МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра многопроцессорных систем и сетей

САВРИЦКИЙ Игорь Александрович

РАЗРАБОТКА МИКРОСЕРВИСА ДЛЯ ОБРАБОТКИ БАНКОВСКИХ ТРАНЗАКЦИЙ

Дипломная работа

Научный руководитель:
старший преподаватель
О.М. Кондратьева

Допущена к защите
«___»_____2025 г.
Зав. кафедрой многопроцессорных систем и сетей
кандидат физ.-мат. наук, доцент И.Е.Андрушкевич

Минск, 2025

РЕФЕРАТ

Дипломная работа, 50 страниц, 9 рисунков, 8 таблиц, 13 источников.
МИКРОСЕРВИСНАЯ АРХИТЕКТУРА, API GATEWAY, KEYCLOAK, НАГРУЗОЧНОЕ ТЕСТИРОВАНИЕ, SPRING WEBFLUX, БРОКЕР СООБЩЕНИЙ, АУТЕНТИФИКАЦИЯ И АВТОРИЗАЦИЯ

Объект исследования: распределённая микросервисная система, компоненты обеспечения безопасности и маршрутизации.

Цель работы: разработать и протестировать распределённую систему с микросервисной архитектурой, реализующую надёжную и масштабируемую инфраструктуру для обеспечения финансовых транзакций.

Методы исследования: архитектурное проектирование, написание программного кода, нагрузочное тестирование, визуализация результатов, сравнительный анализ.

Результаты: реализована микросервисная система для обеспечения банковских транзакций, проведено нагрузочное тестирование разработанной системы, внедерены современные технологические решения разработки приложений, предложены варианты улучшения разработанной системы.

РЭФЕРАТ

Дыпломная работа, 50 старонакі, 9 малюнкаў, 8 табліц, 13 крыніц.

МІКРАСЭРВІСНАЯ АРХІТЭКТУРА, API GATEWAY, KEYCLOAK, НАГРУЗАЧНАЕ ТЭСЦІРАВАННЕ, SPRING WEBFLUX, БРОКЕР ПАВЕДАМЛЕННЯЎ, АЎТЭНТЫФІКАЦЫЯ І АЎТАРЫЗАЦЫЯ

Аб'ект даследавання: размеркаваная мікрасэрвісная сістэма, кампаненты забеспячэння бяспекі і маршрутызацыі.

Мэта працы: распрацаваць і пратэставаць размеркаваную сістэму з мікрасэрвіснай архітэктурай, якая рэалізуе надзейную і маштабуемую інфраструктуру для забеспячэння фінансавых транзакцый.

Метады даследавання: архітэктурнае праектаванне, напісанне праграмнага кода, нагрузачнае тэставанне, візуалізацыя вынікаў, параўнальны аналіз.

Вынікі: рэалізаваная мікрасэрвісная сістэма для забеспячэння банкаўскіх транзакцый, праведзена нагрузачнае тэставанне распрацаванай сістэмы, укаранёны сучасныя тэхналагічныя рашэнні для распрацоўкі прыкладанняў, прапанаваны варыянты паляпшэння створанай сістэмы.

ABSTRACT

Thesis, 50 pages, 9 figures, 8 tables, 13 references.

MICROSERVICE ARCHITECTURE, API GATEWAY, KEYCLOAK, LOAD TESTING, SPRING WEBFLUX, MESSAGE BROKER, AUTHENTICATION AND AUTHORIZATION

Object of study: distributed microservice system, components for security and routing.

Purpose of the work: to develop and test a distributed system based on microservice architecture that implements a reliable and scalable infrastructure for processing financial transactions.

Research methods: architectural design, software development, load testing, results visualization, comparative analysis.

Results: a microservice system for banking transactions was implemented, load testing of the developed system was conducted, modern application development technologies were integrated, and several improvements to the system were proposed.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Современный банковский сектор является одной из наиболее динамично развивающихся отраслей экономики. Современный этап развития банковского сектора характеризуется устойчивым ростом числа операций и усилением конкуренции. В этих условиях особую значимость приобретают вопросы, связанные с мошенничеством и защитой финансовых данных, что повышает требования к надежности, производительности и безопасности банковских систем.

Однако основной проблемой, с которой сталкиваются финансовые организации, является возрастающая нагрузка на ПО, вызванная увеличением числа пользователей банковских сервисов. Часть банковских учреждений продолжает эксплуатировать программное обеспечение с устаревшей монолитной архитектурой, которое не соответствует современным требованиям. Это приводит к возникновению задержек при обработке запросов клиентов и снижению общей эффективности работы системы.

В качестве одного из решений данной проблемы рассматривается переход к микросервисной архитектуре. Данный подход предполагает декомпозицию системы на независимые функциональные модули, каждый из которых отвечает за выполнение конкретной задачи.

Целью данной дипломной работы является создание микросервиса, предназначенного для обработки банковских транзакций. Для достижения цели предполагается изучение и применение различных концепций распределенных систем, инструментов для создания микросервисов, методики разработки на базе REST API (Representational State Transfer Application Programming Interface).

Работа направлена на изучение современных технологий и подходов для построения распределенных систем, а также последующего их применения в ходе разработки приложения, соответствующего современным требованиям безопасности и производительности, с возможностью дальнейшего внедрения в реальный банковский процессинг.

ГЛАВА 1 БАНКОВСКОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

1.1 Требования к программному обеспечению

Банковская система представляет собой сложную сеть элементов, включающую в себя сами финансовые учреждения, а также их техническую инфраструктуру, обеспечивающую обработку и передачу финансовых операций. Эволюция финансовых рынков и усложнение банковских операций создают устойчивую потребность в разработке специализированного программного обеспечения. Современные технологические проблемы, включая экспоненциальный рост транзакционной нагрузки, диктуют необходимость создания высокоэффективных программных платформ, сочетающих надежность, производительность и способность к масштабированию.

Динамика развития сектора, характеризующаяся появлением инновационных финансовых продуктов, ужесточением требований и усилением вопросов безопасности, вынуждает организации осуществлять постоянную модернизацию своего ПО. Подобные обновления позволяют не только соответствовать актуальным стандартам, но и обеспечивают эффективную обработку возрастающих массивов данных, снижая при этом операционные риски. Особое значение в данном контексте приобретают такие технологические решения, как микросервисные архитектуры, системы на основе искусственного интеллекта и распределенные реестры. К программным решениям для банковского сектора предъявляются исключительно высокие требования, обусловленные критической важностью обеспечения безопасности, стабильности и эффективности. С методологической точки зрения данные требования целесообразно классифицировать на функциональные и нефункциональные аспекты.

Функциональная составляющая предполагает реализацию базовых банковских операций, включая транзакционную обработку, управление клиентскими счетами и кредитными продуктами. Не менее важным является обеспечение аналитических возможностей и интеграционных механизмов, позволяющих взаимодействовать с внешними финансовыми системами и регуляторными органами. Особые требования предъявляются к поддержке мультивалютных операций и обеспечению работы в режиме реального времени.

Нефункциональные требования:

1. Безопасность:

– Защита данных клиентов и транзакций (шифрование, защита от взлома, соответствие стандартам, таким как PCI DSS и GDPR).

– Многоуровневая аутентификация (например, использование двухфакторной аутентификации).

– Постоянный мониторинг угроз и внедрение механизмов предотвращения мошенничества (Fraud Detection).

2. Надёжность:

– Обеспечение бесперебойной работы системы 24/7.

– Возможность восстановления данных после сбоя (аварийное восстановление и резервное копирование).

– Высокая доступность системы с минимальным временем простоя.

3. Масштабируемость:

– Возможность обработки растущего числа пользователей и операций без снижения производительности.

– Поддержка горизонтального и вертикального масштабирования.

4. Производительность:

– Быстрая обработка запросов, даже при пиковых нагрузках.

– Минимизация задержек при выполнении транзакций.

5. Удобство использования:

– Понятный интерфейс для сотрудников банка и клиентов.

– Адаптивный дизайн для работы на различных устройствах (ПК, смартфоны, планшеты).

6. Интеграция:

– Поддержка взаимодействия с существующими системами банка (CRM, ERP, платёжные шлюзы).

– Поддержка стандартных протоколов и API для упрощённой интеграции с внешними платформами.

7. Адаптивность:

– Возможность легко вносить изменения для соответствия новым нормативным требованиям или бизнес-процессам.

– Гибкость для добавления новых функциональных модулей.

Требования к программному обеспечению для банковских систем чрезвычайно обширны и разнообразны. Каждый из этих аспектов предъявляет свои уникальные требования, и их успешная реализация требует тщательной проработки ещё на стадии проектирования.

Одной из ключевых задач является обеспечение соответствия ПО нефункциональным требованиям. Достижение этих характеристик невозможно без выбора правильной архитектуры, которая станет основой для

реализации всех функций и обеспечения гибкости в дальнейшем развитии системы.

Неправильное проектирование архитектуры может привести к серьёзным последствиям, включая низкую производительность системы, сложности масштабирования, уязвимость к угрозам безопасности и высокую стоимость изменений.

Таким образом, успешная реализация банковского ПО начинается с выбора архитектуры, способной обеспечить покрытие всех требований.

1.2 Архитектуры банковских приложений

Архитектура банковских приложений играет ключевую роль в обеспечении надёжности, безопасности, масштабируемости и простоты обслуживания системы. Существуют различные подходы к архитектурному проектированию, которые применяются в зависимости от потребностей конкретной организации. Рассмотрим два самых популярных и широко используемых вида архитектур в банковской сфере, их преимущества и недостатки.

1.2.1 Монолитная архитектура

Монолитная архитектура представляет собой традиционный подход к созданию программного обеспечения, в котором все компоненты системы объединены в единое целое. В банковской сфере этот подход использовался в течение десятилетий, так как он обеспечивал стабильность и простоту в условиях ограниченных возможностей ранних вычислительных технологий.

Монолитные системы функционируют как единое приложение, включающее в себя все необходимые функциональные модули. Все модули используют одну централизованную базу данных, что облегчает доступ к данным и упрощает управление ими.

Основным преимуществом монолитной архитектуры является её простота. Разработка такого приложения требует меньших усилий на этапе проектирования, так как оно создаётся в виде единого приложения. Развертывание также не представляет сложности: все модули запускаются одновременно, в рамках одного исполняемого файла. Такая организация системы обеспечивает высокий уровень производительности и упрощает реализацию транзакционной целостности, так как все операции выполняются в едином пространстве без необходимости синхронизации данных между отдельными модулями.

Однако с увеличением масштабов и сложности системы монолитная архитектура начинает проявлять существенные ограничения. Основная проблема заключается в трудностях масштабирования - при необходимости увеличения производительности отдельного функционального модуля разработчики вынуждены масштабировать всю систему целиком, что приводит к нерациональному использованию ресурсов и значительному увеличению эксплуатационных расходов.

Другим существенным недостатком является снижение гибкости системы. Любые модификации в одном компоненте могут вызвать непредвиденные последствия в работе других модулей, что требует проведения комплексного тестирования и увеличивает вероятность возникновения ошибок. Кроме того, интеграция современных технологических решений или переход на новую платформу разработки сопряжены со значительными сложностями, поскольку изначально система создается с использованием единого технологического стека. Так же при большой нагрузке на систему, отказ одного модуля повлечет за собой соответственно отказ всей системы. Это особенно актуально для самых популярных банковских приложений, где малейшие сбои могут привести к значительным последствиям.

ПО с монолитной архитектурой все еще составляет большой процент в банковском секторе из-за проверенной временем надежности и простоте разработки. Так же, стоит отметить, что изменение архитектуры связано с высокими материальными тратами. Однако с усложнением логики и бизнес-процессов ограничения монолитных архитектур становятся все более очевидными. Поэтому организации вынуждены искать альтернативные подходы, способные обеспечить необходимую гибкость, которую требуют постоянно меняющейся и растущие ожидания клиентов.

1.2.2 Микросервисная архитектура

В отличие от традиционных монолитных систем, данный подход предполагает декомпозицию приложения на совокупность автономных, слабосвязанных сервисов, каждый из которых отвечает за определенную бизнес-функцию. Такая организация позволяет независимо разрабатывать, тестировать и масштабировать отдельные компоненты системы, что особенно ценно для финансовых организаций, работающих в условиях высоких требований к гибкости и отказоустойчивости.

Концептуальной основой микросервисной архитектуры выступает принцип модульности. В зависимости от выбранной схемы проектирования, каждый сервис может обладать собственной базой данных или использовать

разделяемое хранилище с ограниченным кругом компонентов. Взаимодействие между сервисами осуществляется через стандартизированные интерфейсы, такие как REST API или брокеры сообщений, что обеспечивает технологическую гибкость.

Ключевым достоинством рассматриваемого подхода является его эластичность. Для банковской сферы, где нагрузка подвержена существенным колебаниям в зависимости от временных факторов, возможность избирательного масштабирования отдельных сервисов приобретает особую значимость. Например, в периоды пиковой нагрузки может быть увеличено количество экземпляров сервиса обработки платежей, в то время как другие компоненты системы продолжают работать в штатном режиме. Подобная избирательность не только оптимизирует ресурсоемкость, но и существенно повышает экономическую эффективность.

Не менее важным аспектом является повышенная отказоустойчивость микросервисных систем. Локальный сбой отдельного компонента, например сервиса генерации отчетов, не приводит к полному прекращению работы системы - критически важные функции, такие как проведение транзакций или управление счетами, продолжают функционировать. Данное свойство значительно повышает общую надежность банковских приложений и уровень их доступности для конечных пользователей.

«Важным преимуществом микросервисов является то, что они позволяют организациям быстрее масштабироваться и внедрять инновации за счет разделения служб и предоставления командам возможности независимо работать над различными частями приложения» [13]. Разработчики могут работать над разными сервисами одновременно, используя разные языки программирования, фреймворки и базы данных. Это особенно полезно в крупных организациях, где множество команд может разрабатывать и поддерживать различные части системы. Новый функционал можно добавить только в нужный сервис, без риска затронуть работу остальных.

Тем не менее, переход на микросервисную архитектуру связан с определёнными сложностями. Управление распределённой системой требует более сложной инфраструктуры, такой как оркестрация контейнеров (например, Kubernetes), системы мониторинга и балансировки нагрузки при горизонтальном масштабировании. Также возникает необходимость в координации их работы при выполнении распределенных транзакций – операций, затрагивающих несколько сервисов одновременно. Например, обработка банковского перевода может включать уменьшение баланса на одном счёте, увеличение баланса на другом и запись этой операции в журнал. Эти действия выполняются разными микросервисами, поэтому возникает

риск, что часть операции будет завершена успешно, а часть – нет, что может привести к неконсистентному состоянию системы.

В целом, микросервисная архитектура становится стандартом для современных банковских систем, предоставляя необходимую гибкость, масштабируемость и надёжность. Однако её успешное внедрение требует тщательного планирования и построения надёжной инфраструктуры.

1.2.3 Сравнение архитектур

Сравнение по критериям монолитной и микросервисной архитектур отражено на таблице 1.1.

Таблица 1.1 – Сравнение монолитной и микросервисной архитектур

Критерий	Микросервисная архитектура	Монолитная архитектура
Масштабируемость	Горизонтальное масштабирование отдельных компонентов.	Масштабируется только целиком, что дороже и сложнее.
Независимость	Разработка и развертывание отдельных сервисов.	Любое изменение затрагивает всю систему.
Устойчивость	Отказ одного микросервиса не влияет на работу остальных.	Сбой в одной части может остановить всю систему.
Безопасность	Ограничение доступа к каждому сервису, меньший объём данных на каждом уровне.	Вся система уязвима, если защита где-то нарушена.
Скорость внедрения изменений	Быстрое добавление новых функций.	Медленное внедрение, тестирование всей системы.
Поддержка технологий	Возможность использования разных технологий для каждого микросервиса.	Одна технология для всей системы, сложность модернизации.
Проблемы	Усложнение разработки на начальных этапах. Требуется настройка сетевой инфраструктуры, что увеличивает время и стоимость. Сложность мониторинга и отладки работы множества сервисов.	Затруднена масштабируемость и поддержка крупной системы. Долгое время на развертывание и обновления. Трудности при добавлении новых функций или модернизации компонентов.

Таблица демонстрирует ключевые различия между микросервисной и монолитной архитектурами. Как было сказано ранее, современные банковские

системы имеют очень обширный ряд требований к ПО, которые монолитная архитектура покрыть не может. Поэтому в дипломной работе выбрана распределенная архитектура.

1.3 Подходы к построению микросервисных систем

Микросервисная архитектура предоставляет широкий спектр возможностей для гибкого и эффективного построения систем. Опыт в решениях рядовых проблем в микросервисной архитектуре привел к созданию паттернов, представляющих собой проверенные временем решения базовых задач.

1.3.1 Паттерн API-Gateway

API-Gateway представляет собой модуль, который отвечает за маршрутизацию всех клиентских запросов. «API-Gateway находится перед вашими микросервисами и выступает в роли единой точки входа для клиентов. Он может обрабатывать маршрутизацию запросов, компоновку ответов и даже преобразование протоколов» [11].

С точки зрения системной архитектуры, API Gateway обеспечивает значительные преимущества в нескольких ключевых аспектах:

Централизованное управление: шлюз API представляет собой точку входа, через которую клиенты получают доступ к микросервисам. Это обеспечивает простоту эксплуатации и согласованное сопоставление, поскольку траектория API и стратегии управления трафиком находятся в шлюзе, а следовательно, политики безопасности, управление тарифами и аутентификация становятся более управляемыми.

Повышенная безопасность: шлюзы API играют ключевую роль в обеспечении безопасности микросервисов, поскольку они действуют как защитный слой, который позволяет организациям централизованно применять такие меры безопасности, как аутентификация, авторизация и шифрование. Они могут выполнять такие задачи, как управление ключами API, проверка токенов OAuth и завершение SSL-соединения, что снижает нагрузку на безопасность отдельных микросервисов.

Агностичность в отношении протоколов: API можно использовать в качестве переводчика, и таким образом клиенты микросервисов могут взаимодействовать с другими системами,используя предпочитаемые ими протоколы. Такая гибкость является дополнительным преимуществом для пользователей, которое делает возможной интеграцию различных клиентских

приложений и серверных служб без необходимости поддерживать один и тот же протокол связи.

Балансировка нагрузки и масштабируемость: Балансировка нагрузки и масштабируемость: с помощью API-шлюзов входящие запросы балансируются по нагрузке между инстансами микросервисов для достижения высокой доступности и масштабируемости. Они могут свободно изменять правила перенаправления, такие как состояние сервера, объем запросов, географическое положение, для достижения максимальной производительности.

«Кэширование и оптимизация: шлюз API может использовать и кэшировать ответы от микросервисов, тем самым повышая производительность и сокращая задержки. Сохраняя наиболее часто запрашиваемые данные, они обеспечивают ответ, который не требует больших нагрузок на серверные системы и клиентские приложения. Кроме того, они позволяют оптимизировать требования, объединяя или преобразуя данные, поступающие от нескольких микросервисов, в один ответ, тем самым сокращая количество обращений клиентов и серверов» [6].

Как демонстрирует рисунок 1.1, API Gateway организует четкую структуру взаимодействий в распределенной системе. Широкое применение данного паттерна в современных микросервисных архитектурах подтверждает его важность как инструмента управления сложными взаимодействиями между клиентами и множеством микросервисов. В контексте банковских систем, где особенно важны безопасность, надежность и производительность, API Gateway становится практически незаменимым компонентом, обеспечивающим стабильную работу всей распределенной инфраструктуры.

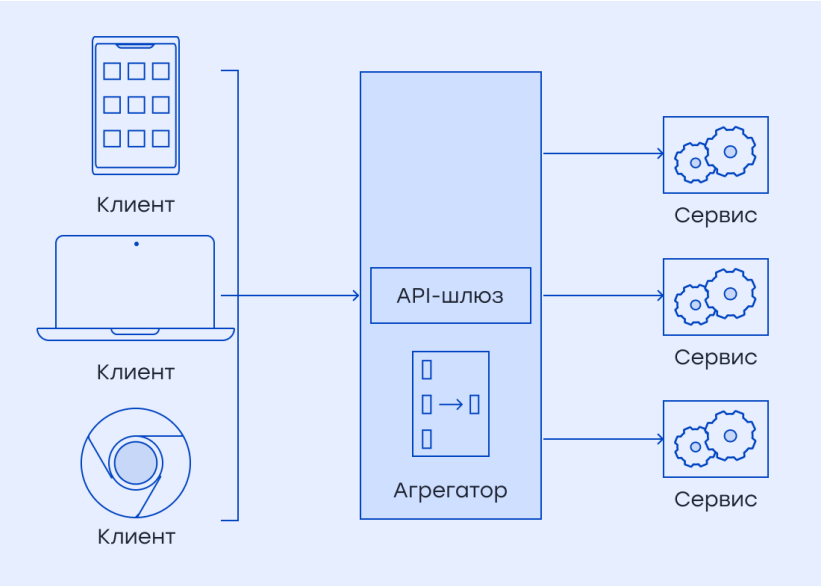

Рисунок 1.1 – Паттерн API-Gateway

1.3.2 Паттерн Saga

В контексте микросервисных архитектур проблема обеспечения согласованности данных приобретает особую сложность. Распределённые транзакции, под которыми понимается процесс синхронизации состояния между независимыми базами данных или сервисами в рамках единой логической операции, требуют специальных механизмов реализации. Традиционные подходы, такие как двухфазный коммит (2PC), оказываются не всегда применимыми в условиях распределённых систем из-за требований к доступности и производительности.

Паттерн Saga предлагает альтернативное решение данной проблемы, основанное на принципе eventual consistency (согласованности в конечном счёте). «Eventual consistency означает, что данные в разных сервисах могут временно быть несинхронизированными, но в конечном итоге достигнут согласованного состояния» [11].

Существует два подхода к реализации данного паттерна: оркестрированный и хореографированный. Оркестрированный использует централизованного оркестратора, который управляет выполнением и откатом шагов транзакции. Хореографированный предполагает то, что каждый сервис самостоятельно инициирует следующий шаг, реагируя на события, происходящие в системе. Изображение взаимосвязей, предлагаемых данным паттерномпредставлено на рисунке 1.2.

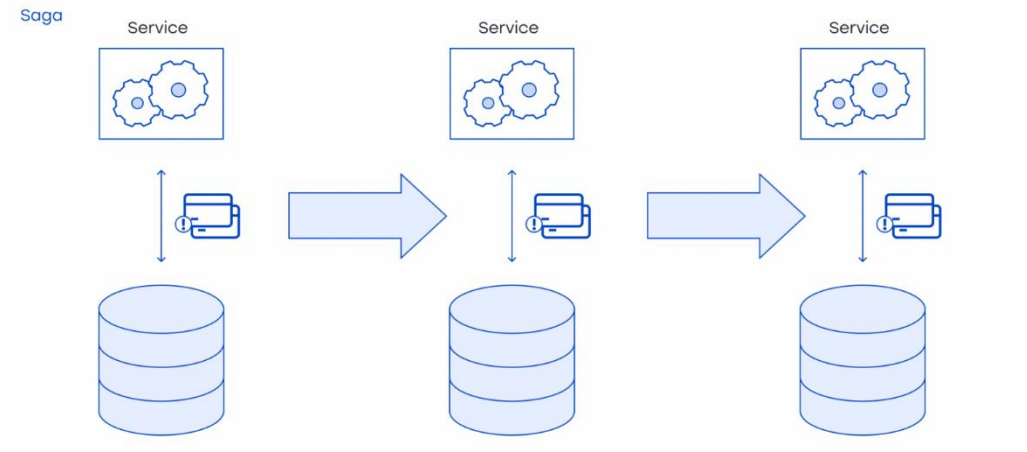

Рисунок 1.2 – Паттерн Saga

Распределённые транзакции являются сложным, но неотъемлемым компонентом микросервисной архитектуры, особенно в системах с высокими требованиями к согласованности данных, таких как банковские приложения. Поэтому правильный выбор подхода позволит обеспечить гибкость и устойчивость системы.

1.3.3 Паттерн Database per Service

Паттерн Database per Service предполагает, что каждый микросервис управляет своей собственной базой данных, изолированной от других сервисов. Этот паттерн организации данных в микросервисных системах предлагает ряд существенных преимуществ, определяющих его растущую популярность в современных распределённых системах. Важнейшим достоинством является предоставляемая сервисам полная автономия в управлении данными. Каждый микросервис получает возможность самостоятельного выбора наиболее подходящего типа базы данных: реляционная, документо-ориентированная или графовая СУБД.

«У паттерна есть и недостатки: усложняется обмен данными между сервисами и предоставление транзакционных гарантий ACID. Паттерн не стоит применять в небольших приложениях — он предназначен для крупномасштабных проектов с большим числом микросервисов, где каждой команде требуется полное владение ресурсами для повышения скорости разработки и лучшего масштабирования» [1].

ГЛАВА 2 ПЕРВОНАЧАЛЬНО ИСПОЛЬЗУЕМЫЕ ТЕХНОЛОГИИ

Разработка требует выбора подходящего технологического стека, который должен обеспечивать все требования к программному продукту. Следует отметить, что представленный технологический стек является первоначальным и может изменяться на последующих этапах разработки в зависимости от новых требований, ограничений или возникающих технических задач и проблем.

2.1 Язык программирования Java

Java – это строго типизированный объектно-ориентированный язык программирования общего назначения, изначально разработанный компанией Sun Microsystems, которая впоследствии была приобретена корпорацией Oracle. Развитие языка осуществляется сообществом через процесс Java Community Process, а сам язык и основные технологии, его реализующие, распространяются под лицензией GPL. Торговая марка Java находится в собственности Oracle.

Одной из ключевых особенностей Java является её кроссплатформенность: для запуска программ, написанных на этом языке, достаточно установить Java Virtual Machine (JVM). Эта особенность делает Java универсальным инструментом, подходящим для разработки корпоративных приложений и серверной части.

Строгая типизация Java представляет собой важное архитектурное преимущество, обеспечивающее надежную работу с разнообразными типами данных в сложных информационных системах. Эта характеристика языка приобретает особую значимость в корпоративной среде, где критически важны предсказуемость выполнения операций и минимизация ошибок времени выполнения. Кроссплатформенность Java, реализуемая через механизм виртуальной машины (JVM), существенно упрощает процессы миграции приложений между различными операционными средами, снижая потенциальные риски, связанные с переносом программного обеспечения.

Эволюционное развитие Java привело к созданию специализированной платформы Java EE (Enterprise Edition), ориентированной на потребности корпоративного сектора. Данная платформа расширяет базовые возможности языка, предлагая развитую инфраструктуру для создания распределенных систем. В частности, Servlet API как ключевой компонент Java EE обеспечивает полноценную поддержку веб-сервисов и работу с сетевыми

протоколами, включая защищенные соединения через HTTPS, что особенно важно для финансовых приложений.

Организация взаимодействия с СУБД в Java представлена интерфейсом JDBC (Java Database Connectivity). Этот механизм предоставляет единый API для выполнения запросов и управлением транзакциями. JDBC основан на стандарте ODBC и обеспечивает высокую совместимость с различными базами данных.

2.2 Платформа Spring Framework

«Spring Framework обеспечивает решения многих задач, с которыми сталкиваются Java-разработчики и организации, которые хотят создать информационную систему, основанную на платформе Java. Этот фреймворк предлагает последовательную модель и делает её применимой к большинству типов приложений, которые уже созданы на основе платформы Java. Считается, что Spring реализует модель разработки, основанную на лучших стандартах индустрии, и делает её доступной во многих областях Java» [5].

Архитектурной основой Spring является его модульная организация, позволяющая разработчикам использовать только необходимые компоненты. Основой является Spring Core, реализующий принцип инверсии управления (Inversion of Control). Такой подход способствует созданию слабосвязанных систем с высокой степенью заменяемости компонентов, что существенно повышает гибкость и тестируемость приложений.

Для веб-разработки фреймворк предлагает модуль Spring MVC, реализующий популярный шаблон Model-View-Controller. Его ключевой особенностью является использование аннотаций, которые значительно сокращают объем стандартного кода и позволяют сосредоточиться на реализации бизнес-логики. При работе с данными Spring Data предоставляет унифицированный интерфейс для различных хранилищ, включая как реляционные (через JPA), так и NoSQL-решения.

Вопросы безопасности эффективно решаются средствами Spring Security, предлагающего готовые механизмы аутентификации и авторизации с поддержкой современных стандартов. Аспектно-ориентированное программирование реализовано в Spring AOP, позволяющем выносить сквозную функциональность в отдельные модули.

Особого внимания заслуживает Spring Boot, значительно упрощающий процесс настройки приложений за счет конвенционального подхода и автоматической конфигурации. Для разработки распределенных систем Spring Cloud предлагает специализированные инструменты, включая

централизованное управление конфигурациями (Spring Cloud Config), механизм обнаружения сервисов (Eureka), API-шлюзы и средства распределенной трассировки.

Такая комплексная экосистема делает Spring Framework предпочтительным выбором для создания современных корпоративных приложений, особенно в условиях перехода к микросервисной архитектуре. Сочетание богатого функционала с продуманной архитектурой позволяет разработчикам эффективно решать широкий спектр задач, от базовой бизнес-логики до сложных распределенных систем.

2.3 Средство контейнеризации Docker

Docker представляет собой современную открытую платформу, предназначенную для разработки, распространения и выполнения приложений в изолированных средах, называемых контейнерами. В основе данной технологии лежит концепция контейнеризации, которая позволяет инкапсулировать приложение вместе со всеми его зависимостями в автономный исполняемый пакет. Важной характеристикой Docker-контейнеров является их способность одинаково функционировать на различных платформах - от локальных рабочих станций разработчиков до облачных инфраструктур, включая такие популярные решения как AWS, Google Cloud Platform и Microsoft Azure.

В Docker все строится из образов. Образ представляет собой шаблон, содержащий всё необходимое для запуска приложения: операционную систему, библиотеки, зависимости и саму программу. Образы создаются на основе файла конфигурации Dockerfile, где описываются все шаги для сборки окружения. После создания образа его можно использовать многократно для запуска контейнеров. Контейнеры - это изолированные экземпляры, созданные на основе образов. Они потребляют меньше ресурсов, чем виртуальные машины. Благодаря этому контейнеры запускаются быстрее и более экономичны.

Docker отлично подходит для микросервисов, поскольку позволяет каждой службе работать в отдельном контейнере со своими собственными зависимостями, не мешая работе других служб. «Контейнеры имеют малый вес и могут быть быстро запущены и остановлены, что обеспечивает ускоренный цикл разработки, типичный для архитектур микросервисов» [10].

Инструмент Docker Compose позволит запускать несколько контейнеров одновременно. Для запуска Docker Compose нужно создать файл docker-compose.yml и нем определить все образы и их взаимосвязи.

Для работы с масштабируемыми распределенными системами Docker предлагает решение в виде Docker Swarm. Этот инструмент обеспечивает возможность создания кластеров контейнеров с автоматической балансировкой нагрузки, что особенно важно для высокодоступных сервисов. Однако в профессиональной среде все чаще предпочтение отдается Kubernetes - более функциональной системе оркестрации, которая, сохраняя совместимость с Docker, предоставляет расширенные возможности управления контейнеризованными приложениями.

Современная практика разработки программного обеспечения, особенно в области микросервисных архитектур, демонстрирует возрастающую роль Docker как фундаментального технологического решения. Его адаптивность к различным сценариям использования, от локальной разработки до промышленного развертывания, в сочетании с богатой экосистемой сопутствующих инструментов, делает Docker незаменимым компонентом среди инструментов разработки.

2.4 Спецификация JPA и Hibernate

В современной Java-экосистеме особое место занимают технологии работы с реляционными базами данных, среди которых ключевую роль играет Java Persistence API (JPA) и его наиболее популярная реализация - Hibernate. Эти технологии представляют собой мощный инструментарий для объектно-реляционного отображения (ORM), значительно упрощающий взаимодействие приложений с системами управления базами данных.

JPA, будучи стандартизированной спецификацией, устанавливает единые подходы к персистентности данных в Java-приложениях. Его основное преимущество заключается в возможности работы с данными на уровне объектной модели, что существенно повышает продуктивность разработки. Вместо написания сложных SQL-запросов разработчики могут использовать JPQL - специализированный язык запросов, оперирующий сущностями предметной области, а не таблицами базы данных. Такой подход делает код более понятным и поддерживаемым.

Hibernate как наиболее зрелая реализация JPA предлагает расширенный функционал, выходящий за рамки стандартной спецификации. Его отличительной чертой является богатый набор возможностей для маппинга объектных моделей на реляционные структуры, включая поддержку различных стратегий наследования и ассоциаций между сущностями. Особого внимания заслуживают механизмы кэширования и автоматической генерации

схемы базы данных, которые значительно ускоряют процесс разработки, особенно на начальных этапах проекта.

2.5 Базы данных Postgre SQL и Redis

PostgreSQL – это объектно-реляционная система управления базами данных. Она написана на языке C и распространяется свободно. PostgreSQL базируется на языке SQL и поддерживает многие из возможностей стандарта SQL:2011 и ряд возможностей SQL:2016 в части работы с данными в формате JSON.

Рассмотрим ключевые особенности PostgreSQL.

«*Масштабируемость*. PostgreSQL позволяет распределить входящие запросы между несколькими инстансами – экземплярами (репликами) БД. Также зачастую на облачных решениях с предустановленным PostgreSQL можно в любой момент добавить дополнительные ресурсы» [2].

Кроссплатформенность. СУБД PostgreSQL можно установить практически на любую известную операционную систему. В число поддерживаемых ОС входят дистрибутивы Linux (Ubuntu, Debian, CentOS и другие), macOS, Windows 10 и 11.

«*Безопасность*. PostgreSQL позволяет шифровать данные на разных уровнях, а также выбирать средства защиты информации от несанкционированного доступа. В состав PostgreSQL входит большое количество инструментов безопасности: например, LDAP, SSPI, Kerberos, GSSAPI и многие другие. Эти технологии можно использовать для создания дополнительных опций безопасности, а также регулировать настройки доступа к объектам базы данных на разных уровнях: от самой БД до столбцов в ней» [2].

Поддержка различных типов данных. PostgreSQL может работать не только со стандартными типами данных, которые существуют в большинстве популярных языков, но и со специфическими типами. Среди них можно встретить геометрические расчеты, сетевые адреса, типы полнотекстового поиска и многие другие.

Расширяемость. Вы можете написать собственные функции для Постгрес на Python, PHP, Java и Ruby. Также для загрузки доступны готовые модули на языке C из репозитория PGXN.

Открытый код. PostgreSQL распространяется по лицензии BSD. Это значит, что проект можно использовать, редактировать и распространять среди других пользователей бесплатно. Также разработчики могут модифицировать оригинальный код, скачав его с официального сайта.

Активное сообщество. Пользователи и разработчики собственного ПО еженедельно получают новости о PostgreSQL из рассылок. Также участники сообщества делятся опытом, ведут дискуссии и помогают в решении вопросов разработки.

«*Поддержка NoSQL*. PostgreSQL поддерживает форматы XML, JSON и JSONB. Это позволяет записать JSON-документ в базу данных, не разбирая его — от этого производительность сервера БД становится выше» [2].

Также PostgreSQL имеет два недостатка.

«*Требовательность к ресурсам*. Так как PostgreSQL способна работать со сложными запросами и большими объемами данных, она может требовать больше ресурсов в сравнении с другими популярными СУБД. В моменты высокой нагрузки особенно активно расходуется оперативная память (ресурсы ОЗУ) и процессорное время» [2].

«*Сложная настройка*. Так как СУБД обладает гибким функционалом для настройки, для работы с ней потребуется знание архитектуры и понимание параметров. Кроме того, необходимо освоить основы языка SQL. Все это может создать сложности в PostgreSQL для начинающих пользователей» [2].

Теперь рассмотрим Redis. Redis хранит данные в оперативной памяти. Такая память работает быстрее, чем системы, которые используют жёсткий диск. Ещё одна особенность Redis – это модель, по которой она хранит данные, – «ключ-значение». Другие СУБД, в отличие от неё, используют для этого двумерные таблицы.

С помощью модели «ключ-значение» можно быстро искать информацию в базе. Поэтому Redis часто используют для задач, где важна высокая скорость: например, при публикации сообщений или кэшировании. СУБД Redis увеличивает производительность веб-приложений. Она помогает сократить время отклика и снизить нагрузку на серверы. Его ключевое преимущество заключается в экстремально низкой задержке при выполнении операций, что делает его незаменимым для сценариев, требующих мгновенного доступа к информации. Поддержка разнообразных структур данных позволяет эффективно решать такие задачи, как управление сессиями, реализация очередей сообщений и кэширование.

2.6 Брокер сообщений RabbitMQ

RabbitMQ является брокером сообщений, реализующим протокол AMQP (Advanced Message Queuing Protocol). AMQP – это протокол передачи сообщений, разработанный для обеспечения надежного, безопасного и совместимого обмена данными между различными системами. Его основная

цель – стандартизация взаимодействия между отправителями и получателями сообщений, независимо от их реализации, что делает AMQP универсальным выбором.

Основная архитектура AMQP базируется на понятиях обменов (exchanges), очередей (queues) и сообщений (messages). Сообщения передаются от отправителя через обмен, который определяет их маршрут в одну или несколько очередей. Получатели подключаются к очередям для обработки сообщений. Такая модель разделяет отправителей и получателей, устраняя их прямую зависимость и упрощая разработку и обслуживание системы.

Модель маршрутизации сообщений в AMQP предоставляет гибкость для различных сценариев. Например, сообщения могут быть доставлены одному конкретному получателю (точка-точка), нескольким получателям (мультитрансляция), либо распределены между получателями на основе их содержания (фильтрация).

«Брокеры сообщений могут использоваться для шифрования и аутентификации сообщений, что позволяет защитить данные, передаваемые между сервисами» [3]. Схема работы брокера сообщений RabbitMQ представлена на рисунке 2.1.

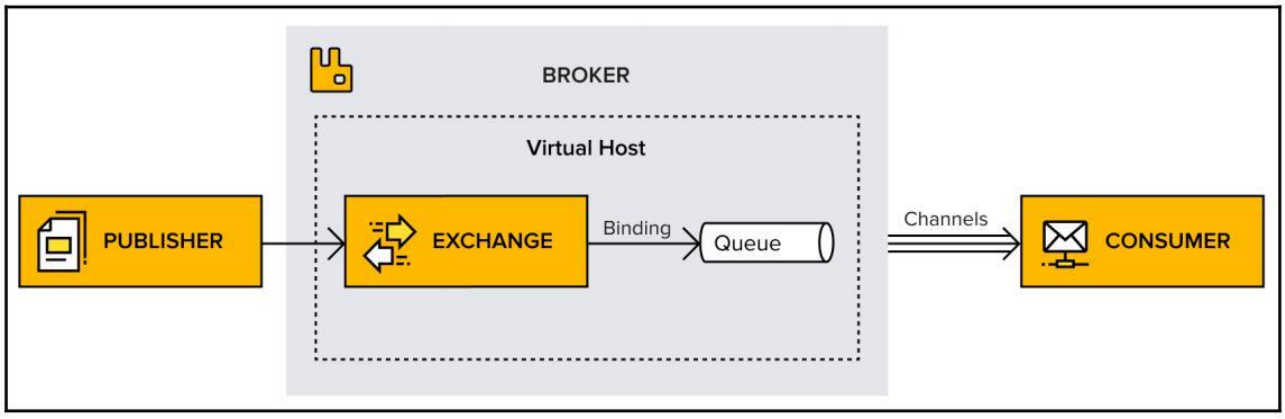

Рисунок 2.1 - Схема работы RabbitMQ

Брокер сообщений управляет маршрутизацией и доставкой данных между системами. Издатель инициирует передачу данных, направляя сообщение в специализированный компонент - обменник. На этом этапе происходит критически важный процесс маршрутизации, где на основании предустановленных правил привязки сообщение распределяется в соответствующие очереди назначения. Очереди выполняют функцию временного буфера хранения. Сообщения могут сохраняться либо в высокоскоростной оперативной памяти для обеспечения минимальных задержек, либо на устойчивом дисковом хранилище для гарантии сохранности

при возможных сбоях. Когда потребитель заявляет о своей готовности к обработке, система инициирует процесс доставки. Важно отметить, что брокер не просто передает оригинал сообщения, а создает его копию, сохраняя таким образом исходные данные до момента подтверждения успешной обработки. После получения подтверждения от потребителя система последовательно удаляет сообщение из очереди и окончательно освобождает соответствующие ресурсы хранения.

ГЛАВА 3 ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ПРИЛОЖЕНИЯ

3.1 Начальный этап проектирования системы

Проведя комплексный анализ требований, предъявляемых к современным банковским приложениям, и оценив существующие технологические решения, было принято решение о реализации веб-сервиса с использованием REST-архитектуры. Такой выбор обусловлен необходимостью обеспечения масштабируемости, простоты интеграции и соответствия современным стандартам разработки распределенных систем.

В качестве основного языка программирования был выбран Java, чьи характеристики идеально соответствуют требованиям банковской сферы. Объектно-ориентированная природа Java в сочетании со строгой типизацией обеспечивает надежность и поддерживаемость кодовой базы. Обширная стандартная библиотека и поддержка современных парадигм программирования позволяют эффективно решать сложные задачи, характерные для финансовых систем. Для ускорения процесса разработки и упрощения создания микросервисов выбор пал на фреймворк Spring Boot, который благодаря своей модульной структуре и богатой экосистеме значительно сокращает время вывода продукта на рынок.

Для хранения структурированных данных выбрана СУБД PostgreSQL. Временные данные будут храниться в Redis. Это обеспечит высокую производительность системы аутентификации и быстрый доступ к часто используемым данным.

Взаимодействие между микросервисами организуется брокером сообщений, а их конфигурация реализуется с использованием Spring Cloud, предоставляющего комплексные инструменты для построения распределенных систем. Данный фреймворк значительно упрощает процессы конфигурации, обнаружения сервисов и обеспечения отказоустойчивости, что критически важно для банковских приложений, работающих в режиме 24/7.

Вопросы безопасности решаются через внедрение механизма JWT-токенов, который обеспечивает защищенную передачу аутентификационных данных между сервисами. Такой подход позволит реализовать распределенную систему авторизации, соответствующую современным стандартам безопасности в финансовой сфере.

3.1.1 Модульное проектирование системы

Дипломная работа сосредоточена на разработке ключевого микросервиса для обработки транзакций, однако для обеспечения его полноценного функционирования потребовалось проектирование всей системы в целом. Следует отметить, что вспомогательные модули, включая сервисы аутентификации и управления пользователями, реализованы с минимальным функционалом, достаточным для поддержки основной деятельности системы, но без детальной проработки, характерной для основного транзакционного компонента. Анализ требований банковской сферы привел к проектированию системы, состоящей из четырех взаимосвязанных сервисов, каждый из которых выполняет строго определенную бизнес-функцию.

AuthService отвечает за безопасности системы, осуществляя процессы аутентификации и авторизации пользователей через механизм JWT-токенов. В его обязанности входит централизованное хранение учетных данных: зашифрованные пароли и refresh-token.

AccountService специализируется на обработке персональных данных клиентов и управлении их банковскими аккаунтами. Этот сервис поддерживает основные CRUD-операции с информацией о пользователях, одновременно предоставляя интерфейс для доступа к финансовой истории и текущим балансам.

TransactionalService, отвечает за обработку всех видов финансовых операций. Его основная задача получать, проводить и записывать данные о транзакциях, интегрируясь с другими сервисами через механизм брокера сообщений.

FundTransferService должен выполнять валидацию денежных переводов, проверяя соблюдение лимитов и корректность реквизитов. После успешной проверки он инициирует процесс транзакции через систему очередей сообщений. А API-Gateway выступает в роли единой точки входа, обеспечивая маршрутизацию запросов, проверку авторизации и абстрагирование внутренней архитектуры от конечных пользователей.

Представленная архитектура демонстрирует четкое разделение ответственности между компонентами, обеспечивая при этом необходимый уровень масштабируемости и асинхронного взаимодействия. Последующие разделы работы будут посвящены проектированию схем баз данных, соответствующих функциональным требованиям каждого из сервисов.

3.1.2 Проектирование баз данных

Проектирование базы данных для банковского приложения требует особого внимания к фундаментальным аспектам информационной безопасности, производительности и целостности данных. Архитектура хранения информации должна обеспечивать не только эффективную обработку транзакций, но и поддерживать принципы микросервисной разработки, где каждый компонент системы обладает автономией в управлении своими данными. Ключевым принципом организации данных выступает строгая изоляция микросервисов через выделение отдельных хранилищ для каждого компонента системы. Такой подход минимизирует каскадные эффекты при внесении изменений и позволяет независимо масштабировать отдельные функциональные блоки. Изоляция данных создает естественные границы между сервисами.

Для хранения структурированных финансовых данных основным решением выбрана реляционная СУБД PostgreSQL. Ее выбор обусловлен строгой поддержкой ACID-транзакций, что критически важно для операций с денежными средствами. Богатый инструментарий для индексации и оптимизации запросов позволяет эффективно работать с большими объемами данных, сохраняя высокую производительность даже при сложных аналитических операциях. Временные данные вынесены в хранилище на основе Redis. Такой способ хранения данных позволяет достичь баланса между согласованностью постоянных и высокой доступностью временных данных.

Далее рассмотрим схемы данных, разработанные для каждого микросервиса.

Микросервис *AuthService* имеет подключение к двум базам данных: реляционной PostgreSQL и нереляционной Redis.

Схема реляционной базы данных PostgreSQL представлена в таблице 3.1.

Таблица 3.1 – База данных auth

Поле	Тип данных	Описание
id	SERIAL PRIMARY KEY	Уникальный идентификатор записи
login	VARCHAR(255) UNIQUE	Уникальный логин пользователя
password	VARCHAR(255)	Хеш пароля пользователя
created_at	TIMESTAMP	Дата и время создания записи
updated_at	TIMESTAMP	Дата и время последнего обновления

Нереляционная база данных Redis хранит JWT в формате «ключ-значение», где:

- Ключ: `refresh_token:{user_id}`.
- Значение: JSON, содержащий токен и время истечения.

Микросервис *AccountService* использует реляционную базу данных PostgreSQL. Схема данных представлена в таблице 3.2.

Таблица 3.2 – База данных accounts

Поле	Тип данных	Описание
id	SERIAL PRIMARY KEY	Универсальный идентификатор аккаунта
user_id	INTEGER UNIQUE	Идентификатор пользователя (связь с auth.id)
first_name	VARCHAR(255)	Имя пользователя
last_name	VARCHAR(255)	Фамилия пользователя
phone_number	VARCHAR(20) UNIQUE	Номер телефона
account_number	VARCHAR(20) UNIQUE	Номер счета
balance	NUMERIC(12,2)	Баланс счета
created_at	TIMESTAMP	Дата и время создания
updated_at	TIMESTAMP	Дата и время последнего обновления

Микросервис *TransactionalService* использует реляционную базу данных PostgreSQL. Схема данных представлена в таблице 3.3.

Таблица 3.3 – База данных transactions

Поле	Тип данных	Описание
id	SERIAL PRIMARY KEY	Универсальный идентификатор транзакции
from_account_id	INTEGER	Идентификатор счета отправителя (связь с accounts.id)
to_account_id	INTEGER	Идентификатор счета получателя (связь с accounts.id)
amount	NUMERIC(12,2)	Сумма транзакции
status	VARCHAR(20)	Статус транзакции
created_at	TIMESTAMP	Дата и время создания

Микросервис *FundTransferService* использует реляционную базу данных PostgreSQL. Схема данных представлена в таблице 3.4.

Таблица 3.4 – База данных fund_transfers

Поле	Тип данных	Описание
Id	SERIAL PRIMARY KEY	Универсальный идентификатор перевода
transaction_id	INTEGER	Ссылка на транзакцию (связь с transactions.id)
Description	TEXT	Описание перевода
Fee	NUMERIC(12,2)	Комиссия за перевод
initiated_at	TIMESTAMP	Время инициирования перевода

3.2 Взаимодействие модулей системы

Взаимодействие модулей системы, представлено на рисунке 3.1.

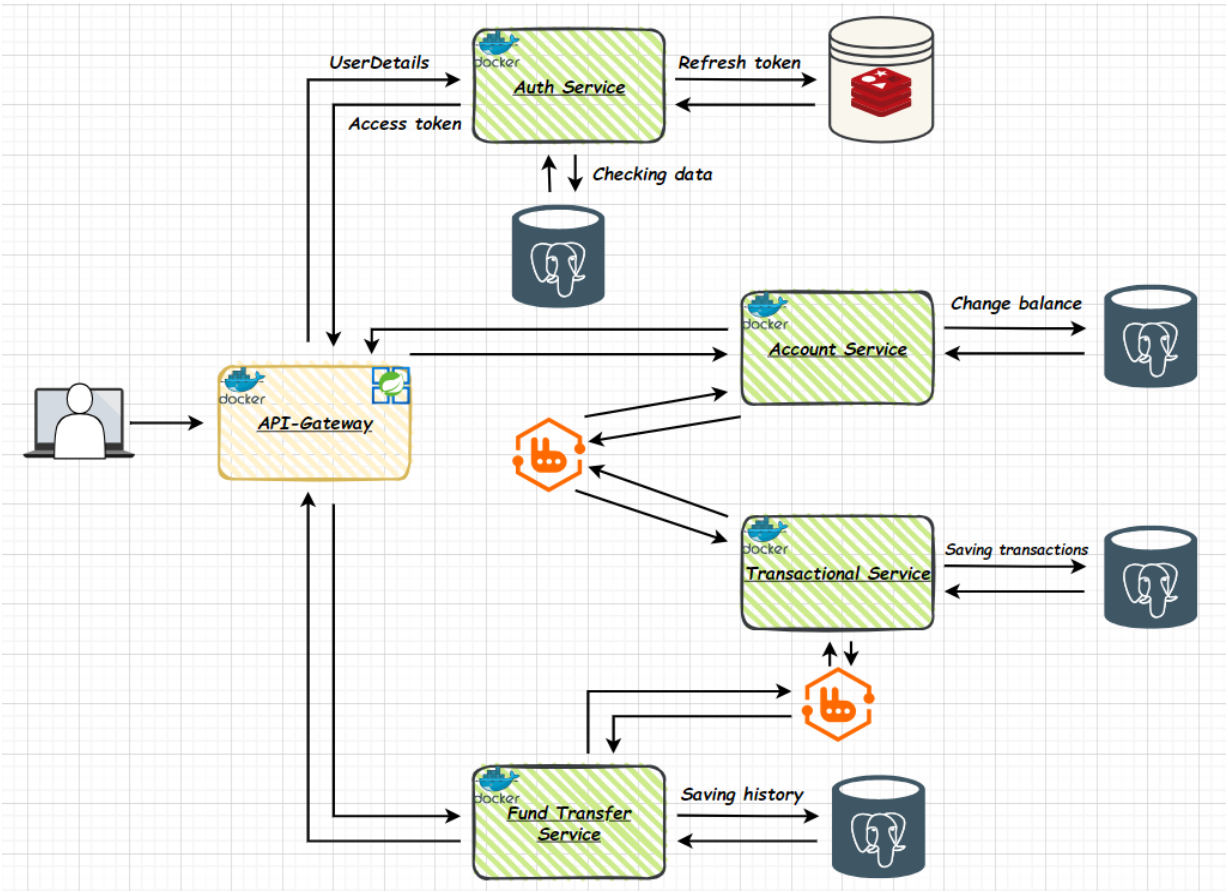

Рисунок 3.1 – Взаимодействие модулей системы

Схему взаимодействия между модулями системы можно описать следующим образом:

– Центральным координатором выступает API Gateway, выполняющий роль единой точки входа для всех клиентских запросов. Его функциональность включает аутентификацию входящих запросов и их интеллектуальную маршрутизацию к соответствующим сервисам. При получении запросов, касающихся операций с пользовательскими данными или финансовыми транзакциями, шлюз первоначально направляет их в Auth Service для верификации токенов доступа. После успешной проверки подлинности токена запросы перенаправляются в целевые сервисы с сохранением контекста авторизации.

– Account Service обрабатывает персональных данных клиентов и управленяет их аккаунтами. В его обязанности входит предоставление полной информации о пользователе, включая личные данные и состояние счетов, а также корректировка балансов по запросам других компонентов системы. Для синхронизации данных о балансе после выполнения транзакций сервис интегрируется с Transactional Service используя брокер сообщений.

– Transactional Service отвечает за обработку всех видов денежных операций. При получении запроса на выполнение транзакции сервис проверяет ее валидность, взаимодействуя с Account Service, после фиксирует результат операции в своей базе данных и передает ответ через цепочку обратно клиенту. В случае обнаружения ошибок сервис оперативно уведомляет инициатора транзакции через Fund Transfer Service.

– Fund Transfer Service занимается обработкой заявок на переводов средств между аккаунтами. Он создает заявку на выполнение транзакции и помещает ее в брокер сообщений, взаимодействуя с Transactional Service, а затем возвращает результаты обратно в API Gateway.

3.3 Доступ к модулям системы и их функциональность

Для взаимодействия с системой пользователям необходимо обращаться к базовому адресу шлюза API-Gateway, который в локальной среде разработки имеет вид http://localhost:8777, где 8777 представляет собой выделенный порт для доступа к gateway-сервису.

Архитектурно система реализует принципы REST API, что предполагает обмен данными в формате JSON. Для доступа к функциональности конкретного сервиса в URL-адресе указывается соответствующий модуль, что соответствует RESTful-подходу к организации конечных точек. Особенностью системы безопасности является обязательная предварительная валидация токенов доступа через Auth Service для всех входящих запросов. Только после успешной аутентификации запрос перенаправляется в целевые сервисы, такие как Fund Transfer Service или Account Service. Такой подход гарантирует соблюдение принципа минимальных привилегий и обеспечивает защиту от несанкционированного доступа.

Предварительно стоит настроить конфигурационный файл Spring Cloud Gateway, где определяются правила перенаправления запросов.

Фрагмент файла конфигурации:

```
spring:
  cloud:
    gateway:
      routes:
        - id: auth-service
          uri: lb://auth-service
          predicates:
            - Path=/auth/validateToken
        - id: fund-transfer-service
          uri: lb:// fund-transfer-service
          predicates:
            - Path=/fundTransfer
          filters:
            - TokenAuthenticationFilter
```

Программная реализация TokenAuthenticationFilter:

```
@Component
public class TokenAuthenticationFilter implements GatewayFilter {

    @Autowired
    private WebClient.Builder webClientBuilder;

    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {

        String token =
exchange.getRequest().getHeaders().getFirst(HttpHeaders.AUTHORIZATION)
;

        return webClientBuilder.baseUrl("http://auth-service")
                .get()
                .uri(uriBuilder ->
uriBuilder.path("/auth/validateToken").queryParam("token",
token).build())
                .retrieve()
                .bodyToMono(String.class)
                .flatMap(response -> {
                    if (response.equals("valid")) {

                        return chain.filter(exchange);
                    } else {

exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
                        return
exchange.getResponse().setComplete();
                    }
                });
    }
}
```

3.3.1 Модуль регистрации транзакций

Основная задача данного модуля – это инициировать транзакцию перевода средств между пользователями и передать сообщение в транзакционный сервис для дальнейшей обработки. Взаимодействие с сервисом аккаунтов и другие проверки происходят внутри транзакционного сервиса.

Доступ к сервису регистрации транзакций осуществляется по следующему адресу*:* `http://localhost:8777/fundTransfer`.

Рассмотрим эндпоинты, содержащиеся в сервисе.

POST-запрос на */make*: На данный эндпоинт отправляется тело запроса на создание заявки на новую транзакции. Оно включает в себя следующие поля: ID счета отправителя, ID счета получателя, сумму перевода и описание транзакции. При обработке запроса происходит публикация сообщения с телом в брокер сообщени. После обработки сообщения сервисом транзакций, и получения ожидаемого ответа, в базу данных сохраняется запись о инициации транзакции. Если сервис транзакций вернул ошибку, то возвращается HTTP-ответ со статусом 400 (Bad Request) и описанием ошибки. В случае успешного произведения транзакции, формируется HTTP ответ, с пустым телом и статусом 200 (OK).

POST-запрос на */cancel:* На данный эндпоинт отправляется тело запроса на отмену транзакции. Оно включает в себя следующие поля: ID счета отправителя, ID счета получателя, сумму перевода и описание транзакции. После происходит публикация сообщения с телом в брокер сообщений. После обработки сообщения сервисом транзакций, возвращается HTTP-ответ со статусом 200(OK), если же был указаной транзакции несуществует, или же она уже была выполнена , возвращается HTTP-ответ со статусом 400 (Bad Request) и описанием ошибки.

GET-запрос на */status/{transactionId}:* Отправляется тело запроса, содержащее ID транзакции. После происходит публикация сообщения с телом запроса и команды, которую следует выполнить сервису транзакций, в очередь RabbitMQ. После обработки сообщения сервисом транзакций, возвращается HTTP-ответ с состоянием произведенной транзакции и статусом 200(OK), если же был указан ID несуществующей транзакции возвращается HTTP-ответ со статусом 400 (Bad Request) и описанием ошибки.

3.3.2 Модуль управления счетами

Сервис аккаунтов обеспечивает операции со счетами пользователей, включая их создание, изменение и получение информации о балансе. Каждый

запрос проходит предварительную аутентификацию через сервис аутентификации. Доступ к сервису управления счетами осуществляется по следующему адресу*: http://localhost:8777/account-service*.

Рассмотрим эндпоинты, содержащиеся в сервисе.

GET-запрос на */accounts/{accountId}:* Отправляется тело запроса, содержащее ID счета, для получения общей информации.После обработки запроса возвращается HTTP-ответ с именем, фамилией, номером телефона, а также с датой последнего изменения данных и статусом 200(OK), если же был указан ID несуществующего счета возвращается HTTP-ответ со статусом 400 (Bad Request) и описанием ошибки.

GET-запрос на */accounts/{accountId}/balance*: Отправляется тело запроса, содержащее ID счета, для получения баланса. Ответ включает информацию только о балансе. После обработки запроса возвращается HTTP-ответ с состоянием с информацию о балансе и статусом 200(OK), если же был указан ID несуществующего счета возвращается HTTP-ответ со статусом 400 (Bad Request) и описанием ошибки.

PATCH-запрос на */accounts/changeData:* Отправляется тело запроса, содержащее новую или не заполненную общую информацию о владельце счета. Например имя, фамилию, номер телефона для изменения текущей. После обработки запроса возвращается HTTP-ответ со статусом 200(OK), если же произошла ошибка в обработке информации, возвращается HTTP-ответ со статусом 400 (Bad Request) и описанием ошибки.

3.3.3 Модуль обработки транзакций

Модуль транзакций является основным и отвечает за выполнение финансовых операций, таких как переводы средств между счетами, создание новых транзакций и обработка ошибок в процессе перевода.

Доступ к сервису обработки транзакций осуществляется предварительно через сервис регистрации транзакций, который является посредником, и позволяет скрыть все возможности от пользователя и выполнять операции только по непосредственной надобности. Данный модуль запускается локально по адресу: `http://localhost:8777/transactional-service`. Однако любое обращение к этому модюлу, будет маршрутизироваться обратно через API-Gateway.

Рассмотрим все эндпоинты, содержащиеся в сервисе.

POST-запрос на */make*: На данный эндпоинт отправляется тело запроса на создание транзакции. Оно включает в себя следующие поля: ID счета отправителя, ID счета получателя, сумму перевода и описание транзакции.

При обработке запроса происходит публикация сообщения с телом запроса в брокер сообщений. После обработки сообщения AccountService и валидации транзакции, AccountService отправляет сообщение со статусом 200 (OK) и в базу данных сохраняется запись о новой транзакции в случае успеха. Так же формируется ответ отправляемый на Fund Transfer Service со статусом 200 (OK). Если сервис аккаунтов вернул ошибку, то в Fund Transfer Service возвращается ответ со статусом 400 (Bad Request) и описанием ошибки.

GET-запрос на */{transactionId}:* Получение информации о конкретной транзакции по ее ID. В случае успеха — ответ со статусом 200 (OK) и данными о транзакции. В случае отсутствия транзакции — 400 (Bad Request).

POST-запрос на */cancel:* На данный эндпоинт отправляется тело запроса на отмену транзакции. Оно включает в себя следующие поля: ID счета отправителя, ID счета получателя, сумму перевода и описание транзакции. Если транзакция еще не была завершена, она отменяется, и возвращается статус 200 (OK). В случае неудачи — 400 (Bad Request).

ГЛАВА 4 РАЗВИТИЕ СИСТЕМЫ И ПЕРЕХОД К НОВОМУ СТЕКУ ТЕХНОЛОГИЙ

4.1 Причины пересмотра стека технологий

Первоначальная архитектура системы базировалась на стандартном наборе технологий, включающем Spring Boot для базовой функциональности, RabbitMQ для межсервисного взаимодействия, PostgreSQL и Redis для хранения данных, а также Docker для контейнеризации. Данный технологический выбор на начальном этапе разработки позволил оперативно реализовать рабочий прототип системы, сосредоточив основные усилия на реализации ключевых бизнес-процессов.

Однако в процессе дальнейшей разработки и благодаря приобретенному опыту коммерческой реализации распределенных систем, стало очевидным наличие более совершенных архитектурных подходов. Практическая работа над промышленными проектами предоставила ценную возможность для сравнительного анализа различных технологических решений, выявления их преимуществ и ограничений, а также глубокого понимания реальных требований к масштабируемости и отказоустойчивости в production-средах.

Проведенные нагрузочные тесты выявили существенные ограничения текущей реализации в условиях повышенной нагрузки, что послужило катализатором для пересмотра архитектурных решений. Полученные результаты продемонстрировали необходимость внедрения более современных и устойчивых к нагрузкам технологий, способных обеспечить гибкое масштабирование системы в соответствии с изменяющимися требованиями. Подробное описание проведённого нагрузочного тестирования будет представлено в одном из следующих разделов данной главы.

Было принято архитектурное решение о переходе на обновлённый стек технологий, включающий:

1. Apache Kafka в качестве отказоустойчивой и масштабируемой системы обмена сообщениями.
2. Apache Camel как средство маршрутизации и интеграции микросервисов.
3. Keycloak для централизованного управления аутентификацией и авторизацией.
4. Kubernetes для автоматизации развертывания и масштабирования.

4.2 Нагрузочное тестирование первоначальной системы

Нагрузочное тестирование проводилось с целью оценки стабильности и производительности системы при различных уровнях одновременной пользовательской активности. В процессе моделировались типовые сценарии, за исключением аутентификации пользователей, генерации и обновления токенов.

В качестве основных эндпоинтов были выбраны */fundTrunsfer/make*, */account-service/accounts/changeData* и */account-service/accounts/{accountId}*. Выбор данных эндпоитов обусловлен тем, что некоторые их них, связанные с сервисом аккаунта, работают обособленно и не затрагиваются другими сервисам, а эндпоинт */fundTrunsfer/make* косвенно вызывает оставшиеся эндпоинты системы. Таким образом, протестировав систему на этих трех эндпоинтах, можно судить о работоспособности системы в целом.

Для проведения тестов использовался инструмент Jmeter, осуществлявший генерацию запросов к трём ключевым эндпоинтам системы. Общий объём тестовой выборки составил 5000 запросов, распределённых равномерно между различными типами операций. Тестирование проводилось в течение ограниченного временного интервала с 12 запросами в секунду, что позволило отследить динамику поведения системы под нагрузкой.

JMeter предоставляет возможность не только задавать параметры нагрузочного тестирования, но и автоматически формировать отчёты в виде графиков и таблиц. Эти отчёты отражают такие ключевые метрики, как время отклика, процент успешных запросов, средняя и максимальная задержка, а также уровень ошибок. Благодаря визуализации стало возможным наглядно оценить поведение каждого эндпоинта в разные периоды времени и выявить узкие места.

Далее представлены результаты, полученные в процессе тестирования. На рисунке 4.1 отражена статистика всех запросов, направленных к системе. На рисунке 4.2 изображена скорость ответа на запрос к эндпоинтам.

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**5000**	**1096**	**21.92%**	**573.40**	**102**	**1103**	**525.00**	**1067.90**	**1094.00**	**1118.00**	**11.05**	**13.83**	**0.00**
GET /account-service/accounts/{accountId}	1669	394	23.61%	570.82	102	1078	513.00	1069.00	1094.50	1118.30	3.69	4.60	0.00
PATCH /account-service/accounts/changeData	1699	343	20.19%	551.82	106	1096	504.00	1064.00	1093.00	1117.00	3.75	4.77	0.00
POST /fundTransfer/make	1632	359	22.00%	598.49	133	1103	561.00	1069.00	1095.35	1118.67	3.61	4.47	0.00

Рисунок 4.1 – Результаты нагрузочного тестирования системы

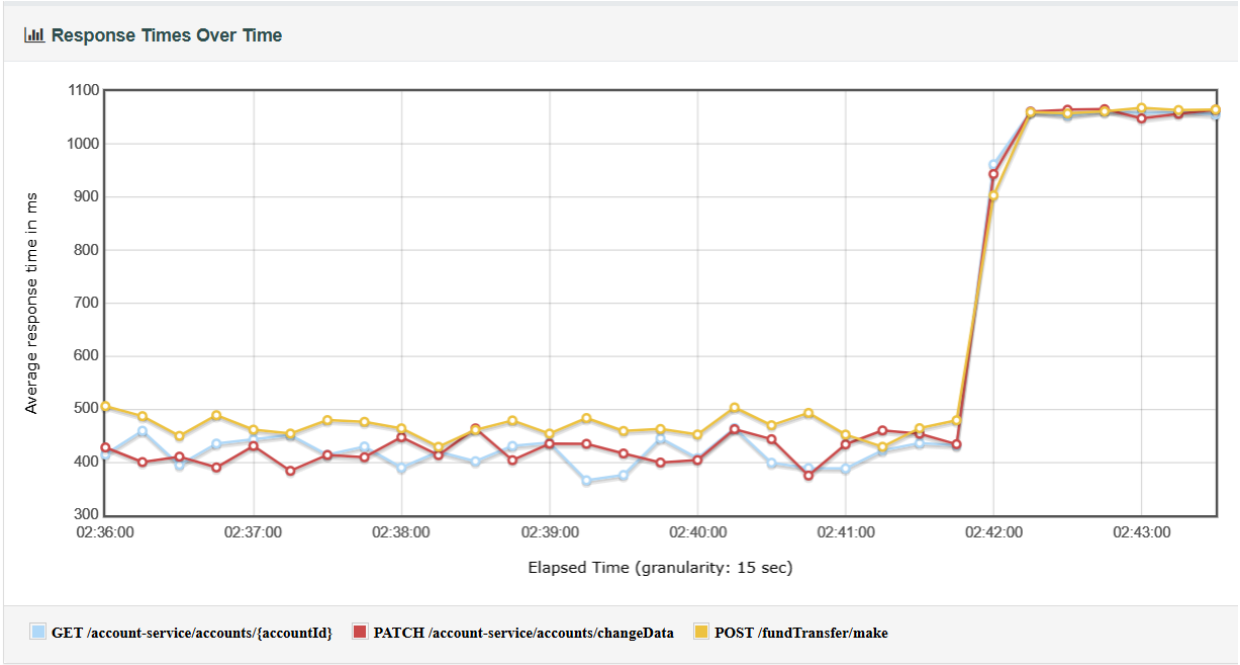

Рисунок 4.2 – Скорость ответа на запрос к эндпоинтам

Проанализировав таблицу и график, можно сделать вывод, что API Gateway испытывал значительные трудности при обработке поступающей нагрузки. Это можно понять исходя из того, что скорость обработки запросов всеми сервисами одновременно возросла. Высокий процент ошибок (21.92% в среднем по всем меткам) свидетельствует о возникновении каких-либо критических сбоев. Значения времени выполнения запросов, достигающие 1187 мс, а также высокие значения 90–99-го процентилей, указывают на существенное увеличение времени отклика в пиковых ситуациях. Это может быть связано как с перегрузкой самого API Gateway, так и с недоступностью или замедлением работы зависимых сервисов. Таким образом, полученные данные демонстрируют снижение устойчивости системы под нагрузкой, что свидетельствует о необходимости оптимизации производительности и повышения отказоустойчивости API Gateway.

Так же стоит рассмотреть рисунок 4.3, отражающий количество успешных и ошибочных транзакций в секунду. На графике можно заметить, что в определенный момент, все из них не завершились успехом. Это подтверждает то, что API Gateway, в первичной реализации, являлся узким местом системы и единой точкой отказа.

Исходя из результатов тестирования, можно сделать вывод, что данная версия системы является не оптимальной и подлежит доработке.

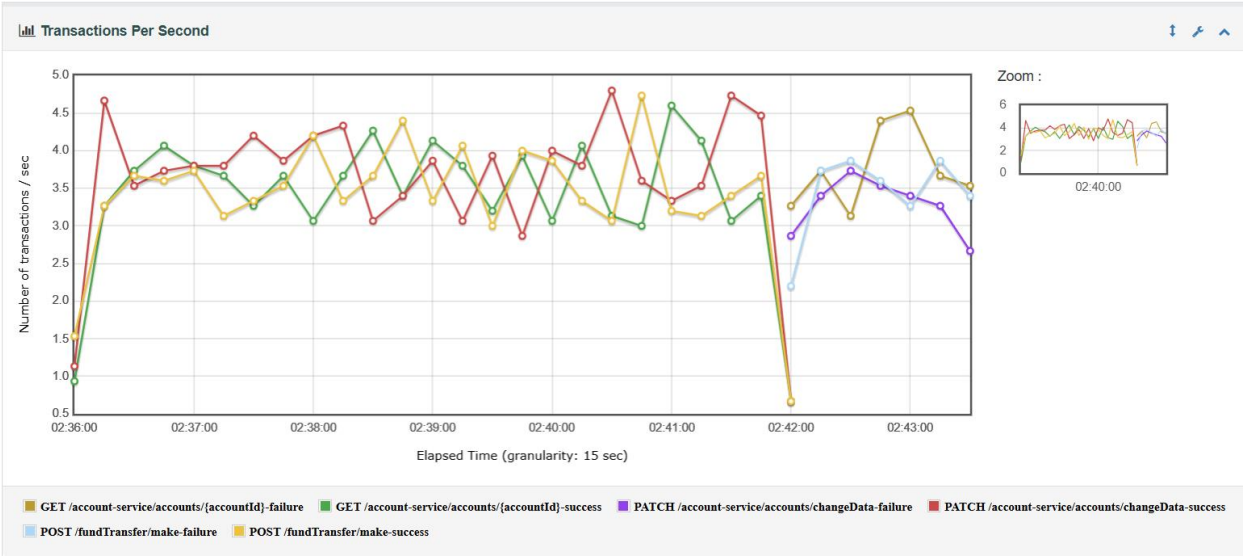

Рисунок 4.3 – Количество успешных транзакций

4.3 Пересмотр изначального стека технологий

4.3.1 Ограничения RabbitMQ при обработке сообщений

На начальном этапе в качестве брокера сообщений использовался RabbitMQ – надёжное решение для реализации очередей и маршрутизации. Однако по мере усложнения бизнес-логики и увеличения объёмов транзакционных данных, стало очевидно, что необходима более масштабируемая и отказоустойчивая система.

Было принято решение перейти на Apache Kafka, специально спроектированную для обработки больших объёмов данных в реальном времени. Сравнительный анализ двух технологий отражен в таблице 4.1.

Таблица 4.1 – Сравнительный анализ Apache Kafka и RabbitMQ

Характеристика	Apache Kafka	RabbitMQ
Модель обмена сообщениями	Лог (log-based), pub-sub и stream processing	Очереди (queue-based), pub-sub
Подход к данным	Хранит сообщения на диске, может сохранять их неделями и даже месяцами	Сообщения удаляются после получения
Производительность	Очень высокая, особенно при большом объёме данных и количестве потребителей	Подходит для умеренных нагрузок

Характеристика	Apache Kafka	RabbitMQ
Масштабируемость	Изначально проектировалась для горизонтального масштабирования	Масштабируется хуже, требует дополнительных усилий
Гарантии доставки	At most once, at least once, exactly once (с настройками и idempotent producer)	At most once, at least once (exactly once — только частично)
Поддержка очередей	Нет обычных "очередей", потребители читают независимо	Очередь — объект, из которого потребители вытаскивают сообщения
Порядок сообщений	Гарантирован в пределах одного partition	Гарантирован в пределах одной очереди
Задержки	Меньше задержек при больших нагрузках (batching, zero-copy)	Иногда быстрее на маленьких объёмах, но не масштабируется
Требования к ресурсам	Выше (особенно для ZooKeeper/Quorum)	Ниже, можно запустить на лёгкой VM
Надёжность хранения	Очень высокая: хранение на диске, репликации	Основной упор на доставку, но не на долговременное хранение
Управление	Сложнее (особенно при кластерах и партиционировании)	Проще в настройке и запуске

Основные преимущества Kafka:

– Высокая отказоустойчивость за счёт репликации сообщений между брокерами.

– Гарантированную доставку и сохранение истории сообщений с возможностью повторного воспроизведения.

– Kafka легко работает в Kubernetes-кластере с множеством подов-потребителей, обрабатывающих сообщения параллельно.

– Гибкую балансировку нагрузки через распределение партиций между группами консьюмеров.

Эти преимущества особенно важны в финансовом приложении, где критически важно обеспечить надежную доставку сообщений между микросервисами, возможность восстановления после сбоев и масштабируемость без потери целостности данных.

4.3.2 Централизованная маршрутизация и интеграция

Первоначально взаимодействие между микросервисами реализовывалось с помощью стандартных инструментов Spring (таких как @RestTemplate, @KafkaListener, ручная настройка маршрутов и логики обмена). Такой подход оказался трудоёмким, особенно при необходимости создания цепочек взаимодействий, обработки ошибок, логирования или трансформации данных.

В процессе коммерческой разработки стало очевидно, что для построения устойчивых и читаемых интеграционных маршрутов необходим более удобный инструмент. Таким решением стал Apache Camel — открытый интеграционный фреймворк, разработанный для упрощения и стандартизации процесса интеграции различных программных систем и приложений. В контексте современных распределенных архитектур особую актуальность приобретает задача организации взаимодействия между разнородными компонентами, использующими различные протоколы передачи данных и форматы сообщений. Apache Camel предлагает комплексное решение для подобных интеграционных задач через реализацию проверенных шаблонов корпоративной интеграции (Enterprise Integration Patterns).

В Apache Camel все построено на маршрутах(routes). Каждый маршрут определяет последовательность обработки данных при их передаче между конечными точками системы. Для маршрутов Apache Camel предоставляет специализированные языки (DSL), позволяющими повысить читаемость кода.

Важнейшим преимуществом Apache Camel является его поддержка разнообразных протоколов и технологий передачи данных. Фреймворк предлагает готовые компоненты для работы с такими широко распространенными стандартами как HTTP и REST API, системы обмена сообщениями (JMS), файловые протоколы (FTP), технологии баз данных (JDBC) и веб-сервисы (SOAP). Это достигается благодаря модульной архитектуре, где каждый компонент реализует взаимодействие с определённой технологией. Apache Camel широко применяется в корпоративных интеграционных решениях, микросервисных архитектурах и системах обработки сообщений, позволяя снизить сложность интеграционных задач и повысить надёжность обмена данными.

Внедрение Apache Camel позволило повысить читаемость и модульность кода и значительно упростить реализацию взаимодействия между сервисами в условиях быстрорастущей системы.

4.3.3 Масштабирование и оркестрация

На ранних этапах разработки основным инструментом для контейнеризации приложения выступал Docker. Он обеспечивал удобную упаковку микросервисов и их изоляцию, что позволяло быстро запускать систему как на локальных машинах, так и в тестовых средах. Однако по мере усложнения проекта и роста числа компонентов начали проявляться ограничения. Процесс управления конфигурацией системы сопровождался существенными операционными сложностями. Любые модификации параметров подключения к внешним ресурсам, будь то базы данных или брокеры сообщений, неизбежно требовали полного цикла пересборки контейнерных образов и их повторного развертывания.

Дополнительные ограничения проявлялись в отсутствии механизмов автоматического масштабирования и восстановления после сбоев, что снижало отказоустойчивость системы и делало ее уязвимой при возникновении пиковых нагрузок. Решением этих проблем стал Kubernetes – платформа для оркестрации контейнеров в кластере. Внедрение Kubernetes устранило необходимость постоянной пересборки контейнеров при изменении параметров конфигурации и предоставило встроенные механизмы динамического масштабирования. Kubernetes позволил не только решить первоначальные проблемы, но и заложить фундамент для дальнейшего масштабирования системы.

4.3.4 Преодоление ограничений API Gateway

На начальном этапе архитектура приложения предусматривала использование API Gateway как основного инструмента для маршрутизации запросов и организации контроля доступа. Такой подход позволял реализовать базовые механизмы аутентификации и авторизации, но по мере развития системы потребности в более гибкой и масштабируемой системе управления доступом возросли.

Хотя API Gateway и был способен выполнять функции фильтрации и перенаправления, реализация сложной логики авторизации, работы с ролями, токенами, сессиями и политиками безопасности требовала значительных усилий и ручной настройки. Кроме того, отсутствовала встроенная интеграция с внешними системами аутентификации, возможностями single sign-on и централизованным управлением пользователями.

В этой связи было принято решение внедрить Keycloak – специализированное решение для управления пользователями и безопасностью. Keycloak предоставляет широкий набор возможностей «из

коробки»: поддержку OAuth2, OpenID Connect, интеграцию с LDAP и другими провайдерами, администрирование пользователей через веб-интерфейс, настройку ролей, прав доступа, а также детальную настройку политики безопасности.

Переход на Keycloak позволил значительно упростить реализацию авторизации в системе, разгрузить микросервисы от необходимости обрабатывать токены и проверять права вручную, а также централизовать управление пользователями. В то же время API Gateway остался в архитектуре, но теперь он сосредоточен исключительно на маршрутизации и делегирует задачи безопасности специализированному сервису.

Параллельно с интеграцией Keycloak было принято решение модернизировать технологическую основу сервиса, перейдя с классической модели Spring MVC на реактивную модель Spring WebFlux. Такой подход позволит более эффективно обрабатывать большое количество одновременных запросов благодаря асинхронной и неблокирующей архитектуре. WebFlux обеспечивает высокую масштабируемость и лучше подходит для сценариев с высокой нагрузкой, что особенно актуально в условиях микросервисной архитектуры, где каждый сервис может выступать как потребителем, так и поставщиком данных.

Интеграция WebFlux также должна повысить отзывчивость и устойчивость системы при пиковой нагрузке, сократив время отклика и снизив потребление ресурсов.

4.4 Схема взаимодействия и реактивное программирование

В процессе модернизации технологического стека было осуществлено существенное изменение модели взаимодействия между микросервисами, направленная на оптимизацию ключевых характеристик системы. Важно подчеркнуть, что проведенные изменения, не затронули базовую функциональную логику бизнес-процессов, сохранив неизменными основные принципы работы приложения.

Архитектурные преобразования затронули прежде всего аспекты межсервисной коммуникации, механизмы обработки запросов и стек используемых технологий, что в совокупности позволит достичь значительного улучшения таких показателей как производительность, надежность и отказоустойчивость системы. Все внесенные изменения систематизированы и представлены в таблице 4.2, где наглядно отражена эволюция архитектурных решений.

Таблица 4.3 – Изменения первоначальной системы

Компонент	Было	Стало	Описание улучшений
API Gateway	Синхронный REST, блокирующий I/O	WebFlux, неблокирующий I/O	Быстрее, лучше масштабируется
Взаимодействие между сервисами	RabbitMQ	Kafka	Меньше задержек, выше отказоустойчивость
Аутентификация	Auth Service + Redis+refresh token	Keycloak + JWT	Упрощено, меньше зависимостей, нет Redis
Обработка запросов	Последовательная, много HTTP-вызовов	Распределённая реактивная через сообщения	Лучшая параллелизация, меньше конкуренции за ресурсы
Брокер сообщений	RabbitMQ	Apache Kafka	Выше пропускная способность, лучше масштабирование
Refresh-токены	Использовались, хранились в Redis	Отказались	Безопаснее и проще

Основным направлением изменений стал переход от синхронного взаимодействия к асинхронной реактивной архитектуре. В предыдущей реализации API Gateway выступал в роли синхронного посредника, обрабатывая запросы блокирующим образом и посредством REST API взаимодействуя с целевыми микросервисами. Такой подход создавал ограничения по пропускной способности и мог стать узким местом при высокой нагрузке из-за конкуренции за ресурсы.

Новая архитектура основана на использовании Spring WebFlux – реактивного фреймворка с event-loop моделью, позволяющей эффективно обрабатывать большое количество параллельных HTTP-запросов без блокировок. Это позволило существенно улучшить масштабируемость и сократить задержки в обработке.

Коммуникация между компонентами была переработана с применением асинхронного обмена сообщениями через Apache Kafka. API Gateway теперь преобразует входящие запросы в сообщения с уникальными UUID и публикует их в тематические Kafka-топики, соответствующие конкретным микросервисам.

Микросервисы, подписанные на соответствующие топики, обрабатывают сообщения и публикуют результаты в отдельные топики ответов. API Gateway, используя UUID, ассоциирует ответы с исходными запросами и возвращает клиенту данные в реактивном режиме, что должно минимизировать блокировки при обмене сообщениями. Это позволит

оптимизировать распределение нагрузки и повысить устойчивость всей системы.

Отказ от отдельного микросервиса для аутентификации (auth service) стал ещё одним важным шагом в упрощении архитектуры. Вместо этого аутентификация была делегирована фреймворку Keycloak — надёжного решения для управления идентификацией и доступом (IAM). Keycloak теперь напрямую обрабатывает все запросы на получение токенов, выступая в роли единого OAuth 2.0 провайдера. Клиентские сервисы используют flow Client Credentials, получют access-токены в формате JWT, которые передаются с каждым запросом через API Gateway.

Устранение промежуточного сервиса позволило избавиться от лишних зависимостей, упростить логику управления токенами и снизить потенциальные точки отказа. Кроме того, больше нет необходимости хранить токены или сессионные данные во внешнем хранилище, так как аутентификация и авторизация полностью перенесены на API Gateway, который работает с Keycloak. В этой схеме refresh-токены не используются, что соответствует современным тенденциям в области безопасности. Отказ от них снижает риск компрометации и устраняет необходимость управления их жизненным циклом. Такой подход обеспечивает надёжную и криптографически защищённую авторизацию, минимизирует эксплуатационные расходы и делает систему более прозрачной и безопасной.

Внедрение реактивного API Gateway, событийно-ориентированной системы обмена сообщениями на базе Kafka и централизованной аутентификации через Keycloak позволило создать архитектуру, которая отличается масштабируемостью, отказоустойчивостью и высоким уровнем безопасности. Такой подход помог избавиться от синхронных узких мест в коммуникации между сервисами. Удаление отдельного auth-сервиса и отказ от хранения токенов упростили сопровождение системы и повысили её надёжность. В результате получилась архитектура, полностью соответствующая требованиям банковских приложений, устойчивая кросту нагрузки и готовая к масштабированию под сложные бизнес-задачи. Полная схема взаимодействия отражена на рисунке 4.1.

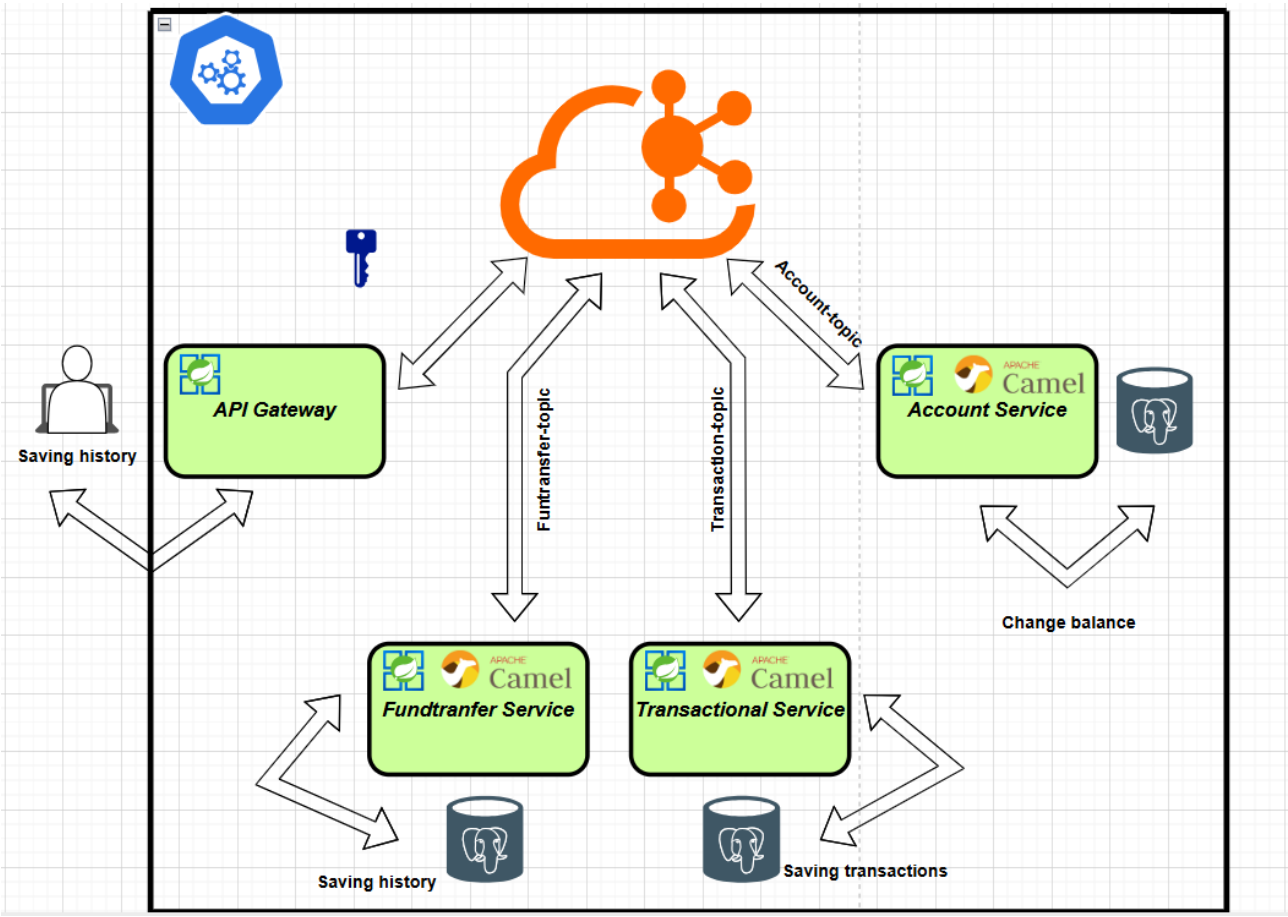

Рисунок 4.1 – Взаимодействие компонентов системы

4.5 Нагрузочное тестирование обновленной системы

Нагрузочное тестирование обновлённой версии системы проводилось с целью оценки изменений в производительности и устойчивости архитектуры после внедрения нового стека технологий и изменения логики взаимодействия модулей системы. Основной задачей было сравнение результатов с предыдущей реализацией, чтобы определить, насколько внедрённые улучшения повлияли на общее поведение системы под нагрузкой.

Для обеспечения корректности сравнения использовались те же самые эндпоинты: /fundTrunsfer/make, /account-service/accounts/changeData и /account-service/accounts/{accountId}. Методология тестирования также осталась неизменной, за исключением использования kubernetes. Kubernetes позволил запускать сразу несколько экземпляров одного сервиса одновременно. Было решено использовать по два экземпляра каждого из сервисов во время тестирования. Исключались сценарии, связанные с аутентификацией, генерацией и обновлением токенов, поскольку основное внимание уделялось внутренней логике бизнес-процессов и взаимодействию между микросервисами.

В качестве инструмента нагрузочного тестирования вновь применялся JMeter, который использовался для генерации 5000 запросов с равномерным распределением между тремя ключевыми эндпоинтами. Скорость генерации запросов составляла 12 запросов в секунду, как и в предыдущем тестировании, что позволило обеспечить одинаковые условия и провести объективное сравнение.

JMeter также использовался для сбора и визуализации метрик: отчёты включали графики и таблицы с данными по времени отклика, количеству успешных и ошибочных запросов, пиковым задержкам и среднему времени обработки.

Рассмотрим результаты проведенного нагрузочного тестирования. В таблице 4.4 отражена общая статистика по всем запросам, направленным к обновленной системе.

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**5000**	**81**	**1.62%**	**312.74**	**84**	**1123**	**301.00**	**476.00**	**528.95**	**1057.96**	**11.02**	**13.66**	**0.00**
GET /account-service/accounts/{accountId}	1682	33	1.96%	289.31	84	1094	281.00	426.00	446.00	1082.00	3.71	4.60	0.00
PATCH /account-service/accounts/changeData	1653	15	0.91%	298.43	87	1123	291.00	458.60	479.00	551.46	3.65	4.52	0.00
POST /fundTransfer/make	1665	33	1.98%	350.61	103	1108	341.00	538.40	560.00	1048.72	3.67	4.56	0.00

Рисунок 4.4 – Результаты нагрузочного тестирования обновленной системы

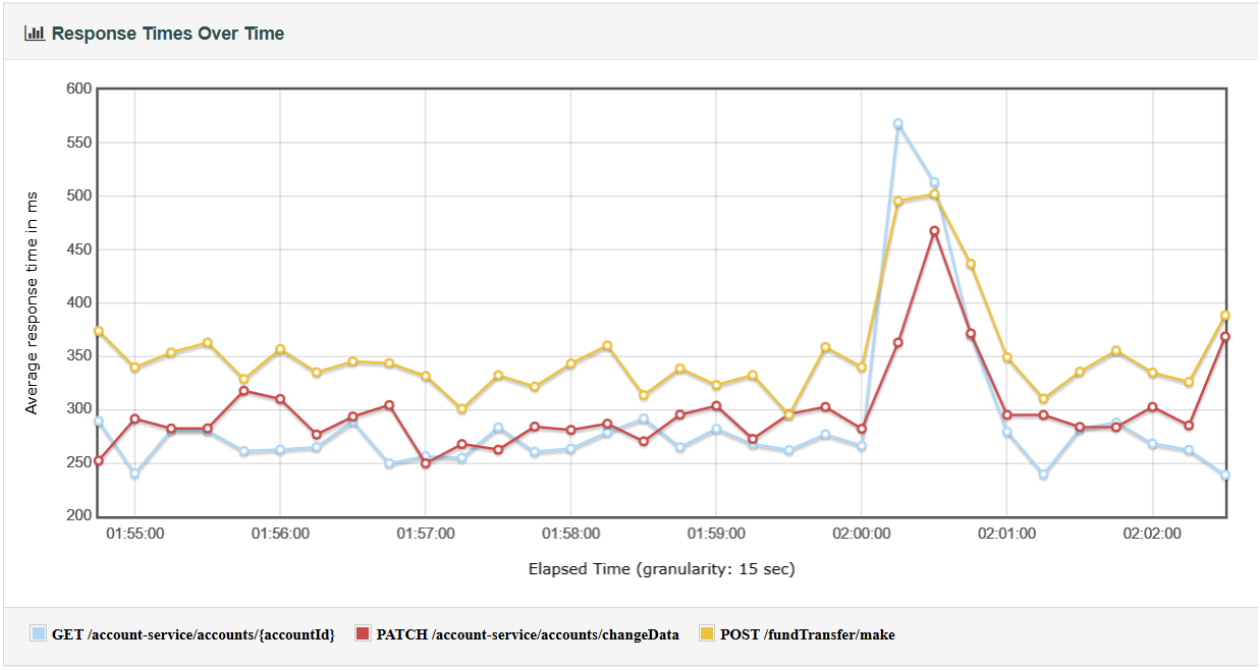

Рисунок 4.3 – Скорость ответа на запрос к эндпоинтам

Сравнив результаты полученный сейчас с результатами, полученными ранее, можно сделать вывод что переход на новый стек технологий и изменения логики взаимодействия между сервисами, ускорил скорость обработки запросов в 1,8 раза. Так же, процент запросов, не закончившихся успехом, уменьшился почти в 13 раз.

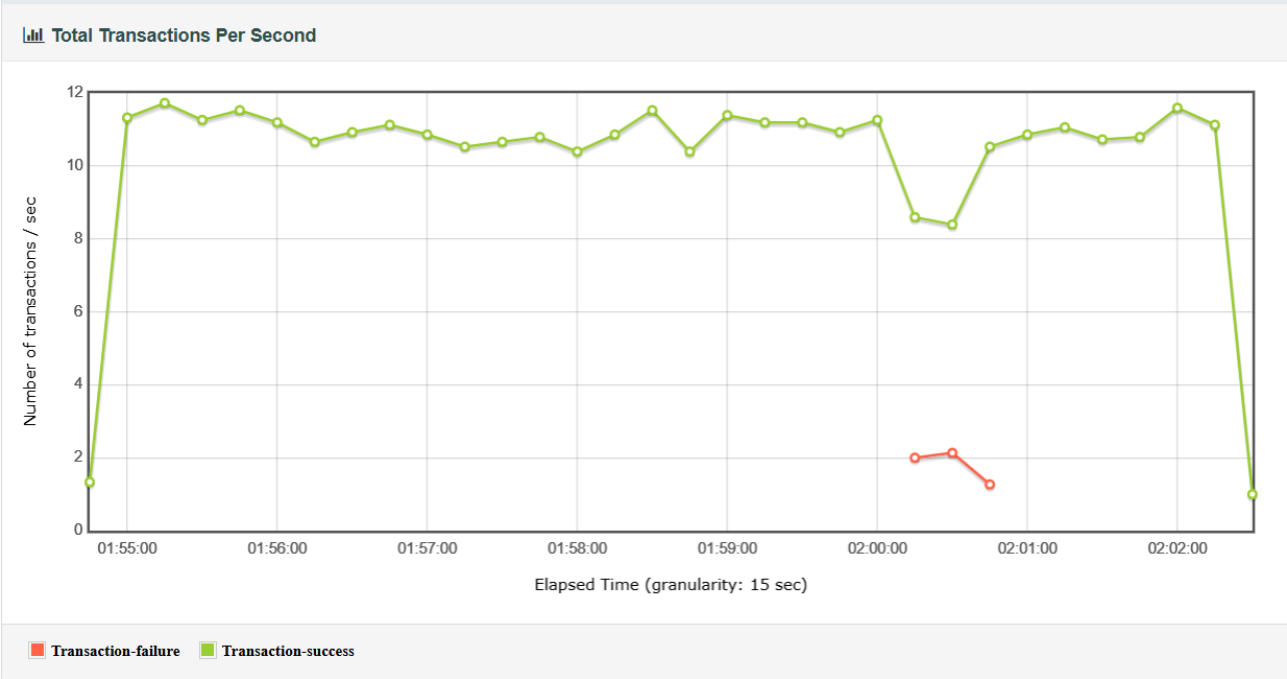

Рисунок 4.4. – Количество успешных транзакций

Изучив графики 4.3 и 4.4, можно заметить, что в некоторый промежуток времени, система испытывала трудности, и отдавала ошибку 503 Gateway Timeout, так же скорость ответа в этот промежуток заметно снизилась. Это можно объяснить тем, что один из сервисов в кластере Kubernetes не справился с нагрузкой и отключился, однако другой сервис смог взять всю нагрузку на себя, пока другой сервис перезапускался.

Полученные результаты наглядно демонстрируют важность правильного проектирования механизмов отказоустойчивости в распределённых системах. Даже при наличии нескольких экземпляров сервиса отсутствие механизмов быстрого восстановления может привести к существенному уменьшению производительности.

Проведённое нагрузочное тестирование позволило выявить как сильные стороны текущей реализации системы, так и направления для дальнейшей оптимизации.

На основании проведённого анализа можно предложить ряд мер по улучшению характеристик системы. Внедрение механизма автоматического

масштабирования (Horizontal Pod Autoscaler) позволит динамически адаптировать количество экземпляров сервисов к текущей нагрузке. Реализация шаблонов устойчивости, таких как Circuit Breaker и Retry, поможет предотвратить каскадные сбои при временной недоступности отдельных компонентов системы. Дополнительное кэширование данных account-service может существенно снизить нагрузку на этот критически важный компонент.

ЗАКЛЮЧЕНИЕ

В рамках дипломной работы была разработана система для обработки банковских транзакций на основе микросервисной архитектуры. Изначально предполагалась реализация одного микросервиса, однако в процессе проектирования была построена полноценная распределённая банковская система, включающая необходимые вспомогательные компоненты.

Первоначально разработанная система прошла нагрузочное тестирование, в ходе которого были выявлены направления для оптимизации. Это позволило улучшить первоначальную версию и внедрить более эффективные подходы к маршрутизации, безопасности и асинхронному взаимодействию.

Выполнены следующие ключевые задачи:

1. Изучены подходы к построению микросервисной архитектуры.
2. Освоены и применены современные инструменты разработки.
3. Спроектирована архитектура системы и реализована схема взаимодействия микросервисов.
4. Внедрены механизмы аутентификации и авторизации (Keycloak), маршрутизации и асинхронной коммуникации.
5. Проведено нагрузочное тестирование и выполнена оптимизация компонентов, путем изменения логики взаимодействия компонентов и внедрения нового стека технологий.
6. Разработан микросервис для обработки транзакций в полном объёме.

Система в текущем виде демонстрирует устойчивую работу и соответствует большинству требований.

Таким образом, поставленные цели дипломной работы достигнуты. Разработанная система может служить основой для масштабируемого банковского решения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. 26 основных паттернов микросервисной разработки [Электронный ресурс] // Cloud VK. – Режим доступа: https://cloud.vk.com/blog/26-osnovnyh-patternov-mikroservisnoj-razrabotki. – Дата доступа: 30.04.2025.
2. PostgreSQL: все, что нужно знать для быстрого старта [Электронный ресурс] // REG.RU. – Режим доступа: https://www.reg.ru/blog/postgresql-vsyo-chto-nuzhno-znat-dlya-bystrogo-starta/. – Дата доступа: 28.04.2025.
3. Сервис-ориентированная архитектура (SOA) [Электронный ресурс] // Habr. – Режим доступа: https://habr.com/ru/companies/vk/articles/342526/. – Дата доступа: 12.04.2025.
4. Spring Cloud и Spring Boot. Использование Eureka Server [Электронный ресурс] // Habr. – Режим доступа: https://habr.com/ru/companies/otus/articles/539348/. – Дата доступа: 16.04.2025.
5. Spring Framework [Электронный ресурс] // Википедия. – Режим доступа: https://ru.wikipedia.org/wiki/Spring_Framework. – Дата доступа: 02.05.2025.
6. Шаблоны шлюзов API в микросервисах [Электронный ресурс] // GeeksforGeeks. – Режим доступа: https://www.geeksforgeeks.org/api-gateway-patterns-in-microservices/#benefits-of-using-api-gateway-in-microservices. – Дата доступа: 30.04.2025.
7. Bauer, C. Java Persistence with Hibernate / C. Bauer, G. King, G. Gregory. – 2nd ed. – Shelter Island : Manning, 2015.
8. Carnell, J. Spring Microservices in Action / J. Carnell. – Shelter Island : Manning, 2017.
9. How Eureka Server and Client Communicate with Each Other in Microservices? [Электронный ресурс] // GeeksforGeeks. – Режим доступа: https://www.geeksforgeeks.org/how-eureka-server-and-client-communicate-with-each-other-in-microservices/. – Дата доступа: 16.04.2025.
10. Miell, I. Docker in Practice / I. Miell, A. H. Sayers. – 2nd ed. – Shelter Island : Manning, 2020.
11. Newman, S. Building Microservices: Designing Fine-Grained Systems / S. Newman. – Sebastopol : O’Reilly Media, 2015.
12. NoSQL Use-Cases: When to Use a Non-Relational database [Электронный ресурс] // DataStax. – Режим доступа: https://www.datastax.com/blog/sql-vs-nosql-whats-the-difference. – Дата доступа: 16.04.2025.
13. Richardson, C. Microservices Patterns: With Examples in Java / C. Richardson. – Shelter Island : Manning, 2019.