Дата последней сборки документа:
19 марта 2024 г. в 12:39

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 оши-

бок, 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 вообще 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.

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.

Список литературы

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