Современные подходы к разработке микросервисов и их архитектура

В современном мире мы всё чаще сталкиваемся с микросервисами, и многие компании уже перешли или переходят на них.

> **MSA** - *Micro Service Architecture* (архитектура микросервисов) — воплощение паттернов High Cohesion и Low Coupling
>
> **Микросервисы** — это небольшие автономные сервисы, которые выполняют свою конкретную задачу в рамках микросервисной архитектуры — подхода к разработке программного обеспечения.

Микросервисы представляют собой распределенную систему, где каждый сервис отвечает за определенную бизнес-функцию. Известные примеры использования микросервисной архитектуры — Amazon (крупнейший маркетплейс в мире) и Netflix (стриминговая платформа). Эти компании демонстрируют высокую скорость реакции и масштабируемость своих систем благодаря микросервисному подходу.

Компонент — это единица программного обеспечения, код которой может быть независимо заменен или обновлен.

Свойства микросервисов

1. **Небольшой размер.** Сервис может разрабатывать команда не более ±12 человек. Один сервис может быть полностью переписан одной командой за одну Agile-итерацию.
2. **Независимость.** Каждый микросервис работает в своем процессе и поэтому должен явно обозначить свой API. Учитывая, что другие компоненты могут использовать только этот API, и к тому же он удаленный, минимизация связей становится жизненно важной. Различные языки вводят конструкции, позволяющие явное создание независимых компонентов (например, модули в Java 9), и это перестает быть прерогативой микросервисного подхода. Использование библиотек не приветствуется, но допускается — это допущение распространяется на инфраструктурные функции вроде логирования, вызова удаленного API, обработки ошибок и тому подобного.
3. **Ориентация на бизнес-потребности.** Микросервис строится вокруг бизнес-потребности и использует ограниченный контекст (Bounded Context).

4. **Сетевое взаимодействие.** Микросервис взаимодействует с другими микросервисами по сети на основе паттерна Smart endpoints and dumb pipes.
5. **Design for failure.** Распределенная суть микросервисов обязывает использовать подход Design for failure (проектирование с учетом отказов).
6. **Минимальная централизация.** Централизация ограничена сверху на минимуме.
7. **Автоматизация.** Процессы разработки и поддержки требуют автоматизации.
8. **Итерационное развитие.** Развитие микросервиса происходит итерационно.

Определение микросервисной архитектуры

Если перефразировать Википедию, то определение микросервисной архитектуры может быть таким:

MSA — принципиальная организация распределенной системы на основе микросервисов и их взаимодействия друг с другом и со средой по сети, а также принципов, направляющих проектирование архитектуры, её создание и эволюцию.

Определение границ микросервиса

Когда начинаешь проектировать новый микросервис, определение его границ — самый важный шаг. От этого будет зависеть вся дальнейшая жизнь микросервиса, и это серьёзно повлияет на жизнь команды, отвечающей за него.

Основной принцип определения зоны ответственности микросервиса — сформировать её вокруг некоторой бизнес-потребности. И чем она компактнее (чем формализованнее её взаимоотношения с другими областями), тем проще создать новый микросервис.

Когда границы микросервиса заданы и он выделен в отдельную кодовую базу, защитить эти границы от постороннего влияния не составляет труда. Далее внутри микросервиса создают свой микромир, опираясь на паттерн «ограниченного контекста». В микросервисе для любого объекта, для любого действия может быть своя интерпретация, отличная от других контекстов.

Проблемы неправильных границ

Если границы оказались неправильными, то изменение функциональности в новом микросервисе ведет к изменению функциональности в других микросервисах. В результате «поплывут» интерфейсы всех зависимых микросервисов, а за ними интеграционные тесты. Всё превращается в снежный ком. А если эти микросервисы ещё и принадлежат разным командам, то начинаются межкомандные встречи, согласования и тому подобное. Так что правильные границы микросервиса — это основа здоровой микросервисной архитектуры.

Подход Monolith First

Чтобы избежать ошибок, нужно их сначала продумать. Подход **Monolith First** предполагает, что вначале систему развивают в традиционной парадигме (монолит), а когда появляются устоявшиеся области, их выделяют в микросервисы. Этот подход рекомендуется для постепенного перехода к микросервисной архитектуре.

Такой подход к постепенному формированию набора микросервисов похож на итерационное развитие, используемое в Agile, ещё его называют «эволюционным проектированием» (Evolutionary Design).

Интеграция микросервисов

Интеграция микросервисов обходится без ESB (Enterprise Service Bus — сервисная интеграционная шина). ESB — это связующее программное обеспечение, которое обеспечивает передачу сообщений между компонентами и преобразовывает данные. В микросервисной архитектуре ESB не используется как центральное промежуточное звено.

Для интеграции обычно используются простые текстовые протоколы, основанные на HTTP, чтобы уменьшить возможную технологическую разность микросервисов. REST-подобные протоколы являются практически стандартом. Иногда могут использоваться бинарные протоколы типа Java RMI или .NET Remoting.

Синхронные вызовы

При взаимодействии микросервисов (это их отличительная черта) синхронные вызовы лучше минимизировать. Рекомендуется использовать не более одного синхронного вызова на один запрос пользователя или вообще их не делать, предпочитая асинхронную коммуникацию.

Проблемы распределенных систем

Одно из наиболее критичных мест в микросервисной архитектуре — необходимость разрабатывать код для распределенной системы, составные элементы которой взаимодействуют через сеть. Из-за того, что сеть сама по себе ненадежна, всё это может легко сломаться: медленнее отвечать или вообще не отвечать. Поэтому нужно уметь обрабатывать такие ситуации. Дополнительный уровень сложности привносит событийная архитектура.

Решения проблем надежности

Чтобы избежать проблем, можно сделать следующее:

1. **Не доводить систему до состояния идеала**, так как это очень дорого. Конечно, это не значит, что система валится от всего подряд — она просто отвечает необходимым

нефункциональным требованиям, но в ней могут присутствовать ошибки, незначительно влияющие на её устойчивость и производительность.

2. **Вкладываться в инфраструктуру**, которая помогает быстрее устранять нештатные ситуации:
    - Должно быть полное покрытие кода unit-тестами, интеграционными тестами и тестами производительности
    - Должен быть интеллектуальный мониторинг, который не только моментально показывает неработающие места, но и сигнализирует об ухудшении состояния системы с прогнозированием возможных сбоев
    - Должно быть продвинутое распределенное логирование, позволяющее оперативно проводить расследования. Часто по результатам расследований исправляются скрытые ошибки

База данных для каждого сервиса

Обязательно, чтобы у каждого сервиса была своя база данных. Это обеспечивает независимость сервисов и позволяет им развиваться независимо друг от друга.

Проблема: много баз данных — как всё согласовать?

Решение: отказ от постоянной согласованности данных (eventual consistency). Микросервисный подход несет довольно много проблем, но разбиение на независимые компоненты даёт безусловные и неоспоримые преимущества: легкое понимание контекста, гибкость развития, управления и масштабирования. Независимость и небольшой размер дают и неожиданные плюсы с точки зрения инфраструктуры.

Каждому микросервису по своему серверу! (или контейнеру)

Монолит vs Микросервисы

Монолит — это единая программная система, в которой все модули и компоненты приложения тесно связаны и работают как единое целое.

В современной разработке проблема сравнения монолитов с микросервисами уже решена. Теперь основное внимание уделяется следующим вопросам:

- Как разложить код на осмысленные и самостоятельные модули/компоненты
- Как правильно определить границы между поддоменами
- Как организовать взаимодействие между компонентами

Модули и их роль

Модули нужны, чтобы разбивать код на границы между поддоменами. Разбивать нужно только там, где есть четкое понимание предметной области и бизнес-логики.

Когда нужно разбивать на модули:

- Когда без этого в коде сложно разбираться
- Когда нужно всё тестировать, а тестирование всего целиком будет долгим
- Когда каждый из команды выполняет свою часть, чтобы всё было понятно и организовано

Важно: модули ≠ микросервисы. Они не обязаны общаться между собой по HTTP/ Kafka, хоть это и можно. Они не обязаны деплоиться отдельно, хоть и можно.

Источники

- Просто о микросервисах
- Современная микросервисная архитектура: принципы проектирования
- Архитектура и проектирование
- Архитектура микросервисов: эволюция, реализация преимуществ и решение проблем в эпоху современного программного обеспечения