Министерство науки и высшего образования РФ

Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский технологический университет «МИСИС»

Институт компьютерных наук (ИКН)

Кафедра Инфокоммуникационных технологий (ИКТ)

**Курсовая работа**

по дисциплине

«Архитектурирование»

Направление подготовки:

09.03.01 «Информатика и вычислительная техника»

**Выполнила:**  
Самикова Е.А.  
ст. гр. БИВТ-22-СП-2

Москва, 2025

Оглавление

[**Введение** 3](#_Toc198150448)

[**Цель работы** 3](#_Toc198150449)

[**Используемые технологии** 3](#_Toc198150450)

[**Теоретическая часть** 5](#_Toc198150451)

[**Контейнеризация в современном программировании** 5](#_Toc198150452)

[**Кеширование и его роль в архитектуре** 5](#_Toc198150453)

[**Event-driven архитектура** 7](#_Toc198150454)

[**Сочетание подходов в архитектуре** 8](#_Toc198150455)

[**Проектирование архитектуры** 10](#_Toc198150456)

[**Общая архитектура** 10](#_Toc198150457)

[**Функциональность приложения** 11](#_Toc198150458)

[**Стратегия кеширования** 12](#_Toc198150459)

[**Асинхронная обработка через Producer-Consumer** 13](#_Toc198150460)

[**Бизнес-логика** 14](#_Toc198150461)

[**Реализация** 15](#_Toc198150462)

[**Структура проекта** 15](#_Toc198150463)

[**Реализация компонентов** 16](#_Toc198150464)

[**Тестирование** 19](#_Toc198150465)

[**Redis** 19](#_Toc198150466)

[**RabbitMQ** 21](#_Toc198150467)

[**Grafana** 21](#_Toc198150468)

[**Prometheus** 22](#_Toc198150469)

[**Mailplt** 23](#_Toc198150470)

[**CQRS и Event Sourcing** 24](#_Toc198150471)

[**Заключение** 29](#_Toc198150472)

[**Список литературы** 31](#_Toc198150473)

# **Введение**

Современная разработка программного обеспечения нуждается в применении гибких и масштабируемых архитектурных подходов, которые позволяют создавать устойчивые, надежные и легко развертываемые приложения. По мере постоянного повышения требований к производительности и отказоустойчивости программных систем, особенно важными становятся технологии, которые обеспечивают эффективное управление взаимодействием между компонентами системы и их изоляцию. Данная курсовая работа посвящена проектированию и реализации программной системы, построенной на основе микросервисной архитектуры с применением ключевых технологий: Docker, Redis и RabbitMQ.

## **Цель работы**

Целью данной работы является проектирование архитектуры программного решения, которое будет ориентировано на обработку и передачу данных между различными компонентами с использованием современных инструментов контейнеризации, кэширования и обмена сообщениями. Предполагается разработка архитектурной схемы, обоснование выбора технологий, а также демонстрация взаимодействия сервисов в рамках предложенной архитектуры.

## **Используемые технологии**

Docker — это платформа для создания и управления контейнерами, которые представляют собой изолированные среды для запуска приложений со всеми необходимыми зависимостями. Благодаря Docker процесс развертывания и управления сервисами становится проще, поскольку контейнеры обеспечивают одинаковую работу приложения в любой среде — будь то локальная машина разработчика или облачный сервер. Это особенно ценно в микросервисных архитектурах, где каждый компонент может требовать разные версии библиотек и настройки. Docker помогает избежать конфликтов в окружении, повышает стабильность системы и упрощает интеграцию с CI/CD-процессами. Кроме того, контейнеры потребляют меньше ресурсов и запускаются быстрее, чем виртуальные машины, что делает их эффективным решением для разработки и масштабирования приложений.

Redis — это высокоскоростное хранилище данных, работающее в оперативной памяти (RAM). Оно часто используется для кэширования и временного хранения информации, обеспечивая более быстрый доступ по сравнению с традиционными базами данных. Благодаря поддержке различных структур данных и функции автоматического удаления устаревших записей Redis идеально подходит для работы с сессиями пользователей, временными токенами и часто запрашиваемыми данными. В данной работе Redis применяется как вспомогательный инструмент для снижения нагрузки на основные хранилища, ускорения обработки запросов и повышения общей производительности системы.

RabbitMQ — это брокер сообщений, который организует асинхронный обмен данными между компонентами системы через очереди. Вместо прямого взаимодействия сервисы обмениваются сообщениями: отправитель публикует задачу в очередь, а получатель обрабатывает её, когда будет готов. Такой подход уменьшает связность компонентов, повышает отказоустойчивость и упрощает масштабирование системы. RabbitMQ особенно полезен в распределённых архитектурах, где требуется надёжная доставка сообщений, фоновая обработка задач или сложная маршрутизация данных между микросервисами. В рамках данной работы RabbitMQ выступает в роли центрального звена, обеспечивающего стабильное взаимодействие сервисов и обработку событий в распределённой среде.

# **Теоретическая часть**

## **Контейнеризация в современном программировании**

Контейнеризация — это способ упаковки приложения вместе со всеми его зависимостями, библиотеками и настройками в изолированную среду — контейнер. В отличие от виртуальных машин, контейнеры не создают отдельную ОС, а используют ядро основной системы, благодаря чему работают быстрее и потребляют меньше ресурсов. Ключевое преимущество контейнеров — их предсказуемость: приложение внутри контейнера будет вести себя одинаково, где бы его ни запустили — на локальном компьютере, тестовом стенде или в облачном кластере.

Эта технология кардинально упростила процессы разработки, тестирования и деплоя ПО. Она стала основой для микросервисных архитектур, где каждый компонент системы может быть развёрнут в собственном контейнере. Такой подход обеспечивает гибкость, лёгкое масштабирование отдельных сервисов и исключает конфликты между разными версиями библиотек и настроек.

Стандартом в области контейнеризации стал Docker, предлагающий удобные инструменты для создания, управления и запуска контейнеров. Его использование упрощает DevOps-процессы, автоматизирует сборку образов и гарантирует идентичность сред на всех этапах — от разработки до промышленной эксплуатации. А с помощью Docker Compose можно описывать сложные мультисервисные приложения в виде конфигурационных файлов и разворачивать их одной командой, что особенно полезно при проектировании и отладке распределённых систем.

## **Кеширование и его роль в архитектуре**

Кеширование — это технология оптимизации производительности, при которой часто используемые данные сохраняются в быстродоступном хранилище для мгновенного доступа. Этот подход позволяет значительно снизить нагрузку на основные системы хранения и ускорить обработку запросов, особенно при работе с относительно статичными, но востребованными данными. Реализации кеша могут быть различными — от клиентских решений (например, браузерного кеша) до серверных систем, чаще всего использующих оперативную память для максимального быстродействия.

Типичные сценарии применения включают:

* Кеширование результатов сложных запросов к базам данных
* Сохранение ответов внешних API-сервисов
* Хранение промежуточных результатов вычислений
* Буферизация часто запрашиваемого контента

Redis (Remote Dictionary Server) — одно из наиболее эффективных решений для кеширования, представляющее собой хранилище данных в оперативной памяти. Его ключевые преимущества:

1. Высокая производительность — обработка до сотен тысяч операций в секунду
2. Гибкость структур данных — поддержка строк, хешей, списков, множеств и др.
3. Дополнительные функции:
   * Автоматическое удаление данных по истечении срока (TTL)
   * Публикация/подписка (Pub/Sub)
   * Транзакции и атомарные операции
4. Масштабируемость — поддержка кластеризации и репликации

Области применения Redis выходят далеко за рамки простого кеширования:

* Управление пользовательскими сессиями
* Хранение временных токенов и ключей доступа
  + Реализация очередей задач
  + Системы реального времени (чаты, уведомления)
  + Геопространственные данные
  + Ограничение частоты запросов (rate limiting)

Благодаря своей архитектуре, Redis обеспечивает время отклика на уровне миллисекунд, что делает его незаменимым компонентом в высоконагруженных системах, где критически важна скорость доступа к данным. Его использование особенно эффективно в микросервисных архитектурах, распределенных системах и решениях, требующих обработки данных в реальном времени.

## **Event-driven архитектура**

Событийно-ориентированная архитектура представляет собой современный подход к проектированию программных систем, где взаимодействие между компонентами осуществляется через генерацию и обработку событий. В отличие от традиционных синхронных моделей, где компоненты напрямую вызывают методы друг друга, EDA предполагает асинхронную коммуникацию через специальную шину событий.

Основу этой архитектуры составляют события - сообщения, фиксирующие факт изменения состояния системы или произошедшего действия. Например, "пользователь совершил покупку" или "доставка подтверждена". Эти события публикуются в общее пространство, откуда их могут получать все заинтересованные подсистемы. Такой механизм обеспечивает принципиально новый уровень гибкости и масштабируемости программных решений.

Главное преимущество событийного подхода заключается в радикальном снижении связанности компонентов системы. Поскольку модули взаимодействуют только через события, а не через прямые вызовы, система становится значительно более модульной. Добавление нового функционала часто сводится к простой подписке на соответствующие события без необходимости модификации существующего кода. Это особенно ценно в условиях постоянного развития и масштабирования сложных распределенных систем.

Еще одним важным аспектом EDA является повышение отказоустойчивости. Благодаря асинхронной природе взаимодействия временная недоступность одного из обработчиков не приводит к остановке всей системы. События могут накапливаться в очередях и обрабатываться по мере восстановления сервисов. Кроме того, событийная модель естественным образом поддерживает горизонтальное масштабирование, позволяя легко добавлять новые экземпляры обработчиков для распределения нагрузки.

Для технической реализации EDA используются специализированные брокеры сообщений, такие как RabbitMQ, Apache Kafka или AWS SQS. Эти системы обеспечивают надежную доставку сообщений, поддерживают различные модели маршрутизации и предлагают механизмы для управления жизненным циклом событий. RabbitMQ, в частности, реализует протокол AMQP и предоставляет богатые возможности по организации очередей, что делает его популярным выбором для построения событийно-ориентированных систем.

Области применения EDA чрезвычайно широки. Эта архитектура особенно эффективна в системах с высокой степенью параллелизма, где требуется оперативная реакция на многочисленные события. Типичные примеры включают электронную коммерцию (обработка заказов, платежей и доставки), финансовые системы (мониторинг транзакций), IoT-платформы (обработка данных с устройств) и многие другие сценарии, где важны гибкость, масштабируемость и отзывчивость системы.

## **Сочетание подходов в архитектуре**

Современные программные системы все чаще используют комплексный подход, гармонично сочетая различные технологические решения. Контейнерные технологии, такие как Docker, создают стабильную и воспроизводимую среду выполнения для каждого компонента системы. Механизмы кеширования, реализованные через Redis, значительно повышают скорость доступа к часто используемым данным, одновременно уменьшая нагрузку на основные хранилища информации.

Для организации взаимодействия между сервисами все шире применяется асинхронная модель обмена сообщениями, где такие инструменты как RabbitMQ выступают в роли надежного посредника. Такой подход обеспечивает свободное масштабирование отдельных компонентов и повышает общую отказоустойчивость системы.

Сочетание этих технологий образует мощный фундамент для создания распределенных приложений, отличающихся высокой производительностью, надежностью и способностью адаптироваться к растущим нагрузкам. Каждый из элементов этого технологического стека дополняет другие, позволяя разработчикам сосредоточиться на бизнес-логике, не беспокоясь о проблемах инфраструктурного уровня. В результате получаются гибкие, масштабируемые и легко поддерживаемые программные решения, способные эффективно работать в современных условиях высоких нагрузок и постоянно меняющихся требований.

# **Проектирование архитектуры**

## **Общая архитектура**

Система реализована на основе микросервисной архитектуры и включает следующие ключевые компоненты:

1. Сервер на Java, представляющий REST API интерфейс\
2. Хранилище данных на PostgreSQL
3. Кеширующий механизм, работающий по алгоритму LRU
4. Асинхронная система обмена сообщениями через RabbitMQ по схеме «Поставщик-Потребитель»
5. Все сервисы развернуты в изолированных контейнерах Docker с оркестрацией через docker-compose

Предложенная архитектура обеспечивает ряд ключевых преимуществ для разработки и эксплуатации программного решения. В первую очередь, применение микросервисного подхода придает системе повышенную гибкость и масштабируемость, позволяя независимо развивать и масштабировать отдельные компоненты в соответствии с изменяющимися требованиями.

Использование реляционной СУБД PostgreSQL гарантирует надежное и структурированное хранение данных с поддержкой транзакционной целостности и сложных запросов. Многоуровневая система кеширования, реализованная с применением современных алгоритмов, значительно повышает производительность системы за счет сокращения времени доступа к часто запрашиваемым данным.

Асинхронная модель обработки запросов через брокера сообщений RabbitMQ обеспечивает эффективное распределение нагрузки и повышает отказоустойчивость системы. Наконец, контейнеризация всех компонентов с помощью Docker упрощает процесс развертывания и гарантирует согласованность рабочих окружений на всех этапах разработки и эксплуатации, от локальных сред до промышленных серверов.

## **Функциональность приложения**

Разработанное решение предоставляет следующий набор функций:

1. Веб-интерфейс REST API
   * Доступен по эндпоинту /api/v1/get
   * Обрабатывает GET-запросы, фиксируя каждый запрос в системе
   * Возвращает полную историю всех выполненных запросов
2. Механизм хранения данных
   * Вся информация о запросах сохраняется в реляционной БД PostgreSQL
   * Обеспечивается надежное и структурированное хранение истории обращений
3. Оптимизация производительности
   * Реализован кеширующий механизм LRU (Least Recently Used)
   * Значительно снижается нагрузка на основную базу данных
   * Ускоряется обработка повторяющихся запросов
4. Асинхронная обработка событий
   * Реализована модель Producer-Consumer через брокера сообщений
   * Продюсер (Producer) генерирует события:
     + Фиксация входящих запросов
     + Создание фоновых задач
     + Генерация системных уведомлений
   * Консьюмер (Consumer) обрабатывает события:
     + Запись логов
     + Отправка уведомлений
     + Выполнение фоновых операций
5. Особенности реализации
   * Все компоненты работают асинхронно
   * Обеспечивается высокая отказоустойчивость системы
   * Поддерживается масштабируемость под нагрузку

**Файл docker-compose.yaml**

**Изображение выглядит как текст, снимок экрана, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.**

Рисунок 1 – Файл docker-compose.yaml

## **Стратегия кеширования**

В системе используется кеширующий механизм типа LRU (Least Recently Used) с фиксированным размером, который автоматически удаляет наиболее давно неиспользуемые данные при достижении лимита. Реализация построена на базе LinkedHashMap с переопределением метода removeEldestEntry:

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2 – Пример механизма LRU

Преимущества кеширования:

* Сокращает количество прямых обращений к базе данных
* Увеличивает скорость отклика для повторяющихся запросов
* Разгружает основной сервер приложения

## **Асинхронная обработка через Producer-Consumer**

Система использует асинхронную модель обработки событий:

Генератор событий (Producer):

* Интегрирован в API-сервер
* Создает сообщения о событиях (запросы пользователей, системные действия)
* Отправляет события в очередь сообщений (RabbitMQ)

Обработчик событий (Customer):

* Постоянно мониторит очередь сообщений
* Выполняет обработку полученных событий (логирование, обновление данных, интеграция)

Архитектурные преимущества:

* Возможность горизонтального масштабирования
* Изоляция ресурсоемких операций от основного потока выполнения
* Поддержка асинхронной обработки фоновых задач

## **Бизнес-логика**

1. Пользователь инициирует GET-запрос к API

* Система проверяет наличие данных в кеше:
* При наличии - мгновенно возвращает кешированный результат

1. При отсутствии - запрашивает данные из БД, сохраняет в кеш и возвращает
2. Параллельно генерируется событие о запросе (для логирования/анализа)
3. Событие помещается в очередь для последующей обработки
4. Фоновый обработчик получает событие и выполняет соответствующие действия

# **Реализация**

Приложение разработано с использованием современных подходов микросервисной архитектуры, где каждый сервис отвечает за определенную бизнес-функцию. Все компоненты системы работают в контейнерах Docker, что обеспечивает их изолированность и простоту развертывания. В основе архитектуры лежат принципы CQRS (разделение команд и запросов) и Event Sourcing (хранение истории изменений).

## **Структура проекта**

Проект организован как единый репозиторий (монорепозиторий), содержащий несколько независимых модулей:

1. Сервис пользователей (user-service):
   * Управление учетными записями (создание, чтение, обновление, удаление)
   * Кеширование данных с помощью Redis
   * Сбор метрик производительности
2. Сервис уведомлений (notification-service):
   * Обработка системных событий
   * Отправка оповещений
3. Сервис балансов (balance-service):
   * Управление финансовыми операциями
   * Реализация CQRS и Event Sourcing
4. Общие компоненты (shared-contract):
   * Стандартные модели данных
   * Определения событий
   * Метрики
5. Инфраструктура (dockercompose):
   * Конфигурация для развертывания
   * Базы данных (PostgreSQL)
   * Брокер сообщений (RabbitMQ)
   * Системы мониторинга (Prometheus, Grafana)

Каждый модуль представляет собой самостоятельное Spring Boot-приложение на Java.

## **Реализация компонентов**

#### Пользовательский сервис (user-service)

В основе user-service лежит многослойная архитектура, обеспечивающая полный цикл операций с пользовательскими данными через REST API. На верхнем уровне находится UserRestApiController, который принимает HTTP-запросы и делегирует их обработку сервисному слою UserService, где сосредоточена основная бизнес-логика. Для обеспечения высокой производительности реализован механизм кеширования по стратегии "cache-aside" с использованием Redis в качестве хранилища кеша с временем жизни записей 60 секунд. При получении запроса система сначала проверяет наличие данных в кеше, и только в случае промаха обращается к основной базе данных, после чего сохраняет результат в Redis для последующих обращений. Все операции модификации данных автоматически обновляют соответствующие записи в кеше.

#### Система мониторинга и аналитики

Для комплексного наблюдения за состоянием системы реализован стек мониторинга на базе Prometheus, Grafana и Alertmanager. Spring Actuator в сочетании с Micrometer предоставляет детальные метрики работы приложения, включая количество созданных пользователей и время выполнения ключевых запросов, которые доступны через эндпоинт /actuator/prometheus. Grafana визуализирует эти данные, отображая такие важные показатели как загрузка процессора, динамика регистраций пользователей и состояние очередей сообщений. Alertmanager обеспечивает оперативное реагирование на критические ситуации, такие как превышение пороговых значений нагрузки или аномальное время отклика, с возможностью тестирования системы оповещений через SMTP-сервер Mailpit.

#### Механизм асинхронной обработки событий

Взаимодействие между сервисами организовано через брокера сообщений RabbitMQ по модели «producer/consumer». UserService выступает в роли producer событий, публикуя через UserEventPublisher уведомления о значимых действиях, таких как создание новых пользователей. Сервис уведомлений notification-service, являясь consumer, асинхронно обрабатывает эти события, выполняя соответствующие действия - логирование, отправку уведомлений и другие фоновые операции. Такой подход обеспечивает слабую связанность компонентов системы и повышает общую отказоустойчивость.

#### CQRS и Event Sourcing в balance-service

Сервис управления балансами (balance-service) построен с использованием архитектурных паттернов CQRS и Event Sourcing через фреймворк Axon. Операции изменения состояния (создание баланса, пополнение и списание средств) выполняются через отдельные команды, каждая из которых генерирует соответствующее событие. Все события сохраняются в Event Store, формируя полную и надежную историю изменений. Для чтения данных используются проекции, преобразующие поток событий в удобные для отображения модели - текущее состояние баланса и историю операций. Механизм снапшотов, создающий снимки состояния после каждых 10 событий, значительно ускоряет процесс восстановления данных. Функциональность четко разделена между BalanceCommandController для операций изменения состояния и BalanceQueryController для запросов информации, что обеспечивает оптимальное распределение нагрузки и масштабируемость системы.

## **Ссылка на GitHub**

* <https://github.com/woodwyyn/architecture>

# **Тестирование**

## **Redis**

В ходе тестирования проверяется механизм кэширования данных в Redis. Сначала в системе создаётся новый пользователь. Затем выполняется два последовательных запроса его данных по UUID, чтобы продемонстрировать разницу между получением информации из базы данных и из кэша.

При первом запросе система обнаруживает, что запрашиваемый пользователь отсутствует в Redis. В этом случае данные извлекаются напрямую из базы данных, после чего сохраняются в кэш для ускорения последующих обращений.

При втором запросе те же данные уже находятся в Redis, поэтому система возвращает их мгновенно, без дополнительных запросов к БД. Это наглядно показывает эффективность кэширования: первый запрос выполняется дольше из-за необходимости загрузки из БД, а второй отрабатывает значительно быстрее благодаря Redis.

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 3 – Запрос создания пользователя

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 4 – Ответ при создании пользователя

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 5 – Получение пользователя по UUID (запрос-ответ)



Рисунок 6 – Логи приложения

## **RabbitMQ**

При работе с RabbitMQ мы можем убедиться в корректной асинхронной коммуникации между микросервисами. Для этого создадим нового пользователя через Swagger-интерфейс, отправив POST-запрос на эндпоинт /users с данными пользователя. В результате в логах сервисов мы должны увидеть два важных события: сначала запись о создании пользователя в user-service, а затем соответствующее обработанное событие в notification-service.

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 7 – Запрос создания пользователя через Swagger



Рисунок 8 – Лог в notification-service

## **Grafana**

В Grafana мы разработали информативную панель мониторинга, которая наглядно отображает динамику двух ключевых метрик системы: уровень загрузки процессора и количество сообщений, ожидающих обработки в очередях RabbitMQ.

Изображение выглядит как снимок экрана, График

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 9 – Дашборд Grafana с отображением метрик производительности системы

## **Prometheus**

Для проверки работы системы мониторинга мы провели нагрузочное тестирование: через Swagger интерфейс было создано 10 тестовых пользователей. В результате этого теста мы наблюдали срабатывание системы алертинга - сначала алерт перешел в состояние ожидания (Pending), а через заданный интервал времени (1 минуту) подтвердил проблему, перейдя в статус Firing. Это демонстрирует корректную работу всего цикла мониторинга - от сбора метрик до оповещения о проблемах.

Изображение выглядит как текст, программное обеспечение, число, веб-страница

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 10 – Создание 10 пользователя

Изображение выглядит как снимок экрана, текст, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 11 – Алерт в состоянии Pending в интерфейсе Prometheus

Изображение выглядит как снимок экрана, текст, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 12 – Алерт в состоянии Firing в интерфейсе Prometheus

## **Mailpit**

Alertmanager автоматически отправляет два типа email-уведомлений при мониторинге аномальной активности: уведомление о срабатывании алерта (Firing) и уведомление о разрешении проблемы (Resolved)

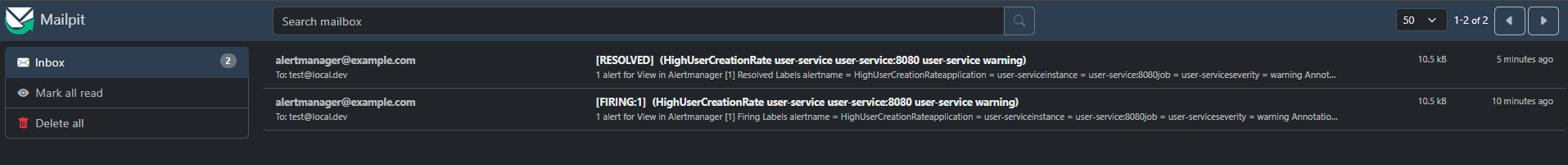


Рисунок 13 – Письма с уведомлением о срабатывании алерта в Mailpit

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 14 – Письмо о разрешении инцидента (Firing) в Mailpit

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 15 – Письмо о разрешении инцидента (Resolved) в Mailpit

## **CQRS и Event Sourcing**

Мы протестировали систему на основе CQRS и Event Sourcing с Axon Framework. После подключения к Axon Server создали баланс пользователя, выполнили несколько операций пополнения и списания. Read-модель точно отобразила итоговый баланс, а запрос истории показал все транзакции с детализацией по сумме и времени. Это подтвердило преимущества подхода - полная воспроизводимость всех изменений системы.

Изображение выглядит как снимок экрана, текст, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 16 – Подключение приложения к Axon Server через порт 8024

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 17 – Создание баланса (запрос-ответ)

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 18 – Успешное зачисление средств на баланс

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 19–20 – Успешное списание средств с баланса

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 21 – Ответ системы с текущим состоянием баланса

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 22 – Полная история событий баланса

# **Заключение**

В рамках курсового проекта была успешно реализована распределенная система, построенная на базе современных микросервисных принципов и технологий. Основной акцент делался на создании надежной и масштабируемой архитектуры, соответствующей промышленным стандартам разработки.

Ключевым достижением стало внедрение паттернов CQRS и Event Sourcing в модуле управления балансами пользователей. Это решение позволило эффективно разделить операции записи и чтения данных, обеспечив высокую производительность системы даже при значительных нагрузках. Event Sourcing предоставил уникальную возможность полного аудита всех изменений состояния системы через журнал событий, что особенно важно для финансовых операций.

Для обеспечения высокой доступности и отзывчивости системы была реализована многоуровневая архитектура кэширования на основе KeyDB, что позволило существенно снизить нагрузку на основную базу данных. Асинхронная коммуникация между сервисами через RabbitMQ обеспечила надежность и отказоустойчивость взаимодействия между компонентами системы.

Особое внимание уделялось вопросам мониторинга и оперативного реагирования на инциденты. Интеграция стека Prometheus, Grafana и Alertmanager позволила создать комплексную систему наблюдения за состоянием всех компонентов в реальном времени с возможностью автоматического оповещения о критических ситуациях.

Все компоненты системы были упакованы в Docker-контейнеры, что обеспечило простоту развертывания, масштабирования и поддержания идентичности сред на всех этапах жизненного цикла приложения.

Практическая значимость работы заключается в успешной демонстрации применения современных архитектурных подходов и технологий, которые широко используются в промышленной разработке. Реализованное решение сочетает в себе все преимущества микросервисной архитектуры, включая модульность, масштабируемость и отказоустойчивость, что делает его применимым для решения реальных задач в предметной области.

# **Список литературы**

1. Docker Inc. Docker Documentation. — <https://docs.docker.com>
2. Redis Labs. Redis Documentation. — <https://redis.io/docs/>
3. RabbitMQ. RabbitMQ Documentation. — <https://www.rabbitmq.com/docs>
4. Prometheus Authors. Prometheus Documentation. — <https://prometheus.io/docs/>
5. Grafana Labs. Grafana Documentation. — <https://grafana.com/docs/>