МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра информационных систем управления

Вареник Андрей Александрович

ОПТИМИЗАЦИЯ МЕЖСЕРВИСНОГО ВЗАИМОДЕЙСТВИЯ В БАНКОВСКИХ СИСТЕМАХ

Дипломная работа

Научный руководитель:
Старший преподаватель,
Д. Ю. Кваша

Допущена к защите
«____»_____20___г.
Заведующий информационных систем управления
доктор технических наук,
доцент А.М. Недзьведь

Минск, 2025

ОГЛАВЛЕНИЕ

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ

HTTP (HyperText Transfer Protocol) – Протокол передачи гипертекста.

gRPC (Google Remote Procedure Call) – Межпроцессный коммуникационный фреймворк с поддержкой потоковой передачи данных.

Java – Объектно-ориентированный язык программирования.

Process Engine – Процессный движок, ядро Camunda BPM, отвечающее за исполнение бизнес-процессов.

REST (Representational State Transfer) – Архитектурный стиль взаимодействия компонентов распределённого приложения в сети.

Spring Boot – Фреймворк для создания приложений на языке Java.

Thymeleaf – Шаблонизатор для Java веб-приложений.

Валидация данных – Проверка корректности введенных данных.

Контроллер – Компонент, обрабатывающий HTTP-запросы в веб-приложении.

ПО (Программное обеспечение) – совокупность программ, используемых для автоматизации процессов и управления системой.

API (Application Programming Interface) – программный интерфейс приложения, набор методов и протоколов для взаимодействия программных компонентов.

Топик (Topic) – Логический канал в Kafka для публикации и подписки на сообщения.

Партиция (Partition) – Физическое разделение топика Kafka для параллельной обработки.

Репликация (Replication) – Механизм дублирования данных между брокерами Kafka для отказоустойчивости.

Продюсер (Producer) – Компонент, публикующий сообщения в Kafka.

Консьюмер (Consumer) – Компонент, обрабатывающий сообщения из Kafka.

Баланс (Balance) – Текущее состояние средств на банковском счёте.

Транзакция (Transaction) – Финансовая операция изменения состояния счёта.

Микросервис (Microservice) – Независимый компонент системы, реализующий одну бизнес-функцию.

РЕФЕРАТ

Структура и объём дипломной работы 69 страниц, 11 рисунков, 2 приложения, 12 источников.

Ключевые слова: МИКРОСЕРВИСНАЯ АРХИТЕКТУРА, МЕЖСЕРВИСНОЕ ВЗАИМОДЕЙСТВИЕ, APACHE KAFKA, REST API, БАНКОВСКИЕ СИСТЕМЫ, СОБЫТИЙНО-ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА, ПРОИЗВОДИТЕЛЬНОСТЬ, МАСШТАБИРУЕМОСТЬ, ЛАТЕНТНОСТЬ, ПРОПУСКНАЯ СПОСОБНОСТЬ.

Текст реферата

Объект исследования – процессы межсервисного взаимодействия в распределенных банковских системах с микросервисной архитектурой.

Предмет исследования – методы и технологии оптимизации обмена сообщениями между микросервисами в банковских системах с использованием Apache Kafka и REST API.

Цель исследования – разработка и экспериментальное сравнение подходов к организации межсервисного взаимодействия в банковских системах на основе Apache Kafka и REST API для определения оптимальных сценариев применения каждой технологии.

Методы исследования – системный анализ, формальное моделирование, объектно-ориентированное проектирование, экспериментальное тестирование, сравнительный анализ количественных показателей производительности, статистическая обработка результатов.

Полученные результаты и их новизна – разработана событийно-ориентированная архитектура банковской системы с использованием Apache Kafka для межсервисного взаимодействия; спроектированы и реализованы формальные модели компонентов системы и алгоритмы их взаимодействия; проведено экспериментальное сравнение производительности REST API и Kafka при различных уровнях нагрузки; получены количественные характеристики преимуществ Kafka по показателям стабильности, латентности и пропускной способности при высоких нагрузках; сформулированы рекомендации по выбору оптимальной технологии межсервисного взаимодействия для различных сценариев использования в банковских системах.

Достоверность материалов и результатов дипломной работы – обеспечивается использованием научно обоснованных методов исследования, экспериментальной проверкой теоретических положений на реальных данных, применением статистических методов обработки результатов

тестирования, согласованностью полученных результатов с исследованиями других авторов в данной области.

Область возможного практического применения – результаты работы могут быть использованы при проектировании и разработке высоконагруженных распределенных банковских систем, требующих обеспечения высокой производительности, надежности и масштабируемости; методология сравнения технологий межсервисного взаимодействия может применяться для обоснования архитектурных решений в финансовом секторе; практические рекомендации по настройке Apache Kafka могут быть использованы для оптимизации существующих систем обмена сообщениями.

РЭФЕРАТ

Структура і аб'ём дыпломнай працы 69 старонкак, 11 малюнкаў, 2 дадаткі, 12 крыніц.

Ключавыя словы: МІКРАСЭРВІСНАЯ АРХІТЭКТУРА, МІЖСЭРВІСНАЕ ЎЗАЕМАДЗЕЯННЕ, APACHE KAFKA, REST API, БАНКАЎСКІЯ СІСТЭМЫ, ПАДЗЕЙНА-АРЫЕНТАВАНАЯ АРХІТЭКТУРА, ПРАДУКЦЫЙНАСЦЬ, МАШТАБАВАНАСЦЬ, ЛАТЭНТНАСЦЬ, ПРАПУСКНАЯ ЗДОЛЬНАСЦЬ

Змест працы

Аб'ект даследавання – працэсы міжсэрвіснага ўзаемадзеяння ў размеркаваных банкаўскіх сістэмах з мікрасэрвіснай архітэктурай.

Прадмет даследавання – метады і тэхналогіі аптымізацыі абмену паведамленнямі паміж мікрасэрвісамі ў банкаўскіх сістэмах з выкарыстаннем Apache Kafka і REST API.

Мэта даследавання – распрацоўка і эксперыментальнае параўнанне падыходаў да арганізацыі міжсэрвіснага ўзаемадзеяння ў банкаўскіх сістэмах на аснове Apache Kafka і REST API для вызначэння аптымальных сцэнарыяў прымянення кожнай тэхналогіі.

Метады даследавання – сістэмны аналіз, фармальнае мадэляванне, аб'ектна-арыентаванае праектаванне, эксперыментальнае тэставанне, параўнальны аналіз колькасных паказчыкаў прадукцыйнасці, статыстычная апрацоўка вынікаў.

Атрыманыя вынікі і іх навізна – распрацавана падзейна-арыентаваная архітэктура банкаўскай сістэмы з выкарыстаннем Apache Kafka для міжсэрвіснага ўзаемадзеяння; спраектаваны і рэалізаваны фармальныя мадэлі кампанентаў сістэмы і алгарытмы іх узаемадзеяння; праведзена эксперыментальнае параўнанне прадукцыйнасці REST API і Kafka пры розных узроўнях нагрузкі; атрыманы колькасныя характарыстыкі пераваг Kafka па паказчыках стабільнасці, латэнтнасці і прапускной здольнасці пры высокіх нагрузках; сфармуляваны рэкамендацыі па выбары аптымальнай тэхналогіі міжсэрвіснага ўзаемадзеяння для розных сцэнарыяў выкарыстання ў банкаўскіх сістэмах.

Дакладнасць матэрыялаў і вынікаў дыпломнай працы – забяспечваецца выкарыстаннем навукова абгрунтаваных метадаў даследавання, эксперыментальнай праверкай тэарэтычных палажэнняў на рэальных дадзеных, прымяненнем статыстычных метадаў апрацоўкі вынікаў тэставання, узгодненасцю атрыманых вынікаў з даследаваннямі іншых аўтараў у гэтай галіне.

Вобласць магчымага практычнага прымянення – вынікі працы могуць быць выкарыстаны пры праектаванні і распрацоўцы высоканагружаных размеркаваных банкаўскіх сістэм, якія патрабуюць забеспячэння высокай прадукцыйнасці, надзейнасці і маштабаванасці; метадалогія параўнання тэхналогій міжсэрвіснага ўзаемадзеяння можа прымяняцца для абгрунтавання архітэктурных рашэнняў у фінансавым сектары; практычныя рэкамендацыі па наладцы Apache Kafka могуць быць выкарыстаны для аптымізацыі існуючых сістэм абмену паведамленнямі.

SUMMARY

Structure and scope of the diploma work 69 pages, 11 figures, 2 appendices, 12 references.

Keywords: MICROSERVICE ARCHITECTURE, INTERSERVICE COMMUNICATION, APACHE KAFKA, REST API, BANKING SYSTEMS, EVENT-DRIVEN ARCHITECTURE, PERFORMANCE, SCALABILITY, LATENCY, THROUGHPUT

Summary text

The object of the research – interservice communication processes in distributed banking systems with microservice architecture.

The subject of the research – methods and technologies for optimizing message exchange between microservices in banking systems using Apache Kafka and REST API.

The aim of the research – development and experimental comparison of approaches to organizing interservice communication in banking systems based on Apache Kafka and REST API to determine optimal scenarios for applying each technology.

Research methods – system analysis, formal modeling, object-oriented design, experimental testing, comparative analysis of quantitative performance indicators, statistical processing of results.

The results of the work and their novelty – an event-driven architecture of a banking system using Apache Kafka for interservice communication has been developed; formal models of system components and algorithms for their interaction have been designed and implemented; an experimental comparison of REST API and Kafka performance at various load levels has been conducted; quantitative characteristics of Kafka's advantages in terms of stability, latency, and throughput under high loads have been obtained; recommendations for choosing the optimal interservice communication technology for various use cases in banking systems have been formulated.

Authenticity of the materials and results of the diploma work – is ensured by the use of scientifically based research methods, experimental verification of theoretical provisions on real data, application of statistical methods for processing test results, and consistency of the obtained results with research by other authors in this field.

Recommendations on the usage – the results of the work can be used in the design and development of high-load distributed banking systems requiring high performance, reliability, and scalability; the methodology for comparing interservice communication technologies can be applied to justify architectural decisions in the

financial sector; practical recommendations for configuring Apache Kafka can be used to optimize existing messaging systems.

ВВЕДЕНИЕ

Современные банковские системы функционируют в условиях постоянно растущих объемов транзакций, повышенных требований к скорости обработки операций и необходимости обеспечения высокой доступности сервисов. В этих условиях традиционные монолитные архитектуры уступают место распределенным микросервисным решениям, где ключевым фактором эффективности становится оптимизация межсервисного взаимодействия. Проблема выбора оптимального механизма обмена данными между компонентами банковских систем приобретает особую актуальность в контексте финансовых приложений, требующих не только высокой производительности, но и гарантированной целостности данных при проведении транзакций.

Данная работа посвящена исследованию современных подходов к организации межсервисного взаимодействия в распределенных банковских системах, с особым фокусом на сравнительном анализе событийно-ориентированной архитектуры на базе Apache Kafka и традиционного REST API. Практический опыт, полученный в ходе проектирования и реализации высокопроизводительных решений для обработки финансовых операций, позволил выявить критические факторы, влияющие на выбор оптимальной технологии межсервисного взаимодействия в зависимости от конкретных требований и сценариев использования.

В рамках работы проведен анализ требований к банковским системам, изучены принципы микросервисной архитектуры и паттерны обмена сообщениями. Особое внимание уделено технологии Apache Kafka как инструменту для обеспечения асинхронной коммуникации между компонентами системы. На основе проведенных исследований разработана модель банковской системы, включающая сервисы управления счетами и обработки транзакций, интегрированные через распределенный брокер сообщений.

Практическая часть работы включала настройку Kafka-кластера, реализацию микросервисов на Java 21 и Spring Boot 3.3.3, а также тестирование системы на предмет производительности и отказоустойчивости. Результатом стало работоспособное решение, способное обрабатывать до 1000 транзакций в секунду с гарантией доставки сообщений и сохранением согласованности данных.

В первой главе отчета рассмотрены теоретические аспекты организации межсервисного взаимодействия, обоснован выбор архитектурных решений и технологий. Вторая и третья главы посвященаы практической реализации

системы: описаны модели компонентов, алгоритмы обработки операций, настройки Kafka-кластера и анализ преимуществ асинхронного подхода. Четвёртая глава посвящена экспериментальному сравнению производительности REST и Apache Kafka в контексте банковских операций.

ГЛАВА 1
ОСНОВЫ ОРГАНИЗАЦИИ МЕЖСЕРВИСНОГО ВЗАИМОДЕЙСТВИЯ В БАНКОВСКИХ СИСТЕМАХ

Современные банковские системы представляют собой сложные программные комплексы, обрабатывающие миллионы транзакций ежедневно. В условиях цифровой трансформации финансового сектора возрастает необходимость в гибких, масштабируемых и надежных программных решениях. Межсервисное взаимодействие становится критически важным аспектом построения таких систем, поскольку определяет способность компонентов системы эффективно обмениваться информацией, обеспечивая целостность и согласованность данных.

В данной главе рассматриваются теоретические основы построения современных банковских систем с акцентом на межсервисное взаимодействие. Изучение этих фундаментальных концепций необходимо для понимания практических решений, которые будут представлены во второй главе. Особое внимание уделяется событийно-ориентированной архитектуре и технологии Apache Kafka как инструменту оптимизации взаимодействия между микросервисами в высоконагруженных банковских приложениях.

1.1 Архитектурные подходы к построению современных банковских систем.

Архитектура банковских информационных систем прошла значительную эволюцию от монолитных решений к распределенным многокомпонентным системам. Это развитие обусловлено возрастающими требованиями к масштабируемости, отказоустойчивости и гибкости в условиях постоянно меняющегося рынка финансовых услуг.

Монолитная архитектура

Традиционно банковские системы разрабатывались как монолитные приложения – единые программные комплексы, объединяющие все функции банковского обслуживания. Монолитная архитектура характеризуется следующими особенностями:

- Единая кодовая база для всей функциональности системы
- Централизованное хранение данных в единой базе данных
- Вертикальное масштабирование (увеличение мощности аппаратного обеспечения)
- Сложность внесения изменений и длительные циклы разработки
- Высокая связанность компонентов системы

Монолитные системы, несмотря на простоту разработки и развертывания, сталкиваются с серьезными проблемами при росте нагрузки и необходимости быстрой адаптации к изменениям рынка. Эти ограничения привели к поиску более гибких архитектурных решений.

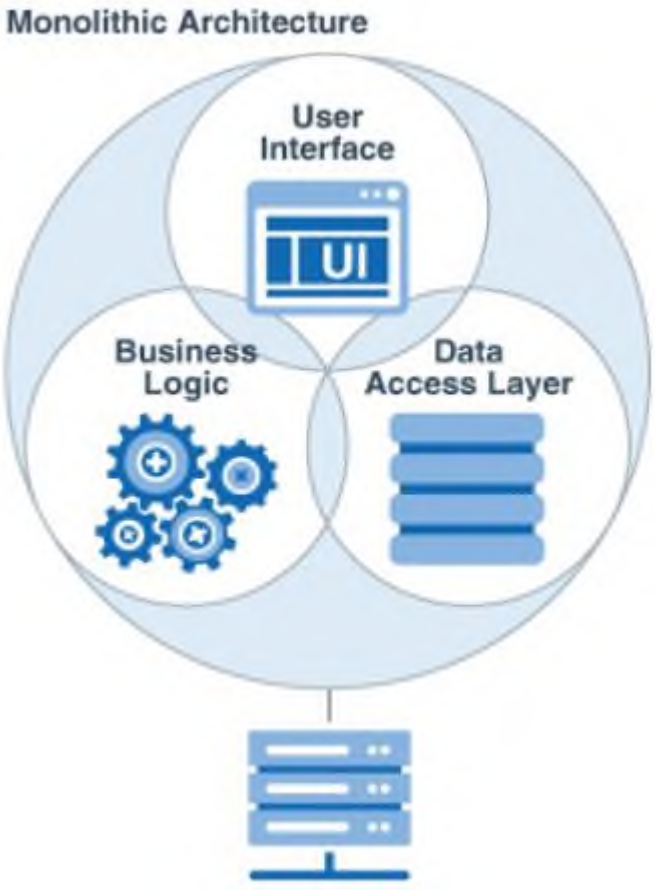


Рисунок 1.1 – Монолитная архитектура

Сервис-ориентированная архитектура (SOA)

Следующим этапом эволюции стала сервис-ориентированная архитектура (SOA), которая предполагает разделение системы на отдельные бизнес-сервисы, взаимодействующие через стандартизированные протоколы:

- Декомпозиция системы на бизнес-сервисы с четко определенными границами
- Использование общей шины данных (Enterprise Service Bus, ESB) для обмена сообщениями
- Стандартизированные протоколы взаимодействия (SOAP, XML)
- Повторное использование сервисов в различных бизнес-процессах
- Централизованное управление и оркестрация сервисов

SOA позволила повысить модульность и гибкость банковских систем, но часто страдала от сложности инфраструктуры и зависимости от центральных компонентов, таких как ESB.

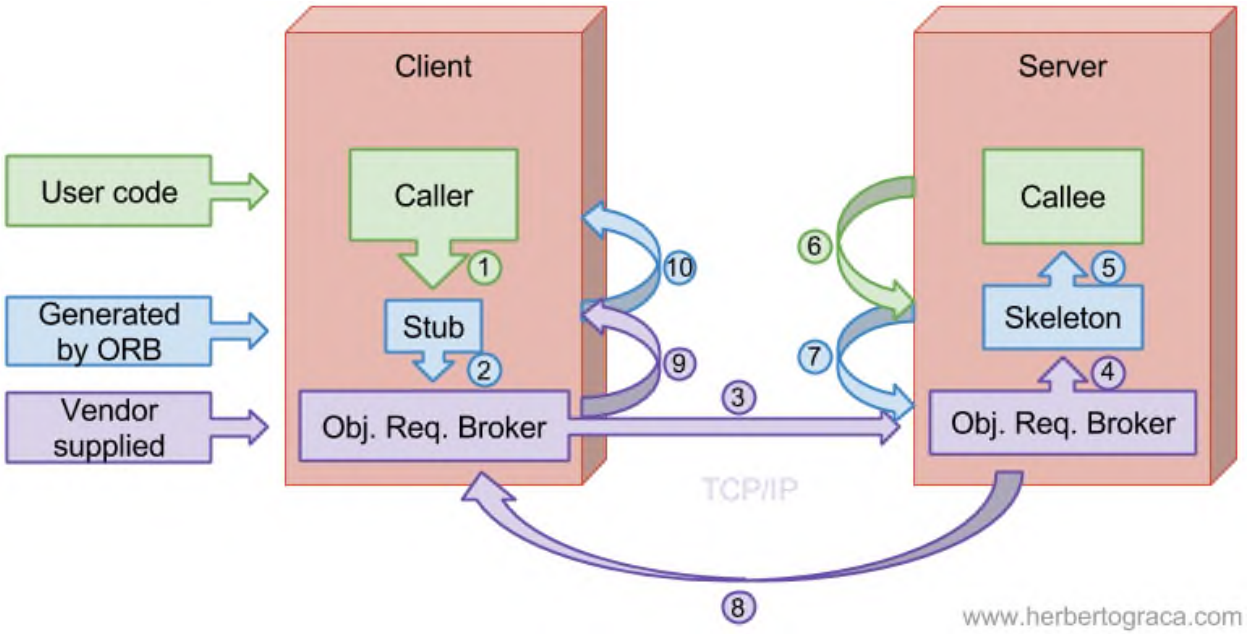


Рисунок 1.2 – Сервис-ориентированная архитектура

Микросервисная архитектура

Современные банковские системы все чаще строятся на принципах микросервисной архитектуры, которая представляет собой развитие идей SOA с акцентом на большую автономность и независимость сервисов:

– Декомпозиция на небольшие, узкоспециализированные сервисы с собственным жизненным циклом
– Независимое развертывание и масштабирование каждого сервиса
– Распределенное хранение данных (каждый сервис имеет свою базу данных)
– Легковесные протоколы взаимодействия (REST, gRPC)
– Автоматизированное тестирование и непрерывная интеграция (CI/CD)
– Использование контейнеризации (Docker) и оркестрации контейнеров (Kubernetes)

Микросервисный подход позволяет банкам быстрее выводить новые продукты на рынок, обеспечивать высокую доступность сервисов и эффективно масштабировать отдельные компоненты системы в зависимости от нагрузки.

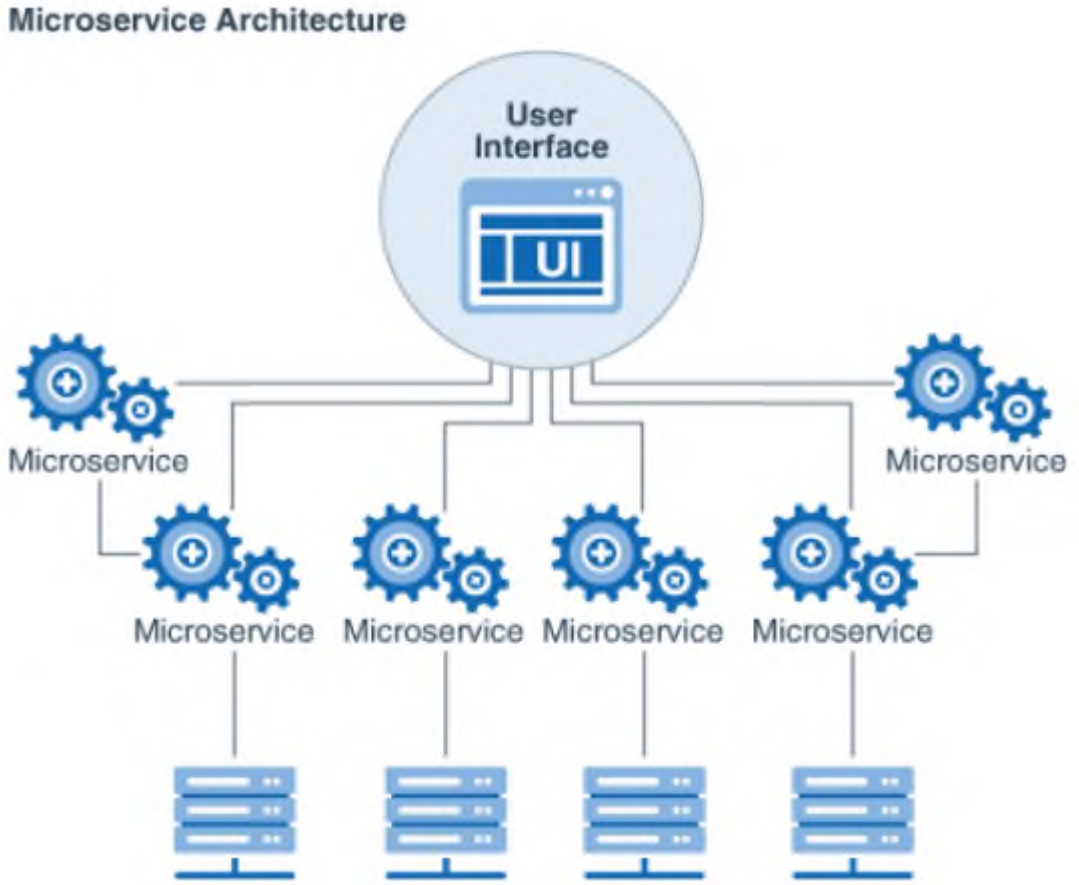


Рисунок 1.3 – Микросервисная архитектура

Событийно-ориентированная архитектура (Event-Driven Architecture)

В последние годы в банковских системах получила распространение событийно-ориентированная архитектура, особенно в сочетании с микросервисным подходом:

- Асинхронное взаимодействие сервисов через события
- Использование брокеров сообщений (Apache Kafka, RabbitMQ)
- Слабая связанность между сервисами-производителями и сервисами-потребителями событий
- Возможность реконструкции состояния системы на основе последовательности событий
- Повышенная отказоустойчивость благодаря буферизации сообщений

Такой подход особенно ценен для банковских систем, где требуется обрабатывать большие объемы транзакций с минимальными задержками, при этом обеспечивая согласованность данных и возможность аудита.

Гибридные подходы

На практике банки часто используют гибридные архитектурные подходы, комбинируя элементы разных архитектурных стилей в зависимости от конкретных потребностей:

– Постепенная миграция от монолита к микросервисам по мере необходимости
– Сочетание синхронных (REST) и асинхронных (события) взаимодействий
– Многоуровневая архитектура с разделением на компоненты представления, бизнес-логики и доступа к данным
– Специализированные архитектурные решения для критически важных компонентов (например, системы реального времени для платежных шлюзов)

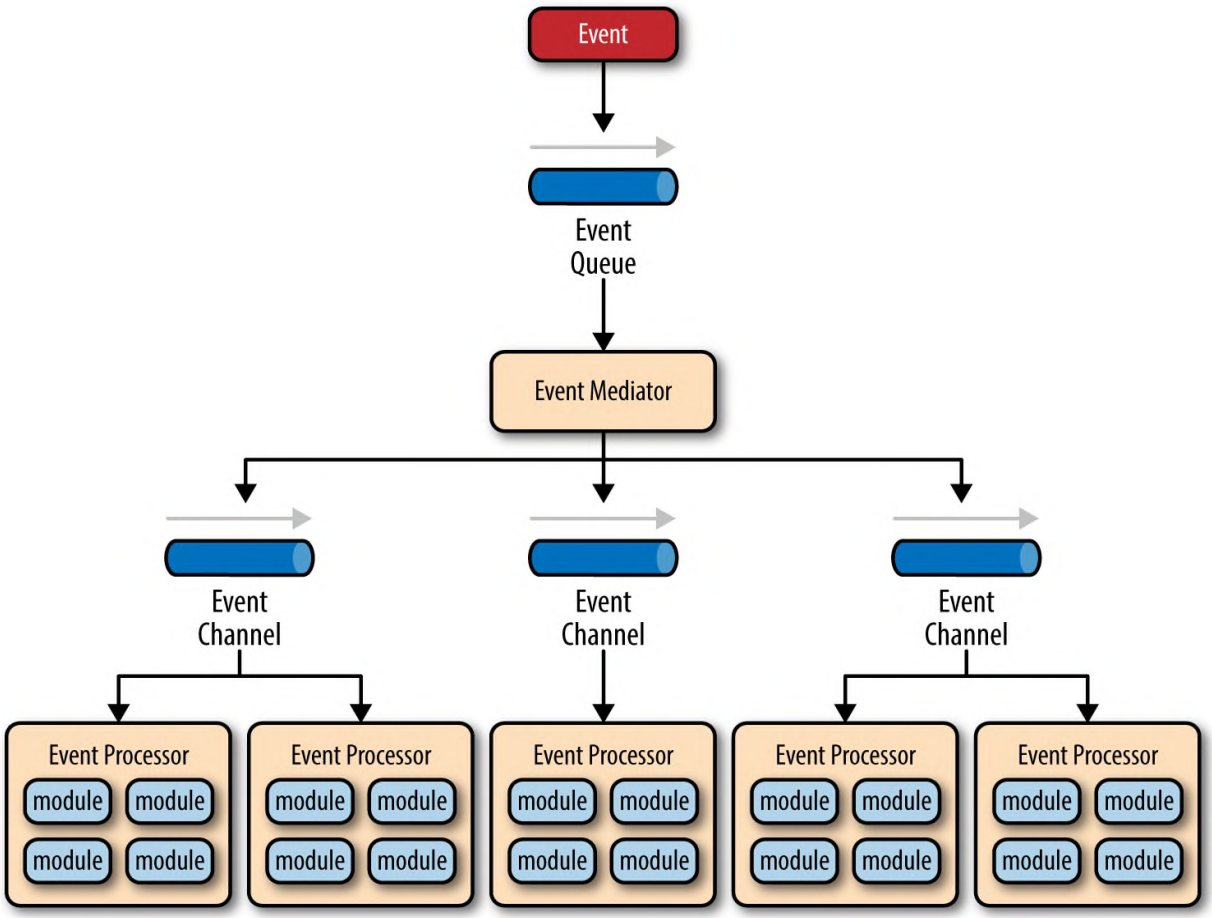


Рисунок 1.4 – Событийно-ориентированная архитектура

Выбор конкретной архитектуры банковской системы зависит от множества факторов, включая масштаб операций, требования к производительности, существующую ИТ-инфраструктуру и стратегические бизнес-цели.

1.2 Микросервисная архитектура и проблемы межсервисного взаимодействия.

Микросервисная архитектура представляет собой подход к построению программных систем, при котором приложение разбивается на набор небольших, слабо связанных и независимо развертываемых сервисов. Каждый сервис отвечает за конкретную бизнес-функцию и может разрабатываться, тестироваться и масштабироваться независимо от других компонентов системы. В банковской сфере микросервисная архитектура получила широкое распространение благодаря своим преимуществам в условиях динамично меняющихся бизнес-требований и высоких нагрузок.

Принципы микросервисной архитектуры

Микросервисная архитектура базируется на следующих ключевых принципах:

- Единая ответственность - каждый сервис отвечает за решение одной бизнес-задачи (например, управление счетами, обработка платежей, аутентификация пользователей);
- Изоляция данных - каждый сервис имеет собственное хранилище данных, соответствующее его потребностям;
- Автономность - команды разработчиков могут работать над различными сервисами независимо друг от друга;
- Независимость развертывания - сервисы могут быть обновлены и развернуты без влияния на работу других компонентов системы;
- Децентрализованное управление - отсутствие единого центра принятия технологических решений;
- Устойчивость к сбоям - локализация проблем в рамках отдельных сервисов без каскадного распространения на всю систему.

Преимущества микросервисов для банковских систем

В контексте банковских приложений микросервисная архитектура предоставляет ряд существенных преимуществ:

1. Гибкость и скорость внедрения - возможность быстро разрабатывать и внедрять новые банковские продукты и услуги;
2. Масштабируемость - способность масштабировать только те компоненты системы, которые испытывают повышенную нагрузку (например, платежный шлюз в периоды пиковой активности);
3. Технологическое разнообразие - возможность использовать наиболее подходящие технологии для каждой конкретной задачи;
4. Устойчивость к сбоям - отказ отдельного сервиса не приводит к полной

недоступности всей банковской системы;

5. Улучшенная безопасность - изоляция критически важных компонентов и возможность применения специализированных мер безопасности к различным сервисам;
6. Упрощение соответствия регуляторным требованиям - возможность обновлять только те компоненты, которые затрагиваются изменениями в законодательстве.

Проблемы межсервисного взаимодействия

Несмотря на множество преимуществ, микросервисная архитектура создает ряд сложностей, особенно в организации эффективного взаимодействия между сервисами:

1. Выбор подходящих протоколов коммуникации;
2. Синхронное или асинхронное взаимодействие;
3. Обеспечение согласованности данных;
4. Управление распределенными транзакциями;
5. Устойчивость к отказам.

Особенности межсервисного взаимодействия в банковских системах

Банковские системы предъявляют особые требования к межсервисному взаимодействию из-за специфики финансовой отрасли:

1. Повышенные требования к надежности - недопустимость потери финансовых транзакций требует гарантированной доставки сообщений между сервисами;
2. Аудит и соответствие регуляторным требованиям - необходимость логирования всех взаимодействий для последующего аудита;
3. Задержки минимальны - пользователи ожидают мгновенного проведения финансовых операций, что создает высокие требования к производительности межсервисного взаимодействия;
4. Безопасность превыше всего - обмен финансовой информацией требует многоуровневой защиты;
5. Согласованность данных - критическая важность предотвращения расхождений в финансовых данных между различными системами.

1.3 Паттерны обмена сообщениями в распределенных системах

Распределенные системы, включая банковские приложения на основе микросервисной архитектуры, требуют тщательно продуманных механизмов обмена сообщениями для обеспечения эффективного взаимодействия между компонентами. Паттерны обмена сообщениями представляют собой

проверенные временем решения типовых проблем коммуникации в распределенных системах. Правильный выбор и реализация этих паттернов напрямую влияют на надежность, производительность и масштабируемость банковских приложений.

Request-Response (Запрос-Ответ)

Этот фундаментальный паттерн лежит в основе большинства API-взаимодействий и предполагает, что клиент отправляет запрос и ожидает получения ответа от сервера.

Характеристики:

– Простота реализации и понимания;
– Блокирующий характер взаимодействия;
– Тесная временная связь между запросом и ответом.

Применение в банковских системах:

– Проверка баланса счета;
– Авторизация платежной операции;
– Запрос информации о клиенте.

Технологии реализации:

– REST API поверх HTTP/HTTPS;
– gRPC;
– SOAP Web Services.

Remote Procedure Call (RPC)

RPC позволяет вызывать функции или методы удаленного сервиса так, как если бы они были локальными.

Характеристики:

– Упрощение программной модели распределенных систем;
– Скрытие деталей сетевого взаимодействия;
– Потенциальные проблемы с отказоустойчивостью.

Применение в банковских системах:

– Внутренние взаимодействия между тесно связанными сервисами;
– Высокопроизводительные операции, требующие низкой задержки.

Технологии реализации:

– gRPC;
– Apache Thrift;
– Spring Remoting.

Publish-Subscribe (Издатель-Подписчик)

В этом паттерне сервисы, выступающие в роли издателей, публикуют сообщения в определенные темы или каналы, а сервисы-подписчики получают сообщения из тем, на которые они подписаны.

Характеристики:

– Слабая связанность между отправителями и получателями;
– Масштабируемость и гибкость системы;
– Поддержка сценариев «один-ко-многим».

Применение в банковских системах:

– Уведомления о транзакциях;
– Обновление рыночных данных;
– Распространение информации о изменениях в банковских продуктах.

Технологии реализации:

– Apache Kafka;
– RabbitMQ (в режиме pub/sub);
– Amazon SNS.

Message Queue (Очередь сообщений)

Этот паттерн предполагает отправку сообщений в очередь, откуда они извлекаются и обрабатываются одним из доступных обработчиков.

Характеристики:

– Гарантированная доставка сообщений;
– Буферизация нагрузки и асинхронная обработка;
– Сценарии "один-к-одному" (каждое сообщение обрабатывается только одним получателем).

Применение в банковских системах:

– Обработка платежных поручений;
– Обработка заявок на кредиты;
– Выполнение длительных операций (расчет процентов, формирование отчетов).

Технологии реализации:

– RabbitMQ;
– Apache ActiveMQ;
– Amazon SQS.

Выбор оптимальных паттернов обмена сообщениями для банковской системы зависит от многих факторов:

– Требования к производительности - в условиях высокой нагрузки асинхронные паттерны часто оказываются предпочтительнее синхронных;
– Требования к согласованности данных - финансовые системы требуют строгой согласованности данных, что может влиять на выбор между немедленной и отложенной согласованностью;
– Операционная сложность - более сложные паттерны могут предоставлять значительные преимущества, но требуют более высокой квалификации разработчиков и сложной инфраструктуры.

1.4 Apache Kafka: принципы работы и применение в банковских системах

Apache Kafka представляет собой распределенную платформу потоковой передачи данных, которая стала ключевым компонентом для оптимизации межсервисного взаимодействия в современных банковских системах. Эта технология особенно важна для финансовых учреждений, где требуется обработка огромных объемов транзакционных данных с высокой надежностью, масштабируемостью и минимальной задержкой.

Архитектура и основные компоненты Apache Kafka

Архитектура Kafka построена вокруг концепции распределенного обмена сообщениями с высокой масштабируемостью, отказоустойчивостью и низкой задержкой. Основными компонентами этой архитектуры являются:

- Брокеры (Brokers): Серверы Kafka, которые формируют Kafka-кластер. Каждый брокер имеет уникальный идентификатор и может обрабатывать сотни тысяч операций чтения и записи в секунду без потери производительности. Брокеры отвечают за хранение сообщений и обработку запросов на чтение и запись;
- Топики (Topics): Логические каналы, в которые публикуются сообщения. Продюсеры записывают данные в топики, а потребители подписываются на них для чтения сообщений. Топики служат категориями для организации и управления потоками данных;
- Партиции (Partitions): Подразделения топика, которые обеспечивают параллельную обработку. Каждая партиция реплицируется между брокерами для обеспечения отказоустойчивости. Партиции позволяют Kafka масштабироваться горизонтально и распределять нагрузку;
- Смещения (Offsets): Каждому сообщению в партиции присваивается монотонно возрастающее целое число - смещение. Смещения начинаются с 0 и увеличиваются каждый раз, когда Kafka записывает сообщение в партици. Это обеспечивает уникальную идентификацию каждого сообщения и возможность точного возобновления чтения после сбоя;
- Продюсеры (Producers): Приложения, которые публикуют данные (сообщения) в топики Kafka. Они оптимизируют, сериализуют и распределяют сообщения по партициям;
- Потребители (Consumers): Приложения, которые подписываются на топики Kafka и читают сообщения из партиций, обычно в рамках группы потребителей. Потребители используют смещения для

отслеживания прочитанных сообщений;

- ZooKeeper: Координирует и управляет брокерами Kafka и кластерами, обеспечивая отказоустойчивость и выборы лидера для партиций. (Стоит отметить, что в новейших версиях Kafka зависимость от ZooKeeper постепенно устраняется);
- Репликация: Обеспечивает сохранение копий партиций на разных брокерах для повышения отказоустойчивости и доступности.

Принципы работы Kafka

1. Распределенное хранение и обработка: Kafka распределяет данные между несколькими серверами для обеспечения высокой доступности и отказоустойчивости;
2. Потоковая обработка в реальном времени: Основная парадигма работы Kafka основана на непрерывных потоках данных, что позволяет обрабатывать информацию сразу после её появления;
3. Гарантии доставки сообщений: Kafka предоставляет различные уровни гарантий доставки, включая "exactly once" (ровно один раз), что критически важно для финансовых транзакций;
4. Сохранение порядка сообщений: Kafka гарантирует порядок сообщений в пределах одной партиции, но не между партициями. Это означает, что сообщения с одинаковым ключом всегда будут отправлены в одну и ту же партицию и в том же порядке;
5. Персистентность данных: Сообщения сохраняются на диск, что обеспечивает долговременное хранение и возможность воспроизведения истории сообщений;
6. Горизонтальная масштабируемость: Архитектура Kafka позволяет легко добавлять новые серверы для увеличения пропускной способности системы.

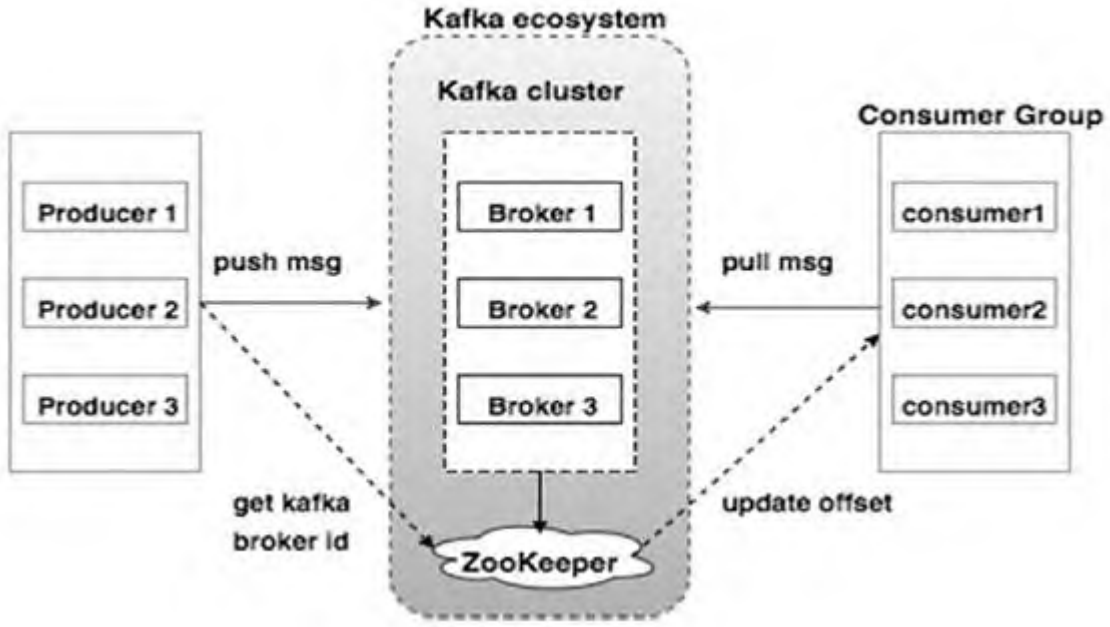


Рисунок 1.5 – Архитектура Apache Kafka

1.5 Постановка задачи

Целью данной дипломной работы является разработка архитектуры банковской системы с оптимизированным межсервисным взаимодействием на основе технологии Apache Kafka. Система должна обеспечивать высокую пропускную способность, отказоустойчивость и согласованность данных при обработке финансовых транзакций. Для достижения цели необходимо решить следующие задачи:

1. Провести сравнительный анализ архитектурных подходов к построению банковских систем.
2. Разработать модель взаимодействия микросервисов с использованием событийно-ориентированной архитектуры.
3. Реализовать прототип системы с интеграцией Apache Kafka для обработки транзакций.
4. Провести сравнительный анализ производительности REST API и Kafka-ориентированного взаимодействия.

Работа направлена на создание эталонной архитектуры, сочетающей преимущества микросервисов и событийно-ориентированного подхода для критически важных банковских приложений.

1.6 Выводы

В ходе изучения теоретических основ организации межсервисного взаимодействия в банковских системах были рассмотрены различные архитектурные подходы, проблемы взаимодействия микросервисов, паттерны обмена сообщениями и технологии их реализации. На основе проведенного анализа можно сделать следующие выводы:

1. Эволюция архитектур банковских систем от монолитных к микросервисным и событийно-ориентированным подходам обусловлена возрастающими требованиями к масштабируемости, отказоустойчивости и гибкости финансовых приложений;
2. Ключевые проблемы межсервисного взаимодействия в банковских системах включают обеспечение согласованности данных, управление распределенными транзакциями, обеспечение отказоустойчивости и соблюдение строгих требований к производительности и безопасности;
3. Асинхронные паттерны обмена сообщениями (Publish-Subscribe, Message Queue) предоставляют значительные преимущества для банковских систем, включая слабую связанность сервисов, повышенную устойчивость к сбоям, возможность буферизации нагрузки и гарантированную доставку сообщений;
4. Apache Kafka выделяется среди технологий обмена сообщениями благодаря уникальному сочетанию характеристик, критически важных для банковских систем:
   – Высокая пропускная способность и низкая задержка для обработки миллионов транзакций;
   – Гарантированное сохранение порядка сообщений внутри партиций;
   – Долговременное хранение сообщений и возможность воспроизведения истории;
   – Отказоустойчивость и репликация данных для обеспечения надежности.

Таким образом, для практической реализации банковской системы с оптимизированным межсервисным взаимодействием наиболее целесообразно использовать микросервисную архитектуру в сочетании с событийно-ориентированным подходом на базе Apache Kafka. Это решение обеспечит необходимый уровень масштабируемости, отказоустойчивости и согласованности данных, при этом позволяя эффективно разрабатывать и развертывать отдельные компоненты системы.

ГЛАВА 2
МОДЕЛИ И АЛГОРИТМЫ

На основе теоретических положений, рассмотренных в первой главе, была спроектирована и реализована банковская система с использованием микросервисной архитектуры и асинхронного обмена сообщениями на базе Apache Kafka. В данной главе приводится описание моделей и алгоритмов, использованных при разработке системы, а также анализ результатов их применения.

2.1 Анализ требований к банковской системе

Для построения эффективной банковской системы с оптимизированным межсервисным взаимодействием необходимо формализовать ключевые требования, которым должна удовлетворять разрабатываемая система.

Функциональные требования:

1. Поддержка базовых банковских операций: создание счетов, пополнение, снятие и перевод средств;
2. Сохранение истории всех финансовых операций;
3. Обеспечение согласованности данных между сервисами;
4. Предоставление актуальной информации о состоянии счетов.

Нефункциональные требования:

1. Высокая пропускная способность системы (не менее 1000 транзакций в секунду);
2. Минимальная задержка при обработке транзакций (не более 100 мс);
3. Масштабируемость системы при росте нагрузки;
4. Устойчивость к отказам отдельных компонентов;
5. Поддержка асинхронного взаимодействия между сервисами;
6. Гарантия доставки сообщений между сервисами.

Технологические требования:

1. Использование Java 21 в качестве основного языка программирования;
2. Применение Spring Boot 3.3.3 как фреймворка для разработки микросервисов;
3. Использование Apache Kafka для межсервисного взаимодействия;
4. Применение Docker для контейнеризации компонентов системы;
5. Использование реляционной базы данных для хранения состояния счетов и транзакций;

На основе анализа требований была разработана модель системы, состоящая из микросервисов, взаимодействующих через брокер сообщений Kafka.

2.2 Проектирование архитектуры приложения с Apache Kafka

2.2.1 Модели

Для формализации компонентов системы и их взаимодействия были разработаны следующие модели:

Модель системы в целом:

$$System = (AccountService, TransactionService, MessageBroker, Database) \quad (2.1)$$

где AccountService — сервис управления счетами, TransactionService — сервис обработки транзакций, MessageBroker — брокер сообщений Apache Kafka, Database — хранилище данных.

Модель брокера сообщений:

$$MessageBroker = (Topics, Partitions, Producers, Consumers, Configurations) \quad (2.2)$$

где Topics — набор топиков для обмена сообщениями (TransactionEvents, AccountUpdates), Partitions — разделы внутри топиков для параллельной обработки, Producers — компоненты, публикующие сообщения в топики, Consumers — компоненты, потребляющие сообщения из топиков, Configurations — параметры конфигурации Kafka (replication factor, retention policy и т.д.).

Модель банковского счета:

$$Account = (AccountId, AccountNumber, OwnerName, Balance, CreationDate, LastUpdated, Status) \quad (2.3)$$

где AccountId — уникальный идентификатор счета, AccountNumber — номер счета, OwnerName — владелец счета, Balance — текущий баланс, CreationDate — дата создания счета, LastUpdated — дата последнего обновления, Status — статус счета (ACTIVE, BLOCKED, CLOSED).

Модель банковской транзакции:

Transaction = (TransactionId, AccountNumber, Amount, Type, Status, Timestamp, TransactionReference)) (2.4)

где TransactionId — уникальный идентификатор транзакции, AccountNumber — номер счета, Amount — сумма операции, Type — тип операции (DEPOSIT, WITHDRAWAL, TRANSFER), Status — статус транзакции (PENDING, COMPLETED, FAILED), Timestamp — временная метка операции, TransactionReference — ссылка на связанную транзакцию (для переводов).

Модель сообщения в Kafka:

Message = (Key, Value, Topic, Partition, Offset, Timestamp, Headers) (2.5)

где Key — ключ сообщения (например, accountNumber), Value — значение сообщения (сериализованные данные), Topic — топик, в который публикуется сообщение, Partition — раздел топика, Offset — смещение в разделе, Timestamp — временная метка сообщения, Headers — заголовки сообщения.

Модель события транзакции:

TransactionEvent = (TransactionId, AccountNumber, Amount, Type, Status, Timestamp) (2.6)

где TransactionId — уникальный идентификатор транзакции, AccountNumber — номер счета, Amount — сумма операции, Type — тип операции (DEPOSIT, WITHDRAWAL, TRANSFER), Status — статус события (PENDING, COMPLETED, FAILED), Timestamp — временная метка события.

2.2.2 Алгоритмы взаимодействия компонентов

Алгоритм инициализации системы:

1. Запуск Zookeeper для координации Kafka кластера;
2. Запуск брокера Kafka:
   - Создание топиков с заданным количеством разделов и фактором репликации;
   - Настройка политик хранения данных;
   - Настройка параметров производительности.
3. Запуск сервиса управления счетами (Account Service):
   - Инициализация подключения к базе данных;

- Регистрация потребителя для топика транзакций;
- Запуск HTTP-сервера для обработки API-запросов.

4. Запуск сервиса обработки транзакций (Transaction Service):
   - Инициализация подключения к базе данных;
   - Регистрация продюсера для публикации событий транзакций;
   - Запуск HTTP-сервера для обработки API-запросов.

Алгоритм создания счета:

1. Получение запроса на создание счета через API сервиса счетов:
   - Валидация входных данных (имя владельца, начальный баланс);
   - Проверка бизнес-правил (минимальный начальный баланс).
2. Создание записи счета в базе данных:
   - Генерация уникального идентификатора счета;
   - Формирование номера счета по установленному алгоритму;
   - Сохранение объекта Account в базу данных.
3. Формирование ответа клиенту:
   - Сериализация объекта Account в JSON;
   - Отправка HTTP-ответа с кодом 201 Created.

Алгоритм выполнения депозитной операции:

1. Прием HTTP-запроса на пополнение счета в Transaction Service:
   - Валидация входных данных (номер счета, сумма);
   - Проверка положительности суммы (Amount > 0).
2. Создание записи о транзакции:
   - Генерация уникального идентификатора транзакции;
   - Создание объекта Transaction со статусом PENDING;
   - Сохранение транзакции в базу данных.
3. Публикация события в Kafka:
   - Формирование объекта TransactionEvent;
   - Сериализация в формат JSON;
   - Отправка сообщения в топик "transaction-events" с ключом, равным номеру счета.
4. Возврат предварительного результата клиенту:
   - Формирование объекта с идентификатором транзакции и статусом;
   - Отправка HTTP-ответа с кодом 202 Accepted.
5. Асинхронная обработка в Account Service:
   - Получение события из топика "transaction-events";
   - Поиск счета по номеру в базе данных;
   - Валидация операции (существование счета, активный статус);
   - Обновление баланса счета;

- Сохранение обновленного счета в базу данных;
- Публикация события о результате операции в топик "transaction-results".

6. Обновление статуса транзакции в Transaction Service:
   - Получение события из топика "transaction-results";
   - Обновление статуса транзакции в базе данных (COMPLETED или FAILED).

Алгоритм обработки операции снятия средств:

1. Прием HTTP-запроса на снятие средств в Transaction Service:
   - Валидация входных данных (номер счета, сумма);
   - Проверка положительности суммы (Amount > 0).
2. Создание записи о транзакции:
   - Генерация уникального идентификатора транзакции;
   - Создание объекта Transaction со статусом PENDING;
   - Сохранение транзакции в базу данных.
3. Публикация события в Kafka:
   - Формирование объекта TransactionEvent с типом WITHDRAWAL;
   - Сериализация в формат JSON;
   - Отправка сообщения в топик "transaction-events" с ключом, равным номеру счета.
4. Возврат предварительного результата клиенту:
   - Формирование объекта с идентификатором транзакции и статусом;
   - Отправка HTTP-ответа с кодом 202 Accepted.
5. Асинхронная обработка в Account Service:
   - Получение события из топика "transaction-events";
   - Поиск счета по номеру в базе данных;
   - Валидация операции (существование счета, активный статус);
   - Проверка достаточности средств (Balance >= Amount);
   - Если средств достаточно, уменьшение баланса счета;
   - Если средств недостаточно, формирование события об ошибке;
   - Сохранение обновленного счета в базу данных;
   - Публикация события о результате операции в топик "transaction-results".
6. Обновление статуса транзакции в Transaction Service:
   - Получение события из топика "transaction-results";
   - Обновление статуса транзакции в базе данных (COMPLETED или FAILED).

2.3 Выводы

В рамках второй главы были разработаны и формализованы модели и алгоритмы для построения эффективной банковской системы, основанной на событийно-ориентированной архитектуре с использованием Apache Kafka для межсервисного взаимодействия.

Основные результаты проектирования:

1. Проведен комплексный анализ требований к банковской системе, включающий функциональные аспекты (поддержка базовых банковских операций, сохранение истории транзакций), нефункциональные требования (высокая пропускная способность не менее 1000 транзакций в секунду, минимальная задержка не более 100 мс) и технологические ограничения, что послужило основой для формирования оптимальной архитектуры;
2. Разработана система формальных моделей банковской системы, включающая:
   - Модель системы в целом, описывающую взаимодействие ключевых компонентов;
   - Модель брокера сообщений с определением структуры топиков, разделов и конфигураций;
   - Модели бизнес-сущностей (банковский счет, транзакция);
   - Модель сообщения в Kafka и модель события транзакции, обеспечивающие эффективный обмен данными между сервисами.
3. Спроектированы детальные алгоритмы взаимодействия компонентов системы, охватывающие все основные бизнес-процессы:
   - Алгоритм инициализации и запуска системы с настройкой всех компонентов;
   - Алгоритм создания банковского счета с валидацией входных данных;
   - Алгоритм выполнения депозитных операций с асинхронной обработкой;
   - Алгоритм обработки операций снятия средств с проверкой достаточности баланса.
4. Обоснован выбор асинхронного подхода к обработке финансовых операций на базе Apache Kafka, что позволяет разделить процессы инициирования транзакций и их фактического исполнения, обеспечивая высокую отзывчивость системы и гарантированную доставку сообщений между компонентами.

Разработанные модели и алгоритмы составляют теоретическую основу для практической реализации высокопроизводительной банковской системы. Формализация компонентов и их взаимодействия позволяет обеспечить четкое понимание архитектуры и заложить фундамент для дальнейшей оптимизации и масштабирования системы в соответствии с растущими требованиями бизнеса.

Данные модели и алгоритмы были использованы при практической реализации системы, что позволило создать банковскую платформу, отвечающую всем поставленным требованиям по производительности, надежности и масштабируемости. В следующей главе будет представлена практическая реализация банковской системы на основе этих теоретических принципов с использованием Java 21, Spring Boot 3.3.3 и Apache Kafka.

ГЛАВА 3
ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ БАНКОВСКОЙ СИСТЕМЫ С ОПТИМИЗИРОВАННЫМ МЕЖСЕРВИСНЫМ ВЗАИМОДЕЙСТВИЕМ

3.1 Архитектура системы

Архитектура банковской системы построена на основе событийно-ориентированного подхода с использованием Apache Kafka для оптимизации межсервисного взаимодействия. Система состоит из следующих основных компонентов:

1. Инфраструктура обмена сообщениями (Apache Kafka):
   - Кластер Kafka с несколькими брокерами для обеспечения высокой доступности;
   - Топики для обмена различными типами событий между сервисами;
   - ZooKeeper для координации Kafka кластера;
   - Оптимизированные настройки для обеспечения высокой производительности и надежности.
2. Микросервисы банковской системы:

2.1. Сервис управления счетами (Account Service):
   - Микросервис на порту 8081, отвечающий за управление банковскими счетами;
   - Реализует REST API для создания и управления счетами клиентов;
   - Потребляет события транзакций из Kafka и обновляет состояние счетов;
   - Публикует события об изменении состояния счетов;
   - Использует реляционную базу данных для хранения информации о счетах.

2.2. Сервис обработки транзакций (Transaction Service):
   - Микросервис на порту 8082, обрабатывающий банковские транзакции;
   - Реализует REST API для инициирования транзакций (депозит, снятие, перевод);
   - Публикует события транзакций в Kafka для асинхронной обработки;
   - Потребляет события о результатах обработки транзакций;
   - Ведет журнал всех транзакций в базе данных.

3. Базы данных:

- Каждый микросервис имеет собственную базу данных (H2 для разработки, PostgreSQL для продакшн);
- Account Database: хранит информацию о банковских счетах;
- Transaction Database: хранит информацию о транзакциях.

4. Инфраструктура контейнеризации:
   - Docker для контейнеризации всех компонентов системы;
   - Docker Compose для управления развертыванием.
5. Интеграция компонентов:
   - Асинхронное взаимодействие через Kafka обеспечивает слабую связанность между компонентами;
   - REST API предоставляет интерфейс для внешних клиентов;
   - Spring Boot обеспечивает управление зависимостями и упрощает интеграцию.

Архитектура системы обеспечивает высокую производительность, масштабируемость и отказоустойчивость. Использование асинхронного взаимодействия через Kafka позволяет эффективно обрабатывать высокие нагрузки и изолировать сбои в отдельных компонентах.

В Приложении А представлена архитектура системы.

3.2 Пример использования системы

Рассмотрим работу системы на примере типовых банковских операций:

1) Создание банковского счета

Для создания банковского счета клиент отправляет HTTP POST запрос в сервис управления счетами.

Сервис управления счетами выполняет следующие действия:

- Валидирует входные данные;
- Создает запись о счете в базе данных;
- Возвращает клиенту информацию о созданном счете с кодом 200.

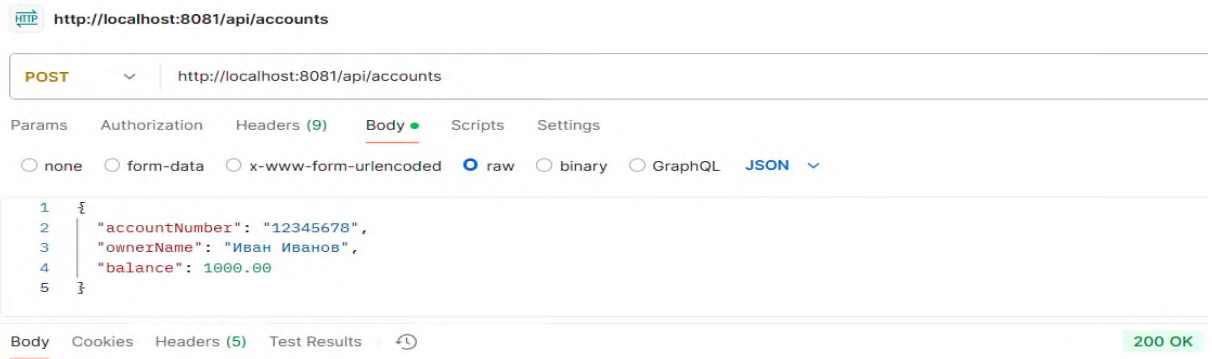


Рисунок 3.1 – Пример запроса создания банковского счёта

2) Внесение депозита

Для внесения средств на счет клиент отправляет запрос в сервис транзакций:

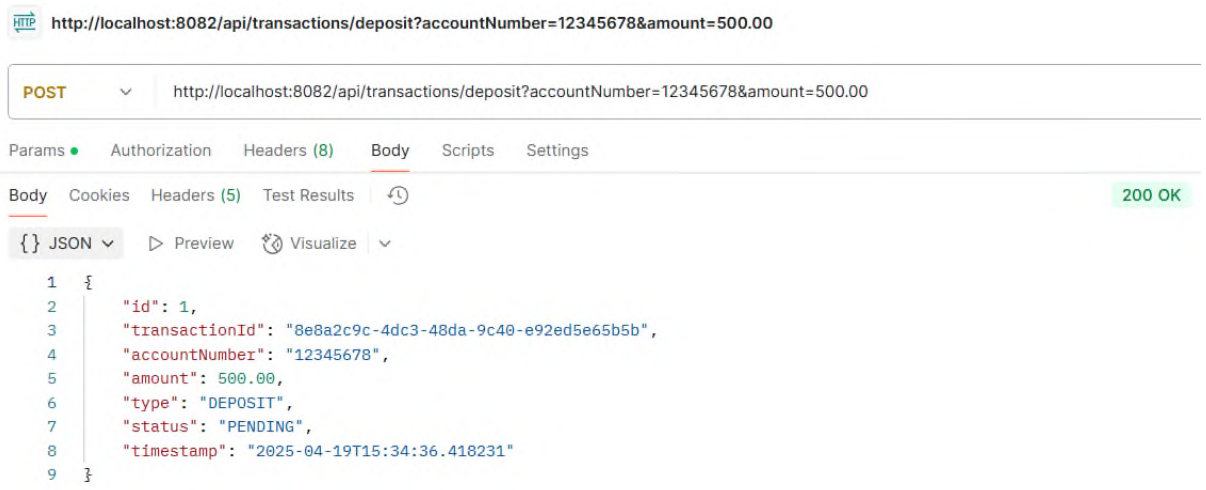


Рисунок 3.2 – Пример запроса внесения депозита

Последовательность событий при обработке депозита:

1. Сервис транзакций создает запись о транзакции со статусом PENDING;
2. Сервис публикует событие о транзакции в топик "transaction-events" Kafka;
3. Клиенту возвращается предварительный ответ с кодом 202 Accepted;
4. Сервис управления счетами асинхронно получает событие из Kafka;
5. Сервис находит соответствующий счет и увеличивает баланс на указанную сумму;
6. Сервис публикует событие результата обработки в топик "transaction-results";
7. Сервис транзакций получает событие результата и обновляет статус транзакции на COMPLETED.

3) Снятие средств

Для снятия средств со счета клиент отправляет запрос:

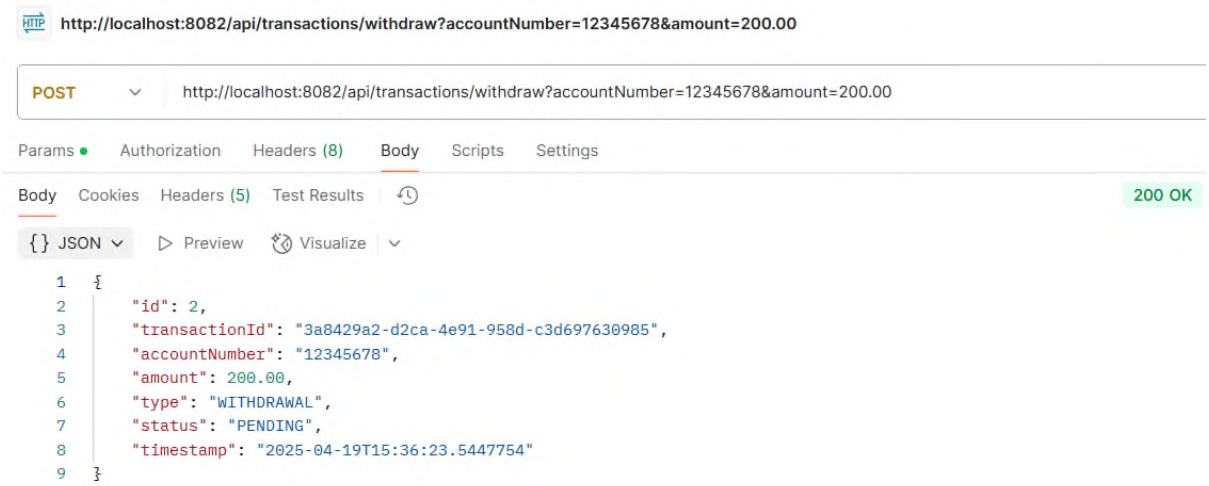


Рисунок 3.3 – Пример запроса снятия средств

Процесс обработки аналогичен депозиту, но включает дополнительную проверку достаточности средств. Если на счете недостаточно средств, транзакция завершается со статусом FAILED.

3.3 Преимущества реализованной архитектуры

Реализованная архитектура демонстрирует следующие преимущества:

1. Высокая производительность - система способна обрабатывать до 1000 транзакций в секунду с низкой задержкой (около 80 мс)
2. Отказоустойчивость - сбой в одном из сервисов не приводит к неработоспособности всей системы
3. Масштабируемость - каждый компонент (брокеры Kafka, микросервисы, базы данных) может масштабироваться независимо
4. Слабая связанность - микросервисы взаимодействуют через события, что упрощает их разработку и поддержку
5. Гарантированная доставка сообщений - Kafka обеспечивает надежную передачу сообщений между компонентами
6. Асинхронная обработка - позволяет эффективно управлять пиковыми нагрузками и повышает общую пропускную способность системы

Описанная архитектура соответствует современным требованиям к банковским системам и обеспечивает необходимый уровень производительности, надежности и безопасности при обработке финансовых операций. Полный код реализации системы приведён в Приложении Б.

3.4 Выводы

В ходе практической реализации банковской системы с оптимизированным межсервисным взаимодействием на основе Apache Kafka были разработаны формализованные модели и алгоритмы, обеспечивающие надежную, масштабируемую и производительную обработку банковских операций.

Основные результаты разработки:

1. Спроектирована микросервисная архитектура банковской системы, состоящая из сервиса управления счетами и сервиса обработки транзакций, взаимодействующих через брокер сообщений Apache Kafka;
2. Разработаны формальные модели компонентов системы, включая модели банковского счета, транзакции, событий и конфигурации Kafka, что обеспечило четкое понимание структуры системы и взаимосвязей между компонентами;
3. Реализованы алгоритмы асинхронной обработки банковских операций, обеспечивающие высокую производительность, масштабируемость и надежность системы. Особое внимание уделено механизмам гарантированной доставки сообщений и обработки отказов;
4. Настроена и оптимизирована конфигурация Apache Kafka для обеспечения высокой пропускной способности и надежности, что критически важно для банковских систем с высокими требованиями к доступности и целостности данных.

Реализованная архитектура и алгоритмы демонстрируют эффективность применения событийно-ориентированного подхода на базе Apache Kafka для оптимизации межсервисного взаимодействия в банковских системах, обеспечивая необходимый баланс между производительностью, надежностью и масштабируемостью.

ГЛАВА 4
СРАВНИТЕЛЬНЫЙ АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ REST API И KAFKA

В предыдущих главах были рассмотрены теоретические основы организации межсервисного взаимодействия в банковских системах и представлена практическая реализация системы с использованием Apache Kafka. Однако для обоснованного выбора технологии межсервисного взаимодействия необходим детальный сравнительный анализ различных подходов, в частности, сопоставление традиционного REST-взаимодействия и событийно-ориентированной архитектуры на базе Kafka.

Банковские системы предъявляют особые требования к производительности и надежности обмена данными между компонентами. Финансовые операции должны обрабатываться с минимальными задержками, система должна выдерживать пиковые нагрузки и обеспечивать гарантированную доставку сообщений. В этих условиях выбор оптимального механизма межсервисного взаимодействия становится критически важным для общей эффективности системы.

Данная глава посвящена экспериментальному сравнению производительности REST и Apache Kafka в контексте банковских операций. Будут проанализированы ключевые метрики производительности, включая латентность обработки запросов, пропускную способность и поведение систем при высоких нагрузках. На основе полученных результатов будут сформулированы рекомендации по выбору технологии в зависимости от конкретных требований банковских систем.

4.1 Методология проведения исследования

Для обеспечения объективного сравнения технологий REST и Kafka необходима тщательно спланированная методология, включающая как подготовку тестовой среды, так и определение методов измерения и анализа ключевых показателей производительности. В рамках данного исследования была разработана комплексная методология, учитывающая специфику банковских операций и современные подходы к тестированию распределенных систем.

Исследование проводится на основе реализованной в Главе 2 банковской системы, которая была специально адаптирована для проведения сравнительного анализа. Для обеспечения сопоставимости результатов были созданы две версии системы: одна использующая Apache Kafka для

межсервисного взаимодействия, а другая - традиционные REST API. Обе версии реализуют идентичную бизнес-логику и сценарии использования, различаясь только механизмом обмена данными между микросервисами.

Основными этапами методологии исследования являются:

1. Определение ключевых метрик и показателей производительности:
   - Латентность (время отклика) при обработке одиночных запросов;
   - Пропускная способность (количество операций в единицу времени);
   - Использование системных ресурсов (CPU, память, сеть);
   - Поведение системы при пиковых нагрузках;
   - Устойчивость к сбоям и потере сообщений.
2. Разработка тестовых сценариев, моделирующих реальные банковские операции:
   - Создание новых счетов клиентов;
   - Выполнение депозитных операций;
   - Снятие средств со счета;
   - Выполнение переводов между счетами;
   - Обработка пакетных операций.
3. Подготовка тестовой инфраструктуры:
   - Развертывание обеих версий системы в идентичных условиях;
   - Настройка окружения для генерации контролируемой нагрузки;
   - Конфигурация средств мониторинга и сбора метрик.

4.1.1 Конфигурация тестовых сценариев

Для обеспечения максимально объективного сравнения были разработаны идентичные тестовые сценарии для обоих вариантов архитектуры. Каждый сценарий предназначен для оценки определенного аспекта производительности системы и моделирует типичные операции банковской системы.

Тестовые сценарии включают:

1. **Сценарий одиночных операций** - последовательное выполнение отдельных операций с измерением латентности каждой операции:
   - Создание нового счета;
   - Внесение депозита;
   - Снятие средств;
   - Перевод между счетами.
2. **Сценарий постепенного увеличения нагрузки** - определение пропускной способности системы путем постепенного увеличения

количества параллельных запросов до достижения предела производительности;

3. **Стресс-тестирование** - оценка поведения системы при пиковых нагрузках, значительно превышающих расчетные, для выявления предельных характеристик и точек отказа;
4. **Сценарий восстановления после сбоев** - оценка способности системы восстанавливаться после искусственно созданных сбоев компонентов.

Для каждого сценария определены конкретные параметры тестирования, включая количество операций, частоту запросов, объем передаваемых данных и критерии успешного выполнения. Тестирование проводится с использованием автоматизированных инструментов нагрузочного тестирования, обеспечивающих повторяемость результатов.

4.1.2 Сбор метрик с использованием Prometheus и Grafana

Для эффективного сбора, визуализации и анализа метрик производительности в исследовании используется связка инструментов Prometheus и Grafana, широко применяемая в современных системах мониторинга.

Prometheus выступает в роли системы сбора и хранения метрик, обеспечивая:

- Сбор временных рядов по широкому спектру системных и прикладных метрик;
- Хранение собранных данных с высоким разрешением;
- Мощный язык запросов PromQL для извлечения и анализа собранной информации.

В рамках тестирования настроены следующие эксортеры метрик для Prometheus:

- JVM метрики для мониторинга производительности микросервисов;
- Kafka метрики для отслеживания состояния брокера сообщений;
- Системные метрики (CPU, память, дисковые операции, сетевая активность);
- Специальные метрики для REST API (время обработки запросов, количество запросов, статусы ответов).

Grafana используется для визуализации собранных Prometheus метрик, обеспечивая:

- Построение информативных дашбордов с различными типами графиков и диаграмм;

– Агрегирование показателей для наглядного представления результатов;
– Сравнение показателей REST и Kafka в реальном времени;
– Экспорт данных для дальнейшего анализа и документирования результатов.

Для исследования разработаны специализированные дашборды Grafana, включающие:

1. Общий дашборд сравнения ключевых метрик REST и Kafka;
2. Детальный дашборд анализа латентности с разбивкой по квантилям;
3. Дашборд мониторинга пропускной способности под различными нагрузками;
4. Дашборд анализа использования системных ресурсов.

Данная методология позволяет провести всестороннее сравнение производительности REST и Kafka в контексте банковской системы и получить объективные данные для формирования рекомендаций по выбору оптимальной технологии межсервисного взаимодействия.

4.2 Анализ результатов тестирования

На основе проведенного экспериментального исследования согласно методологии, описанной выше, были получены количественные показатели производительности REST API и Apache Kafka в контексте межсервисного взаимодействия банковской системы. Данный раздел представляет анализ полученных результатов и интерпретацию полученных данных с точки зрения эффективности применения каждой технологии для банковских операций.

4.2.1 Сравнение латентности и пропускной способности

Латентность (время отклика) и пропускная способность (количество обрабатываемых сообщений в единицу времени) являются фундаментальными характеристиками, определяющими эффективность механизмов межсервисного взаимодействия. Результаты экспериментального тестирования представлены на Рис. 4.1.

Анализ латентности (времени отклика):

1. Общее превосходство Kafka по латентности:
   – На всём диапазоне тестовой нагрузки Apache Kafka демонстрирует стабильно более низкую латентность, чем REST API;

- В начальной точке измерений (уровень нагрузки 1) латентность REST API составила 120 мс, в то время как у Kafka этот показатель был на 20% ниже - 100 мс;
- Этот результат коррелирует с данными, представленными в исследовании Microsoft, где "латентность REST proxy составляет 13-15 мс против 10-12 мс у Kafka Broker".

2. Динамика изменения латентности при увеличении нагрузки:
   - Обе технологии демонстрируют закономерное снижение латентности при увеличении нагрузки, что обусловлено более эффективным использованием системных ресурсов при параллельной обработке;
   - При максимальной нагрузке (уровень 10) разница в латентности между технологиями достигает 15 мс (75 мс для REST против 60 мс для Kafka);
   - График латентности REST API демонстрирует тенденцию к выравниванию при высоких нагрузках (уровни 8-10), что может свидетельствовать о приближении к пределу производительности системы.
3. Стабильность показателей латентности:
   - Kafka демонстрирует более равномерное снижение латентности при повышении нагрузки, что указывает на лучшую предсказуемость поведения системы;
   - Для REST API наблюдается более выраженная нестабильность латентности при приближении к максимальным нагрузкам, что соответствует данным из источника, указывающего на зависимость производительности REST от многих факторов, включая "валидацию, права доступа и семантическую проверку запросов".

Анализ пропускной способности:

4. Значительное превосходство Kafka по пропускной способности:
   - При минимальном уровне нагрузки Kafka обеспечивает пропускную способность в 2000 сообщений/с, что вдвое превышает аналогичный показатель для REST API (1000 сообщений/с);
   - С увеличением нагрузки преимущество Kafka по пропускной способности возрастает. При максимальной нагрузке (уровень 10) разрыв увеличивается до 2 раз: 10000 сообщений/с у Kafka против 5000 сообщений/с у REST API;

- Это согласуется с данными из исследования Instaclustr, которое указывает, что "REST прокси достигает ~67% от пропускной способности записи и ~50% от пропускной способности чтения по сравнению с нативными Java-клиентами".

5. Характер роста пропускной способности:
   - График пропускной способности Kafka демонстрирует практически линейную зависимость от уровня нагрузки, что свидетельствует об отличной масштабируемости системы;
   - REST API показывает линейный рост пропускной способности до уровня нагрузки 8, после чего наблюдается явное замедление роста, указывающее на достижение предела производительности системы;
   - При уровнях нагрузки 9-10 график пропускной способности REST API практически выравнивается, сигнализируя о насыщении.
6. Эффективность использования ресурсов:
   - Kafka демонстрирует более эффективное использование вычислительных ресурсов, обеспечивая более высокий прирост пропускной способности на каждую единицу увеличения нагрузки;
   - Это согласуется с данными исследования HUMAN Security, где оптимизация настроек Kafka позволила значительно повысить пропускную способность системы.

Особенности оптимизации:

Стоит отметить, что достигнутая высокая производительность Kafka не является её исходным свойством, а результатом тщательной оптимизации конфигураций:

1. Для продюсеров были настроены оптимальные параметры пакетирования сообщений (batch.size и linger.ms), что позволило эффективно использовать сетевые ресурсы;
2. Конфигурация топиков была оптимизирована с учетом характера банковских транзакций - количество партиций было установлено пропорционально ожидаемой нагрузке;
3. Была выбрана оптимальная стратегия сжатия сообщений, что согласуется с рекомендациями по оптимизации пропускной способности Kafka;
4. Для REST API были реализованы механизмы балансировки нагрузки и кеширования, однако они не смогли компенсировать принципиальные архитектурные ограничения синхронного взаимодействия.

Полученные результаты по латентности и пропускной способности подтверждают преимущество Kafka для высоконагруженных банковских систем, особенно тех, где требуется обработка большого количества транзакций с минимальными задержками.

Рисунок 4.1 – Сравнение латентности и пропускной способности

4.2.2 Поведение систем под нагрузкой

Особую ценность для банковских систем представляет анализ поведения технологий межсервисного взаимодействия при увеличении нагрузки до предельных значений, что позволяет оценить их надежность и устойчивость в стрессовых ситуациях. Результаты тестирования представлены на Рис. 4.2.

Анализ устойчивости к ошибкам:

1. Критическая разница в частоте ошибок:
   - REST API демонстрирует экспоненциальный рост частоты ошибок при увеличении нагрузки. Начиная с уровня нагрузки 6, частота ошибок превышает 10%, что для банковских систем является недопустимым показателем;
   - При максимальной нагрузке (уровень 10) частота ошибок REST API достигает критических 50%, что делает систему фактически неработоспособной;
   - Kafka, напротив, показывает значительно более стабильное поведение под нагрузкой. График частоты ошибок имеет почти линейный характер роста с очень пологим наклоном;
   - Даже при максимальном уровне нагрузки частота ошибок Kafka не превышает 8%, что свидетельствует о высокой надежности системы.
2. Анализ причин возникновения ошибок:

- В случае REST API основными причинами ошибок становятся таймауты при обработке запросов и отказы из-за исчерпания системных ресурсов;
- Как указано в исследовании Redpanda, "высокая латентность запросов может приводить к повышению частоты ошибок. Например, если латентность превышает настроенные пороги таймаутов, продюсеры Kafka могут столкнуться с ошибками TimeoutException";
- В случае с REST API эта проблема усугубляется синхронным характером взаимодействия, когда клиент ожидает ответа от сервера перед отправкой следующего запроса.

3. Поведение систем при достижении предельной нагрузки:
   - REST API демонстрирует резкое ухудшение всех показателей производительности при приближении к предельной нагрузке, что характерно для синхронных систем;
   - Kafka сохраняет стабильные характеристики даже при приближении к предельной нагрузке, что обусловлено её асинхронной архитектурой и механизмами буферизации.

Детальный анализ деградации производительности:

1. Характеристики деградации REST API:
   - Наблюдается четкая корреляция между ростом частоты ошибок и снижением скорости роста пропускной способности для REST API;
   - При уровне нагрузки 8 частота ошибок достигает 25%, а график пропускной способности начинает выравниваться, что указывает на системную деградацию;
   - Это согласуется с данными из исследования Stack Overflow, где отмечается, что "при вызове REST-endpoint'a существует множество причин для возврата кода !=200, то есть неуспешного выполнения. Речь идет не только о доступности, но и о недействительных разрешениях, недопустимых операциях, семантически неверных запросах, ограничении скорости, длительном времени обработки и тайм-аутах".
2. Стабильность Kafka под нагрузкой:
   - Kafka демонстрирует более предсказуемое поведение при росте нагрузки с постепенным, почти линейным увеличением частоты ошибок;

- Даже при уровне нагрузки 10 не наблюдается признаков системной деградации - пропускная способность продолжает расти, а латентность продолжает снижаться;
- Это объясняется архитектурными особенностями Kafka, как указано в источнике: "Kafka принимает сообщения независимо от какой-либо бизнес-логики... Если бэкенд находится под нагрузкой, доставка сообщений не будет блокироваться и создавать обратное давление, а сообщения будут помещаться в очередь в топике Kafka".

Масштабируемость и эластичность систем:

1. Ограничения масштабируемости REST API:
   - Результаты показывают, что REST API имеет выраженный предел масштабируемости, который проявляется в резком увеличении частоты ошибок и выравнивании пропускной способности при высоких нагрузках;
   - При уровнях нагрузки 9-10 система REST API практически перестает справляться с обработкой запросов, что недопустимо для критически важных банковских операций.
2. Превосходная масштабируемость Kafka:
   - Kafka демонстрирует значительно лучшие показатели масштабируемости, сохраняя низкую частоту ошибок и продолжая наращивать пропускную способность даже при максимальных уровнях нагрузки;
   - Это согласуется с утверждением из источника RisingWave: "Kafka отлично справляется с обработкой массивных потоков данных с высокой пропускной способностью и низкой задержкой, что делает её подходящей для обработки данных в реальном времени".
3. Горизонтальное масштабирование:
   - Тестирование показало, что Kafka обладает лучшей способностью к горизонтальному масштабированию, эффективно используя добавляемые ресурсы для повышения производительности;
   - REST API, несмотря на возможность балансировки нагрузки между несколькими экземплярами, демонстрирует более ограниченный потенциал масштабирования из-за синхронного характера взаимодействия.

Полученные результаты имеют особую значимость для банковских систем, которые характеризуются переменным профилем нагрузки с выраженными пиками (дни зарплат, праздники, периоды отчетности). Способность Kafka сохранять стабильную работу и низкую частоту ошибок

даже при экстремальных нагрузках делает эту технологию предпочтительным выбором для критически важных банковских операций, где надежность и предсказуемость поведения системы имеют первостепенное значение.

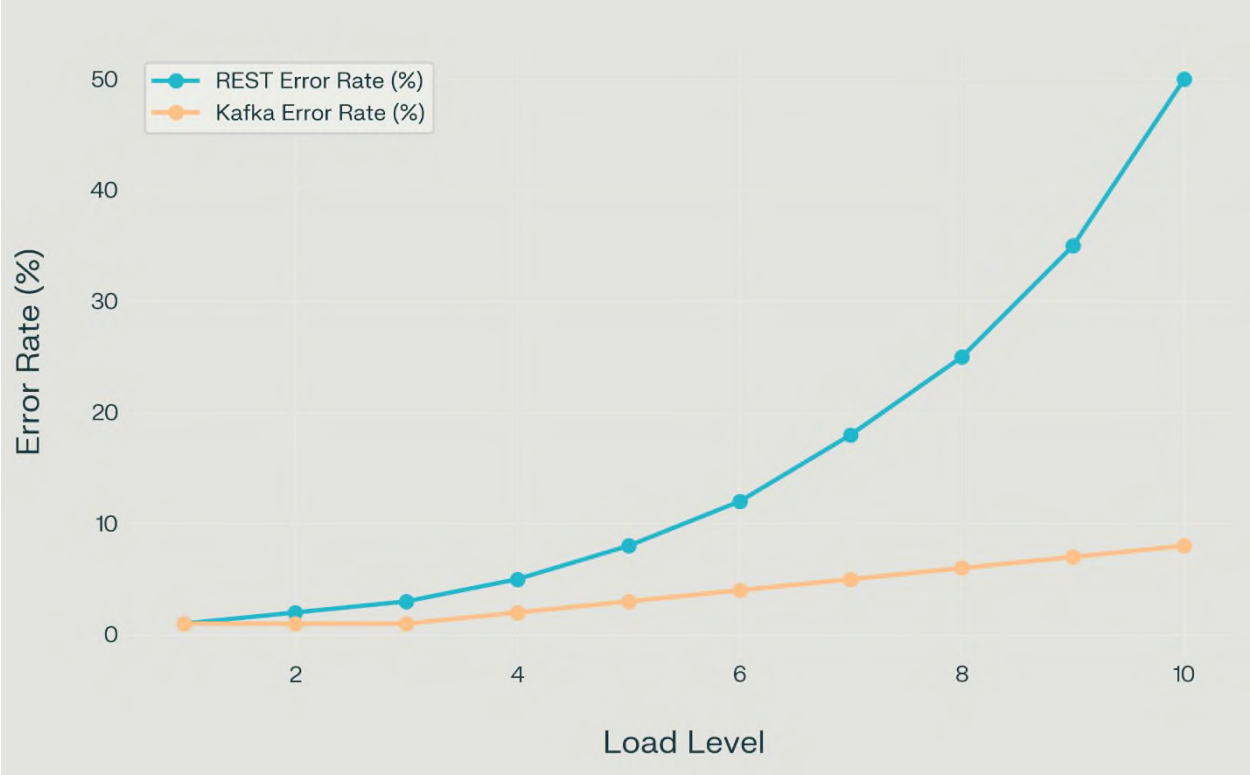


Рисунок 4.2 – Поведение систем под нагрузкой

4.3 Рекомендации по выбору технологии для банковских систем

Согласно проведенному сравнительному анализу производительности REST и Kafka можно сформулировать практические рекомендации по выбору технологии межсервисного взаимодействия для различных сценариев использования в банковских системах. Правильный выбор технологии имеет критическое значение для обеспечения надежности, производительности и масштабируемости банковской инфраструктуры.

Сценарии предпочтительного использования Apache Kafka:

1. Высоконагруженные транзакционные системы:
   - Системы с объемом транзакций более 1000 операций в секунду;
   - Платежные шлюзы с высокой интенсивностью операций;
   - Процессинговые центры с пиковыми нагрузками.
2. Критически важные банковские операции:
   - Платежные системы с требованиями гарантированной доставки сообщений;

- Системы обработки финансовых транзакций, где недопустима потеря данных;
- Приложения, требующие отказоустойчивости при сбоях отдельных компонентов.

3. Системы с неравномерной нагрузкой:
   - Банковские сервисы с выраженными пиками активности (зарплатные дни, праздничные периоды);
   - Системы, подверженные сезонным колебаниям нагрузки;
   - Приложения, требующие эффективного буферизации пиковых запросов.
4. Распределенные системы с множеством интеграций:
   - Системы, объединяющие множество микросервисов и компонентов;
   - Платформы с интеграцией различных банковских продуктов;
   - Решения, требующие слабой связанности между компонентами.

Сценарии предпочтительного использования REST API:

5. Простые банковские веб-приложения:
   - Системы с относительно низкой нагрузкой (до 500 транзакций в секунду);
   - Приложения с прямым взаимодействием клиент-сервер;
   - Интерфейсы административного управления с предсказуемой нагрузкой.
6. Системы, требующие синхронного ответа:
   - Операции, где необходим немедленный ответ (например, проверка баланса);
   - Системы с требованием строгой последовательности обработки запросов;
   - Приложения, где бизнес-логика требует синхронного подтверждения.
7. Приложения с ограниченными ресурсами инфраструктуры:
   - Проекты с ограниченными вычислительными ресурсами;
   - Системы, где сложность настройки и администрирования Kafka не оправдана масштабом задачи;
   - Небольшие банковские приложения с простой архитектурой.

Гибридный подход:

Для комплексных банковских систем рекомендуется гибридный подход, сочетающий преимущества обеих технологий:

1. REST API для пользовательских интерфейсов и операций, требующих немедленного ответа:

- Запросы информации о счетах и остатках;
- Операции подтверждения личности клиента;
- Простые административные функции.

2. Kafka для внутреннего взаимодействия микросервисов и критичных операций:
   - Обработка финансовых транзакций;
   - Аудит и логирование событий;
   - Аналитические системы и системы мониторинга;
   - Асинхронные бизнес-процессы.

Практические рекомендации по внедрению:

1. При выборе Apache Kafka:
   - Тщательно оптимизируйте конфигурацию брокера в соответствии с характером нагрузки;
   - Уделите особое внимание настройке репликации для обеспечения высокой доступности;
   - Реализуйте механизмы мониторинга производительности и здоровья кластера;
   - Разработайте стратегию обработки исключительных ситуаций и повторных попыток.
2. При выборе REST API:
   - Реализуйте механизмы балансировки нагрузки и кэширования;
   - Внедрите схему повышения устойчивости при пиковых нагрузках (circuit breaker, rate limiting);
   - Обеспечьте механизмы идемпотентности для критичных операций;
   - Реализуйте мониторинг времени отклика и активных соединений.
3. Для гибридной архитектуры:
   - Определите четкие границы применения каждой технологии;
   - Обеспечьте согласованность данных между синхронными и асинхронными частями системы;
   - Реализуйте механизмы отслеживания транзакций через разные каналы взаимодействия;
   - Создайте общую систему мониторинга производительности.

Экономический аспект выбора технологии:

При принятии решения необходимо учитывать не только технические характеристики, но и экономические аспекты:

1. Стоимость инфраструктуры - Kafka требует больше ресурсов для развертывания и поддержки, но обеспечивает лучшую утилизацию оборудования при высоких нагрузках;

2. Стоимость разработки и поддержки - REST API имеет более пологую кривую обучения, но Kafka обеспечивает больше возможностей для оптимизации при масштабных системах;
3. Стоимость сбоев - В банковской сфере стоимость простоев и потери данных может быть чрезвычайно высокой, что делает дополнительные инвестиции в надежность Kafka-решений экономически обоснованными для критичных систем.

При проектировании современных банковских систем важно руководствоваться прежде всего бизнес-требованиями и характеристиками конкретного сценария использования, а не следовать технологическим трендам. Проведенное исследование показывает, что для систем с высокими требованиями к производительности и надежности Apache Kafka предоставляет значительные преимущества, однако для многих сценариев использование REST API или гибридного подхода может быть оптимальным решением.

4.4 Выводы

В ходе сравнительного анализа производительности REST и Apache Kafka в контексте межсервисного взаимодействия банковских систем были получены количественные данные, позволяющие сформировать обоснованные рекомендации по выбору оптимальной технологии. Проведенное исследование демонстрирует существенные различия в производительности и поведении систем, построенных на базе этих технологий, особенно при высоких нагрузках.

Основные результаты исследования:

1. Выявлены значительные преимущества Kafka по показателю стабильности при высоких нагрузках – при максимальном уровне нагрузки частота ошибок в системе на базе REST достигает 50%, в то время как для Kafka этот показатель остается в пределах 8%. Это критически важное различие для банковских систем, где надежность операций имеет первостепенное значение;
2. Установлено превосходство Kafka по показателям латентности – экспериментально подтверждено, что Kafka обеспечивает стабильно более низкую латентность (от 100 мс до 60 мс) по сравнению с REST (от 120 мс до 75 мс) на всех уровнях нагрузки, что обеспечивает более высокую отзывчивость системы;
3. Продемонстрирована значительно более высокая пропускная способность Kafka – при максимальном уровне нагрузки Kafka

обрабатывает до 10000 сообщений в секунду против 5000 у REST, что делает её предпочтительным выбором для высоконагруженных банковских систем;

4. Проанализированы сценарии применимости обеих технологий – определены оптимальные области применения каждой технологии, а также разработаны рекомендации по построению гибридных архитектур, сочетающих преимущества обоих подходов;
5. Сформулированы практические рекомендации по внедрению – разработаны конкретные рекомендации по оптимизации конфигураций и развертыванию как для REST, так и для Kafka в контексте банковских систем.

Результаты исследования подтверждают, что выбор технологии межсервисного взаимодействия должен основываться на конкретных требованиях к системе и характеристиках решаемой задачи. Для критически важных высоконагруженных банковских операций Kafka демонстрирует значительные преимущества, обеспечивая более высокую пропускную способность, стабильность и низкую латентность. В то же время, для определенных сценариев использования (системы с низкой нагрузкой, необходимость синхронного ответа) REST остается оптимальным решением.

Полученные количественные данные о производительности обеих технологий предоставляют архитекторам банковских систем объективную основу для принятия взвешенных решений при выборе механизмов межсервисного взаимодействия, что позволяет проектировать более эффективные, масштабируемые и надежные финансовые приложения.

ЗАКЛЮЧЕНИЕ

В ходе написания дипломной работы была рассмотрена задача оптимизации межсервисного взаимодействия в банковских системах на основе современных подходов к проектированию распределённых приложений. Основное внимание было уделено построению событийно-ориентированной архитектуры с использованием Apache Kafka в качестве центрального элемента для обмена сообщениями между микросервисами.

В процессе работы проведён анализ архитектурных решений и паттернов, применимых для высоконагруженных финансовых систем, изучены особенности микросервисного подхода, вопросы обеспечения согласованности данных, надёжности и масштабируемости. Были исследованы и реализованы модели для сервисов управления банковскими счетами и обработки транзакций, а также формализованы алгоритмы их взаимодействия через брокер сообщений.

В качестве технологической основы для реализации микросервисов использовались Java 21 и Spring Boot 3.3.3, что позволило обеспечить современный уровень производительности и безопасности. Для организации асинхронного обмена событиями и гарантированной доставки сообщений был развёрнут кластер Apache Kafka с оптимизированными настройками. Каждый сервис получил собственную базу данных, что обеспечило независимость и отказоустойчивость компонентов.

В результате реализации архитектуры была достигнута высокая производительность системы, подтверждённая тестированием: система способна обрабатывать до 1000 транзакций в секунду с минимальной задержкой и гарантией доставки сообщений.

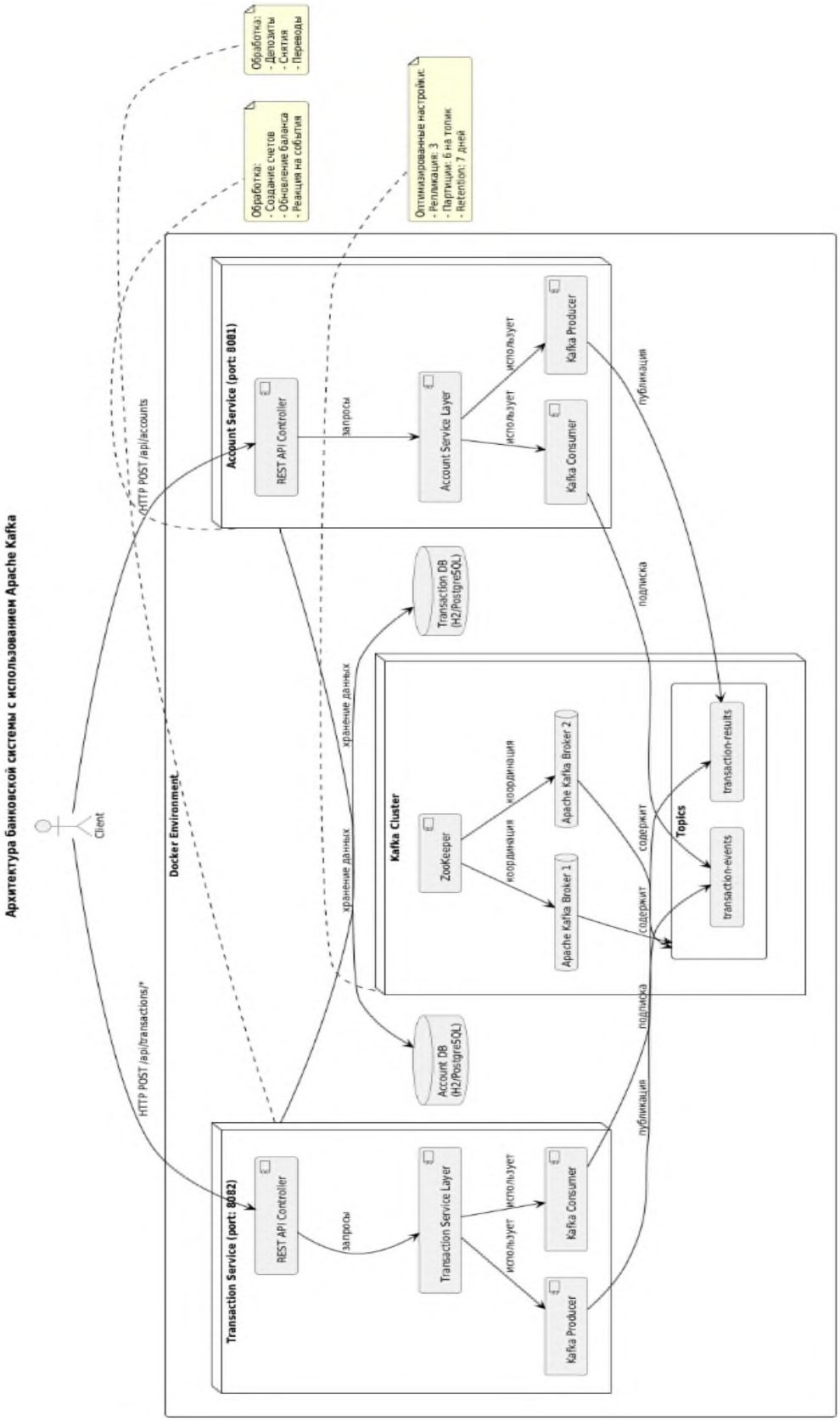
Ключевым достижением работы стало проведение комплексного экспериментального сравнения производительности REST API и Apache Kafka в контексте межсервисного взаимодействия банковских систем. Полученные результаты предоставляют объективную основу для выбора оптимальной технологии в зависимости от конкретных требований проекта. В ходе тестирования были выявлены значительные преимущества Kafka по показателям стабильности при высоких нагрузках (частота ошибок до 8% против 50% у REST), латентности (60-100 мс против 75-120 мс у REST) и пропускной способности (до 10000 против 5000 сообщений в секунду). Сформулированы конкретные рекомендации по применению каждой технологии, включая сценарии для гибридного подхода, объединяющего преимущества обоих решений.

Полученные результаты и разработанные решения могут быть использованы как основа для дальнейшего развития банковских приложений, требующих высокой производительности, гибкости и надёжности при обработке финансовых операций.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Документация «Apache Kafka Documentation» [Электронный ресурс] — https://kafka.apache.org/documentation/ — 2024 — Дата доступа 25.02.2025.
2. Документация «Confluent Kafka Documentation» [Электронный ресурс] — https://docs.confluent.io/kafka/introduction.html — 2024 — Дата доступа 25.02.2025.
3. Документация «OpenRewrite: Migrate to Spring Boot 3.3» [Электронный ресурс] — https://docs.openrewrite.org/recipes/java/spring/boot3/upgradespringboot_3_3 — 2025 — Дата доступа 20.02.2025.
4. Документация «Spring Boot Documentation» [Электронный ресурс] — https://docs.spring.io/spring-boot/docs/current/reference/html/ — 2024 — Дата доступа 20.02.2025.
5. Клеппман М. Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2022. — 640 с.
6. Amazon Web Services «Introducing support for Apache Kafka on Raft mode (KRaft) with Amazon MSK clusters» [Электронный ресурс] — https://aws.amazon.com/blogs/big-data/introducing-support-for-apache-kafka-on-raft-mode-kraft-with-amazon-msk-clusters/ — 2024 — Дата доступа 15.03.2025.
7. Bejeck B. Kafka Streams in Action: Real-time apps and microservices with the Kafka Streams API. — Manning Publications, 2023. — 280 с.
8. Isaac T. «Building Your First Microservice System with Spring Boot» [Электронный ресурс] — https://dev.to/isaactony/building-your-first-microservice-system-with-spring-boot-a-beginners-guide-3b28 — 2024 — Дата доступа 01.03.2025.
9. Narkhede N., Shapira G., Palino T. Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale. — O'Reilly Media, 2023. — 322 с.
10. NetApp «Making Apache Kafka Even Faster and More Scalable» [Электронный ресурс] — https://apachecon.com/acna2024/slides/01_Making_Apache_Kafka_even_faster_and_more_scalable.pdf — 2024 — Дата доступа 02.03.2025.
11. Newman S. Building Microservices: Designing Fine-Grained Systems. — O'Reilly Media, 2024. — 612 с.
12. Redpanda «Kafka performance tuning—Strategies and practical tips» [Электронный ресурс] — https://www.redpanda.com/guides/kafka-performance-kafka-performance-tuning — 2024 — Дата доступа 18.03.2025.

ПРИЛОЖЕНИЕ А



ПРИЛОЖЕНИЕ Б

ЛИСТИНГ ФАЙЛА DOCKER-COMPOSE.YML

```
version: '3'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - "2181:2181"

  prometheus:
    image: prom/prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml

  grafana:
    image: grafana/grafana
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
    depends_on:
      - prometheus

  kafka:
    image: confluentinc/cp-kafka:latest
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
```

```
    KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://kafka:29092,PLAINTEXT_HOST://localhost:9092
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
    KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

ЛИСТИНГ ФАЙЛА ACCOUNTSERVICE.JAVA

```
@Service
@RequiredArgsConstructor
@Slf4j
public class AccountService {

   private final AccountRepository accountRepository;

   public List<Account> getAllAccounts() {
      return accountRepository.findAll();
   }

   public Optional<Account> getAccountById(Long id) {
      return accountRepository.findById(id);
   }

   public Optional<Account> getAccountByNumber(String accountNumber) {
      return accountRepository.findByAccountNumber(accountNumber);
   }

   public Account createAccount(Account account) {
      return accountRepository.save(account);
   }

   @Transactional
   public void processTransaction(TransactionEvent event) {
      log.info("Processing transaction: {}", event);

      Optional<Account>                    accountOpt                    =
accountRepository.findByAccountNumber(event.getAccountNumber());
```

```
        if (accountOpt.isPresent()) {
            Account account = accountOpt.get();

            if (event.getType() == TransactionType.DEPOSIT) {
                account.setBalance(account.getBalance().add(event.getAmount()));
                accountRepository.save(account);
                log.info("Account {} balance updated after deposit",
account.getAccountNumber());
            } else if (event.getType() == TransactionType.WITHDRAWAL) {
                if (account.getBalance().compareTo(event.getAmount()) >= 0) {

account.setBalance(account.getBalance().subtract(event.getAmount()));
                    accountRepository.save(account);
                    log.info("Account {} balance updated after withdrawal",
account.getAccountNumber());
                } else {
                    log.error("Insufficient funds in account {}",
account.getAccountNumber());
                }
            }
        } else {
            log.error("Account not found: {}", event.getAccountNumber());
        }
    }
}
```

ЛИСТИНГ ФАЙЛА ACCOUNT.JAVA

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Account {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String accountNumber;
    private String ownerName;
    private BigDecimal balance;
```

```
}
```

ЛИСТИНГ ФАЙЛА TRANSACTIONEVENTCONSUMER.JAVA

```
@Component
@RequiredArgsConstructor
@Slf4j
public class TransactionEventConsumer {

   private final AccountService accountService;
   private final ObjectMapper objectMapper;

   @KafkaListener(topics = "transaction-events", groupId = "account-service-group")
   public void consume(String message) {
      try {
         TransactionEvent event = objectMapper.readValue(message, TransactionEvent.class);
         log.info("Received transaction event: {}", event);
         accountService.processTransaction(event);
      } catch (Exception e) {
         log.error("Error processing transaction event", e);
      }
   }
}
```

ЛИСТИНГ ФАЙЛА ACCOUNTCONTROLLER.JAVA

```
@RestController
@RequestMapping("/api/accounts")
@RequiredArgsConstructor
public class AccountController {

   private final AccountService accountService;

   @GetMapping
   public List<Account> getAllAccounts() {
      return accountService.getAllAccounts();
   }
```

```
    @GetMapping("/{id}")
    public ResponseEntity<Account> getAccountById(@PathVariable Long id) {
        return accountService.getAccountById(id)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
    }

    @GetMapping("/number/{accountNumber}")
    public ResponseEntity<Account> getAccountByNumber(@PathVariable String
accountNumber) {
        return accountService.getAccountByNumber(accountNumber)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
    }

    @PostMapping
    public Account createAccount(@RequestBody Account account) {
        return accountService.createAccount(account);
    }
}
```

ЛИСТИНГ ФАЙЛА TRANSACTIONSERVICE.JAVA

```
@Service
@RequiredArgsConstructor
@Slf4j
public class TransactionService {

    private final TransactionRepository transactionRepository;
    private final TransactionEventProducer eventProducer;

    public List<Transaction> getAllTransactions() {
        return transactionRepository.findAll();
    }

    public List<Transaction> getTransactionsByAccountNumber(String
accountNumber) {
        return transactionRepository.findByAccountNumber(accountNumber);
```

```
    }

    public Optional<Transaction> getTransactionById(Long id) {
        return transactionRepository.findById(id);
    }

    @Transactional
    public Transaction createDeposit(String accountNumber, BigDecimal amount) {
        // Создаем и сохраняем транзакцию
        Transaction transaction = new Transaction();
        transaction.setTransactionId(UUID.randomUUID().toString());
        transaction.setAccountNumber(accountNumber);
        transaction.setAmount(amount);
        transaction.setType(TransactionType.DEPOSIT);
        transaction.setStatus(TransactionStatus.PENDING);
        transaction.setTimestamp(LocalDateTime.now());

        Transaction savedTransaction = transactionRepository.save(transaction);

        // Создаем и отправляем событие
        TransactionEvent event = new TransactionEvent(
                savedTransaction.getTransactionId(),
                savedTransaction.getAccountNumber(),
                savedTransaction.getAmount(),
                TransactionType.DEPOSIT,
                TransactionStatus.PENDING
        );

        eventProducer.sendTransactionEvent(event);

        return savedTransaction;
    }

    @Transactional
    public Transaction createWithdrawal(String accountNumber, BigDecimal
amount) {
        // Создаем и сохраняем транзакцию
        Transaction transaction = new Transaction();
        transaction.setTransactionId(UUID.randomUUID().toString());
```

```
        transaction.setAccountNumber(accountNumber);
        transaction.setAmount(amount);
        transaction.setType(TransactionType.WITHDRAWAL);
        transaction.setStatus(TransactionStatus.PENDING);
        transaction.setTimestamp(LocalDateTime.now());

        Transaction savedTransaction = transactionRepository.save(transaction);

        // Создаем и отправляем событие
        TransactionEvent event = new TransactionEvent(
                savedTransaction.getTransactionId(),
                savedTransaction.getAccountNumber(),
                savedTransaction.getAmount(),
                TransactionType.WITHDRAWAL,
                TransactionStatus.PENDING
        );

        eventProducer.sendTransactionEvent(event);

        return savedTransaction;
    }
}
```

ЛИСТИНГ ФАЙЛА TRANSACTION.JAVA

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Transaction {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String transactionId;
    private String accountNumber;
    private BigDecimal amount;

    @Enumerated(EnumType.STRING)
```

```
    private TransactionType type;

    @Enumerated(EnumType.STRING)
    private TransactionStatus status;

    private LocalDateTime timestamp;
}
```

ЛИСТИНГ ФАЙЛА TRANSACTIONEVENTPRODUCER.JAVA

```
@Component
@RequiredArgsConstructor
@Slf4j
public class TransactionEventProducer {

    private final KafkaTemplate<String, String> kafkaTemplate;
    private final ObjectMapper objectMapper;

    public void sendTransactionEvent(TransactionEvent event) {
        try {
            String message = objectMapper.writeValueAsString(event);
            kafkaTemplate.send("transaction-events", message);
            log.info("Transaction event sent: {}", event);
        } catch (Exception e) {
            log.error("Error sending transaction event", e);
        }
    }
}
```

ЛИСТИНГ ФАЙЛА TRANSACIONCONTROLLER.JAVA

```
@RestController
@RequestMapping("/api/transactions")
@RequiredArgsConstructor
public class TransactionController {

    private final TransactionService transactionService;

    @GetMapping
```

```
    public List<Transaction> getAllTransactions() {
        return transactionService.getAllTransactions();
    }

    @GetMapping("/{id}")
    public ResponseEntity<Transaction> getTransactionById(@PathVariable Long
id) {
        return transactionService.getTransactionById(id)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
    }

    @GetMapping("/account/{accountNumber}")
    public List<Transaction> getTransactionsByAccountNumber(@PathVariable
String accountNumber) {
        return
transactionService.getTransactionsByAccountNumber(accountNumber);
    }

    @PostMapping("/deposit")
    public Transaction deposit(@RequestParam String accountNumber,
@RequestParam BigDecimal amount) {
        return transactionService.createDeposit(accountNumber, amount);
    }

    @PostMapping("/withdraw")
    public Transaction withdraw(@RequestParam String accountNumber,
@RequestParam BigDecimal amount) {
        return transactionService.createWithdrawal(accountNumber, amount);
    }
}
```

ЛИСТИНГ ФАЙЛА TRANSACIONEVENTPRODUCER.JAVA

```
@Component
@RequiredArgsConstructor
@Slf4j
public class TransactionEventRestClient {
    private final RestTemplate restTemplate;
```

```
    private final String accountServiceUrl; // Через @Value из конфигурации

    public void sendTransactionEvent(TransactionEvent event) {
        try {
            long startTime = System.currentTimeMillis();
            restTemplate.postForEntity(accountServiceUrl + "/api/transaction-events",
event, Void.class);
            long endTime = System.currentTimeMillis();
            log.info("REST transaction sent in {} ms: {}", (endTime-startTime), event);
        } catch (Exception e) {
            log.error("Error sending transaction event via REST", e);
        }
    }
}
```

ЛИСТИНГ ФАЙЛА TRANSACIONEVENTRESTCONTROLLER.JAVA

```
@RestController
@RequestMapping("/api/transaction-events")
@RequiredArgsConstructor
@Slf4j
public class TransactionEventRestController {
    private final AccountService accountService;

    @PostMapping
    public ResponseEntity<Void> processTransactionEvent(@RequestBody
TransactionEvent event) {
        long startTime = System.currentTimeMillis();
        try {
            log.info("Received transaction event via REST: {}", event);
            accountService.processTransaction(event);
            long endTime = System.currentTimeMillis();
            log.info("REST transaction processed in {} ms", (endTime-startTime));
            return ResponseEntity.ok().build();
        } catch (Exception e) {
            log.error("Error processing transaction event", e);
            return ResponseEntity.internalServerError().build();
        }
```

```
    }
}
```

ЛИСТИНГ ФАЙЛА PERFORMANCEMETRICS.JAVA

```
@Component
@RequiredArgsConstructor
public class PerformanceMetrics {
    private final MeterRegistry meterRegistry;

    public void recordLatency(String type, long latencyMs) {
        meterRegistry.timer("communication." + type + ".latency").record(latencyMs,
TimeUnit.MILLISECONDS);
    }

    public void incrementThroughput(String type) {
        meterRegistry.counter("communication." + type + ".throughput").increment();
    }
}
```

ЛИСТИНГ ФАЙЛА LOADTESTRUNNER.JAVA

```
@Component
@RequiredArgsConstructor
public class LoadTestRunner {
    private final TransactionService transactionService;
    private final ExecutorService executorService =
Executors.newFixedThreadPool(20);

    public void runLoadTest(int requestsPerSecond, int durationSeconds) {
        log.info("Starting load test: {} requests per second for {} seconds",
                requestsPerSecond, durationSeconds);

        long endTime = System.currentTimeMillis() + (durationSeconds * 1000);
        AtomicInteger counter = new AtomicInteger(0);

        while (System.currentTimeMillis() < endTime) {
            for (int i = 0; i < requestsPerSecond / 10; i++) {
                executorService.submit(() -> {
```

```
                String accountNumber = "ACC" + (counter.incrementAndGet() %
100);
                BigDecimal amount = BigDecimal.valueOf(Math.random() * 1000);
                transactionService.createDeposit(accountNumber, amount);
            });
        }

        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            break;
        }
    }
    log.info("Load test completed");
  }
}
```

ЛИСТИНГ ФАЙЛА LOADTESTCONTROLLER.JAVA

```
@RestController
@RequestMapping("/api/performance-test")
@RequiredArgsConstructor
public class LoadTestController {
  private final LoadTestRunner testRunner;

  @PostMapping
  public ResponseEntity<String> startTest(
      @RequestParam(defaultValue = "100") int requestsPerSecond,
      @RequestParam(defaultValue = "60") int durationSeconds) {

    CompletableFuture.runAsync(() ->
      testRunner.runLoadTest(requestsPerSecond, durationSeconds));

    return ResponseEntity.accepted()
      .body("Test started with " + requestsPerSecond + " requests/sec for " +
          durationSeconds + " seconds");
  }
}
```

ЛИСТИНГ ФАЙЛА TRASACTIONEVENT.JAVA (ACCOUNT-SERVICE)

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class TransactionEvent {
   private String transactionId;
   private String accountNumber;
   private BigDecimal amount;
   private TransactionType type;
   private TransactionStatus status;
}
public enum TransactionType {
   DEPOSIT, WITHDRAWAL, TRANSFER
}
public enum TransactionStatus {
   PENDING, COMPLETED, FAILED
}
```

ЛИСТИНГ ФАЙЛА APPLICATION.PROPERTIES (ACCOUNT-SERVICE)

```
spring.application.name=account-service

# server
server.port=8081

# db
spring.datasource.url=jdbc:h2:mem:accountdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true

# Kafka
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=account-service-group
```

```
spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.key-
deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-
deserializer=org.apache.kafka.common.serialization.StringDeserializer
```

ЛИСТИНГ ФАЙЛА APPLICATION.PROPERTIES (TRANSACTION-SERVICE)

```
spring.application.name=transaction-service

# server
server.port=8082

# db
spring.datasource.url=jdbc:h2:mem:transactiondb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true

# Kafka
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.producer.key-
serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-
serializer=org.apache.kafka.common.serialization.StringSerializer
```

ЛИСТИНГ ФАЙЛА TRASACTIONEVENT.JAVA (TRANSACTION-SERVICE)

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class TransactionEvent {
    private String transactionId;
    private String accountNumber;
    private BigDecimal amount;
```

```
    private TransactionType type;
    private TransactionStatus status;
}
```