурс». Неплохое решение когда нет больших нагрузок.

gRPC – бинарный формат, утилизация трафика лучше, можно сдать тольок хначения без ключей, сообщение не прочитать в консоли (парсим ответ). Есть схема данных для генерации DTO, сохраняем обратную совместимость. Можно стримить передавать объекты один за одним. Есть механизм backpressure – если сообшения отправляются слишком бытсро и полусатель не успел их переварить, механизм замдляет передачу. Отправка запроса выглядит как вызов методы используется RPC стиль и надо подклбчать доп библиотеку в коде своего языка. Подходит если в облаке куча сервисов, которым надо передавать много данных, но он не удобен для внешних пользователей. Жетская схема и меньше трафика потребляют.

SOAP – xml надо парсить. Формат сложнее и многословней.

Все очереди асинхронные и шлют сообщения, бывают с брокером и без него. Бывают с хранилищем и без него, кафка пишет все на диск, бывают с балансировкой нагрузки, с одним или разными получателями, со схемой данных и без, с подствержднием получения и без него.Хорошо там, где нужна быть отказоучтойчивость.

В асинхронном взаимодейтсвии сложнее разобраться,

Шардинг и репликация. Связаны с производительностью баз данных. Масштабирование баз данных. Сервер пересьает справляться с нагрузкой. Анализ медленных запросов надо произвести сначала.

Масштабирование баз данных как и с веб сервисами – разделение их на группы и размещение на разные сервера

Репликация – создается полный дубликат бд. Есть мастербд сэйв вспомогательный сервер, который все копирует. Позволяет разгрузить мастер, одно будет для чтения, другое для записи. Репликация поддерживается самой субд.

Но есть рассинхрониация данных, но всегда можно подлкючить с одного на лругой

Шардирование – разделение (партиционированние) ьазы данных на отдельные части, чтобы каждую их них можно было вынести на отдельный сервер, так распределяем нагрузку. Каждая малая таблица – шард. Вертикальный шардинг выделение таблицы или группы таблиц на отдельной сервер. Для каждой табл будет свое соединение. Горизонатльный – одной таблицы на разные сервера. Используется для очень больших таблиц. Принцип разделения: На нескольких серверах создается одна и та же таблица, в прилоэение условие, по которому будет определяться нужное соединение (четные на один сервер, нечетные на другой). Перед каждым обращение выбирают нужное соединение.

Отличие кафки и рэббитмк

Кафка-

Рэббит- программный брокер сообщений на основе стандарта AMQP.

Какие есть нотации C4 – состоит из 4 уровней
https://habr.com/ru/companies/nspk/articles/679426/

Модель данных - это концептуальное представление для выражения и передачи бизнес требований. Оно наглядно показывает характер данных, бизнес-правила, управляющие данными и то, как они будут организованны в БД.

ERD (entity relationship diagram) показывает сущности и отношения между ними. Она может принимать форму концептуальной, логической или физической модели данных

Концептуальная модель данных включает все основные сущности и связи, не содержит подробных сведений об атрибутах и часто используется на начальном этапе планирования

Логическая модель данных – это расширение концептуальной модели данных. Она включает в себя все сущности, атрибуты, ключи и взаимосвязи, которые представляют бизнес-информацию и определяют бизнес-правила

Физическая модель данных включает все необходимые таблицы, столбцы, связи, свойства БД для физической реализации БД. Производительность БД, стратегия, индексации, физическое хранилизе и денормализация – важные параметры физической модели.

Реляционная модель – метод проектирования, направленный на устранение избыточности данных. Жаннеы делятся на множество дискретных сущностей, каждая из которых, становится таблицей в реляционной БД. Таблицы обычно нормализованны до 3 нормальной формы.

Размерная модель – данные денормализованы для повышение производительности. Данные разделены на измерения и факты и упорядочены таким образом, чтобы пользователю было легче извлекать информацию и создавать отчеты.

OpenId – предназначен для аутентификации – для того, чтобы понять, что этот конкретный пользователь является тем, кем представляется

OAuth – протокол авторизации, т.е. позволяет выдать права на действия, которые Клиент может производить от лица Resource owner-а в другом сервисе

Оконная функция – функция, которая работает с выделенным набором строк (окном, партицией) и выполняет вычисление для этого набора строк в отдельном столбце. Ключевое слово over

Партиции (окна из набора строк) – это набор строк, указанных для оконной функции по одному из столбцов или группе столбцов таблицы. Партиции для каждой оконной функции в запросе могут быть разделены по различным колонкам таблицы.

При группировке кол-во строк уменьшается, а при картиции для каждой строки среднее

REST – все что мы хотим показать миру

Отличие URI от URL

Идемпотентность api, ключ идемпотентности в заголовке передается

Вопросы с бота:

1. Для передачи токена аутентификации при выполнении HTTP-запросов обычно используется заголовок Authorization.
2. В SOAP ответе информация об ошибке возвращается в элементе <Fault>. Этот элемент является частью стандартной структуры SOAP и используется для указания на наличие ошибок в процессе обработки запроса.
3. Процедура предоставления пользователю прав на выполнение определенного набора действий называется **авторизацией**
4. Процедура проверки того, что пользователь является тем, за кого себя выдаёт, называется **аутентификацией**. Обычно это осуществляется с помощью различных методов, таких как ввод пароля, использование биометрических данных, токенов или двухфакторной аутентификации.

виды аутентификации

- Базовая аутентификация (Basic access authentication)
- Дайджест-аутентификация (Digest access authentication)
- API ключи (API Keys)
- Аутентификация на предъявителя (Bearer authentication, also called token authentication)
  Базовая аутентификация (Basic access authentication)

  Наиболее простая схема, при которой username и password пользователя передаются в заголовке Authorization в незашифрованном виде, а в закодированном (base64-encoded).

  Обратите внимание, что даже несмотря на то, что ваши учетные данные закодированы, они не зашифрованы! Получить имя пользователя и пароль при базовой аутентификации

очень просто. Не используйте эту схему аутентификации на обычном HTTP, а только при использовании HTTPS (HTTP через SSL/TLS).

Дайджест-аутентификация (Digest access authentication)

Один из общепринятых методов, используемых веб-сервером для обработки учетных данных пользователя веб-браузера. Данный метод отправляет по сети хеш-сумму логина, пароля, адреса сервера и случайных данных, и предоставляет больший уровень защиты, чем базовая аутентификация, при которой данные отправляются в открытом виде.

При этой схеме сервер посылает уникальное значение nonce, а браузер передает MD5 хэш пароля пользователя, вычисленный с использованием указанного nonce.

API ключи (API Keys)

Некоторые API используют API ключи для авторизации. API ключ – это длинная уникальная строка содержащая произвольный набор символов, по сути заменяющие собой комбинацию username/password.

В большинстве случаев, сервер генерирует ключи доступа по запросу пользователей, которые далее сохраняют эти ключи в клиентских приложениях. При создании ключа также возможно ограничить срок действия и уровень доступа, который получит клиентское приложение при аутентификации с помощью этого ключа. Этот ключ может быть отправлен как параметр запроса

API ключи предполагаются быть секретными, т.е. только клиент и сервер знают этот ключ. Поэтому, как и в базовой, аутентификацию с помощью API ключей следует использовать только вместе с другими механизмами безопасности, такими как HTTPS/SSL.

Bearer authentication (also called token authentication)

Аутентификация на предъявителя (также называемая аутентификацией токена) - это схема проверки подлинности HTTP, которая включает токены безопасности, называемые токенами на предъявителя. Имя «Аутентификация на предъявителя» можно понимать как «предоставить доступ носителю этого токена». Токен-носитель - это загадочная строка, обычно генерируемая сервером в ответ на запрос входа в систему.

Такой способ аутентификации чаще всего применяется при построении распределенных систем Single Sign-On (SSO), где одно приложение (service provider или relying party) делегирует функцию аутентификации пользователей другому приложению (identity provider или authentication service). Типичный пример этого способа — вход в приложение через учетную запись в социальных сетях. Здесь социальные сети являются сервисами аутентификации, а приложение доверяет функцию аутентификации пользователей социальным сетям.

Реализация этого способа заключается в том, что identity provider (IP) предоставляет достоверные сведения о пользователе в виде токена, а service provider (SP) приложение использует этот токен для идентификации, аутентификации и авторизации пользователя.

Существует несколько стандартов определяющих протокол взаимодействия между клиентами и IP/SP-приложениями и формат поддерживаемых токенов. Среди наиболее популярных стандартов — OAuth, OpenID Connect, SAML, и WS-Federation.

Сам токен обычно представляет собой структуру данных, которая содержит информацию, кто сгенерировал токен, кто может быть получателем токена, срок действия, набор сведений о самом

пользователе (claims). Кроме того, токен дополнительно подписывается для предотвращения несанкционированных изменений и гарантий подлинности.

При аутентификации с помощью токена SP-приложение должно выполнить следующие проверки:

- токен был выдан доверенным identity provider приложением (проверка поля issuer).
- токен предназначается текущему SP-приложению (проверка поля audience).
- срок действия токена еще не истек (проверка поля expiration date).
- токен подлинный и не был изменен (проверка подписи).

Форматы токенов:

- Simple Web Token (SWT) - наиболее простой формат, представляющий собой набор произвольных пар имя/значение в формате кодирования HTML form.
- JSON Web Token (JWT) – токен, основанный на формате JSON. Подробнее можно почитать на специальном ресурсе посвящённом данному формату токета.
- Security Assertion Markup Language (SAML) - определяет токены (SAML assertions) в XML-формате, включающем информацию об эмитенте, о субъекте, необходимые условия для проверки токена, набор дополнительных утверждений (statements) о пользователе.

OAuth

Ключевая особенность применения OAuth заключается в том, что, если пользователь имеет подобный хорошо защищённый аккаунт, то с его помощью и технологии OAuth он может пройти аутентификацию на других сервисах, и при этом ему не требуется раскрывать свой основной пароль.

Стандарт OAuth (Open Authorization) не описывает протокол аутентификации пользователя, вместо этого он определяет механизм получения доступа одного приложения к другому от имени пользователя.

В чём отличия OAuth 2.0 от OAuth 1.0:

добавлены функционал для лучшей поддержки приложений не основанных на браузере.

OAuth 2.0 больше не требует, чтобы клиентские приложения имели криптографию (это было настоящим испытанием для реализации) теперь это переносится на плечи протокола HTTPS.

OAuth 2.0 предназначен для четкого разделения ролей между сервером, отвечающим за обработку запросов OAuth, и сервером, который обрабатывает авторизацию пользователя.

подписи OAuth 2.0 менее сложны.

токены OAuth 2.0 не «долговечны». Как правило, токены доступа OAuth 1.0 могут храниться в течение года или более (Twitter никогда не дает им истечь). OAuth 2.0 имеет понятие токенов обновления.

- токен доступа (access token): отправляется как ключ API, он позволяет приложению получать доступ к данным пользователя; опционально токены могут иметь срок действия.

- токен обновления (refresh token): опциональная часть, токены обновления получают новый токен доступа, если срок их действия истек.

OpenID Connect является средством аутентификации: с помощью этой системы можно удостовериться, что пользователь — именно тот, за кого себя выдаёт. Какие действия сможет совершать пользователь, прошедший аутентификацию посредством OpenID, определяется стороной, проводящей аутентификацию.

OpenID Connect предоставляет пользователю возможность создать единую учётную запись для аутентификации на множестве не связанных друг с другом интернет-ресурсов, используя услуги третьих лиц.

OpenID Connect позволяет клиентам всех типов (веб-клиенты, мобильные клиенты и клиенты JavaScript) запрашивать и получать информацию об аутентифицированных сеансах и конечных пользователях. Идентификационная информация пользователя закодирована в защищенном веб-токене JSON (JWT), который называется токеном идентификатора.

5. Процедура определения пользователя называется идентификацией. **Идентификация** (англ. identification) — это процесс, при котором пользователь сообщает системе информацию, позволяющую его однозначно определить, например, логин, идентификатор или имя пользователя. Идентификация обычно предшествует аутентификации, после которой система проверяет подлинность предоставленных данных.
6. Свойство системы, которое определяет, как просто пользователю изучить, запомнить и использовать её в своей работе, называется удобством использования или юзабилити (англ. usability).
7. В диаграммах последовательностей для отображения одного из множества вариантов, который выполнится при срабатывании заданного условия, используется фрейм alt (от англ. alternative).
8. **Требования можно описывать следующими способами:**

1. Текстовое описание (text) — это традиционный способ представления требований, когда они описываются в виде сплошного текста.
2. Пользовательские истории (user story) — это краткие описания функций с точки зрения конечного пользователя, которые включают цель и ожидаемый результат.
3. Сценарии использования (use case) — это описание взаимодействий между пользователем (актером) и системой для достижения конкретной цели.

9. Наиболее подходящий тип базы данных для хранения больших объемов данных с повышенной скоростью записи и чтения, но без необходимости сложных транзакций и структурированных запросов — это 3) Ключ-значение (Key-Value) базы данных.

Ключ-значение базы данных обеспечивают быструю работу с данными за счет простоты их структуры, где данные хранятся в виде пар "ключ-значение". Этот тип базы данных отлично подходит для сценариев, где требуется высокая производительность при больших объемах операций записи и чтения, а сложные запросы не являются необходимыми.

10. **Репликация** — это процесс копирования и синхронизации данных между несколькими серверами или базами данных с целью обеспечения доступности, отказоустойчивости и повышения производительности.

Какие нотации вы знаете?

4C - подход к описанию архитектуры системы, состоящий из 4 диаграмм.

Context diagrams (level 1) Диаграмма контекстов подходит для бизнес-пользвателей, которым не нужно включаться в технические детали. Диаграмма нужна чтобы ответить на вопросы какие пользвоатели используют систему, с какими системами она взаимодействует и для чего.

Container diagrams (level 2) Диаграмма контейнеров подходит для пользователей, которым нужно понять архитектуру приложения без глубокого погружения в техническую часть. Контейнер – это некая самостоятельная часть система, наприер, мобильное приложение. Диаграмма отвечает на вопросы Какие технологии использует система, из каких контейнеров состоит как они взаиможейству.т между собой, с внешними системами, как пользователи взаимодействуют с ними.

Component diagrams (level 3) Диаграмма компонентов Целевая аудитория диаграммы компонентов, это программисты и архитекторы. Компонент – это абстракция, из которых состоит контейнер. Отвечает на впорсоы из каких компонентов состоит контейнер, И как компоненты взаимодейсвтуют друг с лругом, пользователями, внешними системами.

Code diagrams (level 4) - Диаграмма кода используется для низкоуровневой детализации системы. Это uml диаграмма классов

UML, BPMN, ERD, DFD, IDEF

Какие виды диаграмм UML?

Диаграммы классов (Class Diagram), используемая для моделирования статической структуры системы, диаграмма прецендентов Показывает взаимодействия между пользователями (акторами) и системой через прецеденты (use cases), диаграмма последовательностей Моделирует динамическое взаимодействие объектов через последовательность сообщений между ними., диаграмма активности, диаграмма состояний,

Расскажите про межсистемную интеграцию?

Системная интеграция — способ заставить две или несколько программ, систем, технологией обмениваться информацией друг с другом.

Для выбора подходящего способа интеграции нужно ответить на вопросы:

1. какими данными должны обмениваться системы?
2. с какой скоростью и в каком объёме системы обмениваются данными?
3. как долго и как часто системы будут обмениваться данными?
4. какие ограничения и особенности есть у интегрируемых систем?
5. сколько временных и финансовых ресурсов есть на интеграцию?

Выделяют 4 основных стиля интеграции:
https://habr.com/ru/articles/841862/
https://systems.education/integrations-fundamentals-one

1. **Файловый обмен.** способ интеграции, при котором данные между системами передаются в виде файлов: система-источник размещает на файловом сервере файлы, а система-приёмник забирает их с файлового сервера и использует для своих нужд.
   файлообменники, ftp серверы, облачные серверы (гугл диск).
   Файловый обмен как способ интеграции подходит для решения задач, в которых нужно передавать файлы больших объёмов и обмен данными в реальном времени не нужен.

2. **Общая база данных.** шарить данные для других систем, например репликация или ETL (из нескольких источников в один источник загружаем). Когда все подключаются к одной базе данных – это плохо. Блокировки могут возникать, когда много систем обращаются параллельно к одной сущности в БД и внутренний механизм СУБД блокирует возможность работы с этой сущностью. Системы, работающие с общей БД, должны учитывать возможность блокировки и свою реакцию на блокировку.

Общая БД как способ интеграции применяется, когда все интегрируемые ИС внутренние; когда обмен данными происходит по расписанию или эпизодически, обмен частый и в реальном времени; когда объем данных не слишком большой (небольшие файлы); когда нужно видеть изменения, происходящие в ИС, сразу.

Синхронные интеграции – запрос ответ в режиме реального времени. Может заблокироваться во время ожидания, усложняет масштабирование, пожирает ресурсы. НАДО РАСКРЫТЬ ЕЩЕ

3. **Дистанционный вызов процедур** — это способ интеграции ИС, при котором клиент удалённо вызывает функцию или процедуру на сервере по определённому протоколу (набору правил).

Сервер в свою очередь выполняет какой-то процесс или формирует набор данных и передаёт ответ клиенту. (Вызов метода API или удаленный вызов процедур (RPC)).
API – интерфейс взаимодейтсвия между сервисами. Походы:
RPC — Remote Procedure Call;
CORBA — Common Object Request Broker Architecture;
SOAP — Simple Object Access Protocol
REST — Representational State Transfer
GraphQL — Graph Query Language
gRPC — Google Remote Procedure Calling

SOAP

стандартизированный протокол обмена сообщениями между приложения по сети. Протокол имеет набор жёстких правил, без соответствия которым взаимодействие по SOAP невозможно. Обмен данными с помощью SOAP реализуется в следующем порядке.

1. Клиент формирует запрос и направляет HTTP-запрос на сервер. Чаще всего это POST-запрос (даже на получение данных), так как исторически SOAP поддерживал только их (новые версии протокола поддерживают и GET). В теле запроса передаётся набор данных в формате XML: SOAP поддерживает только этот формат данных. Запрос всегда выполняется к одному ресурсу — название конкретной функции передаётся в теле запроса.
2. Данные запроса обрабатываются на веб-сервере по правилам, закреплённым в XSD-схеме.
3. Веб-сервер осуществляет вызов удаленной процедуры на сервере приложений.
4. Веб-сервер получает ответ от сервера приложений и возвращает его клиенту.

5. Важная часть проектирования интеграции через SOAP — WSDL-схема и XSD-схема. WSDL содержит описание всех функций, которые доступны для вызова на веб-сервисе. XSD-схема содержит структуру и содержимое запросов. В качестве транспорта для реализации SOAP чаще всего используется HTTP, но протокол может быть реализован поверх любого транспортного протокола прикладного уровня по модели OSI.
Как спроектировать интеграцию через SOAP

При проектировании SOAP API важно определить ряд параметров интеграции:

1. Перечень функций на сервере и их названия нужно закрепить в WSDL-схеме.
2. Структуру и содержимое XML-сообщений запросов, включая типы данных, нужно закрепить в XSD-схеме, которая будет использоваться для валидации запросов (XSD — это язык описания структуры XML документа. Его также называют XML Schema.).
3. Методы аутентификации клиента. Будет ли это базовая аутентификация или с использованием токена? Параметры для аутентификации передаются в заголовке запроса.
4. Перечень статусов ответов сервера, в том числе для ошибок. Статусы ответов передаются в заголовке ответа.
5. Структуру и содержимое XML-сообщений ответов, включая типы данных, нужно закрепить в XSD-схеме, которая будет использоваться для валидации ответов.

Плюсы soap: стандартизация, все соап сервисы должны быть спроектированы одинаково, поддерживает много стандратов безопасности, например WS-Security, которые обеспечивают шифрование сообщений и аутентификацию. Он надежен стандарты WS-ReliableMessaging, которые гарантируют доставку сообщений даже при сбое сети.
Минусы: сложная разработка, XML тяжеловесен,soap избыточен из-за большого количества методанных. Сложность внесения изменений из-за строгого контракта: любые изменения требуют обновления WSDL и XSD, изменений в коде и клиента и сервера.
Когда используем? SOAP хорошо подходит для интеграции ИС, когда:

1. надёжность и безопасность важнее скорости;
2. строгие контракты важнее высокой частоты изменений;

3. логика удаленных процедур сложна;
4. нужны транзакционные операции.

SOAP как способ интеграции чаще всего используют для решения задач, в которых цена ошибки высока: финтех-проекты; государственные системы; медицинские системы.

XML формат, можно передавать бинарные типы данных бинарные – файлы, различные сетевые протоколы хттп тоже, повышенная безопасность например шифрование сообщений. Он уже стандартизирован. Есть транзакции, транспортировка сообщений.

1. Envelope (оболочка): корневой элемент SOAP-сообщения, который определяет его как SOAP-сообщение. Он заключает в себя все остальные части сообщения. содержит пространства имён, которые указывают, что это SOAP-сообщение и подчиняется определённым стандартам.

2. Header (заголовок) (опционально): Содержит метаданные, такие как информация о безопасности, маршрутизации, транзакциях и другая дополнительная информация. Заголовок является опциональным, но часто используется для передачи важной служебной информации.

3. Body (тело): Основная часть SOAP-сообщения, содержащая фактические данные запроса или ответа. В теле описываются конкретные действия (методы или операции), которые вызываются у веб-службы, и параметры этих действий. Если сообщение содержит ошибки, они описываются в виде элемента Fault внутри тела.

4. Fault (ошибка) (опционально): Специальный элемент, который используется для передачи информации об ошибках или исключениях, произошедших на стороне сервера. Этот элемент может содержать коды ошибок, текстовые описания и подробную информацию о том, что пошло не так.

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

XSD (XML Schema Definition) — это язык определения структуры и правил для XML-документов. Определяет элементы, типы данных, атрибуты и их возможные значения

XML (eXtensible Markup Language) — это расширяемый язык разметки, который используется для представления структурированных данных в виде текста.

```
<Книга название="Война и мир" автор="Лев Толстой">
 <ГодИздания>1869</ГодИздания>
</Книга>
```

- **Книга** — это элемент, который содержит два атрибута: `название` и `автор`.
- **ГодИздания** — это вложенный элемент, который содержит текстовое значение `1869`

Что содержится в XML:

1. **Элементы (Elements)** — основная структура XML-документа. Элементы определяются открывающим и закрывающим тегами. Элементы могут содержать другие элементы или текст.
    - Пример: `<Имя>Иван</Имя>`

2. **Атрибуты (Attributes)** — дополнительные данные, которые задаются внутри открывающего тега элемента. Они служат для описания свойств элементов.
    - Пример: `<Книга название="Война и мир" автор="Лев Толстой"/>`
3. **Теги (Tags)** — используются для обозначения начала и конца элемента. Открывающий тег `<Имя>` и закрывающий тег `</Имя>`.
4. **Корневой элемент (Root Element)** — каждый XML-документ должен иметь один корневой элемент, внутри которого располагаются все остальные элементы.
    - Пример: `<Библиотека>...</Библиотека>`
5. **Пролог** — начало XML-документа, которое указывает версию XML и кодировку.
    - Пример: `<?xml version="1.0" encoding="UTF-8"?>`
6. **Комментарии** — текст, который игнорируется при обработке XML, но может быть полезен для добавления пояснений.
    - Пример: `<!-- Это комментарий -->`

Пример последовательности действий

1. **Разработка WSDL**:
    - Описание метода `GetBook`, который принимает `GetBookRequest` и возвращает `GetBookResponse`.
2. **Создание XSD**:
    - Определение структуры `GetBookRequest` (например, с элементом `BookId`) и `GetBookResponse` (например, с элементами `Title`, `Author`, `Year`).
3. **Генерация XML**:
    - Клиент отправляет XML-документ, например:

WSDL состоит из нескольких разделов:

- **types** — описывает данные, которые передаются, с помощью XSD.
- **message** — определяет структуру сообщений (запросов и ответов).
- **portType** — указывает доступные операции (методы).
- **binding** — описывает, как именно сообщения передаются, например через HTTP.
- **service** — указывает, где веб-служба доступна (URL)

- **SOAP** — это протокол обмена сообщениями.
- **WSDL** описывает интерфейс веб-службы (какие методы доступны, и как с ними взаимодействовать).

- **XSD** описывает структуру данных, используемых в SOAP-сообщениях.
- Клиент использует **WSDL**, чтобы понять, как взаимодействовать с сервером, и отправляет **SOAP-запросы** на основе WSDL и XSD.

SOAP является надстройкой над протоколом HTTP, что означает, что он использует HTTP как транспортный протокол для передачи SOAP-сообщений. При этом SOAP определяет, как структурировать сообщения, а HTTP отвечает за их передачу.

https://babok-school.ru/blog/soap-server-python-example/

Документ WSDL используется для краткого описания веб-службы и предоставляет клиенту всю информацию, необходимую для подключения к веб-службе и использования всех функций, предоставляемых веб-службой. 2
Структура WSDL-документа включает следующие элементы: 1

1. **Определение типов данных (types)**. Определение вида отправляемых и получаемых сервисом XML-сообщений. 1
2. **Элементы данных (message)**. Сообщения, используемые веб-сервисом. 1
3. **Абстрактные операции (portType)**. Список операций, которые могут быть выполнены с сообщениями. 1
4. **Связывание сервисов (binding)**. Способ, которым сообщение будет доставлено. 1
5. **Конкретная точка вызова сервиса (service)**. Эндпоинт/адрес, по которому сервис доступен.

CAP теорема - рассказывает какую систему можно создать, какую нет

консистентность – все данные одинаковые вне зависимости с сайта или мобилки смотришь,

доступность – 24 на 7 можешь зайти в свою систему

доступность по сети (разделение по сети) если интернет провалился все равно можешь зайти в свою систему

REST

REST — это архитектурный стиль, набор принципов проектирования архитектуры распределенных веб-приложений. REST API — применение архитектурного стиля REST к проектированию API. REST не имеет собственных методов и не ограничен никакими протоколами.

Существует 6 принципов стиля REST:

Клиент-серверная архитектура

Stateless

Кэширование

Единообразие интерфейса

Layered system

Code on demand

Если сервис соответствует всем этим принципам, то такой сервис называют RESTful.

Приложение, спроектированное в стиле REST работает следующим образом.

1. Клиент (как правило из веб-браузера) направляет HTTP-запрос на веб-сервер.
2. Веб-сервер направляет запрос к веб-приложению.
3. Веб-приложение выполняет какое-то действие или формирует набор данных из БД и возвращает ответ клиенту через веб-сервер.

В качестве транспортного протокола REST-сервисы используют HTTP/HTTPS. Обмен данными производится чаще всего в формате JSON, но можно использовать любой формат: XML, TXT, CSV, Protobuf.

Для того, чтобы обратиться к веб-серверу, необходимо направить запрос по определённому маршруту, используя один из HTTP-методов (GET, POST, PUT, PATCH). Таким образом, маршрут указывает на то, с каким ресурсом (объектом) нужно работать — это может быть любой объект предметной области: пользователь, заказ, товар и т.д. Важно, что сам по себе HTTP-метод не выполняет никаких действий с ресурсом. REST не обязывает реализовывать конечные точки каким-то конкретным образом, например, удаление ресурса можно реализовать методом GET и наоборот. Несмотря на это использование методов с учетом их семантики является рекомендацией стиля REST и правильной практикой для правильного понимания пользователей вашего API.

При проектировании REST API важно определить ряд параметров интеграции.

1. Набор конечных точек: маршрутов и HTTP-запросов к ним. Часто конечные точки еще называют эндпоинтами (endpoint) или «ручками».
2. Формат, структуру и содержимое полезной нагрузки запроса для каждой конечной точки.
3. Аутентификация клиента для каждой конечной точки.
4. Статусы ответа сервера для каждой конечной точки
5. Формат, структура и содержимое полезной нагрузки ответа для каждой конечной точки.

Для документирования REST API принято использовать спецификацию OpenAPI.

Плюсы: простота реализации, универсальность, поддрежка разных методов аутентификации и кэширования.

Минусы: отсутсвие стандартизации, Низкая производительность: каждый запрос требует отдельного HTTP-соединения и содержит метаданные, которые увеличивают нагрузку на сеть, Изначальная ориентация на stateless (без сохранения состояния), а не stateful, неподходит для большого объема данных, Сложно реализовать транзакционные операции над несколькими ресурсами одновременно.

Когда используем? REST хорошо подходит для интеграции ИС, когда:

- нужно сделать быстро и просто;
- скорость важнее надёжности и безопасности;
- нужно часто вносить изменения в контракты взаимодействия;
- бизнес-логика не очень сложная и операции над бизнес-сущностями ограничены набором CRUD-операций (Create, Read, Update, Delete);
- экономия трафика не важна.

REST как архитектурный стиль интеграции чаще всего используют для: веб-приложений; мобильных приложений.

GraphQL

— технология обработки запросов к приложению с помощью API. GraphQL как технология объединяет в себе язык запросов, среду обработки запросов и архитектуру клиент-серверного взаимодействия.

GraphQL применяется когда нужно собирать данные из разных источников и сократить число запросов от клиента к серверу, передав в ответ только те данные, что запрашивает клиент.

gRPC

— фреймворк реализации удаленного вызова процедур (RPC). Фреймворк gRPC позволяет реализовать различные механики взаимодействия систем или сервисов.

1. Механика «Запрос-Ответ» (Request-Response): синхронный запрос клиента и ответ сервера.
2. Потоковая передача потока данных с сервера на клиент при подключении клиента.
3. Потоковая передача потока данных с клиента на сервер при подключении клиента.
4. Двунаправленная передача потока данных с сервера на клиент и наоборот.

В качестве транспортного протокола gRPS использует HTTP/2. Использование именно второй версии протокола обеспечивает производительность выполнения запросов и передачи данных за счёт реализации в HTTP/2 мультиплексирования и эффективных механизмов сжатия.

Также высокая производительность gRPC достигается за счёт особого формата передачи данных — бинарного формата Protobuf (Protocol Buffers) со встроенной схемой данных. В protobuf данные передаются в закодированном виде, а декодирование производится на стороне получателя данных. Когда использовать gRPC

gRPC хорошо подходит для интеграции ИС, когда:

1. нужны разные варианты обмена данными (по запросу сервера, по запросу клиента, потоковая передача данных и т.д.);
2. требуется реализовать высокое быстродействие в реальном времени;
3. разные технологии у интегрируемых ИС и стек может расширяться;
4. важна безопасность.

gRPC чаще всего применяется для: микросервисов; мобильных приложений; IoT (Internet Of Things)-устройств; машинного обучения; веб-приложений; стриминговых сервисов.

gRPC как технология интеграции подходит в случаях, когда нужны разные варианты обмена данными (по запросу сервера, по запросу клиента, потоковая передача данных и т.д.); требуется реализовать высокое быстродействие в реальном времени; разные технологии у интегрируемых ИС и стек может расширяться; важна безопасность.

Webhook (вебхук)

— технология взаимодействия веб-приложений или сервисов в реальном времени. Вебхук позволяет настроить автоматическую отправку запроса из системы-источника к системе-приемнику при наступлении какого-то события-триггера. По своей сути вебхук — это HTTP-запрос, только сформированный и отправленный автоматически при наступлении события. Использование вебхука позволяет избежать непрерывного опроса API сервера клиентом при реализации асинхронных процессов (это еще называют longpooling).

Вебхук хорошо подходит для решения задач:

- когда характер обмена событийный (не периодический);
- когда клиент не знает, когда на сервере будут новые данные, но должен получить их, когда они появятся;
- когда нужно избежать периодических и безрезультатных обращений клиента к серверу, например, для экономии трафика;

Вебхук часто используется в системах: с событийно-ориентированной архитектурой (Event-Driven Architecture, EDA); с уведомлениями в реальном времени.

Вебхук можно использовать событийном характере обмена; если клиент не знает, когда на сервере будут новые данные, но должен получить их, когда они появятся; когда нужно избежать периодических и безрезультатных обращений клиента к серверу, например, для экономии трафика.

WebSocket

— протокол связи между клиентом и сервером. Протокол позволяет устанавливать двунаправленное постоянное соединение в реальном времени благодаря механизму keep alive, введенному в рамках HTTP 1.1.

Установка websocket-соединения осуществляется с помощью handshake. Handshake — это приветственное сообщение (рукопожатие), которое отправляется клиентом на сервер. Оно сигнализирует, что необходимо перейти с использования протокола HTTP на протокол WebSocket.

Закрытие соединения может быть инициировано как клиентом, так и сервером, и осуществляется с помощью отдельного закрывающего сообщения.

Пока соединение открыто, клиент и сервер могут обмениваться данными в реальном времени, причём отправлять данные может как клиент, так и сервер.

На транспортном уровне WebSocket использует сетевой протокол TCP, что позволяет организовать передачу данных разных схем и размеров. Протокол TCP разбивает данные на фреймы, обеспечивая их целостность через отправку закрывающего бита, который маркирует завершение передачи.

WebSocket хорошо подходит для решения задач, в которых:

- нужна интеграции в реальном времени;
- нужна двусторонняя связь клиента с сервером;
- нужно долговечное решение.

WebSocket часто используется в: онлайн-играх; чатах и мессенджерах; системах мониторинга в реальном времени; IoT-сфере.

Асинхронные интеграции – запрос будет обработан не сразу, нужно уметь получать ответ. Не блокирует поток и ресурсы, увеличивает производительность, облегчает масштабирование. НО реализация сложнее, отсутствие транзакций.

4. **Обмен сообщениями:** например, интеграция с помощью интеграционной шины или внешнего сервиса через брокеры сообщений.

Обмен сообщениями: посредник между системами, возможен синх и асинх обмен, но при синхр теряется смысл использования брокера. Два основных подхода обмена на основе брокеров: точка-точка (требует респонс ответ) ИЛИ издатель –подписчик (не ждет ответа, он обрабатывает тогда когда надо).

Примеры брокеров ну очевидно Кафка, рэббит и ActiveMQ. Самый популярный и надежный кафка, потому что обещает сохранить и не потерять сообщений. Рэбит тоже при настройках так умеет, но не по умолчанию, как в кафке.

Сходства Apache Kafka и RabbitMQ

Оба являются брокерами программных сообщений и используются для обмена информацией между различными приложениями. оба брокера работают по схеме «издатель-подписчик» (отправитель-получатель), когда источники данных направляют потоки информации, а получатели обрабатывают их по мере потребности.оба брокера способны реализовать стратегии «как максимум однократная доставка» и «как минимум однократная доставка», что позволяет сократить риски потери или дублирования сообщений. обе системы обеспечивают репликацию сообщений. Apache Kafka и RabbitMQ гарантируют порядок отправки сообщений с помощью уведомлений и стратегий доставки

Отличия:

- обусловлены принципиально **разными моделями доставки** сообщений, реализуемыми в этих системах. В частности, Kafka действует по принципу вытягивания (**pull**), когда получатели (consumers) сами достают из топика (topic) нужные им сообщения. RabbitMQ, напротив, реализует модель проталкивания (**push**), отправляя необходимые сообщения получателям.
- RabbitMQ помещает сообщение в очередь FIFO (First Input – First Output) и отслеживает статус этого сообщения в очереди, а Kafka добавляет сообщение в журнал (записывает на диск), предоставляя получателю самому заботиться о получении нужной информации из топика. Кролик удаляет сообщение после доставки его получателю, а Кафка хранит сообщение до тех пор, пока не наступит момент запланированной очистки журнала.
- **Балансировка** – благодаря pull-модели доставки сообщений RabbitMQ сокращает время задержки, однако возможно переполнение получателей, если сообщения прибудут в очередь быстрее, чем те могут их обработать. Поскольку в RabbitMQ каждый получатель запрашивает/выгружает разное количество сообщений, то распределение работы может стать неравномерным, что повлечет задержки и потерю порядка сообщений во время обработки. Для предупреждения этого каждый получатель Кролика настраивает предел предварительной выборки (QoS) – ограничение на количество скопившихся неподтвержденных сообщений. В Apache Kafka балансировка нагрузки выполняется автоматически путем перераспределения получателей по разделам (partition) топика.
- **Маршрутизация** – RabbitMQ включает 4 способа маршрутизации на разные обменники (exchange) для постановки в различные очереди, что позволяет использовать мощный и гибкий набор шаблонов обменов сообщениями. Кафка реализует лишь 1 способ записи сообщений на диск, без маршрутизации.
- Kafka говорят «тупой сервер, умный клиент», что означает необходимость реализации логики работы с сообщениями на клиентской стороне, т.е consumer заботится о получении нужных сообщений. RabbitMQ – наоборот, «умный сервер, тупой клиент», поскольку этот брокер сам обеспечивает всю логику работы с сообщениями.

Отправитель или producer — подсистема или сервис, который публикует данные в брокер.

Потребитель или consumer — подсистема или сервис, который подписывается на получение сообщений из брокера и потребляет полученные данные.

Брокер является посредником между отправителем и потребителем. Можно сказать, что брокер выступает сервером, а отправители и потребители — клиентами.

При проектировании интеграции через брокер сообщений важно определить ряд параметров интеграции.

1. Как будет называться канал связи, в который будут публиковаться данные. В RabbitMQ это топик, в Apache Kafka — очередь.
2. Параметры полезной нагрузки сообщений от отправителя: формат, схема, максимальный размер. Максимальный размер важен, потому что для всех брокеров не рекомендуется передавать сообщение больше 1 МБ.

3. Какова надёжность потребителя. Если потребитель ненадёжный, то стоит использовать Kafka, который сохраняет данные на диске длительное время, в отличие от RabbitMQ, который удаляет данные после доставки сообщения.
4. Фактор репликации — количество копий сообщения — в кластере брокера. Это нужно для обеспечения надёжности доставки сообщений.
5. Производительность потока данных в части публикации и потребления.
6. Требуется ли настраивать разделы (партиции) в Kafka (разделение топика на несколько разделов) или предел очереди в RabbitMQ (максимальное количество сообщений в очередь), чтобы предотвратить переполнение очереди.
7. Стратегия разделения в Kafka, если разделение используется; тип обменника (маршрутизатор сообщений) в RabbitMQ.
8. Размер очереди и время жизни сообщения для RabbitMQ; максимальное время хранения сообщений для Kafka.
9. Схема потокового обмена, на которой будут отображены все топики/очереди, отправители и потребители данных. Удобно проектировать такую схему через DFD-диаграмму.
10. Также описать интеграцию через брокер можно с помощью спецификации стандарта AsyncAPI. Спецификация в формате yaml (похожая на Open API спецификацию) содержит состав топиков, схему аутентификации, формат данных полезной нагрузки и т.д.

Когда использовать брокеры сообщений

Брокеры сообщений как технология интеграции отлично подходят для решения задач, когда:

- нужна асинхронная интеграция, в том числе сразу нескольких систем;
- нужна передача данных в реальном времени или низкая задержка;
- много данных, но размер каждого сообщения небольшой;
- событийный характер обмена (EDA-архитектура);
- нужны высокая производительность и масштабирование;
- есть ресурсы на развёртывание и поддержку дополнительной инфраструктуры для брокера.

Брокеры сообщений успешно используются в системах:с микросервисной событийно-ориентированной архитектурой; платформы IoT; с уведомлениями в реальном времени; с межсервисными транзакциями.

Служебная шина данных (ESB)

отдельный промежуточный сервис по середине всех нашихон облегчает взаимодействие между различными приложениями и службами в сервис-ориентированной архитектуре. Получает, маршрутизирует, преобразовывает и доставляет данные с использованием различных протоколов связи.

https://systems.education/integrations-fundamentals-two

https://habr.com/ru/articles/676088/

Архитектура программного обеспечения

Монолитное приложение —

всё приложение — это единая структура, где все компоненты (бизнес-логика, пользовательский интерфейс, доступ к данным) тесно связаны и развёртываются как одно целое.

Монолитная архитектура — это традиционный дизайн программного обеспечения, при котором все приложение строится как единое целое. В этом типе архитектуры все компоненты программной системы, включая пользовательский интерфейс (UI), бизнес-логику и уровни обработки данных, тесно интегрированы в единую кодовую базу.

тип двухуровневой (также называют двухзвенной) архитектуры. При двухуровневой архитектуре приложение разбито на два слоя.
На первом слое находится визуальный интерфейс (UI — User Interface), предназначенный для удобства использования приложения и некоторой валидации данных.

Второй слой — слой хранилища данных. На этом слое кроме хранилища существует некая бизнес-логика, например: бухгалтерия, составление отчётности и т. д. Логика может быть, например, написана на каком-либо транзакционном языке, в зависимости от конкретной СУБД (система управления базами данных).

При необходимости такое приложение можно масштабировать путём увеличения мощности сервера, на котором инсталлировано приложение — «вертикальное масштабирование».

СОА (сервис-ориентированная архитектура) — многослойная (три или более уровня) архитектура, каждый слой которой отвечает за определённую «обязанность»:

1 слой — пользовательский интерфейс, зачастую веб-интерфейс

2 слой — бизнес-логика: расчёты, агрегация данных и т. д.

3 слой — база данных (хранилище)

В его основе лежат несколько основных идей – переиспользование сервисов и корпоративная шина. Разработчики стремятся разбить систему на сервисы таким образом, чтобы их можно было использовать повторно. Взаимодействие и маршрутизация осуществляется через корпоративную шину ESB. Типичная SOA архитектура показана на рисунке ниже.

Давайте посмотрим из каких частей она состоит, и какова их роль.

Шина(ESB): в случае взаимодействия сложных событий действует как посредник и управляет различными рутинными операциями, такими как передача сообщений и координация вызовов.

Инфраструктурные сервисы (infrastructure services): группа легко переиспользуемых сервисов, таких как аутентификация/авторизация, отправка смс и прочее.

Прикладные сервисы (application services): не могут быть переиспользованны под разные задачи, так как ограничены определенным прикладным контекстом, но их можно встраивать в более высокоуровневые сервисы.

Сервисы предприятия (Enterprise services): эти сервисы отвечают за реализацию крупных частей бизнес процессов компании, они потребляют более низкоуровневые сервисы.

API: по сути это бэкенды, предоставляющие API, доступное в интернет, для сайтов и мобильных приложений компании. Они взаимодействуют с ESB и раскрывают функциональность для конечных потребителей.

Как вы могли заметить, проектируя архитектуру в данном стиле, мы имеем очень большое количество слоев и как следствие команд, которые ими владеют. Любой запрос пронизывает все слои системы, в большей степени напоминая монолитную архитектуру. Но данный тип архитектуры намного сложнее, потому что является распределенной архитектурой.

Микросервисная (облачная) архитектура

Для оптимизации затрат представляется возможным переход на облачную архитектуру. Необходимо построить приложение таким образом, чтобы можно было развернуть его не только на своих мощностях, но и на каких-то облачных ресурсах.

Микросервисы — это небольшие автономные совместно работающие сервисы.
Можно выделить следующие основные характеристики MCA:

- Максимальная независимость и автономность
- Реализация подхода «умные сервисы и глупые каналы» (smart endpoints and dumb pipes) при микросервисном взаимодействии
- Поддержка DevOps подходов CI (Continuous integration, Непрерывная интеграция) и CD (Continuous delivery, Непрерывная доставка)
- Гибкая масштабируемость
- Нацеленность на концепцию децентрализации управления данными

Микросервисы в отличие от SOA, наоборот, избегают повторного использования, применяя философию - предпочтительнее дублирование, а не зависимость от других сервисов. Повторное использование предполагает связанность, а архитектура микросервисов в значительной степени старается ее избегать. Это достигается за счет разбиения системы на сервисы по ограниченным контекстам (бизнес областям). Типичная MSA архитектура показана на рисунке ниже.

В отличие от SOA каждый сервис обладает всеми необходимыми для функционирования частями – имеет свою собственную базу данных и существует как независимый процесс. Такая архитектура делает каждый сервис физически разделенным, самодостаточным, что ведет с технической точки зрения к архитектуре без разделения ресурсов.

Сервисы раскрываются для потребителей также через слой API, но его стараются проектировать с полным отсутствием какой-либо логики. Это фактически просто проксирование API сервисов во вне.

Взаимодействие между сервисами сводится к обмену данными, используя брокер сообщений. Именно к обмену данными, а не вызову методов из других сервисов.

Как вы могли заметить, проектируя архитектуру в данном стиле мы имеем небольшое количество слоев, но достаточно много сервисов и команд, которые ими владеют. Основной сложностью данной архитектуры является правильное определение ограниченного контекста и распределение сервисов по командам.

«В чем отличие микросервисов от SOA?». Ответ часто сводится к техническим особенностям реализации SOA, таким как наличие ESB, централизация, крупные сервисы и т.п. Все это действительно так, но сегодня я хочу дать более развернутый ответ и посмотреть на корневые различия, плюсы, минусы этих двух архитектур.

Нормализация – процесс устранения избыточность и предотвращения аномалии при изменении данных. Главная цель нормализации — оптимизация структуры базы данных и обеспечение целостности данных.

1 нормальная форма – каждое поле должно содержать только одно значение, нет дублирования строк, столбцы содержат однородные данные

2 нормальная форма - Все неключевые атрибуты (столбцы) должны зависеть от всего первичного ключа (если ключ составной)

3 нормальная форма - Неключевые атрибуты (столбцы) не должны зависеть друг от друга (устранение транзитивных зависимостей).

Если в таблице нет составного ключа, 2NF будет уже выполнена, и тогда нужно просто проверять на транзитивные зависимости для 3NF.

Аномалии в бд – ошибки возникающие при работе с бд, ошибки вставки, изменения, удаления.

Аномалии-модификации проявляются в том, что изменение одних данных может повлечь просмотр всей таблицы и соответствующее изменение некоторых записей таблицы.

Аномалии-удаления — при удалении какого либо кортежа из таблицы может пропасть информация, которая не связана на прямую с удаляемой записью.

Аномалии-добавления возникают, когда информацию в таблицу нельзя поместить, пока она не полная, либо вставка записи требует дополнительного просмотра таблицы.

ТРЕБОВАНИЯ

Кто собирает требования,? Дискавери фаза – выявляются требования и анализируются поставленные цели – участвует продак и дизайнер, продак приходит к аналитику с задачей, вместе они прорабатывают задачу и отдают в разработку и начинается процесс деливери (доводим задачу до прода).

У кого будем собирать требования? У любого человека который заинтересован в проекте(стейкхолдеры) или системе. Пользователь, бизнес, команды разработки, маркетинг, эксперт предметной области,

КАк согласовывать требования между различными участниками? RACI

R(responsible)

A(accountable)

C

I

Какие есть способы требований? Опрос, анкетирование, интервью, анализ документации, анализ конкурентов,

Анкетировнаие – составляем лист опросник содержащий открытые и закрытые вопросы

Мозговой штурм – участники накидывают идеи

Работа в полях – командировка на производтво

Интервью – разговор, беседа, при котором задают интервью и занимают вопросы, задавайте открытые вопросы

Прототип – наброски интйерфейсов или прототипы, чтобы получить обратную связь.

Какие вопросы задать при внедрение фичи? Вопросы пользователям:

Как опрашивать пользователя о нефункциональных требований? Пользователи нес ильно дуимают о них, больше вопросы продукту, заказчику, система надежная производительная, защищает данные, скольок челолвек будет заходить,

Виды требований:

1) Функциональные требования: Определяют, что система должна делать.
2) Нефункциональные требования: Определяют характеристики системы, такие как производительность, масштабируемость, безопасность и надежность.
3) Бизнес-требования: Определяют цели и задачи бизнеса, которые система должна поддерживать.
4) Требования к пользовательскому интерфейсу: Определяют внешний вид и функциональность пользовательского интерфейса.
5) Требования к интеграции: Определяют, как система должна взаимодействовать с другими системами.
6) Требования к производительности: Определяют ожидаемую производительность системы.
7) Требования к безопасности: Определяют меры, необходимые для обеспечения безопасности данных.
8) Требования к обучению: Определяют, какое обучение должно быть проведено для пользователей системы.

Нефункциональные требования включают:

Нефункциональные требования определяют характеристики системы, которые не связаны с ее функциональностью, но могут влиять на ее качество и удовлетворенность пользователей. Некоторые из них включают:

1) Производительность: Требования к скорости работы системы, количеству обрабатываемых запросов в секунду и т.д.

2) Масштабируемость: Требования к возможности системы расширяться и адаптироваться к увеличению нагрузки или добавлению новых функций.

3) Надежность: Требования к стабильности работы системы и отсутствию сбоев.

4) Безопасность: Требования к защите системы от атак, утечек данных и других угроз.

5) Удобство использования: Требования к простоте и интуитивности пользовательского интерфейса, легкости обучения и т.д.

например: “Система должна обеспечивать скорость обработки запросов не менее 1000 запросов в секунду”.

Критерии требований:

1) Релевантность - Требование должно быть связано с целью или задачей, которую необходимо достичь.

2) Четкость - Требование должно быть сформулировано ясно и однозначно, чтобы все участники проекта понимали его одинаково.

3) Измеримость - Требование должно иметь возможность быть измеренным или оцененным.

4) Выполнимость - Требование должно быть реалистичным и осуществимым в рамках проекта.

5) Приоритетность - Требование должно иметь определенный приоритет в зависимости от его важности для проекта.

6) Атомарность - Требование не должно быть разделено на более мелкие требования без необходимости.

7) Тестируемость - Требование должно допускать возможность тестирования для подтверждения его выполнения.

8) Недвусмысленность - Требование не должно допускать различных толкований или неопределенности.

9) Проверяемость - Требование должно предусматривать возможность проверки его выполнения.

Use case (случай использования) - это описание сценария взаимодействия пользователя с системой, которое включает в себя цель использования системы, последовательность действий пользователя и ожидаемый результат.

Написание use case включает следующие шаги:

1) Определение контекста использования. Это включает в себя определение цели использования системы, участников (актеров) и их ролей, а также условий использования (например, время суток, место использования и т.д.).

2) Описание основных сценариев использования. Для каждого актера определяются основные сценарии использования системы, которые включают последовательность действий актера и ожидаемый результат для каждого сценария.

3) Детализация сценариев использования. Каждый сценарий разбивается на более мелкие шаги, описываются возможные варианты развития событий и определяются требования к системе для успешного выполнения каждого шага.

4) Документирование use case. После того как все сценарии описаны, они оформляются в виде документа, который включает в себя всю необходимую информацию о каждом сценарии использования.

User story (история пользователя) - это краткое описание конкретной задачи или функции, которую пользователь хочет выполнить с помощью системы. Она включает в себя имя пользователя, его цель и шаги, которые пользователь предпринимает для достижения этой цели.

Написание user story включает следующие шаги:

1) Определите пользователя. Кто будет использовать систему?

2) Определите его цель. Что пользователь хочет достичь?

3) Опишите шаги, которые пользователь предпримет для достижения цели. Как он будет взаимодействовать с системой?

4) Оцените сложность реализации. Насколько сложно реализовать эту функцию или задачу?

5) Определите зависимости от других функций или задач. Нужно ли реализовать еще что-то, чтобы эта функция работала?

6) Документируйте user story. Запишите всю полученную информацию и оформите ее в удобном для чтения виде.

User stories более краткие и простые для понимания, в то время как Use cases могут быть более подробными и сложными.

User stories лучше подходят для выявления и описания требований на ранней стадии разработки, в то время как Use cases больше подходят для детальной проработки требований.

Что содержится в вашей типой задаче для разработчиков?

Название проекта и его цель;

Описание проблемы или задачи, которую нужно решить;

Требования к решению (функциональные, нефункциональные, ограничения и т. д.);

Критерии оценки результата;

Сроки выполнения задачи;

Необходимые ресурсы и среда разработки;

Вопросы для обсуждения с командой и с владельцем продукта

CAP теорема

Одним из краеугольных камней построения распределенных систем является **CAP-теорема**, утверждающая то, что в любой распределенной системе с данными возможно обеспечить не более двух из трёх следующих свойств:

- согласованность данных (англ. **c**onsistency) — во всех распределенных системах в один момент времени данные не противоречат друг другу — они актуальны и одинаковы в любой момент времени на каждом узле системы;
- доступность (англ. **a**vailability) — любой запрос к распределённой системе завершается откликом, но без гарантии, что ответы всех узлов системы одинаковы;
- устойчивость к разделению (англ. **p**artition tolerance) — потеря связи между узлами распределённой системы не приводит к некорректности отклика от каждого из узлов.

Несмотря на то, что на практике возможно достичь компромисса в решениях, основными вариантами сочетания свойств являются:

- **AP** — все запросы к системе получат ответ, даже если между узлами потеряно соединение, но вероятны случаи, когда пользователю будут возвращены устаревшие данные.
- **CP** — если изменения удалось распространить по всем узлам, система выполнит транзакцию. При отказе узлов запросы могут быть не обработаны или обработаны с задержкой до восстановления целостности системы.
- **CA** — во всех узлах распределенной системы данные согласованы и обеспечена доступность, при этом система жертвует устойчивостью к разделению. Такие системы возможны на основе решений, поддерживающих **ACID**-транзакции.

ACID (Atomicity, Consistency, Isolation, Durability) — набор характеристик, обеспечивающих надежность транзакций в базах данных.

- **Атомарность** (Atomicity): Транзакция является единым объектом. Она либо выполняется полностью, либо не выполняет вообще.
- **Согласованность** (Consistency): Транзакция должна переводить базу данных из одного согласованного состояния в другое (например, в каждом столбце БД значения имеют нужный тип данных, не нарушены ограничения, операции выполнены по порядку).
- **Изолированность** (Isolation): Каждая транзакция должна быть изолирована от других и ее выполнение не должно влиять на другие транзакции.
- **Долговечность** (Durability): После успешного завершения транзакции изменения в БД должны сохраняться даже в случае сбоев системы. Если пользователь получил подтверждение от системы, что транзакция выполнена, он может быть уверен, что сделанные им изменения не будут отменены.

Что такое ACID хорошо разобрано на примере в этой статье.

Расширением теоремы CAP является **теорема PACELC**.

Она гласит, что в случае сетевого разделения (P) в распределенной компьютерной системе необходимо выбирать между доступностью (A) и согласованностью (C) (согласно теореме CAP), но в противном случае (E), даже когда система работает нормально при отсутствии разделения, необходимо выбирать между задержкой (L) и потерей согласованности (C).

На Хабре написана не одна статья на эти темы, поэтому не являясь экспертом в этой области, дам ссылку на очень хороший материал с более подробным обзором этих проблем.

Говоря про транзакции нельзя не сказать о существовании понятия **уровня изоляции транзакций** в базе данных — условное значение, определяющее меру допустимости получения несогласованных данных в результате выполнения *параллельных* транзакций в БД.

Существует шкала из **четырёх уровней** изоляции: Read uncommitted, Read committed, Repeatable read, Serializable. Первый из них является самым слабым, последний — самым сильным, каждый последующий включает в себя все предыдущие.

- **Read uncommitted** (чтение незафиксированных данных) — если несколько параллельных транзакций пытаются изменять одну и ту же строку таблицы, то в окончательном варианте строка будет иметь значение, определённое всем набором успешно выполненных транзакций. При этом возможно считывание данных, изменения которых ещё не зафиксированы (**грязное чтение**).
- **Read committed** (чтение фиксированных данных) — параллельно исполняемые транзакции видят только зафиксированные изменения других транзакций. Таким образом, данный уровень обеспечивает защиту от грязного чтения (каждая транзакция видит незафиксированные изменения другой транзакции), но не защищает от **чтения фантомов** (каждая транзакция видит вставленные другой транзакцией строки) и **неповторяющегося чтения** (каждая транзакция видит обновленные и удаленные другой транзакцией строки).
- **Repeatable read** (повторяющееся чтение) — уровень , при котором читающая транзакция «не видит» изменения читаемых данных другой транзакцией, но видит добавленные строки. При этом никакая другая транзакция не может изменять данные, читаемые текущей транзакцией, пока та не окончена.

- **Serializable** (упорядочиваемость) — транзакции полностью изолируются друг от друга. Достигается за счет того, что изменяющая транзакция блокирует всю таблицу или строки для изменяющих и читающих транзакций, а читающая транзакция блокирует всю таблицу или строки для изменяющих транзакций.

Waterfall (водопад/каскадная модель) — модель процесса разработки программного обеспечения, в которой процесс разработки выглядит как поток, последовательно проходящий фазы анализа требований, проектирования, реализации, тестирования, интеграции и поддержки.

Как правило, водопадная модель применяется в командах, работающих в парадигме проектного подхода.

Agile — это семейство гибких методологий управления проектами. К Agile относят основные фреймворки: Scrum, Kanban, Lean, LeSS, SAFe.

Суть Agile содержится в четырёх пунктах его **манифеста**:
- Люди и их взаимодействие важнее процессов и инструментов.
- Работающий продукт важнее исчерпывающей документации.
- Сотрудничество с клиентами важнее условий контракта.
- Реагирование на изменения важнее следования плану.

Основными преимуществами Agile является высокая гибкость к изменениям, скорость вывода нового функционала и снижение рисков в виду итеративной разработки и возможности получать обратную связь продукте по мере его разработки.

Основными недостатками являются сложности планирования сроков разработки и бюджета.
При Scrum продукт разрабатывается не разом, а по частям — каждая из них реализуется в рамках спринта.

Атрибуты команды, которая работает по Scrum:
- **Бэклог продукта** — упорядоченный список задач, который ведет и обновляет Product Owner.
- **Инкремент продукта** — законченная часть продукта, которая закрывает потребность пользователя.
- **Спринт** — это временной интервал (как правило, две недели), в течение которого идёт разработка функционала продукта.
- **Бэклог спринта** — упорядоченный список задач, которые запланированы на текущий спринт.
- **Доска разработки** — визуальное отображение задач команды разработки в текущем спринте со статусами (как правило: Бэклог, В работе, Тестирование, Готово).

предлагает набор инструментов и практик, которые позволяют улучшить работу команды.

Kanban — методология, в которой делается акцент на визуализацию задач, которые сейчас находятся в работе у команды. В зависимости от приоритетов, новые задачи могут поступать и браться в работу каждый день.

Главным инструментов в работе Kanban команды является **Kanban-доска**, состоящая из столбцов со статусами, как правило: **To Do** (планируется) **— In Progress** (в работе) **— In QA** (в тестировании) **— Done** (сделано).
в Agile-команде **рабочий процесс системного аналитика** в большинстве случаев выглядит примерно так:

- собрали требования;
- описали требования (бизнес-требования, функциональные и нефункциональные требования);
- согласовали с заинтересованными сторонами (стейкхолдерами);
- спроектировали HLD (high-level design);
- обсудили с командой;
- спроектировали LLD (solution-архитектуру, интеграции);
- обсудили с командой — декомпозировали задачи и оценили сроки;
- передали в разработку — поучаствовали в тестировании и приемке результатов.

Существует **методика** (матрица) **RACI**, которая является удобным и наглядным средством, определяющим участие различных ролей в процедурах и процессах. Обычно удобно вести список ролей, для того, чтобы не путаться в коммуникациях при работе на проекте или над задачей.

Термин RACI является аббревиатурой:

- R — Responsible (исполняет);
- A — Accountable (отвечает);
- C — Consult before doing (консультирует);
- I — Inform after doing (информируется).

В каждой процедуре каждой роли должен быть присвоен тот или иной литер, при этом Accountable — должен быть только один, Responsible — должен быть в наличии по каждой процедуре, каждая процедура обязательно должна иметь Accountable и Responsible.
ГОСТ 19 применяется для описания программного обеспечения, ГОСТ 34 используется для документирования автоматизированных систем, а ГОСТ 2 – для всего остального.

Что содержится в вашей типовой постановке задач для разработчика?

- Название проекта и его цель;
- Описание проблемы или задачи, которую нужно решить;
- Требования к решению (функциональные, нефункциональные, ограничения и т. д.);
- Критерии оценки результата;
- Сроки выполнения задачи;
- Необходимые ресурсы и среда разработки;
- Вопросы для обсуждения с командой и с владельцем продукта

Безопасность

- **SOAP**: SOAP изначально поддерживает WS-Security для защиты сообщений, что делает его более подходящим для приложений с высокими требованиями к безопасности (например, банковские системы).

- **REST**: REST не имеет встроенных стандартов безопасности, как SOAP, но безопасность может быть добавлена через использование HTTPS, OAuth, JWT и других методов.

WS-Security — это дополнение к SOAP, которое предоставляет расширенные функции безопасности, такие как шифрование отдельных частей сообщения, цифровые подписи, аутентификация, и оно работает на уровне содержимого сообщения, а не на уровне транспорта.

Основное отличие между хореографией и оркестрацией заключается в том, кто контролирует и координирует взаимодействие между компонентами. В хореографии каждый компонент самостоятельно принимает решения о своих действиях, в то время как в оркестрации центральный оркестратор определяет последовательность и координацию действий компонентов.

Оба подхода имеют свои преимущества и недостатки и могут быть применены в различных сценариях в зависимости от требований и особенностей системы.

Пример:

Пример Хореографии:

Представим систему электронной коммерции, в которой участвуют три сервиса: сервис заказов, сервис оплаты и сервис доставки. В хореографии каждый сервис знает о своих обязанностях и взаимодействует с другими сервисами на основе определенных правил и соглашений.

- Сервис заказов отправляет сообщение с информацией о заказе сервису оплаты.

- Сервис оплаты проверяет информацию о заказе и отправляет сообщение с результатом оплаты сервису доставки.

- Сервис доставки получает сообщение о результате оплаты и начинает процесс доставки заказа.

Каждый сервис принимает решения о своих действиях на основе полученных сообщений и событий. В данном примере, хореография определяет, как каждый сервис взаимодействует с другими сервисами для успешного выполнения заказа.

Пример Оркестрации:

Рассмотрим систему обработки заказов в интернет-магазине, где есть центральный оркестратор, который контролирует и координирует взаимодействие между сервисами.

- Оркестратор получает запрос на создание заказа от клиента.

- Оркестратор отправляет запрос на проверку наличия товара сервису инвентаризации.

- Сервис инвентаризации проверяет наличие товара и отправляет результат оркестратору.

- Оркестратор отправляет запрос на оплату заказа сервису платежей.

- Сервис платежей обрабатывает оплату и отправляет результат оркестратору.

- Оркестратор отправляет запрос на доставку заказа сервису доставки.

- Сервис доставки обрабатывает доставку и отправляет результат оркестратору.

- Оркестратор завершает процесс и отправляет клиенту уведомление о статусе заказа.

В данном примере, оркестратор определяет последовательность действий и координирует взаимодействие между сервисами для успешного выполнения заказа. Каждый сервис выполняет инструкции, полученные от оркестратора, и сообщает о своем состоянии или результате выполнения задач.

Поисковые пути в бд - порядок, в котором будут просматриваться схемы при поиске объекта (таблицы, типа данных, функции и т. д.), к которому обращаются просто по имени, без указания схемы.

Если объекты с одинаковым именем находятся в нескольких схемах, использоваться будет тот, что встретится первым при просмотре пути поиска.

К объекту, который не относится к схемам, перечисленным в пути поиска, можно обратиться только по полному имени (с точкой), с указанием содержащей его схемы.

Код ответа апи

Коды ответов HTTP (HTTP status codes) используются для информирования клиента (обычно браузера или API) о результате выполнения HTTP-запроса. Они делятся на пять основных категорий:

1. Информационные (1xx) — Запрос принят и продолжает обрабатываться.

- **100 Continue**: Сервер получил начальную часть запроса и клиент должен продолжить отправку оставшейся части.
- **101 Switching Protocols**: Клиент запросил изменение протокола, и сервер согласился на это.
- **102 Processing**: Сервер обрабатывает запрос, но это занимает длительное время.

2. Успешные (2xx) — Запрос был успешно выполнен.

- **200 OK**: Запрос успешно выполнен. Наиболее часто используемый код.
- **201 Created**: Запрос был выполнен, и в результате был создан новый ресурс.
- **202 Accepted**: Запрос принят для обработки, но сам по себе еще не выполнен.
- **204 No Content**: Запрос выполнен успешно, но возвращаемых данных нет.
- **206 Partial Content**: Возвращены только часть данных, обычно используется для работы с диапазоном данных.

3. Перенаправления (3xx) — Требуется дополнительное действие для завершения запроса.

- **301 Moved Permanently**: Запрашиваемый ресурс был перемещен на новый URL постоянно.
- **302 Found (Moved Temporarily)**: Ресурс временно перемещен на другой URL.
- **303 See Other**: Ресурс находится по другому URL, нужно использовать метод GET для его получения.
- **304 Not Modified**: Ресурс не был изменен, и можно использовать кэшированную версию.
- **307 Temporary Redirect**: Временная переадресация на другой URL с сохранением метода запроса.
- **308 Permanent Redirect**: Постоянная переадресация на новый URL с сохранением метода запроса.

4. Клиентские ошибки (4xx) — Проблема на стороне клиента.

- **400 Bad Request**: Некорректный запрос. Ошибка в синтаксисе или параметрах запроса.
- **401 Unauthorized**: Необходима аутентификация для доступа к ресурсу.
- **403 Forbidden**: Сервер понял запрос, но отказывается его выполнять (доступ запрещен).
- **404 Not Found**: Запрашиваемый ресурс не найден на сервере.
- **405 Method Not Allowed**: Метод запроса (GET, POST и т.д.) не поддерживается для этого ресурса.
- **408 Request Timeout**: Сервер ждал слишком долго на получение данных от клиента.

- **409 Conflict**: Конфликт при обработке запроса, например, при конфликте версий данных.
- **410 Gone**: Ресурс был удален и больше не доступен.
- **429 Too Many Requests**: Клиент отправил слишком много запросов за короткий период (ограничение скорости).

5. Ошибки сервера (5xx) — Проблема на стороне сервера.

- **500 Internal Server Error**: Общая ошибка сервера, возникшая при выполнении запроса.
- **501 Not Implemented**: Сервер не поддерживает функциональность, необходимую для выполнения запроса.
- **502 Bad Gateway**: Сервер, выступающий как шлюз, получил некорректный ответ от другого сервера.
- **503 Service Unavailable**: Сервер временно недоступен, обычно из-за перегрузки или обслуживания.
- **504 Gateway Timeout**: Сервер, выступающий как шлюз, не получил вовремя ответ от другого сервера.
- **505 HTTP Version Not Supported**: Сервер не поддерживает версию HTTP, указанную в запросе.

URI означает унифицированный идентификатор ресурса. Это строка, которая идентифицирует ресурс на сервере. Каждый ресурс имеет свой уникальный URI-идентификатор, который, будучи включенным в HTTP-запрос, позволяет клиентам обращаться к этому ресурсу и выполнять над ним действия. Процесс обращения к ресурсу с помощью его URI называется "адресацией".

GraphQL — это язык запросов, который позволяет клиентам запрашивать только те данные, которые им нужны. В GraphQL клиент определяет структуру и формат данных, которые он хочет получить, и сервер возвращает их в соответствии с этим запросом.

Ключевая разница заключается в том, что REST имеет фиксированный формат запроса и ответа для каждого ресурса, в то время как GraphQL позволяет клиентам определять свой запрос и получать только необходимую информацию, что делает его более эффективным и гибким в использовании.

REST и SOAP (Simple Object Access Protocol) — это два разных подхода к построению API. Вот 3 основные различия между ними:

- SOAP — это строгий протокол для построения безопасных API. REST — это не протокол, а архитектурный стиль, продиктованный набором рекомендаций, еще называемых принципами REST.
  - REST API проще в построении, легче и, как правило, быстрее, чем SOAP API.
- SOAP API считаются более безопасными, чем REST API, хотя в REST API все же могут быть реализованы средства защиты, делающие их достаточно надежными.
  - REST позволяет кэшировать ответы, в то время как SOAP этого не делает.

- SOAP кодирует данные в формате XML.
  - REST позволяет кодировать данные в любом формате, хотя наиболее популярны XML и JSON.

Асинхронный JavaScript, или AJAX — это набор технологий веб-разработки, используемых в веб-приложениях. По своей сути AJAX позволяет веб-странице выполнять запросы к серверу и обновлять интерфейс страницы без необходимости обновления всей страницы.

AJAX-клиент может использовать в своих запросах REST API, но AJAX не обязательно должен работать только с REST API. REST API могут взаимодействовать с любым клиентом, независимо от того, использует он AJAX или нет.

Кроме того, в отличие от REST, где для обмена сообщениями используются HTTP-запросы и ответы, AJAX посылает свои запросы на сервер с помощью объекта XMLHttpRequest, встроенного в JavaScript. Ответы сервера выполняются JavaScript-кодом страницы для изменения ее содержимого.

подход Contract First в разработке REST API — это методология, при которой спецификация и контракт API создаются и определяются до начала фактической разработки. Этот контракт служит важным документом, который определяет, как клиенты могут взаимодействовать с API и какие ожидаемые результаты будут получены от различных запросов.

Можно назвать следующие преимущества подхода Contract First:

- **Четкое определение API**: Спецификация и контракт API определяют, как API должно взаимодействовать с клиентами.
- **Уменьшение рисков**: Предварительное согласование контракта с заказчиками помогает уменьшить риски недопонимания и несоответствия ожиданиям от разработки API.
- **Улучшенная документация**: Текст контракта часто служит документацией для API, что упрощает его использование и интеграцию.

25. Что такое Code First подход к разработке REST API?

Ответ: Подход Code First в разработке REST API — это методология, при которой сначала разрабатывается функциональность API, а затем на основе этой функциональности автоматически генерируется спецификация PI. Отличительной чертой Code First подхода является то, что разработчики фокусируются на написании логики API и используют инструменты, которые позволяют автоматически создавать документацию и спецификацию на основе этой логики.

В принципе, оба подхода, Code First и Contract First, можно сочетать в рамках одного проекта разработки API. В этом случае, Code First используется для быстрого прототипирования, а затем Contract First для формализации контракта.

Основные этапы работы HTTPS:

1. Клиент запрашивает безопасное соединение.

Когда клиент (например, браузер) хочет подключиться к веб-серверу по HTTPS, он отправляет запрос с указанием домена и сообщает, что хочет установить защищенное соединение.

2. Сервер отвечает сертификатом.

Сервер отправляет клиенту **сертификат SSL/TLS**, который содержит **открытый ключ** и информацию о владельце сайта. Сертификат подписан доверенным центром сертификации (CA), чтобы убедить клиента в подлинности сервера.

3. Проверка сертификата.

Клиент проверяет сертификат на подлинность. Он проверяет, является ли сертификат действительным и подписан ли он центром сертификации, которому клиент доверяет. Если сертификат недействителен (истек срок действия, самоподписанный сертификат и т.д.), клиент предупреждает пользователя.

4. Обмен ключами шифрования.

После успешной проверки сертификата клиент и сервер обмениваются информацией для установки **сеансового ключа** шифрования. Обычно это происходит следующим образом:

- Клиент генерирует случайный **симметричный ключ** (сеансовый ключ) для шифрования данных.
- Этот ключ шифруется с использованием открытого ключа сервера (полученного из сертификата) и отправляется серверу.
- Сервер, имея свой **закрытый ключ**, расшифровывает этот сеансовый ключ.

5. Установка зашифрованного соединения.

Теперь обе стороны (клиент и сервер) имеют общий симметричный ключ, который используется для **симметричного шифрования** всех последующих данных, передаваемых между ними. Симметричное шифрование быстрее и эффективнее, чем асимметричное (RSA), поэтому его используют для самого обмена данными.

6. Передача данных по защищенному каналу.

После того как зашифрованное соединение установлено, весь трафик между клиентом и сервером шифруется с использованием симметричного ключа. Это защищает данные от перехвата и расшифровки злоумышленниками.

7. Аутентификация и целостность данных.

HTTPS также обеспечивает **аутентификацию** сервера и **целостность данных**. Это значит, что клиент может быть уверен, что общается с правильным сервером, и что передаваемые данные не были изменены по пути. Используются **хэш-функции** для проверки целостности данных.

Преимущества HTTPS:

1. **Шифрование:** Все данные, передаваемые между клиентом и сервером, зашифрованы и не могут быть перехвачены или прочитаны третьими лицами.
2. **Аутентификация:** Сертификат сервера подтверждает подлинность сайта и гарантирует, что клиент взаимодействует с правильным ресурсом, а не с поддельным сервером.
3. **Целостность данных:** HTTPS защищает передаваемые данные от изменений и манипуляций.

А если рассказать им про варианты использования JSON-RPC, то примут с распростертыми объятиями. А если объяснить интервьюеру про корреляцию между JSON-RPC и GraphQL, то сразу синьером сделают :)

Что такое SQL-инъекция -

https://practicum.yandex.ru/blog/sistemnaya-integraciya/

Задачи экран интернет магазины с фильтрами, описать процесс работы этого экрана. Какие постановки задач надо сделать на программистов, чтобы это заработало.