Дата последней сборки документа:
23 марта 2024 г. в 22:21

Создание микросервисов

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 изме-

нится 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 изолированы 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 комапнией 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 созадавть 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 – декомпозиция 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 шаб-

лона 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. Большиство 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 блокриует 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.

2.7. 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.

2.7.1. 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.

2.7.2. 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 ощщтимые 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.

Список литературы

1. *Ньюмен С.* Создание микросервисов. – СПб.: Питер, 2024. – 624 с.