*Министерство науки и высшего образование РФ*
*Федеральное государственное автономное*
*образовательное учреждение высшего образования*
**«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**
Институт космических и информационных технологий
Кафедра «Вычислительная техника»

УТВЕРЖДАЮ
Заведующий кафедрой
_____________ О.В. Непомнящий
подпись

« _____» ______________ 2024 г.

**МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ**

Организация взаимодействия микросервисов в задачах разработки веб-приложений

09.04.01 Информатика и вычислительная техника

09.04.01.4 Технология разработки программного обеспечения

| | | | |
|---|---|---|---|
| Руководитель | ________<br>подпись, дата | доцент, канд. техн. наук | М.С. Медведев |
| Выпускник | ________<br>подпись, дата | | К.В. Ермош |
| Рецензент | ________<br>подпись, дата | доцент, канд. техн. наук | В.Г. Демин |
| Нормоконтролер | ________<br>подпись, дата | | М.С. Медведев |

Красноярск 2024

## РЕФЕРАТ

Выпускная квалификационная работа по теме «Организация взаимодействия микросервисов в задачах разработки веб-приложений» содержит 72 страницы текстового документа, 15 рисунков, 17 таблиц, 8 приложений и 8 использованных источников.

МИКРОСЕРВИСЫ, ВЕБ-ПРИЛОЖЕНИЕ, KAFKA, RABBITMQ, REST API, RPC, ВЗАИМОДЕЙСТВИЕ МИКРОСЕРВИСОВ, ПЕРЕДАЧА ДАННЫХ.

Цель работы – проектирование и разработка веб-приложения, основанного на принципах микросервисной архитектуры, а также в сравнение различных методов взаимодействия между сервисами.

Для достижения цели были поставлены следующие задачи: изучение принципов микросервисной архитектуры, анализ и сравнение микросервисной архитектуры с монолитной, проектирование и разработка каждого отдельного сервиса, и всей системы в целом, а также сравнение различных методов и технологий взаимодействия между микросервисами в различных сценариях использования.

Тема работы является актуальной, поскольку в наше время веб-приложения наиболее популярны, и все чаще для их разработки стала использоваться именно микросервисная архитектура.

В процессе выполнения работы было произведено исследование популярных в наше время архитектур для разработки, а также методов обмена данными между системами/сервисами. Было спроектировано и разработано веб-приложение на базе микросервисной архитектуры. На основе созданного приложения было проведено сравнение способов взаимодействия микросервисов по таким параметрам: время отклика, пропускная способность, нагрузка на процессор, потребление памяти.

**СОДЕРЖАНИЕ**

Введение ... 5
1 Анализ предметной области ... 6
1.1 Постановка задачи ... 6
1.2 Архитектурные решения ... 7
1.3 Монолитная архитектура ... 7
1.4 Микросервисная архитектура ... 8
1.5 Способы взаимодействия микросервисов ... 10
1.5.1 API ... 11
1.5.2 RPC ... 12
1.5.3 SOAP ... 13
1.5.4 Обмен сообщениями ... 14
1.5.5 Стриминг ... 16
1.6 Выводы по главе ... 17
2 Проектирование архитектуры и основных технических решений ... 18
2.1 Архитектура разрабатываемой системы ... 18
2.2 Выбор средств разработки ... 20
2.2.1 REST API ... 21
2.2.2 gRPC ... 22
2.2.3 RabbitMQ ... 23
2.2.4 Kafka ... 24
2.3 Разработка структуры базы данных ... 25
2.3.1 Сервис авторизации ... 25
2.3.2 Сервис товаров ... 26
2.3.3 Сервис заказов ... 27
2.3.4 Сервис уведомлений ... 28
2.3.5 Сервис отзывов ... 28
2.4 Выводы по главе ... 29
3 Разработка веб-приложения ... 30

3.1 API-шлюз ........................................................................................................ 30
3.2 Сервис авторизации......................................................................................... 32
3.3 Сервис уведомлений ....................................................................................... 34
3.4 Сервис заказов ................................................................................................ 36
3.5 Сервис товаров ............................................................................................... 37
3.6 Сервис отзывов ................................................................................................ 38
3.7 Развертывание системы ................................................................................. 39
3.8 Выводы по главе.............................................................................................. 40
4 Сравнение способов взаимодействия микросервисов между собой ............... 41
4.1 Время отклика.................................................................................................. 42
4.2 Пропускная способность................................................................................. 45
4.3 Нагрузка на процессор .................................................................................... 47
4.4 Потребление памяти........................................................................................ 50
4.5 Заключение к главе.......................................................................................... 52
Заключение ............................................................................................................ 55
Список использованных источников ................................................................. 56
ПРИЛОЖЕНИЕ А Класс RoutesConfig.java отвечающий за маршрутизацию запросов ................................................................................................................ 57
ПРИЛОЖЕНИЕ Б Класс AuthController.java...................................................... 59
ПРИЛОЖЕНИЕ В Класс конфигурации RabbitMq ............................................ 61
ПРИЛОЖЕНИЕ Г Dockerfile сервиса заказов .................................................... 63
ПРИЛОЖЕНИЕ Д Файл docker-compose.yml для развертывания всего приложения........................................................................................................... 64
ПРИЛОЖЕНИЕ Е Класс RestSender.java для отправки уведомлений при помощи Rest API .................................................................................................... 66
ПРИЛОЖЕНИЕ Ж Класс DataController.java для получения отправленного сообщения.............................................................................................................. 67
ПРИЛОЖЕНИЕ З Bash скрипт для отслеживания нагрузки на процессов ........ 68
ПРИЛОЖЕНИЕ И Отчет о проверке сервисом «Антиплагиат» ........................ 69

# ВВЕДЕНИЕ

В разработке веб-приложений сегодня можно выделить два основных подхода. Первый из них — это монолитная архитектура, где весь функционал приложения сосредоточен в одном большом приложении. Второй ключевой подход, заключается в использовании микросервисной архитектуры. Этот подход представляет собой декомпозицию приложения на небольшие, независимые компоненты, называемые микросервисами, каждый из которых занимается решением определенной задачи или функционала. Такой подход обеспечивает гибкость, масштабируемость и легкость поддержки системы.

Микросервисы стали неотъемлемой частью инструментария современных разработчиков веб-приложений [1]. Они позволяют создавать сложные и высоконагруженные системы, учитывая современные требования к производительности, надежности и масштабируемости. Однако за всеми преимуществами микросервисов стоят и вызовы.

Цель данной работы состоит в проектировании разработке веб-приложения, а также в исследовании и анализе организации взаимодействия между микросервисами в контексте веб-разработки. Будут рассмотрены различные аспекты этой проблемы, включая плюсы и минусы использования микросервисов, основные методы и технологии взаимодействия между ними, а также проанализируем различные подходы к организации взаимодействия и их применимость в конкретных сценариях разработки.

В конечном итоге, результаты работы помогут разработчикам лучше понимать проблемы и возможности, связанные с организацией взаимодействия микросервисов в веб-разработке, и принимать обоснованные решения при проектировании и развертывании современных веб-приложений.

# 1 Анализ предметной области

## 1.1 Постановка задачи

Цель данной исследовательской работы заключается в проектировании и разработке веб-приложения, основанного на принципах микросервисной архитектуры, а также в анализе различных методов взаимодействия между сервисами данной архитектуры. Дополнительно проводится сравнительный анализ микро-сервисной архитектуры с монолитной.

Реализация поставленной цели предполагает решение следующих задач:

– изучение основных принципов микросервисной архитектуры с целью понимания её преимуществ и особенностей;

– проведение анализа и выявление преимуществ микросервисной архитектуры в сравнении с монолитной архитектурой в контексте разработки веб-приложений;

– проектирование структуры и функционала веб-приложения на базе микро-сервисной архитектуры, включая определение компонентов и их взаимосвязей;

– разработка отдельных микросервисов, включая их функционал, интерфейсы и способы взаимодействия;

– создание серверной (бэкенд) части веб-приложения, включая реализацию серверной логики и базы данных для микросервисов;

– проведение анализа и сравнения различных методов и технологий взаимодействия между микрсервисами в различных сценариях использования.

Данная работа направлена на изучение и практическое применение современных подходов к разработке веб-приложений на основе микросервисной архитектуры, а также на выявление их преимуществ и недостатков по сравнению с традиционным монолитным подходом.

## 1.2 Архитектурные решения

Когда речь идет про веб-приложения, при выборе архитектурного решения есть два основных варианта: микросервисная архитектура или монолитная.

## 1.3 Монолитная архитектура

Монолитная архитектура в веб-приложениях представляет собой подход, при котором весь функционал приложения выполняется в одном целом, обычно в виде единого исполняемого файла или приложения [2]. Основные компоненты приложения, такие как интерфейс пользователя, бизнес-логика и доступ к данным, находятся внутри одного приложения и взаимодействуют напрямую.

Плюсы монолитной архитектуры:

1. Простота развертывания: так как все компоненты находятся в одном месте, развертывание приложения обычно происходит быстрее и проще.

2. Простота разработки: отсутствие сложной инфраструктуры облегчает процесс разработки и отладки приложения.

3. Удобство масштабирования в начале: при небольшом объеме пользовательского трафика монолитное приложение может обеспечить достаточную производительность без необходимости разделения на отдельные сервисы.

Минусы монолитной архитектуры:

1. Сложность поддержки: при увеличении размера приложения и добавлении новых функций может возникнуть сложность в поддержке и изменении кода из-за его единого монолитного характера.

2. Ограниченная масштабируемость: при достижении предела производительности или необходимости масштабирования отдельных компонентов приложения могут возникнуть сложности из-за их взаимосвязи внутри монолита.

3. Ограниченная гибкость: из-за единого характера приложения изменения в одной его части могут затронуть другие компоненты, что затрудняет изменения и внедрение новых технологий.

Таким образом, монолитная архитектура может быть хорошим выбором для простых приложений с невысокой нагрузкой и небольшими требованиями к масштабируемости и гибкости [2].

### 1.4 Микросервисная архитектура

Микросервисная архитектура в веб-приложениях представляет собой подход, при котором приложение разбивается на небольшие автономные сервисы, каждый из которых отвечает за отдельный функционал. Каждый сервис обычно разворачивается и масштабируется независимо [3].

Плюсы микросервисной архитектуры:

1. Гибкость и масштабируемость: благодаря разделению приложения на независимые сервисы, каждый из них можно масштабировать и обновлять независимо от остальных компонентов, что улучшает гибкость и масштабируемость приложения.

2. Легкость поддержки и разработки: каждый сервис обычно меньшего размера и имеет четко определенные границы ответственности, что облегчает поддержку и разработку приложения.

3. Технологическое разнообразие: разные сервисы могут быть написаны на разных языках программирования или использовать разные технологии, что позволяет выбирать наиболее подходящие инструменты для каждой конкретной задачи.

Минусы микросервисной архитектуры:

1. Сложность конфигурации и управления: управление большим количеством независимых сервисов требует наличия сложной инфраструктуры и механизмов управления, что может повлечь за собой дополнительные сложности и затраты.

2. Сетевая задержка: поскольку каждый сервис взаимодействует с другими через сеть, возникает задержка, которая может оказать влияние на общую производительность приложения.

3. Сложность тестирования: тестирование микросервисов может быть сложным из-за необходимости управления их взаимодействием и зависимостями [3].

Таким образом, микросервисная архитектура часто применяется в больших и сложных приложениях, где требуется высокая гибкость, масштабируемость и возможность использования различных технологий.

Если подвести итог касательно архитектурных решений, то можно выделить то, что главный плюс микросервисной архитектуры — это большая гибкость и масштабируемость. Разбиение приложения на отдельные сервисы позволяет разрабатывать и внедрять новый функционал быстрее и эффективнее, поскольку изменения в одном сервисе не затрагивают остальные. Это особенно важно для современных веб-приложений, которые предполагают высокую динамичность и быстрое внедрение новых возможностей. Использование микросервисной архитектуры, также может помочь ускорить разработку, так как работа над каждым сервисом ведется независимо и может выполняться разными людьми/командами в разные сроки. Каждый микросервис может быть масштабирован независимо в зависимости от его нагрузки, что обеспечивает более эффективное использование ресурсов и повышает надежность всей системы. Но также данная архитектура имеет много нюансов, которые могут вызвать проблемы, сюда входит, например, сложность построения инфраструктуры и сложность управления всем, могут (но не обязательно) возникнуть проблемы с безопасностью, так как приходится часто передавать данные между сервисами, соответственно растет потенциальный риск утечек информации, ну и тот факт, что данные разделены по разным сервисам (по разным БД скорее всего), делает процесс сборки и компоновки всей информации сложнее [4].

Что же касается монолитной архитектуры, то всех этих минусов можно избежать, но монолитная архитектуры удобна ровно до определенных размеров приложения, дальше его становится достаточно проблематично поддерживать, есть даже такой термин “Big Ball of Mud” (термин, используемый в контексте монолитной архитектуры, обозначающий ситуацию, когда приложение становится слишком сложным и тяжелым для поддержки и развития из-за того, что все его компоненты тесно связаны друг с другом и их сложно разделить на более мелкие и независимые части). Что касается масштабируемости, то там и вовсе есть свой предел.

## 1.5 Способы взаимодействия микросервисов

Способы взаимодействия микросервисов представляют собой ключевой аспект при разработке веб-приложений на основе микросервисной архитектуры. Если в монолитной архитектуре информация может просто передаваться между различными классами внутри приложения, то в микросервисах её надо отправлять в другие системы. Для обеспечения эффективной работы системы необходимо выбрать подходящие методы взаимодействия, которые будут соответствовать требованиям проекта и обеспечивать высокую производительность, надежность и масштабируемость [5]. В данном разделе будут рассмотрены основные способы взаимодействия микросервисов.

Всего можно выделить 2 основных направления взаимодействия микросервисов:

– Синхронное — это процесс обмена информацией, при котором отправитель ожидает ответа от получателя перед тем, как продолжить выполнение своей работы. В этом случае, когда микросервис **A** отправляет запрос микросервису **B**, он блокирует свою работу и ждет, пока микросервис B обработает запрос и вернет ответ. Таким образом, синхронное взаимодействие подразумевает прямую зависимость между отправителем и получателем, где отправитель блокируется до получения ответа от получателя. Этот тип

взаимодействия часто используется в случаях, когда ответ получается немедленно и без значительных задержек;

– Асинхронное — это способ обмена информацией, при котором отправитель не блокирует свою работу в ожидании ответа от получателя. Вместо этого, отправитель отправляет запрос получателю и продолжает свою работу без ожидания ответа. Получатель обрабатывает запрос асинхронно и отправляет ответ, когда он будет готов. В случае асинхронного взаимодействия, отправитель может продолжать выполнять другие задачи, не останавливаясь на ожидании ответа от получателя. Асинхронное взаимодействие широко используется в распределенных системах, где ответ может занять значительное время, например, когда требуется обработка больших объемов данных или, когда получатель не доступен в данный момент. Этот подход позволяет увеличить пропускную способность системы и повысить отказоустойчивость, так как отправитель не блокируется в ожидании ответа и может продолжать свою работу независимо от состояния получателя [6].

Рассмотрим несколько вариантов синхронного взаимодействия микросервисов.

### 1.5.1 API

API (Application programming interface) – это программный интерфейс, то есть описание способов взаимодействия одной компьютерной программы с другими посредство различных протоколов. API является представителем синхронного направления.

Синхронное взаимодействие микросервисов с использованием API происходит путем отправки HTTP-запросов (или каких-либо других) от одного сервиса к другому. Например, микросервис **А** может отправить HTTP-запрос на определенный URL-адрес микросервиса **В**, передавая при этом необходимые данные. Микросервис **В** обрабатывает запрос, выполняет необходимые операции

и возвращает HTTP-ответ с результатом обработки обратно микросервису **A**. При этом микросервис **A** блокируется и ожидает ответа от микросервиса **B**.

API является одним из наиболее распространенных подходов к взаимодействию между микросервисами в современных веб-приложениях благодаря своей простоте, гибкости и поддержке стандартных протоколов HTTP.

### 1.5.2 RPC

RPC (Remote Procedure Call) - это архитектурный стиль, также предназначенный для построения распределенных систем, основанных на веб-сервисах. В рамках RPC-архитектуры, каждый микросервис представляет собой набор удаленных процедур, к которым можно обращаться для выполнения определенных операций. RPC также как и REST является синхронным способом.

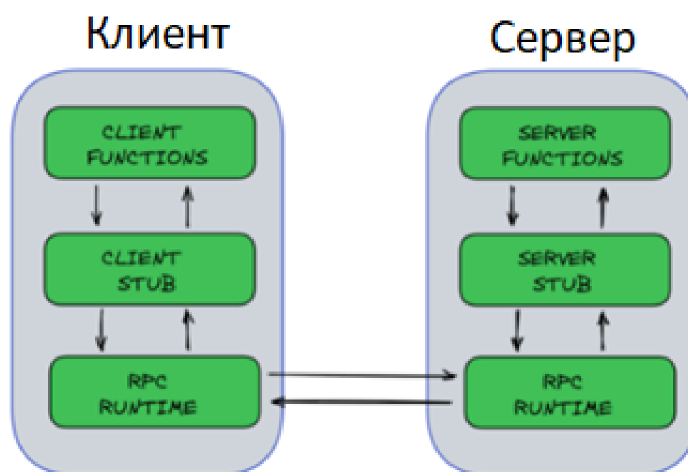



Рисунок 1 – Взаимодействия при помощи RPC

Синхронное взаимодействие микросервисов с использованием RPC происходит путем вызова удаленных процедур. Например, микросервис **A** может вызвать удаленную процедуру на микросервисе **B**, передавая при этом необходимые параметры. Микросервис **B** обрабатывает запрос и возвращает

результат обработки обратно микросервису **A**. При этом микросервис **A** также блокируется и ожидает завершения вызова удаленной процедуры.

Основные компоненты RPC включают в себя клиентский интерфейс, который обеспечивает вызов удаленных процедур, и серверный интерфейс, который обрабатывает эти вызовы и выполняет соответствующие операции. Популярными протоколами для реализации RPC являются gRPC, Apache Thrift и JSON-RPC.

RPC также является распространенным подходом к взаимодействию между микросервисами в веб-приложениях. Он обеспечивает высокую производительность и эффективное использование сетевых ресурсов благодаря компактным форматам данных и оптимизированным протоколам передачи данных.

### 1.5.3 SOAP

SOAP (Simple Object Access Protocol) - это протокол обмена сообщениями, который используется для создания распределенных систем, основанных на веб-сервисах. В рамках SOAP-архитектуры, каждый микросервис представляет собой веб-сервис, к которому можно обращаться для выполнения определенных операций.

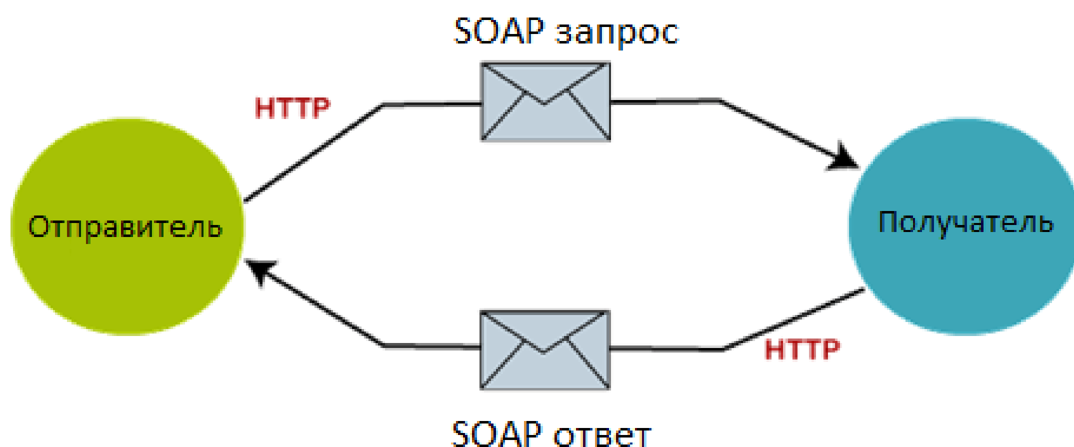



Рисунок 2 – Взаимодействие при помощи SOAP

Синхронное взаимодействие микросервисов с использованием SOAP происходит путем отправки SOAP-сообщений от одного сервиса к другому. Например, микросервис **A** может отправить SOAP-запрос на определенный эндпоинт микросервиса **B**, содержащий информацию о вызываемом методе и передаваемых параметрах. Микросервис **B** обрабатывает запрос, выполняет необходимые операции и возвращает SOAP-ответ с результатом обработки обратно микросервису **A**.

Основные компоненты SOAP включают в себя XML-схемы для определения структуры сообщений, WSDL (Web Services Description Language) для описания доступных операций и портов, а также SOAP-заголовки для передачи метаданных и дополнительной информации.

SOAP обеспечивает высокую надежность и безопасность взаимодействия между микросервисами благодаря использованию промышленных стандартов и протоколов, таких как WS-Security для обеспечения конфиденциальности и целостности данных.

Хотя SOAP является менее распространенным в сравнении с REST и RPC, он по-прежнему используется в различных сценариях, где требуется высокая надежность и безопасность при обмене данными между микросервисами.

### 1.5.4 Обмен сообщениями

Обмен сообщениями или месседжинг - это архитектурный подход, используемый для построения распределенных систем, основанный на передаче сообщений между микросервисами. В рамках месседжинга, каждый микросервис может публиковать сообщения в определенные темы (в таком случае сервис называется Producer) или очереди, а другие микросервисы могут подписываться на эти темы или очереди для получения сообщений (в этом случае его называют Consumer). Таким образом месседжинг – это основной представитель асинхронного подхода в общение между микросервисами (и не только).

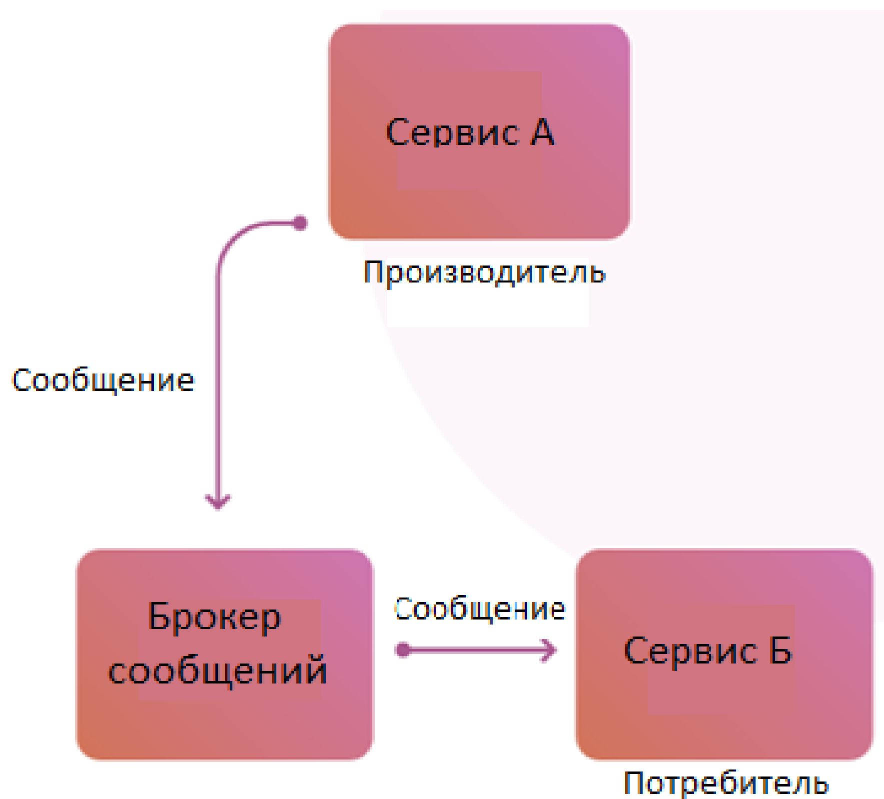



Рисунок 3 – Общение при помощи обмена сообщениями

Взаимодействие микросервисов с использованием месседжинга происходит асинхронно. Например, микросервис A может опубликовать сообщение в очередь, а микросервис B, который подписан на эту очередь, получит сообщение и обработает его в соответствии с логикой своего приложения.

Основные компоненты месседжинга включают в себя брокера сообщений (например, ZeroMq, ActiveMq, RabbitMQ), который отвечает за хранение и маршрутизацию сообщений, а также клиентские библиотеки или адаптеры, которые используются микросервисами для отправки и получения сообщений.

Месседжинг обеспечивает гибкость и масштабируемость взаимодействия между микросервисами, позволяя им обмениваться данными независимо от своего текущего состояния или доступности других сервисов. Кроме того, месседжинг позволяет реализовать асинхронные процессы и обеспечивает отказоустойчивость путем сохранения сообщений даже в случае временной недоступности получателя.

Хотя месседжинг является более сложным подходом к взаимодействию между микросервисами, он широко используется в больших и распределенных системах, где требуется высокая гибкость, масштабируемость и надежность обмена данными.

### 1.5.5 Стриминг

Стриминг - это архитектурный подход, используемый для передачи данных в реальном времени между микросервисами в распределенных системах. В рамках стриминга, данные передаются как непрерывный поток событий или сообщений от одного микросервиса к другому.

Взаимодействие микросервисов с использованием стриминга происходит асинхронно и непрерывно. Например, микросервис **A** может создать стрим данных и начать передавать его микросервису **B**. Микросервис **B** может подписаться на этот стрим и получать данные в режиме реального времени, обрабатывая их по мере их поступления.

Основные компоненты стриминга включают в себя стриминговые платформы (например, Apache Kafka, Apache Flink, Apache Pulsar), которые отвечают за управление потоками данных, а также клиентские библиотеки или адаптеры, которые используются микросервисами для создания и подписки на стримы данных.

Стриминг обеспечивает возможность обработки данных в реальном времени, что особенно важно для приложений, требующих мгновенного реагирования на события или операции в реальном времени. Кроме того, стриминг позволяет обрабатывать большие объемы данных с минимальной задержкой и обеспечивает отказоустойчивость путем репликации данных и обработки ошибок в реальном времени.

Хотя стриминг является более сложным подходом к взаимодействию между микросервисами, он широко используется в приложениях, требующих

обработки данных в реальном времени, таких как системы мониторинга, аналитики, обработки потоков событий и т. д.

Стриминг по своей сути очень напоминает месседжинг, но их основное отличие заключается в том, что первый ориентирован на обработку данных в режиме стриминга, предоставляя возможность непрерывной передачи и обработки данных в реальном времени. В отличие от мессенджинга, который чаще используется для передачи сообщений между компонентами системы, стриминг позволяет обрабатывать большие потоки данных, эффективно работая с высокими нагрузками и обеспечивая надежную и масштабируемую архитектуру для обработки потоков данных.

## 1.6 Выводы по главе

В заключении стоит подчеркнуть, что микросервисная архитектура стала популярным выбором для разработки современных приложений благодаря ряду преимуществ и большой вариативностью, благодаря чему люди все чаще выбирают её при разработке своих приложений.

В главе был проведен обзор некоторых подходы взаимодействия, но по мимо этих, есть еще много других, мы рассмотрели только основные и самые популярные. Каждый из этих подходов имеет свои особенности и может быть выбран в зависимости от конкретных требований проекта и бизнес-целей. Важно выбирать подход, который наилучшим образом соответствует целям проекта и обеспечивает эффективное взаимодействие между компонентами системы.

# 2 Проектирование архитектуры и основных технических решений

## 2.1 Архитектура разрабатываемой системы

В качестве разрабатываемой системы было выбрано веб-приложение для интернет-магазина одежды. Для его создания будет использоваться клиент-серверная модель, где основная часть функционала находится на сервере. Это обеспечивает высокий уровень безопасности и защиты информации, так как на сервере происходит обработка основных операций с данными. Также такой подход способствует повышению совместимости различных программных продуктов и платформ, что упрощает их интеграцию и снижает требования к ним.

Основные функциональные возможности приложения включают в себя авторизацию пользователей, возможность оформления заказов, получение уведомлений и оставление отзывов. Эти функции обеспечивают удобство и функциональность для пользователей, делая процесс покупок и взаимодействия с магазином более эффективным и приятным.

Архитектура сервера будет микросервисной, также нужна будет база данных. Было решено использовать паттерн “Database per service”, то есть когда у каждого сервиса своя отдельная база данных. Этот способ удобен тем, что приложение будет иметь слабую связанность сервисов, что облегчает работу с их модификацией, а также при надобности, каждый сервис может использовать различные базы данных (например, для некоторых вариантов возможно NoSql базы буду удобнее, чем обычный Sql), но из минусов данного способа – сложность управления данными. Очень сложно добыть информацию, куски которой разбросаны по разным сервисам, также сложнее организовывать какие-либо бизнес-транзакции, которые охватывают несколько сервисов.

Для доступа пользователя к различным сервисам будет использоваться API Gateway. Это серверный прокси, который предоставляет интерфейс для клиентов (приложений, устройств, пользователей) для доступа к веб-сервисам или

микросервисам. Он выполняет функцию шлюза, контролируя и маршрутизируя запросы от клиентов к соответствующим службам и обеспечивает централизованное управление и безопасность для всех запросов, проходящих через ваше веб-приложение.

Схема приложения представлена на рисунке 4.

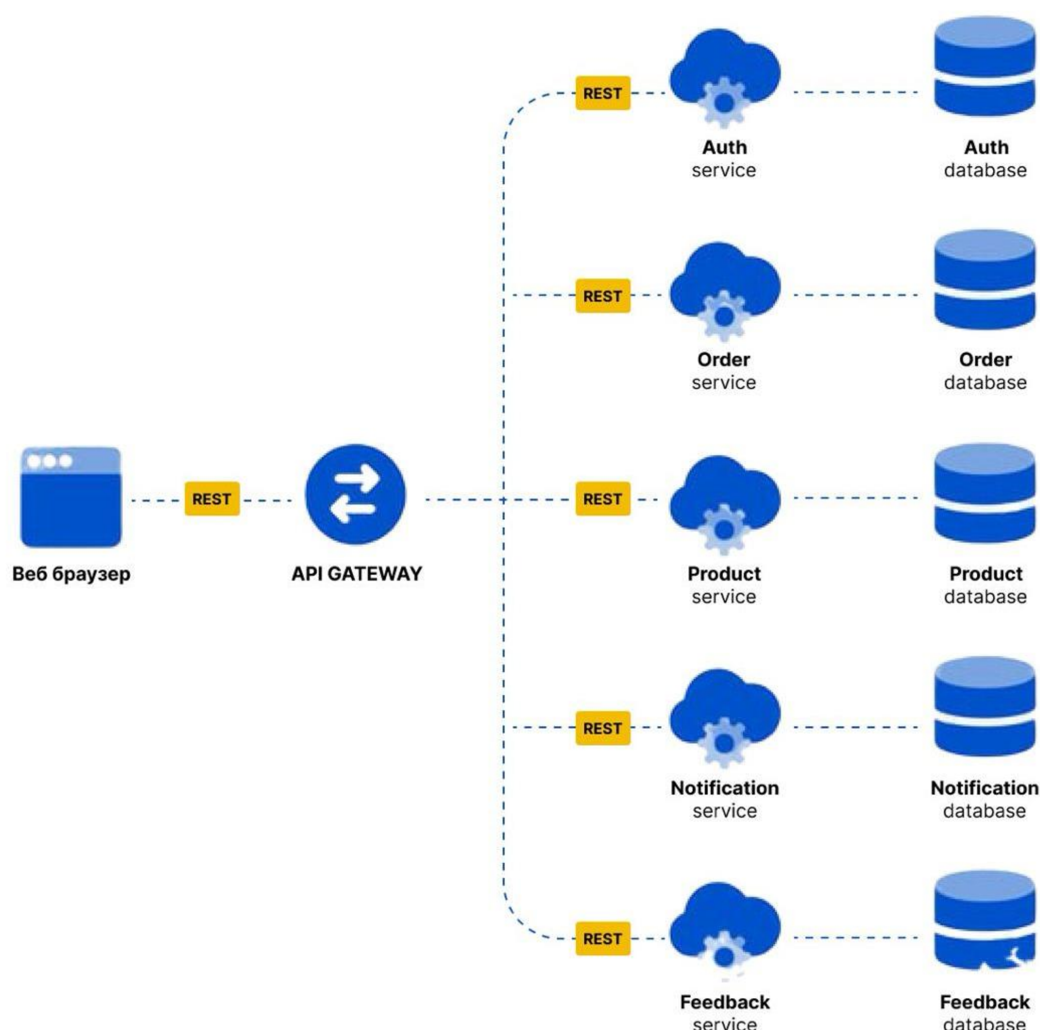



Рисунок 4 – Архитектурная схема приложения

Из Рисунка 4 видно, что в приложении будет пять основных сервисов. Сервис авторизации, сервис продуктов, сервис заказов, сервис уведомлений и сервис отзывов.

## 2.2 Выбор средств разработки

Для разработки веб-приложения интернет-магазина одежды было принято решение использовать язык программирования Java с фреймворком Spring. Этот выбор был обоснован рядом преимуществ:

1. Широкая популярность и экосистема: Java является одним из самых распространенных языков программирования, что обеспечивает большое количество инструментов, библиотек и ресурсов для разработчиков. Фреймворк Spring, в свою очередь, предоставляет мощные инструменты для построения масштабируемых и надежных веб-приложений.

2. Простота разработки: Java и Spring предоставляют высокий уровень абстракции и инструменты для решения типовых задач разработки, таких как управление зависимостями, обработка HTTP-запросов и взаимодействие с базой данных, что значительно упрощает разработку и поддержку приложения.

3. Надежность и производительность: Java обладает высокой степенью стабильности и производительности, что делает его привлекательным выбором для разработки критически важных приложений, таких как интернет-магазины. Spring, в свою очередь, предоставляет множество инструментов для оптимизации производительности и обеспечения надежности приложения.

4. Широкая поддержка сообщества: Java и Spring активно поддерживаются большим сообществом разработчиков и имеют обширную документацию, туториалы и форумы поддержки, что обеспечивает доступ к обширным ресурсам для решения возникающих проблем и вопросов.

В качестве системы управления базами данных была выбрана PostgreSQL. Этот выбор был обусловлен следующими причинами:

1. Надежность и стабильность: PostgreSQL является одной из самых надежных и стабильных систем управления базами данных, обеспечивая высокий уровень целостности данных и отказоустойчивости.

2. Мощный функционал: PostgreSQL предоставляет богатый набор функций и возможностей, включая поддержку сложных запросов, транзакций,

индексацию и расширяемость, что делает его привлекательным выбором для разработки интернет-магазина с разнообразными требованиями к данным.

3. Открытый исходный код: PostgreSQL распространяется под лицензией открытого исходного кода, что обеспечивает свободный доступ к исходному коду, активное сообщество разработчиков и возможность участия в развитии и улучшении продукта.

Таким образом, выбор Java с использованием Spring для разработки веб-приложения интернет-магазина одежды, а также PostgreSQL в качестве системы управления базами данных был сделан с учетом требований к надежности, производительности и удобству разработки.

В качестве взаимодействия микросервисов я выбрал такие способы: REST API и gPRC в качестве представителей синхронного взаимодействия, RabbitMQ для обмена сообщениями и Kafka для стриминга.

### 2.2.1 REST API

REST API — это архитектурный подход, который устанавливает ограничения для API: как они должны быть устроены и какие функции поддерживать. Это позволяет стандартизировать работу программных интерфейсов, сделать их более удобными и производительными. По факту REST это не отдельный протокол взаимодействия, а просто свод рекомендаций, которые в свою очередь применяются при разработке API. Первый способ общения в данном случае – это HTTP протокол, к которому применяется ряд всеобщих правил для стандартизации, удобства и лаконичности кода.

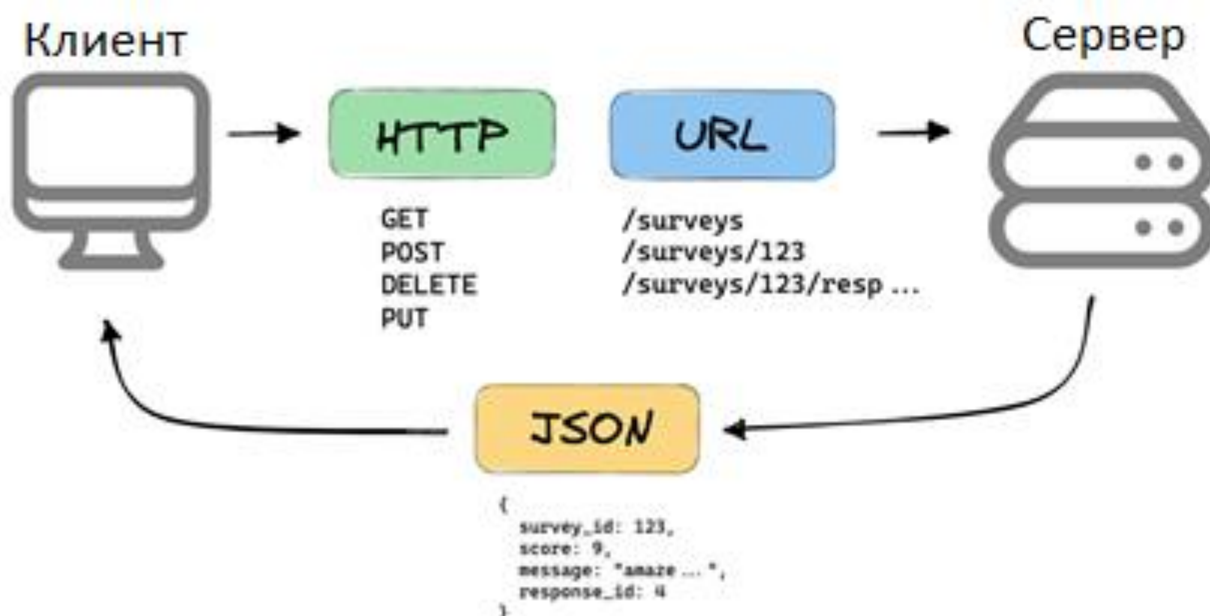



Рисунок 5 – REST API взаимодействие

Сам обмен будет проходить следующим образом. Каждый микросервис представляет собой ресурс, к которому можно обращаться по определенному URL-адресу (endpoint) и выполнять операции CRUD (Create, Read, Update, Delete) с использованием стандартных HTTP-методов (GET, POST, PUT, DELETE). То есть с клиента будет посылаться запрос на какой-либо сервис, посредством HTTP запроса, который будет содержать в себе какие-то данные, дальше эти данные обрабатываются каким-то образом и формируется ответ, который отправляется в виде JSON обратно на клиент.

### 2.2.2 gRPC

gRPC — это высокопроизводительный фреймворк разработанный компанией Google для вызов удаленных процедур (RPC), работает поверх HTTP/2. Он простой в использовании, отлично подходит для создания распределенных систем (микросервисов) и API. Имеет встроенную поддержку для балансировки нагрузки, трассировки, аутентификации и проверки жизнеспособности сервисов. Высокая производительность достигается за счет использования протокола HTTP/2 и Protocol Buffers. Protobuf - формат

сериализации используемый по умолчанию для передачи данных между клиентом и сервером. Используя строгую типизацию полей и бинарный формат для передачи структурированных данных потребляет меньше ресурсов. Время выполнения процесса сериализации/десериализации значительно меньше, как и размер сообщений в отличии от JSON/XML.

Если подытожить, то gRPC также является способом разработки API, но все же есть отличия от REST. В gRPC один компонент (клиент) вызывает определенные функции в другом программном компоненте (сервере), по-другому это называется – вызов удаленных процедур (RPC). В REST вместо вызова функций клиент запрашивает или обновляет данные на сервере. Также gRPC использует протокол Protobuf над данными, что позволяет сократить используемые ресурсы для их передачи.

### 2.2.3 RabbitMQ

RabbitMq – это программный продукт для реализации брокера сообщений на основе протокола AMQP (Advanced Message Queuing Protocol). Простыми словами, программа, которая принимает задания от другой программы или другой части той же программы. Используется для выполнения асинхронных запросов, не заставляя пользователя ждать, пока программа обработает запрос, который можно обработать в фоне.

Принцип работы: сервису нужно передать сообщение (данные, запрос) другому сервису. Но нужно сделать это асинхронно (не ждать пока другой сервис ответит, а продолжать работу), в таком случае используется очередь для сообщений и RabbitMq помещает данное сообщение в это очередь. Дальше, когда второй сервис освободится и сможет принять это сообщение, то он обращается в эту очередь и берет первое сообщение по принципу FIFO (first in, first out), то есть первое отправленное сообщение будет приниматься также первым.

Также существует так называемая DLQ (Dead-letter queue) или очередь недоставленных писем, это отдельная очередь, предназначенная для сообщений, которые по каким-то причинам не могут быть обработаны или доставлены (то есть когда сервис, который должен принять сообщение, отказывается от него по каким-либо причинам).

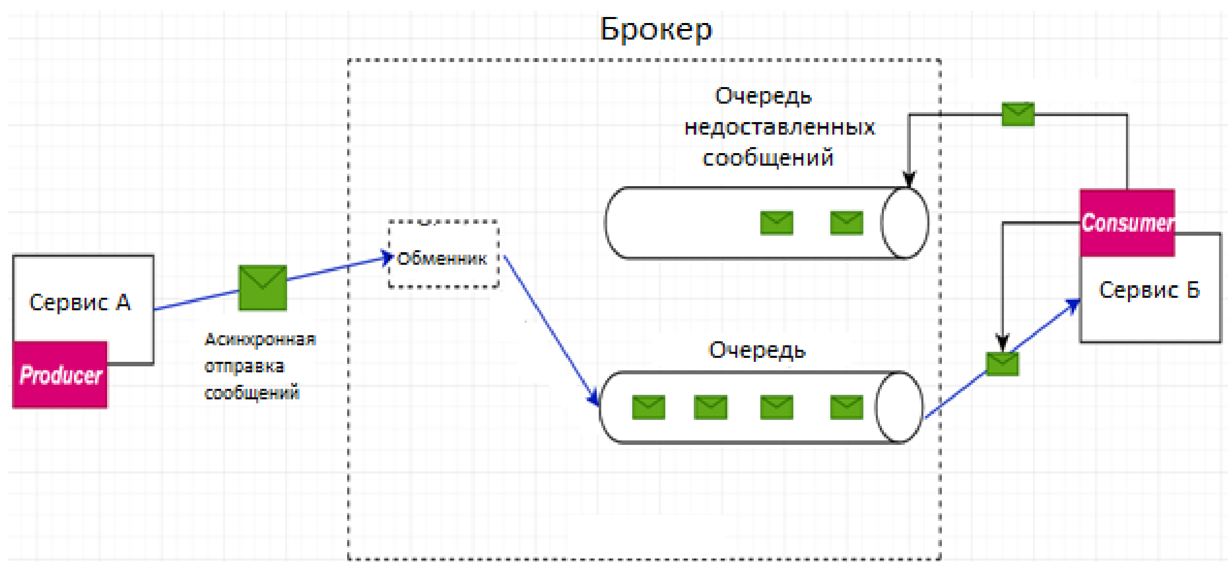



Рисунок 6 – Взаимодействие при помощи RabbitMq

### 2.2.4 Kafka

Apache Kafka - это программный продукт, предназначенный для стриминга данных в реальном времени на основе публикации-подписки. Простыми словами, это система, которая позволяет передавать данные от одного компонента к другому, обеспечивая надежную и эффективную передачу потоков данных.

В контексте стриминга данных, Kafka представляет собой посредника между производителями данных (например, приложениями, устройствами, датчиками) и потребителями данных (например, аналитическими приложениями, хранилищами данных, системами уведомлений). Производители помещают данные в темы (topics) Kafka, а потребители читают эти данные из тем для их дальнейшей обработки.

Принцип работы Kafka таков: данные публикуются в темы Kafka и сохраняются в них в виде непрерывного потока. Потребители могут читать данные из темы в реальном времени, а Kafka гарантирует доставку данных в том же порядке, в котором они были опубликованы (принцип FIFO). Это обеспечивает надежную и последовательную обработку данных.

Кроме того, в Kafka также существует понятие Dead-letter queue (DLQ) - это механизм обработки сообщений, которые не могут быть успешно обработаны или доставлены. Такие сообщения могут быть направлены в отдельную очередь для дальнейшего анализа или обработки.

Таким образом, мы видим сходство Kafka и RabbitMq. Их отличие заключается в том, что первый непрерывно отправляет данные и второй сервис может их читать в реальном времени, а второй отправляет отдельные сообщения.

### 2.3 Разработка структуры базы данных

Как уже было рассказано раньше, будет пять различных сервисов и для каждого из них нам нужна будет база данных. Рассмотрим схемы баз данных для каждого из них.

#### 2.3.1 Сервис авторизации

Сервис авторизации ответственен за работу с пользователями, соответственно у нас есть таблица пользователей (Рисунок 7), со всеми нужными данными, также есть таблица ролей, которая поможет отличить работников от пользователей.

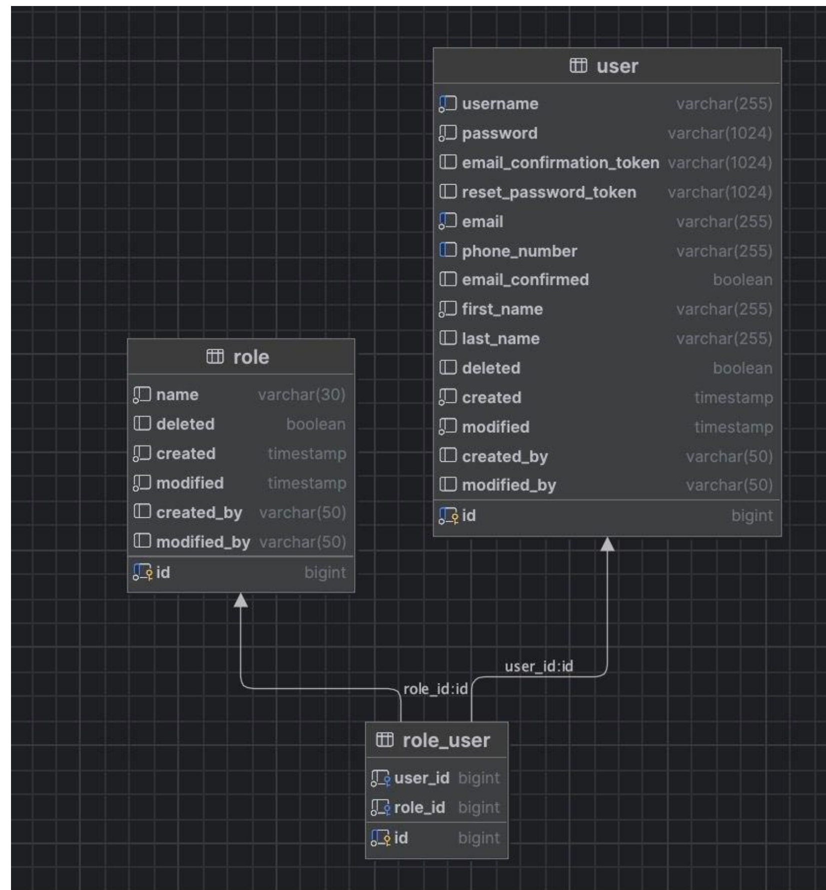



Рисунок 7 – Диаграмма базы данных сервиса авторизации

### 2.3.2 Сервис товаров

Сервис продуктов будет отвечать за хранение информации о товаре (Рисунок 8), например, его тип, цена, описание и название.

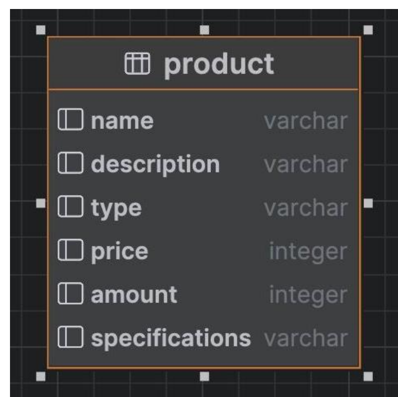



Рисунок 8 – Диаграмма базы данных сервиса продуктов

### 2.3.3 Сервис заказов

Сервис заказов ответственен за создание и отслеживание заказов, поэтому нам нужны таблицы (Рисунок 9) для всего заказа, для одного предмета из заказа, а также таблицы для отслеживания статусов.

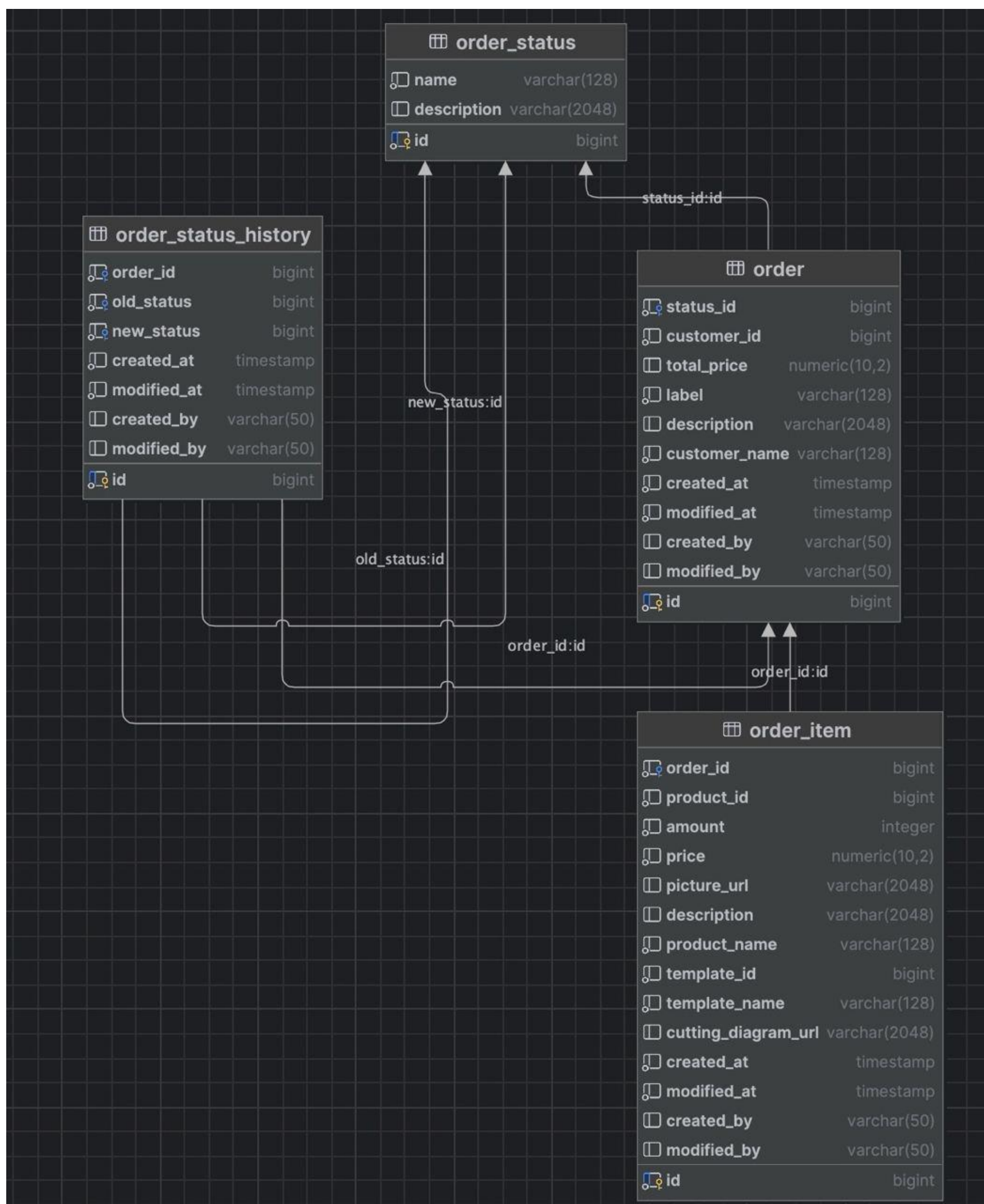



Рисунок 9 – Диаграмма базы данных сервиса заказов

### 2.3.4 Сервис уведомлений

Этот сервис отвечает за составление и отправку уведомлений, поэтому все что ему нужно хранить – это шаблоны уведомлений (Рисунок 10).

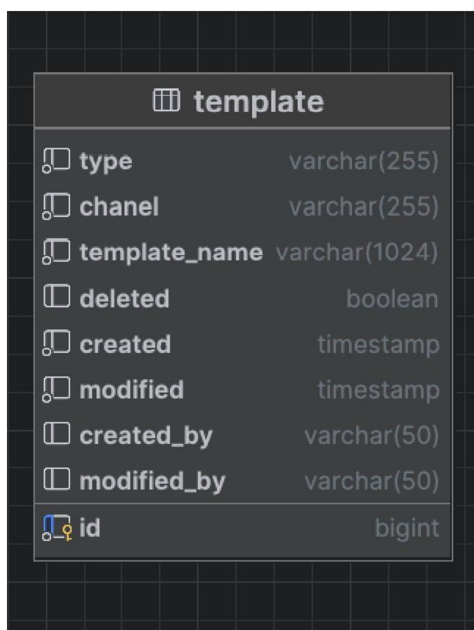



Рисунок 10 – Диаграмма базы данных сервиса уведомлений

### 2.3.5 Сервис отзывов

Сервис будет отвечать за создание и публикацию отзывов, поэтому для хранения ему нужен шаблон самого отзыва (Рисунок 11).

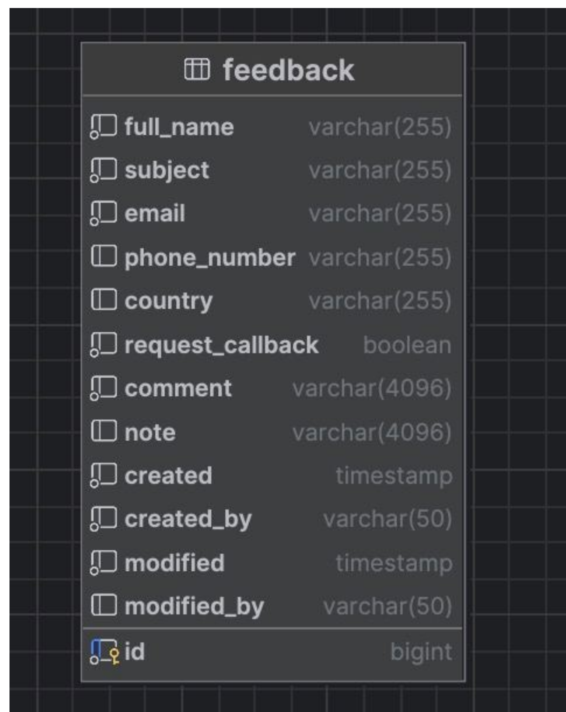



Рисунок 11 - Диаграмма базы данных сервиса отзывов

## 2.4 Выводы по главе

В этой главе были рассмотрены основные средства разработки для приложения, а также спроектирована схема самого приложения (Рисунок 5) и спроектированы схемы баз данных для каждого сервиса.

## 3 Разработка веб-приложения

В этой главе будет описан процесс разработки каждого сервиса моего веб-приложения

### 3.1 API-шлюз

API-шлюз или API-Gateway - это базовый паттерн программирования, вокруг которого строится микросервисная архитектура. По смыслу это маршрутизатор API-запросов, который получает на вход все запросы клиента, обрабатывает и рассылает разным сервисам, а после отсылает ответ обратно клиенту. [1]

Шлюз выполняет следующие функции:

– маршрутизация: перенаправляет входящие запросы к соответствующим микросервисам;

– безопасность: может обеспечивать аутентификацию и авторизацию запросов;

– обработка: может выполнять дополнительные операции над запросами и ответами, такие как логгирование, мониторинг, кэширование и переписывание путей.

Основная задача api-gateway — упростить взаимодействие клиентов с микросервисами, скрывая сложность внутренней архитектуры и предоставляя единый API для доступа к различным сервисам.

Основой этого сервиса является класс RoutesConfig (Приложение А), который отвечает за конфигурацию маршрутов для перенаправления запросов к соответствующим микросервисам. Этот класс аннотирован как @Configuration, что делает его конфигурационным классом Spring.

Основные функции класса RoutesConfig:

– чтение конфигурационных параметров: класс использует аннотацию @Value для чтения URL-адресов микросервисов из конфигурационных файлов;

– определение маршрутов: метод customRouteLocator определяет маршруты для различных микросервисов с помощью объекта RouteLocatorBuilder. Внутри метода используются методы route для указания пути запроса и соответствующего URL-адреса микросервиса;

– фильтры и переписывание путей: для каждого маршрута используется фильтр rewritePath для переписывания пути запроса. Также добавляется пользовательский фильтр PathFilter для дополнительной обработки запросов.

В качестве пользовательского фильтра используется класс PathFilter, который реализует интерфейс «GatewayFilter», предоставляя метод filter, который используется для обработки каждого входящего запроса перед его передачей на целевой микросервис. Задача фильтра состоит в том, чтобы извлечь оригинальный путь запроса, модифицировать его, добавляя туда свой заголовок и передать этот запрос дальше по цепочке.

Примеры запросов:

– auth_route: перенаправляет запросы с путями /api/auth/** к сервису аутентификации;

– order_route: перенаправляет запросы с путями /api/order/** к сервису заказов;

– notification_route: перенаправляет запросы с путями /api/notification/** к сервису уведомлений;

– product_route: перенаправляет запросы с путями /api/product/** к сервису продуктов;

– feedback_route: перенаправляет запросы с путями /api/feedback/** к сервису обратной связи.

В итоге данный сервис позволяет централизованно управлять маршрутами и конфигурациями запросов, обеспечивая гибкость и удобство в изменение различных маршрутов, путем отбрасывания необходимости изменять код каждого отдельного сервиса, и настройкой всего, что касается маршрутизации в одном месте.

### 3.2 Сервис авторизации

Сервис авторизации нужен для того, чтобы реализовать аутентификацию и авторизацию пользователя в приложение. Это нужно для того, чтобы определять и хранить информацию о конкретном пользователе, его правах и ролях на сайте.

Данный сервис содержит 2 основные сущности: Роль и Пользователь. Эти сущности привязаны к соответствующим таблица в базе данных.

Сущность «Роль» отвечает за хранение ролей пользователя. Роль имеет только два поля, они представлены в таблице 1.

Таблица 1 – Основные поля сущность «Роль»

| Тип данных | Название | Описание |
|---|---|---|
| long | id | Идентификатор роли |
| string | name | Название роли |

Сущность пользователя отвечает за хранение информации о человеке, её поля представлены в таблице 2.

Таблица 2 – Основные поля сущность «Пользователь»

| Тип данных | Название | Описание |
|---|---|---|
| long | id | Идентификатор пользователя |
| string | username | Логин пользователя в приложение |
| string | password | Пароль пользователя в зашифрованном виде |
| string | firstName | Имя пользователя |
| string | lastName | Фамилия пользователя |
| string | email | Почта |
| string | phoneNumber | Номер телефона |
| role | roles | Список ролей пользователя |
| boolean | isEmailConfirmed | Флаг для подтверждения почты |
| string | resetPasswordToken | Токен для сброса пароля |

Обе эти сущности имеют соответствующие таблицы в базе данных. Еще есть третья таблица – role_user. Она хранит все связи между пользователями и ролями.

Аутентификация пользователя работает на базе фреймворка Spring Security для Java, который имеет различные вариации аутентификации и авторизации пользователя. В моем случае процесс работает по такому принципу:

1. Пользователь вводит логин и пароль.
2. Создается токен с именем пользователя и его паролем.
3. Spring загружает данные пользователя, декодирует пароль и проверяет его.
4. Если все проверки прошли успешно, то возвращает аутентифицированный объект.
5. Если аутентификация не удалась, то выбрасывает соответствующее исключение.

После того как пользователь аутентифицировался, он добавляется в SecurityContext, который предоставляет глобальный доступ к контексту безопасности. Это нужно для последующих вызовов в пределах текущего потока, так как они могут использовать данные о текущем аутентифицированном пользователе.

Также стоит отметить тот факт, что SecurityContext хранит информацию об аутентификации пользователя только в рамках текущего запроса и для того, чтобы при каждом последующем запросе нам не приходилось проводить аутентификацию заново, мы генерируем accessToken, который хранится на клиентской части и при каждом запросе передается на сервер. Наличие этого токена указывает на то, что держатель токена обладает правом доступа ко всем защищенным ресурсам.

Сервис авторизации имеет свое API, которое реализовано в классе AuthController.java (Приложение Б) и имеет такие методы:

1. InviteUser – метод для приглашение пользователя. В данный момент работает в качестве регистрации.
2. SignIn – метод для аутентификации пользователя.
3. RefreshAccessToken – метод для обновления accessToken.
4. RefreshPassword – метод для обновления текущего пароля.
5. ConfirmEmail – метод для подтверждения почты.

Сервис авторизации общается с сервисом уведомлений и отправляет сообщение пользователю на почту. Это происходит, когда пользователю нужно подтвердить email или, например, обновить пароль. Сервисы общаются посредство RabbitMq и в сервисе авторизации за отправку сообщений отвечает класс RabbitMqMessageProducer. Класс имеет всего два метода, первый подготавливает объект для передачи, второй конвертирует объект в json и отправляет в обменник, который в свою очередь распределяет сообщение в нужные очереди.

В итоге данный сервис имеет очень важную роль, он аутентифицирует и авторизует пользователя, и генерирует для него его личный токен доступа (accessToken), который нужен пользователю, чтобы иметь возможность отправлять запросы в другие сервисы, не проходя при этом повторную авторизацию для каждого нового запроса. Все остальные сервисы имеют свой собственный функционал, который получает всю информацию о пользователе из токена.

### 3.3 Сервис уведомлений

Сервис уведомлений отвечает за отправку уведомлений пользователям. Пока что реализована отправка уведомлений только на email почту пользователя.

Сервис уведомлений получает сообщений от других сервисов и отправляет их. Получение уведомлений реализовано при помощи RabbitMq и работает асинхронно. То есть, когда другой сервис посылает нам сообщение, которое мы

должны отправить пользователю, то этот процесс выполняется в фоне и никак не влияет на текущую работу приложения.

Конфигурация RabbitMq задается в классе RabbitMQConfig (Приложение В), который помечен как конфигурационный класс Spring. В этом классе происходит следующее:

1. Создаются все основные компоненты для обмена сообщениями, если конкретней, то это: обменник (сообщения вначале приходят в обменник, а отсюда он их маршрутизирует в нужные очереди) и очереди (как основная, так и dead-letter).
2. Биндинг (очереди прикрепляются к обменнику, используя специальные ключи).
3. Конфигурация connectionFactory (это объект подключения сервиса к RabbitMq).
4. Настройка шаблонов для отправки сообщений.
5. Инициализация конвертора, который преобразует сообщение в JSON и обратно (в качестве конвертора используется библиотека Jackson).

«Точкой» приема сообщений является класс NotificationConsumer.java (Consumer – потребитель), который по расписанию получает сообщения из очереди. Класса для отправки сообщений (Producer) нет, так как сервис уведомлений сам не отправляет сообщения другим сервисам.

Для отправки сообщений на почту используется класс EmailService.java, который имеет всего 2 метода:

– prepareMessage – подготавливает сообщение, проставляя тему и отправителя;

– sendMessage – отправляет сообщение при помощи JavaMailSender из фреймворка Spring.

Подводя итоги по сервису уведомлений, сейчас он умеет отправлять сообщений на почту пользователям, в будущем будут и другие виды уведомлений. Сервис использует механизмы работы с почтой из Spring, а с

другими сервисами общается при помощи RabbitMq. В данный момент сервис уведомлений получает сообщений только от сервиса авторизации.

### 3.4 Сервис заказов

Сервис заказов отвечает за создание, получения и хранение информации о заказе. Он представляет из себя обычное API, которое принимает запросы на создание, получение, удаление и изменения заказов.

Сервис имеет сущности заказа и позиции заказа, которые хранятся в базе данных.

Сущность заказа – «Order» хранится в таблице order, её поля отображены в таблице 3.

Таблица 3 – Основные поля сущности «Заказ»

| Тип данных | Название | Описание |
| --- | --- | --- |
| long | id | Идентификатор заказа |
| string | status | Статус заказа |
| long | customerId | Идентификатор покупателя |
| bigDecimal | totalPrice | Итоговая цена |
| string | label | Название заказа |
| string | description | Описание заказа |
| string | customerName | Имя покупателя |

Сущность позиции заказа – «OrderItem» хранится в таблице order_item, его поля отображены в таблице 4.

Таблица 4 – Основные поля сущности «Позиция»

| Тип данных | Название | Описание |
| --- | --- | --- |
| long | orderId | Идентификатор заказа |
| long | productId | Идентификатор продукта |
| integer | amount | Количество |
| bigDecimal | price | Цена |
| string | pictureUrl | Картинка позиции |
| string | description | Описание позиции |
| string | productName | Название продукты |

Заказ напрямую не хранит внутри себя информацию о своих позициях. В формах где нужно отобразить все позиции заказа, они будут подтягиваться из таблицы order_item по идентификатору заказа.

Также есть дополнительная таблица, которая хранит информацию про статусы, она называется order_status и хранит его имя и описание. Она тоже напрямую не связана с заказом, но получить статус заказа можно из этой таблицы по его идентификатору.

Последняя таблица в этом сервисе – order_status_history, которая хранит историю статусов определенного заказа.

Основой данного сервиса является класс OrderController.java, который содержит все методы, которые доступны пользователю по api запросу. Он содержит следующие методы:

1. getOrderById – метод получения статуса по его идентификатору.
2. createOrder – метод для создания заказа.
3. updateOrder – изменить информацию в существующем заказе.
4. deleteOrderById – удаление заказа по его идентификатору.
5. findAllOrders – метод получения всех заказов.
6. updateOrderStatus – метод обновление статуса заказа.

Сервис заказов представляет из себя API и позволяет проводить базовые операции над заказом: создание, удаление, получение, обновление. Также хранит эти заказы в базе данных. На данный момент сервис почти не имеет какой-то иной логики, только автоматическое формирование некоторых полей (например, label), но в будущем возможно появятся еще какие-нибудь требования к обработке заказа.

### 3.5 Сервис товаров

Сервис товаров имеет схожее строение с сервисом заказом. Он выполняет роль хранилища и API для товара.

Класс имеет одну сущность «Товар» и соответствующую таблицу в базе данных – product, её поля отображены в таблице 5.

Таблица 5 – Основные поля сущности «Товар»

| Тип данных | Название | Описание |
|---|---|---|
| long | id | Идентификатор товара |
| String | name | Название товара |
| string | type | Категория товара |
| bigDecimal | price | Цена товара |
| string | description | Описание товара |
| string | specifications | Характеристики |
| integer | amount | Количество |

За API отвечает ProductController, который имеет базовые CRUD метода – создание, удаление, обновление, получение.

В итоге сервис имеет очень маленькую функциональность, но был выделен отдельно для удобства и для того, чтобы хранить большое количество товаров в отдельном месте и не нагружать этим другие сервисы.

### 3.6 Сервис отзывов

Последний сервис, это сервис отзывов. Сервис отвечает за хранение и работу с отзывами.

Класс имеет одну сущность «Отзыв» и советующую таблицу feedback, в которой и хранятся отзывы. Основные поля сущности отражены в таблице 6.

Таблица 6 – Основные поля сущности «Отзыв»

| Тип данных | Название | Описание |
|---|---|---|
| long | id | Идентификатор товара |
| string | fullName | Имя и фамилия пользователя |
| string | subject | Тема |
| string | email | Почта |
| string | phoneNumber | Номер телефона |
| string | country | Страна |
| string | comment | Комментарий |
| string | note | Заметка |

Окончание таблицы 6

| Тип данных | Название | Описание |
|---|---|---|
| boolean | requestCallback | Флаг показывающие, требуется ли перезванивать клиенту |

Отзыв можно оставлять под каким-то конкретным товаром или просто в специальной форме. Также можно будет поставить галочку и попросить, чтобы компания перезвонила клиенту и ответила на его вопросы. Отзыв в данном случае подразумевает не обсуждение какого-либо товара, а скорее оставление заявки на его приобретение.

Сам сервис также имеет структуру обычно API и имеет такие методы в контроллере (FeedbackController):

1. createFeedback – создание отзыва.
2. getFeedbackById – получение отзыва по идентификатору.
3. getFeedbackResponsePage – получает все отзывы.
4. addNoteToFeedBack – добавлят заметку к отзыву.

В настоящие время сервис имеет небольшой функционал, но в будущем планируется больше новых функций. Например, автоматическая модерация отзывов, автоматизация ответов для отзывов, которые не требуют вмешательство сотрудников. Как было сказано, сейчас отзывы подразумевают скорее заявку на то, чтобы с вами связались, но в будущем также будет реализована возможность обсуждать товары и ставить свои оценки.

### 3.7 Развертывание системы

Для развертывания системы используется Docker. Это система для контейнеризации приложений, она помогает упаковать все приложения (сервисы) в один контейнер и расположить их в одной среде. [7] У каждого сервиса есть специальный файл Dockerfile, который отвечает за сборку своего

сервиса (Приложение Г). В этом файле содержаться все основные этапы сборки для сервиса. На первом этапе мы загружаем все нужные образы, копируем файл конфигурации и все исходные коды, а потом упаковываем все приложение в jar файл. На втором этапе мы копируем готовый jar файл в нужную директорию и запускаем его. Таким образом запускается сервис.

Для того, чтобы можно было разворачивать все сервисы за раз, а не по одному, существует файл docker-compose.yml (Приложение Д). Docker compose – это инструмент для определения и управления несколькими контейнерами за раз. [8] Он содержит описание каждого сервиса и может развернуть все сервисы за раз. В описание каждого контейнера имеется параметр «context», который содержит путь к Dockerfile нужного сервиса, по этому пути docker compose запускает нужный контейнер и таким образом разворачивает каждый сервис. Также в описание контейнеров есть другие параметры, которые, например, определяют переменные окружения для каждого сервиса, соответствие портов и зависимости сервисов друг от друга (например, когда один сервис использует другой и должен запустится позже него).

### 3.8 Выводы по главе

В этой главе был описан процесс разработки веб-приложения на микросервисной архитектуре. Было описано назначение каждого сервиса, его функционал и основные точки доступа (для API). Был представлен атрибутивный состав основных сущностей приложения, и описано их взаимодействие. Пока общение между сервисами работает только для двух из них, но в будущем планируется добавить еще больше взаимодействий. Также был разобран способ развертывания микросервисного приложения при помощи системы контейнеризации Docker.

# 4 Сравнение способов взаимодействия микросервисов между собой

В качестве исследования было принято решение сравнить методы взаимодействия между собой. В целом если углубится в документацию, то можно понять какой способ лучше в каких ситуациях. Например, очевидно, что Kafka лучше использовать, когда нужно беспрерывно отправлять поток каких-то данных, а Rest API удобен для маленьких приложений, потому что очень прост в развертке и довольно понятен в использовании. gRPC чаще всего используют при больших объемах, благодаря его встроенной особенности сжатия данных. RabbitMq удобен в случаях, когда нужно просто отправить сообщение и забыть про него. Все вполне понятно, но как поведут себя эти технологии не в «своей» ситуации, возможно ли, что при работе с маленьким количеством данных Kafka будет работать тоже вполне уверенно, или что в некоторых, казалось бы, не подходящих сценариях асинхронность RabbitMq будет вполне уместна. Для того, чтобы выяснить это, я провел небольшое исследование. Суть исследования заключается в том, что есть два сервиса между которыми отправляются разные данные и замеряются показатели.

Для сравнения были выбраны следующие метрики:

1. Время отклика – время необходимое для отправки сообщения от одного сервиса к другому.
2. Пропускная способность – количество сообщений, которые могут быть обработаны за определенное количество времени.
3. Нагрузка на процессор – количество используемых ресурсов процессора при передаче данных.
4. Использование памяти – количество используемой памяти при передаче сообщений.

Для тестирования я буду использовать сценарии с различным размером данных (1 Мб, 100 Мб, 500 Мб). Данные будут передаваться в json формате, так-

как формат не сильно влияет на скорость передачи, он скорее влияет на время сериализации/дессериализации данных.

Тестирование будет проводится на готовом приложение. Данные будут отправляться между сервисом авторизации и сервисом уведомлений. В качестве данных буду использоваться сообщения разной длины, которые нужно отправлять пользователю.

Тестирование проводилось на Macbook Air M1, 16 Гб оперативной памяти, 8 ядер.

### 4.1 Время отклика

Время отклика – это время, необходимое для отправки сообщения от одного сервиса к другому. Для подготовки к этим тестам нужно, что бы оба сервиса были развернуты, первый (сервис авторизации) будет отправлять сообщение, второй (сервис уведомлений) будет получать их и возвращать ответ, и уже после получения ответа будет записываться время. Во втором сервисе не будет производится никаких манипуляций над данными, будет сразу отсылаться ответ.

Возьмем в качестве примера Rest API и посмотрим, как это выглядит.

Сервис авторизации будет отвечать за отправку документа. Он содержит один класс RestSender.java (Приложение Е). Нам нужна всего одна зависимость для этого класса – RestTemplate. Это служебный класс в Spring Framework, который упрощает отправку HTTP-сообщений и обработку ответа. И также нам нужна константа, которая будет определять URL адрес, на который мы будем отправлять запрос. В этом классе есть метод sendRequest, на вход которому приходит массив байт, который мы подготавливаем в методе prepareMessage. Для удобства тестирование передается массив байт нужного размера, а не строка. Вначале нужно замерить начальное время в переменной startTime, в которую записывается текущее время, а дальше просто отправляем запрос при помощи метода postForEntity из библиотеки RestTemplate. Конвертировать массив байт в

json не нужно, так как метод postForEntity делает это автоматически. После того как я получил ответ остается только замерить время окончания работы endTime и в качестве результата берем разницу между конечным и начальным временем, это и будет время за которое произошла отправка данных и получение ответа.

Сервис уведомлений будет принимать данные и отправлять ответ. У него также есть новый класс, DataController.java (Приложение Ж), который содержит один метод – recieveData. Этот метод не делает никакой обработки полученных данных, а просто сразу же отправляет нам ответ.

Результаты для набора данных весом 1 Мб, 100 Мб и 500 Мб представлены на таблицах 7, 8 и 9 соотвественно.

Таблица 7 – Время отклика для данных весом 1МБ

| Объем данных | Rest API (мс) | gRPC (мс) | RabbitMQ (мс) | Kafka (мс) |
|---|---|---|---|---|
| 1 Мб | 150 | 50 | 120 | 80 |
| 1 Мб | 148 | 52 | 115 | 79 |
| 1 Мб | 151 | 49 | 122 | 81 |
| 1 Мб | 149 | 51 | 118 | 82 |
| 1 Мб | 150 | 50 | 121 | 80 |
| 1 Мб | 147 | 53 | 117 | 83 |
| 1 Мб | 152 | 48 | 119 | 78 |
| 1 Мб | 148 | 50 | 120 | 81 |
| 1 Мб | 150 | 52 | 118 | 80 |
| 1 Мб | 149 | 51 | 121 | 79 |
| **Среднее** | **149.4** | **50.6** | **119.1** | **80.3** |

Таблица 8 – Время отклика для данных весом 100МБ

| Объем данных | Rest API (мс) | gRPC (мс) | RabbitMQ (мс) | Kafka (мс) |
|---|---|---|---|---|
| 100 Мб | 1700 | 500 | 1200 | 800 |
| 100 Мб | 1680 | 520 | 1150 | 790 |
| 100 Мб | 1710 | 490 | 1220 | 810 |
| 100 Мб | 1690 | 510 | 1180 | 820 |
| 100 Мб | 1700 | 500 | 1210 | 800 |
| 100 Мб | 1670 | 530 | 1170 | 830 |
| 100 Мб | 1720 | 480 | 1190 | 780 |
| 100 Мб | 1680 | 500 | 1200 | 810 |
| 100 Мб | 1700 | 520 | 1180 | 800 |
| 100 Мб | 1690 | 510 | 1210 | 790 |
| **Среднее** | **1694.0** | **506.0** | **1191.0** | **802.0** |

Таблица 9 – Время отклика для данных весом 500МБ

| Объем данных | Rest API (мс) | gRPC (мс) | RabbitMQ (мс) | Kafka (мс) |
|---|---|---|---|---|
| 500 Мб | 8500 | 2500 | 6000 | 4000 |
| 500 Мб | 8400 | 2600 | 5750 | 3950 |
| 500 Мб | 8550 | 2450 | 6100 | 4050 |
| 500 Мб | 8450 | 2550 | 5900 | 4100 |
| 500 Мб | 8500 | 2500 | 6050 | 4000 |
| 500 Мб | 8350 | 2650 | 5850 | 4150 |
| 500 Мб | 8600 | 2400 | 5950 | 3900 |
| 500 Мб | 8400 | 2500 | 6000 | 4050 |
| 500 Мб | 8500 | 2600 | 5900 | 4000 |
| 500 Мб | 8450 | 2550 | 6050 | 3950 |
| **Среднее** | **8475.0** | **2530.0** | **5955.0** | **4015.0** |

Для каждого набора данных и каждой технологии было проведено по 10 тестовых запусков из полученных данных можно сказать следующее:

1. gRPC показывает наименьшее время отклика для обоих форматов данных благодаря бинарной сериализации и использованию HTTP/2.

2. Kafka также демонстрирует хорошую производительность с умеренным временем отклика

3. RabbitMQ показывает средние результаты, что может быть связано с асинхронной природой системы.

4. Rest API имеет самое высокое время отклика, из-за overhead. Overhead - это различные накладные расходы, которые происходят во время передачи данных, например, если речь про HTTP, то он включает дополнительные заголовки в запрос и в ответ, что немного утяжеляет их, а также тратится время на их добавление.

Опираясь на результаты (Рисунок 12) можно сделать такой вывод, если вы собираетесь передавать тяжелые данные, то самым лучшим способом будет являться gRPC, так как у него есть встроенная бинарная сериализация данных, что сильно облегчает ваш запрос, также Kafka показывает себя довольно неплохо, хуже, чем gRPC, но намного лучше, чем остальные способы. Таким образом можно отметить, что, не смотря, на то, что основное преимущество Kafka это непрерывная передача данных, но и при простой отправке он показывает достаточно хорошие результаты и его можно использовать в

ситуациях, когда вам не нужна непрерывная передача. Что касается остальных способов, то Rest API в целом плохо подходит для передачи больших данных, если вам важна скорость, но его очень редко используют для общения между сервисами, чаще всего его используют для выполнения запросов с клиентской части. Касательно RabbitMQ, тут тоже все довольно ожидаемо, так как из-за асинхронности он не рассчитан на скорость передачи, как правило при использовании RabbitMq вы отправляете сообщение в очередь и вам не сильно важно, как долго оно будет обрабатываться (в разумных пределах конечно).

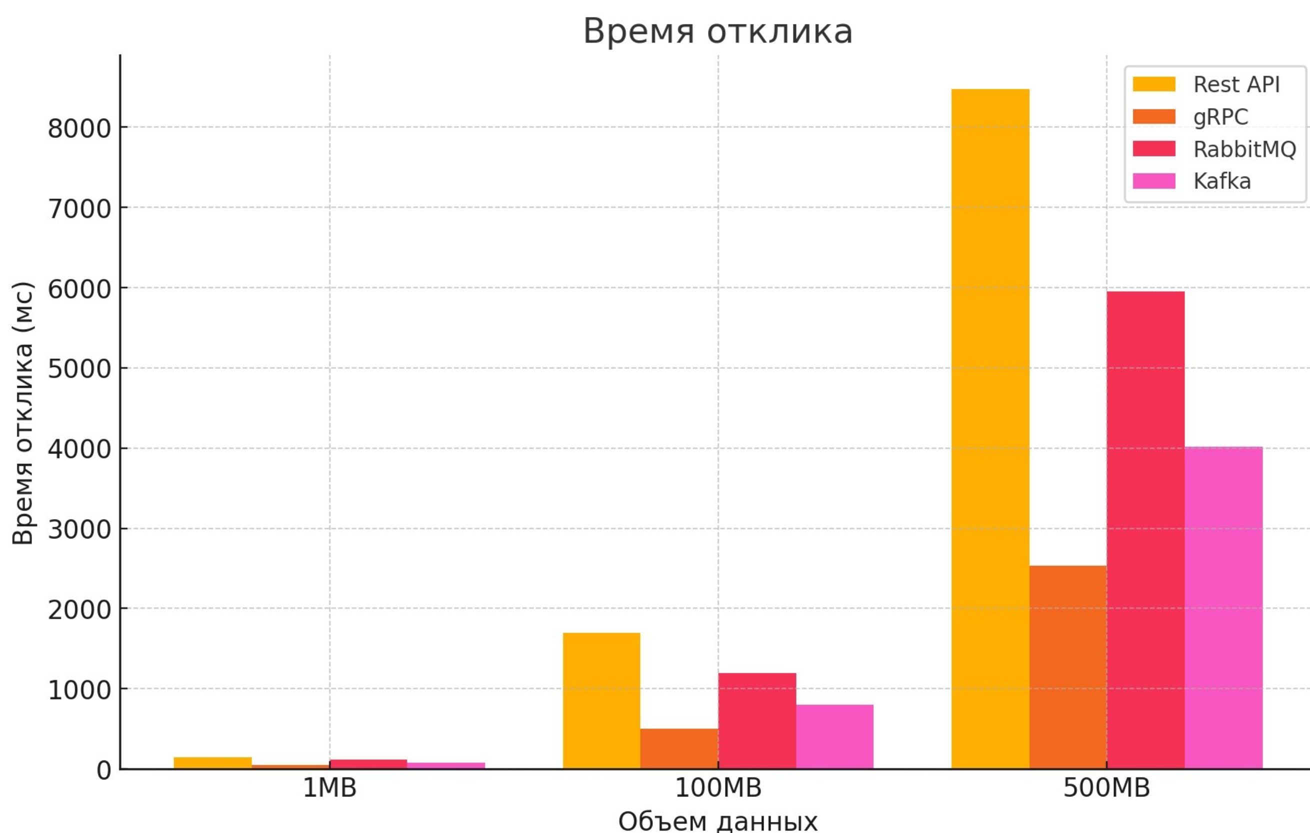



Рисунок 12 – Диаграмма сравнения способов взаимодействия по времени отклика

## 4.2 Пропускная способность

Пропускная способность – это количество сообщений, которые могут быть обработаны за определенный период времени. Суть довольна проста, чтобы

измерить пропускную способность, нам нужны средние значения времени, за которое сервис будет выполнять запрос, то есть это средние значения времени отклика, которые мы измеряли в предыдущем пункте. Дальше для определения пропускной способности нам нужно воспользоваться формулой (1):

$$\text{Пропускнаяспособность}(MB/s) = \frac{\text{Средняязадержка(секунды)}}{\text{Размерданных}(MB)} \tag{1}$$

Результаты средней задержки, выраженные в секундах можно увидеть на таблице 10.

Таблица 10 – Средняя задержка в секундах

| **Объем данных** | **Rest API (с)** | **gRPC (с)** | **RabbitMQ (с)** | **Kafka (с)** |
|---|---|---|---|---|
| 1 Мб | 0,1494 | 0,0506 | 0,1191 | 0,0803 |
| 100 Мб | 1,694 | 0,506 | 1,191 | 0,802 |
| 500 Мб | 8,475 | 2,53 | 5,955 | 4,015 |

Теперь если посчитать пропускную способность по формуле, то получаются следующие результаты (Таблица 11):

Таблица 11 – Средние значения пропускной способности

| **Объем данных** | **Rest API (Мб/с)** | **gRPC (Мб/с)** | **RabbitMQ (Мб/с)** | **Kafka (Мб/с)** |
|---|---|---|---|---|
| 1 Мб | 6,69 | 19,76 | 8,39 | 12,46 |
| 100 Мб | 59,02 | 197,63 | 84 | 124,69 |
| 500 Мб | 58,98 | 197,63 | 84,01 | 124,56 |

Из результатов (Рисунок 13) можно сделать выводы что пропускная способность для больших объемов данных (100 и 500 МБ) практически постоянна, потому что средняя задержка (измеряемая в секундах) остается практически постоянной для этих объемов данных. Это происходит потому, что время передачи данных, относительно объема данных, сохраняется на относительно постоянном уровне благодаря эффективной обработке и передаче данных по выбранным методам связи (Rest API, gRPC, RabbitMQ, Kafka). Таким

образом, при увеличении объема данных средняя задержка остается стабильной, что приводит к почти постоянной пропускной способности для различных методов взаимодействия даже при разных объемах данных. Еще можно отметить, что gRPC является лучшим выбором для высокопроизводительных систем, Kafka также выглядит довольно средне, чего не скажешь о Rest Api и RabbitMq, их результаты выглядят не очень впечатляющи для ситуаций, когда нам нужно передавать много тяжелых сообщений за ограниченный период времени.

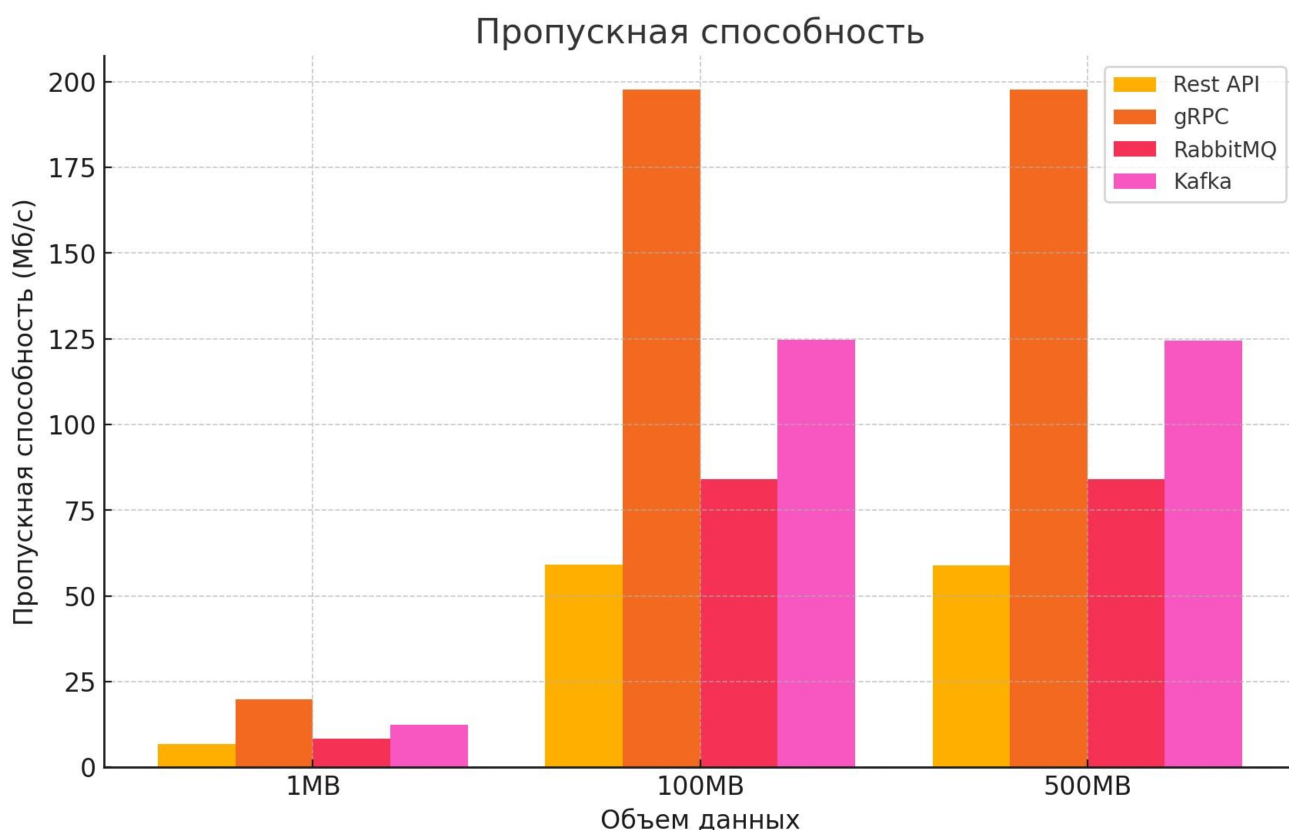



Рисунок 13 - Диаграмма сравнения способов взаимодействия по пропускной способности

### 4.3 Нагрузка на процессор

В этом тесте я собираюсь проверить какую нагрузку каждый способ отправки создает на процессор вашего компьютера.

Для выполнения тестирования нам нужно настроить все сервисы. Дальше при отправке сообщения нам нужно просто смотреть мониторинг нашего процесса, записывая показатели нагрузки на ЦПУ.

Для просмотра процессов я использую встроенную утилиту «ps», также был подготовлен bash скрипт, который будет записывать значение нагрузки на процессор каждую секунду (Приложение 3). Для того, чтобы отделить каждую отправку сообщений друг от друга, после каждого запроса была выставлена пауза в 1 секунду, при помощи стандартного функционала java. Итоговые результаты для данных размером 1 Мб, 100 Мб и 500 Мб можно увидеть на таблицах 12, 13 и 14 соответственно.

Таблица 12 - Результаты тестирования нагрузки на процессор для данных весом 1Мб

| **Объем данных** | **Rest API (%)** | **gRPC (%)** | **RabbitMQ (%)** | **Kafka (%)** |
|---|---|---|---|---|
| 1 Мб | 20 | 15 | 10 | 12 |
| 1 Мб | 18 | 14 | 11 | 11 |
| 1 Мб | 21 | 16 | 10 | 13 |
| 1 Мб | 19 | 15 | 9 | 12 |
| 1 Мб | 20 | 14 | 10 | 11 |
| 1 Мб | 22 | 15 | 11 | 12 |
| 1 Мб | 19 | 14 | 10 | 12 |
| 1 Мб | 18 | 16 | 9 | 11 |
| 1 Мб | 21 | 15 | 10 | 13 |
| 1 Мб | 20 | 15 | 11 | 12 |
| **Среднее** | **19,8** | **14,9** | **10,1** | **11,9** |

Таблица 13 - Результаты тестирования нагрузки на процессор для данных весом 100 Мб

| **Объем данных** | **Rest API (%)** | **gRPC (%)** | **RabbitMQ (%)** | **Kafka (%)** |
|---|---|---|---|---|
| 100 Мб | 30 | 25 | 22 | 20 |
| 100 Мб | 29 | 24 | 21 | 19 |
| 100 Мб | 31 | 26 | 23 | 21 |
| 100 Мб | 30 | 25 | 22 | 20 |
| 100 Мб | 29 | 24 | 21 | 19 |
| 100 Мб | 31 | 25 | 23 | 20 |
| 100 Мб | 30 | 24 | 22 | 19 |
| 100 Мб | 29 | 26 | 21 | 21 |
| 100 Мб | 31 | 25 | 23 | 20 |
| 100 Мб | 30 | 25 | 22 | 20 |

Окончание таблицы 13

| Объем данных | Rest API (%) | gRPC (%) | RabbitMQ (%) | Kafka (%) |
|---|---|---|---|---|
| **Среднее** | **30.0** | **24,9** | **21,9** | **19,9** |

Таблица 14 - Результаты тестирования нагрузки на процессор для данных весом 500 Мб

| Объем данных | Rest API (%) | gRPC (%) | RabbitMQ (%) | Kafka (%) |
|---|---|---|---|---|
| 500 Мб | 35 | 28 | 25 | 22 |
| 500 Мб | 34 | 27 | 24 | 21 |
| 500 Мб | 36 | 29 | 26 | 23 |
| 500 Мб | 35 | 28 | 25 | 22 |
| 500 Мб | 34 | 27 | 24 | 21 |
| 500 Мб | 36 | 28 | 26 | 22 |
| 500 Мб | 35 | 27 | 25 | 21 |
| 500 Мб | 34 | 29 | 24 | 23 |
| 500 Мб | 36 | 28 | 26 | 22 |
| 500 Мб | 35 | 28 | 25 | 22 |
| **Среднее** | **34.9** | **27,9** | **24,9** | **21,9** |

По результатам (Рисунок 14) видно, что RabbitMq и Kafka нагружают процессор намного меньше, связано это с их асинхронной природой, так как приложение по факту не ждет ответа сразу (не зависает, пока не придет ответ), и поэтому нагрузка лучше распределяется по времени, также это достаточно современные технологии и они имеют ряд преимуществ с точки зрения своих алгоритмов, еще они передают данные в двоичном формате, что снижает уровень нагрузки, ну и главное, это то, что они не имеют накладных расходов связанных с протоколом HTTP.

Отсюда можно сделать вывод, что RabbitMq и Kafka являются более подходящими вариантами, если речь идет про задачи, требующие высокой производительности и эффективного использования ресурсов. Это может быть полезно в ситуациях, когда у вас очень большие объемы данных или, например, очень ограниченные ресурсы.

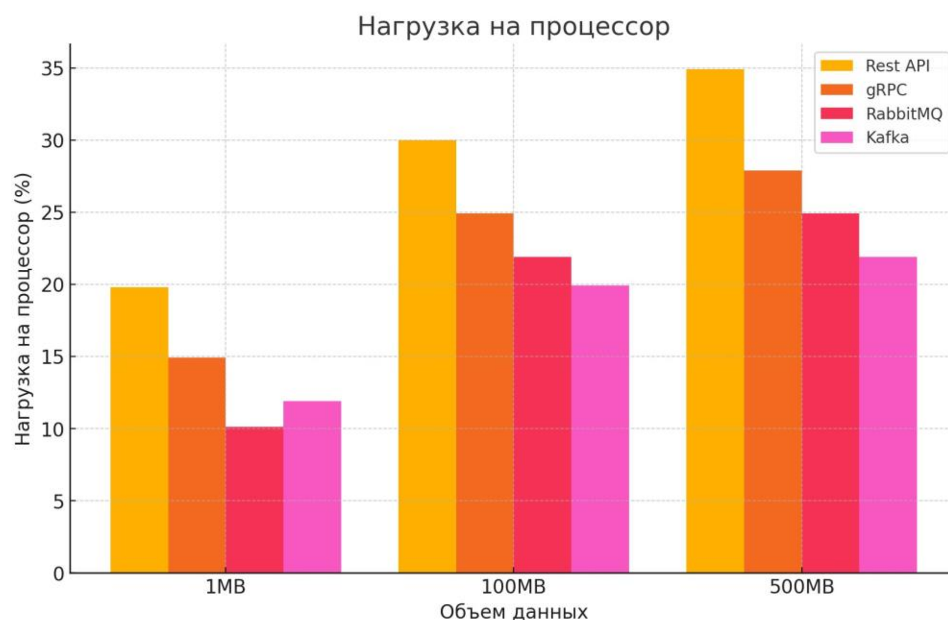



Рисунок 14 - Диаграмма сравнения способов взаимодействия по нагрузке на процессор

## 4.4 Потребление памяти

В последнем тесте я буду измерять объем памяти, используемой при передаче сообщений.

Для того, чтобы измерить количество потребляемой памяти нам нужно проделать все тоже самое, что и с нагрузкой на процессор, только теперь смотрим на потребление памяти.

Для измерений будет также использовать утилиту «ps» и тот же скрипт, в котором был изменен записываемый параметр.

Результаты для 1 Мб, 100 Мб, 500 Мб можно увидеть на таблицах 15, 16 и 17 соответственно.

Таблица 15 - Результаты тестирования потребления памяти для данных весом 1 Мб

| Объем данных | Rest API (%) | gRPC (%) | RabbitMQ (%) | Kafka (%) |
|---|---|---|---|---|
| 1 Мб | 1 | 0,8 | 0,5 | 0,6 |
| 1 Мб | 1,1 | 0,9 | 0,6 | 0,7 |
| 1 Мб | 1 | 0,8 | 0,5 | 0,6 |

Окончание таблицы 15

| **Объем данных** | **Rest API (%)** | **gRPC (%)** | **RabbitMQ (%)** | **Kafka (%)** |
|---|---|---|---|---|
| 1 Мб | 1,1 | 0,9 | 0,6 | 0,7 |
| 1 Мб | 1 | 0,8 | 0,5 | 0,6 |
| 1 Мб | 1,1 | 0,9 | 0,6 | 0,7 |
| 1 Мб | 1 | 0,8 | 0,5 | 0,6 |
| 1 Мб | 1,1 | 0,9 | 0,6 | 0,7 |
| 1 Мб | 1 | 0,8 | 0,5 | 0,6 |
| 1 Мб | 1,1 | 0,9 | 0,6 | 0,7 |
| **Среднее** | **1,05** | **0,85** | **0,55** | **0,65** |

Таблица 16 - Результаты тестирования потребления памяти для данных весом 100 Мб

| **Объем данных** | **Rest API (%)** | **gRPC (%)** | **RabbitMQ (%)** | **Kafka (%)** |
|---|---|---|---|---|
| 100 Мб | 2 | 1,6 | 1,2 | 1,3 |
| 100 Мб | 2,1 | 1,7 | 1,3 | 1,4 |
| 100 Мб | 2 | 1,6 | 1,2 | 1,3 |
| 100 Мб | 2,1 | 1,7 | 1,3 | 1,4 |
| 100 Мб | 2 | 1,6 | 1,2 | 1,3 |
| 100 Мб | 2,1 | 1,7 | 1,3 | 1,4 |
| 100 Мб | 2 | 1,6 | 1,2 | 1,3 |
| 100 Мб | 2,1 | 1,7 | 1,3 | 1,4 |
| 100 Мб | 2 | 1,6 | 1,2 | 1,3 |
| 100 Мб | 2,1 | 1,7 | 1,3 | 1,4 |
| **Среднее** | **2,05** | **1,65** | **1,25** | **1,35** |

Таблица 17 - Результаты тестирования потребления памяти для данных весом 500Мб

| **Объем данных** | **Rest API (%)** | **gRPC (%)** | **RabbitMQ (%)** | **Kafka (%)** |
|---|---|---|---|---|
| 500 Мб | 5 | 4 | 3,2 | 3,5 |
| 500 Мб | 5,1 | 4,1 | 3,3 | 3,6 |
| 500 Мб | 5 | 4 | 3,2 | 3,5 |
| 500 Мб | 5,1 | 4,1 | 3,3 | 3,6 |
| 500 Мб | 5 | 4 | 3,2 | 3,5 |
| 500 Мб | 5,1 | 4,1 | 3,3 | 3,6 |
| 500 Мб | 5 | 4 | 3,2 | 3,5 |
| 500 Мб | 5,1 | 4,1 | 3,3 | 3,6 |
| 500 Мб | 5 | 4 | 3,2 | 3,5 |
| 500 Мб | 5,1 | 4,1 | 3,3 | 3,6 |
| **Среднее** | **5,05** | **4,05** | **3,25** | **3,55** |

По результатам (Рисунок 15) можно сказать следующее, Rest API ведет себя довольно плохо, связано это с накладными расходами HTTP. gRPC показывает результаты лучше, во много благодаря использованию двоичного формата Protobuf. RabbitMq и Kafka имеют достаточно хорошие результаты, опять же из-за того, что передают двоичные данные, а также благодаря своим алгоритмам.

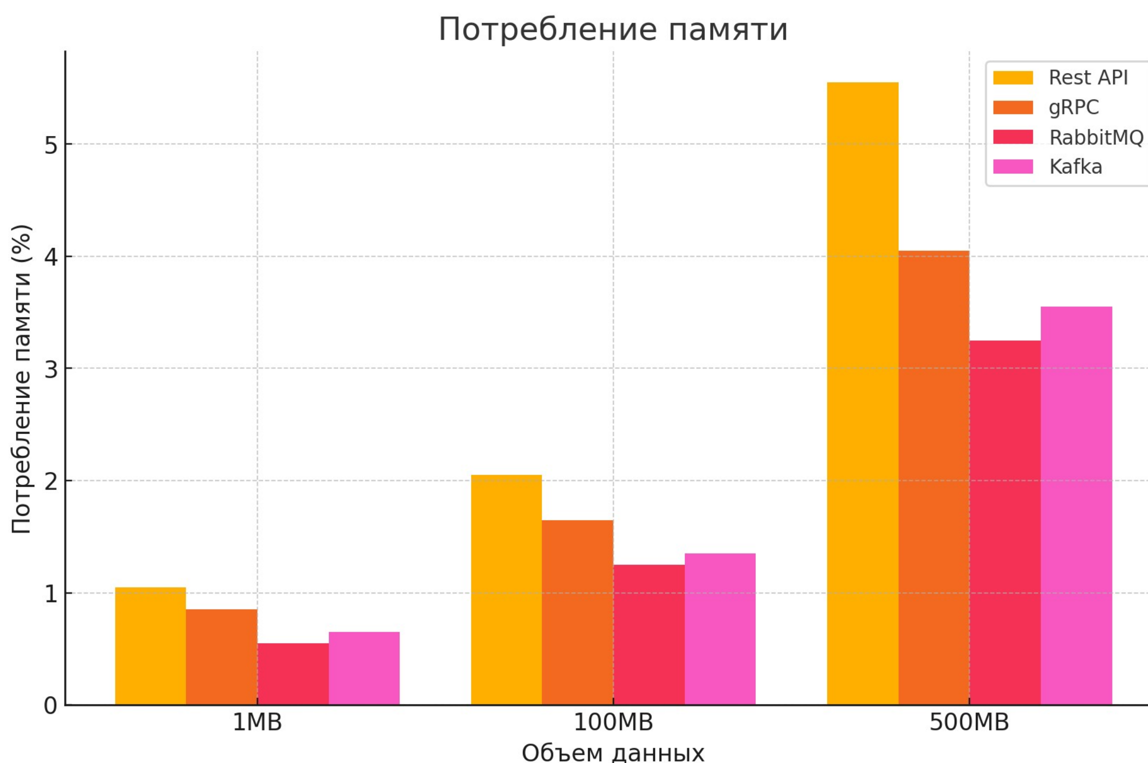



Рисунок 15 - Диаграмма сравнения способов взаимодействия по потреблению памяти

## 4.5 Заключение к главе

В заключение этой главы хочется сказать, что я провел исследование сравнивая 4 самых популярных способа передачи данных между сервисами. Способы достаточно сильно отличаются друг от друга и используют разные технологии. Это было сделано в первую очередь с целью – определить в каких

«нестандартных» для того или иного способа ситуациях можно его использовать. RestApi часто используют для маленьких приложений, потому что считают, что нет необходимости использовать, что то более сложное (с точки зрения конфигурации – Rest API самый простой способ), но из результатов видно, что даже когда у вас небольшой объем данных или небольшое количество запросов, то Rest API выглядит достаточно слабым по сравнению например с gRPC, который благодаря протоколу Protobuf и бинарному формату показывает отличные показатели для всех видов тестов. Также можно понять, что Kafka выглядит очень оптимистично с точки зрения потребления ресурсов и имеет достаточно среднее время отклика, что в целом дает ему возможность соперничать и с gRPC с точки зрения скорости и с RabbitMQ с точки зрения потребления ресурсов. Если говорить про RabbitMQ, то он во всех тестах находится в середине, он очень хорош для своих привычных задач (например, отправка уведомлений), но в целом его можно использовать и в других ситуациях, когда не требуется именно асинхронность.

Если возникает вопрос с выбором способа взаимодействия, то документация ответит на ваши вопросы, но в определенных ситуациях можно использовать и нестандартные подходы, которые могут решить какую-то вашу проблему. Например, если вам просто легче работать с какой-то технологией или это можно использовать как задел на будущее. Например, сейчас вам не нужны асинхронные запросы во время разработки, но в будущем будут нужны. Вы можете их применить уже сейчас, чтобы потом не пришлось переписывать весь сервис, и при это вы не потеряет слишком много с точки зрения производительности.

Что касается конкретного примера на разработанном приложение, то так как отправка уведомлений не требует незамедлительной реакции, то скорость отклика не так важна, а вот показатели касающиеся потребления ресурсов достаточно важны, так как может быть ситуация, когда придется отправлять достаточно объемное сообщение (например, файл с результатами чего-либо) и большое потребление ресурсов в таком случае производит большие затраты на

увеличение этих самих ресурсов, именно по такой причине для отправки уведомлений используются асинхронные способы передачи, которые проигрывают в скорости, но при этом не останавливают работу всего приложение и более экономны с точки зрения ресурсов.

# ЗАКЛЮЧЕНИЕ

В заключение можно сказать, что было произведен обзор того, какие существуют архитектурные подходы для реализации веб-приложение, после которого был выбран вариант микросервисной архитектуры. Также был проведен обзор существующих вариантов взаимодействия сервисов между собой и был разобран принцип работы самых популярных из них, которые в дальнейшем использовались для реализации серверной части веб-приложения для интернета магазина одежды. Приложение было спроектировано и описаны используемые технологии. В итоге получилось реализовать веб-приложение на основе предложенных технологий, и представить описание работы каждого сервиса по отдельности, а также способ развертывания целого приложения. Еще было проведено небольшое исследование касательно сравнения различных способов взаимодействий сервисов между собой. Цель исследования заключалась в том, чтобы понять, какие результаты будут показывать способы взаимодействия в нестандартных условиях применения, что удалось сделать и составить некоторые выводы по этому поводу. В будущем планируется доработать приложение, добавив больше функционала для каждого сервиса. В качестве доработок также планируется добавить больше взаимодействий для сервисов (например, отправка уведомлений пользователям, заказ которых меняет статус).

Касательно выводов какой способ лучше выбрать, то все зависит от конкретной ситуации и целей, но если смотреть на «цифры», то явно видно, что Kafka во всех протестированных сценариях показывает средние результаты, что показывает ее оптимальной по скорости и потреблению ресурсов, поэтому можно сделать вывод, что из всех приведенных способов – Kafka наиболее универсальна.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1 Современная микросервисная архитектуры: принципы проектирования: сайт. – URL: https://habr.com/ru/companies/innotech/articles/683550 (дата обращения: 03.04.2024)

2 Монолитная vs Микросервисная архитектура: сайт. – URL: https://proglib.io/p/monolitnaya-vs-mikroservisnaya-arhitektura-2019-09-16 (дата обращения: 04.04.2024)

3 Микросервисы: преимущества и недостатки: сайт. – URL: https://vc.ru/dev/719244-mikroservisy-preimushestva-i-nedostatki (дата обращения: 04.04.2024)

4 From Monolithic Architecture to Microservices Architecture / Lorenzo De Lauretis. – 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) (дата обращения: 05.04.2024)

5 Взаимодействие в архитектуре микросервисов: сайт. – URL: https://habr.com/ru/companies/slurm/articles/675682/ (дата обращения: 03.04.2024)

6 Способы общения микросервисов для самых маленьких. – URL: https://habr.com/ru/companies/maxilect/articles/677128/ (дата обращения: 15.04.2024)

7 Что такое Docker: сайт. – URL: https://www.oracle.com/cis/cloud/cloud-native/container-registry/what-is-docker/ (дата обращения: 15.05.2024)

8 Docker compose overview: сайт. – URL: https://docs.docker.com/compose/ (дата обращения: 15.05.2024)

**ПРИЛОЖЕНИЕ А**

**Класс RoutesConfig.java отвечающий за маршрутизацию запросов**

```
package com.vixr.apigateway.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.gateway.route.RouteLocator;
import
org.springframework.cloud.gateway.route.builder.RouteLocatorBuilde
r;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RoutesConfig {

  @Value("${route.url.order}")
  private String orderUrl;

  @Value("${route.url.auth}")
  private String authUrl;

  @Value("${route.url.notification}")
  private String notificationUrl;

  @Value("${route.url.product}")
  private String productUrl;

  @Value("${route.url.feedback}")
  private String feedbackUrl;

  @Bean
  public RouteLocator customRouteLocator(RouteLocatorBuilder
builder) {
    return builder.routes()
        .route("auth_route", r -> r.path("/api/auth/**")
            .filters(f -> f
                .rewritePath("/api/auth/(?<remaining>.*)",
"/${remaining}")
                .filter(new PathFilter())
            )
            .uri(authUrl))

        .route("order_route", r -> r.path("/api/order/**")
            .filters(f -> f
                .rewritePath("/api/order/(?<remaining>.*)",
"/${remaining}")
                .filter(new PathFilter())
            )
            .uri(orderUrl))
```

```
        .route("notification_route", r ->
r.path("/api/notification/**")
            .filters(f -> f
                .rewritePath("/api/notification/(?<remaining>.*)",
"/${remaining}")
                .filter(new PathFilter())
            )
            .uri(notificationUrl))

        .route("product_route", r -> r.path("/api/product/**")
            .filters(f -> f
                .rewritePath("/api/product/(?<remaining>.*)",
"/${remaining}")
                .filter(new PathFilter())
            )
            .uri(productUrl))

        .route("feedback_route", r -> r.path("/api/feedback/**")
            .filters(f -> f
                .rewritePath("/api/feedback/(?<remaining>.*)",
"/${remaining}")
                .filter(new PathFilter())
            )
            .uri(feedbackUrl))
        .build();
  }

}
```

**ПРИЛОЖЕНИЕ Б**

**Класс AuthController.java**

```
package com.vixr.auth.api.controller;

import com.vixr.auth.api.dto.request.InviteUserRequest;
import com.vixr.auth.api.dto.request.RefreshPasswordRequest;
import com.vixr.auth.api.dto.request.ResetPasswordRequest;
import com.vixr.auth.api.dto.request.SignInUserRequest;
import com.vixr.auth.api.dto.request.TokenRefreshRequest;
import com.vixr.auth.api.dto.response.SignedInUserResponse;
import com.vixr.auth.api.facade.AuthFacade;
import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import static org.springframework.http.MediaType.TEXT_HTML;

@Slf4j
@CrossOrigin
@RestController
@RequiredArgsConstructor
@RequestMapping("/v1/auth")
public class AuthController {

  private final AuthFacade facade;

  @PostMapping("/invite")
  public ResponseEntity<Void> inviteUser(@Valid @RequestBody
InviteUserRequest request) {
    log.info("Invite user: {} request", request.getEmail());
    facade.inviteUser(request);
    return ResponseEntity.noContent().build();
  }

  @PostMapping("/signin")
  public ResponseEntity<SignedInUserResponse> signInUser(@Valid
@RequestBody SignInUserRequest request) {
    SignedInUserResponse response = facade.signInUser(request);
    log.info("User [{}] signed in.", response.getUsername());
    return ResponseEntity.ok(response);
  }
```

```
  @PostMapping("/refresh-token")
  public ResponseEntity<SignedInUserResponse>
refreshAccessToken(@Valid @RequestBody TokenRefreshRequest
request) {
    log.info("Refresh token request: {}",
request.getRefreshToken());
    SignedInUserResponse response = facade.refreshTokens(request);
    return ResponseEntity.ok(response);
  }

  @PostMapping("/reset-password")
  public ResponseEntity<Void> resetPassword(@Valid @RequestBody
ResetPasswordRequest r) {
    facade.resetPassword(r);
    log.info("Password with email: [{}] requested for reset.",
r.getEmail());
    return ResponseEntity.noContent().build();
  }

  @PostMapping("/refresh-password")
  public ResponseEntity<Void> refreshPassword(@Valid @RequestBody
RefreshPasswordRequest r) {
    facade.refreshPassword(r);
    log.info("New password for user with email: [{}] was set",
r.getEmail());
    return ResponseEntity.noContent().build();
  }

  @GetMapping("/confirmation")
  public ResponseEntity<String>
confirmEmail(@RequestParam("token") String token) {
    facade.confirmEmail(token);
    log.info("Email was confirmed. token: [{}]", token);
    return
ResponseEntity.ok().contentType(TEXT_HTML).body("<h1>Now you can
close window</h1>");
  }
}
```

## ПРИЛОЖЕНИЕ В

## Класс конфигурации RabbitMq

```
package com.vixr.notification.config;

import lombok.RequiredArgsConstructor;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.core.QueueBuilder;
import
org.springframework.amqp.rabbit.config.SimpleRabbitListenerContain
erFactory;
import
org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import
org.springframework.amqp.support.converter.Jackson2JsonMessageConv
erter;
import
org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@RequiredArgsConstructor
public class RabbitMQConfig {

    @Value("${rabbitmq.notification.exchange}")
    private String exchange;

    @Value("${rabbitmq.notification.queue}")
    private String queue;

    @Value("${rabbitmq.notification.queue-dlq}")
    private String queueDlq;

    @Value("${rabbitmq.notification.routing-key}")
    private String routingKey;

    @Value("${rabbitmq.notification.routing-key-dlq}")
    private String routingKeyDlq;

    private final ConnectionFactory connectionFactory;

    private static final String X_DEAD_LETTER_EXCHANGE = "x-dead-
letter-exchange";
```

```
    private static final String X_DEAD_LETTER_ROUTING_KEY = "x-
dead-letter-routing-key";

    @Bean
    DirectExchange exchange() {
        return new DirectExchange(exchange);
    }

    @Bean
    Queue dlq() {
        return QueueBuilder.durable(queueDlq).build();
    }

    @Bean
    Queue queue() {
        return
QueueBuilder.durable(queue).withArgument(X_DEAD_LETTER_EXCHANGE,
exchange)
                .withArgument(X_DEAD_LETTER_ROUTING_KEY,
routingKeyDlq).build();
    }

    @Bean
    Binding binding() {
        return
BindingBuilder.bind(queue()).to(exchange()).with(routingKey);
    }

    @Bean
    public SimpleRabbitListenerContainerFactory
simpleRabbitListenerContainerFactory() {
        SimpleRabbitListenerContainerFactory factory = new
SimpleRabbitListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory);
        factory.setMessageConverter(jacksonConverter());
        return factory;
    }

    @Bean
    public AmqpTemplate amqpTemplate() {
        RabbitTemplate rabbitTemplate = new
RabbitTemplate(connectionFactory);
        rabbitTemplate.setMessageConverter(jacksonConverter());
        return rabbitTemplate;
    }

    @Bean
    public MessageConverter jacksonConverter() {
        return new Jackson2JsonMessageConverter();
    }
}
```

## ПРИЛОЖЕНИЕ Г

## Dockerfile сервиса заказов

```
FROM maven:3.9.6-eclipse-temurin-21-alpine AS build
WORKDIR /app
COPY pom.xml ./
RUN mvn dependency:go-offline
COPY src ./src
RUN mvn package

FROM openjdk:21-slim
WORKDIR /app
COPY --from=build /app/target/order-1-exec.jar ./app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

## ПРИЛОЖЕНИЕ Д

### Файл docker-compose.yml для развертывания всего приложения

```
version: '3'

services:
  postgre:
    image: postgres:13-alpine
    restart: on-failure
    container_name: postgre
    volumes:
      - ./local-initdb.sql:/docker-entrypoint-initdb.d/local-
initdb.sql
    ports:
      - '5432:5432'
    environment:
      POSTGRES_USER: root
      POSTGRES_PASSWORD: root

  rabbit:
    image: rabbitmq:3.8-management
    hostname: rabbitmq
    container_name: rabbit
    ports:
      - "5672:5672"
      - "15672:15672"
    environment:
      RABBITMQ_DEFAULT_USER: admin
      RABBITMQ_DEFAULT_PASS: admin
    volumes:
      - ./rabbitmq:/var/lib/rabbitmq


  api-gateway:
    container_name: api-gateway
    depends_on:
      auth:
        condition: service_started
    build:
      context: ./api-gateway
    ports:
      - "8080:8080"
    environment:
      AUTH_HOST: auth:8080
      ORDER_HOST: order:8080
      CUSTOMER_HOST: customer:8080
      NOTIFICATION_HOST: notification:8080
      INTEGRATION_HOST: integration:8080
      PRODUCT_HOST: product:8080
```

```yaml
auth:
  container_name: auth
  depends_on:
    postgre:
      condition: service_started
  build:
    context: ./auth
  environment:
    USER_DB_HOST: postgre
    NOTIFICATION_URL: notification

order:
  container_name: order
  depends_on:
    api-gateway:
      condition: service_started
  build:
    context: ./order
  environment:
    ORDER_DB_HOST: postgre
    NOTIFICATION_URL: notification


notification:
  container_name: notification
  depends_on:
    api-gateway:
      condition: service_started
  build:
    context: ./notification
  environment:
    NOTIFICATION_DB_HOST: postgre

product:
  container_name: product
  depends_on:
    api-gateway:
      condition: service_started
  build:
    context: ./product
  environment:
    PRODUCT_DB_HOST: postgre
    NOTIFICATION_URL: notification
```

## ПРИЛОЖЕНИЕ Е

### Класс RestSender.java для отправки уведомлений при помощи Rest API

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

public class RestSender {
    private static final String SERVER_URL =
"http://localhost:8080/api/data";
    private RestTemplate restTemplate = new RestTemplate();

    public long sendRequest(byte[] data) {
        long startTime = System.currentTimeMillis();
        ResponseEntity<String> response =
restTemplate.postForEntity(SERVER_URL, data, String.class);
        long endTime = System.currentTimeMillis();
        return endTime - startTime;
    }

    public void prepareMessage() {
        byte[] data = new byte[1024 * 1024]; // 1 MB
        for (int i = 0; i < 10; i++) {
            long latency = sendRequest(data);
            System.out.println("Latency: " + latency + " ms");
        }
    }
}
```

## ПРИЛОЖЕНИЕ Ж

### Класс DataController.java для получения отправленного сообщения

```
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class DataController {
    @PostMapping("/api/data")
    public String receiveData(@RequestBody byte[] data) {
        // Обработка данных
        return "Received";
    }
}
```

## ПРИЛОЖЕНИЕ 3

### Bash скрипт для отслеживания нагрузки на процессов

```
#!/bin/bash

PID=<PID> # Замените <PID> на PID вашего процесса
OUTPUT_FILE="cpu_usage.txt"

echo "Начинаем запись "

echo "Time,%CPU" > $OUTPUT_FILE

for i in {1..100}
do
  # Получение текущего времени
  CURRENT_TIME=$(date "+%H:%M:%S")

  # Получение использования процессора для заданного PID
  CPU_USAGE=$(ps -p $PID -o %cpu=)

  # Запись данных в файл
  echo "$CURRENT_TIME,$CPU_USAGE" >> $OUTPUT_FILE

  # Задержка 1 секунда
  sleep 1
done

echo "Запись завершена."
```

## ПРИЛОЖЕНИЕ И

### Отчет о проверке сервисом «Антиплагиат»

# Отчет о проверке

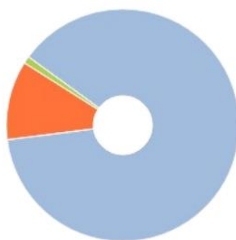
**Автор:** Ермош Константин Владимирович

**Название документа:** Диссертация

**Проверяющий:** Захаров Павел Алексеевич

**Организация:** Сибирский федеральный университет

**РЕЗУЛЬТАТЫ ПРОВЕРКИ**



Совпадения:
10,6%

Оригинальность:
88,21%

Цитирования:
1,19%

Самоцитирования:
0%

«Совпадения», «Цитирования», «Самоцитирования», «Оригинальность» являются отдельными показателями, отображаются в процентах и в сумме дают 100%, что соответствует проверенному тексту документа.

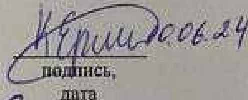
*Министерство науки и высшего образование РФ*
*Федеральное государственное автономное*
*образовательное учреждение высшего образования*
**«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**
Институт космических и информационных технологий
Кафедра «Вычислительная техника»

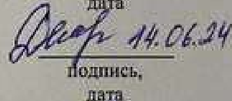
УТВЕРЖДАЮ
Заведующий кафедрой
_______________ О.В. Непомнящий
подпись
« 14 » 06 2024 г.

# МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Организация взаимодействия микросервисов в задачах разработки веб-приложений
09.04.01 Информатика и вычислительная техника
09.04.01.4 Технология разработки программного обеспечения

Руководитель 10.06.24 доцент, канд. техн. наук М.С. Медведев
подпись, дата

Выпускник  10.06.24 К.В. Ермош
подпись, дата

Рецензент  14.06.24 доцент, канд. техн. наук В.Г. Демин
подпись, дата

Нормоконтролер 10.06.24 М.С. Медведев
подпись, дата

Красноярск 2024