МИКРОСЕРВИСНАЯ АРХИТЕКТУРА: СТАНДАРТЫ И ЛУЧШИЕ ПРАКТИКИ РАЗРАБОТКИ

Сладиков А. С., Скиба И. Г.
Кафедра электронных вычислительных машин,
Белорусский государственный университет информатики и радиоэлектороники
Минск, Республика Беларусь
E-mail: pirigovanton906@gmail.com, i.skiba@bsuir.by

В данной работе рассматривается микросервисная архитектура (MSA) как подход к разработке распределённых систем, где приложения делятся на независимые сервисы, отвечающие за конкретные функции. Освещаются ключевые принципы, такие как автономность, легкость коммуникации и возможность горизонтального масштабирования. Также подчеркиваются важные практики разработки, включая четкое разделение обязанностей, использование API-шлюзов и контейнеризацию. Работа акцентирует внимание на важности следования стандартам и лучшим практикам для оптимизации разработки микросервисов.

Введение

Микросервисная архитектура (Microservices Architecture, MSA) представляет собой подход к созданию распределённых систем, где приложение разделяется на небольшие, автономные сервисы, каждый из которых отвечает за определённую задачу или функцию. Каждый микросервис может разрабатываться, развёртываться и масштабироваться отдельно от других, взаимодействуя с ними через стандартизированные и лёгкие интерфейсы, такие как API. В данной работе будут рассмотрены ключевые принципы, стандарты и лучшие практики разработки микросервисной архитектуры.

I. Принципы микросервисной архитектуры

Рассмотрим основные принципы микросервисной архитектуры [1]:

1. Независимые сервисы: Система делится на небольшие сервисы, каждый из которых выполняет конкретную задачу. Это позволяет легко развивать и тестировать их отдельно, а также повторно использовать в других проектах.
2. Автономность: Каждый микросервис может работать независимо, используя свои базы данных и технологии, что обеспечивает гибкость и адаптацию под конкретные задачи.
3. Легкая коммуникация: Взаимодействие между сервисами происходит через легковесные, стандартизированные интерфейсы, таких как RESTful API или сообщения. Это упрощает интеграцию и общение между сервисами.
4. Горизонтальное масштабирование: Микросервисы позволяют масштабировать систему, добавляя или удаляя экземпляры сервисов для балансировки нагрузки и повышения отказоустойчивости.
5. Независимое обновление: Обновление и развертывание сервисов можно выполнять отдельно, без остановки всей системы, что ускоряет разработку и поддержку.

II. Лучшие практики разработки микросервисов

При разработке микросервисов важно следовать ряду проверенных временем практик, которые помогут избежать многих проблем [2–3]:

1. Четкое разделение обязанностей
   Каждый микросервис должен решать одну бизнес-задачу и иметь четкую ответственность, что упрощает его тестирование, сопровождение и модернизацию. Например, если у вас есть приложение для интернет-магазина, вы можете разделить его на микросервисы для обработки заказов, управления инвентарем, управления пользователями и платежной системы. Разделение обязанностей снижает связанность компонентов и уменьшает риск сбоев при изменениях.
2. Использование «API-шлюз» (API Gateway)
   Один из самых простых способов взаимодействия с микросервисами – это прямое обращение клиента к сервису, что подходит для небольших проектов. В крупных приложениях с множеством микросервисов можно использовать API Gateway — шлюз, который размещается между клиентским приложением и микросервисами, предоставляя единую точку входа для клиента.
   В зависимости от задач API-шлюза, выделяют несколько его видов:
   - Gateway Routing: Перенаправляет запросы клиента к нужному сервису;
   - Gateway Aggregation: Распределяет запросы на несколько микросервисов и объединяет ответы;
   - Gateway Offloading: Выполняет общие задачи всех сервисов, как аутентификация, логирование и тд.

   Использование API Gateway позволяет сократить количество запросов на стороне клиента, абстрагировать клиента от про-

токолов, используемых в сервисах (REST, gRPC и др.), и централизованно управлять сквозными функциями. Однако, шлюз может стать единой точкой отказа и при отсутствии масштабирования — узким местом системы, поэтому его необходимо тщательно мониторить.

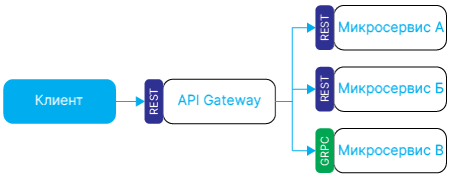

Рис. 1 – пример API Gateway

3. Контейнеризация
   Использование контейнеров (например, Docker) является ключевой практикой для обеспечения стандартизированного развертывания микросервисов. Контейнеры позволяют изолировать каждый микросервис с его собственными зависимостями и окружением, обеспечивая стабильное и предсказуемое поведение на различных платформах.
4. Оркестрация и управление сервисами
   Системы оркестрации контейнеров, такие как Kubernetes, позволяют эффективно управлять развертыванием микросервисов, их масштабированием и отказоустойчивостью. Оркестраторы занимаются автоматическим развертыванием, балансировкой нагрузки, мониторингом и восстановлением сервисов.
5. CI/CD процессы (Непрерывная интеграция и доставка)
   Микросервисные системы требуют хорошо налаженных процессов непрерывной интеграции и доставки (CI/CD). Это необходимо для автоматического развертывания, тестирования и релиза новых версий микросервисов. CI/CD процессы минимизируют риски ошибок и позволяют быстрее и безопаснее внедрять новые функции.
6. Изолированное хранение данных
   Каждый микросервис должен управлять своей собственной базой данных. Это важный аспект микросервисной архитектуры, так как он предотвращает тесную связанность между сервисами и обеспечивает децентрализованное управление данными. Изолированность данных позволяет микросервисам эволюционировать независимо.
7. Масштабирование и автоматизация
   Микросервисная архитектура изначально проектируется с учетом горизонтального масштабирования. Это позволяет эффективно масштабировать отдельные сервисы в зависимости от нагрузки, что делает систему более устойчивой к пиковым нагрузкам. Оркестраторы, такие как Kubernetes, предоставляют встроенные механизмы автоматического масштабирования.
8. Наблюдаемость и трассировка
   В микросервисной архитектуре для обработки клиентских запросов часто участвуют несколько сервисов, выполняющих различные операции. С ростом числа таких сервисов становится все сложнее определить источник ошибок. Для решения этой проблемы, каждому запросу присваивается уникальный идентификатор (TraceId), который передается между сервисами и фиксируется в логах. Это помогает отслеживать запрос на всех этапах обработки.

Примером успешного применения микросервисной архитектуры являются компании, такие как Netflix, Amazon и Spotify. Они используют эту архитектуру для масштабирования систем, обеспечения отказоустойчивости и гибкости. Например, Netflix использует свыше 1000 микросервисов для каждой конкретной задачи, использует систему оркестрации микросервисов для обеспечения масштабируемости своей потоковой платформы, обрабатывающей миллионы запросов в день [4].

Заключение

Микросервисная архитектура предоставляет мощные инструменты для создания гибких, масштабируемых и отказоустойчивых систем. Применение стандартов и лучших практик, таких как разделение обязанностей, независимость развертывания и контейнеризация, помогает упростить разработку, тестирование и эксплуатацию микросервисных приложений. Внедрение микросервисов требует грамотного управления, мониторинга и безопасности, но при этом приносит значительные выгоды, особенно для крупных, динамически развивающихся проектов.

Список литературы

1. 14 Must Know Microservices Design Principles [Электронный ресурс]. – Режим доступа: `https://www.lambdatest.com/blog/microservices-design-principles/`. – Дата доступа: 27.02.2024.
2. 26 основных паттернов микросервисной разработки [Электронный ресурс]. – Режим доступа: `https://cloud.vk.com/blog/26-osnovnyh-patternov-mikroservisnoj-razrabotki`. – Дата доступа: 13.05.2021.
3. Микросервисная архитектура [Электронный ресурс]. – Режим доступа: `https://systems.education/microservices-architecture`. – Дата доступа: 23.07.2020.
4. Netflix Architecture: How Much Does Netflix's AWS Cost? [Электронный ресурс]. – Режим доступа: `https://www.cloudzero.com/blog/netflix-aws/`. – Дата доступа: 19.07.2024.