Хамзин Самат Наилевич, магистрант, Уфимский университет науки и технологий, г. Уфа

Научный руководитель - **Шахмаметова Гюзель Радиковна,** доктор технических наук, заведующий кафедрой Вычислительной математики и кибернетики, Уфимский университет науки и технологий, г. Уфа

АНАЛИЗ СРАВНЕНИЯ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ С МОНОЛИТНОЙ

Аннотация. В статье приведены отличия микросервисного и монолитного подхода к построению серверных веб приложений. В ходе анализа рассматриваются примеры каждой архитектур, их принципы работы, анализируются достоинства и недостатки каждого подхода, который поможет сделать оптимальный выбор, а также сравниваются архитектуры по ключевым критериям. Анализ сравнения позволит разработчикам, архитекторам, аналитикам понять ключевые отличия между микросервисной и монолитной архитектурами, понять их принципы работы, а также определить наилучший для себя вариант для построения серверного веб приложения.

Annotation. The article summarizes the differences between microservice and monolithic approaches to building server-side web applications. The analysis considers examples of each architecture, their operating principles, analyzes the advantages and disadvantages of each approach to help make the best choice, and compares the architectures by key criteria. The comparison analysis will allow developers, architects, analysts to understand the key differences between microservice and monolithic architectures, understand their working principles, and determine the best option for building a server-side web application.

Ключевые слова: монолиты, микросервисы, серверные веб-приложения, ACID-транзакции, контейнеризация, распределенные системы.

Keywords: monoliths, microservices, server-side web applications, ACID transactions, containerization, distributed systems.

На сегодняшний день существует два основных подхода к построению архитектуры веб-серверных приложений: монолиты и микросервисы. При микросервисной архитектуре (далее по тексту МСА) целое приложение состоит из отдельных сервисов, каждый из которых выполняет свою отдельную бизнес-функцию. Чтобы получше узнать причины появления и распространения МСА, необходимо для начала понять отличие МСА от монолитной архитектуры.

Монолитная архитектура сформировалась благодаря появлению технологий JavaScript, PHP, Java Servlet. Эти инструменты позволили обрабатывать пользовательские запросы на стороне сервера и соответственно создавать клиент-серверные веб приложения. Приложения представляли собой, так называемые, монолиты – единые приложения, которые включали в себе всю бизнес-логику, генерацию HTML страниц и слой доступа к данным [1]. Пример монолитной архитектуры интернет-магазина видеоигр приведен на рисунке 1.

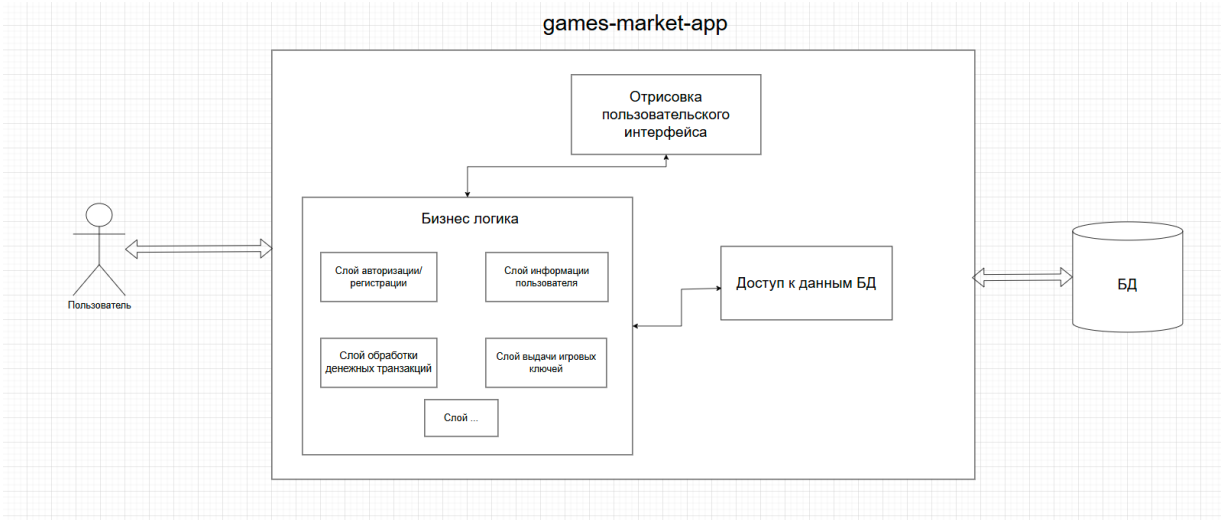

Рисунок 1 – Пример архитектуры монолитного веб-приложения

К плюсам монолитного подхода можно отнести:

- простая разработка и сопровождение на начальных этапах;
- хорошая производительность, все также на начальных этапах;

- согласованность данных в базе данных (далее по тексту – БД) приложения благодаря ACID-транзакциям [2]. На примере магазина видеоигр, если при покупке видеоигры у клиента не хватало денег, вся ACID транзакция в БД совершала откат операции, тем самым сохранив данные клиента в прежнем состоянии (баланс клиента не менялся, товар не зачислился на аккаунт и т.д.).

Но, по мере роста приложения, добавления новой бизнес-логики, роста команды разработчиков, увеличение числа активных пользователей, рост нагрузки, приложение становилось все труднее поддерживать и масштабировать:

- при увеличении данных пользователей в БД, время чтения и сохранения данных увеличивается и требуется все больше дискового пространства.

- приложение начнет выделять все больше потоков для обработки запросов пользователей, что будет требовать увеличения доступной оперативной памяти и количества ядер процессора сервера, приходится прибегать к вертикальному масштабированию;

- если же нагрузка будет увеличиваться на какой-то конкретный слой приложения, придется увеличивать доступные ресурсы для всего приложения, что может привести к нерациональному выделению ресурсов;

- поддержка и дальнейшая разработка приложения тоже начнет усложняться, разработчикам станет трудно локально запускать сервис для проверки своих доработок из-за того, что он требует много ресурсов компьютера, а внедрение небольшого исправление в продовую среду может сильно затянуться из-за долгого процесса тестирования (проведения всех автотестов, проверка на тестовых средах), согласования с другими разработчиками, сборки и развёртывания приложения;

- самым неприятным минусом является то, что, если какой-либо слой приложения упадет в критическую ошибку, он потянет за собой все приложение.

Из-за проблем монолитной архитектуры, описанных выше, начали формироваться идеи микросервисных архитектур. Крупные IT-компании Google, Amazon, Netflix первыми начали вводить в свои системы MCA из-за необходимости масштабирования и увеличении скорости выпуска новых версий приложений [3]. Немаловажным фактором в распространении MCA стало развитие облачных технологий, контейнеризации приложений (прим. Docker) и систем оркестрации контейнеров (прим. Kubernetes) [4]. Они упростили процесс развертывания и управления множеством сервисов. Пример MCA интернет-магазина приведен на рисунке 2.

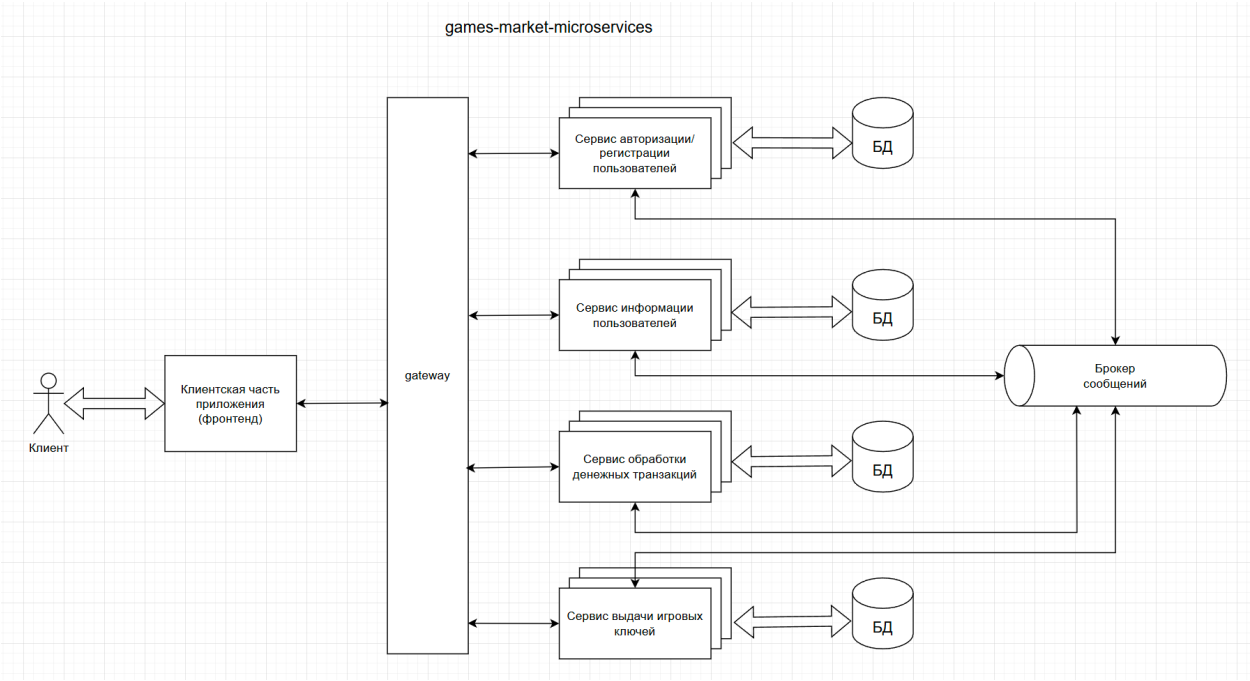

Рисунок 2 – Пример микросервисной архитектуры

Из плюсов использования данного подхода можно выделить:

- масштабирование отдельных сервисов всей системы. Это позволит эффективно использовать ресурсы сервера, на примере системы Рисунка 2, если сервис для информации пользователя испытывает наибольшую нагрузку, в сравнении с другими сервисами, то можно увеличить количество экземпляров или увеличить используемые ресурсы сервера, благодаря технологиям виртуализации и оркестрации;

- систему могут разрабатывать множество команд независимо друг от друга, а также независимо от технологий. Один сервис может использовать один стек технологий для реализации работы, другая команда использовать другой стек;

- сбой одного из сервисов не приведет к отказу всей системы, это также позволит сосредоточить все силы на поиск проблемы только в одном сервисе и быстро решить проблему;

- более простая и быстрая сборка и развертывание сервисов. Небольшая правка может быстро достичь продовой среды благодаря тому, что тестирование сервиса проходит намного быстрее, из-за изолированности бизнес-логики в одном сервисе, это относится к процессу сборки и развертывания.

Но данный подход также содержит определенные недостатки:

- системы находятся изолированно друг от друга, и соответственно нужно настраивать взаимодействие между ними. Определение типа взаимодействия и контракта может отнимать много времени и сил;

- управление сервисами требует множества инструментов. Для сервисов необходимо как минимум настраивать системы сбора логов, мониторинга и систем оркестрации. Также для настройки всех инструментов и инфраструктуры требуется отдельный DevOps специалист;

- при ошибках в сервисах, возникает сложность в анализе огромного количества логов и межсервисных взаимодействий;

- тестирование отдельных сервисов как правило не представляет проблем, однако тестирование взаимодействия всех сервисов, или же интеграционное тестирование, может привнести определенные временные затраты;

- из-за взаимодействия сервисов по сети, возникают определенные риски сбоев;

- проблемы с согласованностью данных, при использовании МСА отсутствует транзакционность между сервисами, так как каждый сервис будет

использовать свою, изолированную БД и провести атомарную ACID транзакцию, затрагивающую все бизнес-шаги (прим. покупки видеоигры в системе на Рисунке 2) невозможно, для обеспечения необходимо использовать паттерны распределенных транзакций [5].

Принимая во внимание вышеизложенное, можно составить таблицу различий монолитной и микросервисной архитектур по ключевым критериям. Сравнение представлена в таблице 1.

Таблица 1 – Сравнение монолитной и микросервисной архитектуры по ключевым критериям

Критерий	Монолит	Микросервисы
Масштабирование	Масштабирование всего приложения	Масштабирование отдельных сервисов
Структура	Единое приложение, включающее все бизнес-слои	Бизнес-слои распределены по отдельным сервисам
Развертывание	Общее развертывание всего приложения	Независимое развертывание каждого сервиса
Тестирование	Все приложение	Возможность тестирования отдельных сервисов
Внесение изменений	Одна общая версия приложения	Независимые версии сервисов
Структура	Одно единое приложение	Множество изолированных сервисов

Таким образом, МСА стала логическим развитием современной разработки к масштабируемости, гибкости и скорости поставки программного обеспечения. Несмотря на определенные сложности, связанных с построением и проработкой такой архитектуры и их сопровождением, они представляют значительные преимущества для построения сложных и высоконагруженных систем.

Литература

1. Сравнение микросервисной и монолитной архитектур // URL: https://www.atlassian.com/ru/microservices/microservices-architecture/microservices-vs-monolith
2. Транзакция, ACID, CAP теорема и уровни изоляций транзакций простыми словами // URL: https://habr.com/ru/companies/alfa/articles/812417/
3. Монолитная и микросервисная архитектура. Сравнение // URL: https://habr.com/ru/companies/haulmont/articles/758780/
4. Введение в Docker и Kubernetes: основы контейнерных технологий // URL: https://habr.com/ru/companies/sibur_official/articles/826964/
5. Распределенные транзакции // URL: https://habr.com/ru/articles/769102/

Literature

1. Comparison of microservice and monolithic architectures // URL: https://www.atlassian.com/ru/microservices/microservices-architecture/microservices-vs-monolith
2. Transaction, ACID, CAP theorem and transaction isolation levels in simple words // URL: https://habr.com/ru/companies/alfa/articles/812417/
3. Monolithic and microservice architecture. Comparison // URL: https://habr.com/ru/companies/haulmont/articles/758780/
4. Introduction to Docker and Kubernetes: basics of container technologies // URL: https://habr.com/ru/companies/sibur_official/articles/826964/
5. Distributed transactions // URL: https://habr.com/ru/articles/769102/