Модуль 1, урок 2

На этом уроке речь пойдет о микросервисной и монолитной архитектуре и стратегиях перехода с одного на другое.

Метрики результата

Стоимость разработки и развертывания	Монолит
Скорость разработки и развертывания	Монолит
Стоимость поддержки	Монолит
Гетерогенный подход	Микросервисы
Горизонтальное масштабирование	Микросервисы
Предсказуемость затрат на будущие изменения	Микросервисы

Поговорим о метриках, которые стоит рассматривать при выборе архитектуры. Первые два пункта — **стоимость** и **скорость разработки** — являются наиболее важными. И здесь выигрывает монолит. Если рассматриваемые метрики имеют для вас особое значение, стоит подумать о монолите.

Стоимость поддержки инфраструктуры. Если у вас «упал» отдельный микросервис, с помощью автоматизации вы можете его перезапускать, отслеживать бизнес-метрики, а также автоматически реплицировать.

За счет этого снижается стоимость поддержки инфраструктуры — и здесь выигрывает микросервисная архитектура.

Гетерогенный подход. Тут на 100% выигрывают микросервисы, так как в монолите тяжело применять различные стеки.

> **Пример.** Вам необходимо реализовать прокси-сервер. Для этого подойдет Golang или Envoy. Если у вас сложная бизнес-логика или тяжелые интеграции, подойдет Java или .NET, которые выигрывают за счет хорошей интеграции с проприетарными системами. Задачу можно решить с помощью микросервисной архитектуры, но все это затруднительно использовать вместе с монолитом.

- **Горизонтальное масштабирование.** Речь идет о горизонтальном масштабировании нагрузки, и в этом случае также выигрывают микросервисы. С ними вы можете позволить себе масштабировать какой-то отдельный кусочек, а с монолитом придется масштабировать все вместе.

- **Предсказуемость затрат на будущие изменения.** Лучше всего рассмотреть на примере.

> **Пример.** У вас есть тикет-система. Можно задать вопрос: где она будет использоваться? Это внутренняя разработка? Там довольно низкая конкурентная среда? Низкая скорость изменений? И здесь подойдет монолит.
> А если мы ориентируемся на широкий рынок, внешний интернет, то здесь будет огромная конкурентная среда, за счет чего — высокая скорость изменений и предсказуемость затрат.

MVP (Minimum Value Product), минимальный жизнеспособный продукт

Это такое решение, которое призвано проверить жизнеспособность идеи или продукта. Затем от него придется отказаться и написать продукт с нуля, на соответствующей архитектуре, с конкретными подходами. Главное — чтобы ваш MVP не развалился в ходе эксплуатации. Он может быть тяжелым для доработок, для развития, но в эксплуатации должен обязательно работать.

Необходимо заглушить своего внутреннего разработчика и заставить себя отказаться от MVP — не пытаться его в дальнейшем дорабатывать и развивать.

Предпосылки для миграции на микросервисы

У вас существует развитая система, хороший монолит и вы задаетесь вопросом: когда переходить на микросервисную архитектуру? Решение зависит от контекста.

- **Большая команда разработки.** Если у вас большая команда, возникают сложности управления. Существует следующее правило менеджмента: менеджер может эффективно управлять командой из 50 человек. Если команда больше — возникают сложности.

  Большая команда будет мешать друг другу, проводить много времени на конф-коллах. Это можно заметить по конкретным метрикам, когда количество времени, проведенного на собраниях, в среднем для инженера увеличивается. Это сигнал, что пора дробить команды и переходить на микросервисы.

  Однако дробить команду нужно не по микросервисам, а по фичам. В монолите же дробить по фичам тяжело из-за сильной связанности компонентов. В микросервисах дробление по фичам приводит к тому, что команде интересно принимать оптимальные решения, расширять границы микросервисов, добавлять новые и так далее, потому что микросервисная архитектура постоянно меняется и развивается.

  Если поделить команды исключительно по микросервисам, то команда будет заинтересована сохранять границы этого микросервиса. Им не нужно будет выходить за пределы, потому что это их зона ответственности, и в конечном итоге вы получите дополнительный монолит.

- **Потребность в масштабировании.** Это следующая предпосылка для миграции. На определенном этапе вы можете достичь лимита в вертикальном масштабировании. Добавлять ресурсы становится либо очень дорого, либо невозможно, и необходимо масштабироваться горизонтально. Отличный повод задуматься о микросервисной архитектуре.

Не хватает собственных вычислительных мощностей?

Облако Elastic Cloud подходит для задач любого масштаба и гибко масштабируется под требования вашего проекта. Воспользуйтесь тест-драйвом сервиса и получите 2 недели бесплатного доступа к облачным ресурсам.

Тест-драйв облака

Разнообразие технических ограничений. Мы уже говорили об этом — так называемое свойство гетерогенности. Это Golang, Java, необходимость использования проприетарных протоколов. В монолите это всегда дорого. А чтобы содержать разные стеки, нужно обучать команду пользоваться разными стеками.

Высокая сложность. Если ваша система настолько большая, что вы не можете доверить ее одному человеку, если каждый релиз вызывает сложности и вы меняете одно, а у вас «падает» другое — значит, у вас плохое покрытие тестами. А тесты выявляют проблему, но не помогают ее исправлять. Такая ситуация возникает из-за сильной связанности, и в этом случае необходимо подумать о микросервисах. Потребуется обеспечить высокую степень изоляции функционала, чтобы уменьшить сложность. Микросервисы — это способ добиться всего обозначенного.

Запрос на рефакторинг. Предположим, у вас есть система, написанная 20 лет назад, и вам стало трудно находить специалистов для работы с ней. Настало время переписывать систему. Вы можете сделать это в виде монолита, но тут есть ограничения. Первое — накладные расходы. Вам нужно содержать два instance: старый монолит и новый. Вам потребуется разрабатывать их параллельно, потому что в старом тоже могут потребоваться доработки. Такое решение будет сильно отложено по времени. Вы не сразу сможете начать им пользоваться. В то время как микросервисный подход позволит взять отдельный кусочек, отдельный фрагмент функционала в монолите, реализовать его в виде микросервиса и проксировать определенные запросы на микросервисы. Таким образом вы быстрее получите результат.

Высокие риски при изменениях в системе. Если вы работаете в финтехе и каждое падение системы — довольно большие потери в деньгах, пора задуматься о микросервисах. Если у вас упал один кусочек микросервисов, у вас не упадет вся система. Плюс вам проще это протестировать и быть уверенным в работоспособности.

Стратегии миграции

Здесь мы рассмотрим пять основных стратегий, хотя в реальности их гораздо больше. Остановимся на ключевых и пойдем по принципу от самого простого и наиболее рискованного к самому сложному и наименее рискованному.

Стратегия Strangler Fig

01 Выделить часть монолита для миграции

02 Разработать микросервис, реализующий нужную функциональность

03 Перенаправить вызовы замененных частей монолита на новый микросервис

Это метафора, придуманная Мартином Фаулером. В Австралии есть растение, которое растет вокруг ствола дерева. Оно постепенно обвивает его, в итоге полностью окружая ствол, и дерево погибает. При этом зачастую остается крона. Это хорошо иллюстрирует наш подход.

Здесь мы берем маленький кусочек монолита, реализовываем его в виде микросервиса. Заменяем функционал в монолите на микросервис. Постепенно мы заменяем весь функционал и отказываемся от монолита.

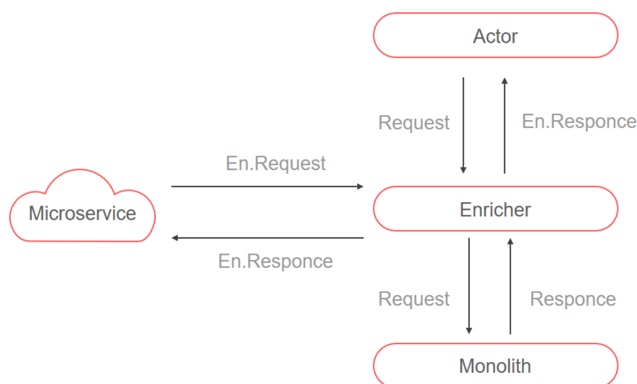
Стратегия Decorating collaborator

01 Разработать микросервис, выполняющий новую обработку

02 Выполнять перехват запросов в монолит и ответов из него

03 Извлекать нужные данные из запроса и ответа

04 Обогатить ответ от монолита обработкой в новом микросервисе



Ее стоит применять, когда монолит продолжает активно развиваться, постоянно появляется новый функционал и мы, получая запрос на новую фичу, реализуем ее в виде отдельного микросервиса.

При этом монолит у нас не меняется, микросервис обогащает запрос в зависимости от того, что необходимо, реализует какую-то логику. Мы отправляем не обогащенный запрос в монолит, он возвращает запрос обратно. Мы его обогащаем и отправляем клиенту.

Пример. Бонусы при оплате. Допустим, у нас есть некий процессинг, который совершает какие-то транзакции, тратит деньги со счета клиента. И мы хотим вознаграждать клиента за определенные суммы определенными бонусами. К нам приходит запрос: потратить 100 рублей. Наш микросервис таким же отправляет его в монолит. Монолит тратит эти деньги, отвечает микросервису, что все в порядке. Деньги списались. Микросервис начисляет клиенту 5 бонусов, складывает их в базу. Мы возвращаем клиенту 5 бонусов и то, что он покупал.

Нужно понимать, что это стратегия миграции не из информационной системы. Это стратегия миграции подходов. Применяя ее, мы не заменим нашу информационную систему, ведь именно таким образом мы реализуем только новый функционал. Чтобы полностью уйти от монолита, нам нужно применить еще один подход. Например, предыдущий.

Стратегия Branch by abstraction

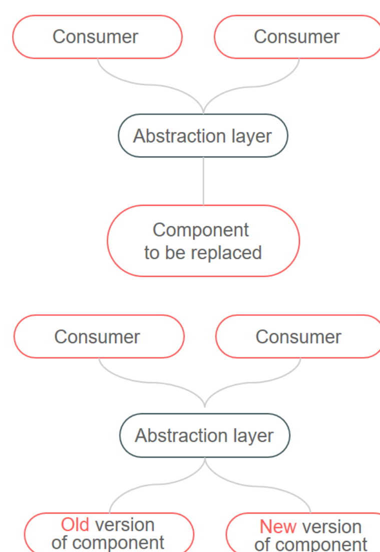
01 Создать абстракцию для заменяемой функциональности

02 Изменить клиентов существующей функциональности так, чтобы они использовали новую абстракцию

03 Создать новую реализацию абстракции, которая будет вызывать новый микросервис вместо старой части монолита

04 Переключиться на новую реализацию

05 Удалить старую реализацию и, возможно, устранить абстракцию



Берем некий кусок функционала монолита, реализуем абстракцию, которая вызывает этот кусок функционала, и учим клиента работать с этой абстракцией. После мы реализуем микросервис, который дублирует этот функционал, и меняем у абстракции версии: с функционала, который внутри монолита, на функционал, который лежит в микросервисе. Таким образом мы постепенно превращаем монолит в набор абстракций, вызывающих отдельные микросервисы. Эта стратегия менее рискованная, чем предыдущая. Здесь мы глубже погружаемся в реализацию монолита.

Стратегия Parallel processing/development

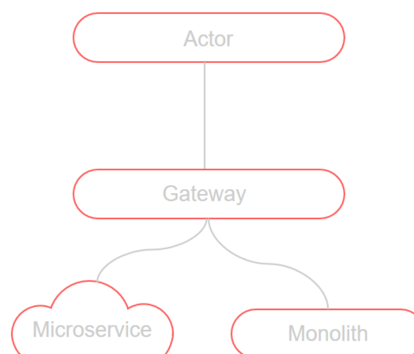
01 Разработать новую реализацию существующей функциональности

02 При поступлении вызова направлять его на обе реализации — старую и новую

03 Возвращать в ответе только один результат: от одной из реализаций

04 Верифицировать результаты новой реализации

05 Вывести старую реализацию из эксплуатации



Эта стратегия похожа на предыдущую, за исключением того, что здесь абстракции меняются на балансировщик, который проксирует запросы и распределяет их между версиями. В такой стратегии мы реализовываем отдельный микросервис, кладем его рядом с монолитом, после чего балансировщик дублирует запросы от клиента и в микросервис, и в монолит. А затем проверяет ответы от обеих систем и возвращает только один клиенту.

Такая стратегия является наименее рискованной и наиболее дорогой. Дорогая — потому что весь новый функционал необходимо реализовывать и в монолите, и в микросервисах. При этом еще добавляются накладные расходы на то, чтобы верифицировать результаты запросов.

Что учитывать при оценке

Важно понимать, что микросервисная архитектура не приносит выгоду сразу же.

Требования бизнеса. Прежде всего необходимо думать о потребностях бизнеса: что бизнесу нужно и действительно ли в его случае здесь будет выигрыш. Строим ли мы VMP или развиваем наш продукт — все это нужно принимать во внимание.

Готовность инфраструктуры. Для микросервисов необходим особый подход к архитектуре. В обязательном порядке нужно автоматизировать процесс выкатки новых релизов, обязательно нужен CI/CD и автоматизация тестирования — иначе расходы на релиз будут очень высокими и вы не почувствуете выгоды от микросервисной архитектуры.

Готовность команды. Команда должна обладать экспертизой, использовать DevOps-практики, уметь правильно распиливать монолит на микросервисы, уметь пользоваться каким-либо транспортом, который обязательно понадобится, если у вас используется микросервисная архитектура.

Доступные ресурсы на миграцию. Стоит задуматься, получится ли у вас выделить команду на то, чтобы мигрировать с монолита на микросервисы. Если не хватит ресурсов закончить миграцию, вы не получите выгоды от микросервисной архитектуры.