Дата последней сборки документа:
29 апреля 2024 г. в 01:54

Создание микросервисов

NB: Текст представляет собой непоследовательные выдержки из книг (иногда с большими смысловыми разрывами) и поэтому материал может показаться равным

Содержание

1. Основы

Микросервисы – это независимо выпускаемые сервисы, которые моделируются вокруг предметной области бизнеса. Сервис инкапсулирует функциональность и делает ее доступной для

других сервисов через сети – вы создает более сложную, комплексную систему из этих строительных блоков. Один микросервис может представлять складские запасы, другой – управление заказами и еще один – доставку, но вместе они могут составлять целую систему онлайн-продаж.

Они представляют собой *тип* сервис-ориентированной архитектуры, в которой ключевым фактором выступает возможность независимого развертывания. Они не зависят от технологий, что является одним из преимуществ.

Снаружи отдельный микросервис рассматривается как черный ящик. Он размещает бизнес-функции в одной или нескольких конечных точках сети (например, в очереди или REST API) по любым наиболее подходящим протоколам.

Потребители, будь то другие микросервисы или иные виды программ, получают доступ к этой функциональности через такие точки. Внутренние детали реализации (например, технология, по которой был создан сервис, или способ хранения данных) полностью скрыты от внешнего мира.

Это означает, что в *микросервисных* архитектурах в большинстве случаев *не используются общие базы данных*. Вместо этого каждый микросервис инкапсулирует свою собственную БД там, где это необходимо.

Микросервисы используют концепцию *скрытия информации*. Это означает, что скрытие как можно большего количества информации внутри компонента и как можно меньшее ее раскрытие через внешние интерфейсы.

Реализацию, скрытую от сторонних участников процесса, можно свободно преобразовывать, пока у сетевых интерфейсов, предоставляемых микросервисом, сохраняется обратная совместимость.

Сервис-ориентированная архитектура (SOA, service-oriented architecture) – это подход к проектированию, при котором несколько сервисов взаимодействуют для обеспечения определенного конечного набора возможностей (*сервис* здесь обычно означает полностью отдельный *процесс ОС*). Связь между этими сервисами осуществляется посредством сетевых вызовов, а не с помощью вызовов методов внутри границ процесса.

Вы должны воспринимать микросервисы как специфический подход к SOA.

1.1. Ключевые понятия микросервисов

Возможность *независимого развертывания* – это идея о том, что мы можем внести изменения в микросервис, развернуть его и предоставить это измененение нашим пользователям без необходимости развертывания каких-либо других микросервисов. Важно не только то, что мы можем это сделать, но и то, что *именно так* вы управляете развертываниеми в своей системе. Подходите к идее независимого развертывания как к чему-то *обязательному*.

Убедитесь, что вы придерживаетесь концепции независимого развертывания ваших микросервисов. Заведите привычку развертывать и выпускать изменения в одном микросервисе в готовом ПО без необходимости развертывания чего-либо еще. Это будет полезно.

Чтобы иметь возможность независимого развертывания, нам нужно убедиться, что наши микросервисы *слабо связаны*, то есть обеспечена возможность изменять один сервис без необходимости изменять что-либо еще. Это означает, что нужны явные, четко определенные и стабильные контракты между сервисами. Некоторые варианты реализации (например, совместное использование баз данных) затрудняют эту задачу.

Моделируя сервисы вокруг предметных областей бизнеса, можно упростить внедрение новых функций и процесс комбинирования микросервисов для предоставления новых функциональных возможностей нашим пользователям.

Развертывание функции, требующей внесения изменений более чем в один микросервис, обходится дорого. Вам придется координировать работу каждого сервиса (и, возможно, отдельных команд) и тщательно отслеживать порядок развертывания новых версий этих сервисов. Это потребует гораздо большего объема работ, чем внесение таких же преобразований внутри одного сервиса. Следовательно, нужно найти способы сделать межсервисные изменения как можно более редкими.

В случае микросервисов мы отдаем приоритет *сильной связности бизнес-функциональности*, а не технической функциональности.

Одна из самых непривычных рекомендаций при использовании *микросервисной* архитектуры состоит в том, что *необходимо избегать использования общих баз данных*. Если микросервис хочет получить доступ к данным, хранящимся в другом микросервисе, он должен напрямую запросить их у него [1, стр. 33].

Если мы хотим реализовать независимое развертывание, нужно убедиться, что есть ограничения на обратно несовместимые изменения в микросервисах. Если нарушить совместимость с вышестоящими потребителями, это неизбежно повлечет за собой необходимость внесения изменений и в них тоже.

1.2. Монолит

Когда все функциональные возможности в системе должны развертываться вместе, такую систему считают *монолитом*. Часто выделяют следующие архитектуры монолитов: однопроцессный, модульный и распределенный монолит.

1.2.1. Однопроцессный монолит

Наиболее распространенный пример, который приходит на ум при обсуждении монолитов, – система, в которой весь код развертывается как *единый процесс*. Может существовать несколько экземпляров этого процесса (из соображений надежности или масштабирования), но в реальности *весь код* упакован в один процесс.

1.2.2. Модульный монолит

Модульный монолит представляет собой систему, в которой один процесс состоит из отдельных модулей. С каждым модулем можно работать независимо, но *для развертывания* их все равно необходимо *объединить*.

Одна из проблем модульного монолита заключается в том, что *базе данных*, как правило, *не хватает декомпозиции*, которую мы находим на уровне кода, что приводит к значительным проблемам, если потребуется разобрать монолит в будущем.

1.2.3. Распределенный монолит

Распределенный монолит – это состоящая из нескольких сервисов система, которая должна быть развернута единовременно. Распределенный монолит вполне подходит под определение SOA, однако он не всегда соответствует требованиям SOA.

Микросервисная архитектура действительно предлагает более конкретные границы, по которым можно определить «зоны влияния» в системе, что дает гораздо больше гибкости.

1.2.4. Преимущества монолитов

Некоторые монолиты, такие как модульные и однопроцессные, обладают целым рядом преимуществ. Их гораздо более простая топология развертывания позволяет избежать многих ошибок, связанных с распределенными системами. Это может привести к значительному упрощению рабочих процессов, мониторинга, устранения неполадок и сквозного тестирования.

Для использования микросервисной архитектуры нужно найти убедительные причины.

1.3. Агрегирование логов и распределенная трассировка

Не используйте слишком много новых технологий в начале работы с микросервисами. Тем не менее *инструмент агрегации логов* настолько важен, что стоит рассматривать его как обязательное условие для внедрения микросервисов.

Такие системы позволяют собирать и объединять логи из всех ваших сервисов, предоставляя возможности для анализа и включения журналов в активный механизм оповещения.

1.4. Контейнеры и Kubernetes

Не стоит спешить с внедрением Kubernetes или даже контейнеров. Они, безусловно, предлагают значительные преимущества по сравнению с более традиционными технологиями развертывания, но в них нет особого смысла, если у вас всего несколько микросервисов.

После того как накладные расходы на управление развертыванием перерастут в серьезную головную боль, начните рассматривать возможность контейнеризации своего сервиса и использования Kubernetes. И если вы все же решитесь на этот шаг, сделайте все возможное, чтобы кто-то другой управлял кластером Kubernetes вместо вас, пусть даже и с использованием управляемого сервиса от облачного провайдера. Запуск собственного кластера Kubernetes может потребовать значительного объема работ!

1.5. Потоковая передача данных

Микросервисы позволяют отойти от использования монолитных баз данных, однако потребуется найти иные способы обмена данными. Поэтому среди людей, применяющих микросервисную архитектуру, стали популярными продукты, дающие возможность легко передавать и обрабатывать внушительные объемы данных.

Для многих людей Apache Kafka – стандартный выбор для потоковой передачи информации в среде микросервисов, и на то есть веские причины. Такие возможности, как постоянство сообщений, сжатие и способность масштабирования для обработки больших объемов сообщений, могут быть невероятно полезными. С появлением Kafka возникла возможность потоковой обработки в виде базы данных ksqlDB.

1.6. Преимущества микросервисов

1.6.1. Технологическая неоднородность

В системе, состоящей из нескольких взаимодействующих микросервисов, могут быть использованы разные технологии для каждого отдельного микросервиса. Так можно выбирать правильный инструмент для конкретной задачи вместо того, чтобы искать более стандартизированный, универсальный подход, который часто приводит к наименьшей отдаче.

1.6.2. Надежность

Вовремя обнаруженный вышедший из строя компонент системы можно изолировать, при этом остальная часть системы сохранит работоспособность. В монолитной системе, если сервис выходит из строя, перестает функционировать все.

1.6.3. Масштабирование

С массивным *монолитным* сервисом масштабировать придется *все целиком*. Допустим, одна небольшая часть общей системы ограничена в производительности, и если это поведение заложено в основе гигантского монолитного приложения, нам потребуется масштабировать *всю систему как единое целое*.

Изменение одной строки в монолитном приложении на миллион строк требует развертывания всего приложения для реализации новой сборки. Чем больше разница между релизами, тем выше риск того, что что-то пойдет не так!

Микросервисы же позволяют внести изменения в один сервис и развернуть его независимо от остальной части системы. Если проблема все же возникает, ее можно за короткое время изолировать и откатиться к предыдущему состоянию.

1.7. Слабые места микросервисов

Большинство проблем, связанных с микросервисами, можно отнести к распределенным системам, поэтому они с такой же вероятностью проявятся как в распределенном монолите, так и в микросервисной архитектуре.

Архитектура микросервисов вполне может предоставить *возможность* написать каждый микросервис на отдельном языке программирования, выполнять его в своей среде или использовать отдельную базу данных, но это *возможности*, а *не требования*. Необходимо найти баланс масштабности и сложности используемой вами технологии с издержками, которые она может повлечь.

Старайтесь внедрять новые сервисы в свою микросервисную архитектуру по мере необходимости. Нет нужды в кластере Kubernetes, когда у вас с системе всего три сервиса! Вы не только не будете перегружены сложностью этих инструментов, но и получите больше вариантов решения задач, которые, несомненно, появятся со временем.

Весьма вероятно, что в краткосрочной перспективе вы увидите увеличение затрат из-за ряда факторов. Во-первых, вам, скорее всего, потребуется запускать больше процессов, больше компьютеров, сетей, хранилищ и вспомогательного программного обеспечения (а это дополнительные лицензионные сборы).

Во-вторых, любые изменения, вносимые в команду или организацию, замедлят процесс работы. Чтобы изучить новые идеи и понять, как их эффективно использовать, потребуется время. Пока вы будете заняты этим, иные задачи никуда не денутся. Это приведет либо к прямому замедлению рабочего процесса, либо к необходимочсти добавить больше сотрудников, чтобы компенсировать эти затраты.

По опыту автора [1, стр. 54], микросервисы – плохой выбор для организаций, в первую очередь озабоченной снижением издержек, поскольку тактика сокращения затрат, когда сфера ИТ рассматривается как центр расходов, а не доходов, постоянно будет мешать получить максимальную отдачу от этой архитектуры.

Переход от *монолита*, где данные хранятся и управляются в *единой базе данных*, к *распределенной* системе, в которой несколько процессов управляют состоянием в *разных БД*, создает потенциальные *проблемы* в отношении *согласованности данных*.

В прошлом вы, возможно, полагались на транзакции базы данных для управления изменениями состояния, но, работая с микросервисной архитектурой, необходимо понимать, что подобную безопасность нелегко обеспечить. Использование *распределенных транзакций* в большинстве случаев оказывается весьма проблематичным при координации изменений состояния.

Вместо этого, возможно, придется использовать такие концепции, как *саги* и согласованность в конечном счете, чтобы управлять состоянием вашей системы и анализировать его. Опять же это еще одна веская причина быть осторожными в скорости декомпозиции своего приложения.

NB: Несмотря на стремление определенных групп сделать микросервисные архитектуры подходом по умолчанию, я считаю, что их внедрение все еще требует тщательного обдумывания из-за многочисленных сложностей.

1.8. Кому микросервисы не подойдут

Учитывая важность определения стабильных границ сервисов, я считаю, что *микросервисные архитектуры часто не подходят для совершенно новых продуктов или стартапов*. В целом я считаю, что более целесообразно подождать, пока модель предметной области не стабилизируется, прежде чем пытаться определить границы сервиса [1, стр. 57].

Действительно существует соблазн для стартапов начать работать на микросервисах, мотивируя это тем, что «если мы добьемся успеха, нам нужно будет масштабироваться!». Проблема в том, что заранее не известно, захочит ли кто-нибудь вооще использовать ваш продукт. Процесс поиска соответствия продукта рынку означает, что в конечном счете вы рискуете получить продукт, совершенно отличный от того, что вы задумывали.

Стартапы, как правило, располагают меньшим количеством людей, что создает больше проблем в отношении микросервисов. Микросервисы приносят с собой источники новых задач и усложнение системы, а это может ограничить ценную пропускную способность. Чем меньше команда, тем более заметными будут эти зартраты. Поэтому при работе с небольшими коллективами, состоящими всего из нескольких разработчиков, я *настоятельно не рекомендую микросервисы*.

Камнем преткновения в вопросе микросервисов для стартапов является весьма ограниченное количество человеческих ресурсов. Для небольшой команды может быть трудно обосновать использование рассматриваемой архитектуры из-за проблем с развертыванием и управлением самими микросервисами. Гораздо проще перейти к микросервисам позже, когда вы поймете, где

находятся ограничения в архитектуре и каковы слабые места системы, – тогда вы сможете сосредоточить свою энергию на внедрении микросервисов.

Архитектуры микросервисов могут значительно усложнить процесс развертывания и эксплуатации. Если вы используете программное обеспечение самостоятельно, можете компенсировать это, внедрив новые технологии, развив определенные навыки и изменив методы работы. Но не стоит ожидать подобного от своих клиентов, которые привыкли получать ваше ПО в качестве установочного пакета для Windows.

1.9. Где микросервисы хорошо работают

Главной причиной, по которой организации внедряют микросервисы, является возможность большему количеству программистов работать над одной и той же системой, при этом не мешая друг другу. Правильно определив свою архитектуру и организационные границы, вы позволите многим людям работать независимо друг от друга, снижая вероятность возникновения разногласий.

Если 5 человек организовали *стартап*, они, скорее всего, сочтут *микросервисную архитектуру обузой*. В то время как *быстро растущая компания*, состоящая из сотен сотрудников, скорее всего, придет к выводу, что ее рост гораздо легче приспособить к рассматриваемой нами архитектуре.

Приложения типа «программное обеспечение как услуга» (software as a service, SaaS) также хорошо подходят для архитектуры микросервисов. Обычно такие продукты работают 24/7. Возможность независимого выпуска микросервисных архитектур предоставляет огромное преимущество. Микросервисы по мере необходимости можно увеличить или уменьшить.

Благодаря технологически независимой природе микросервисов вы сможете получить максимальную отдачу от облачных платформ. Провайдеры публичных облачных сервисов предоставляют широкий спектр услуг и механизмов развертывания для вашего кода.

1.10. Как моделировать микросервисы

Основополагающие концепции:

- скрытие информации,
- связанность (coupling),
- связность (cohesion).

По сути, *микросервисы* – это просто еще одна форма *модульной декомпозиции*, хотя и с сетевым взаимодействием между моделями и всеми вытекающими проблемами [1, стр. 62].

Скрытие информации – это концепция, разработанная Дэвидом Парнасом для поиска наиболее эффективного способа определения *границ модулей*. Скрытие информации подразумевает скрытие как можно большего количества деталей за границей модуля (или в нашем случае микросервиса).

Преимущества модулей:

- ускоренное время разработки: разрабатывая модули независимо, мы можем выполнять больше параллельной работы и уменьшить влияние от добавления большего количества разработчиков в проект,
- понятность: каждый модуль можно рассматривать и понимать изолированно; это, в свою очередь, дает представление о том, что делает система в целом,

- гибкость: модули можно изменять независимо друг от друга, что позволяет вносить изменения в функциональность системы, не требуя преобразований других модулей; кроме того, их можно комбинировать различными способами для обеспечения новых возможностей.

Реальность такова, что наличие модулей не приводит к фактическому достижению необходимых результатов. Многое зависит от того, как формируются границы модуля.

Уменьшая количество предположений, которые один модуль (или микросервис) делает относительно другого, мы напрямую влияем на связи между ними. Сохраняя число предположений небольшим, легче гарантировать, что мы сможем изменить одну часть, не затрагивая другие.

Связность (cohesion) – мера силы взаимосвязанности элементов сервиса; способ и степень, в которой задачи, выполняемые им, связаны друг с другом. Для наших *микросервисных* архитектур мы стремимся к *сильной связности*.

Связанность (coupling) представляет собой степень взаимосвязи *между сервисами*. При создании систем необходимо стремиться к максимальной независимости сервисов, то есть их *связанность* должна быть *минимальной*.

Когда между сервисами наблюдается *слабая связанность*, изменения, вносимые в один сервис, не требуют изменений в другом. Для микросервиса самое главное – возможность внесения изменений в один сервис и его развертывания без необходимости вносить изменения в любую другую часть системы.

Слабо связанный сервис имеет необходимый минимум сведений о сервисах, с которыми ему приходится сотрудничать. Интенсивное общение сервисов может привести к сильной связанности.

Структура стабильна, если связность сильная, а связанность слабая.

Чтобы границы обеспечивали возможность независимого развертывания и позволяли работать над микросервисами параллельно, снижая уровень координации между командами, работающими над этими сервисами, необходима определенная степень стабильности самих границ.

Связность применима к отношениям между вещами *внутри* границы (микросервис в нашем контексте), а связанность представляет отношения между объектами *через* границу. Нет наилучшего способа организовать код. Связанность и связность – всего лишь характеристики, позволяющие сформулировать различные компромиссы, на которые мы идем в отношение кода. Все, к чему мы можем стремиться, – найти правильный баланс между этими двумя идеями, наиболее подходящими для вашего конкретного контекста и проблем.

В конечном счете определенная связанность в нашей системе будет неизбежно. Все, что мы хотим сделать, – уменьшить ее.

1.11. Связанность

1.11.1. Предметная связанность

Предметная (доменная) связь описывает ситуацию, в которой одному микросервису необходимо взаимодействовать с другим, поскольку первому требуется использовать функциональность, предоставляемую вторым микросервисом.

В микросервисной архитектуре данный тип взаимодействия практически неизбежен. Система, основанная на микросервисах, для выполнения своей работы полагается на взаимодействие нескольких микросервисов. Однако нам по-прежнему требуется свести это к минимуму. Ситуация, когда один микросервис зависит от нескольких нижестоящих микросервисов, означает, что он выполняет слишком много задач.

Как правило доменная связанность считается слабой формой связи, хотя даже здесь вполне реально столкнуться с проблемами. Микросервис, которому необходимо взаимодействовать с большим количеством нижестоящих микросервисов, может указывать на то, что слишком много логики было централизовано.

Помните о важности скрытия информации. Делитесь только тем, что необходимо, и отправляйте только минимальный требуемый объем данных [1, стр. 68].

Другая достаточно известная форма связанности – *временная*. Временная связь в контексте расределенной системы имеет немного другое значение: одному микросервису требуется, чтобы другой микросервис выполнял что-то в то же время.

Для завершения операции оба микросервиса дложны быть запущены и доступны для *одновременной* связи друг с другом.

Один из способов избежать временной связанности – использовать некоторую форму *асинхронной связи*, такую как *брокер сообщений*.

1.11.2. Сквозная связанность

Сквозная связанность описывает ситуацию, в которой один микросервис передает данные другому микросервису исключительно потому, что данные нужны какому-то третьему микросервису, находящемуся дальше по потоку. Во многих отношениях это одна из самых проблемных форм связи, поскольку она подразумевает, что вызывающий сервис должен знать, что вызываемый им сервис вызывает еще один. А также вызывающему микросервису необходимо знать, как работает удаленный от него микросервис.

Основная проблема сквозной связи заключается в том, что изменение требуемых данных ниже по потоку может привести к более значительному изменению выше по потоку.

1.11.3. Общая связанность

Общая связанность возникает, когда два или более микросервиса используют общий набор данных. Простым и распространенным примером такой формы связанности могут служить множественные микросервисы, использующие *одну и ту же БД*, но это также может проявляться в использовании *общей памяти* или *файловой системы*.

Основная проблема система с общей связанностью заключается в том, что изменения в структуре данных могут повлиять на несколько микросервисов одновременно. Здесь все множество сервисов считывает статистические справочные данные из общей БД. Если схема этой базы изменится обратно несовместимым образом, это потребует преобразований для каждого потребителя БД. На практике подобные общие данные, как правило, очень трудно изменить.

В конкретном случае важно, чтобы мы рассматривали запросы от сервисов «Склад» и «Обработчик заказов» именно как *запросы*. Задачей сервиса «Заказ» станет упрваление допустимыми переходами статусов, целиком связанными с заказом.

Убедитесь, что у нижестоящего микросервиса есть возможность отклонить недействительный запрос, отправленный в микросервис.

Альтернативным подходом в данном случае будет реализация сервиса «Заказ» в виде чего-то большего, чем просто оболочка для операций CRUD с базой данных, где запросы сопоставляются непосредственно с обновлениями БД.

NB: если вы видете микросервис, который выглядит просто как тонкая оболочка для CRUD-операций с базой данных, – это признак слабой связности и более сильной связанности, поскольку

логика, которая должна быть в этом сервисе для управления данными, распределена по другим местам вашей системы [1, стр. 75].

Источники общей связанности также являются потенциальными виновниками конкуренции за ресурсы. Множество микросервисов, использующих одну и ту же файловую систему или базу данных, могут перегрудать этот ресурс, вызывая серьезные последствия при его замедлении или недоступности. Общая БД особенно подвержена такой проблеме из-за возможной подачи к ней произвольных запросов различной сложности.

Так что общая связанность иногда допустима, но чаще всего – нет.

1.11.4. Связанность по содержимому

Связанность по содержимому проявляется, когда вышестоящий сервис проникает во внутренние компоненты нижестоящего и изменяет его состояние.

Наиболее распространенный вариант этого типа связанности представлен обращением внешнего сервиса к базе данных другого микросервиса и изменением ее напрямую.

Различия между связанностью по содержимому и общей связанностью практически не заметны. В обоих случаях два или более микросервиса выполняют чтение и запись одного и того же набора данных. При общей связанности используется общая внешняя зависимость, которую вы не контролируете. При связанности по содержимому границы становятся менее четкими, а разработчикам все сложнее изменять систему.

«Обработчик заказов» отправляет запросы сервису «Заказ», делегируя не только право на изменение статуса, но и ответственность за принятие решения о том, какие переходы статусов допустимы. С другой стороны, «Склад» напрямую обновляет таблицу, в которой хранятся данные заказа, минуя любые функции сервиса «Заказ», способные проверять допустимые преобразования. Мы должны надеяться, что сервис «Склад» содержит согласованный набор логики. гарантирующий внесение только разрешенных изменений. В лучшем случае логики продублируется. В худщем – мы можем получить заказы в очень необычных местах.

Когда вы разрешаете внешней стороне прямой доступ к своей базе данных, она фактически становится частью внешнего контракта, и даже в таком случае вам будет сложно решить, что можно или нельзя изменить. Теряется способность определять, что относится к общим ресурсам (и, следовательно, не может быть без труда изменено), а что скрыто. Короче говоря, избегайте связанности по содержимому.

1.12. Предметно-ориентированное проектирование

Предметно-ориентированное проектирование (DDD, domain-driven design) применяется, чтобы помочь создать ее модель.

Ключевые идеи DDD:

- *Единый язык.* Общий язык, определенный и принятый для использования в коде и при описании предметной области с целью облегчения коммуникации.
- *Агрегат.* Набор объектов, управляемых как единое целое, обычно ссылающихся на концепции реального мира.
- *Ограниченный контекст.* Четкая граница внутри предметной области бизнеса, которая обеспечивает функциональность более широкой системы, но также скрывает сложность.

Например, в предметной области MusicCorp агрегат «Заказ» может содержать несколько позиций, представляющих товары в заказе. Эти позиции имеют значение только как часть общего агрегата заказов.

В целом агрегат – нечто имеющее *состояние*, идентичность, жизненный цикл, которыми можно управлять как частью системы, – обычно относится к концепциям реального мира.

Здесь важно понимать, что, если внешняя сторона запрашивает переход состояния в агрегате, тот в свою очередь может сказать «нет». В идеале необходимо реализовать свои алгоритмы таким образом, чтобы недопустимые переходы состояний были невозможны.

Одни агрегаты могут взаимодействовать с другими. Агрегат – независимый конечный автомат, который фокусируется на концепции одной предметной области в нашей системе, при этом ограниченный контекст представляет собой набор связанных агрегатов, опять же с явным интерфейсом связи с внешними потребителями.

Агрегаты лучше не разделять – один микросервис может управлять разным количеством агрегатов, но наиболее благоприятный вариант – когда *одним агрегатом* управляет *один микросервис* [1, стр. 85].

Основная причина эффективности подхода DDD заключается в ограниченных контекстах, предназначенных для скрытия информации. Их применение дает возможность представить четкую границу модели предметной области с точки зрения более широкой системы, скрывая внутреннюю сложность реализации. Это также позволяет вносить изменения, не затрагивающие другие части системы. Таким образом, следуя подходу DDD, мы используем скрытие информации, что жизненно важно для определения стабильных границ микросервиса.

Если наши системы разложены по ограниченным контекстам, которые представляют нашу предметную область, любые желаемые модификации с большей вероятностью будут изолированны в пределах одной границы микросервиса. Это сокращает количество мест, в которых требуется внести изменения, и позволяет быстро их внедрить.

Подход DDD может быть невероятно полезен при построении микросервисных архитектур, однако это не единственный метод, применяемый при определении границ микросервиса.

Альтернативные подходы к определению границ микросервисов:

- волатильность,
- данные,
- технологии,
- организационный подход.

1.13. Волатильность

Я все чаще слышу о неприятии предметно-ориентированной декомпозиции, обычно от сторонников того, что волатильность представляет собой основную движущую силу декомпозции. Декомпозция на основе волатильности позволяет определить, какие части вашей системы часто изменяются, а затем извлечь эту функциональность в отдельные сервисы и таким образом более эффективно работать с ней. Если самая большая проблема связана с необходимостью масштабирования приложения, декомпозиция на основе *волатильности* вряд ли принесет большую пользу.

Декомпозиция на базе волатильности, проявляется и в бимодальных ИТ-моделях. Концепция бимодальной ИТ-модели, предложенная компанией Gartner, четко разделяет мир на категории с

краткими названиями «режим 1» (или «система учета») и «режим 2» (или «системы инноваций») в зависимости от скорости работы различных систем.

Системы режима 1 мало меняются и не требуют серьезного участия бизнеса, а в режиме 2 происходит действие с системами, требующими быстрых изменений и тесного вовлечения бизнеса.

Мне не нравится бимодальная ИТ-модель как концепция, поскольку она дает людям возможность упаковать то, что трудно изменить, в красивую аккуратную коробку и сказать: «Нам не нужно разбираться с проблемами там – это режим 1». Это еще одна модель, которую компании могут принять, чтобы объяснить, почему они не меняются. Ведь довольно часто изменения в функциональности требуют преобразований в системах учета (режим 1), чтобы можно было учесть изменения в системах инноваций (режим 2). По моему опыту, организации, внедряющие бимодальные ИТ-модели, в конечном счете получают два режима – медленный и еще медленее.

1.14. Смешивание моделей и исключений

Если вы будете следовать рекомендациям по скрытию информации и учитывать взаимодействие связанности и связности, то, скорее всего, сможете избежать некоторых недостатков любого выбранного механизма. Мне кажется, что, сосредоточившись на этих идеях, вы с большей вероятностью получите предметно-ориентированную архитектуру.

Однако часто могут возникать причины для смешивания моделей, даже если вы решите выбрать «предметно-ориентированную» модель в качестве *основного механизма определения границ микросервиса*.

Декомпозиция на основе волательности не бессмыслена, если вы сосредоточены на повышении скорости доставки.

Организационные и предметно-ориентированные границы сервисов – это моя собственная отправная точка, мой подход по умолчанию [1, стр. 95].

2. Разделение монолита на части

У многих, вероятно, нет возможности начать разработку системы с «чистого листа», и, даже если есть, начинать с микросервисов может оказаться не очень хорошей идеей [1, стр. 97].

2.1. Осознайте цель

Микросервисы не должны быть самоцелью. Вы ничего не выиграете просто от их присутствия в системе. Выбор микросервисной архитектуры – это осознанное, рациональное решение. Думать о переходе следует только в том случае, если вы не можете найти более простого способа достичь своей конечной цели имеющимися средствами.

Без четкого понимания целей создания системы можно попасть в ловушку, перепутав деятельностть с результатом. Я видел команды, одержимые идеей создания микросервисов, которые никогда не зададавались вопросом, зачем они им. И это крайне неразумно, учитывая сложности, которые могут привнести микросервисы.

Например, микросервисы, безусловно, способны помочь масштабировать систему, но есть и ряд альтернативных методов, на которые следует обратить внимание в первую очередь. Развертывание еще *нескольких копий* существующей *монолитной* системы за *балансировщиком нагрузки* вполне может помочь вам масштабировать систему гораздо эффективнее, чем сложная и длительная декомпозиция на микросервисы [1, стр. 97].

Какой микросервис необходимо создать в первую очередь? Без всеобъемлющего понимания конечной цели ответить на этот вопрос практически невозможно.

Поэтому четко определите, каких изменений вы пытаетесь добиться, и поищите более простые способы их реализации, прежде чем рассматривать микросервисы.

Выберите одну или две функции, реализуйте их как микросервисы и внедрите в ваш продукт, а затем подумайте, помогло ли вам это приблизиться к конечной цели. Вы не сможете оценить весь масштаб проблем, которые способна принести микросервисная архитектура, пока не запустите систему в работу [1, стр. 98].

Та или иная форма монолитной архитектуры может быть абсолютно правильным выбором, стоит повторить, что монолитная архитектура не является *плохой* по своей сути и поэтому не должна восприниматься враждебно. Не зацикливайтесь на отказе от монолита. Лучше сосредоточиться на преимуществах, которые должны принести изменения вашей архитектуры.

Не спешите создавть микросервисы, если у вас неясное представление о предметной области.

Преждевременная декомпозиция системы может оказаться *дорогостоящей*, особенно если вы новичок в этой области. Во многих отношениях работать с существующей кодовой базой, требующей разделения на микросервисы, намного проще, чем пытаться создавать микросервисы с самого начала [1, стр. 100].

Как только у вас появится четкое представление о необходимости внедрения микросервисов, вам потребуется определить, какие микросервисы создавать в первую очередь. Функции, в настоящее вермя ограничивающие способность системы справляться с нагрузкой, будут занимать первое место в списке. Хотите ускорить вермя выхода на рынок? Посмотрите на изменчивость системы, чтобы определить наиболее часто изменяющиеся части функциональности, и поймите, будут ли они работать как микросервисы. Для быстрого поиска наиболее изменчивых частей кодовой базы можно использовать инструменты статического анализа, такие как CodeScene https://codescene.com/.

Но также необходимо учитывать, какие варианты декомпозиции будут жизнеспособными. Некоторые функции могут оказаться настолько глубоко встроены в существующее монолитное приложение, что будет невозможно понять, как их извлечь. Или, возможно, рассматриваемая функциональность настолько важна для приложения, что любые модификации связаны с высоким риском. Или, наоборот, функциональность, которую вы хотите перенести, уже может быть автономной, и поэтому извлечение покажется очень простым.

Решение, какую функцию превратить в микросервис, в конечном счете будет представлять собой баланс между двумя факторами: насколько просто извлечение и насколько оно выгодно.

Совет для первой пары микросервисов: выбирайте что-то попроще – нечто оказывающее определенное влияние на достижение конечной цели, но в то же время это должен быть достаточно простой и доступный вариант. При длительном переходе, особенно таком, который может занять месяцы или годы, важно на раннем этапе ощутить динамику, получить результат своей работы.

2.2. Декомпозиция по слоям

2.2.1. Сначала код

Код, связанный с функциональностью списка избранного, извлечен в новый микросервис. На текущим этапе сведения для списка избранного остаются в базе данных монолита – декомпози-

ция не завершена, пока не перемещены данные, относящиеся к новому микросервису «Список избранного».

Если оставить данные в монолитной БД, в будущем накопится много проблем, которые тоже придется решать, однако мы уже многое выиграли от появления нового микросервиса. Извлечение кода приложения обычно проще, чем извлечение данных из БД.

2.2.2. Сначала данные

Ситуация, когда сначала извлекаются данные, а затем код приложения встречается гораздо реже, но он может быть полезен, когда нет уверенности в возможности четкого разделения данных.

2.3. Полезный шаблоны декомпозиции

2.3.1. Шаблон «Душитель»

Шаблон «Душитель» описывает процесс объединения старой и новой систем с течением времени, позволяя актуальной версии постепенно перенимать все больше и больше функций старой системы.

Выполняется перехват вызовов существующей системы – в нашем случае монолитного приложения. Если вызов этой части функциональности реализован в новой микросервисной архитектуре, он перенаправляется на микросервис. Если функциональность по-прежнему обеспечивается монолитом, вызову разрешается продолжить выполнение до самого монолита [1, стр. 104].

Прелесть этого шаблона заключается в возможности реализовать его без внесения каких-либо изменений в базовое монолитное приложение. Монолиту неизвестно, что он был «обернут» в более новую систему.

2.3.2. Параллельное выполнение

Один из способов убедиться в корректности работы новой функциональности, не подвергая риску поведение существующей системы, – использовать шаблон *параллельного выполнения* (parallel run): одновременное выполнение монолитной реализации функциональности и микросеврисной, обслуживание один и тех же запросов и сравнение результатов.

2.3.3. Шаблон переключаемых функций

Переключатель функций (feature toggle) – это механизм, позволяющий выключать или включать функцию или переключаться между двумя различными ее реализациями. У данного шаблона хорошая применимость во многих случаях, однако он особенно полезен при переходе к микросервисам.

2.4. Проблемы декомпозиции данных

Нередко при разделении БД на части нам в конечном счете приходится перемещать операции объединения (JOIN) с уровня данных в сами микросервисы.

Поскольку разделенные таблицы теперь находятся в разных базах данных, у нас больше нет возможности обеспечить целостность модели данных. Ничто не мешает нам удалить строку

в таблице «Альбомы», что вызовет проблему, когда мы попытаемся определить, какой именно товар был продан.

Как только мы начинаем разделять данные по нескольким БД, утрачивается безопасность транзакций ACID, к которым мы привыкли. Распределенные транзакции не только сложны в реализации, то и на самом деле не дают нам тех гарантий, которые мы привыкли ожидать от транзакций с более узким охватом базы данных.

2.5. Стили взаимодействия микросервисов

Вызовы между различными процессами по сети сильно отличаются от вызовов внутри одного процесса. Когда выполняется *внутрипроцессный вызов*, базовый компилятор и среда выполнения могут произвести целый ряд *оптимизаций*, чтобы уменьшить влияние вызова на производительность, включая встраивание вызова в процесс выполнения, будто его никогда и не было. При *межпроцессных вызовах* такая оптимизация невозможна.

Пакеты должны быть отправлены. Накладные расходы на такой вызов ожидаемо будут выше, чем при внутрипроцессном вызове.

С другой стороны, данные фактически должны быть *сериализованы* в некую форму, которую можно передавать по сети при выполнении вызовов между микросервисами. Затем данные необходимо *отправить* и *десериализовать* принимающей стороне. Поэтому нам, возможно, потребуется более внимательно относиться к размеру полезных нагрузок, отправляемых между процессами.

2.6. Стили взаимодействия микросервисов

Синхронная блокировка: микросервис выполняет вызов другого микросервиса и блокирует операцию в ожидании ответа.

Асинхронная неблокирующая связь: микросервис, отправляющий вызов, способен продолжать работу независимо от того, принят ответ или нет.

Запрос – ответ: микросервис отправляет другому микросервису запрос на какое-то действие и ожидает получить ответ, информирующий его о результате.

Событийный стиль: микросервисы генерируют события, которые другие микросервисы потребляют, и реагируют на них соответствующим образом. Микросервис, выпускающий события, не знает их конечного потребителя и имеется ли таковой вообще.

Общие данные: не часто рассматриваются как стиль коммуникации. При таком стиле микросервисы взаимодействуют через какой-то общий источник данных.

В целом рекомендуется начинать с принятия решения о том, какой стиль взаимодействия более подходит для данной ситуации: «запрос – ответ» или событийный стиль. Если рассматривать стиль «запрос – ответ», то будут доступны как синхронные, так и асинхронные реализации, поэтому возникает необходимостть сделать второй выбор. Однако при выборе событийного стиля взаимодействия способы реализации будут ограничены неблокирующим асинхронным вариантом [1, стр. 119].

Микросервисная архитектура в целом может поддерживать *различные стили взаимодействия*, и это считается нормой. Одни виды взаимодействия имеет смысл организовать просто в стиле «запрос – ответ», в то время как другие – в событийном стиле. На самом деле для одного микросервиса обычно реализуется более одной формы взаимодействия.

2.6.1. Шаблон: синхронная блокировка

При синхронном блокирующем вызове микросервис отправляет какой-либо вызов нижестоящему процессу (вероятно, другому микросервису) и блокируется до завершения вызова и, возможно, до получения ответа.

Как правило, синхронный блокирующий вызов – это вызов, ожидающий ответа от нижестоящего процесса. Такой подход связан с необходимостью использовать результат вызова для какой-то дальнейшей операции или, если отклик не получен, предпринять повторную попытку.

Преимущества В блокирующем синхронном вызове есть что-то простое и знакомое. Большинство ситуаций, в которых используются межпроцессные вызовы, вероятно, исполнялись в синхронном, блокирующем стиле – например, выполнение SQL-запроса к базе данных или HTTP-запроса к нижестоящему API.

Недостатки Основная проблема при синхронных вызовах – возникающая *временная связанность*. Когда «Обработчик заказов» вызывал сервис «Лояльность», этот микросервис оставался доступным для вызова. Если микросервис «Лояльность» недоступен, то вызов завершается неудачей и «Обработчику заказов» необходимо выполнить компенсирующее действие: немедленную повторную попытку, буферизацию вызова для повторной попытки позже или, возможно, полный отказ.

Это двустороняя связанность. Временная связанность здесь возникает не только между двумя микросервисами – она существует между двумя конкретными экземплярами этих микросервисов.

Отправитель вызова блокируется и *ожидает ответа* нижестоящего микросервиса, из чего следует, что если нижестоящий микросервис отвечает медленно или если существует проблема задержки сети, то отправитель вызова будет заблокирован в течение длительного периода времени в ожидании ответа. Если микросервис «Лояльность» находится под значительной нагрузкой и долго отвечает на зпросы, это, в свою очередь, приводит к тому, что и «Обработчик заказов» замедляется.

Где использовать В простых микросервисных архитектурах серьезных проблем с использованием *синхронных блокирующих* вызовов не возникает. Для меня использование этих типов вызовов становится спорным, когда появляется больше цепочек вызовов [1, стр. 121].

Если все вызовы в цепочке будут синхронными и блокирующими, вы столкнетесь с рядом сложностей. Проблема в любом из четырех задействованных микросервисов или сетевых вызовах между ними может привести к сбою всей операции. И это помимо конкуренции за ресурсы, которую могут вызвать такие длинные цепочки. За кулисами «Обработчик заказов», вероятно, поддерживает открытое сетевое соединение, ожидающее ответа от сервиса «Оплата», у которого, в свою очередь, имеется открытое сетевое соединение, ожидающее ответа от сервиса «Обнаружение мошенничества», и т.д.

Наличие большого количества *подключений*, которые необходимо держать открытыми, может повлиять на работу системы: либо доступные *подключения закончаться*, либо произойдет *перегрузка сети* [1, стр. 122]

2.6.2. Шаблон: асинхронная неблокирующая связь

При *асинхронной* связи процесс отправки вызова по сети *не блокирует микросервис*, отправляющий вызов. Тот способен продолжать любую другую обработку, не дожидаясь ответа.

Рассмотрим наиболее распространенные варианты неблокирующей асинхронной связи:

- *связь через общие данные*. Вышестоящий микросервис изменяет кое-какие общие данные, которые позже использует один или несколько сервисов.
- *Запрос – ответ*. Микросервис отправляет другому микросервису запрос на какое-то действие. Когда запрошенная операция завершается успешно (или нет), вышестоящий микросервис получает ответ. В частности, любой экземпляр вышестоящего микросервиса должен быть в состоянии обработать ответ.
- *Событийное взаимодействие*. Микросервис транслирует событие, которое можно рассматривать как фактическое утверждение о чем-то, что произошло. Другие микросервисы могут прослушивать интересующие их события и реагировать соответствующим образом.

Преимущества При неблокирующей асинхронной связи микросервис, выполняющий превоначальный вызов, и микросервис (или микросервисы), принимающий вызов, временно утрачивают связанность. Данный стиль связи также полезен, если для обработки запроса требуется много времени. Процесс поиска компакт-дисков на стелажах, упаковки и доставки может занять от пары часов до нескольких дней. Следовательно, имеет смысл «Обработчику заказов» выполнить неблокирующий асинхронный вызов сервиса «Склад», чтобы затем получить от него информацию о продвижении заказа. Это форма асинхронной связи «запрос – ответ».

Недостатки Основным недостатком неблокирующей асинхронной связи являются уровень сложности и диапазон выбора.

Где использовать Очевидным кандидатом представляются *длительные процессы*. Кроме того, хорошими претендентами могут быть ситуации с *длинными цепочками вызовов*, которые нелегко реструктурировать.

2.6.3. Шаблон: связь через общие данные

Стиль взаимодействия, охватывающий множество реализаций, – это связь через общие данные. Данный шаблон используется, когда один микросервис помещает данные в определенное место, а другой (или, возможно, несколько) позже использует их. Представьте, что один микросервис помещает файл в определенное место, а в какой-то момент позже другой микросервис берет этот файл и что-то с ним делает. Такой стиль интеграции принципиально *асинхронен* по своей природе.

Этот шаблон является *наиболее распространенным общим паттерном межпроцессного взаимодействия*. И все же мы иногда вообще не рассматриваем его как шаблон коммуникации, так как связь между процессами часто настолько косвенна, что ее трудно заметить.

Реализация Чтобы реализовать такой шаблон, вам нужно *постоянное хранилище данныех. Файловой системы* во многих случаях будет достаточно. Я создал много систем, которые просто *периодически сканируют файловую систему*, отмечают наличие нового файла и реагируют на него соответствующим образом.

Стоит отметить, что любому нижестоящему микросервису, которому предстоит работать с этими данными, потребуется собственный механизм для определения доступности новых данных – поллинг часто применяется в качестве решения этой проблемы.

Два распространенных примера рассматриваемого шаблона представлены озером данных и хранилищем данных. В обоих случаях эти решения обычно предназначены для обработки больших объемов данных.

При использовании озера данных источники загружают необработанные данные в любом удобном для них формате, а расположенные ниже по потоку потребители этих данных будут знать, как обрабатывать информацию. Хранилище данных представляет собой структурированную систему хранения данных. Микросервисы, передающие данные в хранилище, должны знать его структуру.

Как для хранилища данных, так и для озера данных предполагается, что поток информации идет в одном направлении. Один микросервис публикует данные в общем хранилище данных, а нижестоящие потребители считывают их и выполняют соответствующие действия. Проблемой станет реализация *совместно используемой БД*, в которой множество микросервисов будут *считывать* и *записывать* данные в одно и то же хранилище [1, стр. 127]

Преимущества Если у вас есть возможность выполнять чтение/запись файла или базы данных, вы можете использовать этот шаблон. Объемы данных также не вызывают здесь особого беспокойства: если вы отправляете много данных за один большой подход – это шаблон вполне сгодится.

Недостатки Нижестоящие потребляющие микросервисы обычно узнают о наличии новых данных для обработки с помощью какого-либо механизма поллинга или периодически запускаемого процесса. Это означает, что такой механизм вряд ли будет полезен в ситуациях, требующих минимального отклика. Если вы заинтересованы в отправке больших объемов данных и их обработке в режиме реального времени, то лучше использовать какую-либо потоковую технологию, такую как Kafka.

Еще один большой недостаток заключается в том, что общее хранилище данных становится потенциальным истточником связанности. Если это хранилище каким-либо образом изменит структуру, это может нарушить связь между микросервисами.

Где использовать Где эта модель действительно хороша, так это в обеспечении взаимодействия между процессами, имеющими ограничения на доступные к использованию технологии. Еще одним важным преимуществом этой модели стало совместное использование больших объемов данных. Если требуется отправить очень объемный файл в файловую систему или загрузить несколько миллионов строк в базу данных, то использование этого шаблона станет разумным выходом из ситуации.

2.6.4. Шаблон: связь «запрос – ответ»

При использовании модели «запрос – ответ» микросервис отправляет запрос на какое-либо действие нижестоящему сервису и ожидает получить ответ с результатом запроса. Это взаимодействие можно осуществить с помощью синхронного блокирующего вызова или асинхронным неблокирующим методом.

Извлечение данных из других микросервисов, подобных этому, – распространенный вариант использования для вызова «запрос – ответ».

Реализация: синхронная или асинхронная Подобные вызовы «запрос – ответ» можно реализовать либо в *блокирующем синхронном*, либо в *неблокирующем асинхронном стиле*. При синхронном вызове, как правило, *открывается сетевое соединение* с микросервисом ниже по потоку, а запрос отправляется по этому соединению. Соединение остается открытым, пока вышестоящий микросервис ожидает ответа нижестоящего. Если соединение обрывается, например, если какой-либо из экземпляров микросервиса удален, тогда у нас может возникнуть проблема.

С асинхронным вызовом в стиле «запрос – ответ» все не так просто. Запрос на резервирование отправляется в виде сообщения через своего рода брокер сообщений. Вместо того чтобы сообщение отправлялось непосредственно в микросервис «Запасы» из «Обработчик заказов», оно помещается в очередь. Сервис «Запасы» по возможности считывает сообщения из этой очереди и выполняет связанную с ними работу по резервированию запасов. Микросервису «Запасы» необходимо знать, куда направить ответ. В нашем примере он отправляет его обратно по другой очереди, используемой «Обработчиком заказов».

Таким образом, при неблокирующем асинхронном взаимодействии микросервис, получающий запрос, должен либо неявно знать, куда направить ответ, либо получить указание, куда его послать. Это поможет в ситуациях, когда запросы невозможно обработать достаточно быстро.

Когда микросервис получает ответ таким образом, ему требуется связать его с исходным запросом. В нашем примере резервирования запасов в рамках размещения заказа необходимо знать, как связать ответ «запас зарезервирован» с данным заказом для его дальнейшей обработки. Проще всего было бы сохранить любое состояние, связанное с исходным запросом, в базе данных, чтобы при поступлении ответа принимающий экземпляр мог загрузить какое угодно связанное состояние и действовать соответствующим образом.

Все типы взаимодействия «запрос – ответ», вероятно, потребуют некоторой формы обработки тайм-аута, чтобы избежать пробем, когда система будет заблокирована в ожидании чего-то, что может никогда не произойти.

Где использовать Вызовы типа «запрос – ответ» идеально подходят для ситуации, в которой результат запроса необходим для дальнейшей обработки. Единственный оставшийся вопрос – что выбрать: синхронную или асинхронную реализацию с теми же компромиссами.

2.6.5. Шаблон: событийное взаимодействие

Вместо того чтобы инициировать в другом сервисе какое-либо действие, микросервис выдает события, которые могут быть получены или не получены другими микросервисами. Это по своей сути асинхронное взаимодействие, поскольку прослушиватели событий будут работать в своем собстенном потоке выполнения.

Сервис, его отправляющий, может не знать ни о намерении других сервисов использовать это событие, ни даже об их существовании. Он выдает событие по необходимости, и на этом его обязанности заканчиваются.

Отправитель событий оставляет за получателем право решать, что делать. При использовании модели «запрос – ответ» микросервис, отправляющий запрос, ожидает определенной реакции и сообщает другому сервису, что должно произойти дальше. Это означает, что при работе с

моделью «запрос – ответ» отправитель запроса должен знать, что может сделать нижестоящий получатель. В итоге мы получаем большую степень *предметной связанности*.

При событийном взаимодействии отправителю событий не обязательно знать о нижестоящих микросервисах и об их действиях, в результате чего связанность значительно снижается [1, стр. 134].

Ооообщение – это то, что мы отправляем через асинхронный механизм связи, например через брокер сообщений. При событийном сотрудничестве мы траслируем это событие, помещая его в сообщение. Сообщение – это средство, а событие – полезная нагрузка.

Реализация Здесь необходимо рассмотреть два основных способа: способ, которым микросервисы производят события, и способ, которым потребители узнают, что эти события произошли.

Традиционно брокеры сообщений, такие как RabbitMQ, пытаются справиться с обеими проблемами. Производители используют API для публикации события брокеру, который, в свою очередь, обрабатывает подписки, позволяя потребителям получать информацию о наступлении события.

Если у вас уже есть хороший, надежный брокер сообщений, используйте его для обработки публикации и подписки на события.

События с большим количеством информации могут обеспечить более слабую связанность (это хорошо), таки события также могут использоваться в качестве архивной справки о произошедшем с определенной сущностью.

Хотя этот подход, безусловно, для меня предпочтителен, он не лешен недостатков. Для начала, если объем связанных с событием данных внушителен, у нас могут возникнуть опасения по поводу размера события. У соверменных брокеров сообщений довольно жесткие ограничения по размеру сообщения. Максимальный размер сообщения по умолчанию в Kafka составляет 1 Мбайт, а последняя версия RabbitMQ поддерживает теоретический верхний предел 512 Мбайт для одного сообщения (по сравнению с предыдущим 2 Гбайта!).

Где использовать Событийное взаимодействие лучше всего использовать в ситуациях, когда информацию требуется транслировать, и в ситуациях, когда вы инвертируете цель. Большую привлекательность обретает переход от модели указания другим блокам, что делать, к предоставлению нижестоящим микросервисам возможности решать такие вопросы самостоятельно.

В ситуации, когда вы уделяете больше внимания слабой связанности, чем другим факторам, событийное взаимодействие будет более привлекательным.

Лично я заметил, что таготею к событийному взаимодействию почти по умолчанию. Событийные архитектуры приводят к созданию значительно менее связанных и масштабируемых систем. Но подобные стили взаимодействия на самом деле приводят к повышению общей сложности сиситемы.

3. Реализация

3.1. Реализация коммуникации микросервисов

Существует множество вариантов взаимодействия микросервисов: SOAP, XML-RPC, REST, gRPC etc.

Упростите обратную совместимость. При внесении изменений в микросервисы необходимо убедиться, что мы не нарушаем совместимость с любыми потребляющими микросервисами. В идеале мы хотим получить возможность проверять внесенные изменения на предмет обратной совместимости и иметь возможность получить эту обратную связь до того, как запустим микросервис в эксплуатацию.

Следите за тем, чтобы ваши API не зависели от технологий. Я думаю, что надо всегда оставаться открытым для новых возможностей. Очень важно сделать так, чтобы API, используемые для связи между микросервисами, не зависели от технологий.

Выбор технологий:

- *Удаленный вызов процедур*: фреймворки, позволяющие применять локальные вызовы методов в удаленном процессе. Распространенные варианты включают SOAP и gRPC.
- *REST*. архитектурный стиль, где вы предоставляете ресурсы («Клиент», «Заказ» и пр.), к которым можно получить доступ с помощью общего набора команд (`GET`, `POST`).
- *GraphQL*: относительно новый протокол, позволяющий потребителям определять пользовательские запросы, которые будут извлекать информацию из нескольких нижестоящих микросервисов, фильтруя результаты, чтобы возвращать только требующиеся данные.
- *Брокеры сообщений*: промежуточное ПО, позволяющее осуществлять асинхронную связь через очереди или топики.

3.1.1. Удаленные вызовы процедур

Удаленный вызов процедур (remote procedure call, RPC) относится к технике реализации локального вызова и его выполнения где-то на удаленном сервисе. Большая часть технологий в этой области требует явной схемы, применяемой, например, в системах SOAP или gRPC.

Как правило, использование технологии RPC означает, что вы приобретаете *протокол сериализации*. Фреймворк RPC определяет, как данные сериализуются и десериализуются. Например, gRPC использует для этой цели формат сериализации Protocol Buffers. Некоторые реализации привязаны к определенному сетевому протоколу, в то время как другие могут позволить вам применить различные типы сетевых протоколов, предоставляющих дополнительные функции. Например, TCP предлагает гарантии доставки, а UDP – нет, хотя требует гораздо меньше накладных расходов.

Платформа Avro RPC является исключением, посольку она дает возможность отправлять полную схему вместе с полезной нагрузкой, позволяя клиентам динамически интерпретировать схему.

Простота генерации клиентского кода – одно из основных преимуществ RPC. Возможность вызвать обычный метод и теоретически проигнорировать все остальное – настоящая находка.

Проблемы Технологическая связанность. Некоторые механизмы RPC, такие как Java RMI, сильно привязаны к конкретной платформе, что может ограничить использование технологии на стороне клиента и сервера. Технология RPC иногда имеет ограничения по совместимости.

В некотором смысле эта технологическая связанность может быть формой раскрытия внутренних технических деталей реализации. Например, использование RMI связывает с JVM не только клиент, но и сервер.

Однако стоит отметить, что сущесвтует ряд реализаций RPC, не имеющих подобного ограничения, – gRPC, SOAP и Thrift. Все это примеры, обеспечивающие совместимость между разлиными стеками технологий.

Локальные и удаленные вызовы различаются. Основная идея *RPC – скрыть сложность удаленного вызова*. Необходимо помнить, что удаленные и локальные вызовы методов – это разные вещи, хоть и некоторые формы RPC стремятся их уравнять и скрыть этот факт. Можно совершать большое количество локальных вызовов в процессе, не слишком беспокоясь о производительности. При использовании RPC затраты на машалинг и анмаршалинг полезных нагрузок могут быть значительными, не говоря уже о времени, необходимом на отправку данных по сети. Это означает, что разработка API для удаленных интерфейсов потребует подхода, отличного от разработки локальных версий.

Сети ненаджены. Они могут не сработать, даже если общающиеся клиент и сервер работоспособны. Ожидать стоит чего угодно: моментального выхода из строя, постепенной деградации, даже искажения ваших пакетов. Исходите из того, что ваши сети кишат злонамеренными сущностями, готовыми в любой момент все сломать. В итоге вы столкнетесь с такими типами сбоев, с которыми, возможно, никогда не имели дело в более простом монолитном ПО.

Некоторые из самых популярных реализаций RPC могут привести к отедльным неприятным формам уязвимости. Java RMI является очень хорошим примером.

Если в серверной реализации убрать поле `age` из определения этого типа, и не сделать то же самое для всех потребителей, то даже если они никогда не использовали это поле, код, связанный с десериализацией объекта `Customer` на стороне потребителя, будет нарушен. Чтобы внедрить это данное изменение, нам потребуется преобразовать код клиента для поддержки нового определения и развернуть эти обновленные клиенты одновременно с развертыванием новой версии сервера. Ключевая проблема любого механизма RPC: вы не можете разделить развертывание клиента и сервера.

На практике объекты, используемые как часть двоичной сериализации при передаче данных по сети, стоит рассматривать как типы «только для расширения». Эта уязвимость приводит к тому, что типы подвергаются воздействию процесса передачи по сети и превращаются в массу полей, часть из которых больше не используется, но и не могут быть безопасно удалены.

Где использовать Несмотря на недостатки, мне все равно очень нравится RPC, а его более современные реализации, такие как gRPC, превосходны, в то время как у аналогов имеются существенные проблемы. Например, SOAP довольно тяжеловесен с точки зрения разработчика, особенно по сравнению с более современными вариантами.

Если бы я рассматривал варианты из этой области, gRPC был бы в топе моего списка. Он позволяет использовать преимущества HTTP/2, обладает некоторыми впечатляющими характеристиками производительности и в целом прост в работе.

Платформа gRPC хорошо подходит для *синхронной* модели «запрос – ответ», но также в состоянии работать с *реактивными* расширениями.

Если появляется необходимость поддерживать широкий спектр других приложений, которым может потребоваться взаимодействие с вашими микросервисами, компиляция клиентского кода в соответствии со схемой на стороне сервера может стать проблемой. В этом случае, скорее всего, лучше подойдет какая-либо форма REST API через HTTP.

3.1.2. REST

Передача репрезентативного состояния (representational state transfer, REST) – это архитектурный стиль. Самое важное в REST – это концепция ресурсов. Внешнее отображение ресурса полностью отделено от способа его хранения внутри системы. Например, клиент может запросить JSON-представление объекта `Customer`, даже если оно храниться в совершенно другом формате.

REST чаще всего работает через HTTP. Сам HTTP определяет ряд полезных возможностей, которые очень хорошо сочетаются с REST. Например, методы HTTP-запросов (такие как `GET`, `POST` и `PUT`) уже содержать хорошо понятные значения в спецификации HTTP относительно того, как они должны работать с ресурсами. Архитектурный стиль REST на самом деле позволяет этим методам вести себя одинаково на всех ресурсах, а спецификация HTTP определяет набор доступных для использования команд.

Например, `GET` извлекает ресурс *идемпотентным* способом, а `POST` создает новый ресурс.

Идемпотентность – свойство объекта или операции при повторном применении операции к объекту давать тот же результат, что и при одинарном.

Теперь мы можем с помощью метода `POST` просто отправить представление клиента, чтобы запросить у сервера создание нового ресурса, а затем инициировать запрос `GET` для получения представления ресурса. Концептуально в этих случаях существует одна *конечная точка* в виде ресурса `Customer`, и операции, которые мы можем выполнять с ним, записываются в протокол HTTP.

HTTP также предоставляет обширную экосистему вспомогательных инструментов и технологий. Имеется возможность применять прокси-серверы кэширования HTTP, например Varnish, балансировщики нагрузки `mod_proxy` и многие другие инструменты мониторинга, уже с поддержкой HTTP.

Эти инструменты позволяют обрабатывать большие объемы HTTP-трафика и маршрутизировать их разумно и довольно прозрачно. Обратите внимание, что HTTP может применяться и для реализации RPC.

gRPC был разработан специально для раскрытия потенциала HTTP/2, например для отправки *нескольких потоков* «запрос – ответ» по *одному соединению*. Однако при использовании gRPC вам недоступен REST из-за применения HTTP!

Еще один принцип REST, способный помочь избежать связанности между клиентом и сервером, – это концепция *гипермедиа как двигатель состояния приложения* (часто сокращается как HATEOAS – hyermedia as the engine of application state).

Гипермедиа – это расширение так называемого гипертекста или возможность открывать новые веб-страницы через ссылки на другие данные в различных форматах (например, текст, изображения, звуки). Идея, лежащая в основе HATEOAS, заключается во взаимодействии *клиентов* с *сервером* (потенциально приводящим к переходам состояний) *через ссылки на другие ресурсы*.

Клиенту не нужно знать, где именно на сервере содержится необходимая информация. Вместо этого он использует ссылки и перемещается по ним, чтобы найти то, что ему нужно.

NB: Несмотря на ясность целей HATEOS, я не видел достаточно доказательство того, что дополнительная работа по внедрению этого стиля REST приносит ощутимые выгоды в долгосрочной перспективе. Эта концепция не особо прижилась. Вероятно модель просто не работает для созданных нами в итоге систем

Полезные нагрузки REST через HTTP на самом деле могут быть более компактными, чем SOAP, так как REST поддерживает альтернативные форматы, такие как JSON или даже бинарный, но он все равно далеко не такой экономичный, каким мог бы быть Thrift.

Все основные протоколы HTTP, используемые в настоящее время, требуют применения протокола передачи данных TCP, который *неэффективен* по сравнению с другими сетевыми протоколами.

Ограничения, накладываемые на HTTP из-за требования использовать TCP, устраняются. Проткол HTTP/3, которые в настоящее время находится в процессе разработки, стремится перейти на использование более нового протокола QUIC.

REST через HTTP представляется разумным *стандартным* выбором для взаимодействия между сервисами [1, стр. 156].

Где использовать API на основе *REST через HTTP* представляется очевидным выбором для *синхронного* интерфейса по модели *«запрос – ответ»*, если вы хотите разрешить доступ *как можно большему количеству клиентов.* Это понятный стиль интерфейса, с которым многие знакомы, и он гарантирует совместимость с огромным разнообразием технологий [1, стр. 156].

API на основе REST превосходны в ситуациях, требущих крупномасштабного и эффективного кэширования запросов. Именно по этой причине они стали очевидным выбором для предоставления API внешним потребителям или клиентским интерфейсам. Однако они могут проигрывать по сравнению с более эффективными протоколами связи, и, хотя допустимо создавать протоколы *асинхронного взаимодействия* поверх API на основе REST, это не совсем подходящий вариант по сравнению с альтернативами для общего взаимодействия между микросервисами.

3.1.3. GraphQL

В последние годы язык GraphQL приобрел большую популярность во многом благодаря тому, что он позволяет устройству на стороне клиента определять вызовы, способные избежать выполнения нескольких запросов для получения одной и той же информации. Это может значительно улучшить производительность ограниченных по производительности клиентских устройств, а также поможет избежать необходимости внедрять индивидуальную агрегацию на стороне сервера.

GraphQL позволяет выдавать один запрос, способный извлекать всю необходимую информацию. Чтобы это сработало, нужен микросервис, предоставляющий конечную точку GraphQL клиентскому устройству. Эта точка GraphQL служит входом для всех клиентских запросов и предлагает схему для использования клиентскими устройстами. Уменьшая количество вызовов и объем данных, извлекаемых клиентским устройством, вы можете аккуратно решать некоторые проблемы, возникающие при создании пользовательских интерфейсов с микросервисными архитектурами.

Проблемы Сегодня все основные технологии поддерживают GraphQL. Для начала клиентское устройство может выдавать динамически изменяющиеся запросы, и бывают команды, у которых возникают проблемы с запросами GraphQL, вызывающими значительную нагрузку на серверную часть. При сравнении GraphQL с чем-то вроде SQL мы видим аналогичную проблему. *Ресурсоемкий* SQL-запрос может вызывать значительные проблемы для базы данных и *потенциально оказать серьезное влияние на систему в целом.* Та же проблема возникает и с GraphQL.

Разница в том, что при использовании SQL применются такие инструменты, как планировщик запросов для БД, которые помогают диагностировать проблемные запросы, в то время как аналогичный недостаток с GraphQL сложнее отследить.

Кэширование оказывается более сложным по сравнению с обычными HTTP-API на основе REST. При использовании API на основе REST можно задать один из множества ответов заголовков, чтобы помочь клиентским устройствам или промежуточным кэшам, таким как сети доставки контента (content delivery networks, CDNs), кэшировать ответы, чтобы их не нужно было запрашивать снова. При использовании GraphQL организовать процесс таким же образом невозможно. Советы по этому вопросу сводятся к тому, чтобы просто связать идентификатор с каждым возвращаемым ресурсом, а затем заставить клиентское устройство кэшировать запрос по этому идентификатору. Без лишней работы это делает использование CDN или кэширование обратных прокси невероятно сложным.

Еще одна проблема заключается в том, что, хотя GraphQL теоретически способен обрабатывать процесс записи, он подходит для этого не так хорошо, как для чтения. Это приводит к ситуациям, в которых команды используют GraphQL для чтения, а REST – для записи.

GraphQL дарит ощущение, что вы просто работаете с данными. Это может укрепить мнение о том, что микросервисы, с котоорыми вы взаимодействуете, представляют собой всего лишь оболочки баз данных. Микросервисы предоставляют функциональные возможности через сетевые интерфейсы. Не стоит думать, что ваши микросервисы – это не более чем API для базы данных, только потому, что вы используете GraphQL.

Где использовать Лучшее место для использования GraphQL – по периметру системы, где предоставляется функциональность внешним клиентам. Если у вас есть внешний API, который часто требует от внешних клиентов совершать множество вызовов для получения необходимой им информации, то GraphQL поможет сделать API намного более эффективным и удобным.

По сути, GraphQL – это механизм агрегации и фильтрации вызовов, поэтому в контексте микросервисной архитектуры он будет использоваться для агрегирования вызовов нескольких нижестоящих микросервисов. Сам по себе он не может служить заменителем общей коммуникации между микросервисами.

3.1.4. Брокеры сообщений

Брокеры сообщений – это посредники, часто называемые промежуточным ПО, находящиеся между процессами для управления связью между ними. Они стали *популярным вариантом* выбора для реализации *асинхронной связи*, поскольку предлагают множество мощных возможностей.

Сообщение – это общее понятие, определяющее то, что отправляет брокер. Сообщение может содержать запрос, ответ или событие. Вместо того чтобы напрямую связываться с другим микросервисом сервис передает брокеру сообщение с информацией о том, как оно должно быть отправлено.

Топики и очереди Брокеры, как правило, предоставляют либо очереди, либо топики, либо и то и другое. Очереди обычно бывают точечными. Отправитель помещает сообщение в очередь, а потребитель считывает его из этой очереди. В топик-ориентированной системе несколько пользователей могут подписаться на топик, чтобы получить копию этого сообщения.

Потребитель может предоставлять собой один или несколько микросервисов, обычно моделируемых как *группа потребителей*. Это может быть полезно, если у вас имеется *несколько экземпляров микросервиса* и вы хотите, чтобы любой из них мог получить сообщение. Например, у сервиса «Обработчик заказов» есть три развернутых экземпляра, входящих в одну группу потребителей. Когда сообщение помещается в очередь, только один член группы получит его. Это означает, что *очередь* работает как *механизм распределения нагрузки*.

С помощью *топиков* можно создать несколько групп потребителей. Например, событие, представляющее оплачиваемый заказ, помещается в топик «Статус заказа». Копия этого события принимается как микросервисом «Склад», так и сервисом «Уведомления», которые находятся в отдельных группах потребителей. Только один экземпляр *каждой группы потребителей* увидит это событие.

На первый взгляд очередь выглядит просто как топик с одной группой потребителей. Основное различие между ними заключается в том, что, когда сообщение отправляется через очередь, сохраняется информация, кому оно предназначено. В случае с топиком эта информация скрыта от отправителя – он не знает, кто (если таковой блок вообще будет) в конечном счете получит сообщение.

Топики хорошо подходят для *событийного сотрудничества* (потому что в топиках сообщения не сохраняют информацию о получателе и потому в итоге микросервисная архитектура будет менее связанной), в то время как *очереди* больше для коммуникации в стиле «запрос – ответ» (так как асинхронной версии «запрос – ответ» запрос отправляется не прямо в сервис, а через очередь). Однако это следует рассматривать как общее руководство, а не как строгое правило [1, стр. 161].

Гарантированная доставка Брокеры сообщений могут очень полезны для асинхронной связи. Наиболее интересной особенностью для нас будет гарантированная доставка, которую так или иначе поддерживают все широко используемые брокеры. *Гарантированная доставка* описывает обязательство брокера обеспечить доставку сообщения.

С точки зрения микросервиса, отправляющего сообщение, это может быть очень полезно. Если нижестоящий адресат недоступен, *брокер будет удерживать сообщение до тех пор, пока доставка не осуществиться*, что уменьшит количество проблем для вышестоящего микросервиса.

Сравните этот процесс с синхронным прямым вызовом. Например, с HTTP-запросом: если нижестоящий адресат недоступен, вышестоящему микросервису необходимо решить, что делать с запросом: повторить вызов или отказаться?

Чтобы гарантированная доставка работала, брокер должен обеспечить хранение всех еще не доставленных сообщений в надежном месте до тех пор, пока они не будут доставлены. Чтобы выолпнить это обязательство, *брокер* обычно работает как своего рода *кластерная система*, гарантируя, что потеря одного компьютера не приведет к потере сообщения.

Большинство брокеров могут гарантировать порядок доставки сообщений, но даже в этом случае объем подобной гарантии ограничен. Например, в Kafka порядок гарантируется только в пределах одного раздела.

Некоторые брокеры предоставляют транзакции при записи. Например, Kafka позволяет записывать сообщения в несколько топиков за одну транзакцию.

Это полезно, если вы хотите убедиться, что сообщение может быть обработано потребителем, прежде чем удалять его у брокера.

Еще одна, несколько спорная, функция, обещанная некоторыми брокерами, – *однократная доставка*. Один из самых простых способов обеспечить гарантированную доставку – разрешить поторную отправку сообщения. Это может привести к тому, что потребитель увидит одно и то же сообщение несколько раз (даже если это редкая ситуация).

Лучше построить свои потребители таким образом, чтобы они были готовы к получению сообщения более одного раза и могли справиться с этой ситуацией. Очень простой пример: каждому сообщению присваивается идентификатор, который потребитель проверяет при каждом получении сообщения. Если сообщение с таким идентификатором уже было обработано, то новое может быть проигнорировано.

Популярность Kafka отчасти объясняется возможностью перемещать большие объемы данных в рамках реализации конвейеров потоковой обработки. Это может помочь перейти от пакетной обработки к обработке в режиме реального времени. С Kafka сообщения могут храниться в течение настраиваемого периода или вообще вечно. Такой подход дает возможность потребителям повторно отправлять уже обработанные ими сообщения или разрешает вновь развернутым потребителям обрабатывать отправленные ранее сообщения.

Kafka внедряет встроенную поддержку *потоковой обработки*. Вместо того чтобы использовать Kafka для отправки сообщений в специальный инструмент обработки потоков, такой как Apache Flink, теперь появилась возможность некоторые задачи выполнять внутри Kafka. Используя *потоки KSQL*, можно определить SQL-подобные инструкции, которые обрабатывают один или несколько топиков на лету. Это дает что-то похожее на динамически обновляемое материализованное представление базы данных, при этом источником данных будут топики Kafka, а не БД.

Kafka позволяет отправлять сообщения в различных форматах:

- Текстовые форматы. REST API чаще всего использует текстовый формат для тела запроса и ответа, даже если теоретически вы способны отправлять двоичные данные по протоколу HTTP. На самом деле именно так работает gRPC – использует HTTP-сервер, но отправляет данные через двоичный Protcol Buffers.
- Двоичные форматы. В то время как у текстовых форматов есть такие преимущества, как простота их чтения людьми и обеспечения высокой совместимости с различными инструментами и технологиями, мир протоколов двоичной сериализации – это место, где хочется оказаться, если вы начинаете задумываться о *размере полезной нагрузки* или об *эффективности записи и чтения полезных нагрузок*. Существующие буферы протокола часто используются вне рамок gRPC – они, вероятно, представляют собой самый популярный формат двоичной сериализации для связи микросервисов.

Я выступаю за наличие явных схем для конечных точек микросервисов по двум ключевым причинам:

- Явное представление того, что предоставляет и принимает конечая точка микросервиса.
- Как явные схемы помогают обнаружить случайные поломки конечных точек микросервиса.

Нарушения контракта можно разделить на две категории:

- Структурный разрыв – это ситуация, в которой структура конечной точки перестраивается таким образом, что потребитель становится недоступен.
- При семантическом разрыве структура конечной точки микросервиса остается неизменной, но поведение изменяется, нарушая ожидания потребителей.

3.1.5. Динамические реестры сервисов

Сервис ZooKeeper можно использовать в качестве *сервиса именования*. ZooKeeper полагается на запуск нескольких узлов в кластере для обеспечения различных гарантий. Это означает, что стоит рассчитывать на работу как минимум трех узлов ZooKeeper. Большинство функций ZooKeeper связаны с обеспечением безопасности репликации данных между этими узлами и сохранением целостности данных при сбое узлов.

По сути, ZooKeeper предоставляет иерархическое пространство имен для хранения информации. Клиенты могут вставлять новые узлы в эту иерархию, изменять или запрашивать их. ZooKeeper часто используется как общее хранилище конфигурации, поэтому в нем также хранят специфичную для сервиса конфигурацию, что позволяет выполнять такие задачи, как динамическое изменение уровней лога или отключение функций работающей системы.

На самом деле существуют лучшие решения для динамической регистрации сервисов. Они настолько эффективнее, что в настоящее время я бы активно избегал использования ZooKeeper

Как и ZooKeeper, Consul поддерживает *управление конфигурацией* и *обнаружение сервисов*. Он предоставляет HTTP-интерфейс для обнаружения сервисов. А одна из главных функциональных возможностей Consul заключается в фактическом предоставлении DNS-сервера из коробки.

Consul использует HTTP-интерфейс RESTful для всего, от регистрации сервиса до запроса хранилища ключ/значие или вставки проверок работоспособности.

Если вы работаете на платформе, управляющей рабочими нагрузками контейнеров, скорее всего, у вас уже есть механизм обнаружения сервисов. Хранилище etcd обладает возможностями, аналогичными Consul, и Kubernetes использует его для управления широким спектром конфигурационной информации.

Если в двух словах способ обнаружения сервисов в Kubernetes заключается в том, что вы разертываете контейнер в поде, а затем сервис динамически определяет, какие поды должны быть частью сервиса, путем сопоставления с образцом метаданных, связанных с подом.

Возможности, которые дает Kubernetes из коробки, вполне могут привести к тому, что вы захотите обойтись тем, что поставляется с базовой платформой, отказавшись от использования специализированных инструментов, таких как Consul. Для многих это имеет большой смысл, особенно если более широкая экосистема инструментов вокруг Consul не представляет интереса. Оданко если вы работаете в смешанной среде, где рабочие нагрузки выполняются в Kubernetes и на других платформах, то наличие специального инструмента обнаружения сервисов, который можно использовать на обеих платформах, может оказаться правильным решением.

3.2. Сервисные сети и API-шлюзы

Говоря в общем, API-шлюз расположен по периметру вашей системы и имеет дело с трафиком «север – юг». Его основная задача – управление доступом из внешнего мира к вашим внутренним микросервисам.

Поскольку API-шлюз больше ориентирован на трафик «север – юг», основной его задачей в среде микросервисов является сопоставление запросов от внешних сторон к внутренним микросервисам.

В большинстве случаев API-шлюз используется в основном для управления доступом к микросервисам организации из ее собственных клиентов с графическим интерфейсом (веб-страниц, собственных мобильных приложений) через Интернет. Здесь нет никакой «третьей стороны».

Kubernetes изначально обрабатывает сеть только внутри кластера и ничего не делает для обработки связи с самим кластером. Но в таком случае *API-шлюз*, предназначенный для внешнего стороннего доступа, *будет излишеством*

Где использовать Если речь идет только о предоставлении доступа к микросервисам, работающим в Kubernetes, можно запустить свои собственные обратные прокси-серверы. Или, что еще лучше, выбрать специализированный продукт, такой как Ambassador https://www.getambassador.io/, который был создан с нуля именно для этого.

Два ключевых примера неправильного использования API-шлюзов, с которыми сталкивался я, – это агрегирование вызовов и переписывание протоколов.

GraphQL поможет в ситуации, когда требуется выполнить ряд вызовов, а затем объединить и отфильтровать резльутаты. Если вам понадобится выполнить агрегацию и фильтрацию вызовов, тогда обратите внимание на потенциал GraphQL.

Основная проблема как с возможностью перезаписи протокола, так и с реализацией агрегации вызвов внутри API-шлюзов заключается в том, что мы нарушаем правило сохранения каналов «глупыми», а конечных точек – «умными».

В Kubernetes необходимо развертывать каждый экземпляр микросервиса в поде с его собственным локальным прокси-сервером. Отдельный модуль всегда равзертывается как единое целое, поэтому вы всегда знаете, что у вас есть доступный прокси-сервер. Более того, выход из строя одного прокси повлияет только на этот один под [1, стр. 194].

Многие реализации сервисной сети используются *прокси-сервер* Envoy https://www.envoyproxy.io/ в качестве основы для локально запущенных процессов. Envoy – это облегченный прокси-сервер на C++, часто используемый в качестве строительного блока для сервисных сетей и других типов ПО на основе прокси-серверов.

Сервисные сети подходят не для всех. Во-первых, если вы не используете Kubernetes, ваши возможности ограничены. Во-вторых, они действительно добавляют сложности. Если у вас 5 микросервисов, не уверен, что получится легко оправдать применение сервисной сети (можно поспорить о том, оправдано ли использование Kubernetes, если у вас только 5 микросервисов).

Организациям, у которых больше микросервисов, особенно если они хотят, чтобы эти микросервисы были написаны на разных языках программирования, стоит обратить внимание на сервисные сети. Однако знайте: переход между сервисными сетями достаточно проблематичный! [1, стр. 196]

API-шлюзы и сервисные сети в основном используются для обработки вызовов, связанных с HTTP. Однако все становится немного более туманным, когда вы начинаете рассматривать коммуникацию с помощью других протоколов, например используя такие брокеры сообщений, как Kafka. Обычно на этом этапе *сервисную сеть* обходят стороной – связь осуществляется непосредственно с самим брокером. Это означает, что нельзя считать, что сервисная сеть способна работать в качестве посредника для всех вызовов между микросервисами.

Ambassador довольно популярен в качестве API-шлюза для Kubernetes, а его портал отвечает требованиям автоматического обнаружения доступных конечных точек OpenAPI.

3.3. Рабочий поток

Размышляя о транзакции в контексте вычислений, мы представляем себе одно или несколько действий, которые должны произойти и которые мы хотим рассматривать как единое целое.

В базах данных транзакция используется, чтобы убедиться, что одна или несколько модификаций состояния выполнены успешно, например удаление, вставка или изменение данных.

3.3.1. Транзакции базы данных

Обычно, когда обсуждаются транзакции БД, речь идет о ACID. ACID – это аббривиатура, описывающая *ключевые свойства транзакций базы данных*, приводящие к созданию системы, на которую можно положиться для обеспечения *долговечности* и *согласованности* хранилища данных. ACID означает *атомарность* (atomicity), согласованность (consistency), изоляцию (isolation) и *устойчивость / дологовечность* (durability).

- Атомарность. Гарантирует, что все операции, предпринятые в рамках транзакции, завершатся успешно или неудачей. Если какое-либо из вносимых изменений по какой-то причине завершается неудачей, то вся операция прерывается, и это выглядит так, будто никаких изменений никогда не было,
- Согласованность. Когда в базу данных вносятся изменения, мы следим за тем, чтобы она оставалась в действительном и согласованном состоянии.
- Изоляция. Позволяет нескольким транзакциям, работать одновременно, не мешая друг другу. Это достигается за счет того, что любые промежуточные изменения состояния, внесенные во время одной транзакции, незаметны для других транзакций.
- Устойчивость (долговечность). Гарантирует, что после завершения транзакции данные не будут потеряны в случае каког-либо системного сбоя.

Стоит отметить, что не все базы данных предоставляют транзакции ACID. Все системы реляционных БД, Имейте в виду, что мы все еще можем использовать транзакции в стиле ACID при использовании микросервисов. Микросервис способен свободно использовать транзакцию ACID, например, для операций со своей собственной БД. Просто объем этих транзакций сводится к изменению состояния, происходящему локально в рамках этого единственного микросервиса [1, стр. 204].

Поскольку регистрация уже завершена, нам требуется удалить соответствующую строку из таблицы `PendingEnrollments`. С одной базой данных это выполняется *в рамках одной транзакции базы данных ACID*: либо происходят оба изменения, либо ни одно из них.

Если монолит разделить на два микросервиса, у каждого из которых будет своя БД, и внести те же самые изменения, то необходимо уже будет рассмотреть две транзакции, каждая из которых может сработать или завершиться неудачей независимо от другой.

Теперь нужно признать, что, разложив эту операцию на две отдельные транзакции базы данных, мы утратили гарантированную атомарность операции в целом.

Обычно первым вариантом, который люди начинают рассматривать, по-прежнему остается использование одной транзакции, охватывающей несколько процессов, – распределенной транзакции, но они далеко не всегда правильный путь развития.

Алгоритм двухфазной фиксации (2PC, two-phase commit – «двухфазный коммит») часто используется в попытке дать возможность вносить транзакционные изменения в распределенную систему, где может потребоваться обновление нескольких отдельных процессов в рамках общей операции. 2PC часто рассматриваются командами, переходящими на микросервисные архитектуры, как способ решения встречающихся проблем. Однако обычно они могут не только не решить ваши проблемы, но и еще больше запутать систему.

Важно подчеркнуть, что изменение не вступает в силу сразу после указания о его внесении. Вместо этого исполнитель гарантирует, что он сможет внести это изменение в какой-то момент в будущем. Чтобы гарантировать, что отредактированный статус будет применен позже, один исполнитель, вероятно, *заблокирует запись*, чтобы защитить ее от других изменений.

Если какие-либо рабочие процессы не проголосовали за фиксацию, всем сторонам должно быть отправлено сообщение об откате, чтобы убедиться, что они могут выполнить локальную очистку. Если исполнители согласились внести изменения, мы переходим к фазе фиксации.

Важно отметить, что в такой системе нет никакой гарантии, что эти фиксации произойдут одновременно. «Координатору» необходимо отправить запрос на фиксацию всем участникам, и это сообщение может прийти и быть обработано в разное время.

Чем больше задержка между «Координатором» и участниками двухфазной фиксации и чем медлнее исполнители обрабатывают ответ, тем шире может быть это окно непоследовательности. Возвращаясь к нашему определению ACID, *изоляция* гарантирует, что мы не видим промежуточных состояний во время транзакции. Но с 2PC мы *потеряем* эту *гарантию* [1, стр. 208].

Существует множество видов сбоев, связанных с 2PC. Некоторые из этих видов сбоев могут быть обработаны автоматически, но иные могут оставить систему в таком состоянии, что оператору придется устранять неполадки вручную.

Чем больше у вас участников и чем больше задержка в системе, тем масштабнее проблемы, возникающие при двухфазной фиксации. Алгортим 2PC может стать простым способом ввести огромные задержки в вашу систему, особенно если область блокировки объемна или продолжительность транзакции велика. Чем дольше длится операция, тем дольше у вас заблокированы ресурсы! [1, стр. 208]

По всем причинам, изложенным выше, я настоятельно рекомендую вам *избегать* использования *распределенных транзакций*, таких как двухфазная фиксация, для координации изменений состояния между вашими микросервисами.

Что еще можно сделать? Ну, первый вариант – просто не разделять данные на части. Если у вас есть фрагменты состояния, которыми вы хотите управлять по-настоящему атомарным и согласованным способом, и непонятно, как разумно получить эти характеристики без транзакций в стиле ACID, тогда *оставьте это состояние в отдельной БД и управляющую этим состоянием функциональность в отдельном сервисе* (или в своем монолите). Если вы находитесь в процессе поиска мест разделения своего монолита и выбора определения легких (или трудных) декомпозиций, то вполне можете решить, что разделение данных, которые в настоящее время управляются в транзакции, слишком сложно для обработки на данном этапе. Поработайте над какой-нибудь другой областью системы и вернитесь к этому позже [1, стр. 209].

3.3.2. Саги

В отличие от двухфазной фиксации *сага* по своей конструкции представляет собой алгоритм, способный координировать множественные изменения состояния, но позволяющий избежать необходимости блокировки ресурсов на длительные периоды времени. Данный шаблон делает это путем моделирования этапов как отдельных дейтсвий, которые могут выполняться независимо друг от друга.

Долгоживующие транзакции (long live transactions, LLT) могут занять много времени (минуты, часы или даже дни) и потребовать внесения изменений в базу данных.

Предлагается разбить LLT на *последовательность транзакций*, каждая из которых может обрабатываться независимо. Идея заключается в том, что продолжительнсоть каждой из этих «вспомогательных» транзакций будет короче и изменит только часть данных, на которые влияет LLT целиком. В результате в базовой БД будет гораздо меньше конфликтов, посольку объем и продолжительность блокировок значительно сократяться.

Хотя изначально саги задумывались как механизм, помогающй LLT работать с единой базой данных, они так же хорошо работают для кооридинации изменений в нескольких сервисах. Можно разбить один бизнес-процесс на набор вызовов, которые будут совершаться к взаимодействующим сервисам, – это и есть сага.

Поскольку мы разбиваем LLT на отдельные транзакции, у нас *нет атомарности на уровне самого шаблона*. Она *есть для каждой отдельной транзакции внутри общей саги*, поскольку каждая из них может быть связана с изменением ACID-транзакции, если это необходимо.

Поскольку сага разбивается на отдельные транзакции, нам необходимо придумать, как справиться с *отказом*, то есть восстановиться в случае сбоя. В оригинальной статье о сагах описаны два типа восстановления: обратное и прямое.

Обратное восстановление включает в себя возврат сбоя и последующую очистку – откат. *Прямое восстановление* позволяет начать с места, где произошел сбой, и продолжить обработку. Для этого у нас должна быть возможность повторять транзакции. Это, в свою очередь, подразумевает, что в системе сохраняется достаточно информации для совершения повторной попытки.

Очень важно отметить, что сага позволяет восстанавливаться после *бизнес-сбоев*, а не *технических*. Например, попытка принять платеж от клиента, когда у клиента недостаточно средств, – это бизнес-сбой, с которым шаблон «Сага» должен справиться. С другой стороны, если время ожидания «Платежного шлюза» истекает или выбрасывается ошибка «500 Internet Service Error», тогда это технический сбой, которые следует обрабатывать отдельно.

Чтобы откатить процесс, необходимо запустить компенсирующую транзакцию для каждого уже зафиксированного шага в нашей саге.

Обратите внимание, что эти компенсирующие транзакции могут вести себя иначе, чем при обычном откате базы данных. Откат БД происходит до фиксации, а после него – транзакции будто никогда и не было. В нашем примере транзакция уже произошла, и теперь мы создаем новую транзакцию, отменяющую изменения, внесенные исходной.

Поскольку не всегда есть возможность полностью отменить транзакцию, будем называть эти компенсирующие транзакции *семантическими откатами*.

В *оркестрованных сагах* используется центральный координатор (далее оркестратор) для определения порядка выполнения и запуска любого требуемого компенсирующего действия. Оркестратор контролирует, что происходит и когда, и вместе с этим проявляется хорошая степень наглядности того, что творится с любой конкретной сагой.

Здесь центральный «Обработчик заказов», играющий роль оркестратора, координирует процесс выполнения заказов. Ему известно, какие сервисы необходимы для выполнения операции, и он определяет, когда совершать вызовы этих сервисов. Если вызовы завершаются неудачей, им принимается решение о дальнейших действиях. В целом *в оркестрованных сагах*, как правило, широко используются взаимодействия «запрос – ответ» между сервисами: «Обработчик заказов» отправляет запрос сервисам и ожидает ответа, чтобы узнать, был ли запрос успешным, и предоставить результаты запроса.

Однако есть несколько недостатков, которые следует учитывать. Во-первых, по своей природе данные подход – с несколько повышенной связанностью. Сервис «Обработчик заказов» должен иметь информацию обо всех связанных сервисах, что приводит к более высокой степени доменной связи. Хотя она по своей сути не является плохой лучше бы по возможности свести ее к минимуму.

Во-вторых, логика, которую необходимо внедрить в сервисы, может начать поглощаться оркестратором. Важно, чтобы вы по-прежнему рассматривали сервисы, составляющие эти организованные потоки, как объекты, имеющие свое собственное локальное состояние и поведение.

Хореографическая сага направлена на распределение ответственности за функционирование саги между несколькими взаимодействующими сервисами. Если оркестрация – это командно-контрольный подход, то хореографические саги представляют собой архитектуру «доверяй, но проверяй».

События не отправляются *в* микросервис, а просто запускаются, и микросервисы, которые заинтересованы в этих событиях, могут получать их и действовать соответсвтующим образом.

События могут облегчить параллельную обработку. Как правило, используется какой-то *брокер сообщений* для управления надежной трансляцией и доставкой событий. Возможно, что несколько микросервисов среагируют на одно и то же событие, и именно здесь вы могли бы использовать *топик*. Стороны, заинтересованные в определенном типе событий, будут подписываться на определенный топик, не беспокоясь, откуда вязлись эти события, а брокер гарантирует долговечность топика и то, что события в нем успешно доставляются подписчикам.

В этой архитектуре ни у одного сервиса *нет информации о каком-либо другом микросервисе*. Им нужно знать только, что делать, когда получено определенное событие – мы резко сократили количество предметных связанностей. По сути, это делает архитектуру гораздо менее связанной [1, стр. 220].

Отсутствие четкого представления нашего бизнес-процесса достаточно плохо, но нам также не хватает способа узнать, в каком состоянии находится сага. Отсутствие единого центра для обсуждения статуса саги – большая проблема.

Как правило хореографические саги предполагают интенсивное использование событийного взаимодействия, что не получило широкого понимания. Но, по моему опыту, дополнительную сложность, связанную с отслеживанием прогрсса саги, почти всегда перевешивают преимущества, касающиеся наличия более слабо связанной архитектуры.

3.4. Сборка

CI – это ключевая практика, позволяющая быстро и легко вносить изменения и безболезненно осуществлять переход к микросервисам. Инструмент CI при правильном использовании поможет вам выполнить непрерывную интеграцию, но применение таких инструментов, как Jenkins, CircleCI, Travis, или одного из многих других вариантов не гарантирует, что вы действительно правильно выполняете процесс CI.

Наличие ветвей или форков с очень коротким временем жизни (менее суток) перед объединением в магистраль и менее трех активных ветвей в целом – важные аспекты непрерывной доставки, и все они способствуют повышению производительности.

3.4.1. Сопоставление исходного кода и сборок с микросервисами

Один гигантский репозиторий – одна гигантская сборка Если начать с самого простого варианта, то можно свести все воедино. У нас есть единый гигантский репозиторий, в котором

хранится весь код, и есть единая сборка. Любая фиксация исходного кода репозитории приведет к запуску сборки, где мы выполним все шаги проверки, связанные со всеми микросервисами, и создадим несколько артефактов, привязанных к одной и той же сборке.

По сравнению с другими подходами, на первый взгляд это кажется намного проще: меньше репозиториев, о которых нужно заботиться, и концептуально более простая сборка. С точки зрения разработчика все тоже довольно легко. Я просто выполняю фиксацию кода. Если приходится работать с несколькими сервисами одновременно, необходимо позаботиться только о фиксакции.

Эта модель способна отлично работать, если вы согланы с идеей поэтапных релизов, когда вы не возражаете против одновременного развертывания нескольких сервисов. В целом такой модели стоит избегать, но на очень ранней стадии проекта, особенно если работает только одна команда, эта модель может иметь смысл, но не долго

Если я внесу однострочное изменение в один сервис, например, изменив поведение только в сервисе «Пользователь», *все остальные сервисы подвергнуться проверке и сборке*. Это займет больше времени, чем нужно, а я ожидаю, что тестирование не потребуется. Однако более тревожным будет незнание, какие артефакты следует или не следует развертывать. Определить, какие сервисы действительно изменились, просто прочитав сообщения о фиксации, очень сложно. Организации, использующие данные подход, часто возвращаются к простому комплексному развертыванию, чего мы на самом деле хотим избежать [1, стр. 234].

Я почти никогда не видел, чтобы этот подход использовался на практике, за исключением самых ранних сатдий проектов.

Шаблон: один репозиторий на один микросервис (мультирепозиторий) При использовании шаблона одного репозитория для одного микросервиса (более известного как шаблон «Мультирепозиторий») исходный код для каждого микросервиса храниться в его собственном репозитории.

То есть другими словами у каждого микросервиса свой репозиторий кода. Любое изменение в репозитории исходного кода сервиса «Пользователь» запускает соответствующую сборку, и, если это пройдет, появится ноавя версия микросервиса «Пользователь», доступного для развертывания.

При необходимости повторного использования кода в разных репозиториях этот код должен быть упакован в *библиотеку*, которая затем становится *явной зависимостью* нижестоящих микросервисов.

Если требуется преобразовать библиотеку «Соединение», придется внести изменения в соответствующий репозиторий исходного кода и дождаться завершения его сборки, что даст новую версию артефакта. Чтобы фактически развернуть новые версии сервисов «Инвойс» и «Зарплата» с помощью новой вариации библиотеки, необходимо преобразовать используемую ими версию библиотеки «Соединение».

Учтите, если вам потребуется развернуть новую версию библиотеки «Соединение», необходимо будет также развернуть новые сборки сервисов «Инвойс» и «Зарплата».

Итак, как еще можно внести изменения в нескольких репозиториях, помимо повторного использования кода через библиотеки? Так как код располагается в нескольких репозиториях приходится разбивать изменение на две процедуры фиксации – одна для сервиса «Запасы» и вторая для «Доставки».

Разделяя эти изменения, мы рискуем вызвать проблемы, если одна фиксация завершиться неудачей, а другая сработает. Например, мне требуется внести две модификации, чтобы откатить изменение, и если кто-то за это время успел провести проверку, то это может оказаться непросто.

Многие считают отсутствие атомарного развертывания при этом серьезным недостатком. Если вы постоянно вносите изменения в несколько микросервисов, то границы сервиса могут находиться в неожиданных местах, что, скорее всего означает слишком высокую связанность между сервисами. Мы же пытаемся оптимизировать архитеутуру и границы микросервисов, чтобы *преобразования с большей вероятнотсью применялись в пределах границ.* Сковзные изменения должны быть исключением, а не нормой

Где использовать Применение подхода «один репозиторий на микросервис» работает одинаково хорошо для небольших и больших команд. Но если вы заметите, что много изменений вносится за пределами микросервиса, то это, скорее всего, не лучший вариант для вас, и шаблон *монорепозитория* станет более подходящим.

Шабло: монорепозиторий При *монорепозиторном* подходе код для нескольких микросервисов (или других типов проектов) хранится *в одном репозитории исходного кода.*

Хотя у данного подхода есть и некоторые другие преимущества, такие как улучшенная видимость чужого кода, в качестве основной причины принятия этого шаблона часто упоминается возможность повторного использования кода и внесения изменений, влияющих на множество различных проектов.

Возможность сделать атомарную фиксацию в нескольких сервисах не дает атомарного развертывания.

Некоторые организации, работающие в очень больших масштабах (Google, Microsoft, Facebook, Uber), обнаружили, что подход с монорепозиториями – отличный вариант для них. По моему опыту, основные преимущества подхода с монорепозиториями – более детализированное повторное использование и атомарные фиксации, но по мере масштабирования *подход с одним репозиторием на микросервис (мультирепозитории) более прост.* На текущий момент лучше использовать мультирепозитории [1, стр. 246].

3.5. Развертывание

Развертывание однопроцессного монолита – это просто. Микросервисы с их взаимозависимостью и богатством технологических возможностей – совсем другая история.

До сих пор при обсуждении микросервисов мы говорили о них в логическом смысле, а не в физическом. Логический взгляд на архитектуру обычно абстрагирует от основных проблем физического развертывания.

В действительности у нас будет скорее всего *более одного экземпляра каждого сервиса.* Наличие нескольких экземпляров позволяет обрабатывать большую нагрузку, а также теоретически повышает надежность системы, поскольку отказ одного экземпляра переносится легче.

Количество необходимых экземпляров будет зависеть от характера приложения – вам нужно будет оценить требуемую избыточность, ожидаемые уровни нагрузки и т.п., чтобы получить приемлемое количество экземпляров.

Если у вас по соображениям надежности есть несколько экземпляров сервиса, вы, вероятно, захотите убедиться, что все они не находятся на одном и том же аппаратном обеспечении. В

дальнейшем может потребоваться, чтобы имеющиеся разные экземпляры были распределены не только по нескольким машинам, но и по разным центрам обработки данных (data center, DC), что позволит обеспечить защиту от блокировки всего дата-центра.

Требуется, чтобы микросервис скрывал свое внутренее управление состоянием, поэтому любая используемая в данном ключе БД считается скрытой внутри микросервиса. Это приводит к часто повторяемой матре «не используйте базы данных совместно».

Несколько *экземпляров* одного и того же микросервиса могут *совместно* использовать базу данных. Здесь данные совместно используются разными экземплярами *одного и того же* микросервиса. Логика доступа к состоянию и управлению им по-прежнему хранится в рамках одного логического микросервиса [1, стр. 250].

3.5.1. Принципы развертывания микросервисов

Основные идеи:

- Изолированное выполнение. Запускайте экземпляры микросервиса изолированным образом, чтобы у них были свои собственные вычислительные ресурсы и их выполнение не могло повлиять на другие экземпляры микросервиса, работающие поблизости.
- Сосредоточтесь на автоматизации. По мере увеличения количества микросервисов автоматизация становится все более важной. Сосредоточтесь на выборе технологии, обеспечивающей высокую степень автоматизации, и сделайте ее ключевой частью своей культуры производства.
- Инфрастуктура как код. Представьте конфигурацию вашей инфрастуктуры, чтобы упростить автоматизацию и обеспечить обмен информацией. Сохраните этот код в системе управления версиями, чтобы можно было воссоздать среды.
- Развертывание без простоя. Расширьте возможности независимого развертывания и убедитесь, что развертывание новой версии микросервиса может быть выполнено без каких-либо простоев для пользователей вашего сервиса.
- Управление желаемым состоянием. Используйте платформу, поддерживающую ваш микросервис в определенном состоянии, при необходимости запуская новые экземпляры в случае сбоев или увеличения трафика.

Изолированное выполнение Может возникнуть соблазн, просто поместить все экземпляры микросервиса на одну машину. Сугубо с точки зрения управления хостом эта модель проще. Если на одном хосте размещено больше сервисов, рабочая нагрузка на управление узлом не возрастает с увеличением числа сервисов.

Здесь есть проблемы. Первая – это может затруднить мониторинг. При отслеживании работы центрального процессора нужно ли отслеживать ЦП одного сервиса независимо от других? Или стоит проконтролировать ЦП хоста в целом?

Если один сервис находится под значительной нагрузкой, это может привести к сокращению ресурсов, доступных другим частям системы.

Развертывание сервисов также осложняется, поскольку обеспечить его независимость этого развертывания в данном случае – та еще головная боль. Например, если у каждого микросервиса разные (и потенциально противоречивые) зависимости, которые необходимо установить на общем хосте, как можно заставить это работать?

Такая модель также может препятствовать автономии команд. Если сервисы для разных команд установлены на одном хосте, кто будет настраивать его для их сервисов? По всей видимости, этим займется централизованная команда, а это означает, что для развертывания сервисов потребуется больше координации.

Хотелось бы запускать микросервисы изолированно. Каждый экземпляр микросервиса получает собственную изолированную среду выполнения. Он может устанавливать свои зависимости и иметь набор обособленных ресурсов.

С появлением контейнеризации для создания изолированной среды выполнения появилось больше возможностей, чем когда-либо прежде. В целом мы переходим от использования дорогих выделенных физических машин, обеспечивающих наилучшую изоляцию, к экономичным и быстрым в реализации контейнерам, предлагающим более слабую изоляцию.

При развертывании микросервисов на более абстрактных платформах, например AWS Lambda или Heroku, изоляция была бы обеспечена. В зависимости от характера самой платформы можно ожидать, что экземпляры *микросервиса* в конечном счете будет работать *внутри контейнера* или *выделенной виртуальной машины* незаметно.

В целом изоляция в сфере *контейнера* улучшилась в достаточной степени, чтобы сделать их *более естественным выбором для рабочих нагрузок микросервисов*. Разница в изоляции между контейнерами и виртуальными машинами сократилась до такой степени, что для подавляющего большинства рабочих нагрузок контейнеры «достаточно хороши» [1, стр. 259].

Внедрение культуры автоматизации представляет собой ключевой фактор, если вы хотите держать под контролем сложности микросервисных архитектур.

Инфраструктура как код (Infrastructure as code, IaC) – это концепция, при которой ваша инфраструктура настраивается с помощью машиночитаемого кода. Эта концепция говорит о том, как должна осуществляться автоматизация. Определяя инфраструктуру с помощью кода, эту конфигурацию можно контролировать по версиям, тестировать и повторять по желанию.

Здесь могут пригодиться такие концепции, как последовательные обновления, и это одна из областей, где использование такой платформы, как Kubernetes, значительно облегчает вашу жизнь. При таком обновлении ваш микросервис не отключается полностью до развертывания новой версии, вместо этого количество более старых экземпляров вашего сервиса постепенно сокращается по мере увеличения количества новых, работающих с актуальными версиями вашего ПО. Однако если единственное, что вам нужно, – инструмент для развертывания без простоя, то внедрение Kubernetes, скорее всего, будет огромным излишеством.

Управление желаемым состоянием – это возможность указать требования к инфрастуктуре, которые вы предъявляете к своему приложению, и поддерживать их без ручного вмешательства. Если работающая система изменяется таким образом, что желаемое состояние больше не поддерживается, базовая платформа предпринимает необходимые шаги для возвращения системы в желаемое состояние.

В качестве простого примера можно указать количество экземпляров, требуемых для микросервиса, а также необходимый этим экземплярам объем памяти и ядер процессора. Некоторая базовая платформа принимает эту конфигурацию и применяет ее, приводя систему в желаемое состояние. Платформа должна среди прочего, определить, на каких машинах есть свободные ресурсы, которые можно выделить для запуска требуемого количества экземпляров. Если один из этих экземпляров умирает, платформа распознает, что текущее состояние не соответствует желаемому, и принимает соответствующие действия, запуская запасной экземпляр.

Вся прелесть управления желаемым состоянием в том, что сама платформа отвечает за поддержку этого самого состояния. Это освобождает как разработчиков, так и операторов от необходимости беспокоится, как именно все выполняется, – от них просто требуется сосредоточиться на правильном определении желаемого состояния в первую очередь. Это также означает, что в случае возникновения таких проблем, как смерть экземпляра, сбой аппаратного обеспечения или закрытие дата-центра, платформа может решить их без вмешательства человека.

Все менее распространенным вариантом становится развертывание микросервисов непосредственно на физических машинах. Под «непосредственно» я подразумеваю, что между вами и аппаратным обеспечением нет никаких уровней виртуализации или контейнеризации.

Если на физической машине запущен единственный экземпляр микросервиса, а процессор, память или ввод-вывод, предоставляемые оборудованием, используются только наполовину, то оставшиеся ресурсы тратятся впустую. Эта проблема привела к виртуализации большей части вычислительной инфраструктуры, что позволяет сосуществовать *нескольким виртуальным машинам на одной физической*. Такой подход позволяет значительно повысить эффективность использования инфраструктуры, что дает очевидные экономические преимущества.

В целом прямое развертывание микросервисов на физических машинах почти не встречается в настоящее время, только в случае очень специфических требований.

Виртуализация преобразила ЦОДы, позволив нам разбить существующие *физические компьютеры* на более мелкие, но *виртуальные машины*.Традиционная виртуализация, такая как VMware или используемая основными провайдерами облачных услуг управляемая инфрастуктура виртуальных машин, дала огромные преимущества в повышении эффективности использования вычислительной инфраструктуры при одновременном снижении накладных расходов на управление хостом.

Вы можете перенаправить части ресурсов ЦП, памяти, I/O и хранилища каждой виртуальной машине, что в нашем конексте позволяет разместить гораздо больше изолированных сред выполнения для экземпляров микросервисов на одну физическую машину.

Каждая ВМ содержит *полноценную операционную систему* и набор ресурсов, которые могут использоваться программным обеспечением, запущенным внутри виртуальной машины. Развертывание каждого экземпляра на отдельной ВМ гарантирует очень хорошую степень изоляции между экземплярами.

По мере установки все большего количества виртуальных машин на одном и том же аппаратном обеспечении вы заметите, что получаете все меньшую отдачу с точки зрения вычислтиельных ресурсов, доступных самим ВМ.

Виртуализация типа 2 – это вид, реализуемый AWS, VMware, Xen и KVM. В нашей физической инфрастуктуре есть операционной система хоста, в которой мы запускаем *гипервизор* (VirutalBox, VMware Workstation etc.). У гипервизора есть две ключевые задачи:

- распределение ресурсов (процессор и память) из физического хоста на виртуальный,
- управление самими виртуальными машинами.

Внутри виртуальных машин мы получаем то, что выглядит как совершенно *разные хосты*. Они способны запускать собственные операционные системы со своими ядрами. Их можно считать почти герметично запечатанными машинами, которые гипервизор изолирует от базового физического хоста и других ВМ.

Чем большим количеством узлов управялет гипервизор, тем больше ресурсов ему требуется. В определенный момент эти издержки становятся препятствием для дальнейшего дробления физической инфрастуктуры.

Если вам нужны более строгие уровни изоляции, которые могут обеспечить контейнеры, или у вас нет возможности контейнеризировать свое приложение, виртуальные машины могут стать отличным выбором.

Идея контейнера, объединенная с поддерживающей платформой оркестрации контейнеров Kubernetes, стала для большинства предпочитаемым решением в вопросе масштабирования микросервисных архитектур.

Контейнер способен запускать собственную ОС, но она будет использовать часть общего ядра – именно в нем находится дерево процессов для каждого контейнера. Операционная система нашего хоста может запускать Ubuntu, а наше контейнеры – CentOS при условии, что они оба работают как часть одного и того же базового ядра.

Контейнеры используют виртуализацию на уровне ядра операционной системы, то есть другими словами контейнеры, в отличие от виртуальных машин, используют *ядро операционной системы* хоста *совместно* [2, стр. 18].

С контейнерами Windows у вас также есть возможность обеспечить серьезную изоляцию, запустив контейнеры внутри их собственной виртуальной машины Hyper-V. Следует учитывать, что изоляция Hyper-V, скорее всего, с точки зрения времени развертывания и стоимости выполнения будет ближе к обычной виртуализации.

Контейнеры как концепция прекрасно подходят для микросервисов. Мы получаем изоляцию, но по приемлемой цене, а также возможность скрывать базовые технологии, что позволяет смешивать различные технологические стеки [1, стр. 275].

«Бессерверный» на самом деле представляет собой *обобщающий термин* для множества различных технологий, где, с точки зрения использующего их человека, базовые компьютеры не имеют значения. У вас забирают детали управления и настройки машин [1, стр. 278].

Модель FaaS стала настолько важной частью бессерверной модели, что для многих эти два термина взаимозаменяемы. Это досадно, поскольку упускается из виду важность других бессерверных продуктов, таких как базы данных, очереди, решения для хранения данных и т.п.

Именно продукт AWS Lambda вызвал интерес к FaaS. Вы развертываете некоторый код (функцию), который находится в состоянии бездействия, пока не произойдет что-то, что вызовет этот код.

Вы сами решаете, что будет этим триггером [1, стр. 279]:

- файл, поступивший в определенное местоположение,
- элемент, появляющийся в очереди сообщений,
- вызов, поступающий по протоколу HTTP
- или что-то еще.

При появлении импульса функция запускается, а когда сигнал прекращается – она завершается. Код, который не выполняется, ничего не стоит в денежном выражении – вы платите только за то, что используете. Это делает FaaS отличным вариантом для ситуаций, когда у вас низкая или непредсказуемая нагрузка. По сути, использование платформы FaaS, как и многих других бессерверных предложений, позволяет резко сократить объем операционных издержек, о которых вам нужно беспокоиться.

Все известные реализации FaaS используют некую контейнерную технологию. Это скрыто от пользователя. Вы просто предоставляете некоторую упакованную форму кода. Однако это означает, что вам не хватает определнной степени контроля над запускаемыми процессами. Например, в Google Cloud, Auzre и AWS можно управлять только объемом памяти, выделяемой для каждой функции. Это, в свою очередь, подразумевает, что во время выполнения вашей функции выделяется определенное количество ресурсов ЦП и I/O, но вы не можете контролировать эти аспекты напрямую. В конечном счете придется выделить больше памяти для функции, даже если она в этом не нуждается, просто чтобы получить необходимый объем загрузки процессора.

Если требуется провести большую тонкую настройку доступных для ваших функций ресурсов, то на данном этапе FaaS, скорее всего, не станет для вас оптимальным вариантом.

Еще одно ограничение, о котором следует знать, заключается в том, что для вызовов функции могут быть установлены лимиты по времени выполнения. Например, продолжительность выполнения облачных функций Google в настоящее время ограничено 9 минутами, в то время как функции AWS Lambda могут выполняться до 15 минут. Функции Azure при желании могут работать вечно. Лично я думаю, что если ваши функции работают в течение длительного времени, то для решения проблем они не подходят.

На данный момент некоторым средам выполнения требуется много времени для запуска. Это называют холодным запуском. JVM и .NET сильно страдают от этого недостатка, поэтому время холодного запуска функций, использующих эти среды выполнения, часто оказывается значительным.

Тем не менее, если это вызывает беспокойство, использование языков с незначительным «временем раскрутки» (Go, Python, Node и Ruby) может эффективно решить проблему.

Обычно платформы жестко ограничивают максимальное количество вызовов, что может приводить к перегрузкам инфрастуктуры.

3.6. Сопставление FaaS с микросервисами

Одна функция на один микросервис Один экземпляр микросервиса может быть развернут как одна функция. Такой подход сохраняет концепцию экезмпляра микросервиса как единицы развертывания.

Одна функцию на один агрегат При использовании предметно-ориентированного проектирования, возможно, уже были явно смоделированы агрегаты (набор объектов, управляемых как единая сущность, обычно ссылающихся на концепции реального мира). Одна из моделей заключается в выделении функции для каждого агрегата. Это гарантирует, что вся логика для одного агрегата будет самодостаточной внутри функции, что облегчает обеспечение последовательной реализации управления жизненным циклом агрегата.

Теперь микросервис представляет собой скорее логическую концепцию, состоящую из множества различных функций, которые могут быть развернуты независимо друг от друга (теоретически).

Здесь есть несколько предостережений. Во-первых, я бы настоятельно рекомендовал поддерживать более общий внешний интерфейс. Что касается вышестоящих потребителей, то они все еще общаются с сервисом «Расходы» и не знают, что запросы сопоставляются с агрегатами с меньшим охватом.

Вторая проблема связана с данными. Должны ли эти агрегаты продолжать использовать общую БД? Различные функции могут использовать одну и ту же базы данных, поскольку все они логически являются частью одного и того же микросервиса и управляются одной и той же командой.

Однако со временем, если потребности каждой агрегатной функции будут различаться, я бы предпочел разделить их использование данных. На этом этапе можно утверждать, что эти функции тепрь являются самостоятельными микросервисами, хотя имеет смысл по-прежнему представлять их как единый микросервис для вышестоящих потребителей.

Это сопоставление одного микросервиса с несколькими более детализированными единицами развертывания несколько искажает наше предыдущее определение микросервиса. Обычно мы рассматриваем микросервис как независимо развертываемый модуль, а теперь один микросервис состоит из нескольких различных независимо развертываемых модулей. Концептуально в этом примере микросервис становится сокрее логической, чем физической идеей.

Если вы работаете в публичном облаке и ваша проблема соответствует FaaS как модели развертывания, примените ее и пропустите Kubernetes. Нашли потрясающий PaaS, такой как Heroku, и у вас есть приложение, подходящее под ограничение платформы? Перенесите всю работу на платформу и уделите больше времени работе над своим продуктом. Для остальных же контейнеризация – это правильный путь [1, стр. 287].

3.7. Упрощенный взгляд на концепции Kubernetes

По сути, *кластер Kubernetes* состоит из двух вещей: набора машин, на которых будут выполняться рабочие нагрузки, называемые *узлами*, и набора управляющего ПО, контролирующего эти узлы и называющегося *плоскостью управления*. На узлах можно запустить *физические* или *виртуальные машины*. Вместо планирования контейнера Kubernetes планирует то, что он называет *подом* (набором контейнеров). Под состоит из одного или нескольких контейнеров, развертывающихся совместно.

Обычно в *поде* находится всего один контейнер, например, *экземпляр микросервиса*. Однако в некоторых случаях (довольно редких) развертывание нескольких контейнеров вместо может иметь смысл.

Идея *сервиса* заключается в том, что определенный *под считается эфемерным* – он может быть закрыт по целому ряду причин, в то время как сервис в целом продолжает жить. Сервис существует для маршрутизации вызовов в поды и из них может обрабатывать завершение работы подов или запуск новых. В Kubernetes вы не развертываете сервисы – вы развертываете поды, сопоставляемые с сервисом.

Далее идет *набор реплик*. С помощью набора реплик вы определяете желаемое состояние набора подов. Здесь вы говорите: «Я хочу 4 таких пода», а Kubernetes справляется с остальным.

Развертывание – это способ применения изменений к своим модулям и наборам реплик. При развертывании допускается выполнять такие действия, как выпуск непрерывных обновлений (таким образом, вы постоянно заменяете модули актуальной версией, чтобы избежать простоев), откаты, увеличение числа узлов и много другое.

Чтобы развернуть свой микросервис, вы определяете *под*, внутри которого находится ваш экземпляр микросервиса, и *сервис*, который позволит Kubernetes узнать, как будет осуществляться доступ к вашему микросервису. Затем применяете изменения к запущенным подам с помощью *развертывания* [1, стр. 291]

Если у вас есть горстка разработчиков и всего несколько микросервисов, Kubernetes, скорее всего, станет обузой, даже если вы используете полностью управляемую платформу [1, стр. 299]

3.7.1. Канареечный релиз

Идея канареечного развертывания заключается в том, что ограниченной группе клиентов доступны новые функции. Если возникает проблема с внедрением, это затрагивает только данную часть наших клиентов, но если функция работает бесперебойно, то ее можно распространить на большее количество пользователей, пока она не станет доступна всем.

Для микросервисной архитектуры переключатель настраивается на уровне отдельного микросервиса, включая (или выключая) функциональность для запросов к ней из внешнего мира или других микросервисов. Иной метод заключается в создании двух разных версий микросервиса, работающих параллельно, и использовании переключателя для маршрутизации от старой версии к новой и наоборот. Здесь канареечная реализация должна находиться где-то на пути марштрутизации/сети, а не в одном микросервисе [1, стр. 302].

В настоящее время чаще всего данные процесс обрабатывается автоматически. Инструменты, подобные Spinnaker `https://spinnaker.io/`, например, получили возможность без внешнего вмешательства увеличивать количество вызовов на основе метрик, таких как увеличение процента вызовов новой версии микросервиса, если частота ошибок находится на приемлемом уровне.

3.7.2. Параллельное выполнение

При применении *канареечного релиза* запрос на часть функциональности будет обслуживаться либо старой, либо новой версией продукта. Это означает, что мы не можем сравнить, как два варианта функциональности будут обрабатывать один и тот же запрос. Но это важно, если требуется убедиться, что новая версия работает точно так же, как старая.

При параллельном запуске вы одновременно запускаете две разные модификации одной функциональности и отправляете запрос обеим ее реализациям. При микросервисной архитектуре наиболее очевидным подходом может быть отправка вызова *двум разным версиям* одного и того же сервиса и сравнение результатов. Альтернативой может стать сосуществование *обеих реализаций* функциональности *внутри одного сервиса*, что часто упрощает сравнение.

При одновременном выполнении обеих реализаций важно понимать, что вам, скорее всего, нужны только результаты одного из вызовов. Одна реализация считается источником истины – релазиация, которой вы в настоящее время доверяете, и, как правило, это существующая версия реализации. В зависимости от характера сравниваемой при параллельном запуске функциональности вам, возможно, придется тщательно продумать данный нюанс – вы же не хотите отправлять клиенту два одинаковых обновления заказа или, например, дважды оплачивать счет!

3.8. Тестирование

Модульные тесты (юнит-тесты) обычно проверяют один вызов функции или метода [1, стр. 310]. Юнит-тесты помогают разработчикам и ориентированы на технологии, а не на бизнес. Кроме того, именно с их помощью мы надеемся обнаружить большинство своих ошибок. Модульные тесты в отношении микросервиса будут охватывать *небольшие части кода изолированно*.

Сервисные тесты предназначены для непосредственного тестирования *микросервисов*. В монолитном приложении можно было бы просто тестировать набор классов, предоставляющих *сер-*

вис пользовательскому интерфейсу. Для системы, состоящей из *нескольких микросервисов*, подобный тест будет проверять возможности *отдельного микросервиса*. Причина появления сбоя теста должна быть *в рамках только тестируемого микросервиса*. Чтобы добиться такой изоляции, необходимо отключить все внешние связанные блоки, чтобы *в поле зрения попал только сам микросервис* [1, стр. 313].

Некоторые подобные тесты могут быть такими же быстрыми, как и юнит-тесты с небольшим охватом. Но если вы решите протестировать их на реальных БД или перейти по сети к заблокированным нижестоящим блокам, время тестирования, скорее всего, увеличится. Эти тесты также охватывают больший объем, чем простой модульный, поэтому, когда они терпят неудачу, определить, что именно привело к провалу, может быть сложнее.

Сквозные тесты выполняются *для* всей вашей *системы целиком*. Часто они управляют графическим интерфейсом через браузер, но также могут легко имитировать другие виды взаимодействия с пользователем, например загрузку файла.

Нам нужны быстрая обратная связь и гарантии работоспособности системы. Юнит-тесты невелеки по охвату, поэтому, когда они терпят неудачу, можно быстро найти проблему. Они также быстры в написании и очень быстры в запуске. По мере увеличения охвата тестов мы становимся более уверенными в нашей системе, но обратная связь начинает ухудшаться, поскольку продолжительность выполнения тестов возрастает. Кроме того, их написание и обслуживание обходятся дороже.

3.9. Внедрение сервисных тестов

Нашим сервисным тестам требуется протестировать часть функций *во всем микросервисе, и только в нем*. Итак, если потребуется написать тест микросервиса «Покупатель», мы развернем экземпляр этого микросервиса и, как обсуждалось, «отрежем» микросервис «Лояльность», чтобы получить гарантии, что сбой при выполнении теста можно сопоставить с проблемой самого микросервиса «Покупатель».

Нашему набору сервисных тестов необходимо отключить нижестоящие сервисы и настроить тестируемый микросервис для подключения к сервисам-заглушкам. Затем нам потребуется настроить заглушки для отправки ответов, имитирующих реальные микросервисы.

Когда я говорю об отключении ненужных функций, я имею в виду, что мы создаем *микросервис-заглушку*, отвечающую заготовленными ответами на известные запросы от тестируемого микросервиса. Например, я мог бы сообщить заблокированному микросервису «Лояльность», что при запросе баланса клиента 103 он должен вернуть значение 15000. Тесту все равно, вызывается ли заглушка 0, 1 или 100 раз. Разновидностью этого подхода является использование *макета* вместо заглушки. Кросс-платформенные тесты можно выполнять с помощью Mountebank `https://www.mbtest.org/`.

Один экземпляр Mountebank поддерживает создание нескольких самозванцев и потому допускается его использовать для отключения нескольких нижестоящих микросервисов.

3.10. От мониторинга к наблюдаемости

Вы очень быстро заметите, что инструменты и методы, хорошо сработавшие для относительно простых однопроцессных монолитных приложений, не работают хорошо для микросервисной архитектуры.

NB: вы не сможете оценить истинный машстаб проблем от применения микросервисной архитектуры, пока не запустите систему в работу, и она не начнет обслуживать реальный трафик

Наблюдаемость системы – это степень, в которой можно понять внутреннее состояние системы по внешним выводам. На практике чем более наблюдаема система, тем легче оценить ситуацию, когда что-то идет не так.

Вам стоит рассматривать внедрение инструмента агрегации логов как *необходимое условие* для реализации микросервисной архитектуры. Агрегация логов невероятно полезна. Реализовать агрегацию логов не так уж слжоно по сравнению с другими источниками проблем, которые может принести микросервисная архитектура.

Если вы ведете логи в формате JSON, то без дополнительных инструментов человеку будет затруднительно его считывать.

Строки лога генерируются на компьютерах, на которых запущены эти экземпляры микросервиса. После локальной записи в какой-то момент данные логи пересылаются. Это означает, что метки дат в строках лога генерируются на машинах, на которых запущены микросервисы. К сожалению, нельзя гарантировать, что часы на этих разных машинах синхронизированы.

Несоответствие часов вызывает всевозможные проблемы в распределенных системах. Протокол сетевого времени (Network Time Protocol, NTP) стал наиболее широко используемым вариантом. Однако работа NTP не дает гарантий. Он лишь уменьшает расхождение, а не устраняет его. То есть мы не можем получить полностью точную информацию о времени для общего потока вызовов и понять причинно-следственную связь.

Однако основная проблема с логами – генерация *огромного* количества данных по мере увеличения микросервисов и вызовов. Нагрзуки. Невероятные объемы. Это может привести к росту затрат, поскольку потребуется больше оборудования, а также к увеличению платы, которую вы вносите поставщику услуг. И в зависимости от того, как построена ваша цепочка инструментов агрегации логов, это также может привести к проблемам масштабирования. Некоторые решения для агрегирования логов пытаются создать индекс при получении данных лога, чтобы ускорить запросы. Но поддержание индекса требует больших вычислительных мощностей, и чем больше логов вы получаете с увеличением индекса, тем больше проблем это принесет.

Система Promtheus была создана для хранения довольно простых фрагментов информации, таких как частота процессора для выбранной машины. Помните, что каждая уникальная комбинация пар «ключ – значение» меток представляет собой новый временной ряд, что значительно увеличивает объем хранимых данных.

По мере усложнения системы вам потребуется повышать качество выходных данных, предоставляемых системой, чтобы получить возможность улучшить ее наблюдаемость.

Если вы ищете системы хранить данные с высокой кардинальностью и управлять ими для более сложного наблюдения за поведением вашей системы, я бы настоятельно рекомендовал рассмотретить Honeycomb https://www.honeycomb.io/. Хотя эти инструменты часто рассматриваются как решения для распределенной трассировки, они обладают высокой способностью хранить, фильтровать и запрашивать данные с высокой кардинальностью [1, стр. 358].

3.11. Распределенная трассировка

В целом все инструменты *распределенной трассировки* работают одинаково. Локальная активность внутри потока фиксируется в *спане* (span). Эти отдельные спаны коррелируются с ис-

пользованием некоторого уникального идентификатора. Затем они отправляются в центральный коллектор (сборщик), который способен построить их в единый *трейс* (trace).

Система Jaeger по умолчанию фиксирует только 1 из 1000 вызвов. Задача – собрать достаточно информации для понимания, что делает наша система, а не перегружать ее данными.

Для запуска распределенной трассировки в вашей системе необходимо выполнить несколько действий. Во-первых, захватить информацию в спан внутри ваших микросервисов. Если вы используете стандартный API, такой как OpenTracing или более новый OpenTelemetry API, вы обнаружите, что у некоторых из сторонних библиотек и фрейворков есть встроенная поддержка этих API, и они уже будут отправлять полезную информацию (например, автоматический сбор информации о HTTP-вызовах).

В области продуктов с открытым исходным кодом Jaeger стал популярным выбором для *распределенной трассировки*. Что касается коммерческого инструментария, то можно посмотреть все тот же Honeycomb. Однако я настоятельно рекомендую вам выбрать что-то, что предназначено для поддержки OpenTelemetry API.

4. Отказоустойчивость

4.1. Идемпотентность

Идемпотентная операция – это операция, которая при многократном вызове возвращает один и тот же результат.

В *идемпотентных* операциях результат не меняется после первого применения, даже если операция впоследствии применяется несколько раз. Если операции являются индемпотентными, можно повторить вызов несколько раз без неблагоприятных последствий [1, стр. 443].

Некоторые из HTTP-методов, такие как `GET` и `PUT`, определены в спецификации HTTP как идемпотентные, но для того, чтобы это было так, необходимо, чтобы ваш сервис обрабатывал эти вызовы *идемпотентным образом*. Если сделать эти методы неидемпотентными, а вызывающиме абоненты будут думать, что они могут безопасно выполнять их повторно, вы мжоете попасть в затрудинтельное положение. Помните: одно лишь использование HTTP в качестве базового протокола не означает, что вы получаете идемпотентность автоматически!

Один из способов повысить отказоустойчивость – не класть все яйца в одну корзину. То есть убедиться, что у вас нет нескольких сервисов на одном хосте, где перебои в работе повлияют на все из них.

Но что такое *хост*? В большинстве современных ситуаций хост – виртуальная концепция. Что же делать, если *все сервисы храняться на разных хостах*, но все эти хосты на самом деле виртуальные и работают *на одном физическом сервере*? Если он упадет, можно потерять несколько сервисов. Некоторые платформы виртуализации позволяют вам распределить хосты по нескольким различным физическим блокам, чтобы уменьшить вероятность возникновения такой ситуации.

4.2. Теорема CAP

Теорема CAP говорит нам о том, что в распределенной системе есть три понятия, которые можно противопоставить друг другу [1, стр. 446]:

- *согласованность* (consistency),
- *доступность* (availability),

- *устойчивость к разделению* (partition tolerance).

Согласованность – это системная характеристика, с помощью которой мы получим один и тот же ответ, если перейдем к нескольким узлам. *Доступность* означает, что каждый запрос получает ответ, а *устойчивость к разделению* – это способность системы справляться с тем, что связь между ее частями иногда невозможна.

Реальность такова, что даже если в сети между узлами БД не было сбоя, репликация данных не происходит мгновенно. Системы, которые легко уступают в согласованности ради сохранения устойчивости к разделению и доступности, считаются *в конечном счете согласованными*. То есть ожидается, что в какой-то момент в будущем все узлы увидят обновленные данные, но это произойдет не сразу, поэтому придется смириться, что пользователи видят старые данные.

Согласованность между несколькими узлами действительно сложна. В распределенных системах мало что сложнее. Если нужно прочитать запись с локального узла БД. Как мне узнать, что он обновлен? Я должен спросить у другого узла. Но я также должен попросить первый узел БД не допускать обновления до завершения считывания второго. Другими словами, мне нужно инициировать *транзакционное чтение* между несколькими узалами БД, чтобы обеспечить согласованность. Но обычно люди не выполняют транзакционное чтение, правда? Потому что оно происходит медленно. И требует блокировок. Считывание может заблокировать всю систему.

Добиться правильной согласованности с несколькими узлами настолько нелегко, что я настоятельно, ктегорически советую при возникновении потребности в этом не пытаться изобрести решение самостоятельно. Можно использовать, например, Consul. Надо стараться строить согласованные AP-системы.

Итак, как можно пожертвовать устойчивостью к разделению? Если у нашей системы нет устойчивости к разделению, она не может работать через сеть. Другими словами, это должен быть единый процесс, работающий локально. CA-системы не существуют в мире распределенных систем [1, стр. 449].

AP-системы легче масштабируются и их проще создавать, в то время как CP-системы потребует больше работы из-за проблем с поддержкой распределенной согласованности. Но мы не всегда понимаем, какое влияние этот компромисс окажет на бизнес. Устаревшие на 5 минут записи для нашей системы инветаризации – нормально? Если да, то AP-система может стать подходящим вариантом. Но как насчет баланса, хранящегося для клиента в банке? Не зная контекста, в котором используется операция, невозможно определить, какая система будет правильной.

AP-системы в конечном счета оказываются правильным решением во многих ситуациях [1, стр. 451].

Список литературы

1. *Ньюмен С.* Создание микросервисов. – СПб.: Питер, 2024. – 624 с.
2. *Моуэт Э.* Использование Docker. – М.: ДМК Пресс, 2017. – 354 с.