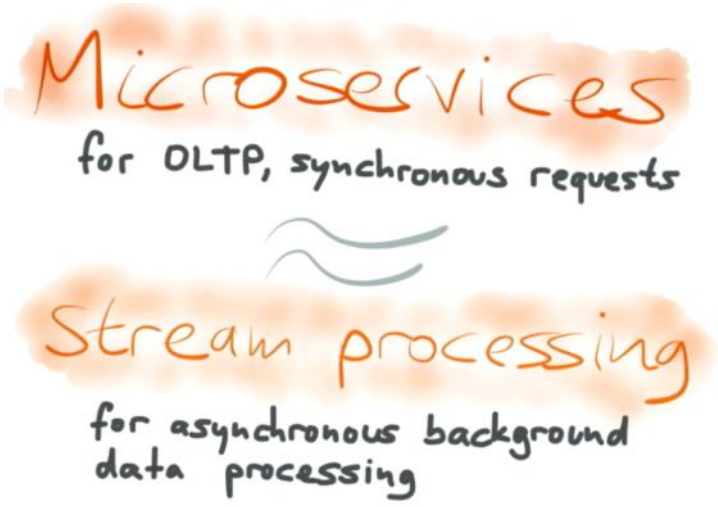
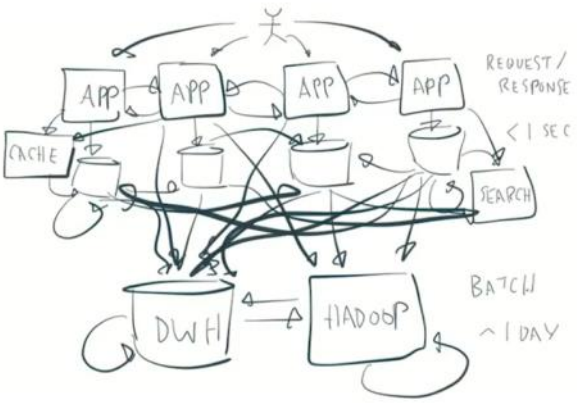
EVENT DRIVEN vs REST

29 декабря 2020 г. 17:50

микросервисы противопоставляются потоковой обработке

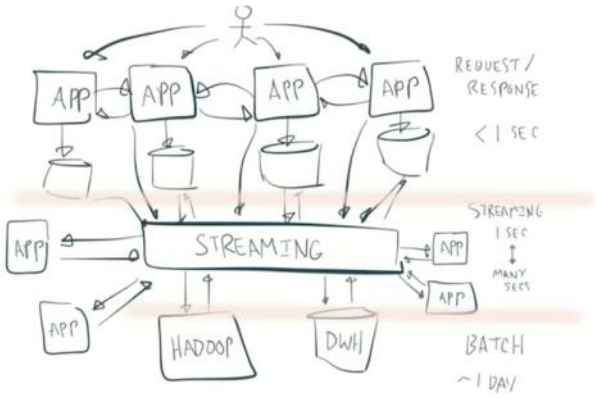


microservices

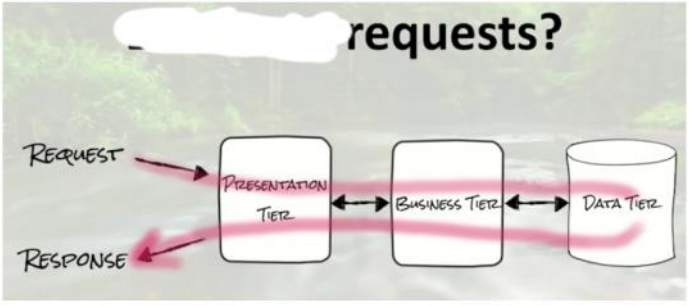


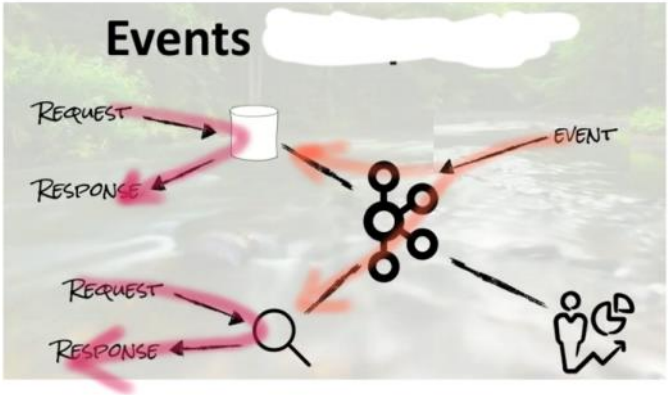
обычная трехзвенка

stream processing



stream processing





REST концептуально, на самом деле, тот же самый RPC

- а также наследует все те же недостатки

RPC/REST in JavaScript

```
let args = {amount: 3.99, currency: 'GBP', /*...*/};
let request = {
  method:  'POST',
  body:    JSON.stringify(args),
  headers: {'Content-Type': 'application/json'}
};

fetch('https://example.com/payments', request)
  .then((response) => {
    if (response.ok) success(response.json());
    else failure(response.status); // server error
  })
  .catch((error) => {
    failure(error); // network error
  });
```

недостатки HTTP (как альтернативный вариант не только KAFKA но и gRPC)

см также * synchronous microservices

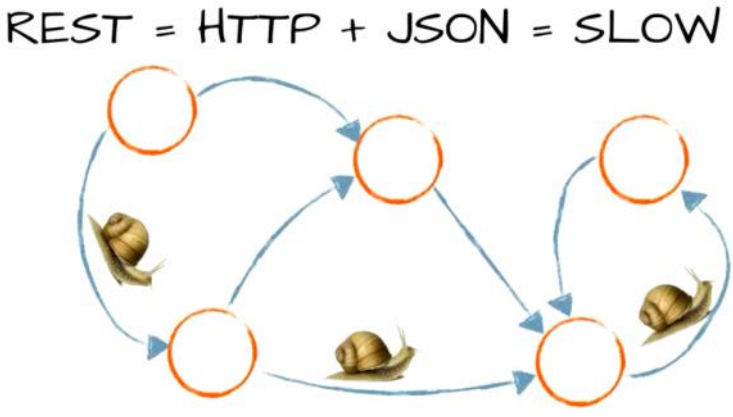
rest - - это термин, обозначающий фундаментальные подходы Всемирной паутины (WWW)

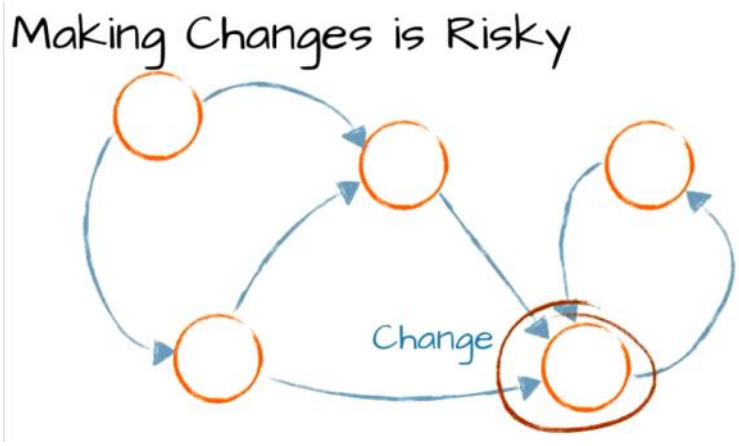
- REST описывает архитектуру WWW и, следовательно, крупнейшую в мире интегрированную компьютерную систему.
- Микросервисы должны иметь возможность вызывать друг друга, чтобы совместно реализовывать логику.Это может поддерживаться разными технологиями.
  REST (передача репрезентативного состояния) - это один из способов обеспечения связи между микросервисами. REST - это термин, обозначающий фундаментальные подходы Всемирной паутины (WWW)

Реализация REST с HTTP называется RESTful HTTP

- Однако REST также может быть реализован с другими протоколами
- Когда службы RESTful HTTP обмениваются данными с использованием JSON или XML вместо HTML, они могут обмениваться данными, а не только получать доступ к веб-страницам.

кафка в 5 раз быстрее





в кафке решается через broadcast сообщений без указания ваного адресата

- здесь показана проблема что в REST-микросервисах сложно подключать новые микросервисы

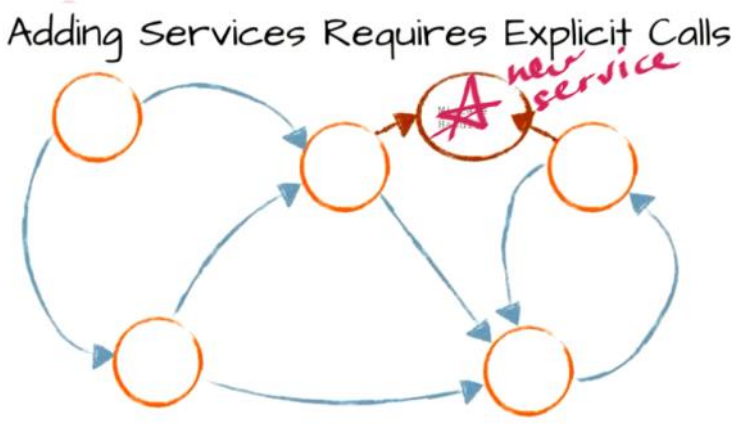
BENEFITS AND DRAWBACKS OF REST

There are numerous benefits to using REST:

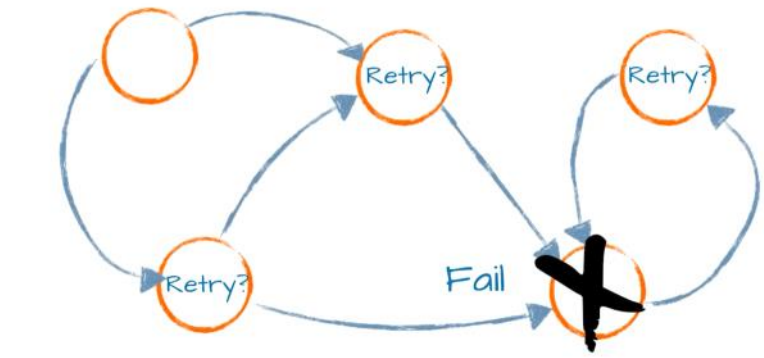
- It's simple and familiar.
- You can test an HTTP API from within a browser using, for example, the Postman plugin, or from the command line using curl (assuming JSON or some other text format is used).
- It directly supports request/response style communication.
- HTTP is, of course, firewall friendly.
- It doesn't require an intermediate broker, which simplifies the system's architecture.

There are some drawbacks to using REST:

- It only supports the request/response style of communication.
- Reduced availability. Because the client and service communicate directly without an intermediary to buffer messages, they must both be running for the duration of the exchange.
- Clients must know the locations (URLs) of the service instances(s). As described in section 3.2.4, this is a nontrivial problem in a modern application. Clients must use what is known as a *service discovery mechanism* to locate service instances.
- Fetching multiple resources in a single request is challenging.
- It's sometimes difficult to map multiple update operations to HTTP verbs.



в кафке сообщения еще и персистятся (но также есть ретрай и циклы опроса на клиентах)



REST упрощает поиск

С другой стороны, службы данных с интерфейсами на основе HTTP упрощают поиск. Кто угодно может войти и запустить запрос. Но они не упрощают перемещение данных. Чтобы извлечь набор данных, вы выполняете запрос, а затем периодически опрашиваете службу на предмет изменений. Это своего рода взлом, и обычно операторы, отвечающие за сервис, который вы опрашиваете, не благодарит вас за это.

Важно понимать, что REST чаще всего является синхронным (даже не псевдоасинхронным в виде двух синхронных запросов)

- , что делает его очень неподходящим протоколом по умолчанию для межсервисного взаимодействия. REST может быть разумным вариантом, когда существует только несколько служб или в ситуациях между конкретными тесно связанными службами
- Необходимость в асинхронной передаче сообщений включает не только ответы на отдельные сообщения или запросы, но и непрерывные потоки сообщений, потенциально неограниченные потоки. За последние несколько лет ландшафт потоковой передачи резко вырос как с точки зрения продуктов, так и с точки зрения того, что на самом деле означает потоковая передача. 1
- REST — это чрезвычайно популярный механизм IPC. У вас может возникнуть соблазн использовать его для межсервисного взаимодействия. Но проблема REST заключается в том, что это синхронный протокол: HTTP-клиенту приходится ждать, пока сервис не вернет ответ. Каждый раз, когда сервисы общаются между собой по синхронному протоколу, это снижает доступность приложения.

  Order Service retrieves consumer information by making an HTTP GET /consumers/id request to the Consumer Service.

  - RESTful HTTP является синхронным: обычно служба отправляет запрос и ожидает ответа, который затем анализируется, чтобы продолжить выполнение программы. Это может вызвать проблемы, если в сети существует большое время задержки. Это может увеличить продолжительность обработки запроса, так как нужно ждать ответов от других служб. После ожидания в течение определенного периода времени запрос должен быть прерван, поскольку вполне вероятно, что на запрос вообще не будет ответа.

проблемы в амазон до AWS

https://gist.github.com/chitchcock/1281611

- Эскалация пейджера становится намного сложнее, потому что билет может пройти 20 сервисных вызовов до того, как будет идентифицирован настоящий владелец. Если каждый отказ проходит через команду с 15 минутным временем ответа, могут пройти часы, прежде чем нужная команда наконец узнает, если только вы не создадите много строительных лесов, показателей и отчетности. pager escalation gets way harder, because a ticket might bounce through 20 service calls before the real owner is identified. If each bounce goes through a team with a 15-minute response time, it can be hours before the right team finally finds out, unless you build a lot of scaffolding and metrics and reporting.

- каждая из ваших коллег внезапно становится потенциальным взломщиком DOS. Никто не сможет добиться реального прогресса, пока не будут введены очень серьезные квоты и ограничения для каждой службы. every single one of your peer teams suddenly becomes a potential DOS attacker. Nobody can make any real forward progress until very serious quotas and throttling are put in place in every single service.

- мониторинг и QA - это одно и то же. Вы бы никогда так не подумали, пока не попробуете создать большую SOA. Но когда ваша служба говорит: «О да, я в порядке», вполне возможно, что единственное, что все еще работает на сервере, - это маленький компонент, который знает, как сказать «Я в порядке, Роджер Роджер, снова и снова». Out "веселым голосом дроида. Чтобы узнать, отвечает ли служба на самом деле, необходимо совершать отдельные звонки. Проблема продолжается рекурсивно до тех пор, пока ваш мониторинг не выполнит всестороннюю проверку семантики всего спектра ваших услуг и данных, после чего он станет неотличим от автоматического контроля качества. Так что они континуум. monitoring and QA are the same thing. You'd never think so until you try doing a big SOA. But when your service says "oh yes, I'm fine", it may well be the case that the only thing still functioning in the server is the little component that knows how to say "I'm fine, roger roger, over and out" in a cheery droid voice. In order to tell whether the service is actually responding, you have to make individual calls. The problem continues recursively until your monitoring is doing comprehensive semantics checking of your entire range of services and data, at which point it's indistinguishable from automated QA. So they're a continuum.

- если у вас есть сотни сервисов, и ваш код ДОЛЖЕН связываться с кодом других групп через эти сервисы, то вы не сможете найти ни одну из них без механизма обнаружения сервисов. И вы не можете этого добиться без механизма регистрации службы, который сам по себе является другой службой. Таким образом, у Amazon есть универсальный реестр услуг, в котором вы можете рефлексивно (программно) узнать о каждой службе, о ее API, а также о том, работает ли она в настоящее время и где. if you have hundreds of services, and your code MUST communicate with other groups' code via these services, then you won't be able to find any of them without a service-discovery mechanism. And you can't have that without a service registration mechanism, which itself is another service. So Amazon has a universal service registry where you can find out reflectively (programmatically) about every service, what its APIs are, and also whether it is currently up, and where

- Отладка проблем с чужим кодом становится НАМНОГО сложнее и в принципе невозможна, если нет универсального стандартного способа запуска каждой службы в отлаживаемой песочнице. debugging problems with someone else's code gets a LOT harder, and is basically impossible unless there is a universal standard way to run every service in a debuggable sandbox.

REST без центрального брокера, а КАФКА с брокером

Брокер - это отдельная часть инфраструктуры, которая передает сообщения любым программам, которые в них заинтересованы, а также хранит их столько времени, сколько необходимо. Так что он идеально подходит для потоковой передачи или обмена сообщениями типа «запустил и забыл».

ESB любит command-query для взаимодействия с системами
а КАФКА любит события и подписку на них систем

ESB ориентированы на интеграцию устаревших и готовых систем, используя эфемерный и сравнительно низкопроизводительный уровень обмена сообщениями, который поддерживает протоколы запрос-ответ (см. Предыдущий раздел).

В некоторых кругах ESB критикуют . Эта критика возникает из-за того, как технология создавалась за последние 15 лет, особенно там, где ESB контролируются центральными командами, которые диктуют схемы, потоки сообщений, проверку и даже преобразование. На практике централизованные подходы, подобные этому, могут ограничивать организацию, затрудняя развитие отдельных приложений и служб в их собственном темпе.

сложно изучать трассировку инстанса бизнес-процесса

Наши операционные инженеры часто оказываются неохотными детективами, разыгрывая распределенные тайны убийств, отчаянно бегая от службы к службе, собирая по кусочкам фрагменты вторичной информации. (Кто что сказал, кому и когда?)

(система должна быть ориентирована на программистов, чтобы им было легко)

Для этого тоже есть очень веская причина. Когда мы проектируем системы в масштабах компании, эти системы больше ориентированы на людей, чем на программное обеспечение.

HTTP реализует протокол без сохранения состояния.

RESTful HTTP-интерфейс может быть очень легко дополнен кешем

- Поскольку RESTful HTTP использует тот же протокол HTTP, что и Интернет, достаточно простого веб-кеша.
- Точно так же стандартный балансировщик нагрузки HTTP также может использоваться для RESTful HTTP.
- Такой размер возможен только из-за свойств HTTP. HTTP, например, обладает простыми и полезными механизмами безопасности - не только шифрованием через HTTPS, но и аутентификацией с помощью заголовков HTTP.

-

HATEOAS (Hypermedia as the Engine of Application State) - еще один важный компонент REST

- Это позволяет моделировать отношения между ресурсами с помощью ссылок. Следовательно, клиенту нужно знать только точку входа, и оттуда он может продолжать навигацию по своему желанию и находить все данные поэтапно.
- HATEOAS не имеет централизованной координации, только ссылки. Это очень хорошо согласуется с концепцией, согласно которой микросервисы должны иметь как можно меньшую централизованную координацию. Клиенты REST должны знать только точки входа, на основе которых они могут обнаружить всю систему. Следовательно, в архитектуре на основе REST сервисы можно перемещать прозрачным для клиента способом. Клиент просто получает новые ссылки. Для этого не требуется централизованная координация- служба REST просто должна возвращать разные ссылки.
- HATEOAS - это концепция, а HAL (Hypertext Application Language)- способ ее реализовать. Это стандарт для описания того, как ссылки на другие документы должны содержаться в документе JSON

Однако HATEOAS дает свободу перемещать ресурсы и устраняет необходимость в маршрутизации.

- тк маршрутизировать теперь будет сам микросервис
- Микросервисы доступны извне через URL-адреса, но их можно переместить в любое время. В конце концов, HATEOAS динамически определяет URL-адреса.
- HATEOAS означает, что отношения между системами представлены как связи. Клиенту необходимо знать только входной URL. Все остальные URL-адреса могут быть изменены, поскольку клиенты не обращаются к ним напрямую, а находят их по ссылкам, начинающимся с URL-адреса входа.

Экспериментирование

- https://www.thoughtworks.com/insights/blog/microservices-evolutionary-architecture
- Экспериментирование - это одна из суперспособностей, которые эволюционная архитектура предоставляет бизнесу. Недорогое в эксплуатации тривиальное изменение приложений позволяет использовать стандартные методы непрерывной доставки, такие как A / B-тестирование, Canary Releases и другие. Часто архитектуры микросервисов проектируются на основе маршрутизации между сервисами для определения приложений, что позволяет нескольким версиям конкретного сервиса существовать в экосистеме. Это, в свою очередь, позволяет экспериментировать и постепенно заменять существующие функции. В конечном итоге эта возможность позволяет вашему бизнесу тратить меньше времени на размышления о незавершенных историях и вместо этого заниматься разработкой, основанной на гипотезах.

фитнес-функция - для каждой системы выбирается свой профиль фитнес функций

- Радарная диаграмма, используемая для выделения важных фитнес-функций, соответствующих этой программной системе.]
- Предварительное размышление о том, какой должна быть функция приспособленности для конкретной системы, дает руководство для принятия решений и сроков принятия решений. Архитектурные решения оцениваются относительно функции пригодности, чтобы мы могли видеть, что архитектура развивается в правильном направлении.

то что приносит боль нужно делать чаще и больше до тех пор пока не автоматизируем (Bring the Pain Forward)

- Многие практики непрерывной доставки и эволюционной архитектуры служат примером принципа устранения боли, вдохновленного сообществом программистов eXtreme. Когда что-то в проекте может вызвать боль, заставьте себя делать это чаще и раньше, что, в свою очередь, побуждает вас автоматизировать боль и выявлять проблемы на раннем этапе. Распространенные практики непрерывной доставки, такие как конвейеры развертывания, автоматическое выделение ресурсов компьютеров и миграция баз данных, упрощают эволюционную архитектуру, устраняя общие болевые точки для изменений.

лего-конструктор уже есть

Если сложить эти штуки в кучу и немного прищуриться, это начинает напоминать лего-версию разработки распределенных систем данных. Вы можете соединить эти ингредиенты вместе, чтобы создать огромное множество возможных систем. Это явно не относится к конечным пользователям, которые, по-видимому, больше заботятся об API, чем о том, как он реализован, но это может быть путь к достижению простоты единой системы в более разнообразном и модульном мире, который продолжает развиваться. . Если время внедрения распределенной системы из-за появления надежных и гибких строительных блоков составляет от нескольких лет до недель, то необходимость объединения в единую монолитную систему исчезает.

- Zookeeper handles much of the system co-ordination (perhaps with a bit of help from higher-level abstractions like Helix or Curator).
- Mesos and YARN do process virtualization and resource management
- Embedded libraries like Lucene and LevelDB do indexing
- Netty, Jetty and higher-level wrappers like Finagle and rest.li handle remote communication
- Avro, Protocol Buffers, Thrift, and umpteen zillion other libraries handle serialization
- Kafka and Bookeeper provide a backing log.

From <https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>

Conflict-free Replicated Data Types (CRDTs)

- CRDT - одна из самых интересных идей, появившихся в результате исследований распределенных систем в последние годы, которые дают нам богатые, в конечном итоге согласованные и составляемые структуры данных, такие как счетчики, карты и наборы, которые гарантированно сходятся последовательно без необходимость явного согласования. CRDT подходят не для всех вариантов использования, но являются очень ценным инструментом 11 при создании масштабируемых и доступных систем микросервисов.

вам необходимо продолжать переоценивать существующие абстракции (те переделывать микросервисы и переосмысливать агрегаты)

- Это означает, что по мере развития вашего понимания предметной области- например, признания необходимости распределения вычислений с гораздо более высокой степенью детализации, чем раньше,- вам необходимо продолжать переоценивать существующие абстракции с учетом того, отражают ли они существенную сложность и как они добавляют много случайных сложностей. Результатом будет адаптация решений, иногда представляющая сдвиг в отношении свойств, которые вы хотите абстрагироваться, и которые вы хотите раскрыть.

все(вся команда) должны понимать прогу

- на самом деле Scrum указал на то, что команда разработчиков очень плохо справляется с управлением конфликтующими приоритетами , и ответственность за их устранение лежит на Владельце продукта до того, как произойдет итерация разработки.
- Скрам не предписывал , что все обучение должно проходить через product owner-а . Это дисфункция, которая снижает качество обучения в целом и конечного продукта

Checklist: How many people understand your system?

- Nobody -> you're probably screwed
- Only one person -> you're probably *even more screwed*, but it's harder to admit.
- A few people are reasonably competent on the whole -> safer.
- Everybody knows the whole story -> just lovely.

* фоновый перенос данных в микросервис

24 января 2021 г. 20:03

[* Data replication]

data replication

фоновый перенос данных в микросервис (oldschool аналог event carried state transfer только без событий)

- так как события не переносят информацию в event fact, то нам с помощью ETL нужно постоянно закачивать в микросервис данные

- Но чтобы извлечь данные из какой-либо службы, а затем поддерживать эти данные в актуальном состоянии, вам нужен какой-то механизм опроса. Хотя это не так уж плохо, но и не идеально
- Более того, поскольку это происходит снова и снова в более крупных архитектурах, когда данные извлекаются и перемещаются от службы к службе, часто возникают небольшие ошибки или идиосинкразии. Со временем они обычно ухудшаются, и качество данных всей экосистемы начинает страдать. Чем больше изменяемых копий, тем больше данных со временем будет расходиться

вариант 1) integration via a shared database

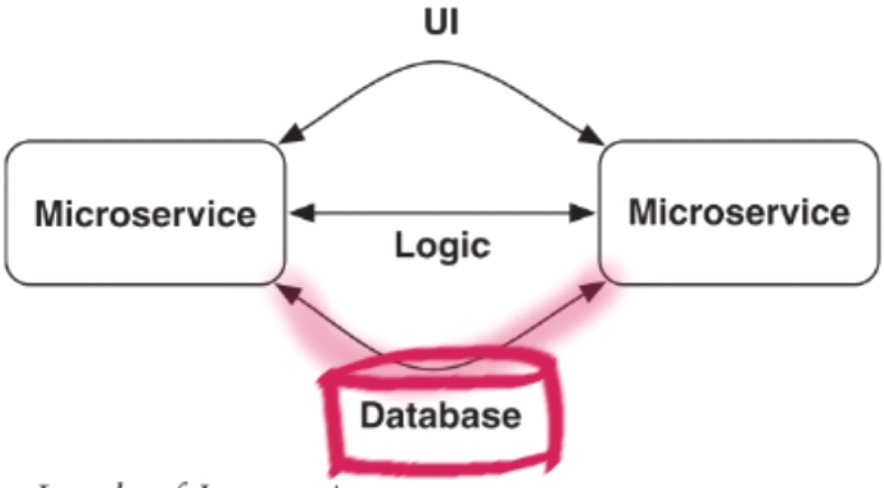


Figure 8.1 *Different Levels of Integration*

- Микросервисы содержат графический пользовательский интерфейс. Это означает, что микросервисы могут быть интегрированы на уровне пользовательского интерфейса. Этот тип интеграции представлен в разделе 8.1.

- Микросервисы также могут быть интегрированы на логическом уровне. Они могут использовать REST (раздел 8.2), SOAP, удаленный вызов процедур (RPC); (раздел 8.3) или обмен сообщениями (раздел 8.4) для этого.

- Наконец, интеграцию можно выполнить на уровне базы данных с помощью репликации данных

- На уровне базы данных микросервисы могут совместно использовать базу данных и получать доступ к одним и тем же данным
- Такой тип интеграции давно используется на практике: нередко база данных используется несколькими приложениями. Часто базы данных служат дольше, чем приложения, поэтому основное внимание уделяется базе данных, а не приложениям, которые находятся над ней.
- Хотя интеграция через общую базу данных широко распространена, у нее есть серьезные недостатки:

Представление данных не может быть легко изменено, поскольку к данным обращаются несколько приложений.

- Изменение может привести к поломке одного из приложений. Это означает, что изменения должны быть согласованы во всех приложениях.
- Это делает невозможным быстрое изменение приложений в ситуациях, связанных с изменениями базы данных. Однако возможность быстрого изменения приложения - это как раз то преимущество, которое должны принести микросервисы.
- Наконец, очень сложно привести схему в порядок - например, удалить столбцы, которые больше не нужны, - потому что неясно, использует ли еще какая-либо система эти столбцы. В конечном итоге база данных будет становиться все более сложной и труднее поддерживать.

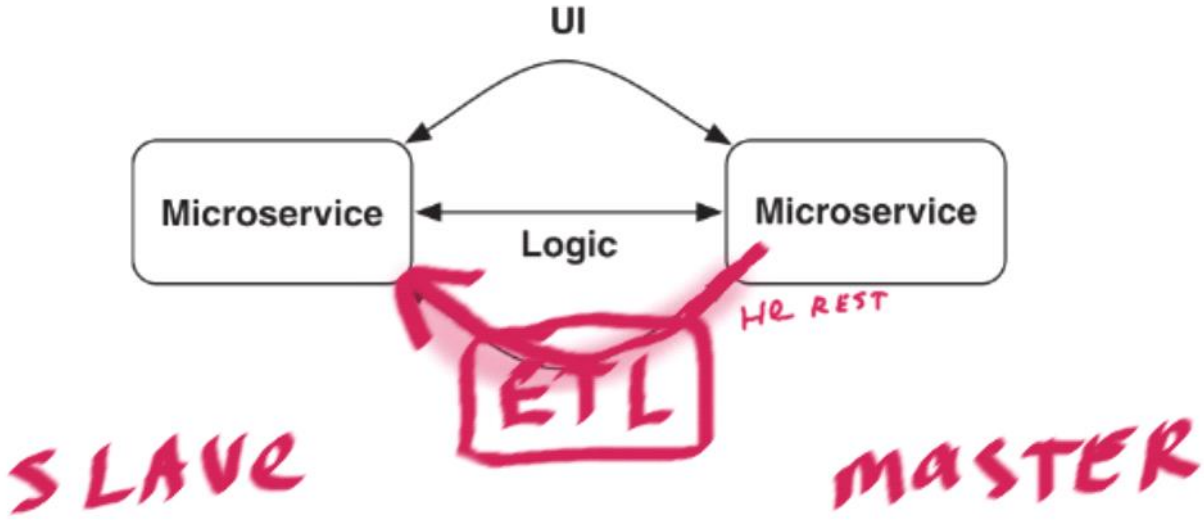
В конечном итоге совместное использование базы данных является нарушением важного архитектурного правила. Компоненты должны иметь возможность изменять свое внутреннее представление данных. без воздействия на другие компоненты.

- Схема базы данных является примером внутреннего представления данных. Когда несколько компонентов совместно используют базу данных, больше невозможно изменить представление данных. Следовательно, микросервисы должны иметь строго отдельное хранилище данных и не использовать общую схему базы данных.

вариант 2) Replicating data

для репликации лучше использовать более быстрый протокол чем REST

- Интерфейс не обязательно должен использовать протокол, такой как REST, но может использовать более быстрые альтернативные протоколы. Для этого может потребоваться другой механизм связи, отличный от того, который обычно используется микросервисами.



Репликация данных - особенно хороший выбор там, где требуется высокопроизводительный доступ к большим объемам данных

Репликация приводит к избыточному хранению данных.

eventualy consistency

- данные не сразу становятся согласованными: требуется время, чтобы изменения были реплицированы во все места.

- Однако немедленная согласованность часто не важна. Для задач анализа, таких как выполняемые хранилищем данных, анализа, который не включает заказы за последние несколько минут, может быть достаточно. Есть также случаи, когда последовательность не так важна. Когда заказу требуется немного времени, прежде чем он станет видимым в микросервисе доставки, это может быть приемлемо, потому что пока никто не будет запрашивать данные
- Высокие требования к согласованности затрудняют репликацию. При определении системных требований часто неясно, насколько согласованными должны быть данные. Это ограничивает возможности репликации данных.

При разработке механизма репликации в идеале должна быть ведущая система,

- содержащая текущие данные. Все остальные реплики должны получать данные из этой системы. Это дает понять, какие данные действительно актуальны. Изменения данных не должны храниться в разных системах, так как это легко вызывает конфликты и усложняет реализацию. Такие конфликты не проблема, когда есть только один источник изменений.

схемы данных двух БД должны различаться, чтобы не было зависимости(иначе будет

реализацию. Такие конфликты не проблема, когда есть только один источник изменений.

схемы данных двух БД должны различаться, чтобы не было зависимости(иначе будет shared database)

- Но следует позаботиться о том, чтобы репликация данных не привела к зависимости от схем базы данных через черный ход. Когда данные просто реплицируются и используется та же схема, возникает та же проблема, что и при совместном использовании базы данных. Изменение схемы повлияет на другие микросервисы, и микросервисы снова станут связанными. Этого следует избегать.
- Данные должны быть перенесены в другую схему, чтобы гарантировать независимость схем и, следовательно, микросервисов

batch replication - например каждый раз передавать полный набор данных

- Репликацию можно активировать пакетно. В этой ситуации можно передать весь набор данных или, по крайней мере, изменения с последнего запуска. При первом запуске репликации объем данных может быть большим, а это означает, что репликация занимает много времени. Тем не менее, может быть разумным каждый раз передавать все данные. Это дает возможность исправить ошибки, возникшие во время последнего запуска репликации.

- Простая реализация может назначать версию каждому набору данных. В зависимости от версии можно специально выбрать и реплицировать измененные наборы данных. Такой подход означает, что процесс может быть легко перезапущен, если он по какой-то причине прерван, поскольку сам процесс не сохраняет состояние. Вместо этого состояние сохраняется вместе с самими данными.

event replication

- Альтернативный метод - запуск репликации при определенных событиях. Например, когда набор данных создается заново, данные могут быть немедленно скопированы в реплики. Этот подход особенно легко реализовать с помощью обмена сообщениями

Многие системы на основе микросервисов обходятся вообще без репликации данных.

* Interaction styles

10 января 2021 г. 14:56

one-to-one/one-to-many все сочетания с synchronous/asynchronous

There are a variety of client-service interaction styles. As table 3.1 shows, they can be categorized in two dimensions. The first dimension is whether the interaction is one-to-one or one-to-many:

- *One-to-one*—Each client request is processed by exactly one service.
- *One-to-many*—Each request is processed by multiple services.

The second dimension is whether the interaction is synchronous or asynchronous:

- *Synchronous*—The client expects a timely response from the service and might even block while it waits.
- *Asynchronous*—The client doesn't block, and the response, if any, isn't necessarily sent immediately.

Table 3.1 The various interaction styles can be characterized in two dimensions: one-to-one vs one-to-many and synchronous vs asynchronous.

	one-to-one	one-to-many
Synchronous	Request/response	—
Asynchronous	Asynchronous request/response One-way notifications	Publish/subscribe Publish/async responses

The following are the different types of one-to-one interactions:

- *Request/response*—A service client makes a request to a service and waits for a response. The client expects the response to arrive in a timely fashion. It might event block while waiting. This is an interaction style that generally results in services being tightly coupled.
- *Asynchronous request/response*—A service client sends a request to a service, which replies asynchronously. The client doesn't block while waiting, because the service might not send the response for a long time.

- *One-way notifications*—A service client sends a request to a service, but no reply is expected or sent.

It's important to remember that the synchronous request/response interaction style is mostly orthogonal to IPC technologies. A service can, for example, interact with another service using request/response style interaction with either REST or messaging. Even if two services are communicating using a message broker, the client service might be blocked waiting for a response. It doesn't necessarily mean they're loosely coupled. That's something I revisit later in this chapter when discussing the impact of inter-service communication on availability.

The following are the different types of one-to-many interactions:

- *Publish/subscribe*—A client publishes a notification message, which is consumed by zero or more interested services.
- *Publish/async responses*—A client publishes a request message and then waits for a certain amount of time for responses from interested services.

асинхронный обмен с брокером и без

- Сервисы могут взаимодействовать путем асинхронного обмена сообщениями. Приложения, основанные на этом подходе, обычно используют брокер сообщений, который играет роль промежуточного звена между сервисами.
- Возможна и архитектура без брокера, когда сервисы взаимодействуют между собой напрямую

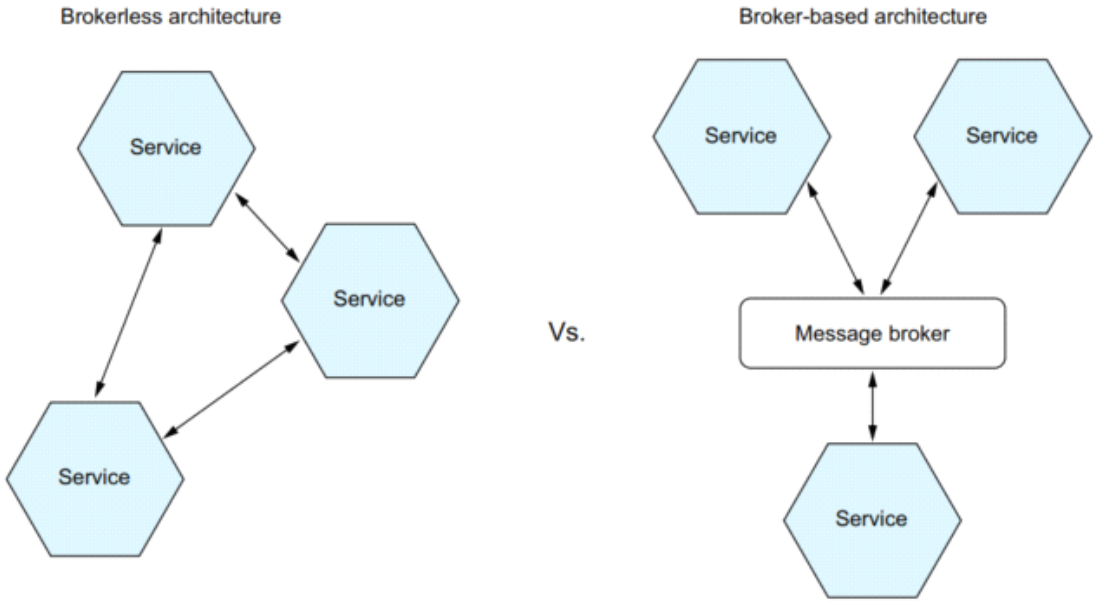


Figure 3.10 The services in brokerless architecture communicate directly, whereas the services in a broker-based architecture communicate via a message broker.

- без брокера нужно знать адрес получателя
- без брокера проблемы с доступностью, так как на время сеанса отправитель и

получатель должны быть доступны
- без брокера проблемы с гарантией доставки так как нужны повторные отправки

The brokerless architecture has some benefits:

- Allows lighter network traffic and better latency, because messages go directly from the sender to the receiver, instead of having to go from the sender to the message broker and from there to the receiver
- Eliminates the possibility of the message broker being a performance bottleneck or a single point of failure
- Features less operational complexity, because there is no message broker to set up and maintain

As appealing as these benefits may seem, brokerless messaging has significant drawbacks:

- Services need to know about each other's locations and must therefore use one of the discovery mechanisms describer earlier in section 3.2.4.
- It offers reduced availability, because both the sender and receiver of a message must be available while the message is being exchanged.
- Implementing mechanisms, such as guaranteed delivery, is more challenging.

Table 3.2 Each message broker implements the message channel concept in a different way.

Message broker	Point-to-point channel	Publish-subscribe channel
JMS	Queue	Topic
Apache Kafka	Topic	Topic
AMQP-based brokers, such as RabbitMQ	Exchange + Queue	Fanout exchange and a queue per consumer
AWS Kinesis	Stream	Stream
AWS SQS	Queue	—

BENEFITS AND DRAWBACKS OF BROKER-BASED MESSAGING

There are many advantages to using broker-based messaging:

- *Loose coupling*—A client makes a request by simply sending a message to the appropriate channel. The client is completely unaware of the service instances. It doesn't need to use a discovery mechanism to determine the location of a service instance.
- *Message buffering*—The message broker buffers messages until they can be processed. With a synchronous request/response protocol such as HTTP, both the client and service must be available for the duration of the exchange. With messaging, though, messages will queue up until they can be processed by the consumer. This means, for example, that an online store can accept orders from customers even when the order-fulfillment system is slow or unavailable. The messages will simply queue up until they can be processed.
- *Flexible communication*—Messaging supports all the interaction styles described earlier.
- *Explicit interprocess communication*—RPC-based mechanism attempts to make invoking a remote service look the same as calling a local service. But due to the laws of physics and the possibility of partial failure, they're in fact quite different.

Messaging makes these differences very explicit, so developers aren't lulled into

Messaging makes these differences very explicit, so developers aren't lulled into a false sense of security.

There are some downsides to using messaging:

- *Potential performance bottleneck*—There is a risk that the message broker could be a performance bottleneck. Fortunately, many modern message brokers are designed to be highly scalable.
- *Potential single point of failure*—It's essential that the message broker is highly available—otherwise, system reliability will be impacted. Fortunately, most modern brokers have been designed to be highly available.
- *Additional operational complexity*—The messaging system is yet another system component that must be installed, configured, and operated.

MESSAGE CHANNELS (канал сообщений - это абстракция инфраструктуры обмена сообщениями)

- A *point-to-point* channel delivers a message to exactly one of the consumers that is reading from the channel. Services use point-to-point channels for the one-to-one interaction styles described earlier. For example, a command message is often sent over a point-to-point channel.
- A *publish-subscribe* channel delivers each message to all of the attached consumers. Services use publish-subscribe channels for the one-to-many interaction styles described earlier. For example, an event message is usually sent over a publish-subscribe channel.

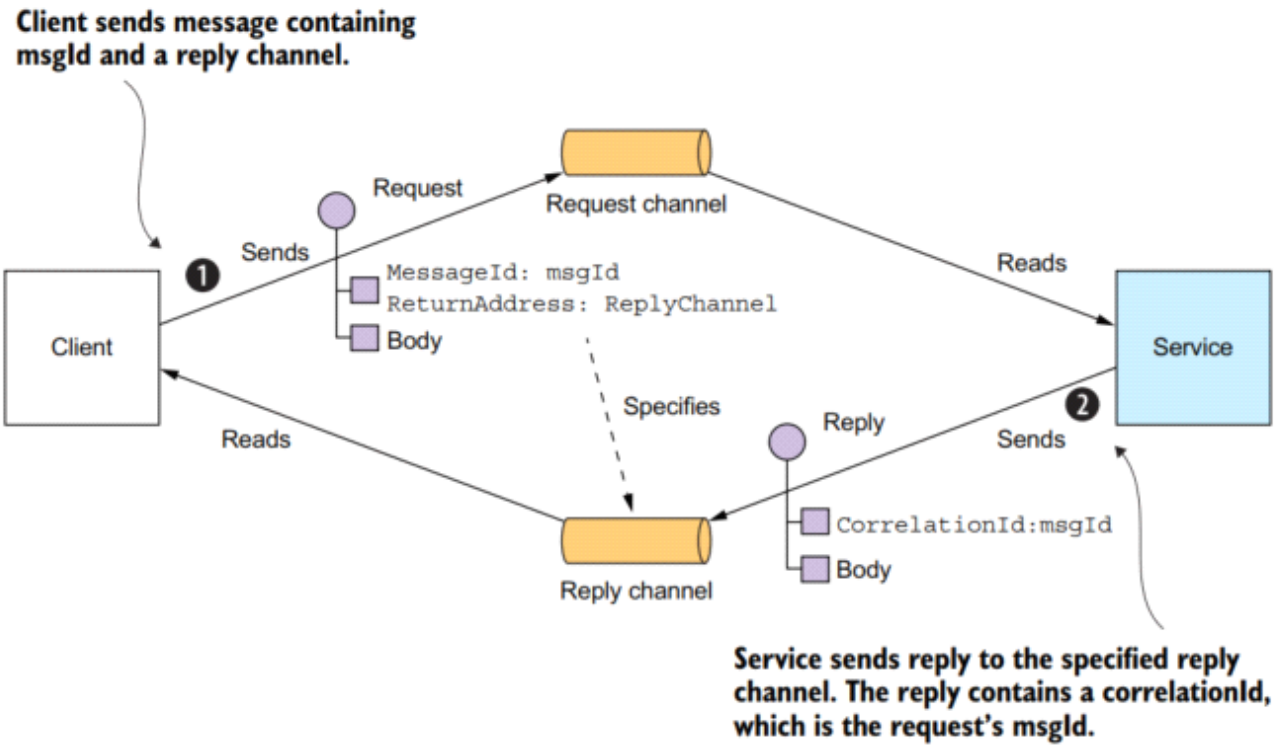


Figure 3.8 Implementing asynchronous request/response by including a reply channel and message identifier in the request message. The receiver processes the message and sends the reply to the specified reply channel.

correlation id

- Клиент должен сообщить сервису, куда тому следует вернуть результат, и сопоставить ответное сообщение со своим запросом. К счастью, решить эти две задачи несложно. Клиент отправляет командное сообщение с каналом ответа в заголовке. Сервер записывает в этот канал свой ответ, содержащий идентификатор соответствия с тем же значением, что и идентификатор запроса. Клиент использует идентификатор соответствия, чтобы сопоставить свое сообщение с ответом.
- Теоретически клиент может заблокироваться, пока не получит ответ, но на практике обработка ответов происходит асинхронно. Кроме того, ответы обычно могут обрабатываться любым экземпляром клиента

PUBLISH/ASYNC RESPONSES (publish/subscribe + request/response)

- Клиент публикует в канале типа «издатель — подписчик» сообщение с каналом ответа в заголовке.
- Потребитель записывает ответное сообщение с идентификатором соответствия в канал ответа.
- Клиент принимает ответы и сверяет их с запросом с помощью идентификатора соответствия.

* проблемы с центральной БД

24 января 2021 г. 18:48

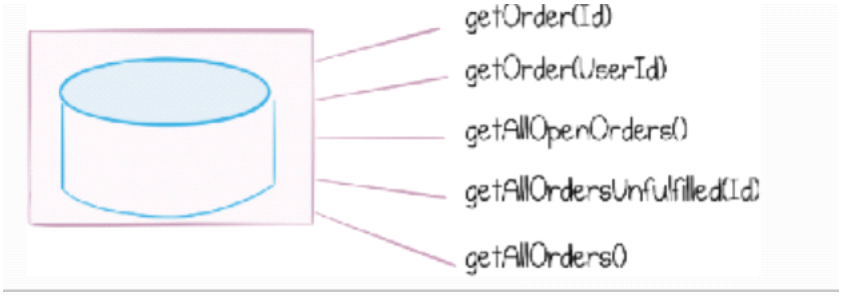
проблемы с центральной БД

максимальное раскрытие данных базой данных , все связывает в клубок (все модули завязываются на эту структуру и ее уже изменить легко невозможно)

Инкапсуляция побуждает нас скрывать данные, но системы данных имеют мало общего с инкапсуляцией. На самом деле, как раз наоборот: базы данных делают все возможное, чтобы раскрыть хранящиеся в них данные (рис. 8-2). Они поставляются с удивительно мощными декларативными интерфейсами, которые могут искажать данные, которые они хранят, в практически любую форму, которую вы пожелаете. Это именно то, что нужно специалисту по данным для исследовательского исследования, но это не так хорошо для управления спиралью межсервисных зависимостей в растущей сфере обслуживания

Итак, мы сталкиваемся с загадкой, дихотомией: базы данных предназначены для раскрытия данных и их использования. Сервисы скрывают это, чтобы они могли оставаться изолированными. Эти две силы фундаментальны. Они лежат в основе большей части того, что мы делаем, тонко борясь за господство в создаваемых нами системах.

oldschool микросервис тоже старается раскрыть все больше данных как БД



ок 8-3. Сервисные интерфейсы со временем неизбежно разрастаются

Более того, в то время, когда были задуманы реляционные СУБД, существовал только один рынок СУБД - обработка бизнес-данных.

За последние 25 лет развился ряд других рынков, включая хранилища данных, управление текстом и потоковую обработку. Требования к этим рынкам сильно отличаются от требований к обработке бизнес-данных.

- архитектуры на порядок или более в нескольких областях применения, включая:
- Текст (специализированные движки от Google, Yahoo и др.)
- Хранилища данных (колоночные хранилища, такие как Vertica, Monet [Bon02] и т. Д.)
- Обработка потоков (движки потоковой обработки, такие как StreamBase и Coral8)
- Научные и интеллектуальные базы данных (механизмы хранения массивов, такие как MATLAB и ASAP [SBC + 07])

раньше операция update берегла место на дорогих дисках, сейчас диски дешовые поэтому мы можем хранить транзакции вечно как в бухгалтерии

- CRUD больше ненужен

- Хорошей новостью является то, что сегодня дисковое пространство невероятно дешево, поэтому практически нет причин использовать обновление на месте для System of Record . Мы можем позволить себе хранить все данные, которые когда-либо были

созданы в системе, давая нам полную историю всего, что когда-либо в ней происходило.
- Нам больше не нужны обновления и удаление .
- Мы просто создаем новые факты либо добавляя дополнительные знания, либо делая новые выводы из существующих знаний - и считывая факты из любой точки истории системы. CRUD больше не нужен.

Подавляющее большинство баз данных OLTP имеют размер менее 1 Тбайта и довольно медленно растут в размере.

Через несколько лет терабайт оперативной памяти не будет чем-то необычным.

OLTP-транзакции очень легкие.

Например, самая тяжелая транзакция в TPC-C читает около 200 записей. В среде с основной памятью полезная работа такой транзакции занимает менее одной миллисекунды на машине низкого уровня.

Здесь можно спросить: «А как насчет долго выполняющихся команд?» В реальных OLTP-системах их нет по двум причинам: во-первых, операции, которые связаны с длительными транзакциями, такими как ввод данных пользователем для покупки в интернет-магазине, обычно разделяются на несколько транзакций для сохранения короткое время транзакции. Другими словами, хороший дизайн приложения позволит сократить объем запросов OLTP.

Следующее десятилетие принесет господство компьютерным системам без совместного использования ресурсов, часто называемым грид-вычислениями или блейд-вычислениями. Следовательно, любая СУБД должна быть оптимизирована для этой конфигурации. Очевидная стратегия - горизонтальное разделение данных по узлам сетки, тактика, впервые исследованная в Gamma [DGS + 90].
Транзакции типа ACID значительно упрощают жизнь разработчиков, создавая иллюзию того, что каждая из них имеет эксклюзивный доступ к данным.

- традиционный подход к управлению распределенными транзакциями не очень подходит для современных приложений

БД будущего должна быть без настрое

Гораздо лучший ответ - полностью переосмыслить процесс настройки и создать новую систему без видимых ручек.

* как альтернативный REST'у вариант не только KAFKA но и gRPC

24 января 2021 г. 18:48

как альтернативный REST'у вариант не только KAFKA но и gRPC

gRPC has several benefits:

- It's straightforward to design an API that has a rich set of update operations.
- It has an efficient, compact IPC mechanism, especially when exchanging large messages.
- Bidirectional streaming enables both RPI and messaging styles of communication.
- It enables interoperability between clients and services written in a wide range of languages.

gRPC also has several drawbacks:

- It takes more work for JavaScript clients to consume gRPC-based API than REST/JSON-based APIs.
- Older firewalls might not support HTTP/2.

gRPC поддерживает любые операции (а не только GET, POST)

- одна из трудностей применения REST связана с тем, что HTTP поддерживает ограниченный набор команд, из-за чего процесс проектирования REST API с поддержкой нескольких операций обновления не всегда оказывается простым. Одна из технологий, которой удается избежать этой проблемы, — gRPC (www.grpc.io)

также поддерживает схему данных

Listing 3.1 An excerpt of the gRPC API for the `Order Service`

```
service OrderService {
  rpc createOrder(CreateOrderRequest) returns (CreateOrderReply) {}

  rpc cancelOrder(CancelOrderRequest) returns (CancelOrderReply) {}
  rpc reviseOrder(ReviseOrderRequest) returns (ReviseOrderReply) {}
  ...
}

message CreateOrderRequest {
  int64 restaurantId = 1;
  int64 consumerId = 2;
  repeated LineItem lineItems = 3;
  ...
}

message LineItem {
  string menuItemId = 1;
  int32 quantity = 2;
}


message CreateOrderReply {
  int64 orderId = 1;
}
...
```

В качестве формата сообщений gRPC использует Protocol Buffers

- Определение сервиса является аналогом интерфейса в Java и представляет собой набор строго типизированных методов.
-

Помимо простых вызовов, состоящих из запроса и ответа, gRPC поддерживает поточный RPC

- Сервер может вернуть клиенту поток сообщений. В то же время клиент может сам отправить поток сообщений на сервер.

* само по себе разбиение на модули/микросервисы НЕ панацея

24 января 2021 г. 18:50

monolith - microlith - microservice

monolith - Система, не состоящая из микросервисов, может быть развернута только полностью. Поэтому его называют монолитом развертывания

- те слово монолит или микросервис это больше про деплоймент
- Конечно, монолит развертывания можно разделить на модули.

microlith - это просто тупое разбиение системы на модули, которые общаются между собой по синхронным протоколам (что на самом деле не отличается от монолита)

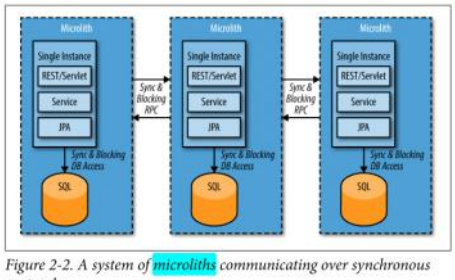


Figure 2-2. A system of microliths communicating over synchronous protocols

(A) само по себе разбиение на модули/микросервисы НЕ панацея (для архитектура)

- Фактически, многие реализации паттерна микросервисов страдают от ошибочного представления о том, что разбиение на модули программного обеспечения по сети каким-то образом повысит его устойчивость. This is no panacea; in fact, many implementations of the microservices pattern suffer from the misconceived notion that modularizing software over the network will somehow improve its sustainability.

- В этом отношении паттерн «микросервисы» необычайно самоуверен. Это сильно упирается в независимость в организации, программном обеспечении и данных. Микросервисы управляются разными командами, имеют разные циклы развертывания, не используют общий код и базы данных. 2 Проблема в том, что замена его сетью вызовов RPC / REST обычно не является предпочтительным решением. Это приводит к серьезному противоречию: мы хотим продвигать повторное использование для быстрой разработки программного обеспечения, но в то же время, чем больше у нас зависимостей, тем сложнее его изменить.

- Когда дело доходит до независимости сервисов, такие шаблоны, как микросервисы, являются самоуверенными: сервисы выполняются разными командами, имеют разные циклы развертывания, не используют общий код и не используют общие базы данных. Проблема в том, что замена этого паутиной вызовов RPC не решает вопроса: как службы получают доступ к этим островам данных для чего-либо, кроме тривиального поиска?

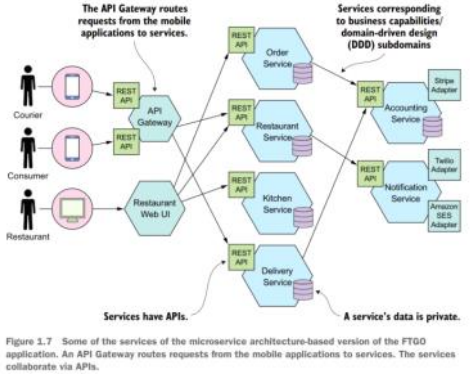


Figure 1.7 Some of the services of the microservice architecture-based version of the FTGO application. An API Gateway routes requests from the mobile applications to services. The services collaborate via APIs.

(B) само по себе разбиение на модули/микросервисы панацея для CI/CD (большой CI/CD теперь разбит на несколько CI/CD, которые теперь работают параллельно)

- Микросервисная архитектура обеспечивает автономность команд разработчиков
- Она обеспечивает уровень развертываемости, необходимый для непрерывных доставки и развертывания. Каждый сервис может быть развернут независимо от других. Если разработчикам, которые занимаются сервисом, нужно выкатить изменение, затрагивающее только этот сервис, они могут не координировать свои действия с другими командами
- Она позволяет сделать команды разработчиков автономными и слабо связанными между собой. Вы можете организовать отдел разработки в виде набора небольших команд (например, по принципу двух пицц). В этом случае каждая команда полностью отвечает за разработку и развертывание одного или нескольких связанных между собой сервисов. Она может разрабатывать, развертывать и масштабировать свои сервисы независимо от остальных разработчиков (рис. 1.8). Это значительно ускоряет темп разработки.
- Each service in a microservice architecture can be scaled independently of other services using X-axis cloning(разбиение на инстансы) and Z-axis partitioning
- только с точки зрения CI/CD пайплайна просто разбиение проги на модули приносит пользу, так как ускоряет процесс за счет параллельных пайплайнов
- в случае монолита, пайплайн всего один
- Основная проблема приложения FTGO заключается в его чрезмерной сложности. Оно слишком большое для того, чтобы один разработчик мог его понять. В итоге исправление ошибок и реализация новых возможностей усложнились и стали занимать много времени. Разработчики не успевали в срок.

- что новейшим подходом к обслуживанию приложений типа SaaS (Software-as-a-Service) является непрерывное развертывание'. доставка изменений в промышленную среду многократно в течение суток и в рабочее время. Как сообщается, по состоянию на 2011 год компания Amazon.com развертывала изменения каждые 11,6 с, никак не затрагивая при этом конечного пользователя!
- Конвейер непрерывной доставки значительно быстрее, потому что единицы развертывания меньше. Следовательно, развертывание происходит быстрее.
- Для микросервисов проще создать конвейер непрерывной доставки . Настроить среду для монолита развертывания сложно. В большинстве случаев требуются мощные серверы

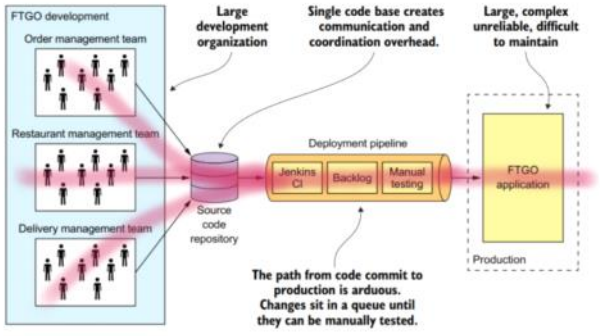


Figure 1.2 A case of monolithic hell. The large FTGO developer team commits their changes to a single source code repository. The path from code commit to production is long and arduous and involves manual testing. The FTGO application is large, complex, unreliable, and difficult to maintain.

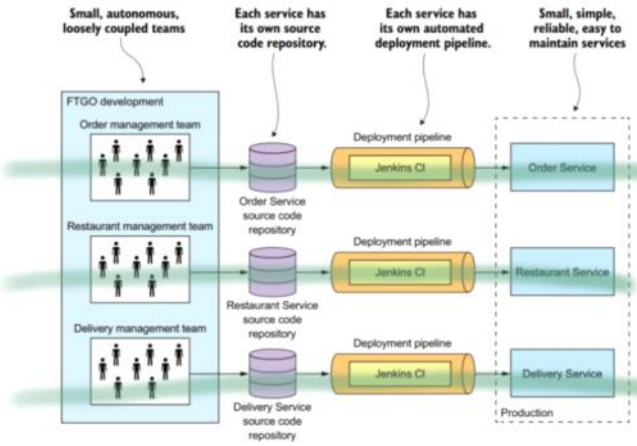


Figure 1.8 The microservices-based FTGO application consists of a set of loosely coupled services. Each team develops, tests, and deploys their services independently.

* реактивное программирование

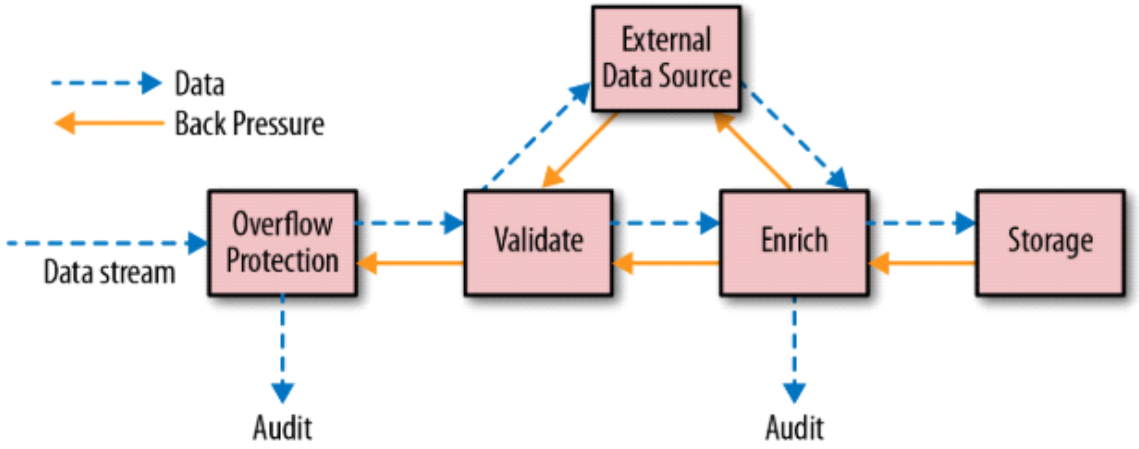
24 января 2021 г. 18:51

реактивное программирование это не то же самое что реактивные системы (есть просто некая путаница терминологии)

https://www.oreilly.com/radar/reactive-programming-vs-reactive-systems/
микросервисы входят в состав систем . Однако менее очевидно то, что микросервисы также являются системами
. Каждый микросервис должен быть спроектирован как система, распределенная, которая должна работать вместе как единое целое.

backpressure - это когда потребитель сам говорит, сколько данных ему нужно

- вы должны всегда применять backpressure .
- Противодавление - это управление потоком, гарантирующее, что быстрый производитель не сможет подавить более медленного потребителя, разрешив ему отправить больше данных, чем он может обработать, как показано на рисунке 5-4
- Противодавление повышает надежность не только отдельных компонентов, но и конвейера данных и системы в целом. Обычно это достигается за счет восходящего обратного канала, в котором нисходящие компоненты(потребители) могут сигнализировать о том, сколько данных им необходимо



стандарт на backpressure в JVM, Flow API

- Started by Lightbend in 2014, which has, together with Netflix, Pivotal, Oracle, and Twitter, created a standard for backpressure on the JVM, now staged for inclusion in Java 9 as the Flow API

SOA vs REST

24 января 2021 г. 19:56

SOA vs microservices

Table 1.1 Comparing SOA with microservices

	SOA	Microservices
Inter-service communication	Smart pipes, such as Enterprise Service Bus, using heavyweight protocols, such as SOAP and the other WS* standards.	Dumb pipes, such as a message broker, or direct service-to-service communication, using lightweight protocols such as REST or gRPC
Data	Global data model and shared databases	Data model and database per service
Typical service	Larger monolithic application	Smaller service

Table 6.1 *Differences between SOA and Microservices*

	SOA	Microservices
Scope	Enterprise-wide architecture	Architecture for one project
Flexibility	Flexibility by orchestration	Flexibility by fast deployment and rapid, independent development of Microservices
Organization	Services are implemented by different organizational units	Services are implemented by different organizational by teams in the same project
Deployment	Monolithic deployment of several services	Each microservice can be deployed individually
UI	Portal as universal UI for all services	Service contains UI

Как и REST, SOAP использует HTTP, но использует только сообщения POST для передачи данных на сервер

SOAP является механизмом RPC

- SOAP call runs a method on a certain object on the server and is therefore an RPC mechanism (remote-procedure call) В конечном счете, вызов SOAP запускает метод для определенного объекта на сервере и, следовательно, является механизмом RPC (удаленный вызов процедуры).

- В SOAP отсутствуют такие концепции, как HATEOAS, которые позволяют гибко управлять отношениями между микросервисами. Интерфейсы должны быть полностью определены сервером и известны клиенту.

-

в SOA Интерфейсы должны быть полностью определены сервером и известны клиенту (в отличие от RESTful HATEOAS)
SOAP может передавать сообщения с использованием различных транспортных механизмов.

- Например, можно получить сообщение через HTTP, а затем отправить его через JMS или по электронной почте через SMTP / POP. Технологии на основе SOAP также поддерживают пересылку запросов. Например, стандарт безопасности WS-Security может шифровать или подписывать части сообщения. После того, как это будет сделано, части могут быть отправлены в различные службы без необходимости расшифровки
- то позволяет сформировать сложный стек WS-* протоколов. Из-за этой сложности может пострадать взаимодействие между различными службами и решениями Когда уровень координации состоит из всех микросервисов, создается монолит, который необходимо изменять при каждой модификации. Это противоречит концепции независимого развертывания микросервисов. WS- * лучше подходит для таких концепций, как SOA.

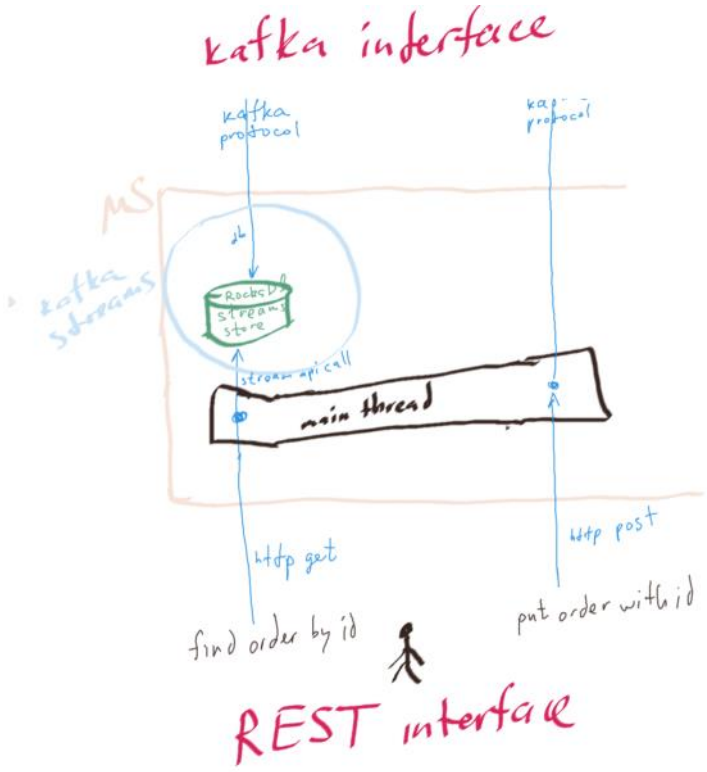
command-query request to microservice

22 декабря 2020 г. 0:24

? отталкиваясь от терминологии command, query, event то получается что CQRS это только про HTTP от клиентского фронта

kafka interface + REST interface для микросервиса нормально иметь два интерфейса

- REST интерфейс будет использоваться для command +query запросов
- kafka интерфейс будет использоваться для event fact + event trigger запросов (в терминологии command, query, event)



Когда поступает запрос от пользователя, вначале он представляет собой команду

- На данном этапе тоже вероятен сбой — например, при нарушении некоего условия целостности

Сначала приложение должно проверить, может ли выполнить команду

- Например, если пользователь пытается зарегистрироваться под каким-либо именем, или забронировать место в самолете, или купить билет в театр, то приложение должно убедиться, что это имя или место еще не занято
- При успешном прохождении проверки приложение может сгенерировать событие для демонстрации того, что данное имя пользователя зарегистрировано и ему присвоен определенный ID или что место зарезервировано для данного клиента.

Если проверка прошла успешно и команда принята, то она становится событием, которое является долговечным и неизменным

- В тот момент, когда событие сгенерировано, оно становится фактом.

- Даже если клиент впоследствии изменит свое решение или отменит бронирование, факт остается фактом: ранее он сделал резервирование определенного места, а изменение или аннулирование — это уже другое событие, которое добавится позже.
- Потребителю потока событий не разрешается отклонять событие: к тому времени, когда потребитель увидит событие, оно уже является неизменной частью журнала и, возможно, его уже видели другие потребители. Таким образом, любая проверка команды должна выполняться синхронно, прежде чем она станет событием, — например, с помощью сериализируемой транзакции, которая проверяет команду как неделимую единицу и затем публикует событие
- Вероятен другой вариант: запрос пользователя на бронирование места может быть разделен на два события — сначала предварительное резервирование, а затем отдельное событие подтверждения после проверки бронирования (как было описано в пункте «Реализация линеаризуемого хранилища с помощью рассылки общей последовательности» подраздела «Рассылка общей последовательности» раздела 9.3). Такое разделение позволяет выполнять валидацию в асинхронном процессе.

для command+query
вместо REST интерфейса , можно использовать KAFKA два топика (один для command, другой для query)

- kafka обеспечивает гарантированную сохранность сообщений если микросервис недоступен: retries, circuit-breaking, rate-limiting, monitoring
- но как альтернатива это в REST можно сделать через service mesh
- Если вы решите использовать команды или запросы через Kafka, используйте темы с одним разделом и уменьшите размер сегмента и хранения, чтобы снизить накладные расходы.

https://www.confluent.io/blog/apache-kafka-for-service-architectures/

Prefer Events over Commands or Queries

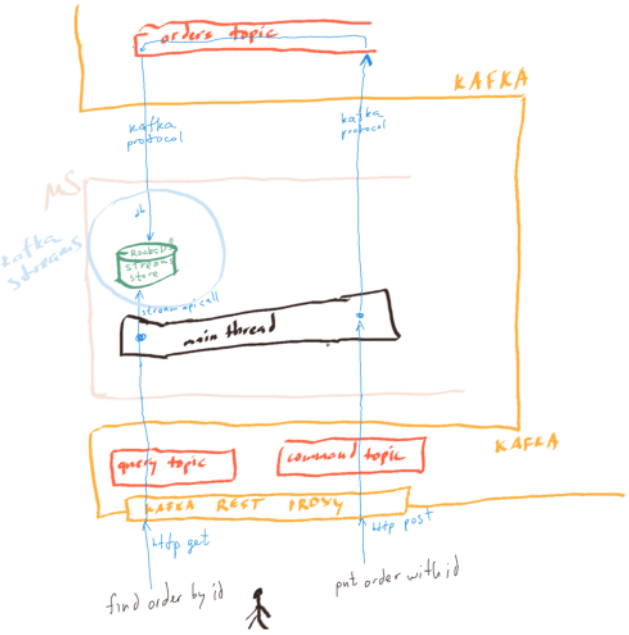
Kafka is used in a wide range of use cases, but like most technologies it has its sweet spot. It works very well for event-driven systems. However sometimes a request-response style is more appropriate, so what do you do?

Say you need a way for the Orders service to query a stock count from the Stock Service. You can synthesise such a request over any asynchronous channel. This is typically done by creating two topics, a request topic and a response topic.

Using Kafka for commands and queries is not uncommon, but it also isn't its sweet spot. Kafka topics are more heavyweight than, say, HTTP. But you shouldn't forget that, while HTTP is a lightweight protocol, real implementations need to worry about retries, circuit-breaking, rate-limiting, monitoring etc. (which is where service meshes often come in) but Kafka provides all these features out of the box. So it's a tradeoff.

If you choose to use commands or queries via Kafka use single partition topics and reduce the segment and retention sizes to keep the overhead low.

Another alternative is to mix in a stateless protocol like HTTP, layered over a backbone of events. For example with a gateway. Alternatively you can open separate interfaces in each service, one for events and another for request response. This is a great pattern, and probably the most commonly used one we see. So Events are created for all state changes and HTTP is used for all request response. See figure below. This pattern makes a lot of sense because events are, after all, the dataset of your system (so Kafka works well). Queries are more like ephemeral chit-chat (so HTTP is perfect). We covered more event-driven design patterns in the last post.



~~только для очень маленьких прог, для микросервисов можно смешивать два протокола~~

- REST для передачи EVENTS trigger (в терминологии command, query, event)
- KAFKA для передачи EVENT fact
- это норм только для сервисов которые с фронтом общаются, те где нельзя избежать command+query запросов

https://www.confluent.io/blog/apache-kafka-for-service-architectures/

те получается что "фронтовый" микросервис который общается с пользователем должен иметь в себе выоху всех данных которые могут понадобится пользователю во фронте

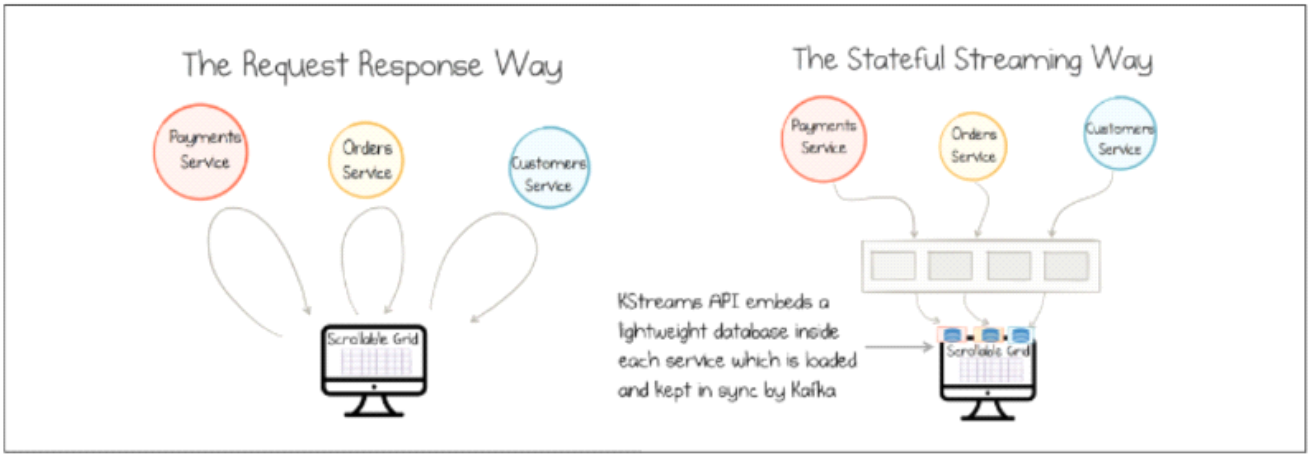


Figure 6-5. Stateful stream processing is used to materialize data inside a web server so that the UI can access it locally for better performance, in this case via a scrollable grid

Во время обработки запроса сервис не обращается синхронно ни к каким другим сервисам. Вместо этого он шлет им асинхронные сообщения

- Замечательной стороной этого подхода является то, что сервис Order сможет создать заказ и сразу ответить клиенту, даже если сервис Consumer окажется недоступным.

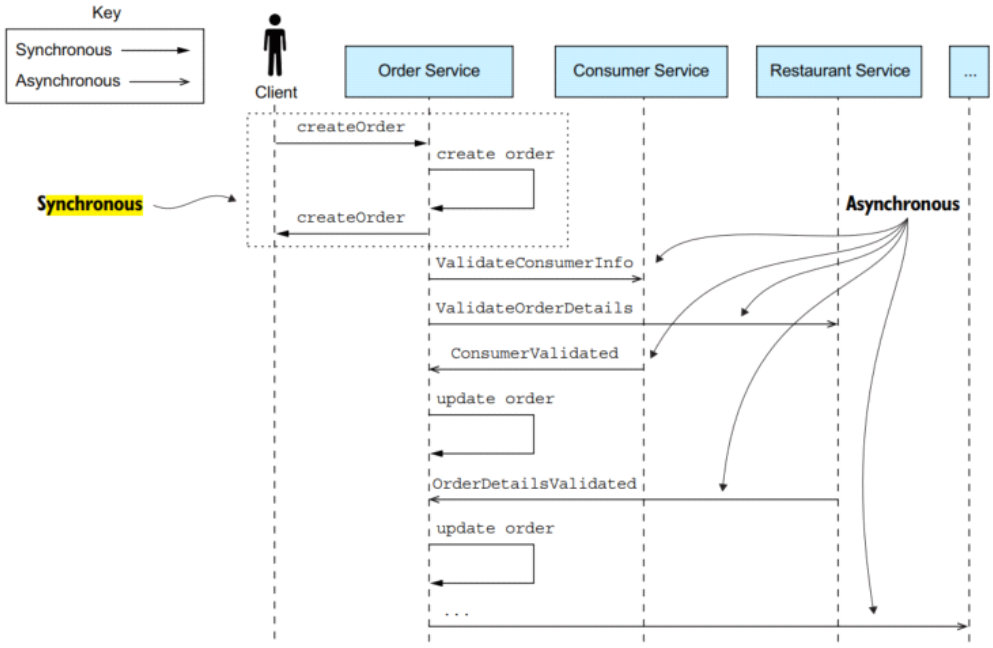


Figure 3.18 `Order Service` **creates an order without invoking any other service. It then asynchronously validates the newly created** `Order` **by exchanging messages with other services, including** `Consumer Service` **and** `Restaurant Service`.

при таком подходе у клиента минимальные гарантии по поводу заказа (клиент не знает создался он или не создался)

- Недостаток возвращения ответа до полной обработки запроса связан с тем, что это делает клиент более сложным. Например, когда сервис Order возвращает ответ, он дает минимальные гарантии по поводу состояния только что созданного заказа
- Он отвечает немедленно, еще до проверки заказа и авторизации банковской карты клиента. Таким образом, чтобы узнать о том, успешно ли создан заказ, клиент должен периодически запрашивать информацию или же сервис Order должен послать ему уведомительное сообщение

CQRS

19 декабря 2020 г. 23:18

отталкиваясь от Confluent терминологии command, query, event то получается что CQRS это только про HTTP от клиентского фронта

- Стоит упомянуть, что CQRS - это общий шаблон проектирования, который можно успешно использовать практически с любой стратегией сохранения и серверными модулями хранения, но он идеально сочетается с источником событий в контексте архитектур, управляемых событиями и сообщениями
- если в супер-общей терминологии то event carried state transfer, а также event fact + event trigger также попадают под CQRS

2 patterns for implementing query operations:

(1) API composition pattern

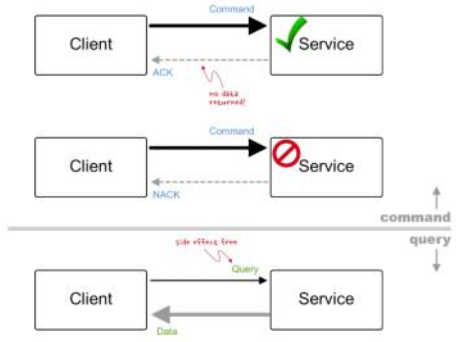
- Это самый простой подход, который следует использовать везде, где возможно. Он заключается в том, что за обращение к сервисам и объединение результатов отвечают клиенты

(2) CQRS Command query responsibility segregation pattern

- Он требует наличия одной или нескольких баз данных, единственное назначение которых заключается в поддержке запросов

основная идея CQRS - в том что запросы не приносят побочных эффектов

CQRS



Sample Commands

- AddItemToShoppingCart
- CheckoutShoppingCart
- ValidateCreditCard
- ReduceTemperature
- ...

Sample Queries

- GetTop10Songs
- GetRelatedProducts
- GetAccountBalances
- ...

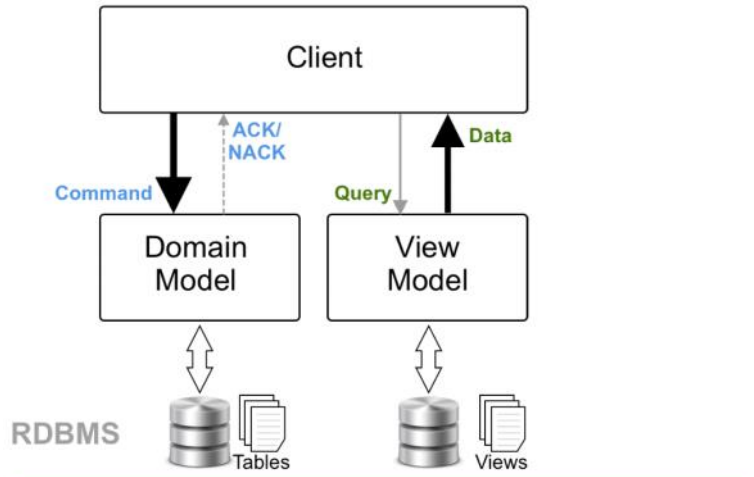
One of the building blocks that makes it easier to later on move to an event streaming platform is CQRS.

Command Query Responsibility Segregation (CQRS) is an important architectural pattern that has the following characteristics:

- When talking about CQRS we picture a **client** and a **service**
- **Commands** are sent by the client to the service
- The service tries to execute the desired **action** and answers with
  - ACK if the command could be executed successfully
  - NACK if the command could not be executed
- The service never returns any data other than ACK/NACK and maybe some additional meta data about the result of the invoked action
- The client uses **queries** to get data from the service
- Queries are **side-effect free**, that is, the do not cause any changes in state on the side of the service
- Since queries are side-effect free, they **can be repeated** multiple times, always delivering the same result

доменная модель может быть сильно нормализована
в то время как view-model наоборот денормализована (и представлять собой одну большую плоскую таблицу составленную из кучи данных "как бы из кучи разных таблиц")

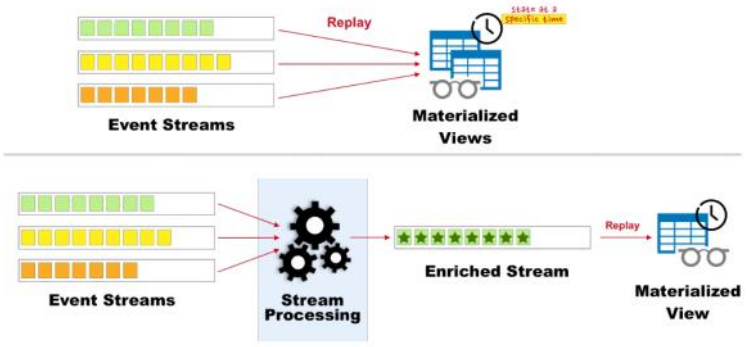
Domain Model and View Model



- Commands or actions are executed agains a domain model
- The resulting state changes are often persisted to a relational DB
- Queries are executed against a denormalized view of the current state of the domain
- Often these denormalized views are provided by database views in an RDBMS

Query в CQRS = materialized view = projection

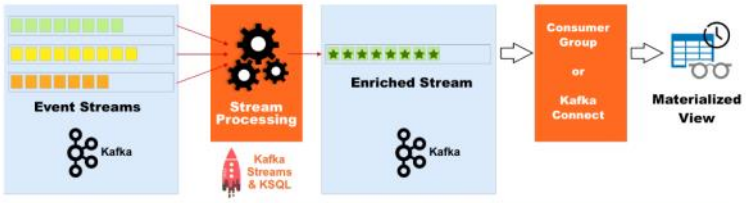
Materialized Views



- Querying a commit log is not efficient and should be avoided at all cost
- But how can we then query the **state of the world**?
- We are creating **materialized views** (sometimes called **projections**) for this purpose
- A materialized view shows the **denormalized state** as of now or any arbitrary point in the past
- materialized views are generated by **replaying streams of events** up to the desired point in time
- Several event streams can be **joined** to achieve denormalization or data enrichment

если вдруг materialized_view упадет, то его будет проще поднять из compacted топика (который уже аггрегировал и сджойнил все нужные события)

Materialized Views



- To generate denormalized or enriched materialized views we can use stream processing such as Kafka Streams or KSQL
- To export the enriched stream and create a materialized view in an external data store we can use either a custom Kafka consumer group or we can use Kafka Connect with the appropriate sink connector

Resilience

- Separating the read and write models gives us temporal decoupling of the writes and the actions performed in response to the writes—whether it's updating an index in a query database, pushing the events out for stream processing, performing a side-effect, or coordinating work across other services. Temporal decoupling means that the service doing the write as well as
- the service performing the action in response to the write don't need to be available at the same time. This avoids the need to
- handle things like retries, increases reliability, stability, and availability in the system as a whole, and allows for event ual consistency (as opposed to no consistency guarantees at all).

Масштабируемость - можно раздельно масштабировать каналы на запись и на чтение (особенно когда чтений много а записей совсем чутьчуть)

- Временное разделение операций чтения и записи дает нам возможность масштабировать их независимо друг от друга. Например, в си льно загруженной системе, ориентированной в основном на чтение , мы можем масштабировать сторону запроса до десятков или даже сотен узлов, сохраняя при этом сторону записи от трех до пяти узлов для обеспечения доступности или наоборот для записи- в основном системный (вероятно, с использованием методов сегментирования). Подобная развязанная конструкция обеспечивает большую гибкост ь, дает нам больше места и возможностей для поворота, а также позволяет лучше оптимизировать эффективность оборудования, вкладывая деньги туда, где это наиболее важно.

разделение write от read (в CQRS) ведет к eventually consistency

- см onenote:///C:\Users\trans\Qsync\vova_from_onenote\tf_algonote_v1\SYSTEMDESIGN\KAFKA__EVENT%20DRIVEN.one#eventually%20consistent&section-id={7172B8D4-BDE4-4FFF-A011-59DE79779A94}&page-id={5712FD9D-3167-40B8-B5C3-C54EB56AF03A}&end
- Another trade-off in moving to an architecture based on CQRS with event sourcing is that the write side and read side will be eventually co nsistent. It takes time for events to propagate between the two storage models, which often reside on separate nodes or clusters. The delay is often only a matter of milliseconds to a few seconds, but it c an have a big impact on the design of your system.

если сервис возвращает ответ фронту, но чтобы составить ответ, ему нужна информация от других сервисов (а запросы realtime он делать не хочет)

- Хороший пример — запрашивающий сервис Order History, который реализует запрос findOrderHistory(). Он подписывается на события, публикуемые нес колькими сервисами, включая Order, Delivery и т.

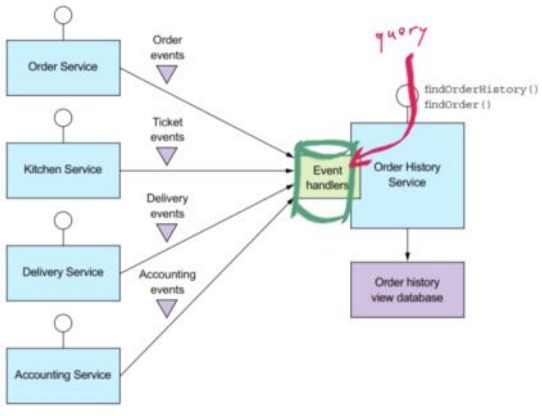


Figure 7.9 The design of `Order History Service`, which is a query-side service. It implements the `findOrderHistory()` query operation by querying a database, which it maintains by subscribing to events published by multiple other services.

шаблон предназначен для разделения обязанностей (как следует из названия)

- По большому счету, CQRS — это обобщенная разновидность популярной методики, которая заключается в том, что СУРБД используется в качестве системы записе й, а поисковая система, такая как Elasticsearch, отвечает за полнотекстовый поиск. Но есть одно отличие: в CQRS применяется более широкий диапазон баз данных, а не только система полнотекстового поиска.
- Еще одной сильной стороной CQRS является разделение ответственности. Доменная модель и соответствующая модель данных занимают ся либо командами, либо запросами. Шаблон CQRS предназначает отдельные программные модули и структуру базы данных для двух разных частей сервиса. Разделение командной стороны и стороны з апросов во многих случаях упрощает код и облегчает его поддержку

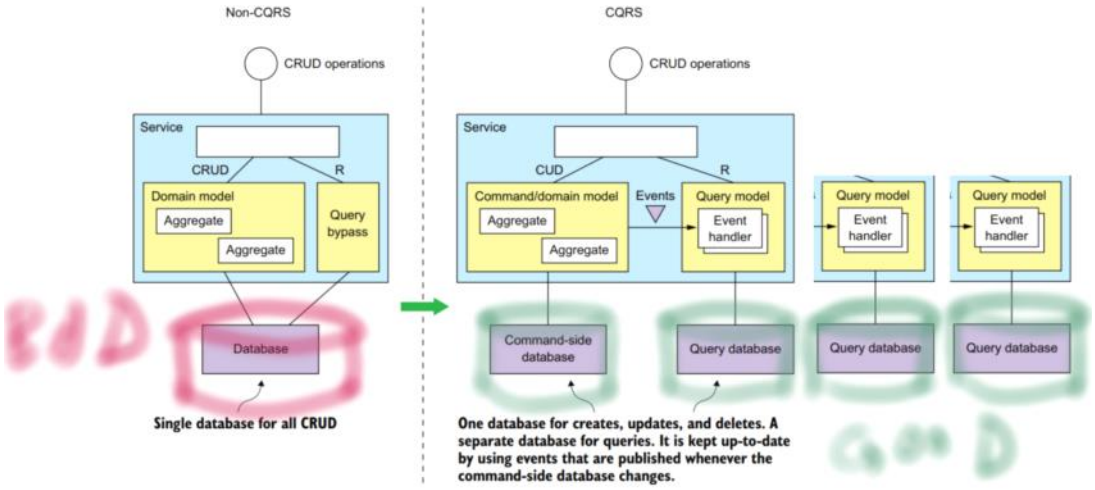
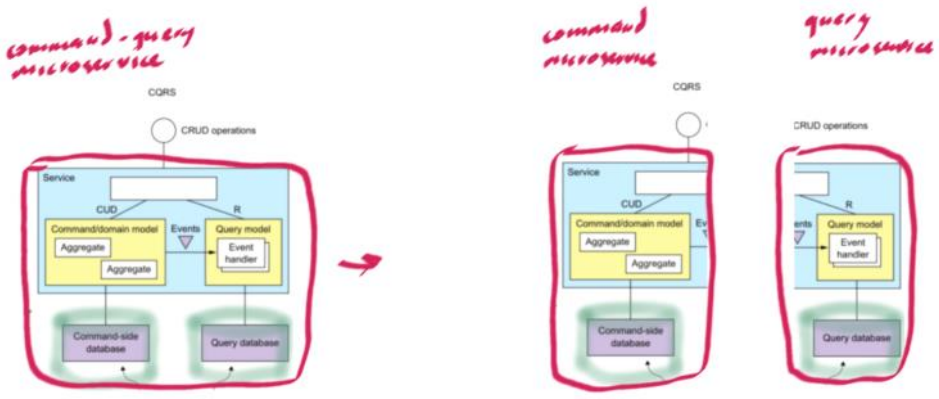


Figure 7.8 On the left is the non-CQRS version of the service, and on the right is the CQRS version. CQRS restructures a service into command-side and query-side modules, which have separate

Обработчик запросов также может быть отдельным микросервисом.

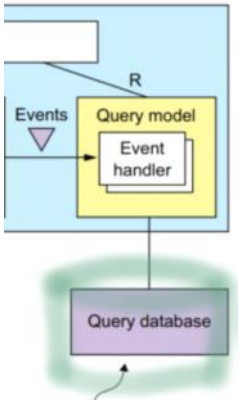
- Обработчик запросов также может быть отдельным микросервисом. Изменения в данных, которые использует обработчик запросов, могут быть внесены обработчиком команд в том же микросервисе. Однако обработчик команд также может быть отдельной микрослужбой. В этой ситуации обработчик запроса должен предложить подходящий интерфейс для доступа к базе данных, чтобы обработчик команд мог изменять данные
- Чтение и запись данных можно разделить на отдельные микросервисы. Это делает возможными микросервисы даже меньшего размера. Когда запись и чтение настолько сложны, что один микросервис для обоих становится слишком большим и трудным для понимания, разделение может иметь смысл
- Запись и чтение можно масштабировать по-разному, запустив разное количество микросервисов обработчиков запросов и микросервисов обработчиков команд. Это поддерживает точную масштабируемость микросервисов
- Легко запускать разные версии обработчиков команд параллельно. Это облегчает развертывание новых версий микросервисов.



QUERY-ONLY SERVICES (Такое представление имеет смысл реализовать отдельно, так как оно не относится ни к одному из существующих се рвисов)

- API запрашивающего сервиса состоит только из запросов и не поддерживает командные операции. Для реализации запросов он обраща ется к базе данных, которую поддерживает в актуальном состоянии, подписываясь на события, публикуемые одним или несколькими сервисами.
- это хороший способ реализовать представление, которое формируется благодаря подписке на события, генерируемые разными сервиса ми.
- Такое представление имеет смысл реализовать отдельно, так как оно не относится ни к одному из существующих сервисов. д.

R operations



паттерн удобен для JOIN-запросов, где нужно извлечь данные из нескольких микросервисов

- Одно из преимуществ шаблона CQRS — эффективная реализация запросов, которые извлекают данные из нескольких сервисов

для каждого типа запроса, можно сделать свое хранилище (эффективное именно для этого конкретного типа запросов)
те сделать много представлений (по одному представлению на каждый запрос)

- Поддержка всех запросов в рамках одной хранимой модели данных часто трудна, а иногда и просто невозможна
- Шаблон CQRS позволяет избежать ограничений конкретного хранилища данных за счет определения одного или нескольких представлений, каждое из которых эффективно реализует тот или иной запрос

паттерн позволяет преодолеть ограничения кафки (у которой примитивный API позволяет только класть и забирать события)

- Хранилище событий поддерживает только запросы по первичному ключу.
- CQRS устраняет эту проблему, создавая для агрегатов одно или несколько представлений, поддерживаемых в актуальном состоянии.

какую выбрать БД для CQRS, чтобы она была эффективной именно для этого типа запросов (например гео-база для гео-запросов)

- эффективной реализации запросов модуля представления. Именно характеристики этих запросов становятся основным фактором при выборе базы данных.
- Не так давно СУРБД на основе SQL были единственным видом баз данных, которые использовались для всего на свете. В некоторых случаях CQRS-представления лучше реализовывать с помощью баз данных с поддержкой SQL. Например когда может понадобиться база данных на основе SQL для поддержки системы отчетов.
- Базы данных NoSQL обычно хорошо подходят для CQRS-представлений, которые способны использовать их сильные стороны и игнорировать недостатки. CQRS-представлению идут на пользу более развитая модель данных и высокая производительность NoSQL

Table 7.1 Query-side view stores

If you need	Use	Example
PK-based lookup of JSON objects	A document store such as MongoDB or DynamoDB, or a key value store such as Redis	Implement order history by maintaining a MongoDB document containing the per-customer.
Query-based lookup of JSON objects	A document store such as MongoDB or DynamoDB	Implement customer view using MongoDB or DynamoDB.
Text queries	A text search engine such as Elasticsearch	Implement text search for orders by maintaining a per-order Elasticsearch document.
Graph queries	A graph database such as Neo4j	Implement fraud detection by maintaining a graph of customers, orders, and other data.
Traditional SQL reporting/BI	An RDBMS	Standard business reports and analytics.

Нецелесообразно внедрять CQRS в каждый микросервис.

- Однако во многих случаях такой подход может оказаться полезным для архитектур на основе микросервисов.

как обновлять БД на основе ивента

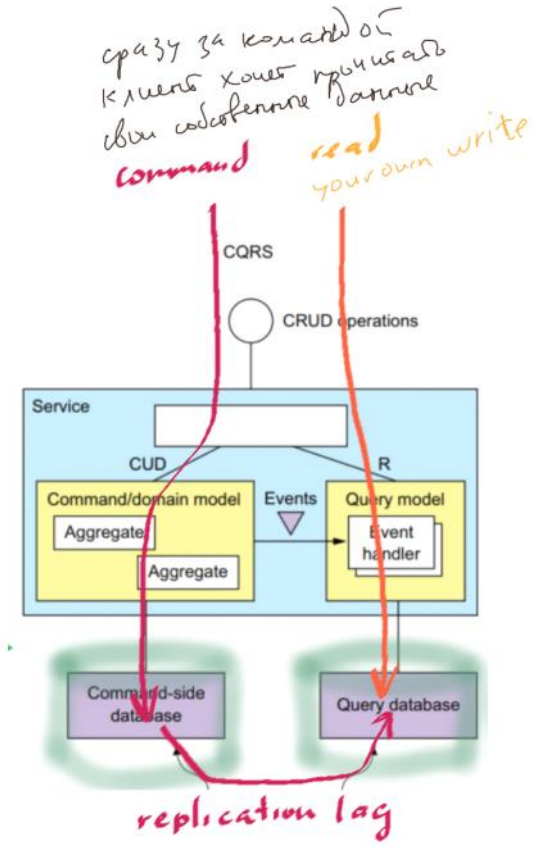
- эффективной реализации запросов модуля представления. Именно характеристики этих запросов становятся основным фактором при выборе базы данных. Но БД также должна эффективно реализовывать операции обновления, выполняемые обработчиками событий
- Некоторые виды баз данных имеют эффективную поддержку обновлений на основе внешнего ключа. Например, при задействовании СУРБД или MongoDB вам нужно создать индекс для соответствующего столбца.
- Однако в других NOSQLхранилищах выполнить обновления без применения первичных ключей может оказаться затруднительно. Приложению придется специально поддерживать некую связь между внешними и первичными ключами, чтобы знать, какую запись следует обновить.
- Например, DynamoDB поддерживает обновления и удаления только по первичному ключу, поэтому приложение, использующее эту БД, сначала должно запросить ее вторичный индекс (о нем чуть позже), чтобы определить первичные ключи обновляемых или удаляемых элементов.

HANDLING CONCURRENCY - проблема параллельного доступа к одной и той же записи БД

- Если представление подписывается на события, публикуемые агрегатами одного типа, никаких проблем с конкурентностью возникнуть не может. Дело в том, что события, которые публикуются определенным экземпляром агрегата, обрабатываются последовательно
- если события, на которые подписано представление, публикуются агрегатами разных типов, существует вероятность того, что несколько обработчиков одновременно попытаются обновить одну и ту же запись
- Например, обработчики событий Order* и Delivery* могут быть вызваны в один и тот же момент для одного и того же заказа. В этом случае они одновременно обратятся к DAO, чтобы обновить запись базы данных для этого экземпляра Order. Объект DAO должен быть написан так, чтобы такие ситуации улаживались корректно. Он не должен позволять одному обновлению перезаписывать другое
- DAO должен использовать пессимистичное или оптимистичное блокирование

последствия рассинхронизации между представлениями для команд и запросов

- Как можно было бы ожидать, между публикацией события командной стороной, его обработкой запрашивающим сервисом и обновлением представления проходит некоторое время.
- Клиентское приложение, которое обновляет агрегат и сразу же обращается к представлению, может получить предыдущую версию агре гата
- As I said earlier, one issue with using CQRS is that a client that updates the command side and then immediately executes a query might not see its own update. The view is eventually consistent because of the unavoidable latency of the messaging infrastructu



1) Клиент может периодически опрашивать представление, пока не получит актуальную информацию.

- Одно из решений заключается в том, чтобы API для команд и запросов предоставляли клиентам сведения о версии. Это позволит определить, устарел ли результат запроса
- * blocking read pattern(с помощью long polling)
- onenote:///C:\Users\trans\Qsync\vova_from_onenote\tf_algonote_v1\SYSTEMDESIGN\KAFKA__LOCAL%20STATE%20STORE.one#*%20blocking%20read%20pattern(с%20помощью%20long%20polling)&section-id={720F6F5D-93F8-4266-949D-E246C9711A22}&page-id={13D1C6D0-94D5-4D77-9EBE-377721365BB5}&end

2) web-клиент может обновить свою локальную модель, вообще не отправляя никакой дополнительный запрос.

- те достаточно только подтверждения того, что команда ушла успешно

- Скомпилированное мобильное приложение или одностраничный веб-сайт на JavaScript, которые реализуют пользовательский интерфейс, могут справиться с отставанием репликации за счет обновления своей локальной модели в ответ на успешное выполнение команды без применения запроса.
- Для обновления модели они могут, к примеру, использовать данные, возвращенные командой, и надеяться на то, что к моменту, когда пользователь инициирует запрос, представление успеет синхронизироваться.
- Один из недостатков этого подхода — то, что для обновления своей модели пользовательский интерфейс, возможно, должен будет дублировать серверный код.

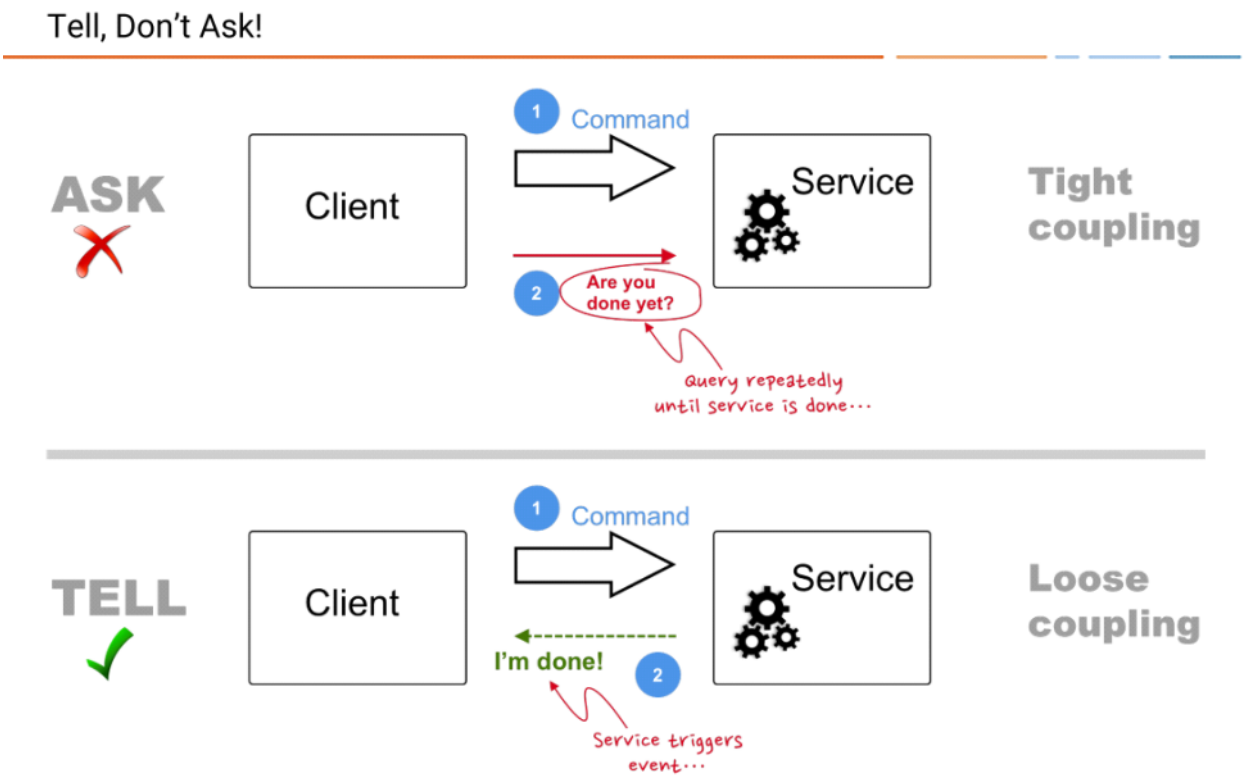
3) web-клиент в ответе от посылки команды получает токен и использует его для запроса
?? клиент многократно запрашивает по токену, до тех пор пока не получит апдейт

- Операция командной стороны возвращает клиенту токен с идентификатором опубликованного события.
- Клиент указывает этот токен в операции запроса.
- Если представление не обновилось в результате этого события, операция вернет ошибку.
- Модуль представления может реализовать этот механизм, используя систему обнаружения повторяющихся событий

Tell, Don't Ask принцип асинхронности

20 декабря 2020 г. 0:29

клиентский периодический пулинг не поощряется (только отдельный ответ от сервера)

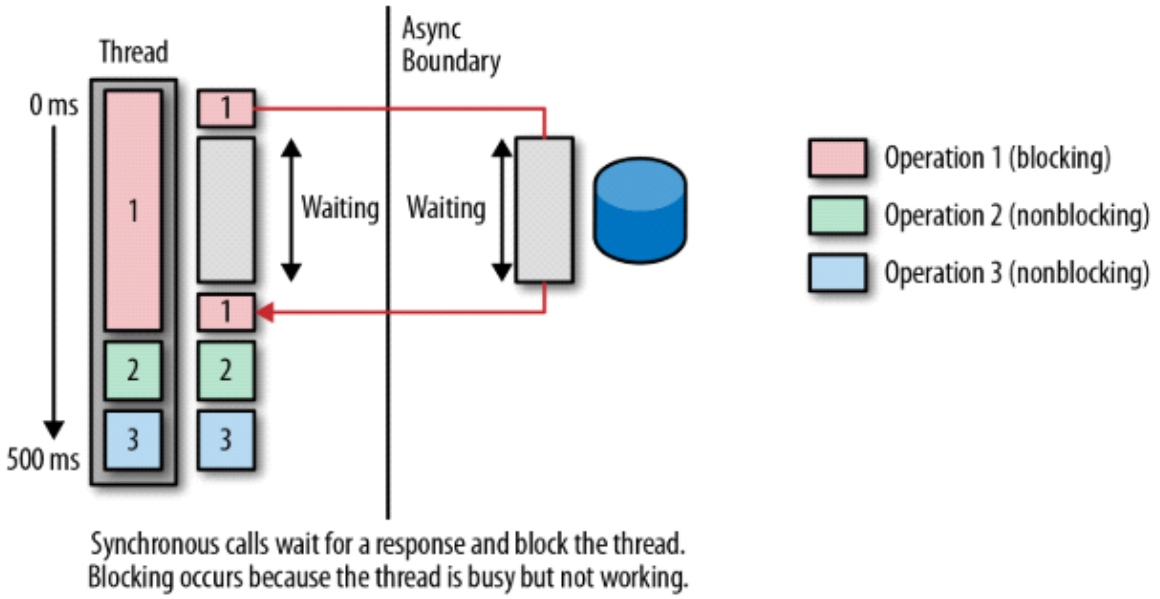


- To decouple individual parts in our application landscape we ideally adhere to the principle of **Tell, don't ask**
- On the slide we see in the upper part a client that requests an action from a service, and then repeatedly asks the service for the result of the action (are you done yet?)
- This is the "ask" case, and it is bad, since it tightly couples the client to the service
- A much better way, as shown in the lower half of the slide, is to have the client wait for a notice from the service when it is done. We call this the "tell" case
- This means that, instead a client to repeatedly ask a service "are you done?", the client waits for an event from the service telling that **it is done**
- Other than indicated on the slide, where the service reacts on a command from the client with an event, the same applies also when the client just is interested in events that happened on the service. E.g. an email service which consumes an event from the shipping service that an order has been shipped. Now the email service can send a confirmation email to the customer...
- The **tell don't ask** principle fits very well into the event driven architecture

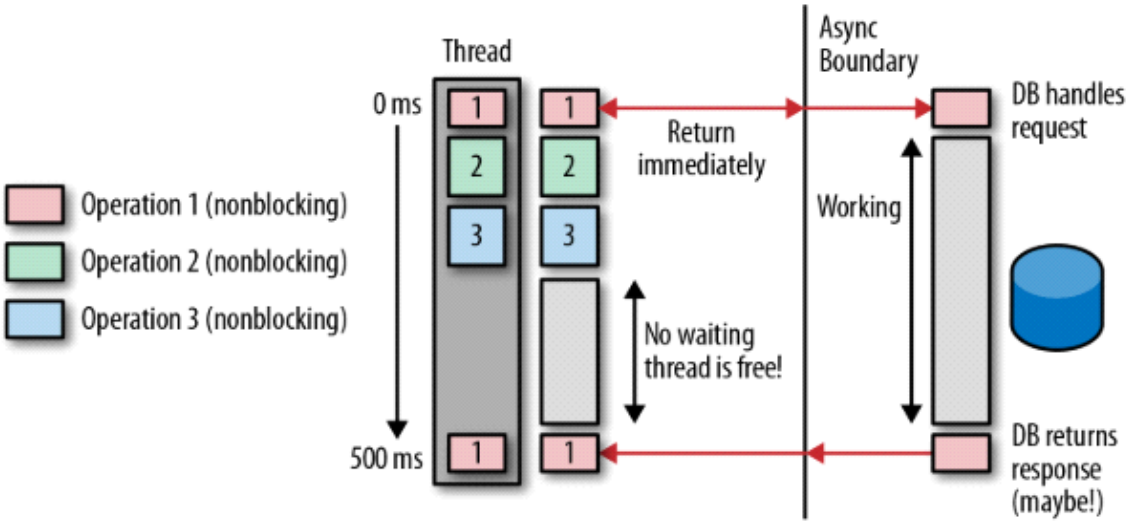
on the **Tell** side, it is worth mentioning, that clients can also just send events (facts) about what's happening, without knowing exactly which service(s) downstream will process those events. In other words, moving away from using events as "commands", and more for "notification".

блокировка микросервиса во время того как он ждет синхронный ответ - это всегда плохо

(The difference between blocking and nonblocking execution)



Synchronous calls wait for a response and block the thread.
Blocking occurs because the thread is busy but not working.



Asynchronous calls issue a request and return immediately.
This one is nonblocking because the response is returned later!

при синхронном общении обе стороны должны быть готовы к общению одновременно.

- При асинхронной связи отправитель может отправлять независимо от того, готов получатель или нет.

- Если сообщения отправляются быстрее, чем они могут быть доставлены, они где-то накапливаются, и в конечном итоге либо система рухнет, либо почта будет потеряна.

- Синхронное общение следует использовать с осторожностью. Это приводит к тесной зависимости от доступности. Если вызываемая система выходит из строя, вызывающая система должна иметь возможность справиться с этим. Степень связи в логике предметной

области также довольно высока. Синхронный вызов обычно точно описывает, что нужно сделать. Синхронная связь может потребоваться, если вы хотите, чтобы последние изменения были видны в других системах как можно скорее. В синхронном вызове всегда используется состояние во время вызова, тогда как асинхронная связь и репликация могут привести к задержке до тех пор, пока текущее состояние не станет известным повсюду.

синхронная передача сообщений бывает более удобна (когда асинхронность не требуется) и проще (так как не требует специальных механизмов и библиотек)

- Явная передача сообщений часто обеспечивает удобный способ взаимодействия изолированных частей приложения или помогает разграничить компоненты исходного кода, даже когда нет необходимости в асинхронном взаимодействии.
- Но если асинхронность не требуется, использование асинхронной передачи сообщений для разделения компонентов приводит к ненужным затратам: административным накладным расходам времени выполнения, связанным с диспетчеризацией задач для асинхронного выполнения, а также дополнительной задержкой планирования. В этом случае обычно более разумным выбором является синхронная передача сообщений.
- Мы упоминаем об этом, потому что синхронное распространение сообщений часто полезно для потоковой обработки. Например, объединение серии преобразований сохраняет преобразованные данные локальными для одного ядра ЦП и, таким образом, позволяет лучше использовать связанные с ним кеши. Таким образом, синхронный обмен сообщениями служит другой цели, чем разделение частей реактивного приложения,
-

* synchronous microservices vs asynchronous microservices

26 января 2021 г. 18:44

существует термин synchronous microservices и asynchronous microservices

* synchronous microservices

* asynchronous Microservices

WebSocket

23 декабря 2020 г. 22:23

ивенты из топика напрямую идут в WebSocket (UI)

- Если мы перейдем к архитектуре, в которой материализованные представления обновляются из потока изменений, это открывает новую захватывающую перспективу: когда клиент читает из одного из этих представлений, он может поддерживать соединение. Если это представление позже обновляется из-за некоторых изменений, появившихся в потоке, сервер может использовать это соединение для уведомления клиента об изменении (например, с помощью WebSocket или событий, отправленных сервером). Затем клиент может соответствующим образом обновить свой пользовательский интерфейс.
- Это означает, что клиент не просто читает представление в определенный момент времени, но фактически подписывается на поток изменений, которые могут произойти впоследствии. При условии, что интернет-соединение клиента остается активным, сервер может отправлять любые изменения клиенту. В конце концов, зачем вам вообще нужна устаревшая информация на экране, если доступна более свежая информация? Представление о статических веб-страницах, которые запрашиваются один раз и никогда не меняются, становится все более анахроничным.

это уже модель ивентов а не command-request

- Вместо того, чтобы думать о запросах и ответах, нам нужно начать думать о подписке на потоки и уведомлении подписчиков о новых событиях. И это должно происходить на всех уровнях стека - базах данных, клиентских библиотеках, серверах приложений, бизнес-логике, интерфейсах и так далее. Если вы хотите, чтобы пользовательский интерфейс динамически обновлялся в ответ на изменения данных, это станет возможным только в том случае, если мы будем систематически применять потоковое мышление повсюду, чтобы изменения данных могли распространяться на все уровни. Я думаю, что мы увидим гораздо больше людей, использующих модели программирования, ориентированные на потоки, основанные на акторах и каналах, или реактивные фреймворки, такие как RxJava .

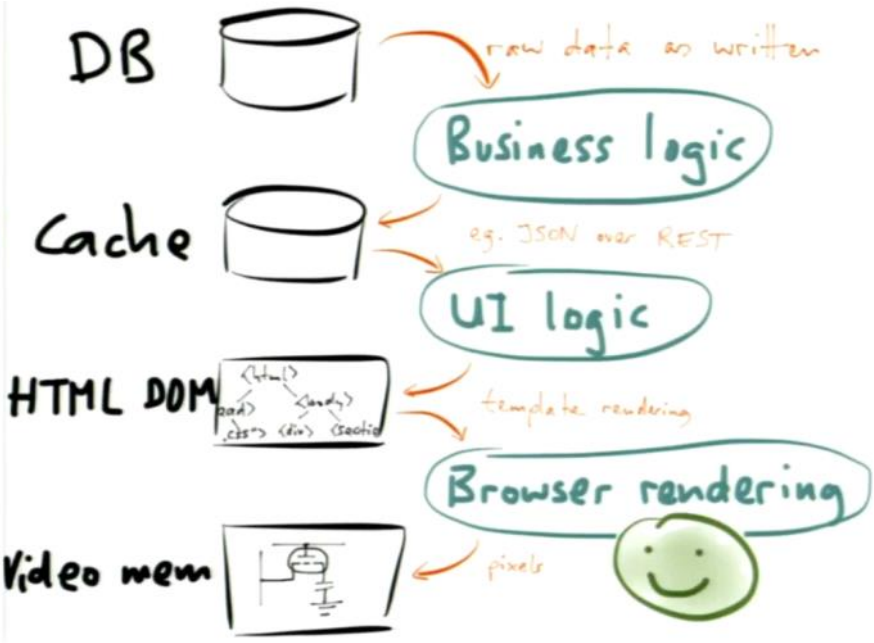
браузер может поддерживать открытое TCP-соединение с сервером

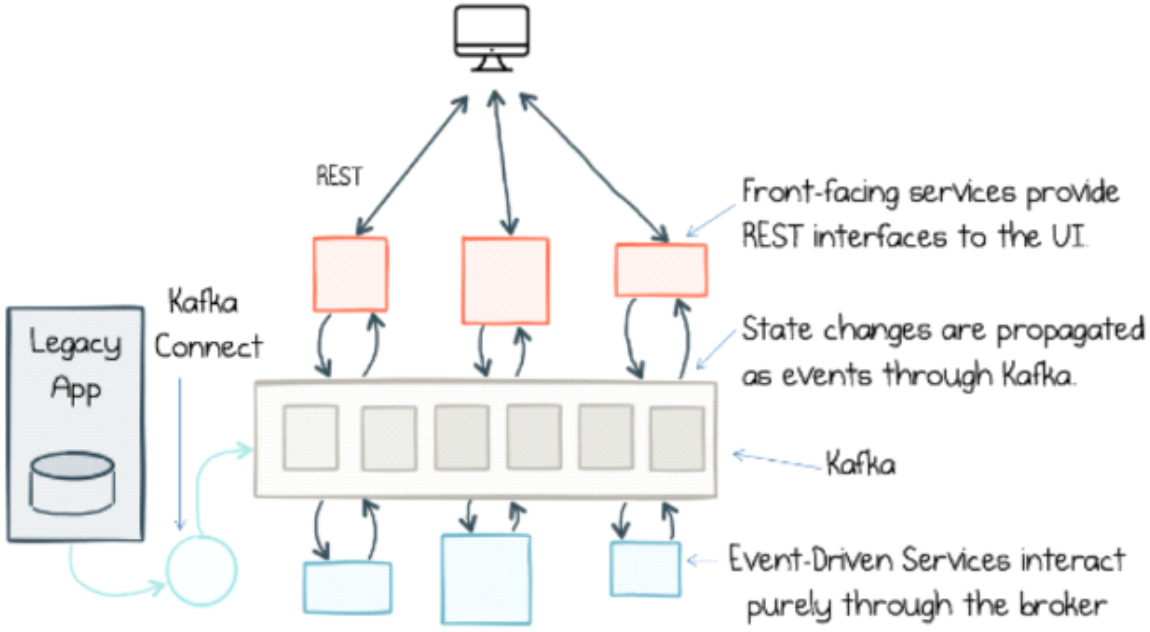
- технология WebSocket предоставляют каналы связи, с помощью которых браузер может поддерживать открытое TCP-соединение с сервером, и последний способен активно передавать сообщения в браузер, пока он остается подключенным.
- Это позволяет серверу активно сообщать конечному клиенту о любых изменениях состояния, хранящегося локально, обновляя состояние на стороне клиента.

front-facing services

1 января 2021 г. 10:01

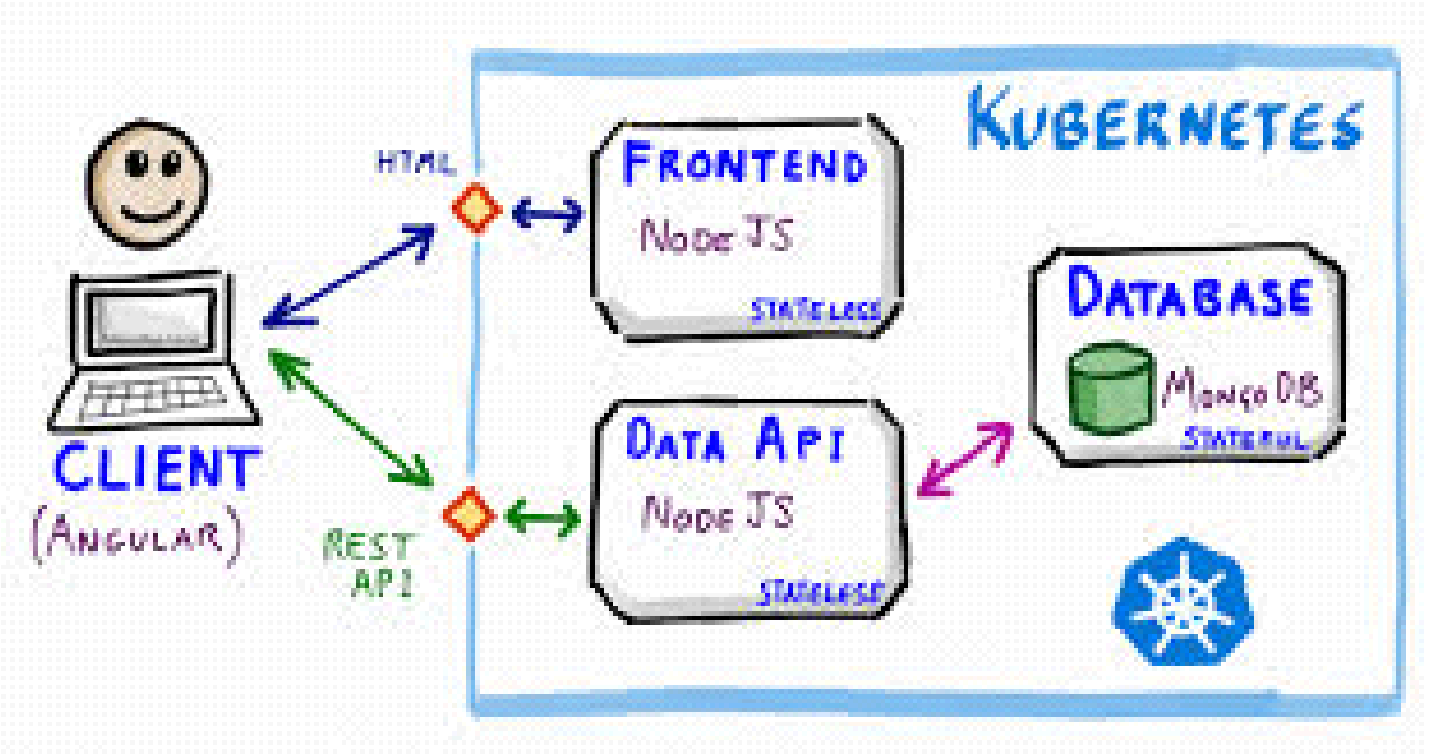
Те UI-микросервисы содержат только UI-логику (и не содержат DDD-бизнес-логики)

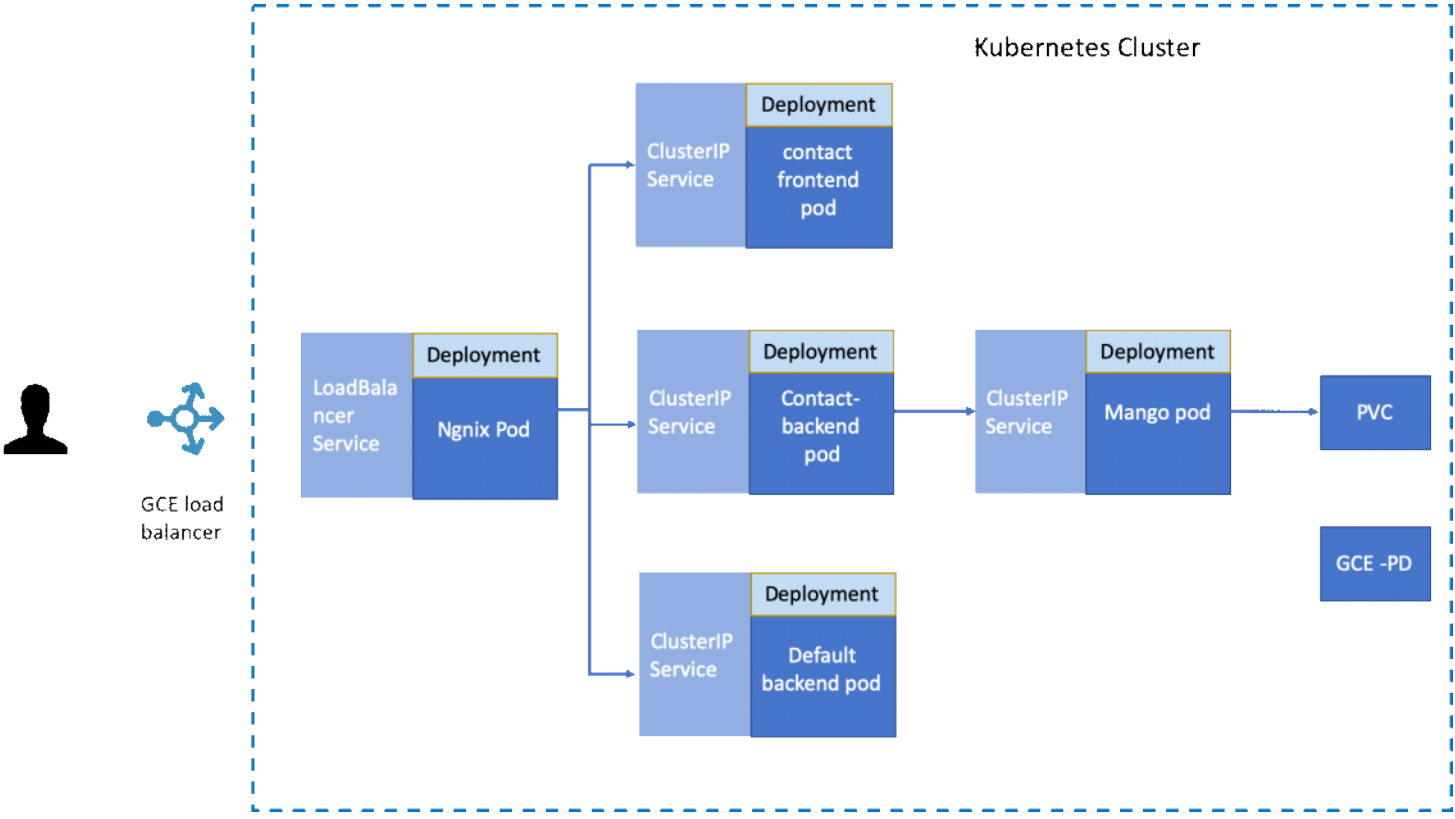


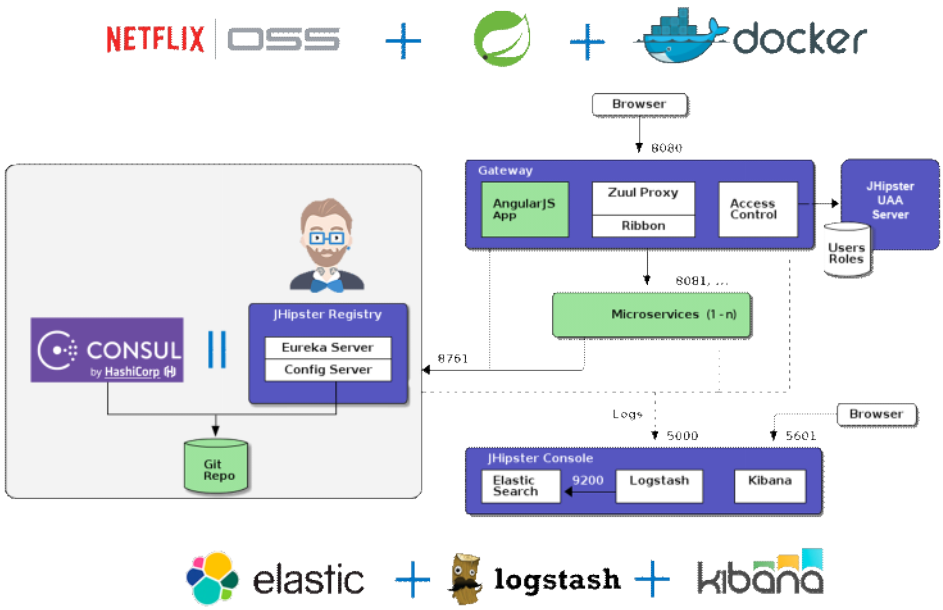


Angular app in Kubernetes

https://archive.azurecitadel.com/cloud-native/kubernetes/

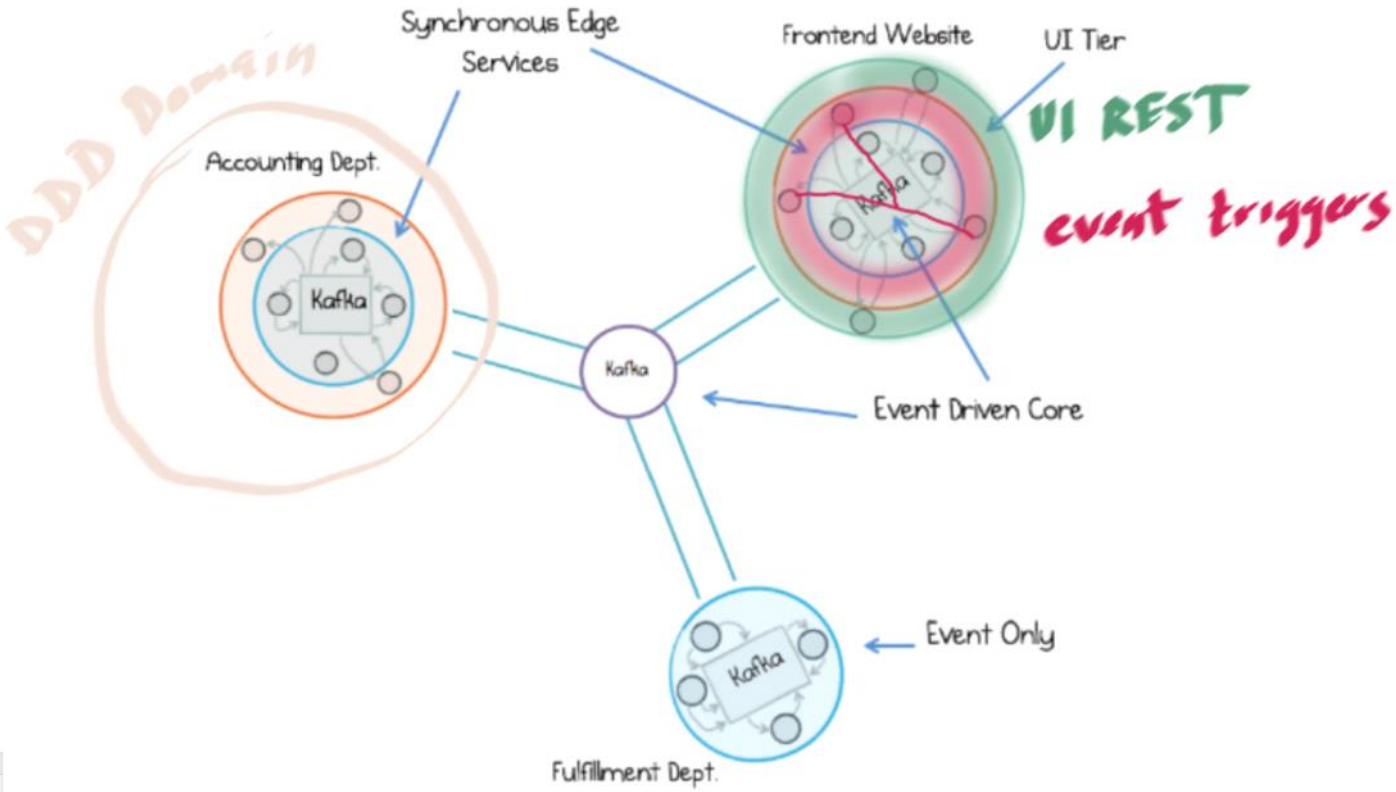

KUBERNETES
HTML
FRONTEND
Node JS
STATELESS
DATABASE
Mongo DB
STATEFUL
CLIENT
(ANGULAR)
REST
API
DATA API
Node JS
STATELESS


Kubernetes Cluster
ClusterIP Service
Deployment
contact frontend pod
LoadBalancer Service
Deployment
Ngnix Pod
ClusterIP Service
Deployment
Contact-backend pod
ClusterIP Service
Deployment
Mango pod
PVC
GCE -PD
ClusterIP Service
Deployment
Default backend pod
GCE load balancer


NETFLIX OSS
docker
Browser
8080
Gateway
AngularJS App
Zuul Proxy
Ribbon
Access Control
JHipster UAA Server
Users Roles
8081, ...
Microservices (1-n)
JHipster Registry
Eureka Server
Config Server
CONSUL
by HashiCorp
8761
Git Repo
Logs
5000
5601
Browser
JHipster Console
Elastic Search
9200
Logstash
Kibana
elastic
logstash
kibana

Clusters of services

1 января 2021 г. 10:09



На рис. 5-10 три отдела общаются друг с другом только посредством событий. Внутри каждого отдела (три больших круга) интерфейсы сервисов используются более свободно, и существуют более детализированные потоки, управляемые событиями, которые стимулируют сотрудничество. Каждый отдел содержит ряд внутренних ограниченных контекстов - небольшие группы служб, которые разделяют модель предметной области, обычно развертываются вместе и тесно взаимодействуют друг с другом. На практике часто существует иерархия совместного использования. На вершине этой иерархии отделы слабо связаны: единственное, что у них общего, - это события. Внутри отдела будет много приложений, и эти приложения будут взаимодействовать друг с другом с помощью механизмов запроса-ответа и событий, как показано на рисунке 5-9.. Каждое приложение может быть составлено из нескольких сервисов, но обычно они будут более тесно связаны друг с другом, разделяя модель предметной области и имея синхронизированные расписания выпуска.

Этот подход, который ограничивает повторное использование в ограниченном контексте, является идеей, которая исходит из проектирования, управляемого предметной областью, или DDD . Одна из важных идей DDD заключалась в том, что широкое повторное использование может быть контрпродуктивным, и что лучший подход - создать границы вокруг областей бизнес-домена и смоделировать их отдельно. Таким образом, в ограниченном контексте модель предметной области является общей, и все доступно для всего остального, но разные ограниченные контексты не используют одну и ту же модель и обычно взаимодействуют через более ограниченные интерфейсы.

см Collapsing CQRS with a Blocking Read паттерн

4 января 2021 г. 20:38

* blocking read pattern(с помощью long polling)

command types

4 февраля 2021 г. 20:44

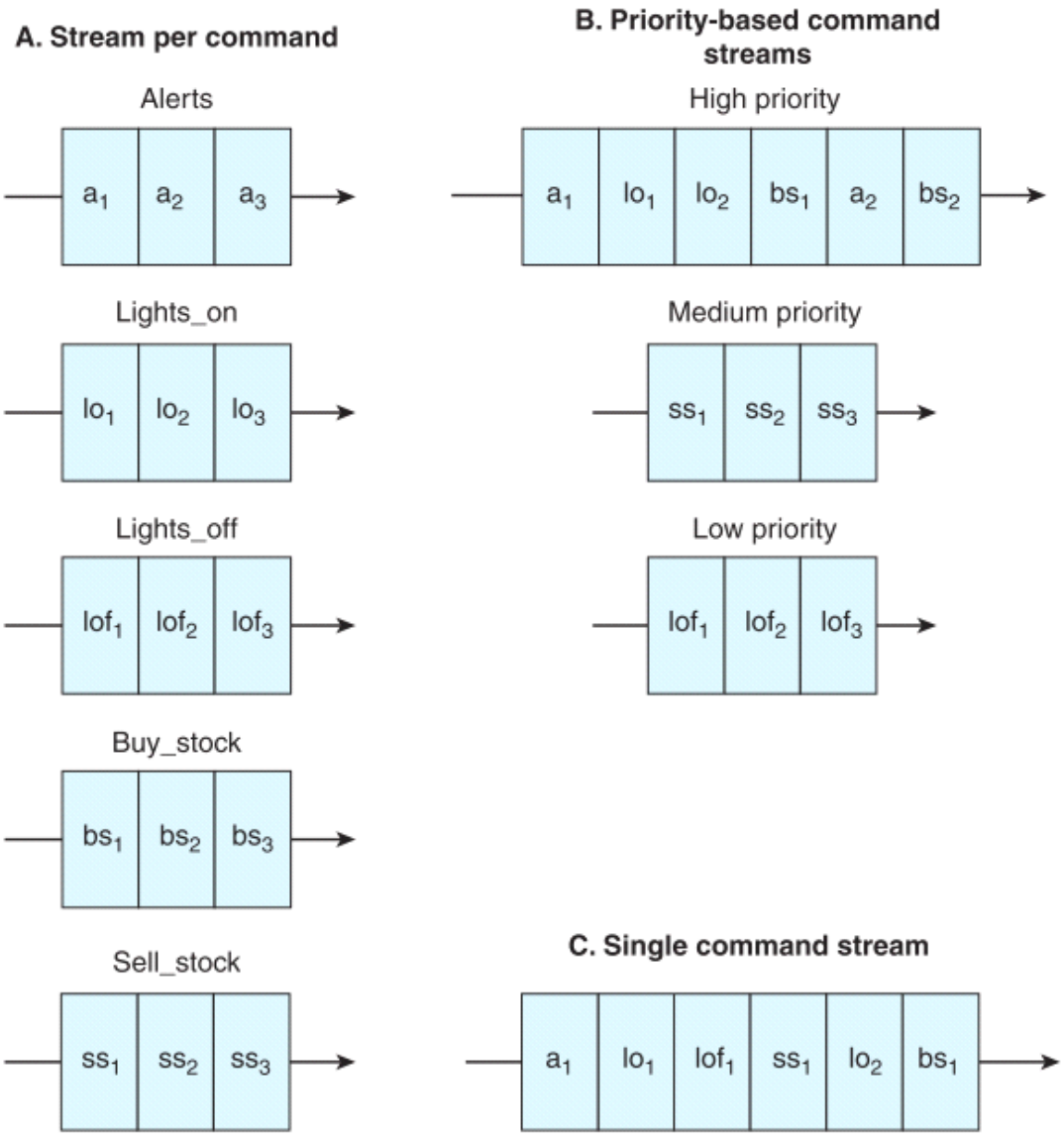


Figure 9.9 We could define one stream for each command type (option A), associate commands to priority-based streams (option B), or use one stream for all commands (option C).

(A) Stream per command (для каждой команды свой топик)

- Have one stream (topic, in Kafka parlance) per command—in other words, hundreds or thousands of streams.

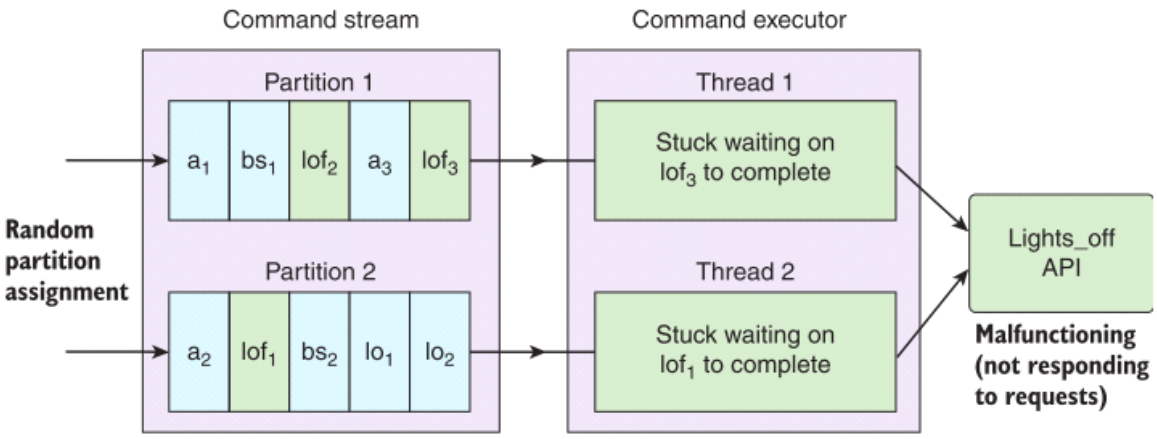


Figure 9.10 With a command stream containing two partitions, a single malfunctioning command can cause both threads within the command executor (one per partition) to get "stuck." Here, the `lights_off` command is failing to execute because of the unavailability of the Lights_off external API.

(B) Priority-based command streams (10 топиков по приоритетам от 1 до 10)

- Make the commands self-describing but write them to say, three or five streams; each stream represents a different command priority.

(C) Single command stream

- Make the commands self-describing and have a single stream. Include a header in each record that tells the command executor what type of command this is

свойства команды

- Commands should be carefully modeled to ensure that they are shrink-wrapped and fully baked. This ensures that decision-making and command execution can stay loosely coupled, with a clear separation of concerns

Shrink-wrapped—

- The command must contain everything that could possibly be required to execute the command; the executor should not need to look up additional information to execute the command.
  команда должна содержать все, что может потребоваться для выполнения команды; исполнителю не нужно искать дополнительную информацию для выполнения команды.
  - Исполнитель команд должен иметь право выбирать, как лучше всего выполнять каждую команду, но исполнителю не нужно искать дополнительные данные или содержать бизнес-логику, специфичную для команды

Fully baked

- —The command should define exactly the action for the executor to perform; we should not have to add business logic into the executor to turn each command into something actionable.
  команда должна точно определять действие, которое должен выполнить исполнитель; нам не нужно добавлять бизнес-логику в исполнитель, чтобы превратить каждую команду во что-то действенное

c

Command hierarchies

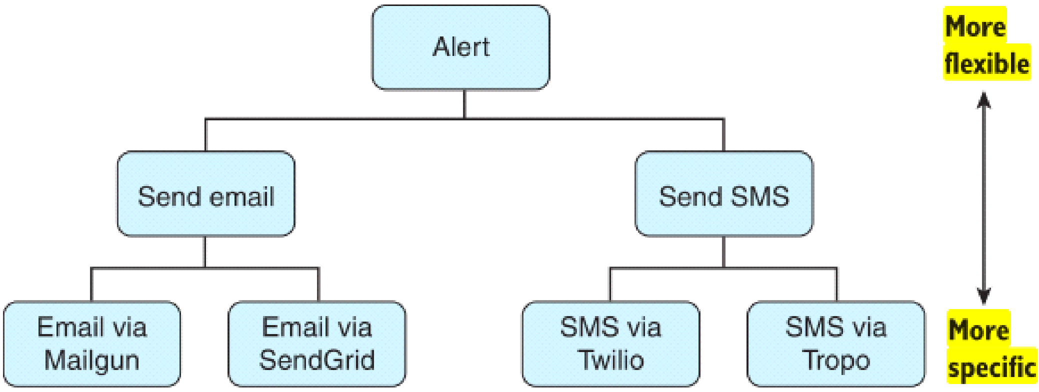


Figure 9.11 In a hierarchy of commands, each generation is more specific than the one above it. Under alert, we have a choice between alerting via email and alerting via SMS; within email and SMS, we have even more-specific commands to determine which provider is used for the sending.

вы можете смоделировать эти более детализированные команды, а также высокоуровневые, чтобы приложение для принятия решений могло быть настолько точным и гибким, насколько ему нравится, в отношении команд, которые оно излучает