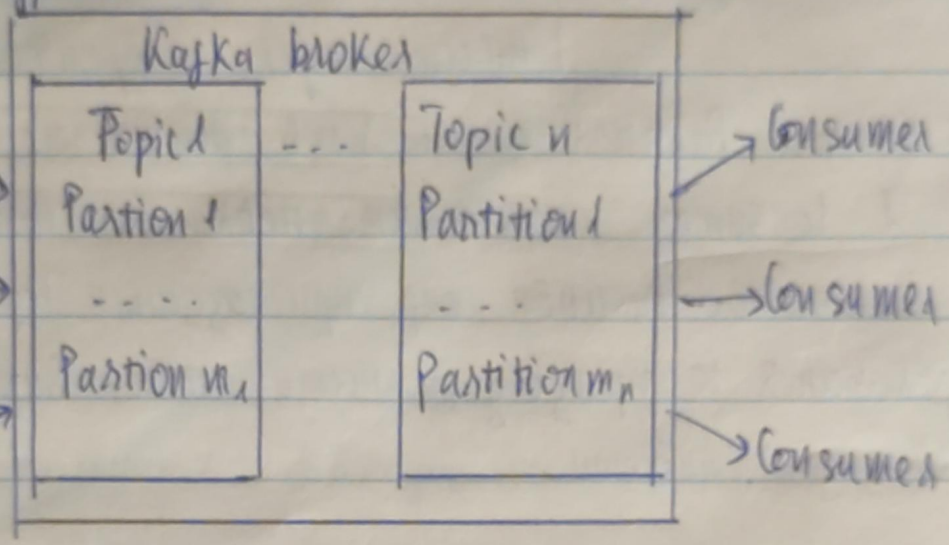
1) Kafka - distributed stream Processing System

*) Broker = Server

Опр: распределенный программный брокер сообщений.

Цель: Создание горизонтально масштабируемой платформы для обработки потоковых данных в реальной времени с
- * высокой пропускной, напр. TCP соед.
- * низкой задержкой

Схема:



→ Kafka хранит сообщения от Producers в формате "Ключ-значение". Consumers могут читать сообщения из Kafka broker.

1) DevOps: Подход, методология процесса обработки, кот. позволяет всей команде работать над продуктом ← эффективнее, быстрее

2 типа развертывания:

- Blue-green deployment: Весь пользовательский трафик одномоментно перераспределяется с одного экземпляра на другой.
- Canary deployment: переключение происходит постепенно,

начиная с части пользователей.

3) Kafka partition: ⇒ параллельная обработка данных ⇒ ↑ пропуск.
Сообщения сохранены в Kafka broker можно разделить на Partion в Topics

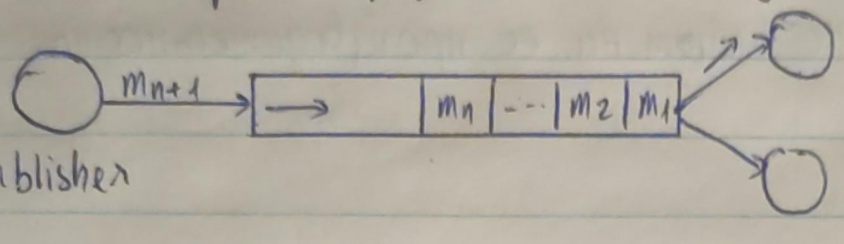
В ● 1 Partition, сообщение строго упорядочено по смещениям (offset) → положение сообщения в разделе.
~~ест~~ есть индекс и время создания.

Каждый Partition может быть ~~исп~~ исп. только 1-ым Consumer
1 Consumer может исп. несколько Partition

Кол-чество Partition опр. при создании Topic и зависит от требования к производительности, масштабируемости и отказоустойчивости.

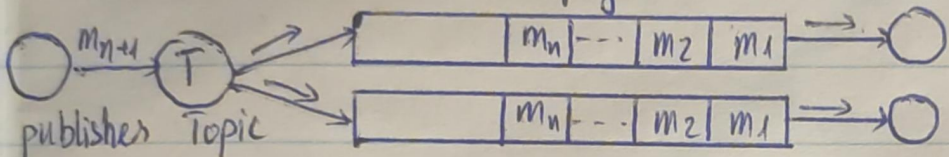
4) Continuous Integration / Continuous Deployment (Delivery): CI/CD
CI: частое объединение нескольких мелких изменений в основную ветку
CD (Delivery): П.О. может быть публикован в любое время.
CD (Deployment): Новые функции П.О. публикования автоматически.
⇒ Цель: ускорить обнаружение дефектов, повысить производите и обеспечить более быстрые циклы выпуска.

5) Очередь (Queue): односторонняя связь между Producer и Consumer через очередь. Producer вогдает команду Consumer.



Любое сообщение в очереди доступен только 1 consumer.

Публикация - Подписка (Pub-Sub): В PubSub, у каждого Consumer есть своя очередь.



→ Все Consumer получать сообщения от Producer

Выбор PubSub или Queue зависит от требования получения сообщения Consumer: Нужно ли в Каждое Consumer получит Каждое сообщ.?

6) Контейнеризация - это технология, которая позволяет упаковывать приложений и все их зависимости в 1 Контейнер.

→ { Обеспечивает изоляцию
Позволяет работы в различных средах

Kubernetes - это платформа для автоматизации развертывания, масштабирования и управления Контейнеризированными приложениями.

Контейнеры в Kubernetes объединяются в логические объекты под названием поды (Pods) - набор из одной или более базовых единиц, готовых к развертыванию на нодах (nodes)

7. Нефункциональные требования к ПО, какие есть характеристики у микросервисов, параметр измерения производительности

Нефункциональные требования к ПО (также известные как "качественные атрибуты") определяют характеристики системы, которые не связаны с ее функциональностью, но влияют на ее производительность, надежность, безопасность и другие аспекты

Для микросервисов:

1. Масштабируемость: система должна способствовать эффективному добавлению новых экземпляров микросервисов для обработки увеличенной нагрузки

Параметр измерения производительности: ремя отклика при увеличении числа экземпляров микросервисов или использование ресурсов при добавлении новых экземпляров

2. Устойчивость: система должна продолжать работать даже при отказе одного или нескольких микросервисов

Параметр измерения производительности: метрик, таких как время восстановления после сбоя и количество запросов, обработанных успешно в условиях отказа.

3. Надежность: система должна обеспечивать высокую доступность и минимизировать вероятность ошибок

Параметр измерения производительности: метрик, таких как время между отказами (MTBF), время восстановления после отказа (MTTR) и процент успешно обработанных запросов.

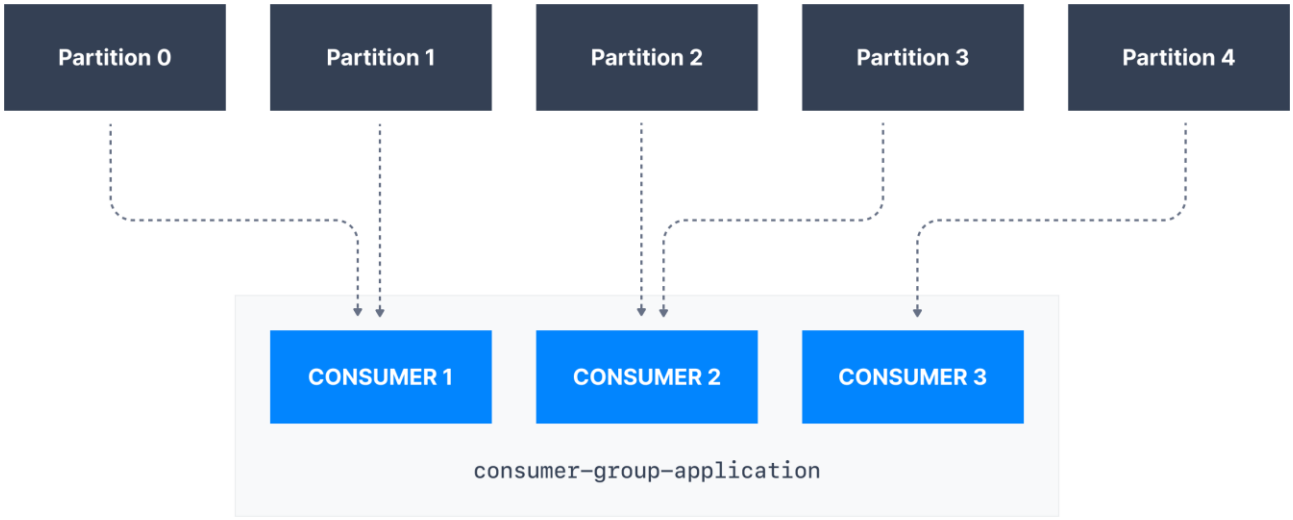
4. Производительность: Микросервисы должны обеспечивать высокую производительность и быстрый отклик на запросы

Параметр измерения производительности: метрик, таких как время отклика, пропускная способность, количество обработанных запросов в единицу времени и использование ресурсов.

8. Kafka: Consumer Group – почему потребители объединяются в группы, что этим достигается

Группа потребителей — это набор потребителей, которые сотрудничают для получения данных по некоторым темам. Разделы всех тем разделены между потребителями группы. По мере прибытия новых членов группы и ухода старых члены разделы переназначаются так, чтобы каждый член получал пропорциональную долю разделов. Это известно как ребалансировка группы.

Преимущество использования группы потребителей Kafka заключается в том, что потребители внутри группы будут координировать свои действия, чтобы разделить работу по чтению из разных разделов.



=> Распределение нагрузки, Обеспечение отказоустойчивости, Поддержка масштабируемости, Гарантированное уникальное прочтение, ...

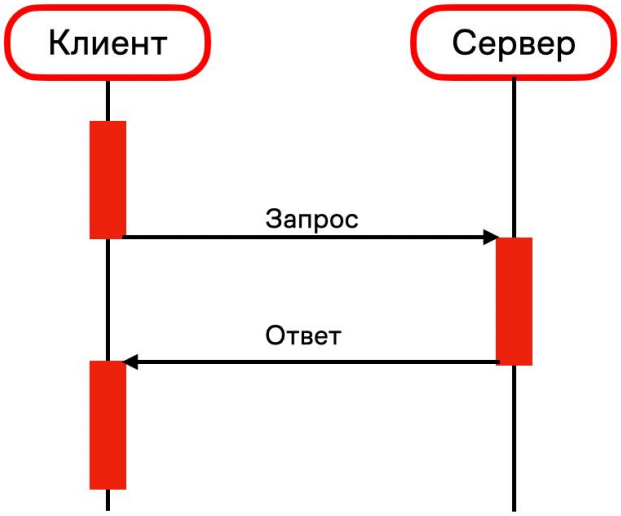
9. Синхронный и асинхронный обмен сообщениями: сравнить, условия использования

При синхронном обмене сообщениями отправитель и получатель дожидаются друг друга для передачи каждого сообщения, и операция отправки считается завершенной только после того, как получатель подтвердит получение сообщения. Или по-простому: «действия не делимы во времени».

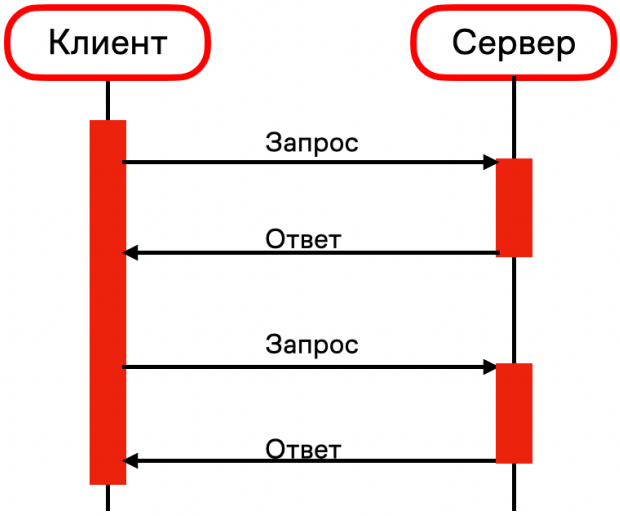
=> процессы синхронизируют свое выполнение во времени, для передачи сообщения не требуется использования дополнительных буферов.

При асинхронном обмене сообщениями не происходит никакой координации между отправителем и получателем сообщения. Для завершения операции отправителю не требуется дожидаться подтверждения получения сообщения процессом-получателем

=> возможность перекрывать вычисления отправителя и получателя во времени, т.к. процессы не будут ожидать друг друга для передачи каждого сообщения.



Синхронный



асинхронный

10. Сравнение SOAP и REST

REST (REpresentational State Transfer) – это набор правил того, как программисту организовать написание кода серверного приложения, чтобы все системы легко обменивались данными и приложение можно было масштабировать.

SOAP (Simple Object Access Protocol) – протокол обмена структурированными сообщениями в формате XML в распределённой вычислительной среде. (ko qu.tr.)

REST и SOAP на самом деле не сопоставимы. REST – это архитектурный стиль. SOAP – это формат обмена сообщениями. Давайте сравним популярные реализации стилей REST и SOAP.

Формат обмена сообщениями: В SOAP вы используете формат SOAP XML для запросов и ответов. В REST такого фиксированного формата нет

Определения услуг: SOAP использует **WSDL** (Web Services Description Language), REST не имеет стандартного языка определения сервиса

Транспорт: SOAP не накладывает никаких ограничений на тип транспортного протокола. REST подразумевает наилучшее использование транспортного протокола HTTP

11. Информационный обмен на основе REST API

Общие шаги для любого вызова REST API:

1. Клиент отправляет запрос на сервер.

2. Сервер аутентифицирует запрос клиента.

3. Сервер получает запрос и обрабатывает его внутри.

4. Сервер возвращает ответ клиенту.

Запрос клиента REST API содержит:

1. Уникальный идентификатор ресурса: сервер идентифирует каждый ресур идентификатором.

2. методы: HTTP-запрос сообщает серверу, что ему нужно делать с ресурсом: GET, POST, PUT, DELETE, ...

3. HTTP-заголовки: Метаданные, которыми обмениваются клиент и сервер, содержат важную информацию о запросе.

4. данные и параметры.

12. Сравнение форматов обмена сообщениями: JSON и XML

JSON – это открытый формат обмена данными, который могут читать как люди, так и машины. XML – это язык разметки, в котором есть правила для определения любых данных.

Формат: JSON использует пары «ключ-значение», чтобы создать картоподобную структуру, XML хранит данные в виде древовидной структуры со слоями информации, которую вы можете отслеживать и читать

Синтаксис: Синтаксис, используемый в JSON, более компактен и прост в написании и чтении

Поддержка типов данных: XML более гибкий и поддерживает сложные типы данных, такие как двоичные данные и временные метки

Безопасность: Синтаксический анализ JSON безопаснее, чем XML.

13. Методы http-запросов: какие знаете, условия использовани

HTTP-запрос (также называемый HTTP-метод) указывает серверу на то, какое действие мы хотим произвести с ресурсом.

GET запрашивает представление ресурса. Запросы с использованием этого метода могут только извлекать данные.

HEAD запрашивает ресурс так же, как и метод GET, но без тела ответа.

POST используется для отправки сущностей к определённому ресурсу. Часто вызывает изменение состояния или какие-то побочные эффекты на сервере.

PUT заменяет все текущие представления ресурса данными запроса.

DELETE удаляет указанный ресурс.

CONNECT устанавливает "туннель" к серверу, определённому по ресурсу.

OPTIONS используется для описания параметров соединения с ресурсом.

TRACE выполняет вызов возвращаемого тестового сообщения с ресурса.

PATCH используется для частичного изменения ресурса.

14. Распределенная архитектура информационных системах: структура и характеристика.

Распределенная система – это набор компьютерных программ, использующих вычислительные ресурсы нескольких отдельных вычислительных узлов для достижения одной общей цели

Характеристики:

- Совместное использование ресурсов: в распределенной системе могут совместно использоваться оборудование, программное обеспечение или данные.

- Параллельная обработка: одну и ту же функцию могут одновременно обрабатывать несколькомашин.

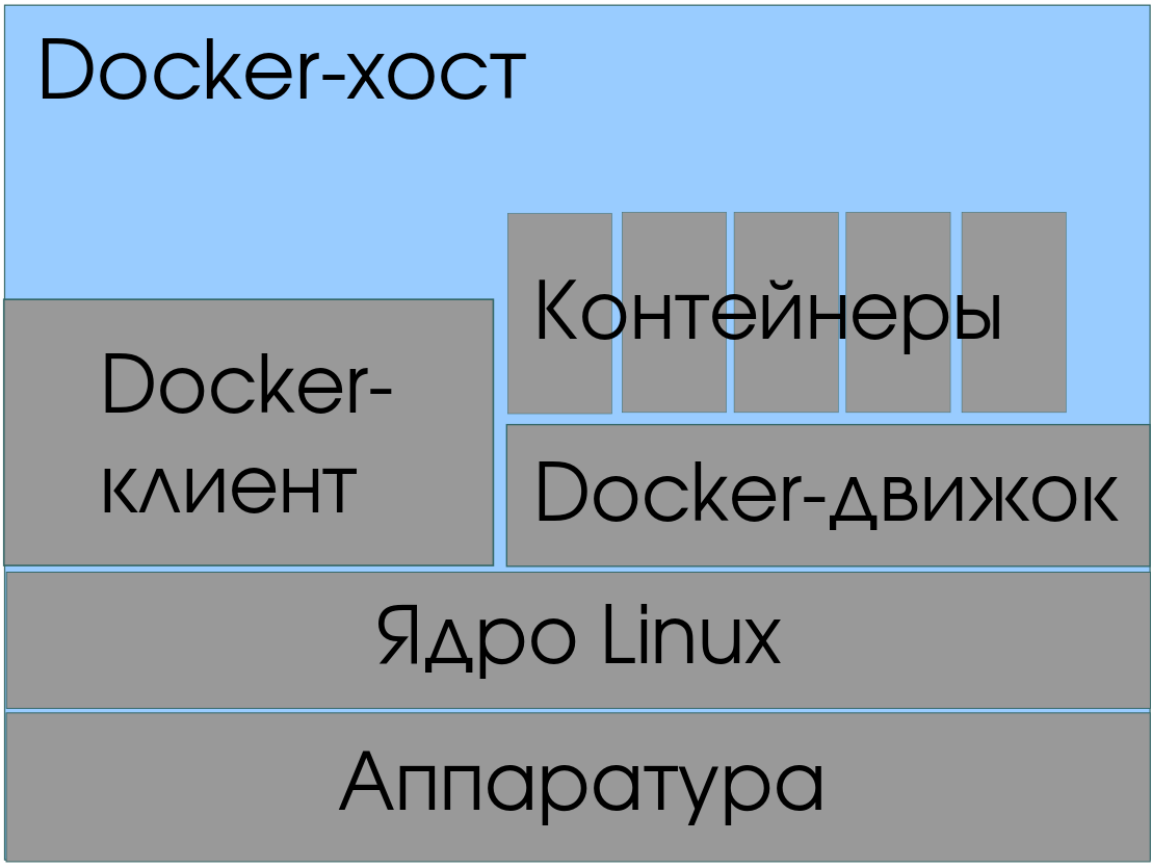
- Масштабируемость: вычислительная мощность и производительность могут масштабироваться по мере необходимости при добавлении дополнительных машин.

- Обнаружение ошибок: упрощается обнаружение отказов.

- Прозрачность: узел может обращаться к другим узлам в системе и обмениваться с ними данными.

15. Docker – что это, условия использования

Docker – программное обеспечение для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации, контейнеризатор приложений. Позволяет упаковать приложение со всем его окружением и зависимостями в контейнер, который может быть развёрнут на любой Linux-системе с поддержкой контрольных групп в ядре, а также предоставляет набор команд для управления этими контейнерами.



Использование:

Постоянное внедрение программного обеспечения. Docker и DevOps позволяют развертывать контейнерные приложения за несколько секунд.

Построение архитектуры на основе микросервисов.

Миграция устаревших приложений в контейнерную инфраструктуру.

Включение гибридных облачных и мультиоблачных приложений.

16. REST API: пример запроса и ответа, опишите состав и назначение параметров в примере

REST API:

Пример запроса:

```
GET /api/products/123 HTTP/1.1
Host: example.com
```

Пример ответа:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
  "id": 123,
  "name": "Example Product",
  "price": 99.99,
  "category": "Electronics"
}
```

1. Метод запроса (GET): Определяет тип операции, которую клиент хочет выполнить над ресурсом. В данном случае, это запрос на получение данных.

2. URL ресурса (/api/products/123): Указывает на конкретный ресурс, к которому обращается клиент. Здесь "123" - идентификатор продукта.

3. HTTP Заголовок Host: Определяет хост (доменное имя) сервера, к которому отправляется запрос.

4. Код состояния HTTP (200 OK): Сообщает клиенту о результате выполнения запроса. Код 200 означает успешный запрос.

5. Заголовок Content-Type: Указывает тип контента, который возвращается в ответе. В данном случае, это JSON.

6. Тело ответа: Содержит фактические данные, запрашиваемые клиентом. В данном примере, это информация о продукте с идентификатором 123 в формате JSON.

17. REST API: структура и назначение http-заголовка

- **Протокол REST** - Representational state transfer => **RESTful web API:**
- - Базовый URL сервиса имеет вид http://example.com/resources/
- - Форматы данных XML, JSON, HTML,…
- - Поддерживаются HTTP-методы (GET, PUT, POST, DELETE)
- - API является гипертекст-ориентированным
- - Поддерживаются операции CRUD (Create, Read, Update, Delete)
- - Легковесный механизм, по сравнению с SOAP
- - Нет встроенного контроля данных!

HTTP-заголовок - это часть HTTP-запроса или ответа, которая содержит информацию о запросе, ответе или об отправителе/получателе сообщения. Заголовок состоит из имени и значения, разделенных двоеточием.

Структура HTTP-заголовка выглядит следующим образом:

Имя_заголовка: Значение_заголовка

```
Host: ru.wikipedia.org
Accept: text/html, application/xml
Accept-Encoding: gzip, deflate
Accept-language: en-GB,en-US;q=0.9,en;q=0.8,de;q=0.7
```

Назначение HTTP-заголовка включает в себя следующие функции:

1. Определение типа контента: заголовки могут указывать тип содержимого (например, текстовый файл, изображение, видео и т. д.).

2. Управление кэшированием: заголовки могут указывать, как и на каком уровне кэшировать содержимое.

3. Управление сеансом: заголовки могут содержать информацию о сеансе, такую как куки, аутентификацию и т. д.

4. Указание параметров запроса: заголовки могут передавать параметры запроса, такие как язык, кодировку и т. д.

5. Указание параметров ответа: заголовки могут содержать информацию о сервере, дате и времени ответа и т. д.

Это лишь несколько примеров функций HTTP-заголовков. Общее назначение заключается в передаче метаданных о запросе или ответе, что позволяет серверу и клиенту взаимодействовать эффективно и безопасно.

18. Формат сообщений json: назначение, условия использования

+) JSON (англ. JavaScript Object Notation) — текстовый формат обмена данными, основанный на JavaScript. Но при этом формат независим от JS и может использоваться в любом языке программирования.

→ JSON используется в REST API. По крайней мере, тестировщик скорее всего столкнется с ним именно там.

+) В качестве значений в JSON могут быть использованы:

- JSON-объект: JSON-объект — неупорядоченное множество пар «ключ:значение», заключённое в фигурные скобки «{ }».
- Массив: Массив — упорядоченный набор значений, разделенных запятыми. Находится внутри квадратных скобок [ ].
- Число (целое или вещественное):
- Литералы *true* (логическое значение «истина»), *false* (логическое значение «ложь») и null
- Строка

→ При тестировании *REST API* чаще всего мы будем работать именно с объектами, что в запросе, что в ответе. Массивы тоже будут, но обычно внутри объектов. Комментариев в JSON нет.

+) Назначение: **Обмен данными**: JSON используется для передачи структурированных данных между клиентом и сервером в веб-приложениях, API и многих других системах; **Хранение данных**: JSON может использоваться для хранения и передачи данных в базах данных, файловых системах и других хранилищах; **Конфигурация**: JSON может использоваться для хранения конфигурационных параметров и настроек приложений.

19. Способы масштабирования баз данных: краткая характеристика и сравнение

Вертикальное масштабирование

Вертикальное масштабирование предполагает наращивание мощностей сервера. Основным преимуществом метода является его простота. Нет необходимости переписывать код при добавлении мощностей, а управлять одним крупным сервером намного проще, чем целой системой. Это же является и основным недостатком — масштабирование ресурсов одного сервера имеет вполне конкретные аппаратные ограничения. Также стоит учесть стоимость такого решения: сервер с кратным объёмом вычислительных ресурсов в большинстве случаев оказывается дороже, чем несколько менее мощных серверов, дающих в сумме такую производительность.

Вертикальное масштабирование баз данных

****

Горизонтальное масштабирование

Горизонтальное масштабирование означает увеличение производительности за счёт разделения данных на множество серверов. Такой способ предполагает увеличение производительности без снижения отказоустойчивости. Существует три основных типа горизонтального масштабирования.

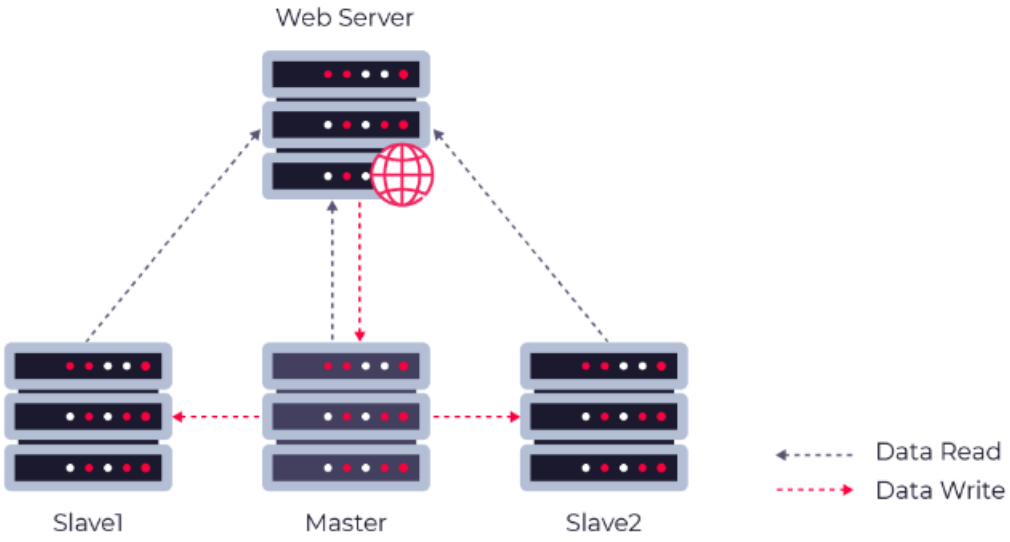
Горизонтальное масштабирование баз данных

****

Репликация

Этот термин подразумевает копирование данных между серверами. При использовании такого метода выделяют два типа серверов: master и slave. Мастер используется для записи или изменения информации, слейвы — для копирования информации с мастера и её чтения. Чаще всего используется один мастер и несколько слейвов, так как обычно запросов на чтение больше, чем запросов на изменение. Главное преимущество

репликации — большое количество копий данных. Так, если даже головной сервер выходит из строя, любой другой сможет его заменить. Однако как механизм масштабирования репликация не слишком удобна. Причина тому — рассинхронизация и задержки при передаче данных между серверами. Чаще всего репликация используется как средство для обеспечения отказоустойчивости вместе с другими методами масштабирования.

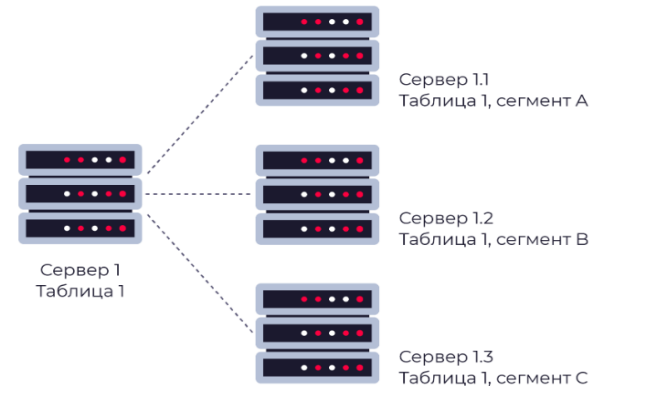


Репликация баз данных

****

Партицирование/секционирование

Данный метод масштабирования заключается в разбиении данных на части по какому-либо признаку. Например, таблицу можно разбить на две по признаку чётности. Причиной для использования партицирования является необходимость в повышении производительности. Это происходит из-за того, что поиск осуществляется не по всей таблице, а лишь по её части. Другим преимуществом этого метода является возможность быстрого удаления неактуального фрагмента таблицы.

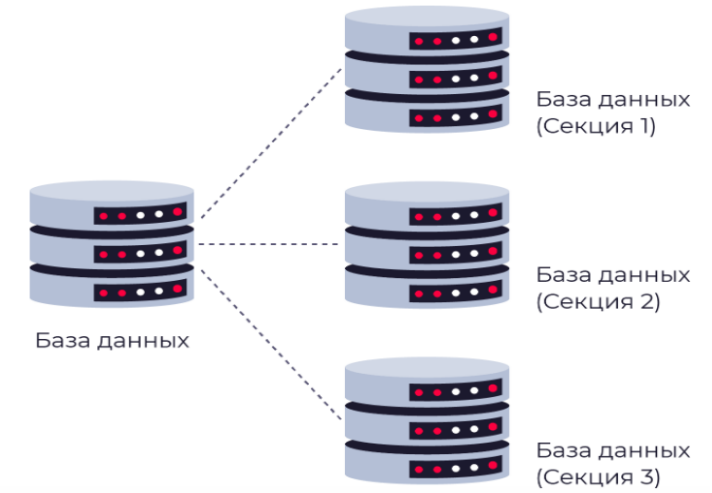


Секционирование баз данных

****

Шардирование/шардинг/сегментирование

Шардинг — это принцип проектирования базы данных, при котором части таблицы хранятся раздельно, на разных физических серверах. Шардинг является наиболее приемлемым решением для крупномасштабной деятельности, особенно если его использовать в паре с репликацией. Но стоит отметить, что это достаточно сложно организовать, так как необходимо учитывать межсерверное взаимодействие.



Сегментирование баз данных

20. Клиент-серверная архитектура.

+) **Клиент-серверная архитектура-** это модель организации вычислительных систем, в которой задачи распределены между клиентами и серверами. В такой архитектуре клиент, обычно являющийся пользователем или программой, запрашивает услуги или ресурсы у сервера, который отвечает на запросы, предоставляя необходимые данные или функциональность.

+) Компоненты клиент-серверной архитектуры

Клиент-серверная архитектура включает в себя следующие компоненты:

- Клиенты: это устройства или приложения, которые запрашивают информацию или услуги у сервера. Клиенты могут быть как программными приложениями (например, веб-браузеры, мобильные приложения), так и аппаратными устройствами (например, смартфоны, планшеты, терминалы, так называемые «тонкие клиенты»).
- Серверы: это компьютеры, которые предоставляют запрашиваемую информацию или услуги клиентам. Серверы могут выполнять различные функции, такие как хранение данных, обработка запросов, вычисления и т. д.
- Протоколы обмена данными: это правила или наборы инструкций, которые определяют, как клиенты и серверы обмениваются информацией. Некоторые из наиболее распространенных протоколов: HTTP/HTTPS (гипертекстовые протоколы), стек протоколов TCP/IP (набор правил, описывающих, как компьютеры соединяются и передают информацию друг другу), протоколы отправки и получения почты (SMTP, POP3, IMAP).
- Базы данных: это хранилища информации, которые используются на серверной стороне для хранения и управления данными. Базы данных позволяют серверу эффективно хранить, организовывать и извлекать информацию по запросу клиента.
- Сеть: это инфраструктура, которая обеспечивает связь между клиентами и серверами. Упрощенно говоря, сети могут быть локальными (LAN) и глобальными (WAN). Сеть обеспечивает передачу данных между клиентами и серверами по протоколам обмена данными.
- Система безопасности: это компонент, который обеспечивает защиту данных, передаваемых между клиентами и серверами. Этот компонент может включать в себя шифрование данных, аутентификацию и авторизацию клиентов, защиту от несанкционированного доступа и другие меры безопасности.
- Хранение и обработка данных: это компоненты, связанные с хранением данных на серверах и их обработкой. Это может включать в себя серверные операционные системы, системы управления базами данных (СУБД), серверы приложений и другие компоненты, необходимые для эффективной работы клиент-серверной архитектуры.

Особенности клиент-серверной архитектуры

Клиент-серверная архитектура является распространенной моделью для построения сетевых приложений. Она состоит из двух основных компонентов: клиента и сервера, которые взаимодействуют друг с другом посредством сетевого соединения.

Основные особенности клиент-серверной архитектуры:

- Разделение функций: как уже упоминалось выше, в клиент-серверной модели клиент выполняет запросы к серверу, а сервер осуществляет обработку этих запросов и предоставляет клиенту необходимые ресурсы или услуги. Разделение функций позволяет распределить нагрузку между клиентом и сервером, улучшить масштабируемость и обеспечить более эффективную обработку запросов.
- Сервер как «черный ящик»: клиент не знает, каким образом сервер выполняет его запросы и какие конкретно ресурсы используются. Для клиента сервер выглядит как единая сущность, с которой он взаимодействует, без необходимости знания о его внутренней работе.
- Надежность: клиент-серверная архитектура позволяет повысить надежность системы за счет распределения нагрузки между серверами. Например, в случае отказа одного сервера, клиенты могут переключиться на другой без прерывания обслуживания.
- Масштабируемость: клиент-серверная архитектура позволяет добавлять новых клиентов и сервера, что обеспечивает горизонтальную и вертикальную масштабируемость. Это позволяет системе эффективно обрабатывать растущую нагрузку и адаптироваться к изменениям в требованиях пользователей.
- Централизованное управление: сервер выполняет управление и контроль за ресурсами, данные и услуги которых предоставляются клиентам. Это упрощает управление системой и обеспечивает централизованные политики безопасности и доступа к данным.
- Клиент-серверная архитектура позволяет взаимодействовать с различными платформами и технологиями, используя открытые стандарты для обмена данными и коммуникаций.

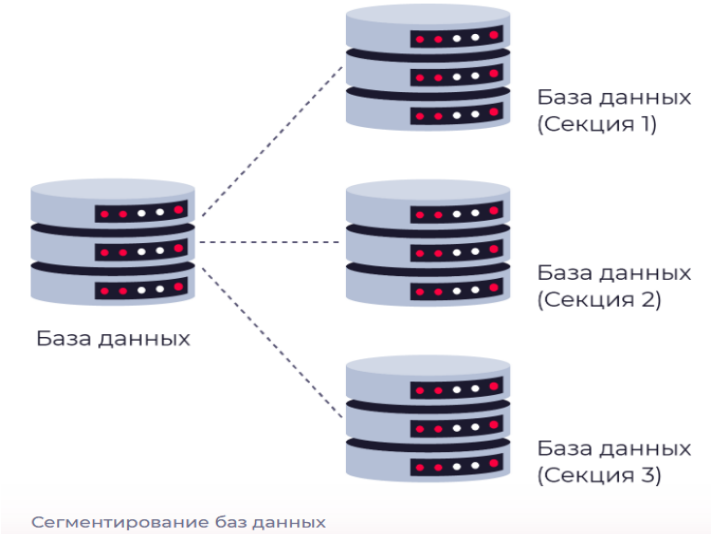
Области применения клиент-серверной архитектуры

Клиент-серверная архитектура широко применяется во многих областях, включая:

- Веб-разработка: клиент-серверная архитектура используется веб-сайтами и веб-приложениями, где клиентское приложение выполняет запросы к серверу для получения данных и отображения пользовательского интерфейса.
- Мобильные приложения также могут использовать клиент-серверную архитектуру, где клиентское приложение на мобильном устройстве взаимодействует с сервером для обработки данных и выполнения иных задач.
- Игры: клиент-серверная архитектура используется в онлайн-играх, где клиентское приложение игрока взаимодействует с игровым сервером для обмена данными и управления игровым процессом.

- Клиент-серверная архитектура применяется при работе с базами данных, где клиентские приложения обращаются к серверу базы данных для выполнения запросов и получения данных.
- В облачных вычислениях клиент-серверная архитектура используется для доступа к удаленным вычислительным ресурсам и хранению данных на удаленных серверах.
- Клиент-серверная архитектура широко используется в сетевых приложениях, где клиентское приложение взаимодействует с сервером для передачи данных и выполнения задач.
- Интернет вещей (IoT): клиент-серверная архитектура также применяется в системах Интернета вещей, где устройства клиентов обмениваются данными с сервером для управления и мониторинга.

21. Сегментирования баз данных



Вот тоже хорошая статья, мб лучше взять материал из неё https://aws.amazon.com/ru/what-is/database-sharding/

+) Сегментирование базы данных — это современный архитектурный шаблон разработки программного обеспечения, который предполагает разделение большой базы данных на более мелкие, более управляемые части, называемые сегментами или сегментами данных. Каждый сегмент представляет собой горизонтальный раздел данных в таблице базы данных и занимает несколько строк, содержащих подмножество данных, определенное ключом сегментирования. Этот подход значительно повышает производительность, масштабируемость и надежность, особенно в крупномасштабных приложениях с высоким трафиком, за счет распределения данных и рабочей нагрузки базы данных по нескольким серверам или кластерам. Таким образом, сегментирование помогает преодолеть ограничения традиционных монолитных баз данных, обеспечивая

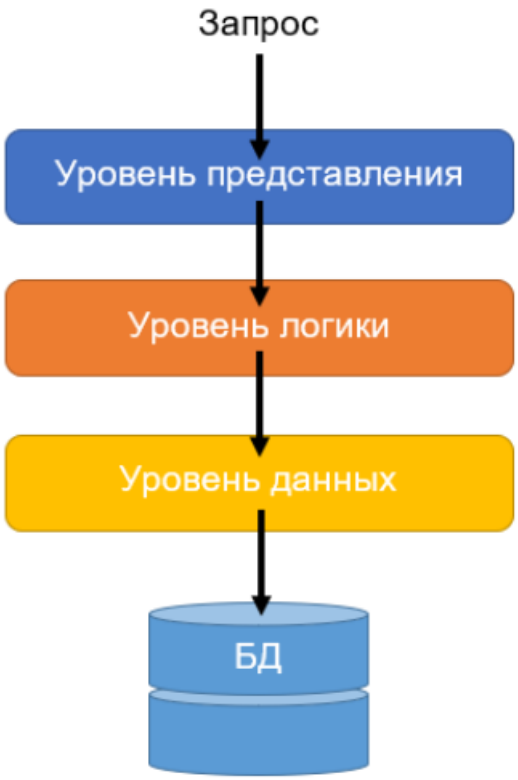
большую отказоустойчивость и эффективное использование вычислительных ресурсов.

+)Концепция сегментирования базы данных основана на более широком принципе горизонтального масштабирования, который предполагает добавление в систему большего количества серверов для равномерного распределения рабочей нагрузки. Эта стратегия позволяет приложениям обрабатывать больший объем трафика и рост данных, снижая вероятность возникновения узких мест и обеспечивая оптимальную производительность. Сегментирование может быть реализовано на разных уровнях, включая уровни приложения, промежуточного программного обеспечения и базы данных, в зависимости от конкретных вариантов использования и требований.

+)Эффективная стратегия сегментирования требует тщательного планирования и реализации, чтобы минимизировать влияние на производительность системы и целостность данных. Общие методы выбора сегментного ключа включают последовательное хеширование, разделение диапазона и хеширование по модулю, каждый из которых имеет свои преимущества и недостатки. Например, последовательные алгоритмы хеширования могут использоваться для обеспечения равномерного распределения данных по сегментам, минимизируя при этом количество переназначений ключей во время масштабирования. Напротив, секционирование по диапазону может обеспечить более высокую производительность запросов для определенных ключей сегментирования за счет поддержания порядка сортировки данных.

22. Виды архитектуры программных систем: краткая характеристика и сравнения условий использования

Многоуровневая архитектура



Это одна из самых распространенных архитектур. На её основе построено множество крупных фреймворков — Java EE, Drupal, Express. Пожалуй, самый известный пример этой архитектуры — это сетевая модель OSI.

Система делится на уровни, каждый из которых взаимодействует лишь с двумя соседними. Поэтому запросы к БД, которая обычно располагается в самом конце цепочки взаимодействия, проходят последовательно сквозь каждый «слой».

Архитектура не подразумевает какое-то обязательное количество уровней — их может быть три, четыре, пять и больше. Чаще всего используют трехзвенные системы: с уровнем представления (клиентом), уровнем логики и уровнем данных.

Событийно-ориентированная архитектура

- Характеристика: Процессы взаимодействия между компонентами строятся на событиях и реакциях на эти события.

- Условия использования: Подходит для систем, где важно быстрое реагирование на изменения и передача сообщений между компонентами.

В этом случае разработчик прописывает для программы поведение (реакции) при возникновении каких-либо событий. Событием в системе считается существенное изменение её состояния.

Можно провести аналогию с покупкой автомобиля в салоне. Когда автомобиль находит нового владельца, его состояние меняется с «продается» на «продано». Это событие запускает процесс предпродажной подготовки — установку дополнительного оборудования, проверку технического состояния, мойку и др.

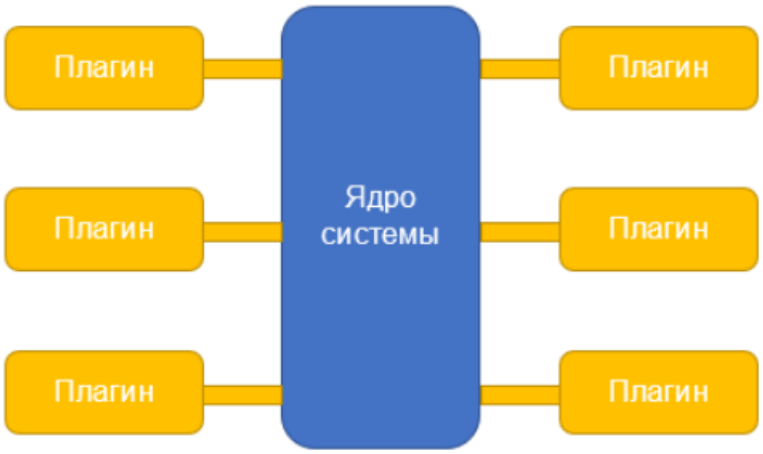
Система, управляемая событиями, обычно содержит два компонента: источники событий (агенты) и их потребители (стоки). Типов событий обычно тоже два: инициализирующее событие и событие, на которое реагируют потребители.

Микросервисная архитектура

- Характеристика: Приложение разбивается на небольшие независимые микросервисы, каждый из которых представляет собой отдельное приложение.

- Условия использования: Подходит для крупных и распределенных систем, требующих масштабируемости, отказоустойчивости и гибкости.

Микроядерная архитектура



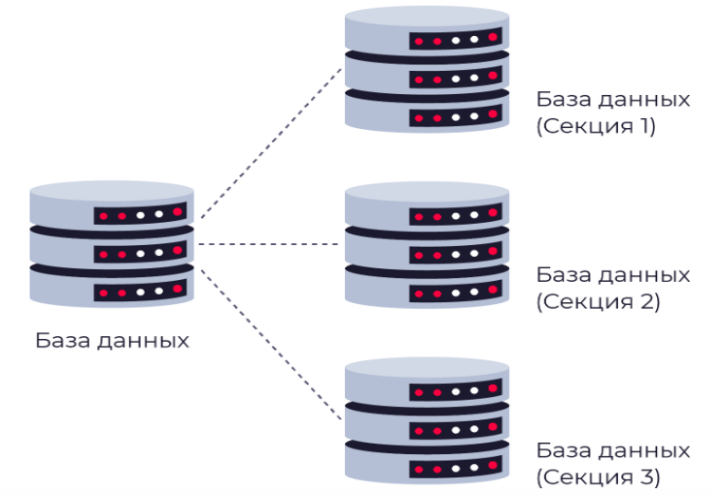
Этот тип архитектуры состоит из двух компонентов: ядра системы и плагинов. Плагины отвечают за бизнес-логику, а ядро руководит их загрузкой и выгрузкой.

Как пример микроядерной архитектуры приводится Eclipse IDE. Это простой редактор, который открывает файлы, дает их править и запускает фоновые процессы. Но с добавлением плагинов (например, компилятора Java) его функциональность расширяется.

23. Принципы шардирования баз данных

Шардинг — это принцип проектирования базы данных, при котором части таблицы хранятся раздельно, на разных физических серверах. Шардинг

является наиболее приемлемым решением для крупномасштабной деятельности, особенно если его использовать в паре с репликацией. Но стоит отметить, что это достаточно сложно организовать, так как необходимо учитывать межсерверное взаимодействие.



Сегментирование баз данных

Шардирование баз данных (Database Sharding) - это техника фрагментации и распределения данных по нескольким базам данных, что позволяет улучшить производительность и масштабируемость системы. Вот некоторые основные принципы, которые следует учитывать при реализации шардирования баз данных:

1. Горизонтальное шардирование: Данные разделяются по горизонтали, то есть по строкам или записям. Каждый шард (фрагмент) базы данных содержит часть данных.; Принцип: Данные разделяются на основе определенного критерия, такого как диапазон значений, хэширование ключей и т. д.

2. Ключ шардирования: Определяет, как данные будут распределены по различным шардам. Это может быть определенное поле или ключ в данных, на основе которого будет производиться разделение данных.

3. Балансировка нагрузки: При разделении данных на шарды необходимо обеспечить равномерное распределение нагрузки между шардами; Распределение должно быть таким, чтобы никакой шард не становился узким местом, обеспечивая тем самым равномерное использование ресурсов.

4. Репликация и отказоустойчивость: Каждый шард может иметь свои реплики для обеспечения отказоустойчивости. Это гарантирует доступность данных даже в случае отказа одного из шардов.

5. Транзакционная поддержка: Управление транзакциями и согласованностью данных между шардами требует специальных механизмов синхронизации и распределенных транзакций.

6. Управление метаданными: Необходимо иметь механизм управления метаданными, который отслеживает распределение данных по различным шардам и позволяет эффективно маршрутизировать запросы к соответствующим шардам.

24. Виды гарантий доставки сообщений

Гарантии доставки сообщений относятся к обеспечению корректной и успешной передачи сообщений в различных системах коммуникации. В зависимости от требований к системе обмена сообщениями и потребностей бизнес-логики, существуют различные уровни гарантий доставки сообщений. Вот некоторые из них:

1. At most once (Максимум один раз): Сообщение может быть доставлено ноль или один раз. Возможна потеря сообщений, но дубликаты исключены.

2. At least once (Минимум один раз): Сообщение будет доставлено минимум один раз. Это гарантирует, что сообщение будет доставлено, но возможно, будут обработаны дубликаты.

3. Exactly once (Точно один раз): Сообщение будет доставлено ровно один раз. Это предполагает исключение любых дубликатов и гарантирует только однократную обработку сообщения.

4. Delivery in order (Доставка в порядке): Гарантирует, что сообщения будут доставлены в том порядке, в котором они были отправлены.

5. Transactional delivery (Транзакционная доставка): Гарантирует, что сообщения будут доставлены как часть транзакции, и если что-то идет не так, она будет откатана, и сообщения не будут обработаны.

6. Guaranteed delivery (Гарантированная доставка): Это наивысший уровень гарантии, который предполагает, что сообщения будут доставлены в любых условиях сети, с учетом возможных отказов, захвата сообщений и т. д.

25. Хаускипинг баз данных

Хаускипинг баз данных (Database Housekeeping) относится к широкому набору процессов и практик, связанных с обслуживанием и управлением баз данных. Он включает в себя управление процессами очистки, оптимизации, архивации данных, а также обеспечение целостности и безопасности баз

данных. Вот несколько ключевых аспектов, которые обычно включаются в процессы хаускипинга баз данных:

+)Очистка данных: Удаление устаревших, неиспользуемых или ненужных данных из базы данных. Это позволяет освободить место, оптимизировать производительность и уменьшить излишнюю нагрузку на базу данных.

+) Управление индексами и структурами данных: Оптимизация индексов, перестройка таблиц, оптимизация запросов и обеспечение эффективности работы структур данных для улучшения производительности базы данных.

+) Архивация данных: Перемещение устаревших или редко используемых данных в архив для уменьшения объема данных, обязательств по соблюдению законодательства и управлению местом хранения.

+) Резервное копирование и восстановление: Регулярное создание резервных копий данных и установка механизмов восстановления для обеспечения защиты данных от потери, повреждения или катастроф.

+) Мониторинг и обслуживание интегритета данных: Проведение регулярных проверок целостности данных, обнаружение и устранение дубликатов, ошибок и проблем, которые могут повлиять на точность и надежность данных.

+) Управление доступом и безопасностью: Обновление политик доступа, управление пользователями, ролями, привилегиями и механизмами шифрования для обеспечения безопасности данных.

26. Application Program Interface

Хорошая статья на хабре https://habr.com/ru/articles/464261/

Application Programming Interface, или сокращенно API, называют интерфейс взаимодействия с приложением, позволяющий сервисам и прочим приложениям коммуницировать друг с другом.

API определяет функциональность, которую предоставляет программа (модуль, библиотека), но в то же время, API дает возможность абстрагироваться способа реализации этой функциональности. Как правило, между компонентами устанавливается иерархия, когда самые высокоуровневые компоненты используют API низкоуровневых и так далее по нисходящей спирали.

Ключевой аспект в работе с API — использование готового кода, либо постоянной функции, которая будет использована в конечном продукте.

Способ реализации здесь отходит на другой план, а разработчики получают возможность легко использовать наработки сторонних программистов.

В самом определении API выделяются два компонента:

+ Фрагмент ПО с конкретной функцией,

+ Отдельная часть приложения, либо полное приложение.

Фрагментирование отдельных частей можно определить по тому, насколько самостоятельным является компонент приложения. Чаще всего по такому сценарию устроены API отдельных библиотек, которые с его помощью взаимодействуют с остальными приложениями или частями сайта.

К API не относится скрытая логика приложения — разработчики вправе оставлять определенные области открытыми только для собственного использования.

В одном приложении таких объектов, взаимодействующих между собой, может быть много. У каждого из них есть свой API — набор характеристик и методов для взаимодействия с другими объектами в приложении.

Использование API на данный момент является повсеместным. Большинство ресурсов в интернете используют сразу несколько API, поскольку это решение также отличается своей надежностью и обширной практикой интеграции. Использование API стало своего рода отраслевым стандартом при создании современных сайтов или приложений.

27. Тиринг баз данных

Тиринг баз данных - это метод организации баз данных, в котором данные разделяются на несколько уровней или тиров в зависимости от их доступности и частоты использования. Обычно данные с более высоким уровнем доступности хранятся на более быстрых и дорогих устройствах хранения, в то время как данные с более низким уровнем доступности могут быть перемещены на медленные и более дешевые устройства. Это позволяет оптимизировать производительность и стоимость хранения данных в базе данных.

→Типичный пример. В течение дня базы данных, к которым постоянно обращаются пользователи, попадают на верхний уровень хранения, как и должно быть. Вечером, при наступлении окна бэкапа, СХД (Система хранения данных) видит, что теперь самые горячие данные — это не база данных, а бэкап, и успешно мигрирует бэкап на быстрые диски, а базы

данных — на медленные. На следующее утро, в начале рабочего дня, во время Boot strom, все пользователи начинают логиниться в систему, а на вернем уровне в это время — ночной бэкап. СХД понимает, что держит необходимые в это время данные на медленных дисках, начинает их потихоньку перемещать. Что в совокупности с одновременными массовыми обращениями к этим данным значительно сказывается на общей скорости работы СХД.

Существует два варианта инсталляций, в которых тиринг хорошо работает:

+)Крупные инсталляции с объемом данных свыше 1 петабайта. При таких объемах администратору сложно разбираться в большом количестве хранимой информации и вручную распределять по уровням.

+)Инсталляции, в которых заранее невозможно прогнозировать горячие данные (электронные библиотеки, игровые сервера, онлайн-кинотеатры, музыкальные архивы и др). Например, если внезапно стал популярен определенный автор или исполнитель, появляется интерес к какому-то видео, горячие данные мигрируют на верхний уровень хранения и серьезно ускоряют к ним доступ. В таких инсталляциях нивелируются слабые места технологии тиринга, и горячие данные поднимаются наверх четко в соответствии с количеством обращений.

→ Несмотря на то, что в некоторых инсталляциях тиринг действительно дает значимые преимущества, для большинства клиентов оптимально будет использовать технологию Flash Cache (Альтернатива — технология Flash Cache), вручную установить распределение данных по уровням хранения или разнести задачи на разные СХД.

28. Шаблон публикация-подписка

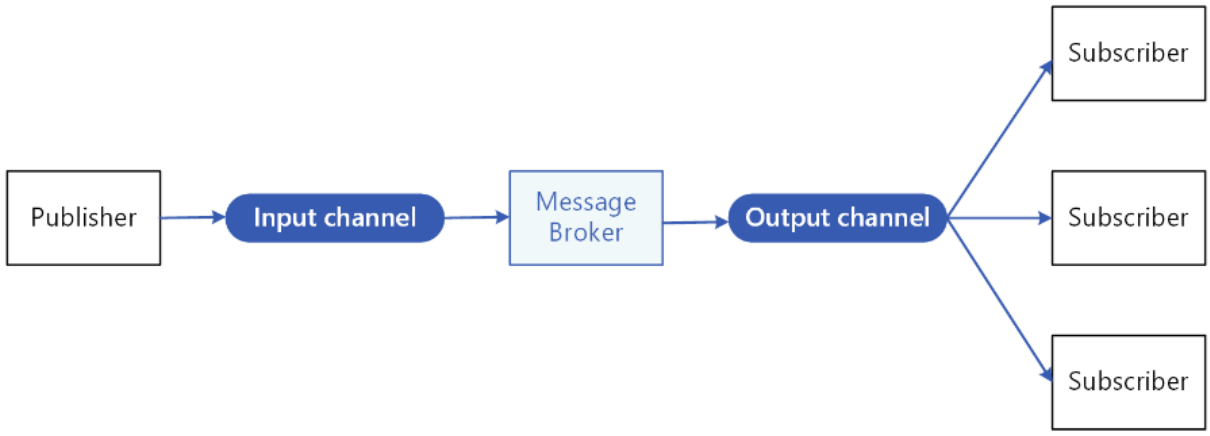
В архитектуре программного обеспечения публикация – подписка - это обмен сообщениями шаблон , где отправители сообщений , называемые издателями, не программируют сообщения для отправки непосредственно определенным получателям, называемым подписчиками, а вместо этого классифицируют опубликованные сообщения по классам, не зная, какие подписчики, если таковые имеются, может быть. Точно так же подписчики выражают интерес к одному или нескольким классам и получают только те сообщения, которые представляют интерес, не зная, какие издатели существуют, если таковые имеются.

Публикация – подписка является аналогом парадигмы очереди сообщений и обычно является частью более крупной системы промежуточного программного обеспечения , ориентированной на

сообщения. Большинство систем обмена сообщениями поддерживают модели pub / sub и очереди сообщений в своем API , например Служба сообщений Java (JMS).

Этот шаблон обеспечивает большую масштабируемость сети и более динамическую топологию сети , что приводит к снижению гибкости для изменения издателя и структуры публикуемых данных.

На следующей схеме показаны логические компоненты этого шаблона:



Обмен сообщениями с публикацией и подпиской имеет указанные ниже преимущества:

+) Разделение подсистем, которые по-прежнему должны взаимодействовать. Подсистемами можно управлять независимо друг от друга, и сообщениями можно должным образом управлять, даже если один или несколько получателей находятся в автономном режиме.

+) Увеличение масштабируемости и повышение отклика отправителя. Отправитель может быстро отправить одно сообщение во входной канал, а затем вернуться к своим основным обязанностям обработки. Инфраструктура обмена сообщениями отвечает за обеспечение доставки сообщений заинтересованным подписчикам.

+) Это повышает надежность. Асинхронный обмен сообщениями помогает приложениям продолжать работать без проблем под повышенной нагрузкой и более эффективно обрабатывать временные сбои.

29. Backend и Frontend – назначение, техника и протоколы взаимодействия

Определения из документа прутика:

Frontend включает в себя все, что видят пользователи: цвета, шрифты, изображения, графики, таблицы и кнопки. Frontend-разработчики реализуют удобный для пользователей интерфейс с помощью кода. Также они отвечают за адаптивность сайтов и веб-сервисов, которые должны правильно отображаться на экранах всех размеров.

Frontend-язык: только 1 - JavaScript. (HTML - стандартизированный язык гипертекстовой разметки документов для просмотра веб-страниц в браузере; CSS (каскадные таблица стилей)).
Backend хранит и упорядочивает данные, а также следит за тем, чтобы на стороне клиента все работало хорошо. Backend скрыт от глаз пользователей.
Backend-языки: *Ruby PHP, Python, Java, NodeJS, C#, Go, VB*
Связь между backend и frontend частями в разработке происходит следующим образом:
1. Frontend отправляет пользовательскую информацию в Backend.
2. В Backend информация обрабатывается.
3. После обработки данные возвращаются и принимают понятный для пользователя внешний вид.

Backend и Frontend - это две основные части веб-приложений, которые взаимодействуют друг с другом для обеспечения полноценного функционирования приложения.

Frontend (или клиентская часть) - это та часть приложения, с которой пользователь взаимодействует непосредственно. Это может быть веб-страница, мобильное приложение или другой пользовательский интерфейс. Frontend отвечает за отображение данных, взаимодействие с пользователем и отправку запросов на сервер для получения или обновления информации. Технологии, используемые в Frontend, включают HTML, CSS, JavaScript, а также фреймворки и библиотеки, такие как React, Angular, Vue.js и другие.

Backend (или серверная часть) - это та часть приложения, которая работает на стороне сервера и отвечает за обработку запросов от Frontend, управление базами данных, бизнес-логику, безопасность и другие аспекты функционирования приложения. Технологии, используемые в Backend, могут включать языки программирования (например, Python, Java, Ruby, PHP), фреймворки (например, Django, Spring, Ruby on Rails) и системы управления базами данных (например, MySQL, PostgreSQL, MongoDB).

Взаимодействие между Frontend и Backend происходит посредством различных протоколов и техник. Наиболее распространенными протоколами для взаимодействия между Frontend и Backend являются HTTP/HTTPS и WebSocket. HTTP используется для передачи запросов от клиента к серверу и ответов от сервера к клиенту. WebSocket позволяет установить постоянное соединение между клиентом и сервером для обмена данными в режиме реального времени.

30. RabbitMQ и Kafka: в чем отличие, условия использования

RabbitMQ и Apache Kafka - две популярные системы обмена сообщениями, которые используются для асинхронной коммуникации, обработки потоков данных и распределенных систем.

	RabbitMQ	Apache Kafka
Тип системы	RabbitMQ является брокером сообщений (message broker), он используется для обмена сообщениями в реальном времени между различными компонентами системы.	Apache Kafka является системой потоковой обработки и управления данными. Он специализируется на управлении и обработке больших объемов данных и осуществляет публикацию/подписку на основе ленты событий.
Условия использования	отлично подходит для случаев, когда требуется быстрая обработка сообщений в реальном времени, обработка очередей сообщений, роутинг сообщений на основе определенных правил и использование шаблонов обмена сообщениями, таких как point-to-point и publish-subscribe.	подходит для случаев, когда требуется обработка и хранение больших объемов данных, стриминговая обработка данных, управление событиями в реальном времени, аналитика данных и построение потоков обработки событий.
Порядок сообщений	RabbitMQ отправляет сообщения и ставит их в очередь в определенном порядке. Если сообщение с более высоким приоритетом не поставлено в очередь в систему, потребители получают сообщения в том порядке, в котором они были отправлены.	Kafka, в свою очередь, использует темы и разделы для постановки сообщений в очередь. Когда производитель отправляет сообщение, оно переходит в определенную тему и раздел. Поскольку Kafka не поддерживает прямые обмены между производителями и потребителями, потребитель получает сообщения из раздела в другом порядке.
Удаление сообщения	Брокер RabbitMQ маршрутизирует сообщение в очередь назначения. После прочтения потребитель отправляет брокеру ответ с подтверждением (ACK), который затем удаляет сообщение из очереди.	В отличие от RabbitMQ, Apache Kafka добавляет сообщение в файл журнала, который сохраняется до истечения срока хранения. Таким образом, потребители могут повторно обрабатывать потоковые данные в любое время в течение указанного периода.

Потребление сообщений	В RabbitMQ брокер гарантирует получение сообщения потребителями. Потребительское приложение играет пассивную роль и ждет, пока брокер RabbitMQ отправит сообщение в очередь. Например, банковское приложение может ждать SMS-предупреждение от программного обеспечения для централизованной обработки транзакций.	Однако потребители Kafka более активно читают и отслеживают информацию. По мере добавления сообщений в физические файлы журналов пользователи Kafka отслеживают последнее прочитанное сообщение и соответствующим образом обновляют свой трекер смещений. Трекер смещений — это счетчик, показатель которого увеличивается после прочтения сообщения. В случае с Kafka производитель не знает о получении сообщений потребителями.

31. Критерии качества API

1. Простота использования: API должен быть легким в использовании и иметь понятную документацию, чтобы разработчики могли быстро начать работу с ним.

2. Надежность: API должен быть стабильным и надежным, обеспечивая высокую доступность и минимальное количество ошибок.

3. Производительность: API должен обеспечивать высокую производительность и быстрые ответы на запросы.

4. Безопасность: API должен обеспечивать защиту от угроз безопасности, такие как атаки на основе инъекций, а также обеспечивать аутентификацию и авторизацию.

5. Масштабируемость: API должен быть способен масштабироваться для обработки больших объемов запросов и поддерживать рост бизнеса.

32. Брокер сообщений RabbitMQ

+) Брокеры сообщений — посредники между сервисами. Они находятся в центре архитектуры и управляют потоками информации. Благодаря этому каждый сервис может послать сообщение другому сервису или целой группе сервисов. Такой подход стал популярен с развитием микросервисов и заставил пересмотреть отношение к отказоустойчивости.

+)Если сломается брокер, в худшем случае вы потеряете часть сообщений, но ядро сервиса всё ещё будет работать, так как берет информацию из базы данных. Сообщения будут накапливаться, и когда

сервер вернётся, то быстро прочитает и обработает образовавшийся долг. Но иногда возникает частичная деградация: какая-то информация может оказаться не самой актуальной, хотя пользователи, скорее всего, этого даже не заметят. Пример — новостная лента в соцсетях. Если она не будет обновляться некоторое время, вы все равно сможете посмотреть опубликованные посты.

+) Бывает, что сервисы, спроектированные и написанные в краткие сроки, не имеют брокера сообщений. Потребность реализовать этот функционал осознаётся значительно позже. И здесь есть два варианта. Первый — сэкономить и в качестве брокера сообщений использовать базу данных. Это не очень хорошо, потому что решает проблему только на раннем этапе. Второй вариант — настроить брокер сообщений. Так вы сможете обеспечить своему приложению технологический запас для будущего развития. Одним из наиболее популярных брокеров сообщений остаётся RabbitMQ.

RabbitMQ — распределённый и горизонтально масштабируемый брокер сообщений. Упрощённо его устройство можно описать так:

+паблишер, который отправляет сообщения;

+очередь, где хранятся сообщения;

+подписчики, которые выступают получателями сообщений.

RabbitMQ передаёт сообщения между поставщиками и подписчиками через очереди. Сообщения могут содержать любую информацию, например, о событии, произошедшем на сайте.

Почему выбирают RabbitMQ

- RabbitMQ поддерживает несколько протоколов: AMQP, MQTT, STOMP и др., что позволяет использовать его в разных сценариях
- Основное преимущество RabbitMQ — гибкая маршрутизация. Сообщения маршрутизируются через exchange (обменник) перед попаданием в очереди. RabbitMQ предлагает несколько встроенных типов обмена для типичной логики маршрутизации.
- RabbitMQ поддерживает приоритезацию в очередях и позволяет настроить диапазон приоритетов. Приоритет каждого сообщения устанавливается при его публикации.
- RabbitMQ предлагает простой пользовательский интерфейс управления. Он позволяет контролировать каждый аспект брокера сообщений.