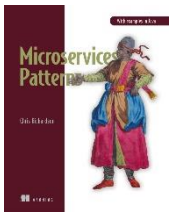
МИКРОСЕРВИСЫ



***Читайте также книгу Криса Ричардсона «Паттерны микросервисов»** (см. также памятку)
Microservices Patterns: With Examples in Java by Chris Richardson

Что такое монолит

Монолитная архитектура — это традиционная модель программного обеспечения, которая представляет собой единый модуль, работающий автономно и независимо от других приложений.

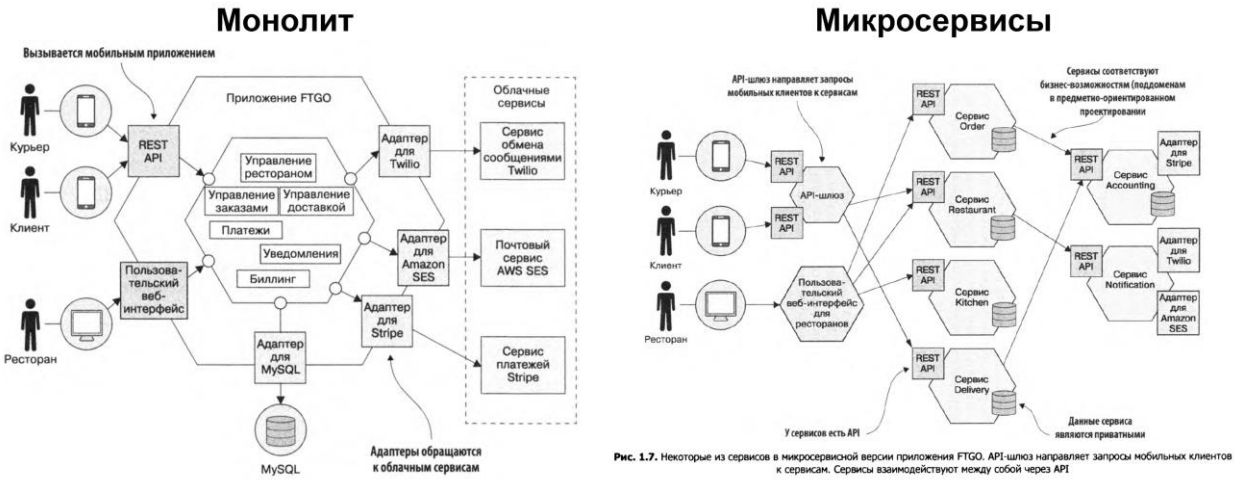


Рис. 1.7. Некоторые из сервисов в микросервисной версии приложения FTGO. API-шлюз направляет запросы мобильных клиентов к сервисам. Сервисы взаимодействуют между собой через API

Что такое микросервисы

Это архитектурный стиль, который создает приложения как набор небольших автономных сервисов, смоделированных на основе бизнес-доменов.

Внутренняя функциональность отдельного микросервиса может быть изменена или радикально модернизирована без влияния на остальную систему. Еще одним преимуществом использования микросервисов является простота развертывания.

Каждый отдельный сервис можно развернуть быстро и независимо от остальной системы, используя стандартные механизмы CI/CD. Кроме того, четко определенные API делают микросервисы удобными для авто тестирования.

Сервис — это мини-приложение, реализующее узкоспециализированные функции, такие как управление заказами, управление клиентами и т. д.

Обобщенное определение микросервисной архитектуры (или микросервисов) звучит так: это **стиль проектирования,** который **разбивает приложение** на отдельные сервисы с разными функциями.

Заметьте, что размер здесь вообще не упоминается. Главное, чтобы каждый сервис имел четкий перечень связанных между собой обязанностей.

Единого идеального решения не существует! Выбор зависит от цели бизнеса.

Отличие микросервисов от монолита. Плюсы и минусы каждого.

Отличия:
Монолитное приложение — это единый общий модуль, в то время как микросервисная архитектура представляет собой набор небольших независимо развертываемых служб.

	Монолит	Микросервисы
Плюсы	Простое развертывание. Использование одного исполняемого файла или каталога упрощает развертывание. Разработка. Приложение легче разрабатывать, когда оно создано с использованием одной базы кода. Производительность. В централизованной базе кода и репозитории один интерфейс API часто может выполнять ту функцию, которую при работе с микросервисами выполняют многочисленные API. Упрощенное тестирование. Монолитное приложение представляет собой единый централизованный модуль, поэтому сквозное тестирование можно проводить быстрее, чем при использовании распределенного приложения. Удобная отладка. Весь код находится в одном месте, благодаря чему становится легче выполнять запросы и находить проблемы.	Независимая разработка - все микросервисы можно легко разрабатывать в соответствии с их функциями Независимое развертывание на основе своих сервисов, их можно развернуть отдельно в любом приложении Изоляция сбоев - даже если служба приложения не работает, система может продолжать работать Стек гибридных технологий - разные языки и технологии могут использоваться для создания разных сервисов одного и того же приложения. Гранулярное масштабирование - отдельный компонент можно масштабировать по мере необходимости, нет необходимости масштабировать все компоненты вместе
Минусы	Снижение скорости разработки. Большое монолитное приложение усложняет и замедляет разработку. Масштабируемость. Невозможно масштабировать отдельные компоненты. Надежность. Ошибка в одном модуле может повлиять на доступность всего приложения. Препятствия для внедрения технологий. Любые изменения в инфраструктуре или языке разработки влияют на приложение целиком, что зачастую приводит к увеличению стоимости и временных затрат. Недостаточная гибкость. Возможности монолитных приложений ограничены используемыми технологиями. Развертывание. При внесении небольшого изменения потребуется повторное развертывание всего монолитного приложения.	Сообщение между самими сервисами сложное. Так как каждый функциональный элемент изолирован, требуется особая тщательность при построении между ними грамотной коммуникации, ведь им в любом случае придётся обмениваться запросами и ответами друг с другом. С увеличением количества сервисов сложность в построении их сообщения будет расти. Рост числа сервисов также влечет за собой и рост числа баз данных, с которыми эти сервисы соотносятся, так как, в отличие от монолитной архитектуры, микросервисы используют не одну общую базу данных. Сложность тестирования выражается в том, что сначала нужно отдельно разбираться с каждым сервисом, а потом тестировать корректное взаимодействие его с другими микросервисами. Микросервисы хуже подходят для использования внутри отдельных организаций, для них они могут оказаться неоправданно сложными в применении, в то время как для массовых интернет-сервисов они подходят отлично.

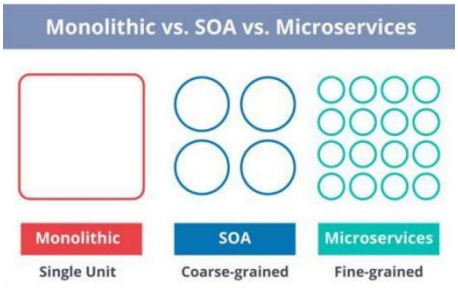
*Посмотрите также на платформе Каты курс Java Advanced, лекция 2.1.2. **Микросервисы VS монолит. В чём разница?**

В чем разница между SOA (service-oriented architecture) и микросервисной архитектурой?

Монолитная архитектура похожа на большой контейнер, в котором собраны и плотно упакованы все программные компоненты приложения.

Сервис-ориентированная архитектура – это набор взаимодействующих между собой сервисов.

Коммуникация может включать простую передачу данных или две или более службы, которые координируют определенные действия.



Архитектура микросервисов – это архитектурный стиль, в котором приложения создаются как набор небольших автономных сервисов, смоделированных на основе бизнес-доменов.

Таблица 1.1. Сравнение SOA и микросервисов

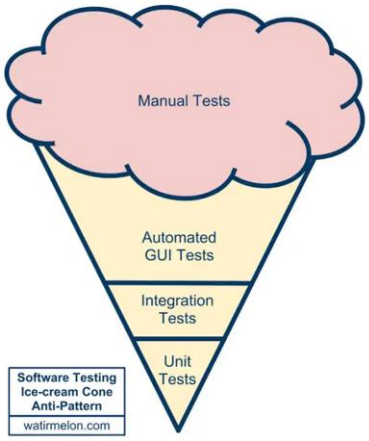
Параметр	SOA	Микросервисы
Межсервисное взаимодействие	Умные каналы, такие как сервисная шина предприятия, с использованием тяжеловесных протоколов вроде SOAP и других веб-сервисных стандартов	Примитивные каналы, такие как брокер сообщений, или прямое взаимодействие между сервисами с помощью легковесных протоколов наподобие REST или gRPC
Данные	Глобальная модель данных и общие БД	Отдельные модель данных и БД для каждого сервиса
Типовой сервис	Крупное монолитное приложение	Небольшой сервис

Приложения, спроектированные в виде микросервисов, обычно задействуют легковесные технологии с открытым исходным кодом. Сервисы взаимодействуют через примитивные каналы, такие как **брокеры сообщений** или простые протоколы, подобные REST или gRPC. SOA и микросервисная архитектура также по-разному обращаются.

Что делает сквозное тестирование в микросервисах?

Сквозное тестирование — это процесс, используемый для проверки того, что все независимые компоненты приложения работают должным образом и поддерживают функциональность бизнеса.

Он также используется, чтобы убедиться, что все компоненты работают вместе, посредством центрального концентратора, чтобы наилучшим образом поддерживать работу приложения.



Антипаттерн «Рожок мороженого»

Рис. 9.5. Пирамида тестов описывает относительные пропорции типов тестов, которые нужно написать. Продвигаясь вверх, вы должны писать все меньше и меньше тестов

Раньше под авто-тестами понимались тесты, которые обращались к приложению через пользовательский интерфейс. Этот подход быстро стал источником проблем, **став "рожком с мороженым".** Тестирование через пользовательский интерфейс довольно медленное, и в целом увеличивает время разработки. Часто оно требует лицензии на специальное ПО для автоматизации тестирования, что означает, что такое ПО может быть использовано только на индивидуальном рабочем месте. Обычно это ПО не может запущенно в консольном режиме, и **не может быть корректно помещено в deployment pipeline.**

Применяйте пирамиду тестов, чтобы определить, где следует приложить усилия при тестировании сервисов.

Большинство тестов должны быть **модульными, то есть быстрыми, надежными и простыми в написании.** Необходимо минимизировать количество сквозных тестов, т.к. они медленные и нестабильные, а их написание занимает много времени.

Какие существуют подходы к описанию микросервисной архитектуры?

Разбиение по поддоменам и бизнес-возможностям — два основных подхода к описанию микросервисной архитектуры приложения. Однако некоторые полезные рекомендации относительно декомпозиции берут свое начало в объектно-ориентированном проектировании.

Разбиение на сервисы по бизнес-возможностям
Бизнес-возможности организации определяются путем анализа ее целей, структуры и бизнес-процессов. Каждую возможность можно представить в виде сервиса, но для этого она должна ориентироваться на бизнес, а не на технические аспекты. Ее спецификация состоит из различных компонентов, включая ввод, вывод и соглашения уровня сервиса.

Разбиение на сервисы по проблемным областям
Предметно-ориентированное проектирование или domain-driven design, **DDD** — это способ построения сложных приложений, основанный на разработке объектно-ориентированной доменной (проблемной) модели.

Доменная модель организует **информацию о проблемной области** в формате, который можно применять для решения проблем в этой области.
Она определяет терминологию, используемую внутри команды, — так называемый язык описания.

Доменная модель находит свое воплощение в проектировании и реализации приложения.

DDD предлагает две концепции, чрезвычайно полезные с точки зрения микросервисной архитектуры: поддомены и изолированные контексты.

Поддомен является частью домена, то есть проблемной области приложения в терминологии DDD. Разбиение на поддомены происходит по тому же принципу, что и определение бизнес-возможностей: путем **анализа работы бизнеса** и определения разных областей знаний. Итоговые поддомены, скорее всего, будут похожи на бизнес-возможности.

В DDD область применения доменной модели называется **изолированным контекстом.** Изолированный контекст включает в себя код, который реализует модель.

Язык шаблонов проектирования микросервисов, паттерны проектирования (см. памятку)

Язык шаблонов — хороший способ описания архитектурных и проектировочных методик и помогает принять решение.

В итоге переход на ту или иную технологию обычно происходит в соответствии с циклом зрелости, состоящим из стадий:

- пик чрезмерных ожиданий (мы нашли панацею)
- избавление от иллюзий (это никуда не годится)
- плато продуктивности (теперь мы понимаем все плюсы и минусы и знаем, когда это лучше применять)

Шаблон (паттерн) проектирования — это многоразовое решение проблемы, возникающей в определенном контексте. Это идея, которая возникает как часть реальной архитектуры и затем показывает себя с лучшей стороны при проектировании программного обеспечения.
Программный шаблон решает архитектурную проблему, определяя ряд программных элементов, взаимодействующих между собой. Почему шаблоны проектирования так ценятся? Они должны **описывать контекст,** в котором их можно применять и то, что решение ограничено определенным контекстом и может плохо работать в других ситуациях.

Распространенная структура шаблонов включает в себя ТРИ особо важных раздела:

- причины;
- итоговый контекст;
- родственные шаблоны.

Раздел **«причины» (forces)** в шаблоне проектирования описывает различные аспекты проблемы, которую вы решаете в заданном контексте.

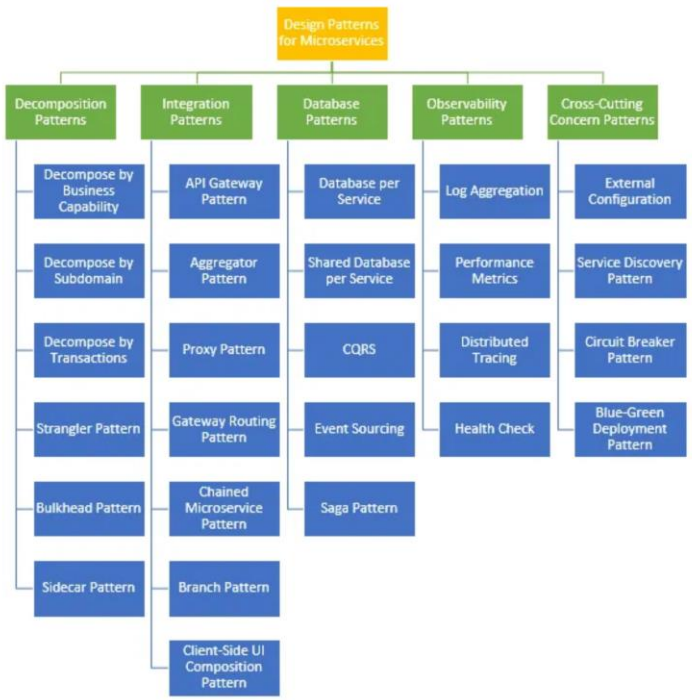
Раздел **«итоговый контекст»** (resulting context) описывает последствия применения шаблона проектирования. Он состоит из трех частей.

- Преимущества — включая аспекты проблемы, которые этот паттерн решает.
- Недостатки — недостатки шаблона, включая нерешенные аспекты проблемы.
- Замечания — новые проблемы, появляющиеся в результате применения шаблона.

Итоговый контекст формирует более комплексное и объективное представление о решении, что помогает сделать правильный выбор при проектировании.

Раздел **«родственные шаблоны»** (related patterns) описывает связь между действующим и другими шаблонами проектирования.

Паттерны микросервисов



Паттерны микросервисов по группам

1) Декомпозиция
2) Интеграция
3) Управление данными
4) Наблюдаемость
5) Решение сквозных проблем

Смотрите памятку «Паттерны микросервисов»

Circuit Breaker (предохранитель)

Шаблон Circuit Breaker обеспечивает стабильность, пока **система восстанавливается после сбоя** и снижает влияние на производительность (читайте подробнее в модуле Паттерны).

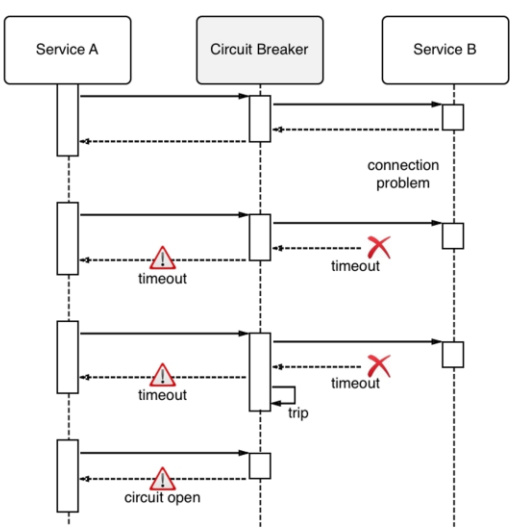
RPI-прокси, который в случае достижения определенного лимита последовательных отказов начинает **отклонять все вызовы,** пока не истечет определённое время.

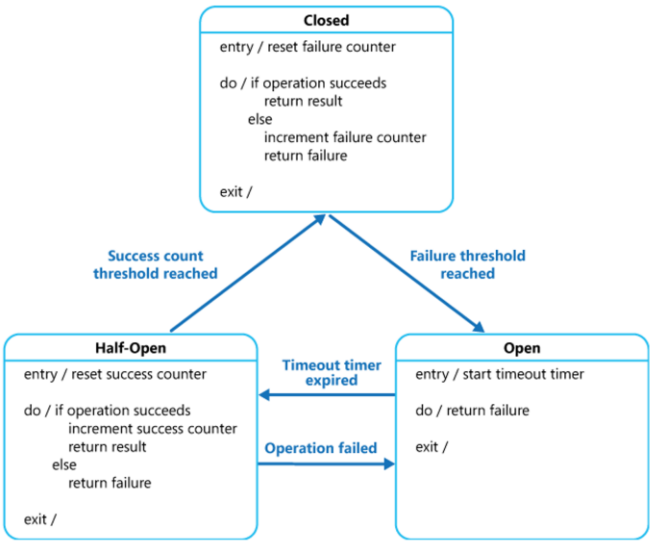
Проблема: как предотвратить каскадирование отказа сети или сервиса на другие сервисы?

Решение: Клиент сервиса должен вызывать удаленный сервис через прокси-сервер, который работает аналогично автомату защиты цепи (предохранителю).

Когда количество последовательных сбоев превышает пороговое значение, срабатывает предохранитель (сеть размыкается), и в течение периода тайм-аута все попытки вызвать удаленный сервис немедленно прекращаются.

По истечении тайм-аута предохранитель пропускает ограниченное количество тестовых запросов. Если эти запросы выполняются успешно, предохранитель возобновляет нормальную работу.
В противном случае, если происходит сбой, период тайм-аута начинается снова.





В Spring обычно берут реализацию из **Netflix** стека, которая называется **Hystrix** — это библиотека задержек и отказоустойчивости, это имплементация паттерна Circuit Breaker.

Как сказано из официальной документации: Hystrix — это библиотека, которая помогает вам контролировать взаимодействие между этими распределенными сервисами, добавляя терпимость к задержкам и логику отказоустойчивости.

Теперь, когда система представляет собой множество отдельно функционирующих веб-сервисов, **отказать может любой узел программы.** В связи с этим, крайне важно не допускать распространение ошибки, когда вышел из строя лишь один микросервис. Вся остальная часть системы должна мочь корректно функционировать, даже без некоторого набора неработающих сервисов.

Пример, который описан в официальном Get Started.
У вас есть система — книжный магазин. Главная страница сайта требует вывода книжных категорий, а также персональных рекомендаций для клиента на основе его статической модели. Конечно же, рекомендательный сервис пилится отдельной командой разработчиков на отдельном языке программирования в отдельном офисе. В общем, это типичный микро-сервис.

В том случае, если этот **сервис вышел из строя,** возможны 2 сценария: показать клиенту 500 ошибку, либо показать страницу с рекомендациями, полученными из других мест, либо вообще без рекомендаций.

В случае использования Hystrix от Нетфликса, вы можете описать fallback-метод, который автоматически исполнится в случае достижения некоторых условий (например, внешний сервис вернул 404/500, или вообще мы схватили тайм-аут ошибку). Всё это происходит крайне прозрачно для вас, вашей кодовой базы и ваших коллег. Слава богам, в Спринге есть отличная AOP реализация. Вот, кусок кода, показывающий вызов внешнего сервиса и fallback.

Код взят тут — http://www.baeldung.com/spring-cloud-netflix-hystrix

```
@Service
public class GreetingService {
    @HystrixCommand(fallbackMethod = "defaultGreeting")
    public String getGreeting(String username) {
        return new RestTemplate()
          .getForObject("http://localhost:9090/greeting/{username}",
          String.class, username);
    }

    private String defaultGreeting(String username) {
        return "Hello User!";
    }
}
```

Источник: https://hixon.ru/programmirovanie-2/spring-boot-circuit-breaker-pishem-svoj-hystrix.html

Saga pattern (повествование)

Проблема. Как реализовать транзакции, охватывающие сервисы?

Решение. Реализуйте каждую бизнес-транзакцию, охватывающую несколько сервисов, в виде саги. **Сага — это последовательность локальных транзакций.**

Каждая локальная транзакция обновляет базу данных и **публикует сообщение или событие,** чтобы инициировать следующую локальную транзакцию в саге.

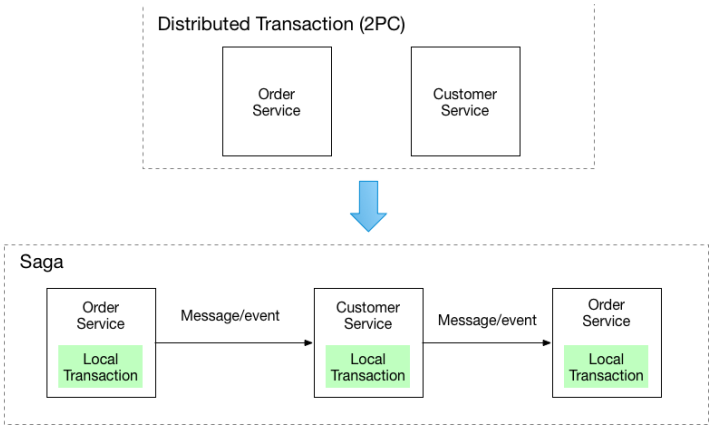
Если локальная транзакция терпит неудачу из-за нарушения бизнес-правила, **сага выполняет серию компенсирующих транзакций,** которые отменяют изменения, внесенные предыдущими локальными транзакциями.

Вместо ACID-транзакций **операция, охватывающая несколько сервисов** и стремящаяся поддерживать согласованность данных, должна использовать то, что называется повествованием или сагой, — **последовательность локальных транзакций на основе сообщений.**

Одна из проблем повествований связана с тем, что по своей природе они являются ACD (Atomicity, Consistency, Durability — «атомарность, согласованность, долговечность»).

Им **не хватает поддержки изолированности,** которая есть в **ACID-транзакциях.** В итоге приложение должно использовать так называемые контрмеры — методики проектирования, которые устраняют или снижают влияние аномалий конкурентности, вызванных нехваткой изолированности.

Этот паттерн предназначен для **управления распределенными транзакциями** в микросервисной архитектуре, где применение традиционного протокола двухфазной фиксации транзакций (Two-phase commit protocol, 2PC) становится трудноосуществимым.



Для координации транзакций существует два основных способа:

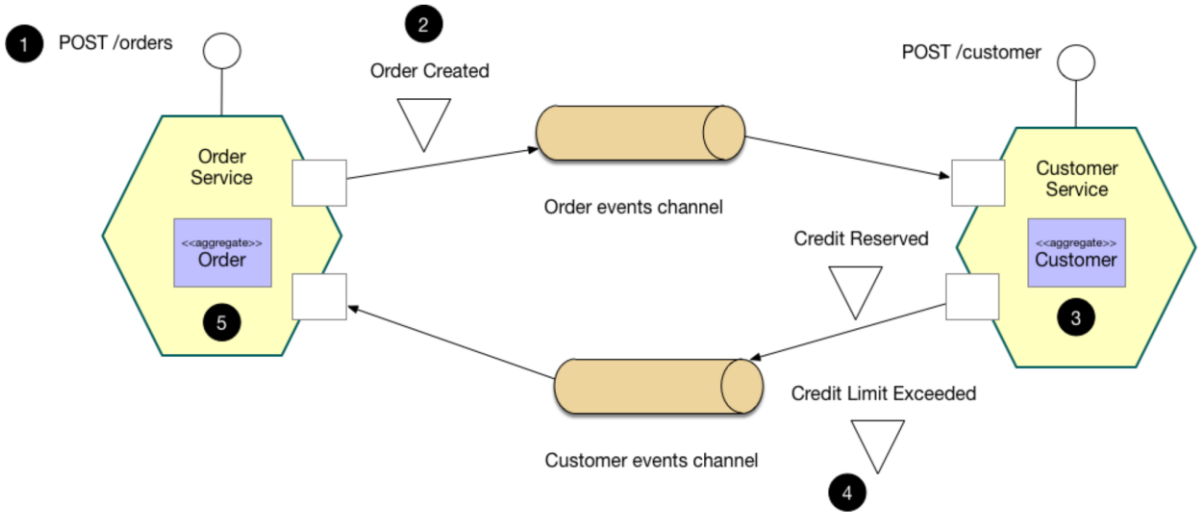
Хореография – децентрализованная координация, при которой каждый микросервис прослушивает события/сообщения другого микросервиса и решает, следует предпринять действие или нет.

Оркестрация – централизованная координация, при которой отдельный компонент (оркестратор) сообщает микросервисам, какое действие необходимо выполнить далее.

Использование шаблона обеспечивает согласованность транзакций в слабосвязанных распределенных системах, **однако увеличивает сложность отладки.**

Saga отлично подходит для систем, управляемых событиями и/или использующих базы данных NoSQL без поддержки 2PC, но **НЕ рекомендуется при использовании БД SQL** и в системах с циклическими зависимостями между сервисами.

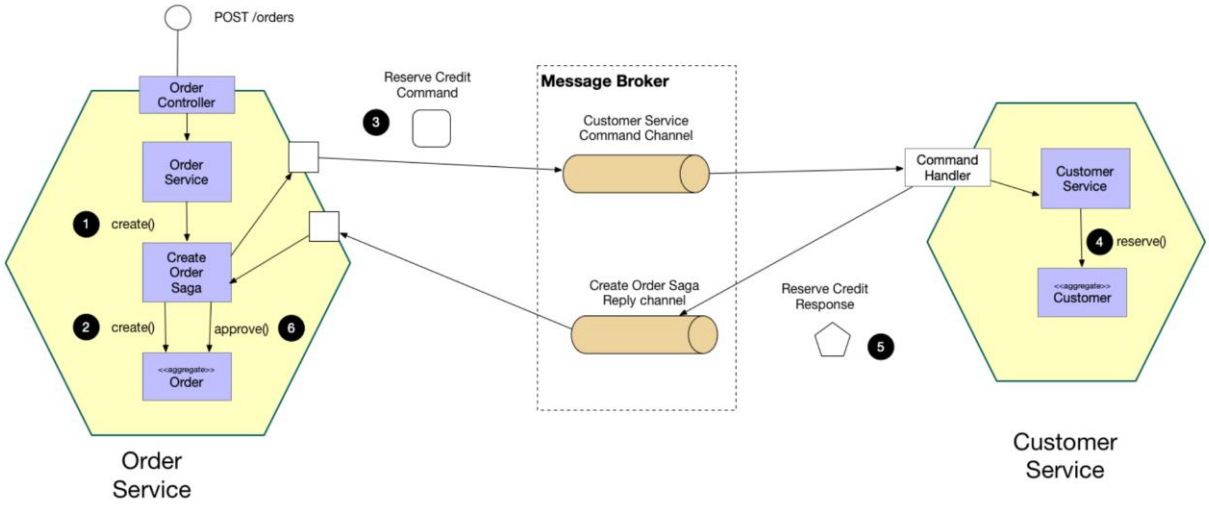
Пример: сага, основанная на хореографии



Приложение электронной коммерции, использующее этот подход, создаст заказ, используя основанную на хореографии сагу, состоящую из следующих шагов:

1. получает `Order Service` запрос `POST /orders` и создает `Order` в `PENDING` состоянии
2. Затем он генерирует `Order Created` событие
3. Обработчик `Customer Service` события пытается зарезервировать кредит
4. Затем он генерирует событие, указывающее результат
5. Обработчик `OrderService` событий либо одобряет, либо отклоняет `Order`

Пример: сага на основе оркестрации



Приложение электронной коммерции, использующее этот подход, создаст заказ, используя сагу на основе оркестровки, состоящую из следующих шагов:

1. Получает `Order Service` запрос `POST /orders` и создает `Create Order` оркестратор саги .
2. Оркестратор саги создает `Order` в `PENDING` штате
3. Затем он отправляет `Reserve Credit` команду на `Customer Service`
4. Попытки `Customer Service` зарезервировать кредит
5. Затем он отправляет ответное сообщение с указанием результата
6. Оркестратор саги либо одобряет, либо отклоняет `Order`

Фреймворк Eventuate Tram Saga – это инструмент оркестрации для Java микросервисов, использующих JDBC/JPA.

Платформа **Eventuate Tram** (код по ссылке) позволяет приложению атомарно обновлять БД и публиковать сообщения без использования JTA. https://github.com/eventuate-tram/eventuate-tram-sagas-examples-customers-and-orders

Load Balancer (балансировщик нагрузки)

«Падение» сервера (а оно всегда происходит неожиданно, в самый неподходящий момент) чревато весьма серьёзными последствиями — как моральными, так и материальными. Балансировка нагрузки может осуществляться при помощи как **аппаратных,** так и **программных инструментов.**

Балансировка на сетевом уровне предполагает решение следующей задачи: нужно сделать так, чтобы за один конкретный IP-адрес сервера отвечали **разные физические машины.** Такая балансировка может осуществляться с помощью множества разнообразных способов.

DNS-балансировка. На одно доменное имя выделяется несколько IP-адресов. Сервер, на который будет направлен клиентский запрос, обычно определяется с помощью алгоритма Round Robin (о методах и алгоритмах балансировки будет подробно рассказано ниже).

Построение NLB-кластера. При использовании этого способа серверы объединяются в кластер, состоящий из входных и вычислительных узлов. Распределение нагрузки осуществляется при помощи специального алгоритма. Используется в решениях от компании Microsoft.

Балансировка по IP с использованием дополнительного маршрутизатора.

Балансировка по территориальному признаку осуществляется путём размещения одинаковых сервисов с одинаковыми адресами в территориально различных регионах Интернета (так работает технология Anycast DNS.
Балансировка по территориальному признаку также используется во многих CDN.

см. интересный пример реализации https://habr.com/ru/company/ivi/blog/237349/

Балансировка на транспортном уровне является самым простым решением: клиент обращается к балансировщику, тот перенаправляет запрос одному из серверов, который и будет его обрабатывать.

Выбор сервера, на котором будет обрабатываться запрос, может осуществляться в соответствии с самыми разными алгоритмами (об этом ещё пойдёт речь ниже): путём простого кругового перебора, путём выбора наименее загруженного сервера из пула и т.п.

Различие между уровнями балансировки можно объяснить следующим образом:

К сетевому уровню относятся решения, которые не терминируют на себе пользовательские сессии. Они просто перенаправляют трафик и не работают в проксирующем режиме. На сетевом уровне балансировщик просто решает, на какой сервер передавать пакеты. Сессию с клиентом осуществляет сервер.

На транспортном уровне общение с клиентом замыкается на балансировщике, который работает как прокси. Он взаимодействует с серверами от своего имени, передавая информацию о клиенте в дополнительных данных и заголовках. Таким образом работает, например, популярный программный балансировщик HAProxy.

При балансировке на прикладном уровне балансировщик работает в режиме «умного прокси». Он анализирует клиентские запросы и перенаправляет их на разные серверы в зависимости от характера запрашиваемого контента. Так работает, например, веб-сервер **Nginx,** распределяя запросы между фронтендом и бэкендом. За балансировку в Nginx отвечает **модуль Upstream.**

Более подробно об особенностях балансировки Nginx на основе различных алгоритмов
https://www.digitalocean.com/community/tutorials/how-to-set-up-nginx-load-balancing

В качестве ещё одного примера инструмента балансировки на прикладном уровне можно привести **pgpool — промежуточный слой между клиентом и сервером СУБД PostgreSQL.** С его помощью можно распределять запросы по серверам баз данных в зависимости от их содержания. Например, запросы на чтение будут передаваться на один сервер, а запросы на запись — на другой.

Существует много различных алгоритмов и методов балансировки нагрузки. Выбирая конкретный алгоритм, нужно исходить, во-первых, из специфики конкретного проекта, а во-вторых — из целей, которые мы планируем достичь.

Очень желательно также, чтобы алгоритм балансировки обладал следующими свойствами

- предсказуемость: нужно чётко понимать, в каких ситуациях и при каких нагрузках алгоритм будет эффективным для решения поставленных задач;
- равномерная загрузка ресурсов системы;
- масштабирумость: алгоритм должен сохранять работоспособность при увеличении нагрузки.

Документация Kubernetes: создание внешнего балансировщика нагрузки
https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/

Gateway или API шлюз

Реализует сервис, который служит **точкой входа** в микро-сервисное приложение для клиентов внешнего API.

Проблема: как клиенты приложения на основе микросервисов получают доступ к отдельным сервисам?

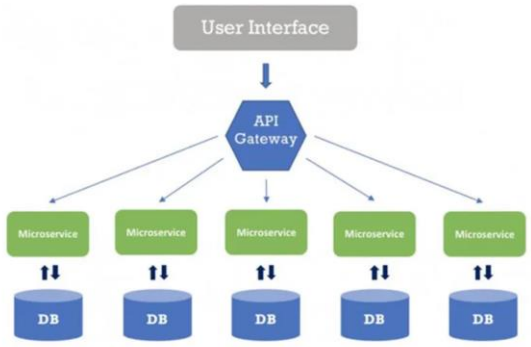
Решение: внедрите шлюз API, который является единой точкой входа для всех клиентов. Шлюз API обрабатывает запросы одним из двух способов. Некоторые запросы просто проксируются/маршрутизируются на соответствующий сервис. Он обрабатывает другие запросы, разветвляясь на несколько сервисов.

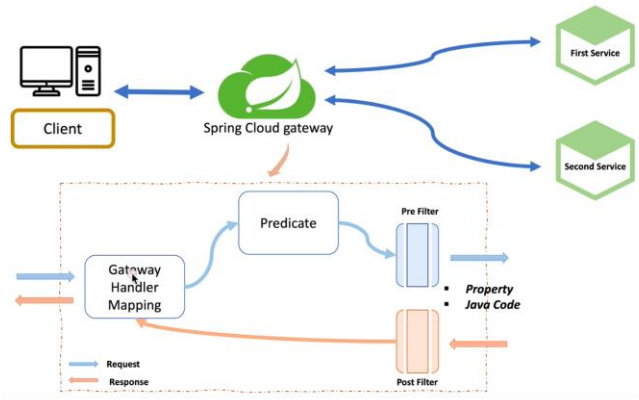
API Gateway — это единая точка входа для всех клиентов. Некоторые запросы просто проксируются или перенаправляются в соответствующую службу.

В микросервисной архитектуре число компонентов растет довольно быстро.

Клиентское приложение могло бы запрашивать каждый из сервисов самостоятельно. Но такой подход сразу натыкается на массу ограничений: необходимость знать адрес каждого эндпоинта, делать запрос по каждому фрагменту информации отдельно и самостоятельно обрабатывать результат.

Для решения таких проблем применяют некий **Gateway** — единую точку входа.
Ее используют для приема внешних запросов и маршрутизации в нужные сервисы внутренней инфраструктуры.





Шаблон Gateway является хорошей отправной точкой для архитектуры микросервисов, поскольку он позволяет направлять определенные запросы к различным службам.

см. также соответствующий раздел курса “Продвинутая Java” на платформе Каты

Паттерн API Gateway решает несколько проблем в архитектуре микросервисов:

1. Отделяет внешних потребителей от внутренней реализации услуг. Это позволяет сервисам развиваться и масштабироваться независимо, не затрагивая внешних потребителей.

2. Предоставляет единую точку входа для внешних потребителей, что упрощает обнаружение служб на стороне клиента и сокращает количество сетевых вызовов.

3. Может решать сквозные задачи, такие как безопасность, ограничение скорости и кэширование, на границе архитектуры, а не разбрасывать их по службам.

4. Может объединять несколько сервисов в один ответ, уменьшая количество сетевых вызовов и повышая производительность на стороне клиента.

5. Может выполнять переводы протоколов и типов содержимого, что позволяет реализовывать службы с использованием различных протоколов и форматов данных.

Он не является обязательным для архитектуры микросервисов, но обычно используется для управления сложностью и повышения производительности микросервисов. Его также можно использовать для обеспечения согласованной безопасности, ограничения скорости и политик кэширования в микросервисах.

Стоит отметить, что в зависимости от размера вашей среды микросервисов и количества запросов может иметь смысл иметь **несколько шлюзов API** для распределения нагрузки и повышения масштабируемости.

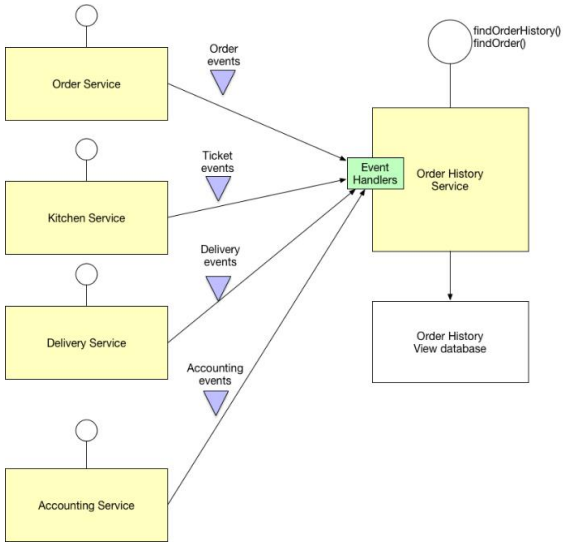
CQRS (command query responsibility segregation) – разделение ответственности командных запросов.

Реализует запрос, которому нужны данные из нескольких сервисов.

Для поддержания представления, реплицирующего данные из разных источников и доступного только для чтения, используются события.

Проблема: как реализовать запрос, извлекающий данные из нескольких сервисов в микросервисной архитектуре?

Решение: определите базу данных представления, которая является доступной только для чтения репликой, предназначенной для поддержки этого запроса. Приложение поддерживает реплику в актуальном состоянии, подписываясь на события домена, публикуемые сервисом, которому принадлежат данные.



Spring Cloud, что это и зачем нужно?

Документация тут https://spring.io/projects/spring-cloud#overview

Spring Cloud – это проект, который позволяет создавать распределенные приложения с микросервисной архитектурой. Использование Spring Cloud упрощает подключение к сервисам и получение возможностей окружения в облачных платформах.

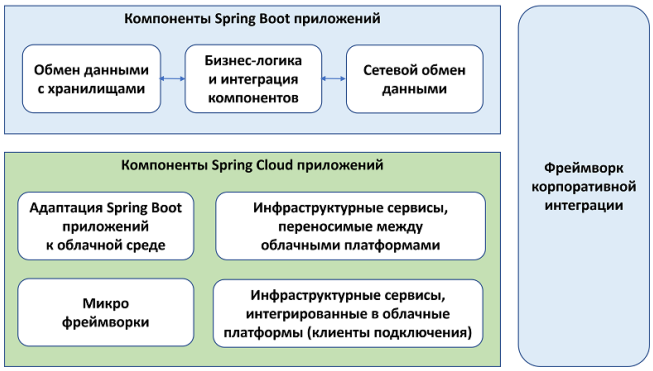
Со Spring Cloud можно легко и быстро создавать шаблоны в распределённых системах. Из примеров таких шаблонов: управление конфигурацией, обнаружение сервисов, интеллектуальная маршрутизация, микро прокси, одноразовые токены и многое другое.

Шаблоны, созданные с помощью Spring Cloud, будут хорошо работать в любой распределённой среде, включая ваш собственный ноутбук, центры обработки данных и PaaS-платформы, такие как Cloud Foundry.

Spring Cloud также состоит из множества под-проектов для разных целей. Так, Spring Cloud Azure интегрирует Spring со службами Azure, Spring Cloud Stream используется для создания управляемых событиями микросервисов (event-driven microservices) и так далее.

Характеристики

- Распределённая / версионная конфигурация
- Регистрация и обнаружение сервисов
- Маршрутизация
- Связь между сервисами (service-to-service calls)
- Балансировка нагрузки
- Выбор лидера и состояние кластера
- Распределённый обмен сообщениями

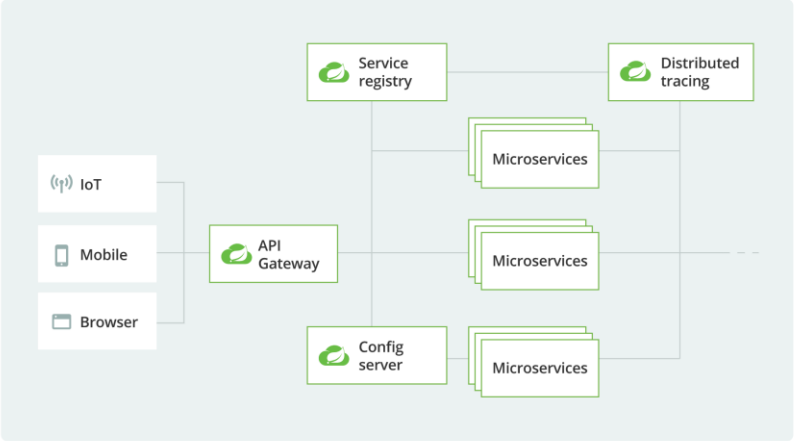


Создав приложение и освоив контейнеризацию, можно легко, буквально одной цифрой в конфиг-файле, настроить запуск контейнеров во множестве экземпляров, для отказоустойчивости и масштабирования. Но возникает проблема, – потребитель сервисов не умеет работать с множеством экземпляров. Для связи прикладных сервисов в целостное приложение, необходимы инфраструктурные сервисы.

Подсистема Spring Cloud состоит из:

- Зависимостей, адаптирующих Spring Boot приложения к облачной среде.
- Инфраструктурных сервисов, в виде готовых Spring-приложений, переносимых между облачными платформами.
- Клиентов сетей Service Mesh, реализующих инфраструктурные сервисы.
- Клиентов облачных платформ, реализующих инфраструктурные и коммерческие сервисы.
- Микро-фреймворков для разработки микросервисов и их оркестрации.

Особенности архитектуры **Spring Cloud**



Чтобы ориентироваться в тексте, важно понимать отличие терминов «инфраструктурные сервисы» от «облачных платформ» и «прикладных сервисов».

Прикладные сервисы реализуют бизнес-логику распределенного приложения.

Инфраструктурные сервисы координируют взаимодействие экземпляров прикладных сервисов.

Основные функции:

- Маршрутизация – распределение запросов между типами сервисов;
- Балансировка – распределение запросов между экземплярами сервисов;
- Проброс пользовательской аутентификации из шлюза в сервисы;
- Защита межсервисных запросов от взлома и повреждения техническими сбоями;
- Механизмы отказоустойчивости;
- Реализация манифеста 12-факторных приложений. https://habr.com/ru/post/258739/

Облачные платформы распределяют по физическим серверам docker-контейнеры или поды (pod).

Основные функции:

- Деплой на физические серверы комплекса контейнеров, реализующих микро-сервисные приложения.
- Контроль и поддержка работоспособности контейнеров, путем удаления неработоспособного контейнера и создания нового экземпляра.
- Обновление версий микросервисного приложения, путем удаления устаревших контейнеров и создания новых экземпляров.

Опционально:

- Инфраструктурные сервисы;
- Коммерческие сервисы (встроенные MQ-брокеры, базы данных, сервисы отправки SMS и т.п.)

Под (pod) – минимальный юнит развертывания в Kubernetes, содержит произвольное количество docker-контейнеров.

Основные модули Spring Cloud

Читать статью https://habr.com/ru/post/280786/

Ссмотреть видео Микросервисы со Spring Boot & Spring Cloud (Александр Бармин)
https://www.youtube.com/watch?v=2yAbbsuNBPc

Config Server (центральная конфигурация)

Для разработчика config server — это еще один микросервис, еще одно простейшее Spring Boot приложение с минимальным набором зависимостей, которое крутится рядом с другими микросервисами.

Spring Cloud Config позволяет хранить настройки конфигурации сервисов в git-репозитории и управлять настройками централизованно. https://habr.com/ru/company/otus/blog/590761/

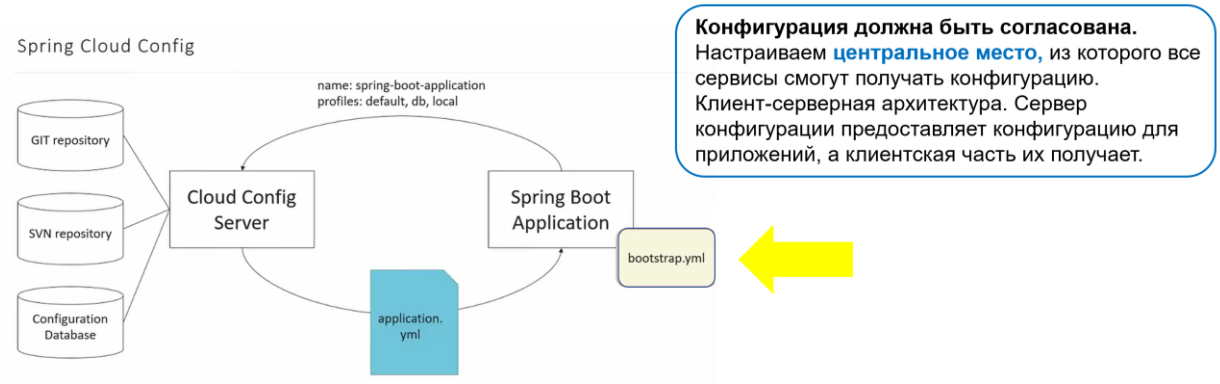
Допустим, у нас есть некоторое количество микросервисов, каждый из них требует проперти для подключения к базе данных: некоторые — для mongo, некоторые — для mysql, а другие — для postgreSql. Для этого нам нужно зайти в каждый микросервис и прописать для каждого свои проперти.

Если мы будем использовать config-server/config-client, нужда заходить в каждый сервис отпадает. Мы храним свои проперти в репозитории — локальном или удаленном (git) — и config-server позволяет нашим сервисам обращаться к нему при своем запуске, а он в свою очередь перенаправит их в репозиторий, где хранятся проперти для него.

Чем еще полезен Config Server?

На практике используется множество различных пропертей для различных нужд, например, мы хотим указать, какое количество раз за неделю будет отправляться рекламное письмо пользователям на почту. Со временем мы решили изменить это число.

В стандартном случае нам придется открывать код микросервиса, править его файл проперти и ре-деплоить приложение в облако/сервер. В случае с config-server мы просто зайдем в репозиторий, изменим там эту настройку и воспользуемся функциональностью спринга для того, чтобы без ре-деплоя в силу вступили актуальные проперти нашего сервиса.



Если конфигурация хранится в одном центральном месте, то ее можно быстро менять.

После перезагрузки источников свойств через вызов /actuator/refresh, в бинах с аннотацией **@RefreshScope** происходит обновление атрибутов с **@Value.**

Spring Cloud Discovery (служба обнаружения сервисов)

Eureka Server — это discovery сервис — приложение, которое содержит информацию о всех клиентских сервисах.

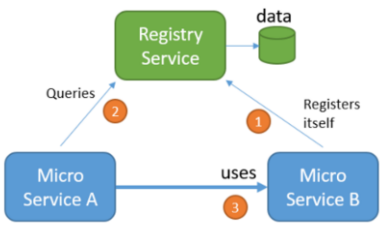
Каждый микросервис самостоятельно регистрируется на сервере Eureka, и Eureka знает все клиентские приложения, работающие на каждом порту и IP-адресе.

Eureka Server также известен как Discovery Server (сервер обнаружения). Если попытаться объяснить очень простыми словами, то это сервер имен или реестр сервисов. Его обязанность давать имена каждому микросервису. Он регистрирует микросервисы и отдает их ip другим микросервисам.

Таким образом, каждый сервис регистрируется в Eureka и отправляет эхо-запрос серверу Eureka, чтобы сообщить, что он активен. Для этого сервис должен быть помечен как **@EnableEurekaClient,** а сервер **@EnableEurekaServer.**

Аннотация **@EnableEurekaClient** сообщает платформе, что сервис является экземпляром какого-то микросервиса и просит зарегистрировать его на сервере Eureka. Также зарегистрированный микросервис может узнать другие службы, которые зарегистрированы в Eureka Server.

На рисунке слева можно увидеть общую схему, как это работает.



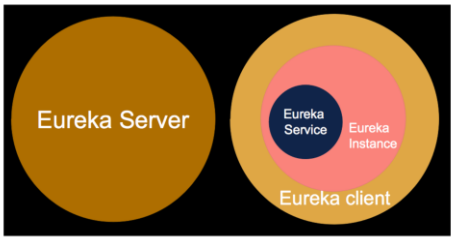


Рисунок справа иллюстрирует понятия, которые часто путают:

- **Eureka Server**: содержит реестр служб и REST API, которые можно использовать для регистрации службы, отмены регистрации службы и определения местоположения других служб.
- **Eureka Service**: любое приложение, которое можно найти в реестре служб Eureka Server и которое может быть обнаружено другими службами. Служба имеет определенный ID (его еще называют VIP) который может ссылаться на один или несколько экземпляров одного и того же приложения.
- **Eureka Instance:** любое приложение, которое регистрируется на Eureka Server для обнаружения другими.
- **Eureka Client:** он только запрашивает реестр служб у Eureka Server, чтобы определить запущенные экземпляры микросервисов.

Declarative REST Client (декларативный REST клиент)

Feign — это первый шаг в реализации архитектуры микросервиса посредством Netflix.

В реальности слабо связанных сервисов очень важно, чтобы общение между ними было легковесным и простым для отладки. Для упрощения связи по REST мы и используем Feign: при помощи него мы будем поглощать сообщения от других сервисов и автоматически превращать их в Java объекты.

Feign — это декларативный HTTP клиент, разработанный компанией Netflix. Основным преимуществом решения является то, что разработчику необходимо только описать интерфейс, в то время как реализация будет создана во время выполнения.

Feign использует интерфейсы, аннотированные **@FeignClient,** чтобы генерировать API запросы и мапить ответ на Java классы. Он шлет http запросы другим сервисам.

Его особенность в том, что нам не нужно знать, где и на каком порту находится какой-либо сервис.

Мы просто говорим нашему «сервис-1», чтобы он через Feign клиент пошел, например, к «сервис-2» и получил у него необходимые нам данные. Далее Feign обращается к Eureka Server и спрашивает, где находится «сервис-2».

Если «сервис-2» регистрировался в Eureka Server, то Eureka будет всё знать о «сервис-2»: где он находится, на каком порту, его URL и т. д. Остаётся только описать, как получить доступ к удаленной службе API, указав такие детали, как URL, тело запроса и ответа, принятые заголовки и т. д. Клиент Feign позаботится о деталях реализации.

Netflix предоставляет Feign в качестве абстракции для вызовов на основе REST, благодаря которым микросервисы могут связываться друг с другом, но разработчикам не нужно беспокоиться о внутренних деталях REST.

Давайте теперь рассмотрим аналог - **RestTemplate**.

RestTemplate - многоцелевой **HTTP** клиент, поддерживающий **http**-взаимодействия при помощи шаблонных методов (**get, post, put**).
Он поддерживает согласование содержимого, используя на стороне сервера ту же реализацию стратегии **HttpMessageConverter**, что и в **SpringMVC**.

Он преобразовывает объекты в тело **http-запроса**, а также **http-ответы** в объекты.
Для облегчения конфигурации в **SpringBoot** есть встроенный **RestTemplateBuilder**.

Аналоги Feign Client: WebClient

WebClient, представленный в **Spring 5**, является неблокирующим клиентом с поддержкой **Reactive Streams**. Он является частью **Spring 5** под названием **Spring WebFlux**.
Не смотря на то, что **Feign Client** теперь (с недавнего времени) тоже умеет работать с реактивностью (был сделан модель **ReactiveFeignClient**), но **WebClient** все равно является как-бы стандартом для реактивного подхода, т.к. был разработан гораздо раньше чем **ReactiveFeign**.
Для его использования надо включить модуль **spring-webflux** в проект.
Он был создан как часть модуля **Spring Web Reactive** и будет заменять классический **RestTemplate** в этих сценариях.
Он по протоколу HTTP/1.1.

Spring Cloud Circuit Breaker (предохранитель) см. также паттерн

Распределенные системы могут быть ненадежными. Запросы могут столкнуться с тайм-аутом или полностью завершиться ошибкой. Предохранитель может помочь смягчить эти проблемы, и Spring Cloud Circuit Breaker предлагает выбор из трех популярных вариантов: **Resilience4J, Sentinel или Hystrix.**

! В стеке Spring Cloud, а конкретно — в Feign Client и Eureka Server все максимально упрощено. **Из Spring Cloud были удалены или перемещены под капот ribbon и hystrix,** которые использовались в более ранних версиях для балансировки нагрузки, теперь балансировка работает «из коробки».

Spring Cloud Gateway (API шлюз) см. также паттерн

При таком большом количестве клиентов и серверов часто бывает полезно включить **API шлюз** в облачную архитектуру. Шлюз может позаботиться о защите и маршрутизации сообщений, сокрытии служб, регулировании нагрузки и многих других полезных вещах.

Spring Cloud Gateway предоставляет точный контроль над уровнем API, интегрируя решения Spring Cloud для обнаружения и балансировки нагрузки на стороне клиента, чтобы упростить настройку и обслуживание.

LoadBalancer (балансировщик нагрузки) см. также паттерн

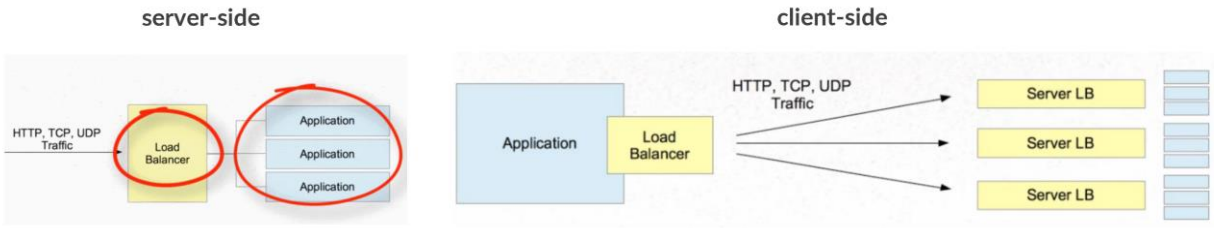
Сейчас недостаточно просто написать и соединить сервисы, нужно быть уверенным, что приложение работает стабильно и не дает никаких side-эффектов при возникновении проблем, например, при перегруженности одного или нескольких/микросервисов.

Допустим, один из сервисов не выдержал нагрузки и просто упал — теперь он полностью недоступен. В таких ситуациях на помощь приходит балансировка нагрузки и несколько инстансов одного и того же сервиса.

Балансировка нагрузки — это процесс распределения трафика между различными экземплярами одного и того же приложения.

Существует два вида балансировки:

server-side — когда запросы поступают от Client, они придут к балансировке нагрузки, и она определит один сервер для этого запроса. Самый простой алгоритм, используемый балансировкой нагрузки — это случайное распределение.



client-side — балансировка нагрузки находится на стороне клиента (Client side), она сама решает, к какому серверу отправить запрос на основании критериев, например, доступность, зона, быстродействие и т. д.

Spring Cloud Sleuth (трассировка)

Отладка распределенных приложений может быть сложной и занимать много времени. Для любого конкретного сбоя вам может потребоваться собрать воедино следы информации от нескольких независимых служб.

Spring Cloud Sleuth может настроить приложения предсказуемым и воспроизводимым образом. А при использовании вместе с **Zipkin** вы можете сосредоточиться на любых проблемах с задержкой, которые могут у вас возникнуть.

В микросервисах мы неизбежно сталкиваемся с большим количеством запросов как внешних, так и внутренних между сервисами. Но как же мониторить и понимать, на каком этапе произошел сбой?

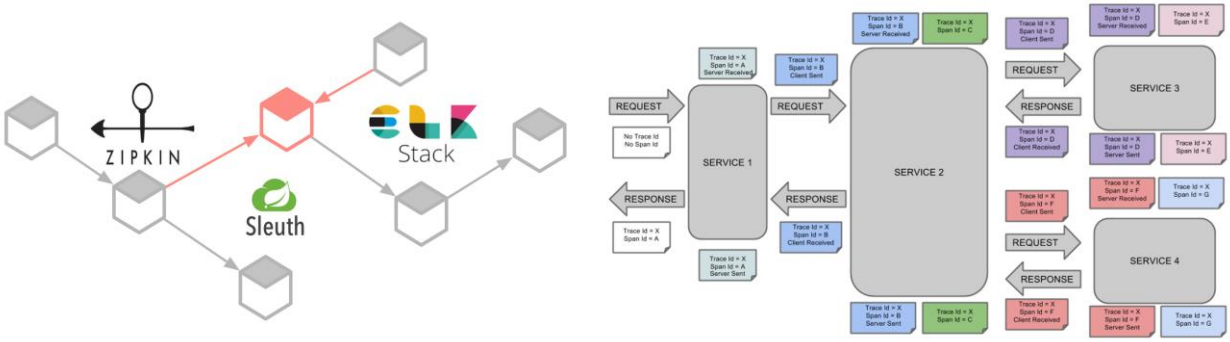
Для этого существуют различные механизмы **трассировки запросов** в микросервисной среде, один из них — Sleuth в сочетании с Zipkin.

Sleuth— механизм трассировки. Zipkin — журнал событий.

Под капотом Sleuth передает информацию в виде:[application name, traceId, spanId, export]

Рассмотрим каждый пункт:

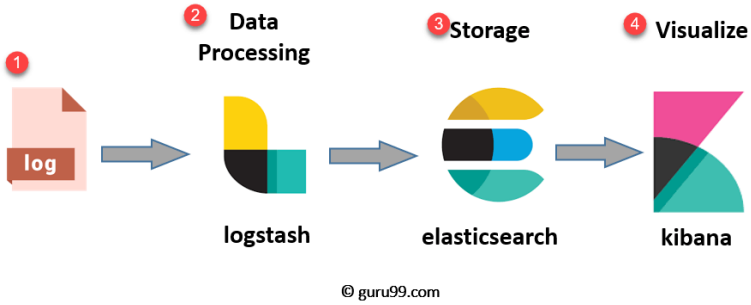
- **application name** — имя приложения
- **traceId** — ID, назначаемый каждому запросу
- **spanId** — используется для отслеживания работы. Каждый запрос может иметь множество шагов, которые будут вызываться по цепочке от одного сервиса к другому. Каждый этот шаг имеет свой уникальный spanId.
- **export** — это флаг, который указывает, следует ли экспортировать определенный журнал в инструмент агрегирования журналов, такой как Zipkin.



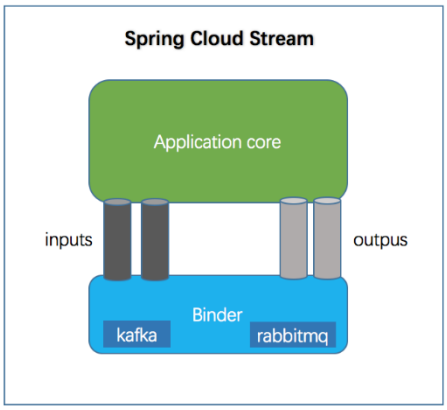
Весь сок состоит не в том, чтобы идентифицировать журналы в пределах одного сервиса, а в том, чтобы отслеживать цепочку запросов между несколькими сервисами. Именно параметр **traceId** — это то, что позволит отслеживать запрос при его переходе от одного сервиса к другому.

Если использовать Feign client от Netflix, информация трассировки также будет добавлена к этим запросам. Кроме того, сам Gateway также будет перенаправлять заголовки через прокси в другие сервисы.

Для мониторинга работы приложения используют ELK-стек (Elastic Search + Kibana).
В сочетании с Sleuth можно легко выполнять поиск по всем собранным журналам при помощи параметра **traceId** и видеть, как запрос передается от одного сервиса к другому по цепочке.



Spring Cloud Stream (стриминг сообщений)



Spring Cloud Stream обеспечивает легкую интеграцию с различными брокерами сообщений с минимальной конфигурацией. Он помогает в обмене сообщениями между двумя приложениями или микросервисами.

Также он легко интегрируется с такими известными и популярными брокерами как Kafka, RabbitMQ и другими.

По умолчанию для сервера используется порт 5672, а для UI дашборда 15672. Логин и пароль для UI по умолчанию: guest/guest.

Spring Cloud Stream – это более упрощенная связь между издателем и подписчиком, т.к. вся магия прячется под капотом и вам нужны минимальные настройки для того, что настроить брокер в вашей системе.

Давайте рассмотрим сразу небольшой пример: Издатель публикует сообщения в брокер и подписчик их читает. Издателю не нужно знать о подписчике. Все, что ему нужно — это подключиться к брокеру сообщений и опубликовать событие. Неважно, Kafka это или другой брокер, принцип работы с Spring Cloud Stream будет одинаковый.

Как могут общаться микросервисы

Микросервисы слабо связаны друг с другом. Чаще всего они общаются синхронно через HTTP или асинхронно с помощью очередей сообщений (RabbitMQ, Amazon SQS) или логстрима (Kafka, Amazon Kinesis).

А в чём проблема?
Мы разворачиваем сервисы в облаках. Как правило, это означает, что и общение микросервисов будет происходить по сети, а это тащит за собой сразу много проблем:

Проблема доступности. Мы не знаем, какой из микросервисов сейчас доступен для общения, а какой упал или потерял коннект.

Задержка передачи, потери, дублирование пакетов. Мы думаем, что отправили пакет, но получатель его не принял или получил в двойном экземпляре, и нам об этом неизвестно.

Хорошо иллюстрирует проблему так называемая задача двух генералов.

Представьте, что вы - генерал на поле боя. Второй генерал (союзник) отрезан от вас вражеской армией, но нужно координировать с ним начало атаки. Вы посылаете гонца. Он может спокойно доехать и передать сообщение или же погибнуть по дороге. Причем, его могут убить по пути обратно, так что он не сможет сообщить, что сообщение на самом деле доставлено. Можно ли в этой ситуации обеспечить себе уверенность в доставке или недоставке сообщения?

Нагрузка на трафик и память. Сервисы бывают нагруженные, поэтому общение приходится оптимизировать. От этого все становится сложнее. Если использовать какие-то асинхронные системы общения, придется хранить информацию какое-то время, а значит появляется вопрос к утилизации памяти или диска.

При этом нужно, чтобы общение было отказоустойчивым. Часто бывает, что сервисы падают каскадом - один упал, запросы не обрабатывает.

Вслед за ним валятся другие сервисы, которые его вызывали - все от того, что они не могут получить ответ на свой запрос. Так по цепочке происходит полный коллапс.

Синхронные способы общения: делаем вызов и ждем получения ответа.

Синхронный REST-like и аналоги. В чистом виде REST встречается редко, но в целом он один из самых популярных. При желании через костыли его можно сделать “асинхронным”, но этот случай мы тут не будем рассматривать.

gRPC - RPC на бинарном формате сообщений поверх HTTP/2 от Google.

SOAP - RPC с форматом XML. Это решение очень любили использовать в энтерпрайзе, оно чаще встречается в более старых системах.

Асинхронные способы общения: отправляем сообщение, а ответ придет когда-нибудь потом или он в принципе не предусмотрен.

Месседжинг - RabbitMQ, ZeroMQ, ActiveMQ. Они все разные.

Стриминг - Kafka. В принципе, Kafka похожа на мессенджинговую платформу, но отличия все-таки есть.

Источник: https://habr.com/ru/company/maxilect/blog/677128/

Какие бывают типы интеграции между сервисами?

Два микросервиса могут общаться через REST или брокеры сообщений.

Apache Kafka является брокером сообщений. С его помощью микросервисы могут взаимодействовать друг с другом, посылая и получая важную информацию. Возникает вопрос, почему не использовать для этих целей обычный POST – request, в теле которого можно передать нужные данные и таким же образом получить ответ? У такого подхода есть ряд очевидных минусов.

Например, **продюсер** (сервис, отправляющий сообщение) может отправить данные только в виде response’а в ответ на запрос консьюмера (сервиса, получающего данные).

Допустим, консьюмер отправляет POST – запрос, и продюсер отвечает на него. В это время консьюмер по каким-то причинам не может принять полученный ответ. Что будет с данными? **Они будут потеряны.**

Консьюмеру снова придётся отправлять запрос и надеяться, что данные, которые он хотел получить, за это время не изменились, и продюсер всё ещё готов принять request.

Apache Kafka решает эту и многие другие проблемы, возникающие при обмене сообщениями между микросервисами. Не лишним будет напомнить, что бесперебойный и удобный обмен данными – одна из ключевых проблем, которую необходимо решить для обеспечения устойчивой работы микросервисной архитектуры.

Что такое Kafka, назови основные элементы

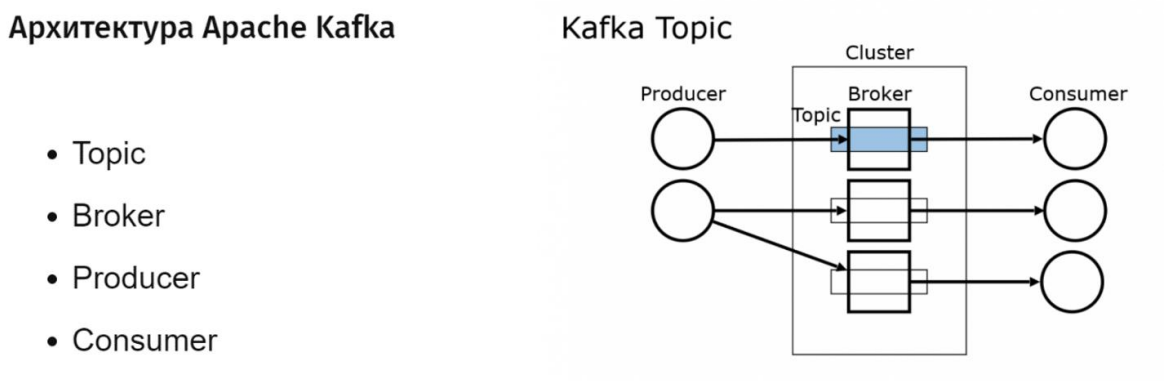
Смотреть видео «Про Kafka (основы)» https://www.youtube.com/watch?v=-AZOi3kP9Js

Читать тут https://habr.com/ru/post/537846/

Apache Kafka – это распределенный брокер сообщений.
Message = (key (набор байтов), value (м.б. любое)).

У Kafka много компонентов. Первый из них – это producer. Producer – это тот, кто создает кучу сообщений и отправляет их куда-то. На другом конце находится Consumer.

Они употребляют эти данные, что-то с ними делают. Передача осуществляется через кластер. Получается, что в кластере есть куча брокеров и передача сообщений осуществляется через них. И брокер выступает звеном, который позволяет от producer -а к consumer -у не напрямую данные передавать, а через такой топик.



Топик – это логическая единица, которая связывает между собой producers, consumers. И есть какое-то физическое хранение. Каждый такой топик – это множество партиций (в них записано сообщение).

Сообщения всегда пишутся в конец партиции. Писать можно параллельно в несколько партиций одновременно. Они между собой никак не связаны, физически это разные вещи. У каждого сообщения есть свой номер, свой offset. Каждая партиция начинается с номера 0. Но партиция может стать очень большой (весить десятки, сотни гигабайт). Поэтому каждая партиция делится на сегменты.

Базовый **offset** — это тот, с которого начинается каждый из сегментов.

Последний **сегмент** еще по-другому называется активным сегментом. Это тот сегмент, в который можно писать данные. Благодаря этому сохраняется правило, что пишем всегда в конец партиции. Три файла, которые лежат в файловой системе: data, index, timeindex.

segment = (**base_offset**, data, index, timeindex)

```
00000000001234567890.log
00000000001234567890.index
00000000001234567890.timeindex
```

Data – это данные, т. е. есть какой-то здоровый файл. И в нем лежат сообщения (могут быть разного размера). Kafka все равно, складывайте, что хотите в нее.

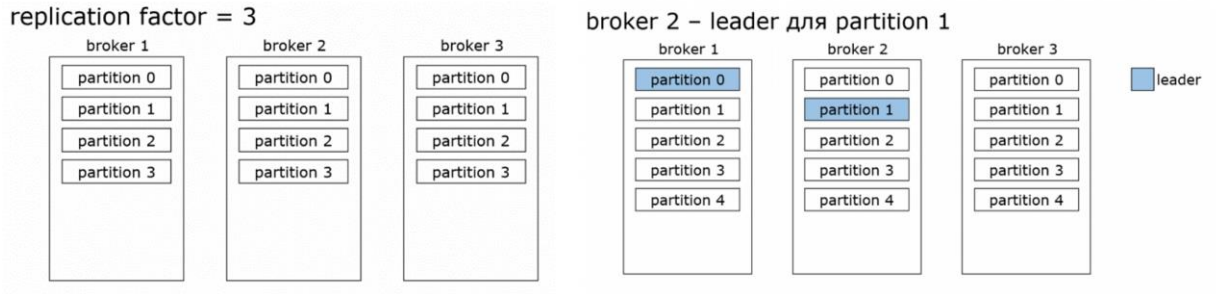
Что такое index? Если мы хотим найти какое-то сообщение по смещению, то надо уметь это делать быстро. Как раз **index** позволяет это делать.

Похожим образом устроен **timeindex.** Для чего он нужен? Каждое сообщение в Kafka имеет какую-то метку времени. И неплохо бы уметь искать по этой ветке времени. Timeindex решает эту задачу.

Что такое кластер, топик, партиция, репликация, зукипер, продюсер и консьюмер

Кластер – это множество брокеров. Один из них отвечает за контроллер. Он координирует работу кластера.

У нас есть топик, который состоит из нескольких партиций. В данном случае мы создали топик на кластере из трех брокеров. И партиции распределились по кластеру. При этом Kafka надежная, у нее есть **replication factor** для каждого топика. Допустим, этот фактор равен трем. Это означает, что каждая партиция должна иметь три копии (реплики) У нас три копии, поэтому на каждом брокере будет по копии. При этом Kafka позволяет добавлять новые партиции к топику.



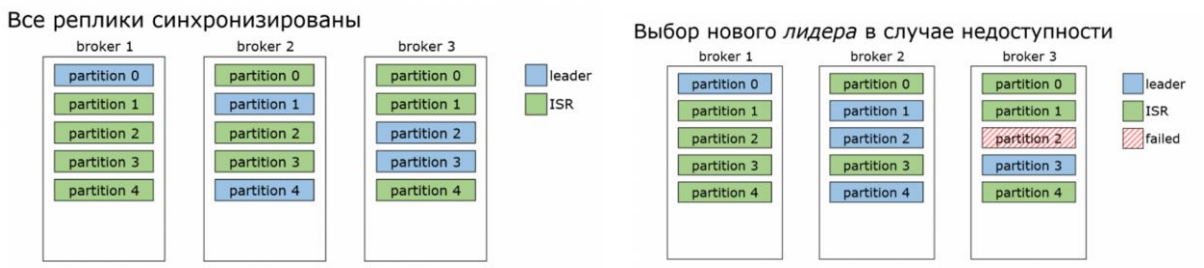
Поэтому, если данных станет много, можно увеличить **единицу параллелизации.**

Партиция – это единица параллелизации, поэтому можно добавить еще одну. И там producers стало больше, которые могут писать с той же эффективностью.

Лидер – это тот брокер, который отвечает за запись в конкретную партицию.
Не может быть у одной партиции несколько лидеров. Иначе каждый из них что-то написал бы свое, а потом это никак нельзя было бы соединить в какую-то одну последовательную партицию. И каждый брокер может стать лидером у некоторых партиций.

Kafka старается это равномерно распределять по кластеру и **балансировать,** чтобы не получилось так, что один брокер пыхтит и в себя все пишет-пишет, а остальные отдыхают.

Данные, которые пишутся в лидера, должны быть распределены по **репликам.** Те, кто в себя тащат данные из лидера, называются **follower.** Они следуют за ним и потихоньку подкачивают к себе данные. После того, как они сохранили в себе все данные, они становятся крутыми, они становятся **in sync replica.** Это реплика, которая *синхронизирована* с лидером.



И в идеале у нас весь кластер должен быть синхронизирован, т. е. все реплики должны быть в списке in sync replica (рис. слева).

У нас был лидер в брокер 3 на партиции 2, а потом пропал (рис. справа). В этот момент мы данные писать не можем, потому что лидера нет. Kafka не теряется, она выбирает нового лидера. И за это отвечает контроллер. Все, отлично. Теперь с него можно

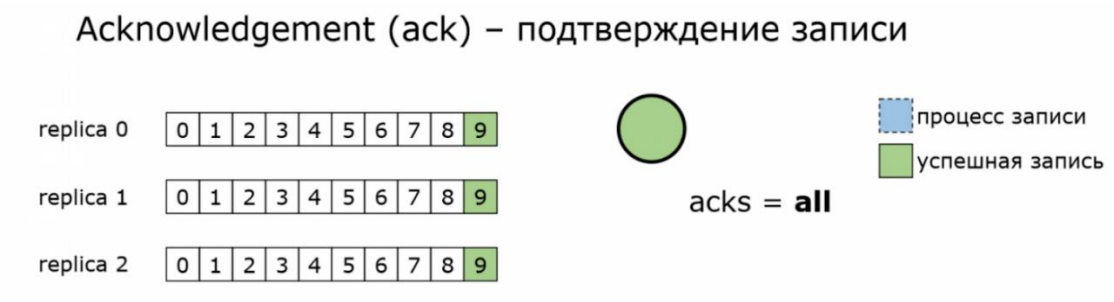
реплицировать данные по другим брокерам. Но в партиции старый лидер может ожить. Ожил, у него все хорошо, но он успел отстать по данным, потому что лидерство изменилось, появились новые данные. И поэтому он вынужден уже реплицироваться с нового лидера. Он это сделал. Кластер восстановился. И все хорошо.

И потом в Kafka случается магия, которая – раз и возвращает лидера обратно.
Для чего это сделано? Чтобы лидерство не выпадало на одного брокера (один работает, другие отдыхают), Kafka периодически умеет это дело перебалансировать. И у нее есть куча настроек, которые это регламентируют.

Продюсер должен всегда писать в лидера. Соответственно, он пишет в конкретного брокера, в конкретную партицию. Это важно.

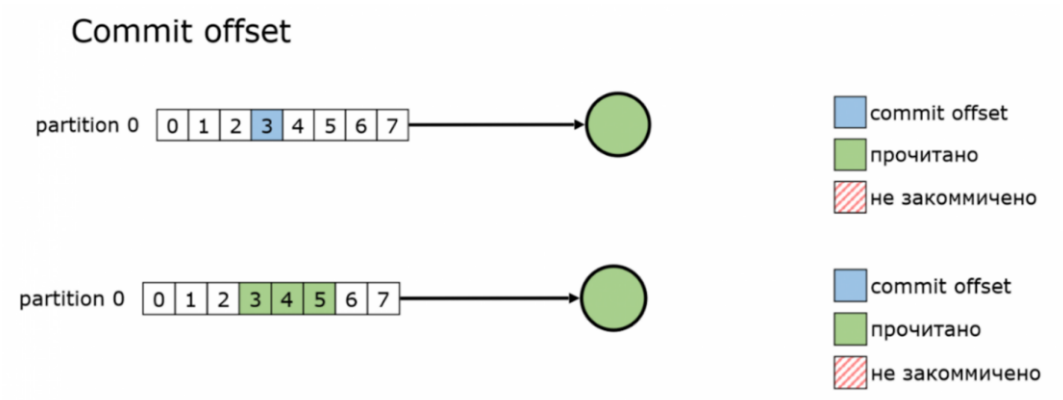
Есть такая штука в Kafka, как **acknowledgement,** т. е. подтверждение записи.
Продюсер должен убедиться, что данные записались.

- Нулевой уровень предоставляет нулевые гарантии (тут может быть потеря данных).
- Если поставим уровень 1, то продюсер записал на брокера. Брокер вернул подтверждение, что сохранил (продюсер считает, что все ок). Followers приходят к лидеру и говорят: «У тебя есть новые данные?». Он говорит: «Да». И они это забирают. А могли и не забрать. Это важно.
- Максимальный уровень all (рис).



Есть еще важная настройка min.insync.replicas.
Эту настройку можно понизить, тогда можно дожидаться не всех, а, допустим, все-1.

У нас также есть **consumer,** он читает.
Допустим, прочитал несколько сообщений. И то значение, с которого надо начинать дальше, он коммитит. Он говорит Kafka: «Я дочитал до вот этого и с этого места я хочу потом продолжить». Соответственно, он потом продолжает и читает. Поэтому в Кафку добавили такую фичу, как **commit offset.**



Настройки у брокера, у consumer должны быть. Но они должны не просто быть, а еще должны быть согласованы.

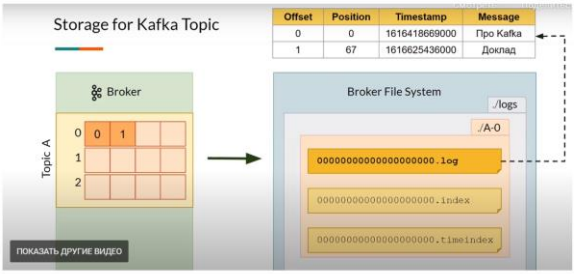
Дополнительно

ПОД КАПОТОМ:

У брокера есть файловая система, в ней есть папка .logs и в этой папке ./logs есть папки под каждую партицию. Каждая такая папка содержит три файла:

- .logs – содержит сами сообщения
- .index – маппинг offset – позиция
- .timeindex – маппинг timestamp на offset

Offset – это номер сообщения в партиции



MurmurHash % кол-во партиций = № брокера (остаток от деления, как в HashMap почти)

Кафка по ключу считает хэш. Далее, зная, сколько партиций у нашего топика, она определяет номер партиции, куда нужно положить эти данные.

Если ключа нет (Key = null): используется round_Robin – это когда кафка кладет сообщение в одну из партиций топика.

Если пришло следующее сообщение этого топика, она просто кладет сообщение в следующую партицию.

Важно: в отсутствии ключа именно продюсер решает, в какую партицию положить сообщение, а не брокер.

Что делает кафку быстрой? Zero-copy magic

1) За счет единого формата данных (в каком формате продюсер передал, в таком консьюмер и прочитал)
2) Чтение и передача непрерывного куска лога. Т.е. читаем куском (batch или пачка) и этим же куском отдаем данные.
   Причем данные от продюсера сначала сжимаются (сжатая пачка хранится в виде Wrapper message), и консьюмер их в полном объему может считатть

Что такое ZooKeeper

Apache ZooKeeper — это распределенный сервис конфигурирования и синхронизации, открытая программная служба для координации распределенных систем, организованная на основе резидентной базы данных категории «ключ — значение». Изначально входила в экосистему Hadoop, впоследствии стала проектом верхнего уровня Apache Software Foundation.

Для работы кафка нужен Zookeeper. Метаданные хранятся в Zookeeper.

- он отвечает за выбор контроллера. Контроллер решает, кто из брокеров будет лидером какой партиции
- в нем хранится конфигурация для топиков (не дефолтные)
- хранит информацию где какие реплики на каких нодах находятся
- хранит claster state – это кол-во брокеров, которые сейчас находятся online.

Брокеры подключаются к зукиперу и говорят: вот у меня такой-то номер, у меня все хорошо.

Это позволяет контролировать какая часть кластера работает, а какая нет

Зукипер можно использовать в роли K/V хранилища. Обычно это горячий кеш. Но лучше посмотреть в сторону более специализированного софта. Redis/Tarantul удобнее для использования в этой роли и более эффективно утилизируют железо при чистой K/V нагрузке.

У Зукипера есть архитектурная проблема - zxcid. Это внутренний 32 битный счетчик операций Зукипера. Когда он переполняется кластер разваливается на время единиц секунд до десятков минут. Надо быть к этому готовым и мониторить текущее значение zxcid. Хорошее решение будет в версии 3.8.0

Всегда ли нужен ZooKeeper

В KIP-833 сообщается, что в версии Kafka 3.3 будет используется протокол согласования метаданных KRaft, работающий внутри Kafka без Zookeeper, который будет признан Production-Ready и далее постепенно зависимость от Zookeeper будет помечена как deprecated и удалена.

Статья об особенностях протокола Kraft, разбирается настройка кластера Kafka без необходимости установки Zookeeper. https://habr.com/ru/company/otus/blog/670440/

Как обеспечивается FIFO

Порядок FIFO обеспечивается по партиции!

Kafka гарантирует упорядочение сообщений, все сообщения будут упорядочены точно в том порядке, в котором они пришли.

Можно ли удалять сообщения из Kafka?

Kafka хранит прочитанные сообщения в течение определенного периода времени (не удаляет сообщения после прочтения — по умолчанию одна неделя).

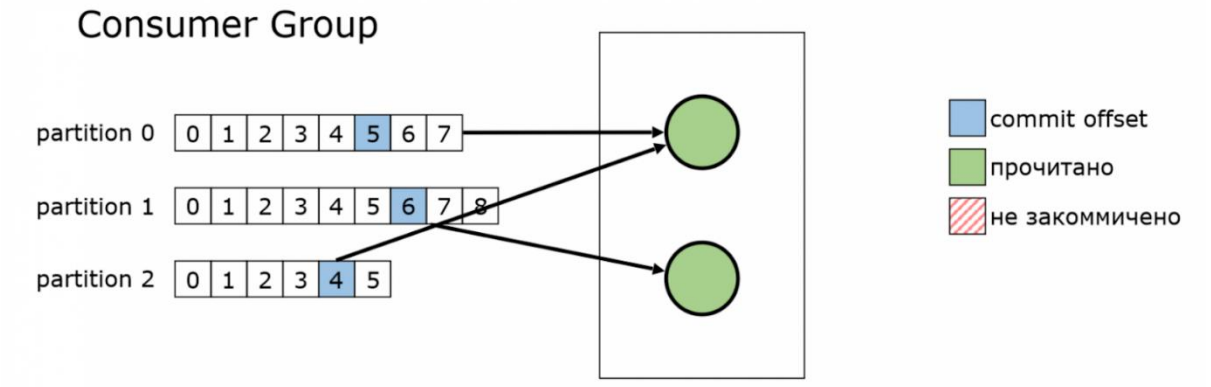
При заданных свойствах срока хранения сообщения имеют TTL (время жизни). По истечении срока действия сообщения помечаются для удаления, тем самым освобождая место на диске.

Как вычитывают сообщения консьюмер групп?

В какой-то момент мы начинаем писать много данных и нам надо уметь масштабироваться (консьюмер не справляется с чтением всех данных).

Можно добавить потребителя и объединить набор consumers в группу. И у них там появляется общий идентификатор. Кафка между ними автоматически расбалансирует все партиции, и они начнут читать. Если один отвалится, то единственный живой consumer подхватит эту партицию (читает и commit - ит).

А потом consumer ожил, все у него хорошо. Он снова присоединяется к группе.



Отличие кафки от rabbitMQ

Основная разница между Apache Kafka и RabbitMQ в способе доставки сообщений. Кролик использует механизм "проталкивания" (push) сообщений клиентам. Можно представить как трубу, из которой идет вода.

Кафка использует механизм "выгрузки" (pull) сообщений клиентам.
Можно представить конвейер с коробками.

Кролик будет пропихивать сообщения клиенту. А клиент может не справляться с таким потоком данных и под нагрузкой может просто не выдержать и «упасть».
Чтобы управиться, нужно будет поднимать дополнительные клиенты.

В случае с Кафкой, клиент сам по мере своих доступных пропускных способностей будет забирать сообщения из "трубы", не падая от напора данных.

Кроме этого, есть ещё одна важная отличительная черта. Когда Кролик пропихнул клиенту сообщение, последнее **удаляется** из "трубы". И если клиент вдруг во время обработки «отвалился», то сообщение, которое он обрабатывал, будет потеряно.

В случае с Кафкой, сообщения продолжают храниться в "трубе", а изменяется только отступ (offset). Теперь, если клиент падает, не дообработав сообщение, другой клиент может это же сообщение подхватить из "трубы" и попробовать его обработать.

RabbitMQ:

- Издатель публикует сообщение в RabbitMQ в Exchange.
- Exchange принимает сообщение и направляет его в одну или несколько очередей при помощи привязки (Binding).
- RabbitMQ шлет подтверждение издателю, когда получил сообщение.
- Подписчик поддерживает TCP соединение с RabbitMQ.
- RabbitMQ отправляет сообщение подписчику.
- Подписчик отправляет подтверждение success/error, что получил сообщение.
- Сообщение удаляется из очереди.

Kafka наиболее масштабируема, чем кролик:

- Издатель отправляет сообщение брокеру сообщений (Kafka).
- Сообщения сохраняются в теме.
- Подписчики подписываются на тему для получения новых сообщений.

В Kafka легко добавить еще одного брокера в систему, потому что брокер Kafka — это кластер. В RabbitMQ это сделать сложнее.

Но главное отличие в том, что темы с сообщениями можно разделить на разделы и распределить внутри кластера (внутри брокера Kafka) и сделать реплику. Несколько брокеров могут обслуживать одну тему. Если один брокер умрет, данные не будут потеряны. Kafka может разбивать очереди на разделы и распределять их по кластеру.

И еще одна важная особенность — **Kafka гарантирует упорядочение сообщений,** все сообщения будут упорядочены точно в том порядке, в котором они пришли. Также Kafka хранит прочитанные сообщения в течение определенного периода времени (не удаляет сообщения после прочтения — по умолчанию одна неделя).

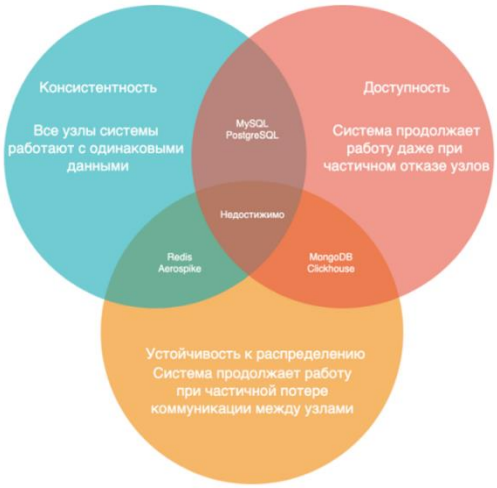
CAP теорема

Эрик Брюер (Eric Brewer) сформулировал **CAP-теорему,** которая гласит, что система может обладать лишь двумя из следующих трех свойств:

- Согласованность
- Доступность
- устойчивость к разделению

В наши дни архитекторы отдают предпочтение **доступным системам,** жертвуя согласованностью.

Это эвристическое утверждение о том, что в любой реализации распределённых вычислений возможно обеспечить не более двух из трёх следующих свойств:



согласованность данных (англ. consistency) — во всех вычислительных узлах в один момент времени данные не противоречат друг другу;

доступность (англ. availability) — любой запрос к распределённой системе завершается корректным откликом, однако без гарантии, что ответы всех узлов системы совпадают;

устойчивость к разделению (англ. partition tolerance) — расщепление распределённой системы на несколько изолированных секций не приводит к некорректности отклика от каждой из секций.

Принципы код-ревью

https://habr.com/ru/post/592071/

Как вы развертываете свои микросервисы Java?

Мы используем Docker и Kubernetes для развертывания наших микросервисов в облаке. Docker используется для создания образа Docker всего сервиса, а затем Kubernetes для его развертывания на AWS или Azure. Сервис управляется K8, поэтому он позаботится о запуске остановленных экземпляров и увеличении их количества при увеличении нагрузки.