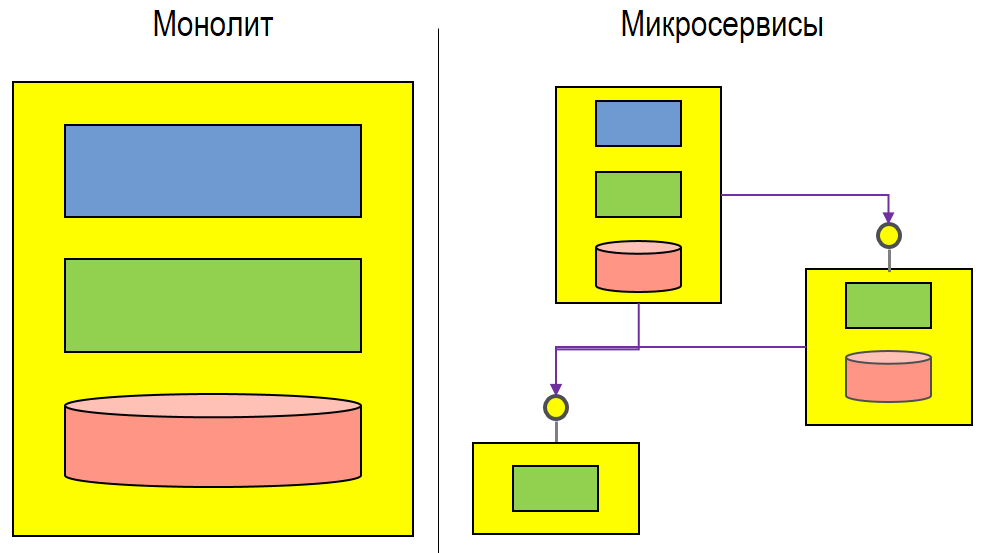
# Что такое микросервисы

### Что такое микросервисная архитектура (MSA)

MSA — [принципиальная](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF) организация распределенной [системы](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) на основе микросервисов и их взаимодействия друг с другом и со средой по сети, а также принципов, направляющих [проектирование](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) архитектуры, её создание и эволюцию.

### Что такое микросервис (MS)

Понять суть микросервиса проще всего на сравнении, или даже противопоставлении его крупному приложению — монолиту.

  
  
Восемь свойств микросервиса:

1. Он небольшой.
2. Он независимый.
3. Он строится вокруг бизнес-потребности и использует ограниченный контекст *(Bounded Context)*.
4. Он взаимодействует с другими микросервисами по сети на основе паттерна *Smart endpoints and dumb pipes.*
5. Его распределенная суть обязывает использовать подход *Design for failure*.
6. Централизация ограничена сверху на минимуме.
7. Процессы его разработки и поддержки требуют автоматизации.
8. Его развитие итерационное.

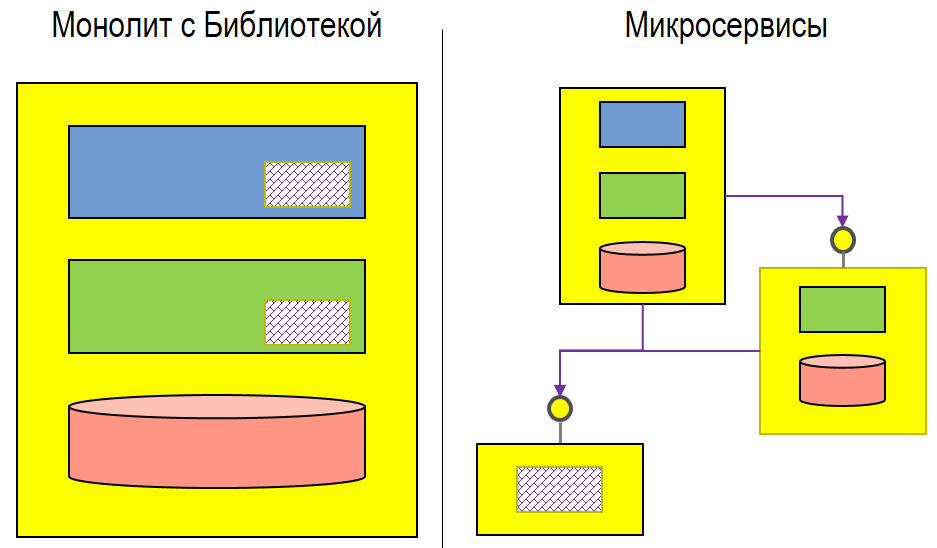
Уже на этом месте те, которые якобы создавали микросервисы в доисторические времена, должны задуматься, действительно ли всё было так продвинуто.

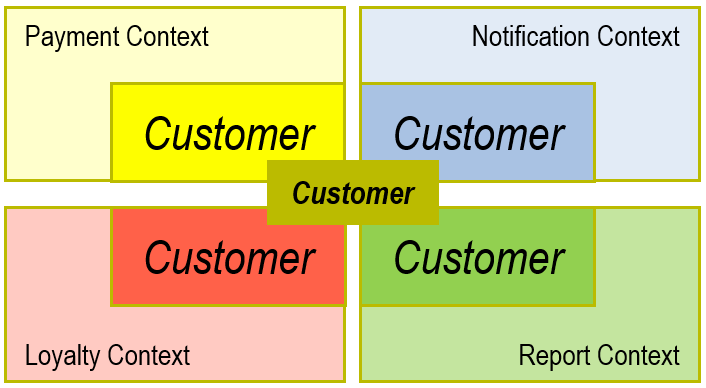
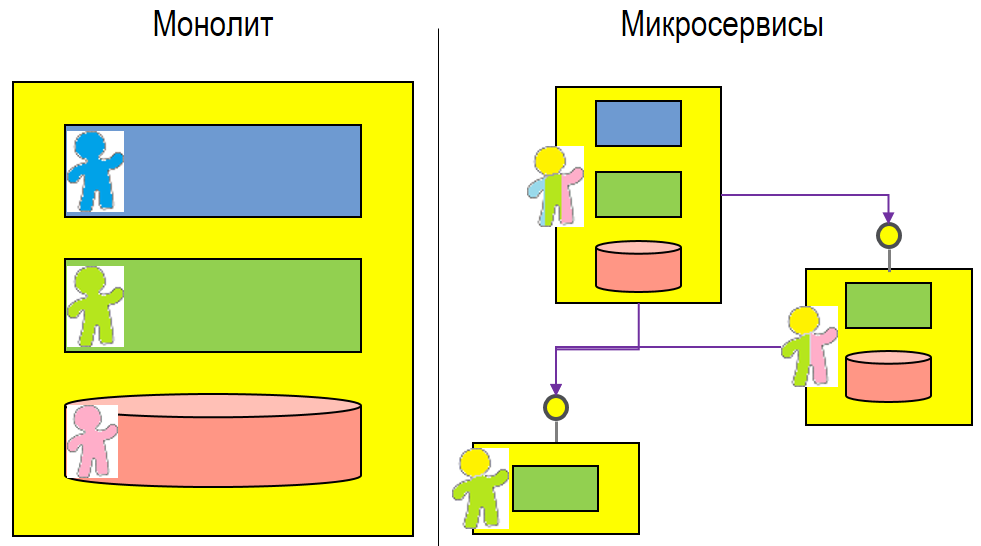
### Небольшой

Что такое «небольшой»? Размер микровервиса должен быть таким, чтобы выполнялось одно из условий:

1. Один сервис может развивать одна команда не более чем из дюжины человек.
2. Команда из полудюжины человек может развивать полдюжины сервисов.
3. Контекст (не только бизнеса, но и разработки) одного сервиса помещается в голове одного человека.
4. Один сервис может быть полностью переписан одной командой за одну Agile-итерацию.

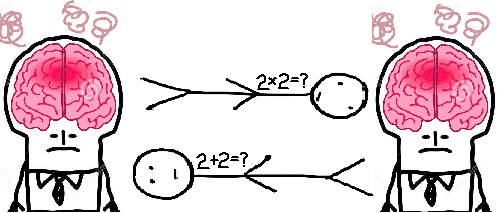
**Независимый**

Микросервисная архитектура — воплощение паттернов High Cohesion и Low Coupling. Всё, что противоречит этому, отвергается беспощадно. В противном случае команду ждут большие проблемы. Микросервис обязан быть независимым компонентом. **Компонент** — это единица ПО, код которой может быть независимо заменен или обновлен.  
Конечно любая мало-мальски серьезная программа пишется с разбиением на компоненты, которые, безусловно, основываются на тех же принципах. Но в монолите общая кодовая база открывает возможности для нарушения низкой связанности. Под такую формулировку компонента подходят и сторонние библиотеки. В то же время методология разбиения на отдельные микросервисы вынуждает придерживаться жесткого их разделения, ведь они должны отвечать более жестким критериям независимости.  
Так, каждый микросервис работает в своем процессе и поэтому должен явно обозначить свой API. Учитывая, что другие компоненты могут использовать только этот API, и к тому же он удаленный, минимизация связей становится жизненно важной.  
  
  
  
Такое разделение дает явный выигрыш с точки зрения независимого развития разных компонентов. И с учетом этого различные языки вводят конструкции, позволяющие явное создание независимых компонентов (например, модули в Java 9), и это перестает быть прерогативой микросервисного подхода. Использование библиотек **не приветствуется**, поскольку так или иначе приводит к зависимостям между микросервисами, но всё же **допускается**. Как правило, это допущение распространяется на инфраструктурные функции вроде логирования, вызова удаленного API, обработки ошибок и тому подобного. Независимость микросервисов позволяет организовать независимый жизненный цикл разработки, создавать отдельные сборки, тестировать и развертывать. Поскольку размер микросервисов невелик, то очевидно, что в крупных системах их будет немало. Управлять ими вручную будет сложно. Поэтому команда обязана иметь приемлемый уровень автоматизации согласно *Continuous integration* и *Continuous Delivery.*  
**Где же микросервис (бизнес-потребность)**

Итак, вы решили спроектировать новый микросервис. Определение его границ — самый важный шаг. От этого будет зависеть вся дальнейшая жизнь микросервиса, и это серьёзно повлияет на жизнь команды, отвечающей за него. Основной принцип определения зоны ответственности микросервиса — сформировать её вокруг некоторой бизнес-потребности. И чем она компактнее, чем формализованней её взаимоотношения с другими областями, тем проще создать новый микросервис. Когда границы микросервиса заданы и он выделен в отдельную кодовую базу, защитить эти границы от постороннего влияния не составляет труда. Далее внутри микросервиса создают свой микромир, опираясь на паттерн «ограниченного контекста». В микросервисе для любого объекта, для любого действия может быть своя интерпретация, отличная от других контекстов.  
  
  
  
Но что делать, если границы оказались неправильными? В этом случае изменение функциональности в новом микросервисе ведет к изменению функциональности в других микросервисах. В результате «поплывут» интерфейсы всех зависимых микросервисов, а за ними интеграционные тесты. Правильные границы микросервиса — это основа здоровой микросервисной архитектуры. Чтобы минимизировать ошибки при определении границ, нужно вначале их продумать. Поэтому оправданным является подход Monolith First, когда вначале систему развивают в традиционной парадигме, а когда появляются устоявшиеся области, их выделяют в микросервисы. Границы тоже могут меняться. Главное, чтобы выигрыш от разбиения превышал сложности пересмотра этих границ. Такой подход к постепенному формированию набора микросервисов похож на итерационное развитие, используемое в Agile, ещё его называют «эволюционным проектированием» *(Evolutionary Design)*. Есть ещё одно интересное следствие создания микросервисов, соответствующее закону Конвея (Conwey Law). Если организация использует монолитное приложение, то оно нарушает соответствие структуре и коммуникациям внутри организации. А команды разработчиков строятся вокруг архитектурных слоев монолита: UI, серверная логика, база данных.  
Микросервисная архитектура приводит IT и бизнес в гармонию, с точки зрения Конвея. Поскольку микросервисы формируются вокруг бизнес-потребностей конкретных бизнес-подразделений, то архитектура предприятия начинает повторять оргструктуру и каналы социальной и бизнес-коммуникации. А команды становятся кроссфункциональными и формируются вокруг этих бизнес-потребностей / бизнес-подразделений.  
  
  
  
Поскольку разные микросервисы получаются независимыми не только логически, но и технологически, а создавать их могут разные команды, ничто не мешает для каждого случая подбирать подходящие языки программирования, фреймворки и даже операционные системы.

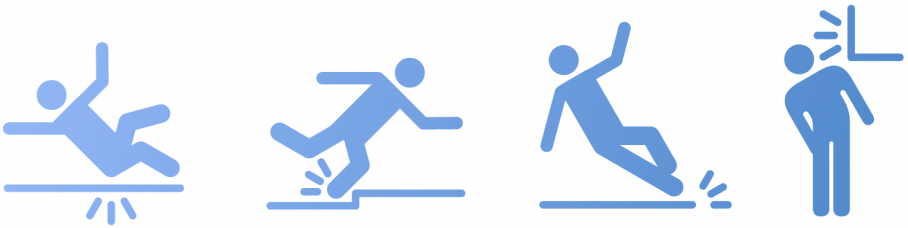
**Интеграция. Smart endpoints and dumb pipes**

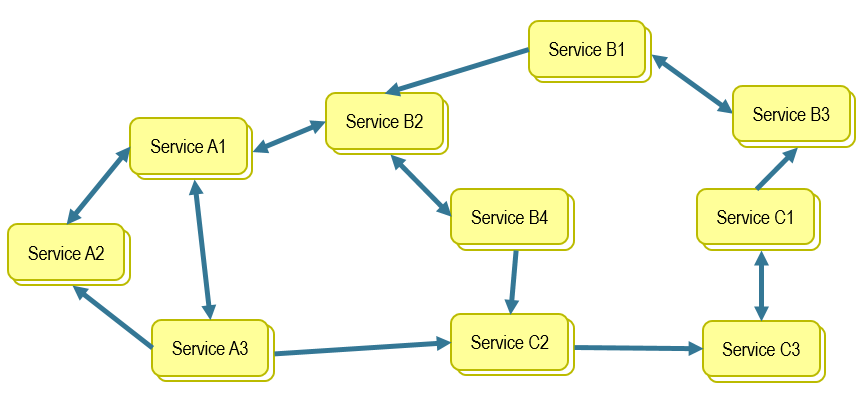
Интеграция микросервисов обходится без ESB, как центрального промежуточного звена. Наверное, комьюнити уже натерпелось от неудачных вариантов реализации этого подхода. То, что были и удачные — не принимается в расчет. Впрочем, ESB ещё и противоречит таким критериям как децентрализация и независимость. Таким образом, сложность интеграции распределяется с центрального звена в виде ESB непосредственно на интегрируемые компоненты: «умные конечные точки».

  
  
Для интеграции, как правило, используются простые текстовые протоколы, основанные на HTTP, чтобы нивелировать возможную технологическую разность микросервисов. REST-подобные протоколы являются практически стандартом. Как исключение, могут использоваться бинарные протоколы типа Java RMI или .NET Remoting.  
Бинарные протоколы гораздо эффективнее. Но, во-первых, появляются технологические ограничения. Во-вторых, на бинарных протоколах сложнее реализовывать шаблон Tolerant Reader, сохраняя эффективность. В-третьих, опять появляется зависимость провайдера и потребителей, поскольку они оперируют одними и теми же объектами и методами, то есть связаны по кодовой базе. Другая отличительная черта взаимодействия микросервисов — синхронные вызовы не приветствуются. Рекомендуется использовать один синхронный вызов на один запрос пользователя, или вообще отказаться от синхронных вызовов. И еще пара замечаний.

1. Основной сложностью разбиения монолита на микросервисы является не определение их границ. Они уже должны сформироваться и устояться. Сложность заключается в том, что локальные вызовы становятся удаленными. А это влияет не только на организацию вызовов, но и на стиль взаимодействия, так как частые вызовы уже не подходят. Скорее всего, надо пересматривать сам API, делать его более крупным, а, как следствие, пересматривать логику работы компонентов.
2. Поскольку асинхронное событийное взаимодействие — практически стандарт в микросервисной архитектуре, то надо разбираться в создании событийной архитектуры (Event Driven Architecture), а сами микросервисы должны соответствовать требованиям Reactive.

***Design for failure* для распределенной системы**

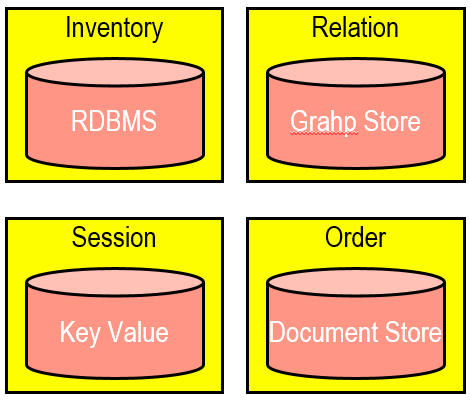
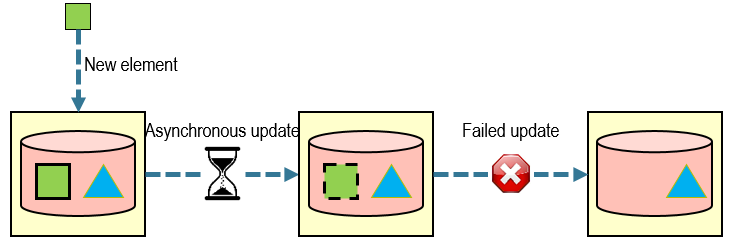
Одно из наиболее критичных мест в микросервисной архитектуре — необходимость разрабатывать код для распределенной системы, составные элементы которой взаимодействуют через сеть. А сеть ненадежна по своей природе. Сеть может просто отказать, может работать плохо, может вдруг перестать пропускать какой-то тип сообщений, потому что изменились настройки файрвола. Десятки причин и видов недоступности.  
  
  
  
Поэтому микросервисы могут вдруг перестать отвечать, могут начать отвечать медленнее, чем обычно. И каждый удаленный вызов должен это учитывать. Должен правильно обрабатывать разные варианты отказа, уметь ждать, уметь возвращаться к нормальной работе при восстановлении контрагента. Дополнительный уровень сложности привносит событийная архитектура. А отладку такой системы — не одного микросервиса, а системы, где много потоков разнонаправленных неупорядоченных событий — даже трудно представить. И даже если каждый из микросервисов будет безупречен с точки зрения бизнес-логики, этого мало.

  
  
И поскольку сложность таких систем очень высока, то проблему решают так:

* Не доводят систему до состояния «без сучка без задоринки». Это очень дорого. Конечно, это не значит, что система валится от первого дуновения. Она просто отвечает необходимым нефункциональным требованиям. Но в ней могут присутствовать ошибки, незначительно влияющие на ее устойчивость и производительность.
* С другой стороны вкладываются в инфраструктуру, которая помогает быстрее устранять нештатные ситуации. Должно быть полное покрытие кода unit тестами, интеграционными и тестами производительности. Должен быть интеллектуальный мониторинг, который не только моментально показывает неработающие места, но и сигнализирует об ухудшении состояния системы с прогнозированием возможных сбоев. Должно быть продвинутое распределенное логирование, позволяющее оперативно проводить расследования. И часто по результатам исправляются скрытые ошибки.

### Децентрализация данных

Еще один из важнейших элементов в парадигме микросервисов.

Каждому микросервису по своей базе данных! На самом деле и в монолите можно побороться за изолированность компонентов, например, на уровне серверного кода. Если время от времени изоляция даёт течь, современные инструменты предлагают продвинутые инструменты рефакторинга. Помимо изолированности есть и побочные плюсы. Например, легче реализовать *Polyglot Persistence*, когда база подбирается под конкретные цели. Ничто не мешает делать это и без микросервисов, и так часто делают. Но всё же в одном случае это закон, в другом — исключение.  
  
  
  
Много баз, много контекстов, как их все согласовать? Старая техника распределенных транзакций сложна и обладает низкой скоростью. Возможно это иногда можно пережить. А вот необходимость синхронного взаимодействия нескольких микросервисов не может устраивать, и это не побороть. Проблема решается нетрадиционно для монолита: отказом от постоянной согласованности данных. Добро пожаловать в мир *Eventual consistency*. На первых порах это вызывает волну «справедливого» гнева. Но если разобраться, то нужна ли повсеместно немедленная согласованность данных по окончании транзакции? При детальном рассмотрении значительную часть случаев можно отбросить. Где возможно, заменяют одну распределённую транзакцию серией локальных с компенсационными механизмами.   
  


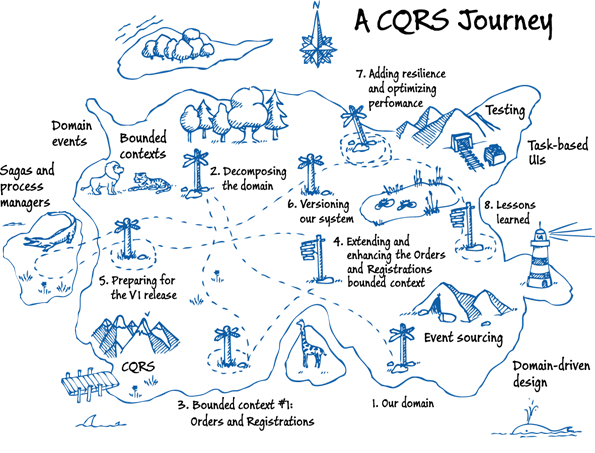
### Монолит против микросервисов

Микросервисный подход несет довольно много проблем. Их найти не трудно и каждый может поупражняться. Например, организационные вопросы. Как удержать в согласованном по версиям состоянии сотню микросервисов, которые еще и постоянно и непредсказуемо редеплоятся. А доступ к средам у каждого инженера каждой команды? Какая команда напишет интеграционные тесты? И если кто-то согласится, то попробуй еще их напиши для такой запутанной конфигурации. А если возникает ошибка, то чья она? Только той команды, у которой сломалось? Как не узнать вечером в пятницу, что версия API N-го сервиса, которой вы пользуетесь, вдруг стала deprecated? Да, это действительно проблемы. Но команды, которые практикуют Agile и DevOps, уже знают решение. Поэтому начинать путь к микросервисной архитектуре стоит с внедрения этих практик. Кроме организационных есть и чисто архитектурные. Как перейти от монолита, где всё синхронно, согласованно и едино, к распределенной событийной архитектуры, основанной на множестве мелких элементов, в которой надо учитывать возможную неконсистентность данных? Одного этого достаточно, чтобы задуматься: а стоит ли игра свеч? На этом фоне, например, падение скорости обработки одного запроса кажется мелочью. Хотя бы работает!

  
Тогда зачем? Если у вас нет проблем с вашим «монолитом», то не надо их искать. Но если проблемы есть, то посмотрите на плюсы MSA, и возможно она спасет вас.  
Разбиение на независимые компоненты даёт безусловные и неоспоримые преимущества: легкое понимание контекста, гибкость развития, управления и масштабирования. Независимость и небольшой размер дают и неожиданные плюсы с точки зрения инфраструктуры. Вам теперь не нужна монструозная машина за 100500 долларов. Микросервисы можно устанавливать на обычные дешевые машинки. И окажется, что даже все вместе они будут стоить на порядок меньше, но работать эффективнее той самой супермашины, на которую у вас в организации, наверняка, молятся и сдувают с неё пылинки. Здесь уместен другой лозунг от популиста. Хотя, как и предыдущий, он вполне серьезен.  
**Каждому микросервису по своему серверу!**

Посмотрим на лидеров IT-индустрии: Amazon, Netflix, Google и другие показывают впечатляющие результаты. Их гибкость и скорость вывода новых продуктов поражают. Поэтому игра точно стоит свеч! Здесь уместно вспомнить, что в упомянутых организациях команд «уровня бог» не одна и не две. Им сложности микросервисной архитектуры вполне по зубам. И если предложить создать монолит, то они и его сделают так, что он будет сверкать путеводной звездой. А, например, Amazon вполне себе работал на монолите, уже будучи гигантом и имея миллиардные обороты. Сайт газеты Guardian до сих пор, а возможно и навсегда, базируется на микросервисах вокруг монолита. Это говорит о том, что значительная часть задач успешно, а зачастую и легче, решается без привлечения микросервисов.

# . Шаблоны проектирования CQRS и Event Sourcing



## CQRS – Command and Query Responsibility Segregation паттерн

CQRS расшифровывается как Command Query Responsibility Segregation (разделение ответственности на команды и запросы). В 1980 Бертран Мейер сформулировал термин CQS. В начале двухтысячных Грег Янг расширил и популяризовал эту концепцию к CQRS. CQRS предлагает разделять операции чтения и записи на отдельные типы операций Query и Commands.

* Command ориентированы на задачи, а не на данные. ("Забронировать номер в отеле", а не установить для ReservationStatus значение "зарезервировано" ).
* Command может помещаться в очередь для асинхронной обработки, а не обрабатываться синхронно.
* Query никогда не должен изменять базу данных. Query возвращает DTO, который не инкапсулирует знания предметной области.

Схематически работу клиента по CQRS модели можно отобразить следующим образом:

Наличие отдельных моделей запросов и команд упрощает проектирование таких систем как независимых друг от друга. Однако один из недостатков заключается в том, что код CQRS не может автоматически формироваться из схемы базы данных с помощью ORM или подобных механизмов. Для дополнительной изоляции часто физически разделяют данные для чтения и данные для записи. Как это описано на диаграмме выше. В этом случае в БД для чтения можно оптимизировать ее работу так, чтобы максимально эффективно выполнять запросы. К примеру заюзать materialized view, чтобы не использовать сложные операции join'ов или сложные связи. Вы можете использовать в том числе другой тип хранилища данных. Например, база данных для записи останется [реляционной](https://bool.dev/blog/tags/sql), а для чтения вы можете применять [NoSQL](https://bool.dev/blog/tags/nosql) или наоборот, в зависимости от бизнес задач. Если вы пошли по пути отдельных БД для чтения и записи, они должны поддерживать синхронизацию. Обычно это реализуют с помощью событий при каждом обновлении базы данных. Обновление базы данных и публикации события должны выполняться в рамках одной транзакции.

### 👍 Преимущества CQRS

* **Независимое масштабирование**. CQRS позволяет раздельно масштабировать рабочие нагрузки чтения и записи, снижая риск конфликтов блокировки.
* **Оптимизированные схемы данных**. Для query применить схему, оптимизированную для запросов, а commands — другую схему, оптимизированную для обновлений.
* **Безопасность**. Разделение операций позволит настроить более гибкую систему доступа.
* **Разделение проблем**. Разделение операций позволяет получить более гибкие и простые в обслуживании классы.
* **Более простые запросы**. Сохраняя в базе данных для чтения materialized view, вы предотвратите использование сложных запросов и join'ов.
* **Не требует 2 хранилища данных**. Отдельные хранилища для query и command это одна из реализаций, а не обязательное требование

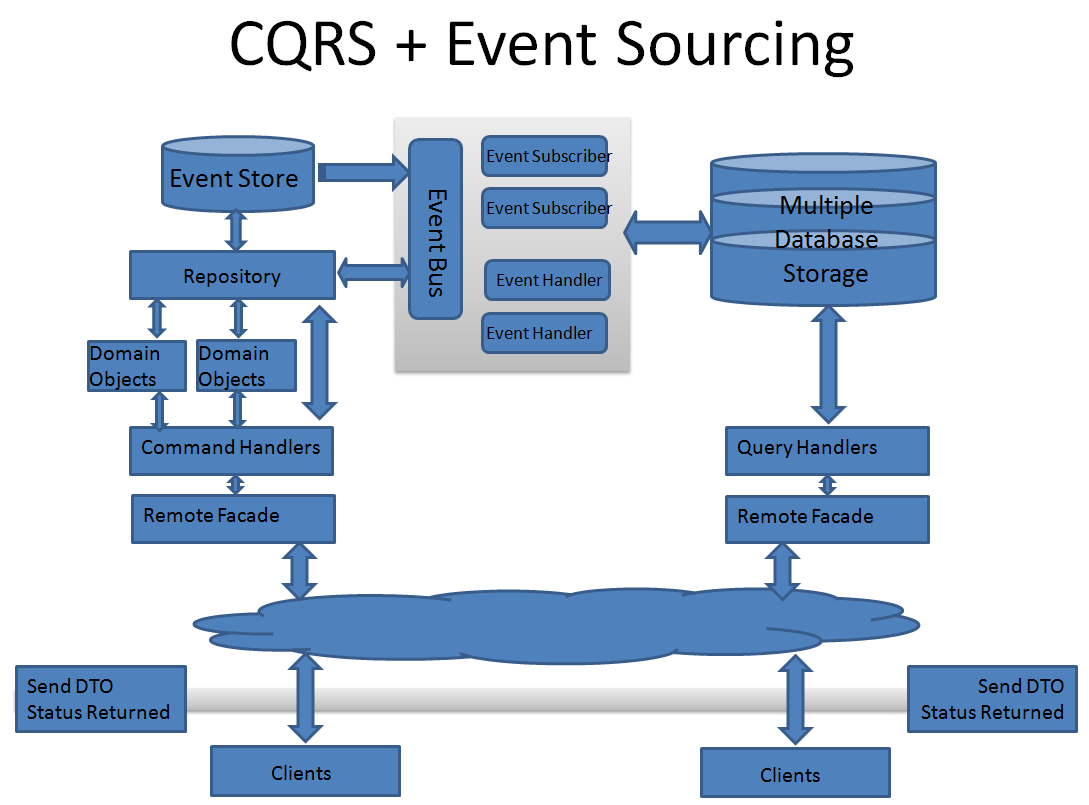
### 👎 Недостатки CQRS

* **Сложность.** Основная идея CQRS звучит просто. Но ее реализация может привести к усложнению проекта приложения, особенно если реализовывать его в связке с Event Sourcing.
* **Обмен сообщениями**. Сама по себе модель CQRS не требует месседжинга, но месседж брокеры часто применяются для обработки команд и публикации событий. Это означает, нужно будет реализовывать обработку сбоев и дубликатов при передаче сообщений.
* **Eventual consistency**. Если вы разделите базы данных для чтения и записи, в базе данных для чтения могут оставаться устаревшие данные. БД для чтения должна быть up to date, чтобы отражать изменения из БД для записи, и может быть трудно трекать, когда пользователь сделал запрос на основе устаревших данных с БД для чтения.

## Event Sourcing

**Event sourcing** (источники событий, регистрация событий, генерация событий) — это архитектурный паттерн, в котором все изменения, вносимые в состояние приложения, сохраняются в той последовательности, в которой они были выполнены. Эти записи служат как источником для получения текущего состояния, так и audit-log'ом того, что происходило в системе. Event sourcing способствует децентрализованному изменению и чтению данных. Такая архитектура хорошо масштабируется и подходит для систем, которые уже работают с событиями или подходят для миграции на такую архитектуру.

Event sourcing идет в ногу с CQRS. "DELUXE" cхема связки CQRS + Event Sourcing выглядит следующим образом:



У Event sourcing есть следующие преимущества и недостатки:

### 👍 Преимущества Event Sourcing

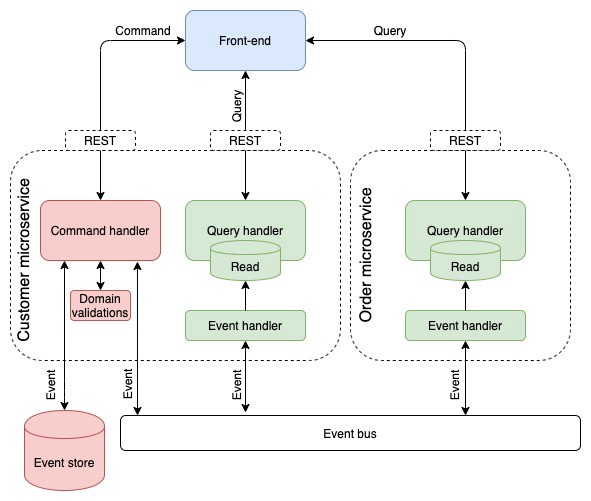
* События immutable, и их можно сохранить с помощью append-only операции.
* События могут раниться на фоне.
* Event sourcing может помочь в предотвращении конфликтов, вызванных параллельными апдейтами, тк исключает необходимость непосредственного обновления объектов в Data store. Однако доменная модель должна уметь себя защищать от запросов, которые могут вызвать несогласованное состояние.
* Append-only storage предоставляет audit log, который можно использовать для мониторинга событий, произошедших в Data store, повторного создания текущего состояния в виде materialized view или проекций путем воспроизведения событий в любое время, а также упрощения тестирования и отладки системы.
* Каждое событие могут обрабатывать несколько задач. Это обеспечивает простую интеграцию с другими службами и системами, которые только слушают новые события, вызванные data stor'ом. Однако event sourcing events зачастую являются низкоуровневыми, из-за чего может потребоваться создание определенных событий интеграции.

### 👎 Недостатки CQRS

* Самые большие сложности обычно связаны с перестроением мышления разработчиков. Разработчики должны забыть про обычные CRUD-приложения и хранилища сущностей. Теперь основной концепцией становятся события.
* При Event Sourcing много сил тратится на моделирование событий. После сохранения событий в сторедж они должны быть immutable, иначе история и состояние могут быть повреждены или искажены. Event Log — это исходные данные, а это значит, что необходимо очень внимательно следить за тем, чтобы они содержали всю информацию, необходимую для получения полного состояния системы на определенный момент времени. Также необходимо учитывать, что события могут интерпретироваться повторно, поскольку система (и бизнес, который она представляет) со временем изменяются.
* Для простой бизнес логики переход на Event Sourcing может быть довольно легким, но для более сложных может стать проблемой (особенно с большим количеством зависимостей и отношений между сущностями). Так же могут возникнуть сложности интеграции с внешними системами, которые не предоставляют данные на определенный момент времени.
* Event Sourcing может работать хорошо в больших системах, так как паттерн «Event Log» естественным образом масштабируется горизонтально. Например, event log одной сущности необязательно должен физически находиться вместе с журналом событий другой сущности. Однако, такая легкость масштабирования приводит к дополнительным проблемам в виде решения проблем и реализацией eventual consistency.
* Важно учитывать структуру событий. Структура событий может измениться в какой-то момент, например набор полей. Могут возникнуть ситуации, когда исторические события должны быть обработаны текущей бизнес-логикой. И наличие расширяемой схемы событий поможет в будущем при необходимости отличать новые события от старых. Периодические снапшоты также помогают отделить серьезные изменения структуры событий.

### Имплементация

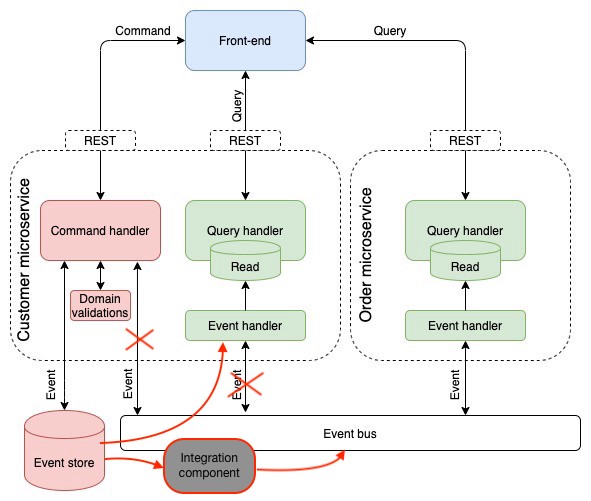
начнем с такой диаграммы:



Это не то, как вы используете Event Sourcing.

Однако важная часть состоит в том, что события предметной области живут внутри контекста, потому что они являются неотъемлемой частью модели предметной области. Следовательно, события предметной области редко открываются внешнему миру как есть, поскольку они связывают модель предметной области с контрактом службы, поскольку они становятся контрактом. Это последнее, что мы хотим сделать. Разделение событий, отправляемых во внешний мир, публикация этих событий в виде контракта на обслуживание и поддержание их стабильности - это правильное решение. Еще одна проблема заключается в том, что возможности Event Sourcing в устранении двухфазных коммитов здесь теряются. На диаграмме видно, что сохранение события и его публикация в шине - это две разные операции. Мы всегда сохраняем события в хранилище и используем хранилище как источник событий. Это возможно с продуктами, которые поддерживают потоки событий в реальном времени, например [EventStoreDB](https://www.eventstore.com/" \t "_blank). Это также лишает нас необходимости в каком-либо event bus внутри службы для построения модели чтения, которая отображена зеленым на диаграмме. Также важно, что чаще всего проекция на стороне чтения должна обрабатывать события по порядку, и ни один продукт, называемый шиной, не может этого сделать.

Вот исправленная, упрощенная схема того, как это должно быть:



Источники событий заботятся о логике записи, где события сохраняются в хранилище событий и транслируются с использованием подхода публикации/подписки, чтобы информировать микросервисы об изменении данных.

### Когда следует использовать Event Sourcing

* Когда в данные необходимо записать намерение, цель или причину. Например, изменения в сущности клиента можно записать как ряд определенных типов событий, таких как Возвращение к исходному, Закрытая учетная запись или Недействительные.
* Когда очень важно свести к минимуму или полностью избежать конфликта операций обновления данных.
* Если требуется записывать происходящие события и иметь возможность воспроизвести их для восстановления определенного состояния системы, отката изменений или сохранения истории и audit-log. Например, если задача включает несколько шагов, необходимых для восстановления обновлений и последующего воспроизведения некоторых действий для восстановления согласованного состояния данных.
* Когда использование событий представляет собой стандартную возможность операции приложения и требует некоторой дополнительной разработки или усилий в отношении реализации.
* Если нужно разбить процесс ввода или обновления данных из задач, необходимых для применения этих действий. Это может быть в целях улучшения производительности пользовательского интерфейса или распределения событий в другие прослушиватели, выполняющие определенные действия при возникновении событий. Например, интеграция платежной системы с веб-сайтом о расходах требуется для того, чтобы события, вызванные с помощью хранилища событий в ответ на обновления данных, реализованные для веб-сайта, использовались как веб-сайтом, так и платежной системой.
* Если необходима гибкость для изменения формата материализованных моделей и данных сущности при изменении требований или —использовании в сочетании с CQRS, необходимо адаптировать модель чтения или представления с данными.
* Если используется в сочетании с CQRS, eventual consistency допустима при обновлении модели чтения или допустимо влияние на производительность при восстановленных сущностях и данных из потока события.

### Когда не следуюет использовать Event Sourcing

* Для небольших или простых доменов, систем, которые обычно хорошо взаимодействуют со стандартными механизмами управления данных CRUD.
* Систем, где для представления данных требуются согласованность и обновления в режиме реального времени.
* Систем, где для действий отката и воспроизведения не требуются определенные функции, история и audit-log.
* Систем, где имеется незначительный конфликт обновлений в базовых данных. Например, это системы, которые преимущественно добавляют данные, а не обновляют их.

## Выводы

CQRS и Event Sourcing — это интересный подход, имеющий свои преимущества. Одно из которых — упрощение расширения системы в будущем. Поскольку event log хранит все события, то их можно использовать во внешних системах. Довольно легко интегрироваться через добавление новых обработчиков событий.

Помните, что прежде чем интегрировать у себя CQRS и Event Sourcing нужно определить действительно ли вам это нужно, принцип KISS еще никто не отменял.

Однако, ограничения, связанные со сложностью предметной области, требованиями к согласованности и доступности данных, а также увеличение объема хранимых данных и масштабируемость в долгосрочной перспективе — все это необходимо учитывать при выборе этих паттернов. Так же важно уделять внимание разработчикам, которые будут разрабатывать и поддерживать такую ​​систему на протяжении всего SDLC.

# Принципы организации данных.

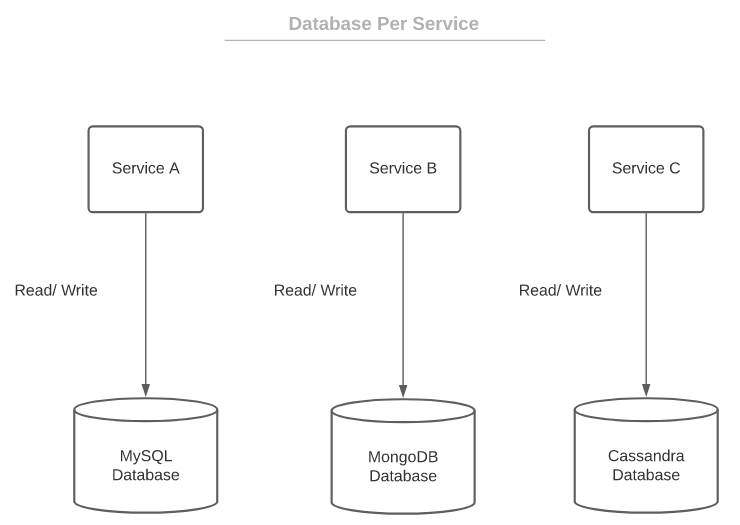
Существует два основных варианта организации баз данных при использовании архитектуры микросервисов:

1. База данных для каждой службы
2. Общая база данных

В этом разделе мы опишем первый.

### Основы

По определению, **микросервисы должны быть слабо связанными, масштабируемыми и независимыми с точки зрения разработки и развертывания**. Поэтому предпочтительным подходом является база данных для каждой службы, поскольку она идеально соответствует этим требованиям. Давайте посмотрим, как это выглядит:



Идея проста. У каждого микросервиса есть свое собственное хранилище данных (целая схема или таблица). Другие службы не могут получить доступ к хранилищам данных, которые им не принадлежат. Такое решение приносит много преимуществ.

Прежде всего, изменения в отдельной базе данных не влияют на другие сервисы. Таким образом, в приложении нет ни одной точки отказа. Так сказать, приложение более устойчиво.

Во-вторых, **отдельные хранилища данных легче масштабировать**. Более того, данные домена инкапсулируются в микросервис. Таким образом, легче понять сервис с его данными в целом. Это особенно важно для новых членов команды разработчиков. Им потребуется меньше времени и усилий, чтобы полностью понять область, за которую они отвечают.

Наконец, с базой данных для каждой службы мы можем использовать сохранение полиглота. Это означает, что мы можем использовать разные технологии баз данных для разных микросервисов. Таким образом, одна служба может использовать базу данных SQL, а другая - базу данных NoSQL. Эта функция позволяет использовать наиболее эффективную базу данных в зависимости от требований к сервису и функциональности.

### Недостатки

Несмотря на все эти преимущества, существуют некоторые серьезные недостатки и проблемы, связанные с подходом "база данных для каждого сервиса". Как мы упоминали ранее, каждый микросервис может напрямую обращаться только к своему собственному хранилищу данных. Поэтому службам необходим способ связи для обмена данными. Таким образом, каждая служба должна предоставлять понятный API.

Следовательно, **существует необходимость в механизме защиты от сбоев в случае сбоя связи**. Допустим, мы отправляем запросы на оплату из службы A в службу B. Служба A ожидает ответа, чтобы выполнить соответствующее действие на основе результата. При этом служба B отключается. Нам нужно разобраться с ситуацией и сообщить службе A о результате, когда B снова подключится к сети. Здесь может помочь механизм автоматического отключения.

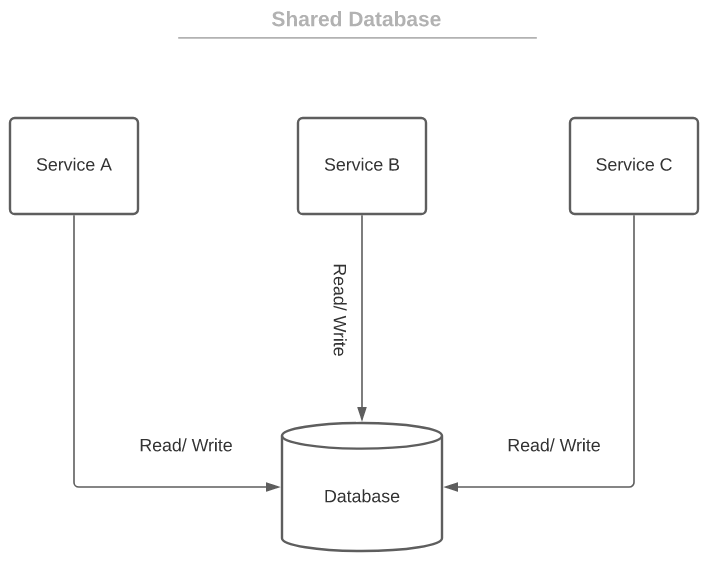
Следующая важная проблема - транзакции. Охват транзакций между микросервисами может негативно повлиять на согласованность и атомарность. Аналогичный недостаток связан со сложными запросами. Не существует простого способа выполнения запросов на объединение в нескольких хранилищах данных.

Наконец, операции, связанные с данными, охватывающие микросервисы, могут быть сложными для отладки в случае каких-либо проблем.

## Общая база данных

Общая база данных считается анти-шаблоном. Хотя это спорно. Дело в том, что **при использовании общей базы данных микросервисы теряют свои основные свойства: масштабируемость, устойчивость и независимость**. Поэтому совместно используемая база данных редко используется с микросервисами.

Когда общая база данных кажется лучшим вариантом для проекта микросервисов, мы должны переосмыслить, действительно ли нам нужны микросервисы. Возможно, монолит был бы лучшим выбором. Давайте посмотрим, как выглядит подход к общей базе данных:



Варианты использования общей базы данных с микросервисами не являются распространенными. Примером может быть временное состояние при переносе монолита на микросервисы. Основным преимуществом общей базы данных по сравнению с сервисом является управление транзакциями. Нет необходимости распределять транзакции по службам.

Более того, данные полностью ограничены, и соответствующие излучения сохраняются. Впоследствии избыточность уменьшается. Мы можем легко выполнять сложные запросы с помощью объединений.

Еще одна важная вещь - нет необходимости обмениваться сохраненными данными между микросервисами. Таким образом, API упрощен, и нет проблем с согласованностью данных и состояния в случае сбоя связи. Однако есть некоторые серьезные недостатки.

**Микросервисы с общими базами данных нелегко масштабировать**. Более того, база данных будет единственной точкой отказа. Изменения, связанные с базой данных, могут повлиять на несколько служб. Кроме того, микросервисы не будут независимыми с точки зрения разработки и развертывания, поскольку они подключаются к одной и той же базе данных и работают с ней.

Этот шаблон можно рассматривать в таких случаях, как:

* существующее хранилище данных должно быть сохранено
* существующая кодовая база уровня данных не должна изменяться
* транзакции имеют решающее значение для приложения

## Шаблоны, связанные с данными

Существует множество шаблонов, которые используются для управления данными в архитектуре микросервисов. В этом разделе мы кратко представим основные из них.

### Шаблон Saga

Ранее мы упоминали, что распределение транзакций между микросервисами может быть проблематичным. Проще говоря, транзакция будет успешной, только если все связанные службы успешно выполнят свою часть. В случае сбоя в одной службе вся транзакция должна завершиться неудачей. Более того, в этом случае службы, которые уже внесли свой вклад, должны откатить изменения.

В общем, за это отвечает шаблон saga.**Шаблон Saga - это последовательность локальных транзакций, которые представляют собой единую распределенную транзакцию**. Каждая служба выполняет локальную транзакцию. Если локальная транзакция завершается успешно, публикуется событие или сообщение, которое запускает следующую локальную транзакцию в последовательности. В случае сбоя saga предоставляет компенсирующие транзакции, которые откатывают изменения.

Существует два типа реализации шаблона saga :

* Управление – центральный контроллер (orchestrator) управляет всеми взаимодействиями между микросервисами
* Хореография – децентрализованный метод трансляции событий

### CQRS

CQRS (Command Query Responsibility Segregation) помогает с другой важной функцией: запрашивать связанные данные из нескольких хранилищ данных. Более того, это упрощает сложность бизнес-логики за счет разделения проблем. Кроме того, это помогает с масштабируемостью микросервисов.

Идея проста. Мы отделяем уровень данных от уровня бизнес-логики. Кроме того, классы могут выполнять только запись в базу данных (команда) или чтение из нее (запрос). Итак, один класс не может выполнять оба. Такой подход дает много преимуществ. Код понятнее и проще в обслуживании или расширении. **Различные компоненты можно отдельно оптимизировать, разрабатывать и, что особенно важно, масштабировать.**

Впоследствии компоненты слабо связаны, и работа может быть эффективно разделена между разработчиками или командами. Наконец, приложение, разделенное на компоненты, легче тестировать. Не существует единственно правильного способа реализации шаблона CQRS. Реализация может основываться на домене, требованиях, структуре, фактическом состоянии проекта и т. Д. CQRS часто используется вместе с шаблоном поиска событий. Давайте опишем это.

### Поиск событий

Многие современные приложения полагаются на события для различных целей. Например, как мы упоминали ранее, служба в последовательности saga атомарно обновляет базу данных и публикует событие или сообщение. Источник событий использует события приложений.

**Поиск событий - это метод представления состояния с помощью сохраняющихся событий, изменяющих состояние**. Каждый раз, когда изменяется бизнес-объект, событие сохраняется в хранилище событий.

Как следует из названия, сайт событий - это база данных для событий. Это может быть SQL, NoSQL или любой другой способ, подходящий для проекта. Кроме того, хранилище событий может выступать в качестве посредника сообщений. Все заинтересованные компоненты подписываются на него. Когда событие сохраняется, хранилище событий предоставляет информацию всем подписчикам. Публикация события - это одна атомарная операция. Таким образом, это обеспечивает надежность и атомарность работы базы данных в микросервисах.

Кроме того, он создает полный журнал аудита. В случае возникновения какой-либо проблемы или ошибки легко исследовать изменения состояния и в конечном итоге восстановить действительное состояние. Таким образом, отладка менее сложна. Кроме того, **поиск событий может избежать несоответствия импеданса между объектно-ориентированными и реляционными данными**. Подводя итог, поиск событий может оказать большую помощь в архитектуре микросервисов или в любом событийном приложении.

## Как выбрать базу данных?

**Первым шагом при планировании проектирования базы данных в микросервисах является выбор модели**. Мы уже упоминали модели базы данных для каждой службы и общих баз данных. Кроме того, мы рассмотрели их плюсы, минусы и общие варианты использования.

Второй шаг - выбрать конкретную технологию (или технологии) базы данных, которая будет наиболее эффективной для проекта или службы. Для этого нам нужно рассмотреть несколько свойств.

Первым важным параметром является производительность чтения. Производительность чтения может быть либо количеством операций в секунду, либо скоростью запросов на выборку. Приложения или сервисы, связанные с электронной коммерцией, CRM, банковским программным обеспечением, обычно содержат функции, требующие быстрой и частой выборки данных.

Второе важное свойство - это производительность записи. Он похож на предыдущий. Просто в этом случае мы записываем данные в базу данных, а не читаем из нее. Если службам необходимо сохранять большое количество данных или даже хранить большие двоичные объекты, это может быть основным параметром.

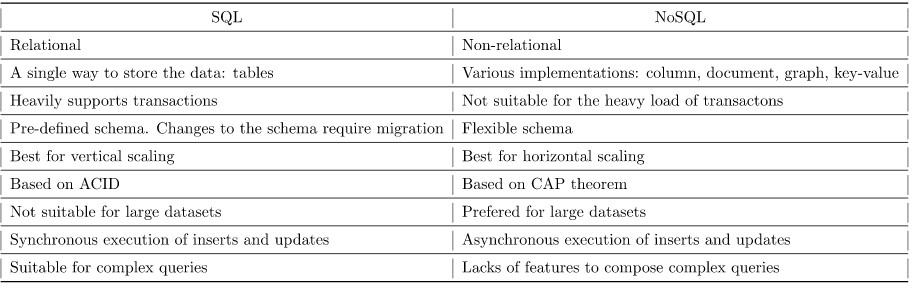
Следующий вопрос - это задержка. Это задержка между действиями пользователя и ответом сервера. Это особенно важно в компонентах, связанных с пользовательским интерфейсом. Хорошими примерами являются приложения для потокового вещания или игры в реальном времени.

Еще одним важным свойством является ресурсоэффективность. Обычно, чем меньше ресурсов потребляется, тем лучше. Это может привести к более быстрому выполнению, снижению нагрузки на хост и возможным затратам в зависимости от платформы.

И последнее, но не менее важное: мы должны учитывать эффективность предоставления ресурсов. В общем, это то, как база данных влияет на разработку, развертывание и тестирование микросервисов. Как мы уже упоминали ранее, независимость микросервисов в этих терминах действительно важна.

### SQL против NoSQL

Чаще всего для проекта или службы рассматриваются две технологии: SQL и NoSQL. В принципе, это сложнее, особенно если речь идет о NoSQL. Существует множество реализаций базы данных NoSQL, а именно. Хотя в этой статье мы не будем подробно останавливаться на низкоуровневой реализации базы данных. Давайте сравним SQL и NoSQL в целом.



# CAP теорема

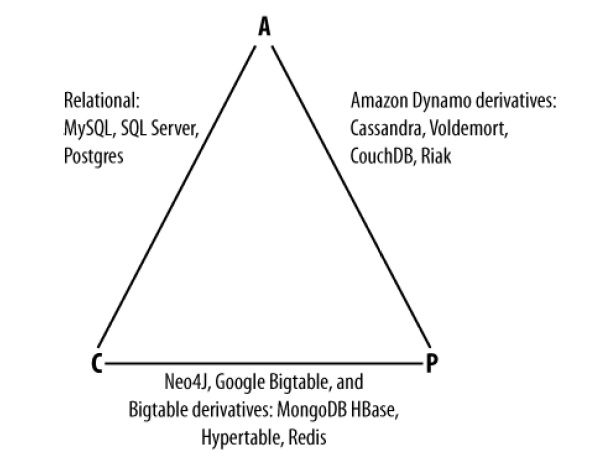
Эта теорема была представлена на симпозиуме по принципам распределенных вычислений в 2000 году Эриком Брюером. В 2002 году Сет Гилберт и Нэнси Линч из MIT опубликовали формальное доказательство гипотезы Брюера, сделав ее теоремой.  
По словам Брюера, он хотел, чтобы сообщество начало дискуссию о компромиссах в распределённых системах и уже спустя некоторое количество лет начал вносить в неё поправки и оговорки.  
***Что стоит за CAP***

В CAP говорится, что в распределенной системе возможно выбрать только 2 из 3-х свойств:

* C (consistency) — согласованность. Каждое чтение даст вам самую последнюю запись.
* A (availability) — доступность. Каждый узел (не упавший) всегда успешно выполняет запросы (на чтение и запись).
* P (partition tolerance) — устойчивость к распределению. Даже если между узлами нет связи, они продолжают работать независимо друг от друга.

Уже есть достаточно наглядных доказательств этой теоремы, поэтому дам ссылки на университет Баумана и доказательство в виде сервиса «Позвони, напомню!».  
В основном это всё треугольник

Многие статьи сводятся к вот такому вот простому треугольнику.



## Применяем на практике

Для применения CAP теоремы на практике, я выбрал 3 наиболее, на мой взгляд, подходящие и достаточно популярные системы баз данных: Postgresql, MongoDB, Cassandra.

**Посмотрим на Postgresql**

Следующие пункты относятся к абстрактной распределенной БД Postgresql:

* Репликация Master-Slave — одно из распространенных решений
* Синхронизация с Master в асинхронном / синхронном режиме
* Система транзакций использует двухфазный коммит для обеспечения consistency
* Если возникает partition, вы не можете взаимодейстовать с системой (в основном случае)

Таким образом, система не может продолжать работу в случае partition, но обеспечивает strong consistency и availability. Это система CA!  
  
**Посмотрим на MongoDB**

Следующие пункты относятся к абстрактной распределенной БД MongoDB:

* MongoDB обеспечивает strong consistency, потому что это система с одним Master узлом, и все записи идут по умолчанию в него.
* Автоматическая смена мастера, в случае отделения его от остальных узлов.
* В случае разделения сети, система прекратит принимать записи до тех пор, пока не убедится, что может безопасно завершить их.

Таким образом, система может продолжать работу в случае разделения сети, но теряется CAP-availability всех узлов. Это CP система!

### Посмотрим на Cassandra

Cassandra использует схему репликации master-master, что фактически означает AP систему, в которой разделение сети приводит к самодостаточному функционированию всех узлов.  
Казалось бы всё просто, но это не так.

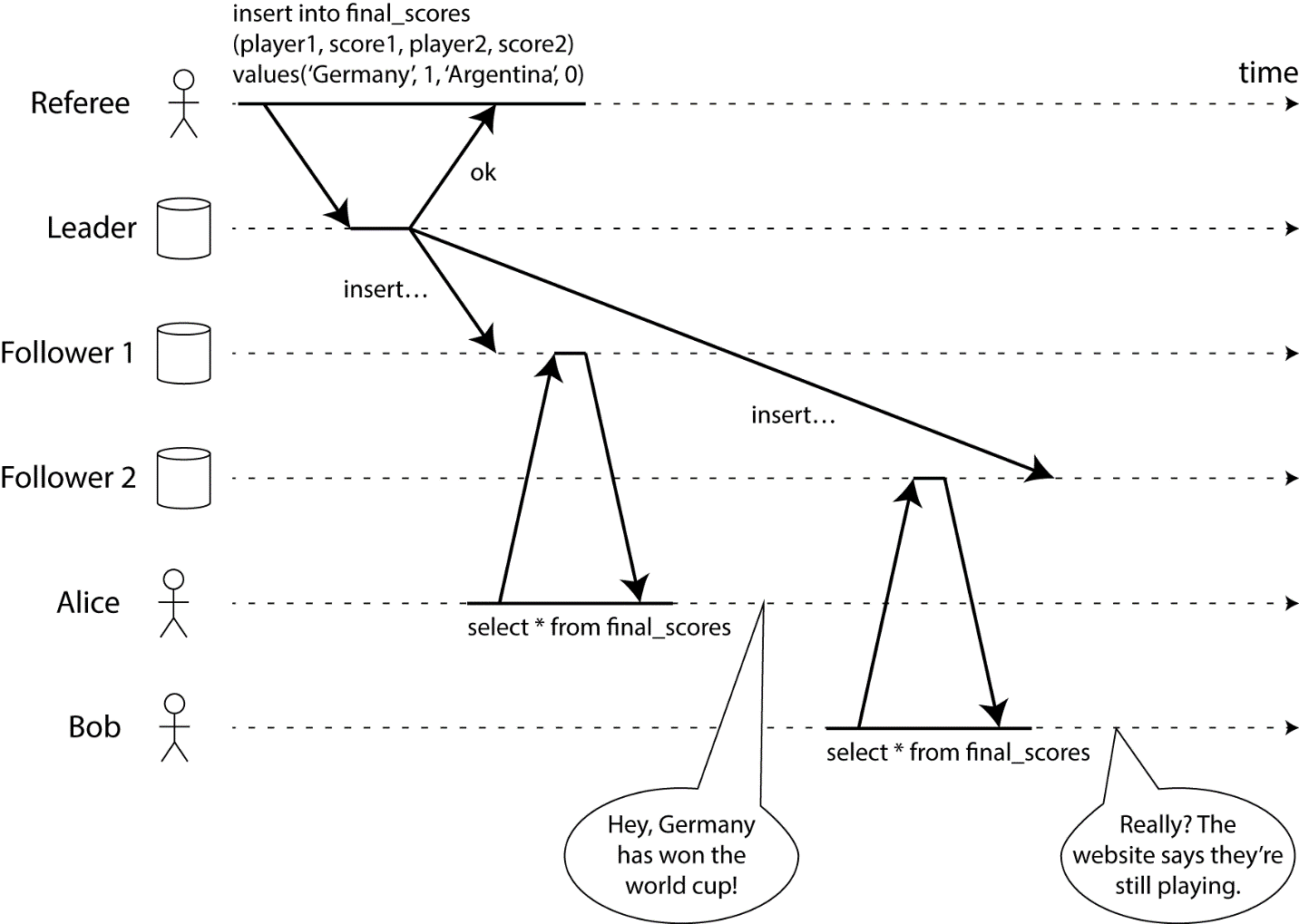
## Проблемы CAP

На тему проблем в CAP теореме написано множество подробных и интересных статей, здесь, на Хабре, поэтому я оставлю ссылку на CAP больше не актуален и мифы о CAP теореме. Обязательно почитайте их, но относитесь к каждой статье, как к своего рода новому взгляду и не принимайте слишком близко к сердцу, потому что одни ругают, другие хвалят. Сам же я не буду слишком углублятся, а постараюсь выдать некоторую необходимую компиляцию.  
Итак, проблемы CAP теоремы:

* Далёкие от реального мира определения
* В рамках разработки, выбор в основном лежит между CP и AP
* Множество систем — просто P
* Чистые AP и CP системы могут быть не тем, что ожидаешь

Что не так с определениями?

Consistency в CAP фактически означает линеаризуемость (и ее действительно трудно достичь). Чтобы объяснить, что такое линеаризуемость, давайте посмотрим на следующую картинку:

  
  
В описанном случае рефери закончил игру, но не каждый клиент получает один и тот же результат. Чтобы сделать его систему линеаризованной, нам нужно мгновенно синхронизировать данные между рефери и другими источниками данных, чтобы, когда рефери закончит игру, каждый клиент получил правильную информацию. Availability в CAP, исходя из определения имеет две серьёзные проблемы. Первая — нет понятия частичной доступности, или какой-то её степени (проценты например), а есть только полная доступность. Вторая проблема — неограниченное время ответа на запросы, т.е. даже если система отвечает час, она всё ещё доступна. Устойчивость к распределению не включает в себя упавшие узлы, и вот почему:

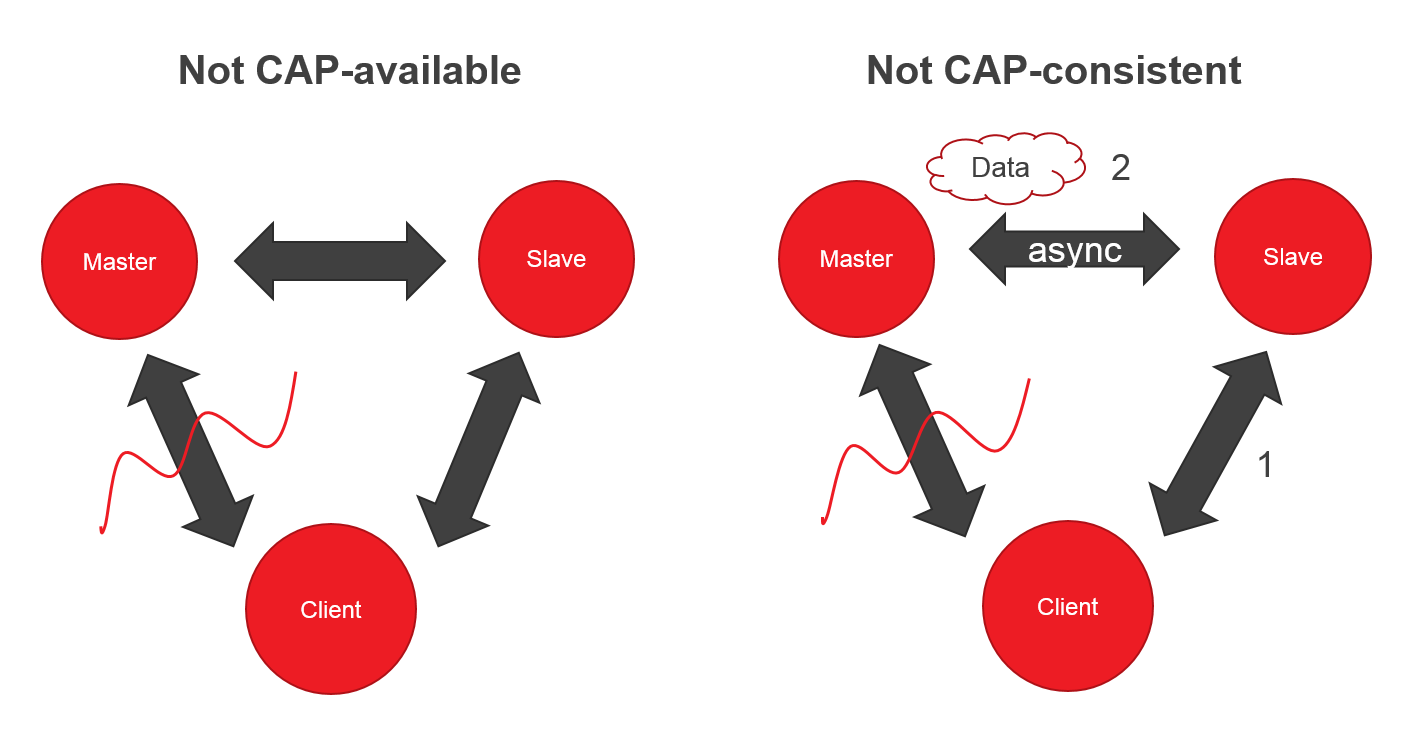
1. По определению. В availability так и прописано "...every node (if not failed) always..."
2. Исходя из доказательства. Доказательства CAP теоремы гласят что на узлах должен исполняться некоторый код.
3. Ну и немного моих (и не только) домыслов. В случае падения узла, система может восстановиться, пообщаться с другими узлами и продолжить работу как ни в чем ни бывало. В случае разделения сети — придётся ждать восстановления соединения.

Поэтому, нужно помнить про способность системы восстанавливаться, но за рамками CAP теоремы.

### AP / CP выбор

Коммуникация узлов между собой обычно происходит через асинхронную сеть, которая может задерживать или удалять сообщения. Интернет и все наши центры обработки данных обладают этим свойством, и это не маловероятные инциденты, поэтому CA системы в рамках разработки рассматриваются крайне редко.

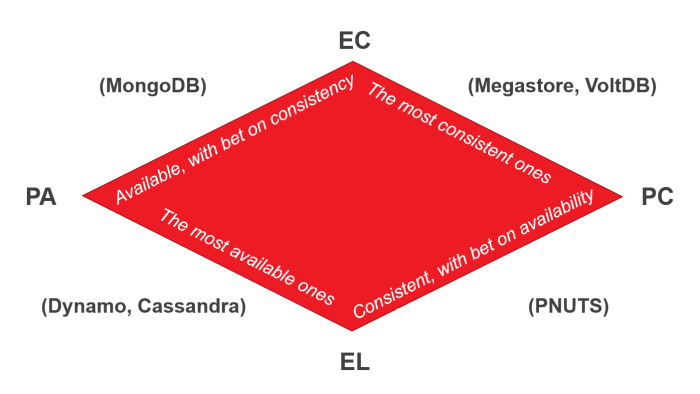
### Многие системы — просто P

Представьте систему, в которой два узла (Master, Slave) и клиент. Если вдруг вы потеряли связь с Master, клиент может читать из Slave, но не может писать — нет CAP-availability. Ок, вроде CP система, но если Master и Slave синхронизируются асинхронно, то клиент, может запросить данные от Slave раньше успешной синхронизации — теряем CAP-consistency.  
  
**Чистые AP и CP системы**

Чистые AP системы, могут включать в себя просто 2 генератора чисел. Чистые CP системы, могут вообще не быть доступны, т.к. буду пытаться придти к согласованному состоянию и не будут нам отвечать. Идём дальше, CP системы дают нам не ожидаемый нами strong consistency, а eventual consistency. О нём поговорим чуть позже. В конце концов, это всего лишь попытка классифицировать что-то абстрактное, поэтому вам не нужно изобретать велосипед. Я рекомендую использовать следующий подход при попытке работать с распределенными БД:

* Помните об определениях CAP и об их ограничениях.
* Используйте теорему PACELC вместо CAP, она позволяет взглянуть на систему ещё с одного ракурса.
* Помните про принципы ACID / BASE и насколько они применимы к вашей системе.
* Любые телодвижения следует делать, учитывая проект, над которым вы работаете.

### PACELC Теорема [PACELC](https://en.wikipedia.org/wiki/PACELC_theorem) была впервые описана и формализована Даниелом Дж. Абади из Йельского университета в 2012 году. Поскольку теорема PACELC основана на CAP, она также использует его определения. Вся теорема сводится к IF P -> (C or A), ELSE (C or L). Latency — это время, за которое клиент получит ответ и которое регулируется каким-либо уровнем consistency. Latency (задержка), в некотором смысле представляет собой степень доступности.



Немного о BASE

BASE — это своеобразный контраст ACID, который говорит нам, что истинная согласованность не может быть достигнута в реальном мире и не может быть смоделирована в высокомасштабируемых системах.  
Что стоит за BASE:

* Basic Availability. Система отвечает на любой запрос, но этот ответ может быть содержать ошибку или несогласованные данные.
* Soft-state. Состояние системы может меняться со временем из-за изменений конечной согласованности.
* Eventual consistency (конечная согласованность). Система, в конечном итоге, станет согласованной. Она будет продолжать принимать данные и не будет проверять каждую транзакцию на согласованность.