МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра Прикладная математика и информатика
(наименование)

09.03.03 Прикладная информатика
(код и наименование направления подготовки / специальности)

Корпоративные информационные системы
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему Разработка программного интерфейса унифицированного доступа к распределенной реактивной информационной системе

Обучающийся А. В. Ерофеев
(Инициалы Фамилия) (личная подпись)

Руководитель канд. пед.наук, доцент, Т. А. Агошкова
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Консультант Т. С. Якушева
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамили

Тольятти 2023

Аннотация

Тема выпускной квалификационной работы – «Разработка программного интерфейса унифицированного доступа к распределенной реактивной информационной системе» в ООО «Квартплата 24».

Актуальность работы заключается в необходимости перехода от устаревших архитектур и инструментов к более современным решениям.

Целью данной работы является разработка программного интерфейса унифицированного доступа к распределенной реактивной информационной системе (РРИС), которая представляет собой совокупность разнородных источников данных (баз данных, файловых систем, веб-сервисов и т.д.), объединенных в единую сетевую инфраструктуру.

Разработан программный интерфейс для унифицированного доступа к распределенной реактивной информационной системе, который позволяет обмениваться данными между сервисами экосистемы с использованием асинхронного программирования и веб-сервисов. Решена проблема производительности, масштабируемости и отказоустойчивости распределенной реактивной информационной системы.

Выпускная квалификационная работа состоит из 55 страниц текста, 22 рисунков, 4 таблиц и 12 источников.

Abstract

The title of the bachelor's thesis is «Development of a software interface for unified access to a distributed reactive information system» at LLC «Kvartplata 24».

The relevance of the work lies in the need to move from outdated architectures and tools to more modern solutions.

The object of research is a software interface for a distributed reactive information system.

The subject of research is methods and technologies for developing a software interface for unified access to a distributed reactive information system.

The goal of the graduation qualification work is to develop a software interface for unified access to a distributed reactive information system based on the analysis of existing approaches and technologies. Research methods - asynchronous programming, web services, microservice architecture, reactive architecture.

A software interface for unified access to a distributed reactive information system has been developed, which allows data exchange between services of the ecosystem using asynchronous programming and web services. The problem of performance, scalability and fault tolerance of a distributed reactive information system has been solved.

The graduation qualification work consists of 55 pages of text, 22 figures, 4 tables and 12 sources.

Оглавление

Введение 5
Глава 1 Характеристика организации и анализ существующих сервисов 7
1.1 Характеристика и анализ сервисов организации с точки зрения архитектуры 7
1.2 Характеристика реактивной архитектуры 12
1.3 Обоснование перехода с проприетарного протокола передачи данных RAP на протокол gRPC 13
1.4 Постановка задачи на разработку 20
Глава 2 Построение интерфейса унифицированного доступа 22
2.1 Требования к разрабатываемому сервису 22
2.2 Проектирование архитектуры сервиса 23
2.3 Описание инструментов разработки сервиса 25
2.4 Разработка диаграммы классов сервиса 30
Глава 3 Разработка интерфейса унифицированного доступа 35
3.1 Выбор средств реализации 35
3.2 Реализация основных модулей сервиса и интегрирование его с другими сервисами экосистемы 37
3.3 Тестирование работоспособности интерфейса унифицированного доступа 43
Заключение 46
Список используемой литературы 47
Приложение А Фрагмент конфигурации Envoy-proxy 49
Приложение Б Листинг кода класса Launcher 52
Приложение В Фрагмент кода класса RestRequestListener 53
Приложение Г Фрагмент кода класса RpcRequestListener 54

Введение

В современном информационном обществе актуальной задачей является обеспечение эффективного доступа к различным источникам данных, которые могут быть распределены по разным узлам сети и иметь разную структуру и формат.

По мере роста клиентской базы, возрастает и нагрузка на сервисы в целом, что приводит к замедлению обработки и передачи ответов на запросы, что может негативно сказаться на клиентском опыте. Также происходит и увеличение числа сервисов, что делает проблематичным работу с их API, и приводит к трудностям в процессе разработки.

Для решения этой задачи необходимо разработать программный интерфейс, который позволит унифицировать и абстрагировать доступ к данным, скрыть детали реализации и хранения данных от пользователя и обеспечить высокую производительность и надежность обмена данными. Такой программный интерфейс должен быть адаптивным, то есть способным подстраиваться под разные типы данных и запросов, а также реактивным, то есть способным отвечать на изменения в данных и среде в реальном времени.

Актуальность работы заключается в необходимости перехода от устаревших архитектур и инструментов к более современным, производительным решениям, которые позволят удовлетворить нужды клиентов.

Объектом исследования является программный интерфейс унифицированного доступа к распределенной реактивной информационной системе.

Предметом исследования являются методы и технологии для разработки программного интерфейса унифицированного доступа к распределенной реактивной информационной системе.

Цель выпускной квалификационной работы – модернизация существующей экосистемы сервисов посредством разработки программного

интерфейс для унифицированного доступа к распределенной реактивной информационной системе на основе анализа существующих подходов и технологий, с учетом принципов реактивной архитектуры.

Для достижения поставленной цели необходимо решить следующие задачи:

— изучить существующие подходы и технологии для организации доступа к данным в распределенных системах.

— спроектировать архитектуру программного интерфейса унифицированного доступа к РРИС, выбрать подходящие протоколы и форматы обмена данными.

— разработать алгоритмы и методы для реализации программного интерфейса унифицированного доступа к распределённой информационной системе, обеспечивающие адаптивность и реактивность.

— произвести тестирование и оценку эффективности разработанного программного интерфейса унифицированного доступа.

Научная новизна работы заключается в разработке программного интерфейса унифицированного доступа к РРИС, который позволяет абстрагировать пользователя от деталей распределения и хранения данных, адаптироваться к различным типам данных и запросов, реагировать на изменения в данных и среде в реальном времени.

В качестве методов исследования выступают микросервисная и реактивная архитектуры.

Работа состоит из трех глав. В первой главе описывается экосистема сервисов в ООО «Квартплата 24», дается характеристика текущим решениям и обосновывается причина отказа от них. Во второй главе обозначаются требования к разрабатываемому продукту, описывается процесс проектирования архитектуры сервисов, осуществляется выбор инструментов реализации и разработка диаграмм класса сервиса. В третьей главе описываются инструменты разработки и процесс разработки интерфейса унифицированного доступа.

Глава 1 Характеристика организации и анализ существующих сервисов

1.1 Характеристика и анализ сервисов организации с точки зрения архитектуры

ООО «Квартплата 24» является IT-компанией, оказывающей услуги в сфере жилищно-коммунального хозяйства для товариществ собственников жилья, управляющих и ресурсоснабжающих организаций в 67 субъектах РФ.

Основные задачи, решаемые компанией в данной сфере, являются расчет и учет платы за ЖКХ, прием и последующее расщепление платежей по организациям-получателям, а также взыскание долгов в соответствии с законодательством Российской Федерации на всей ее территории.

Для решения описанных задач используется экосистема из более чем десяти облачных сервисов, тесно связанных между собой. Все они представляют собой комплексное решение для расчета платы за жилищно-коммунальные услуги, формирование платежных документов (квитанций на оплату), распределения платежей, работы с дебиторской задолженностью и контроля внутренних процессов организации.

Все сервисы можно разделить на две основные группы – внутренние, предназначенные для поддержания технологический процессов, и клиенто-ориентированные – те, с которыми клиенты непосредственно взаимодействуют.

Схематичное изображение основных сервисов экосистемы ООО «Квартплата 24» представлено на рисунке 1.

Ниже дано описание основных, наиболее крупных и важных сервисов, с которыми взаимодействуют клиенты:

— биллинговая система – осуществляет автоматизированный расчет платы за жилищно-коммунальные услуги, также отвечает за расщепление платежей без участия УО и ТСЖ на конечных получателей – РСО и СО;

— личный кабинет жителя – предназначен для плательщиков, предоставляет своевременную информацию о начислениях по ЖКУ, возможность оплаты, внесения показаний, просмотр платежных документов и т.д.;

— сервис аналитики – предназначен для руководителей обслуживающих организаций, и предоставляет им информацию о начислениях, поступлении и расщеплении платежей по управляемому жилому фонду;

— сервис работы с должниками (он же СРД) – предназначен для проведение досудебной и судебной работы с должниками клиентов.

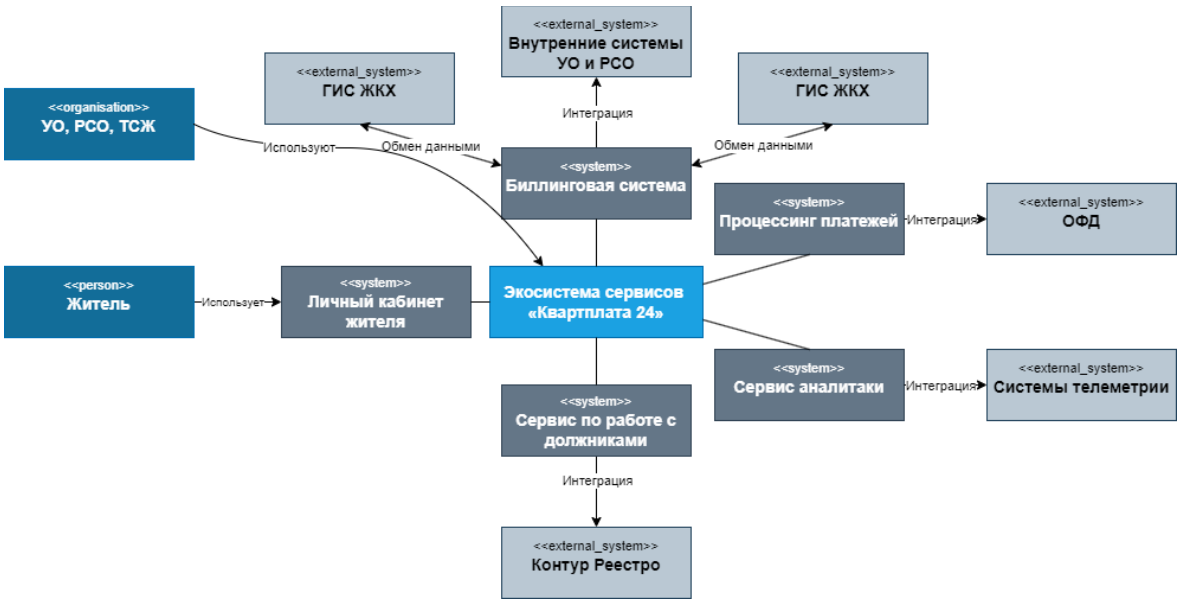

Рисунок 1 – Схема экосистемы сервисов ООО «Квартплата 24»

Помимо этих сервисов, также есть и те, что обеспечивают интеграцию с другими внешними сервисами и/или отвечающие за внутренние процессы экосистемы:

— платежный сервис – специализируется на приеме платежей из более чем 40 банков и платежных систем. Данный сервис интегрирован с биллинговой системой;

— ОФД-шлюз – сервис фискализации платежей, обеспечивает интеграцию с онлайн-кассами, при оплате моментально формируя чеки и передает их в ФНС и плательщику;

— ГИС-шлюз – обеспечивает постоянный обмен данными с Государственной информационной системой жилищно-коммунального хозяйства (ГИС ЖКХ) в соответствии с законодательством;

— сервис телеметрии – обеспечивает интеграцию с системами телеметрии (например, Элдис);

— отдельные интеграции с внешними облачными сервисами и системами (такими как Суд РФ, ПО Контакт, Умное ЖКХ, Контур Реестро и т.д.).

Следует также отдельно выделить такой внутренний сервис, как Remote Access Point (RAP). Как следует из названия, он представляет собой единую точку доступа к сервисам экосистемы, через него проходят все запросы, адресованные сервисам, направленным на взаимодействие с клиентами.

На рисунке 2 представлена схема, демонстрирующая взаимодействие основных сервисов.

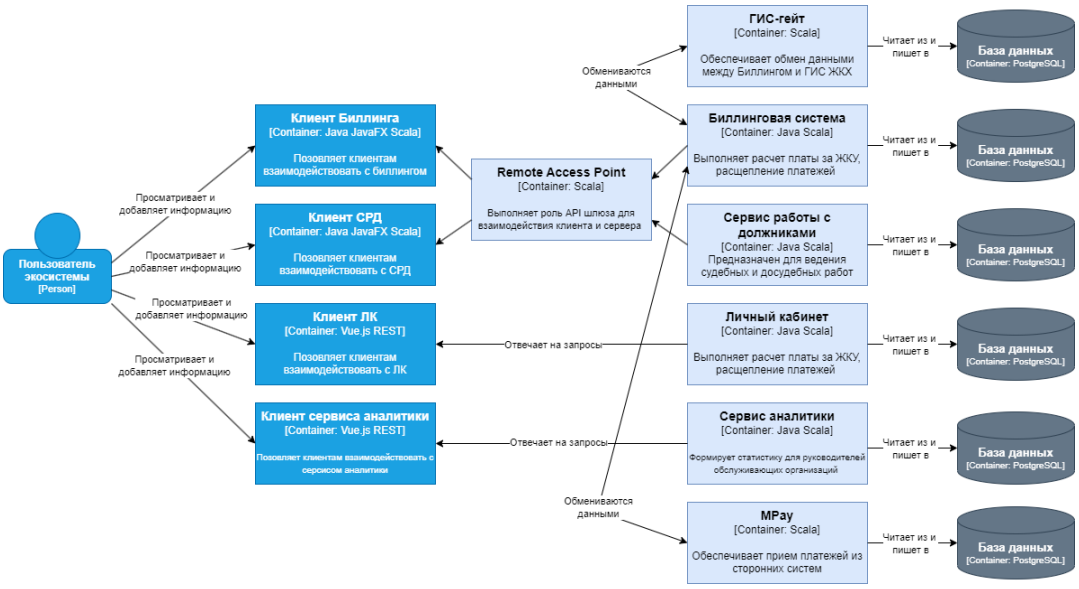

Рисунок 2 – Схема, отражающая взаимодействие основных сервисов ООО «Квартплата 24»

Большая часть сервисов экосистемы представляет собой программное обеспечение, построенное по принципам монолитной архитектуры. Такая архитектура считается традиционной моделью программного обеспечения, в идеале работающего автономно от других приложений.

Монолитной архитектуре присущи определенные преимущества, а именно:

— простое развёртывание – использование одного исполняемого файла или каталога упрощает развертывание;

— разработка – единая кодовая база значительно упрощает этот процесс;

— упрощенное тестирование и отладка – из-за того, что монолит является единым модулем, сквозное тестирование можно проводить быстрее;

— производительность – кодовая база централизована, соответственно, один интерфейс API зачастую может выполнять ту же функцию, которую при микросервисной архитектуре выполняют многочисленные API.

Но основная часть этих плюсов заканчивается в тот момент, когда кодовая база начинает разрастаться до слишком больших размеров, что приводит к появлению следующих недостатков:

— снижение скорости разработки и тестирования – большая кодовая база замедляет соответствующие процессы;

— масштабируемость – отдельные компоненты либо крайне сложно, либо вовсе невозможно масштабировать;

— сложность внедрения технологий – любые изменения в инфраструктуре или языке разработки приводят к большим временным затратам;

— развертывание – даже самое небольшое изменение в кодовой базе требует развертывания всего приложения.

Также постепенное разрастание монолита приводит к тому, что доменная область приложения приобретает крайне запутанный вид, что затрудняет

введение новых разработчиков в работу с таким сервисом [2], [4]. На рисунке 3 представлено схематичное отображение запутанного монолита на примере домена биллинговой системы.

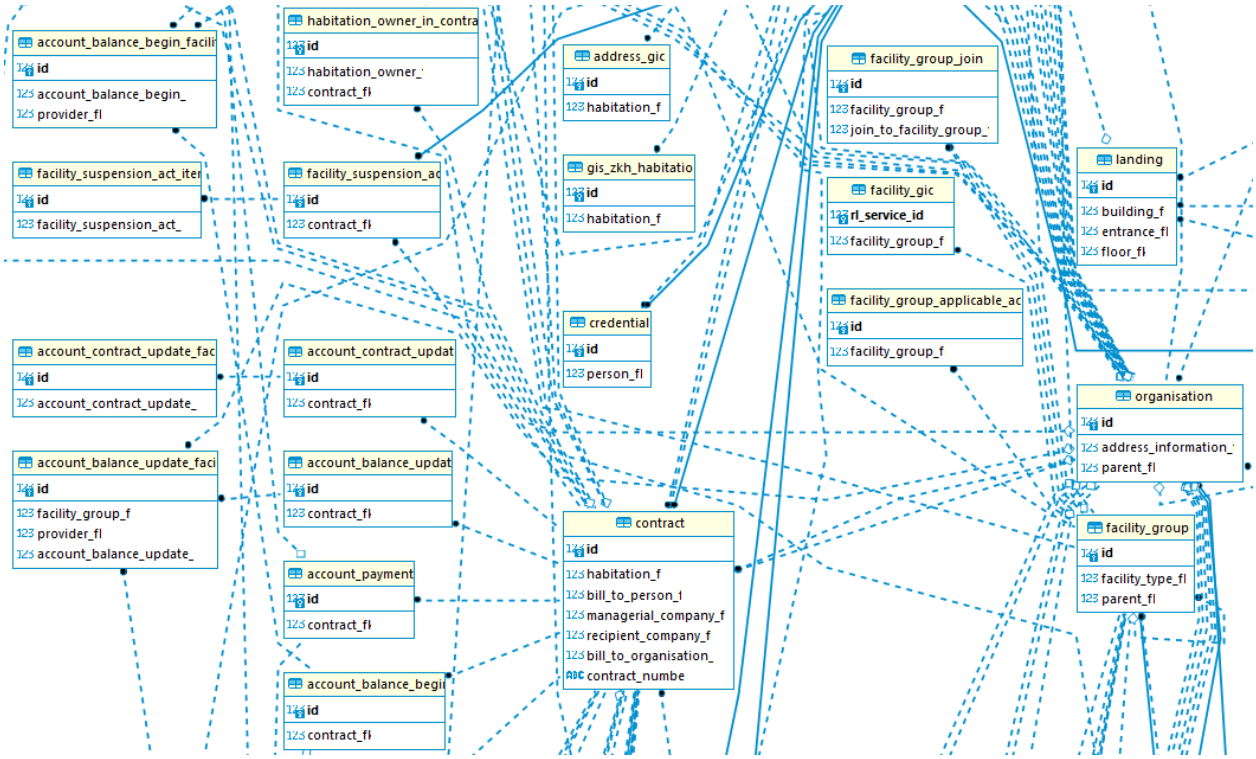

Рисунок 3 – Схема запутанного монолита на примере домена биллинговой системы

Почти всем основным сервисам экосистемы присущи описанные признаки и проблемы монолитной архитектуры. Зачастую, изменения в одном сервисе требуют внесений соответствующих изменений в других, полноценная работа некоторых сервисов невозможна без развернутого экземпляра другого сервиса. Например, СРД не сможет рассчитать сумму задолженности без биллинговой системы, так как именно в ней и происходит ее расчет, несмотря на то что СРД также располагает информацией о финансовом положении должника.

Основываясь на описанных проблемах, можно сделать вывод, что вся экосистема сервисов ООО «Квартплата 24» архитектурно подпадает под

определение «распределенный монолит» [20]. Сервисы развернуты в разных контейнерах и серверах, но все равно имеют очень тесную связь друг с другом.

Таким образом, были рассмотрены основные сервисы экосистемы ООО «Квартплата 24» и дана их подробная характеристика.

1.2 Характеристика реактивной архитектуры

Одним из способов избавления от анти-паттерна «распределенный монолит» является постепенный переход от монолитной архитектуры и ее разновидностей к микросервисной и реактивной архитектуре.

Системы, построенные в соответствии с принципами реактивной архитектуры, отличаются быстрой реакцией на запросы, высокой адаптивностью и возможностью простого масштабирования. Основой всех этих преимуществ перед другими подходами является передача данных с помощью асинхронных неблокирующих сообщений.

На рисунке 4 представлены главные принципы реактивной архитектуры.

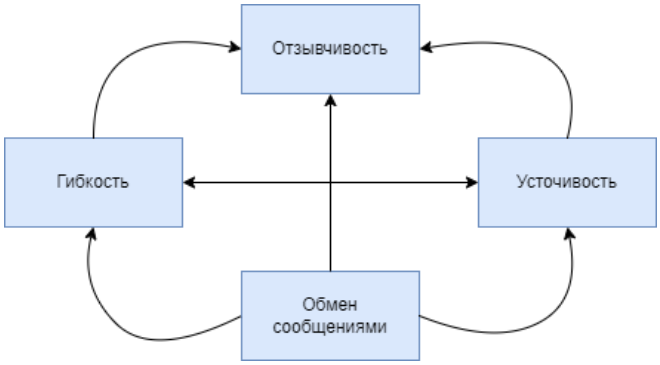

Рисунок 4 – Схема, демонстрирующая основные принципы реактивных систем

Отзывчивость обеспечивается за счет использование асинхронных сообщений при передаче данных между компонентами информационной системы, что делает эти компоненты слабосвязанными. Такой подход

позволяет управлять нагрузкой, осуществлять мониторинг очереди сообщений и предотвращать перегрузку получателя данных их поставщиком.

Гибкость системы достигается за счет динамического масштабирования при увеличении нагрузки. Например, в период расчета многократно возрастает нагрузка на сервис, отвечающий за проведение самого расчета, поэтому, помимо двух существующих экземпляров сервиса будут подняты еще два, и между ними уже будет распределяться нагрузка.

Устойчивость к ошибкам обеспечивается за счет использования паттерна Circuit Breaker, переотправку запросов и других подходов.

Следование принципам реактивной архитектуры позволит создавать системы, быстро реагирующие на запросы пользователей. Помимо улучшения клиентского опыта взаимодействия с приложениями, повышения его уровня доверия к продукту, облегчается и взаимодействие с ними разработчиков за счет простоты определения и устранения ошибок. Все это способствует быстрому выходу продукта на рынок программного обеспечения по сравнению с конкурентами, использующих другие подходы проектирования.

1.3 Обоснование перехода с проприетарного протокола передачи данных RAP на протокол gRPC

Как уже было сказано в пункте 1.2, сервис RAP (Удаленная точка доступа) осуществляет коммуникацию между клиентской и серверной частью таких сервисов, как биллинговая система и сервис работы с должниками.

RAP использует проприетарный протокол передачи данных, реализованный поверх Akka I/O с использованием протокола TCP и Java-сериализации. Помимо коммуникации, сервис играет роль load-balancer-a, распределяя нагрузку между запущенными экземплярами сервисов.

С постепенным ростом клиентской базы начала возрастать и нагрузка на сервисы в целом, увеличился объем данных, с которыми приходится взаимодействовать, что породило проблему замедления работы этих сервисов,

как при передаче данных от сервера к клиентской стороне, так и при выполнении различных повседневных задач. Все это неизбежно приводит к недовольству со стороны клиентов. Для решения этой проблемы было принято решение начать поиск альтернатив существующему подходу. Одним из найденных вариантов был фреймворк gRPC.

Фреймворк gRPC – это современный высокопроизводительный инструмент для удаленного вызова процедур (RPC – Remote Procedure Call), который может работать в любой среде. Он позволяет эффективно соединять сервисы внутри и между центрами обработки данных с поддержкой балансировки нагрузки, трассировки, проверки состояния и аутентификации. gRPC использует Protocol Buffers в качестве языка описания интерфейса (IDL) и формата обмена сообщениями. Protocol Buffers - это механизм сериализации структурированных данных, который позволяет определять структуру данных в специальном текстовом файле с расширением «.proto» и генерировать классы доступа к данным на разных языках [14].

Но, прежде чем отказываться от текущего решения и приступать к реализации нового подхода, необходимо убедиться, что использование gRPC – приемлемый вариант, и он не окажется хуже или идентичным относительно текущего подхода. Было решено провести несколько различных тестов.

Для тестирования было решено взять такую сущность, как «контракт» – виртуальный договор, лицевой счет (далее л/с). Это, по сути, центральная сущность, вокруг которой крутится большинство операций, она имеет множество полей причем как примитивных типов, таких как строки или целочисленные значения, так и связывающих его с другими сущностями, что делает ее достаточно тяжеловесной. Для тестирования было создано новое сообщение, отправка которого на сервер запускала процесс тестирования и сбора результатов. Кроме того, тестирование проводилось с использованием одного клиента, то есть, многопользовательская работа не эмулировалась, но это не помешает провести объективную оценку полученных данных.

Алгоритм проведения тестирования выглядит следующим образом:

— со стороны клиента посылается запрос, в момент его отправки пишется текущее время в csv-файл;

— на сервере при получении запроса формируется массив данных (в данном случае, массив из л/с), размер которых составляет 10, 50, 100, 250, 500, 1000 и 2500 элементов соответственно;

— создаются 200 сообщений, состоящих из сформированного массива л/с и после чего они отправляются на клиент;

— клиент обрабатывает ответ от сервера и пишет в csv-файл время получения ответа.

После, полученный csv-файл используется для формирования отчетов и графиков, позволяющих оценить производительность двух методов.

Тестирование проводилось с использованием прямого и удаленного доступа к системе. Замерялось время выполнения таких операций, как чтение и запись через протоколы RAP и gRPC. Также было решено замерить время выполнения тех же операций с использованием gRPC Streams – этот инструмент позволяет отправлять несколько двунаправленных сообщений через одно соединение, используя протокол HTTP/2, что полезно при сценариях, когда необходимо передавать большие объемы данных, или поддерживать долгоживущие соединения между сервером и клиентами.

На рисунке 5 изображен график, отражающий отношение лицевых счетов в запросе ко времени, которое было затрачено на передачу данных.

На основе данного графика сравним протокол gRPC и RAP. Сразу можно отметить, что с увеличением числа л/с в запросе время, затрачиваемое на передачу данных через RAP, растет нелинейно, тогда как у gRPC достаточно стабилен. Если сравнить время выполнения операции чтения на предельной нагрузке (2,5 тыс. л/с в запросе), то можно увидеть, что скорость передачи данных через gRPC почти в десять раз превышает скорость передачи через RAP.

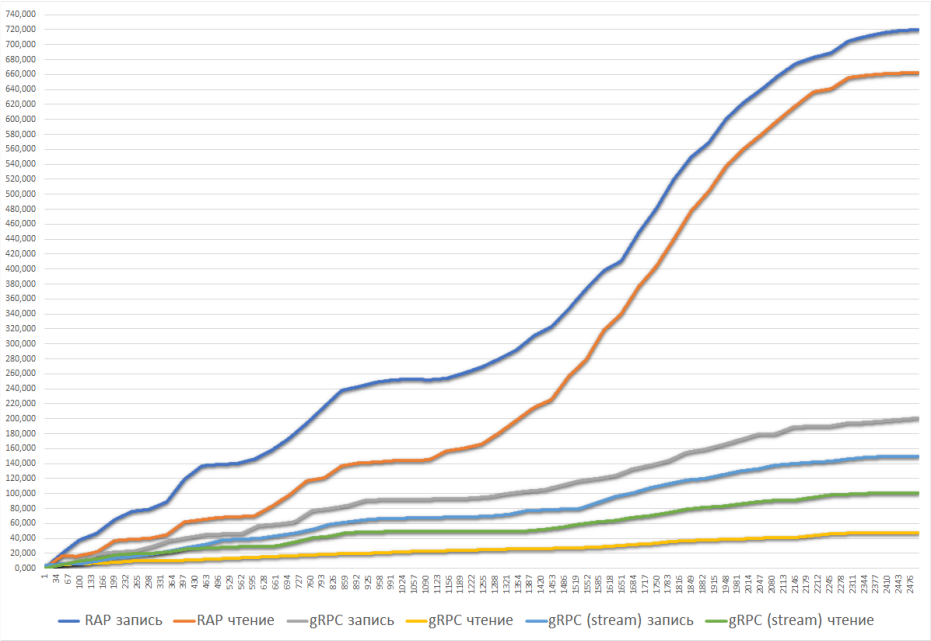

Рисунок 5 – График, демонстрирующий отношение кол-ва л/с и времени, затраченного на их обработку сервером

Теперь рассмотрим другой пример на основе запросов с конкретным размером массива л/с, а именно ста штук. Такое число как правило чаще всего встречается при реальном взаимодействии клиента с сервером. На рисунке 6 представлен график, демонстрирующий количество времени, затраченного на передачу каждого из двухсот запросов.

Как и в предыдущем случае, рассмотрим операции чтения и записи. Первый вывод, к которому можно прийти, оценивая данные на этом графике, это то, что скорость передачи данных через gRPC в четыре раза превышает скорость передачи данных через RAP. Кроме того, следует обратить внимание на скачкообразное время выполнения запросов через RAP – обработка то идет относительно быстро, то, наоборот, сильно замедляется. Самый быстро и самый медленно обработанный запрос через RAP отличаются по времени почти в два-три раза, что говорит о крайне низкой стабильности работы. В случае с gRPC таких проблем не наблюдается, все отклонения можно списать на погрешности в процессе тестирования.

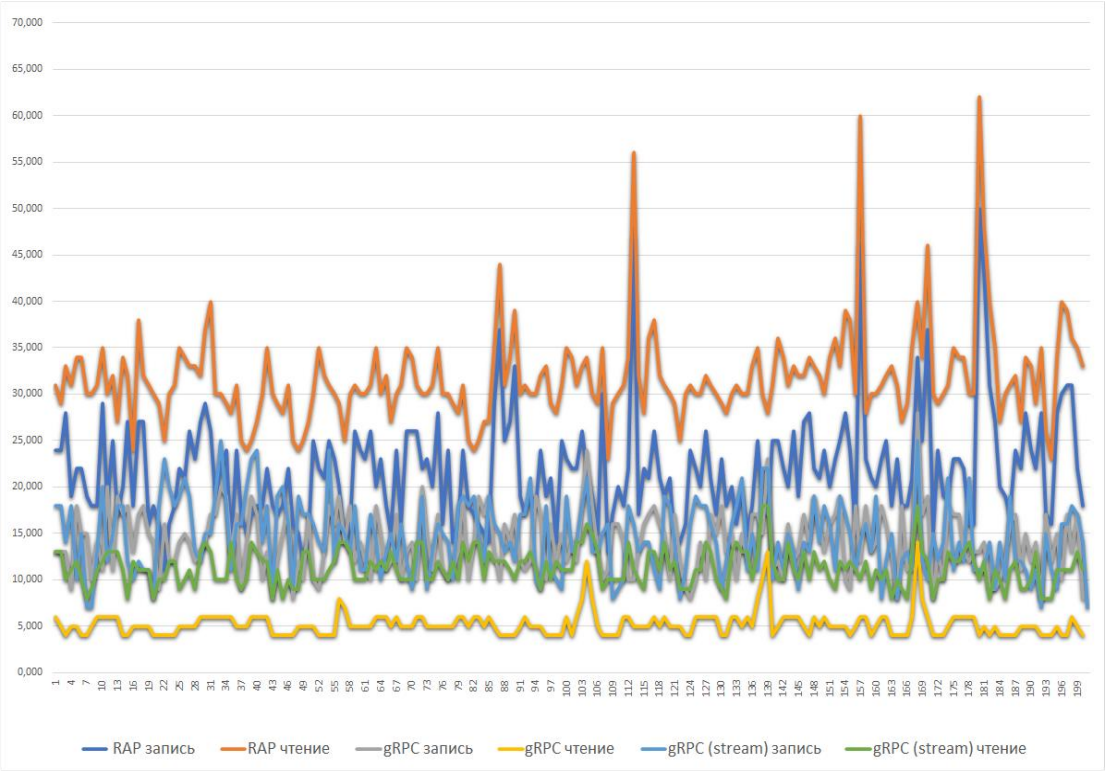

Рисунок 6 – График, отражающий кол-во времени, затрачиваемого на выполнение каждого запроса для массива размером 100 л/с

Далее рассмотрим статистику по времени выполнения каждого запроса для всех размеров массивов л/с. Гистограмма с этими данными представлена на рисунке 7.

На горизонтальной оси обозначен номер запроса, на вертикальной – время выполнения в миллисекундах.

Глядя на данный график, можно увидеть большую дисперсию, возникающую при обработке наиболее тяжеловесных запросов (по 2,5 тыс. л/с в массиве) через RAP, время выполнения может отличаться в 3-4 раза. gRPC и тут демонстрирует крайне стабильную работу и высокую отзывчивость.

Теперь рассмотрим результаты тестирования, проводившегося с использованием удаленного доступа посредством VPN-соединения. Такой подход накладывает некоторые ограничения, такие как пониженная скорость интернета, что позволяет сымитировать более-менее приближенные к реальности условия.

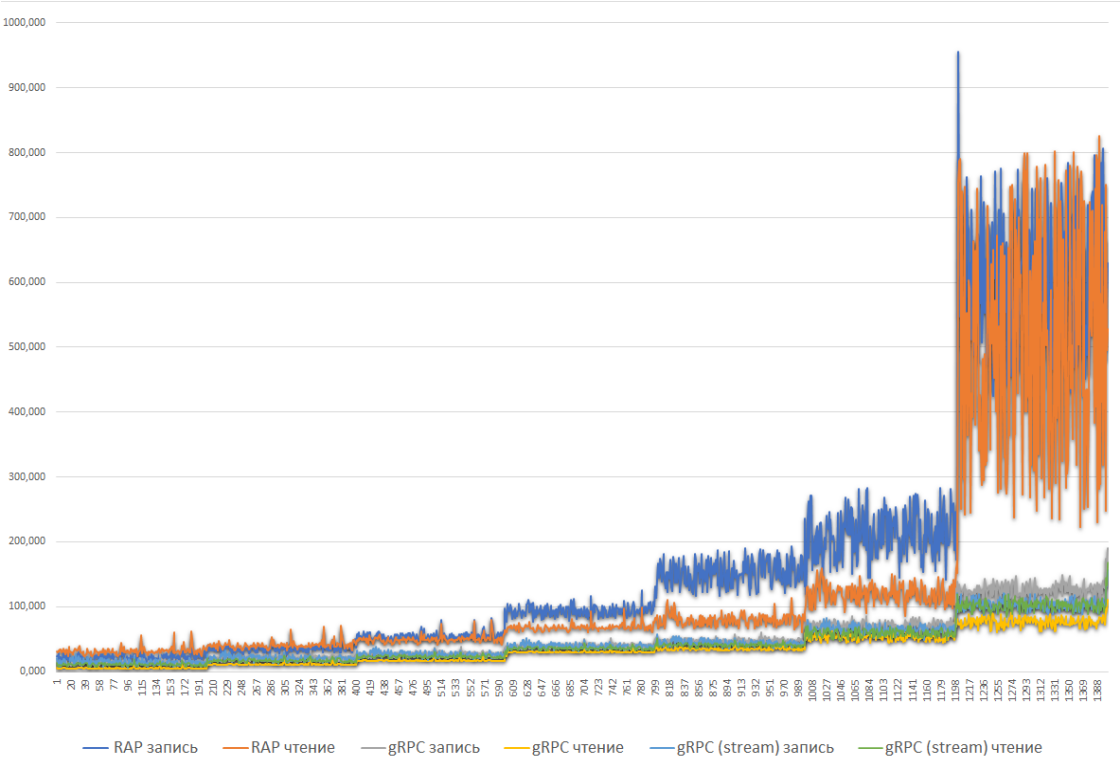

Рисунок 7 – График, демонстрирующий время по каждому запросу для всех семи тестов

На рисунке 8 представлен график, демонстрирующий отношение усредненных значений времени выполнения запросов к количеству лицевых счетов в запросе.

На данном графике можно наблюдать еще больший разрыв по времени обработки запроса между RAP и gRPC. Даже при размере массива в сто л/с время выполнения составляет более секунды, что уже много для клиент-серверного взаимодействия. Если брать предельные значения в виде 2,5 тыс. л/с, то тут среднее время уже превышает 8,5 секунд, что является совсем плохим результатом. В случае с gRPC обработка запросов проходила стабильно, а время выполнения операции чтения и записи не превышают секунды.

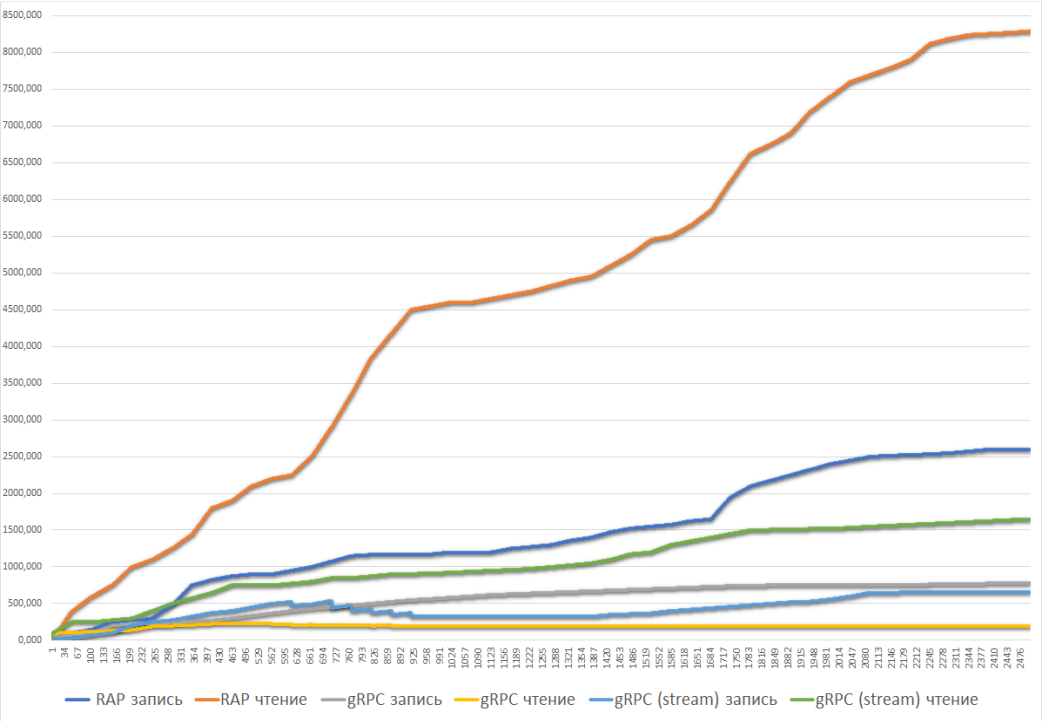

Рисунок 8 – График, демонстрирующий отношение кол-ва л/с ко времени, затрачиваемого на его обработку в приближенных к реальным условиях

Далее рассмотрим график, демонстрирующий время выполнения всех запросов для всех размеров массивов л/с, он представлен на рисунке 9.

Оценивая данные на этом графике, можно сказать, что дисперсия при передаче данных через RAP такая же высокая, как и в тестах с использованием прямого доступа, но время обработки запроса на предельных размерах массива л/с в 2,5 тыс. штук уже достигает отметки более 14 секунд.

При проведении всех тестов gRPC показал себя гораздо лучше, чем RAP. Время обработки запросов при любом размере массива л/с не превышало отметку в секунду, за исключением предельных размеров массива данных. RAP же, в свою очередь, продемонстрировал себя крайне нестабильным инструментом, так как время обработки запросов зачастую превышало 10 секунд, а разница между временем передачи двух одинаковых запросов могла варьироваться от 2 до 4 раз.

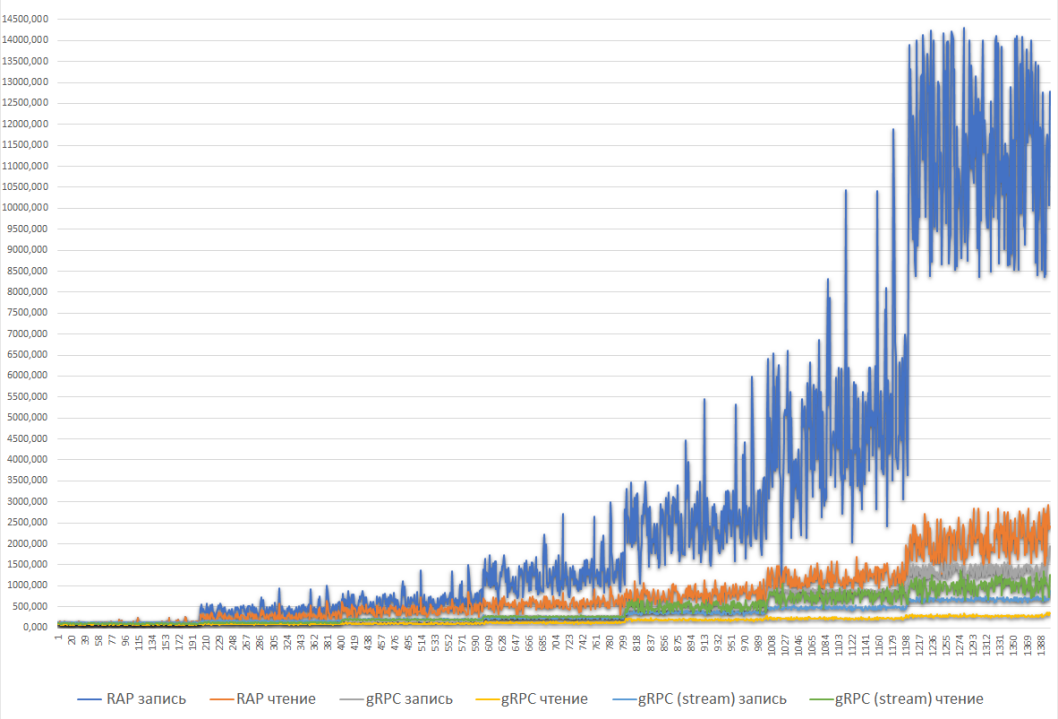

Рисунок 9 – График, отражающий время, затраченное на выполнение каждого запроса для всех семи тестов с использованием удаленного доступа

Проведенное исследование позволяет понять недостатки существующего механизма передачи данных что приводит к необходимости его переработки с использованием современных технологий.

1.4 Постановка задачи на разработку

В виду описанных выше особенностей текущей реализации, было принято решение о переходе с текущего решения в виде сервиса RAP на новый интерфейс унифицированного доступа, который должен отвечать современным требованиям и принципам реактивной архитектуры.

Основной целью данной выпускной квалификационной работы является улучшение архитектуры существующей распределенной информационной системы, которая состоит из тесно связанных монолитных приложений. Для этого предлагается интегрировать в систему программное обеспечение, которое выступит в роли интерфейса унифицированного доступа, что позволит

собрать все API сервисов в едином месте, тем самым упростив разработку и предоставление к нему доступа сторонним разработчикам. Кроме того, использование современных технологий обмена данными позволит значительно ускорить внутренние процессы взаимодействия сервисов между собой, повысит отказоустойчивость.

Основным функционалом разрабатываемого приложения является маршрутизация и контроль трафика, как между клиентом и сервером, там и между внутренними приложениями. Кроме того, данный сервис будет отвечать за аутентификацию и авторизацию пользователей в сервисах, где предполагается их взаимодействие с системой.

Для успешного выполнения данной выпускной квалификационной работы необходимо:

— спроектировать архитектуру интерфейса унифицированного доступа;

— выбрать и описать инструменты реализации приложения, языка программирования и среды разработки;

— разработать интерфейс унифицированного доступа к распределенной системе, протестировать работоспособность;

— описать принципы работы разработанного сервиса [1].

В этой главе были рассмотрели проблемы и недостатки текущего архитектурного подхода к распределенной информационной системе, которая состоит из тесно связанных монолитных приложений. Было также рассмотрено альтернативное решение, которое поможет упростить и ускорить работу существующих сервисов. Была поставлена задача на модернизацию системы путем разработки программного интерфейса унифицированного доступа, который улучшит клиентский опыт работы с приложением и облегчит внутреннее взаимодействие сервисов.

Глава 2 Построение интерфейса унифицированного доступа

2.1 Требования к разрабатываемому сервису

На основании описанных в первой главе преимуществ реактивных систем, было решено реализовать интерфейс унифицированного доступа, а именно, внедрить программное обеспечение, разработанное с соблюдением принципов реактивных систем для последующей декомпозиции монолитных сервисов на микросервисы и упрощенного полного перехода к микросервисным технологиям.

Основной функционал сервиса, разрабатываемого в ходе выполнения данной выпускной квалификационной работы, заключается в маршрутизации запросов, балансировке нагрузки на сервисы, а также в создании единой точки авторизации для пользователей сервисов экосистемы. Согласно микросервисному подходу, разрабатываемый сервис должен быть сконцентрирован на одной бизнес-задаче, чтобы предотвратить последующее расширение функционала и разрастание сервиса до запутанного монолита.

Функциональные требования разрабатываемого сервиса:

— маршрутизация запросов от клиентской стороны к серверной для клиент-серверных приложений;

— аутентификация и авторизация пользователей клиент-серверных приложений, выдача временных токенов для доступа к ресурсам;

— маршрутизация запросов между внутренними сервисами и внешними интеграциями;

Нефункциональные требования:

— возможность масштабирования;

— гибкость при нагрузках, возможность увеличить количество экземпляров сервиса в production-среде;

— отказоустойчивость.

Таким образом, были сформированы основные требования к разрабатываемому интерфейсу унифицированного доступа.

2.2 Проектирование архитектуры сервиса

Разработанное ранее решение Remote Access Point (RAP) имело множество недостатков, среди которых уже описанная ранее проблема с крайне плохой производительностью и низкой стабильностью.

Низкая стабильность порождает проблемы с отказоустойчивостью и отзывчивостью. Помимо прочего, одна из проблем проявилась уже на стадии проектирования – для реализации сервиса был выбран паттерн API-шлюз, но реализован он был не полностью.

В данном случае шлюз покрывал только некоторые приложения, взаимодействующие с клиентской стороной, тогда как другие работали с клиентом напрямую. Это привело к тому, что взаимодействие с API разных сервисов стало крайне запутанным, что усложняет поддержку существующей кодовой базы.

Кроме того, кодовая база самого сервиса сама по себе является достаточно запутанной, поскольку не имеет документации и использует устаревшие технологии, что приводило к проблемам при попытке внесения любых изменений в принцип его работы.

Сервис, разрабатываемый в рамках данной выпускной квалификационной работы, не должен иметь вышеперечисленных недостатков, то есть должен быть отказоустойчив, полностью реализовывать выбранный паттерн проектирования, основной функционал должен быть документирован, а также должен использовать технологии, отвечающие требованиям текущего времени.

Поскольку в экосистеме присутствует множество разнородных сервисов, спроектированных независимо друг от друга, все они работают в отдельных процессах. На данный момент взаимодействие между ними осуществляется

множеством разных способов – часть запросов проходит через RAP, другие напрямую отправляются от одного сервиса к другому, зачастую используя различные протоколы передачи данных.

Для выбора паттерна, необходимо определить, за какие действия должен отвечать проектируемый сервис:

— маршрутизация запросов от клиентов к соответствующим микросервисам;

— агрегация ответов от нескольких микросервисов в один ответ для клиента;

— обработка аутентификации и авторизации клиентов;

— реализация смежных задач, таких как логирование, кэширование, ограничение скорости и т.д.;

— работа с разными протоколами передачи данных, как минимум, с HTTP и gRPC.

Решением данной проблемы является использование такого паттерна, как API Gateway (API шлюз). API шлюз – это промежуточный слой между клиентами и микросервисами, который позволяет управлять доступом к микросервисам и обеспечивает единую точку входа для всех запросов к микросервисам, и позволяет полностью покрыть описанные выше действия.

На рисунке 10 представлена схема взаимодействия клиента с интерфейсами приложения с помощью API Gateway.

Сервис, разрабатываемый в ходе выполнения данной работы, будет выполнять следующие основные функции: перенаправление запросов от клиента на различные сервисы, агрегация нескольких запросов к разным сервисам в один ответ, обработка авторизации и аутентификации пользователей, а также играть роль распределителя нагрузки между запущенными экземплярами сервисов.

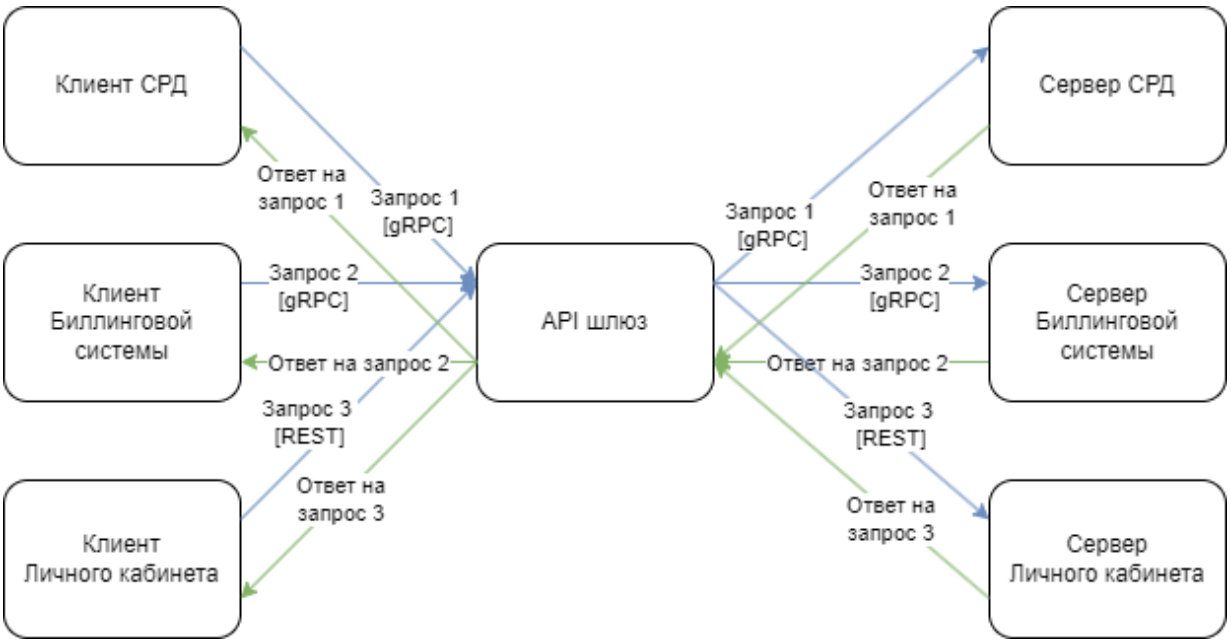

Рисунок 10 – Схема взаимодействия клиента с интерфейсами приложения с помощью API Gateway

Таким образом, с помощью данного промежуточного слоя между сервисами внутренние системы организации абстрагируются от пользователей приложений и становится возможным централизованно управлять всеми подсистемами.

2.3 Описание инструментов разработки сервиса

В данный момент, на рынке присутствует достаточно много инструментов, помогающих в реализации паттерна API Gateway, поэтому одной из задач стала необходимость подобрать такой набор инструментов, который будет соответствовать всем требованиям.

Для работы с gRPC был выбрана библиотека Akka gRPC – это библиотека, которая предоставляет поддержку для создания потоковых gRPC-серверов и клиентов поверх Akka Streams и Akka HTTP. Она содержит генератор, который начинается с определения службы protobuf и создает модельные классы, API службы и код сервера и клиента.

Одним из условий при выборе прокси была его возможность работать с gRPC без дополнительных расширений или плагинов. Всего было отобрано три кандидата:

— Nginx – высокопроизводительный веб-сервер и обратный прокси, который может использоваться в качестве API-шлюза для серверных и безсерверных приложений [19];

— Envoy – современный прокси, который разработан для облегчения сетевого взаимодействия между микросервисами. Он также может выступать в роли API-шлюза для управления трафиком на уровне L7 [18];

— Kong – масштабируемый и расширяемый API-шлюз и платформа для ускорения и управления трафиком к сервисам. Он работает на основе Nginx и Lua и позволяет добавлять различные функции через плагины [13];

Далее будет проводиться оценка их производительности и удобства работы. На рисунке 11 и рисунке 12 продемонстрированы диаграммы с результатами сравнения количества передаваемых запросов в секунду по HTTP и HTTPS.

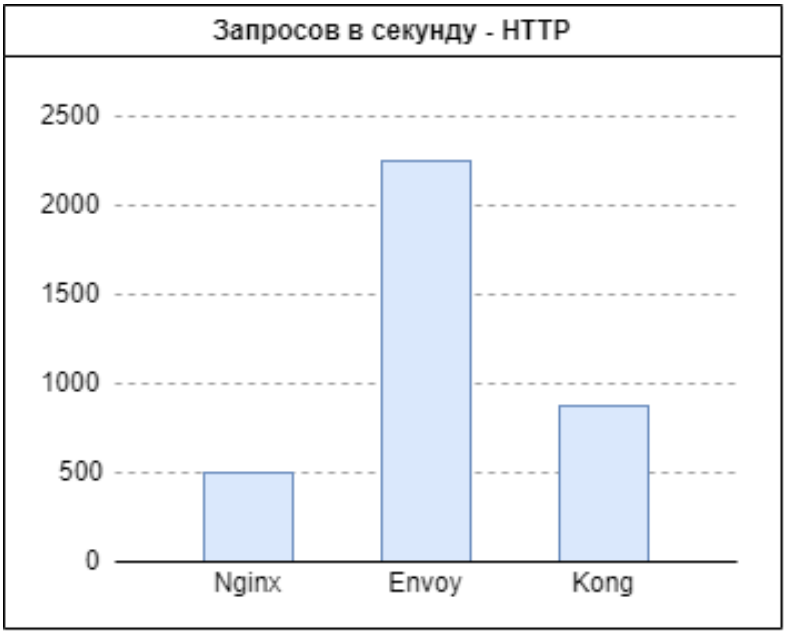

Рисунок 11 – Диаграмма, отражающая кол-во запросов в секунду по протоколу HTTP

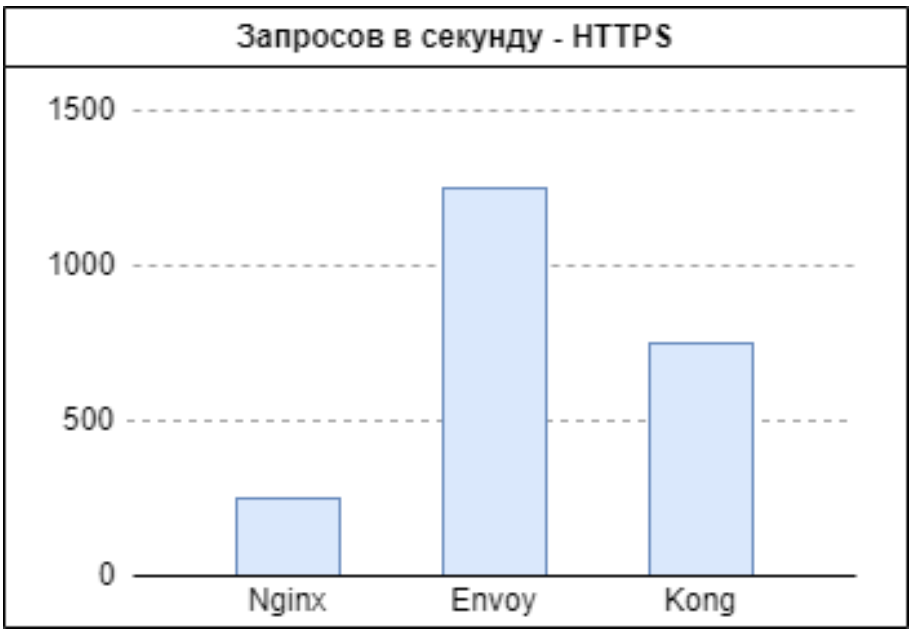

Рисунок 12 – Диаграмма, отражающая кол-во запросов в секунду по протоколу HTTPS

По полученным данным можно сделать вывод, что Envoy значительно обгоняет конкурентов в производительности, поэтому, для реализации интерфейса унифицированного доступа был выбран именно он. Поскольку в требованиях к сервису была заявлена необходимость выдачи пользователям временных токенов, или же JWT – JSON Web Token. Как следует из названия, JWT – это открытый стандарт (RFC 7519) для создания токенов доступа, основанных на формате JSON. Токены создаются сервером, подписываются секретным ключом и передаются клиенту, который в дальнейшем использует полученный токен для подтверждения своей личности [11].

На рисунке 13 представлена uml-диаграмма последовательностей, демонстрирующая процесс авторизации пользователя на примере биллингового сервиса.

Для реализации авторизации пользователей с использованием JWT-токенов существует несколько готовых инструментов – OAuth 2.0 и OpenID Connect. OpenID, по сути, представляет собой надстройку над OAuth, поэтому

нужно определить, достаточно ли будет функционала, предоставляемого OAuth, или же необходимо использовать OpenID.

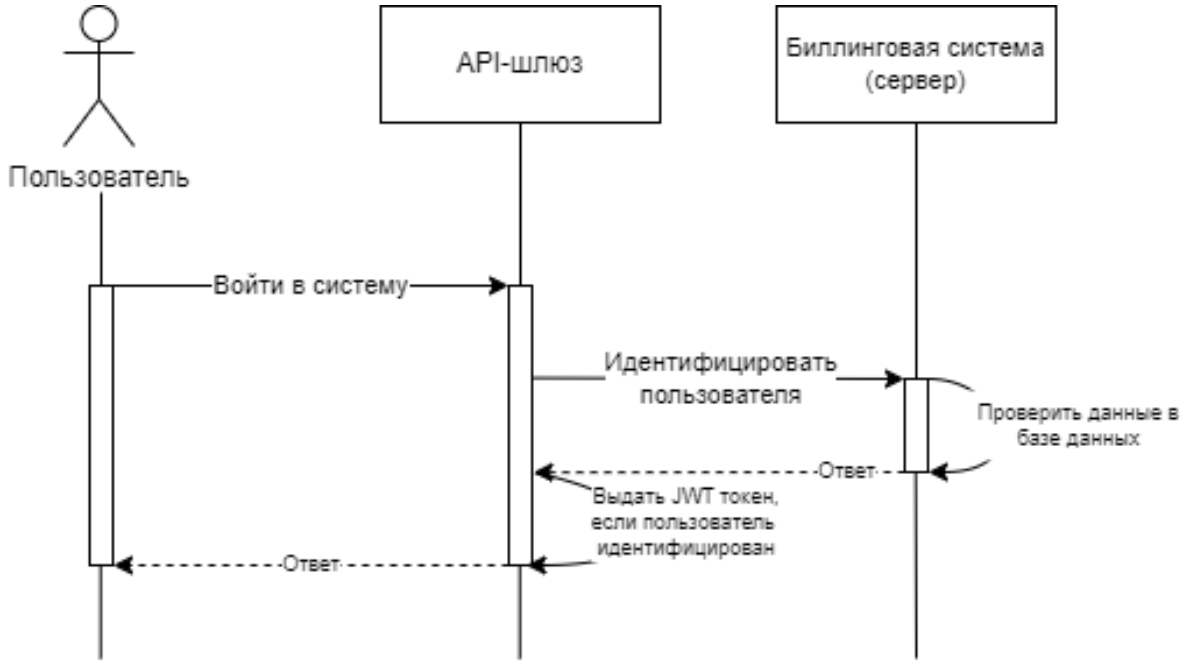

Рисунок 13 – Диаграмма последовательностей, демонстрирующая процесс выдачи JWT-токена при попытке пользователя авторизоваться

OAuth позволяет только авторизовать пользователя, при этом, про пользователя сервер не знает ничего, кроме его access токена. OpenID Connect же позволяет хранить информацию о логине и профиле пользователя, что позволяет реализовать его аутентификацию. Также, OpenID дает возможность динамической регистрации и обнаружения сервисов, если те используют единую базу для хранения информации о пользователях для их авторизации в разных, не связанных между собой сервисах.

На данный момент, в экосистеме сервисов ООО «Квартплата 24» нет отдельного сервиса, отвечающего за авторизацию пользователя в остальных сервисах, поэтому необходимости в использовании OpenID нет. Кроме того, при проектировании сервисов стоит избегать использования различных

инструментов «на будущее», что является еще одним аргументом за использование OAuth 2.0, а не OpenID.

Необходимо хранить информацию о выданных токенах, из-за чего возникает необходимость работы с реляционной базой данных и удобного представления ее сущностей в коде. Есть несколько вариантов фреймворков, обеспечивающих такую возможность.

Выбор стоит между фреймворками Hibernate и Slick. Главное отличие между ними заключается в подходах. Hibernate реализует такой подход, как Object Relational Mapping (сокр. ORM) – объектно-реляционное отображение, техника, позволяющая отображать структуры реляционной базы данных (например, SQL Server, Oracle, MySQL и т.д.) на структуру объектов в объектно-ориентированном языке программирования [12]. Slick же, в свою очередь, реализует подход Functional Relation Mapping (сокр. FRM) – функциональное реляционное отображение, используется в функциональных языках программирования, таких как Scala или Haskell [17].

Оба этих фреймворка позволяют работать с данными как с объектами или функциями, а не как с кортежами или таблицами, и абстрагироваться от конкретного синтаксиса и особенностей базы данных.

Для выбора одного из них был проведен сравнительный анализ:

— Slick предоставляет типобезопасные и композиционные API для запросов, которые проверяются на этапе компиляции. Hibernate же использует JPQL или HQL, которые являются строковыми языками запросов, и могут вызывать ошибки во время выполнения. Таким образом, использование Slick позволить повысить стабильность работы сервиса;

— Slick поддерживает асинхронный и неблокирующий стиль программирования с использованием Future и Reactive Streams. Hibernate же работает в синхронном и блокирующем режиме с использованием сессий и транзакций. В данном случае это играет большую роль, ведь необходимо придерживаться принципов реактивной архитектуры;

— Slick позволяет легко переключаться между чистым SQL и FRM в зависимости от потребностей. Hibernate же требует использования специальных API для выполнения SQL-запросов или вызова хранимых процедур;

На основе вышеперечисленных преимуществ фреймворка Slick, для обеспечения работы сервиса с базой данных.

Поскольку другие сервисы экосистемы используют PostgreSQL для хранения данных, то и для хранения данных в данном сервисе будет использоваться он.

PostgreSQL – базирующаяся на языке SQL объектно-реляционная СУБД. Основными ее особенностями являются: надежность, расширяемость, производительность [15].

Таким образом, были выбраны программные инструменты, необходимые для разработки интерфейса унифицированного доступа.

2.4 Разработка диаграммы классов сервиса

Разрабатываемый интерфейс унифицированного доступа будет состоять из нескольких модулей, а именно из API и реализации. Команды будут поступать в виде REST-запросов и gRPC-вызовов.

В таблице 1 представлено краткое описание существующих классов.

Таблица 1 – Описание существующих классов

Название класса	Роль класса
Launcher	Запуск сервиса
HttpRequestListener	Слушает http-запросы
RpcRequestListener	Слушает rpc-вызовы
ConfigParser	Вычитывает файл конфигурации
HttpRequestHandler	Обрабатывает т http-запросы
RpcRequestHandler	Обрабатывает rpc-вызовы
OAuthProvider	Обрабатывает запросы на авторизацию, ведет работу с токенами

Продолжение таблицы 1

Название класса	Роль класса
ServiceRpcAbstractMapper	Реализует основные методы для перевода http ответов в rpc вызовы
DebtServiceRpcMapper	Реализует перевод http ответов в rpc вызовы для СРД
SsbsServiceRpcMapper	Реализует перевод http ответов в rpc вызовы для биллинговой системы

На рисунке 14 представлена диаграмма классов разрабатываемого сервиса.

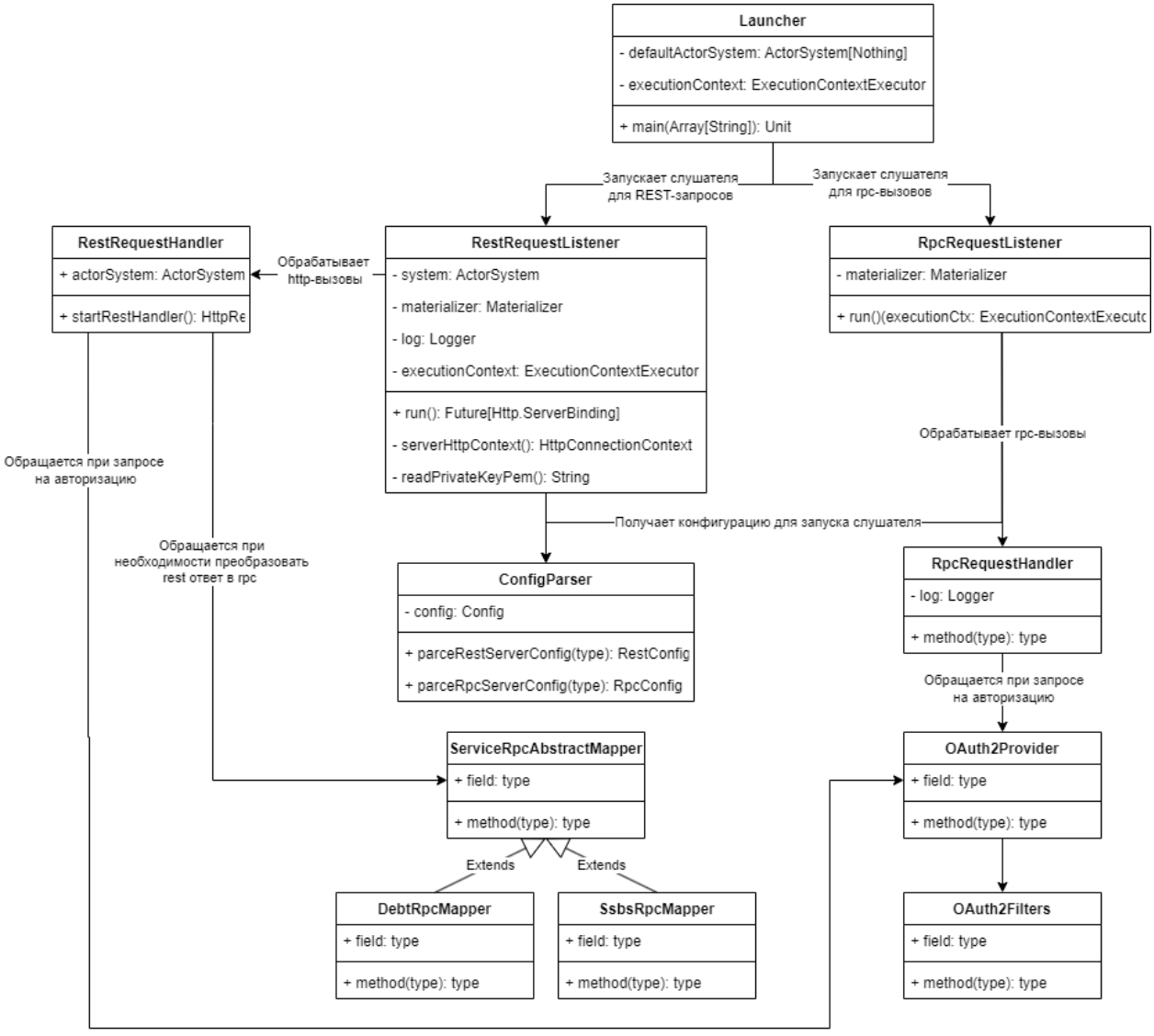

Рисунок 14 – Диаграмма классов интерфейса унифицированного доступа

Также необходимо определить структуру базы данных, в которой будет храниться информация о выданных токенах доступа к сервисам.

На рисунке 15 представлена диаграмма ER-диаграмма сущностей.

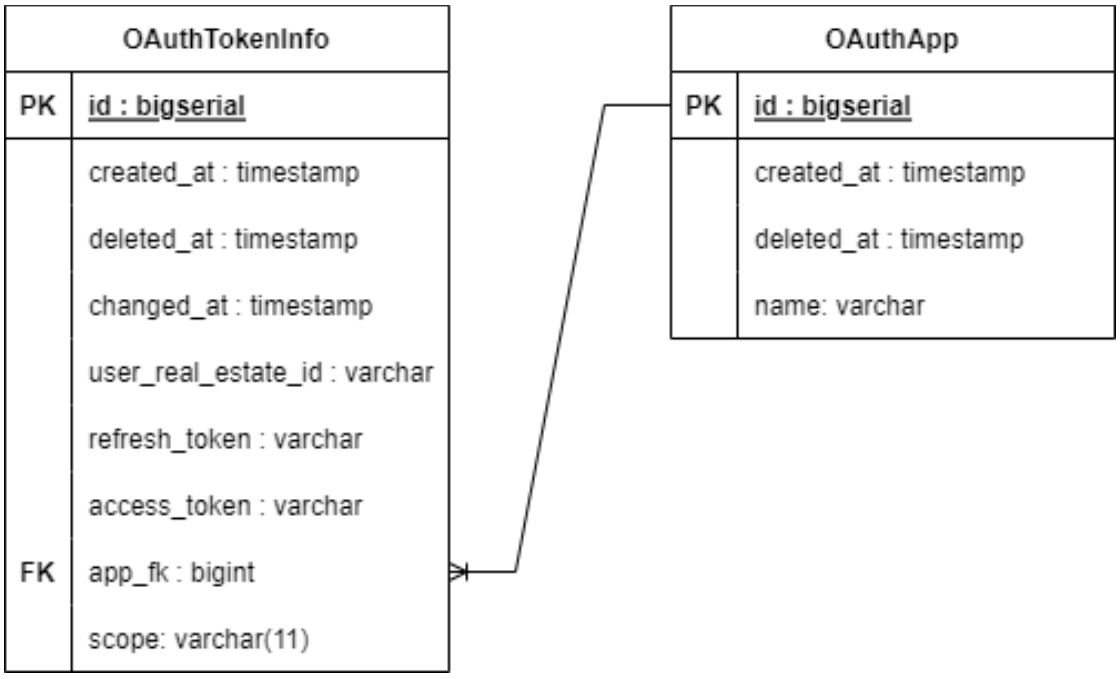

Рисунок 15 – ER-диаграмма базы данных для хранения токенов

Таблица «OAuthTokenInfo» хранит информацию о выданных токенах для пользователей. Поля created_at, deleted_at и changed_at, как следует из их названия, хранят время создания, удаления, и изменения записи таблицы. Поля refresh_token и access_token соответственно хранят в себе сами токены. Поле user_real_estate_id хранит в себе уникальный идентификатор клиента, соединенный через точку с запятой с id пользователя в базе целевого сервиса

Таблица «OAuthApp» хранит в себе информацию о сервисах, для доступа к которым используется OAuth. Поле name хранит в себе название сервиса, created_at и deleted_at, соответственно, информацию о дате добавления сервиса и дате его удаления.

Таблица «OAuthTokenInfo» связана с таблицей «OAuthApp» через поле app_fk «связью многие к одному».

Таким образом, была построена диаграмма классов сервиса, описано назначение этих классов. Также были описаны основные сущности базы данных, и сформирована их ER-диаграмма.

2.5 Архитектура разрабатываемого сервиса в контексте интеграции в существующую экосистему сервисов

На данный момент в экосистеме сервисов в ООО «Квартплата 24» часть приложений представляет собой монолит, часть реализует микросервисную архитектуру. Находясь в кластере, они общаются между собой напрямую.

При разработке данного сервиса использовались современные технологии, повышающие отказоустойчивость системы. Интегрированный в экосистему сервис представлен на рисунке 16.

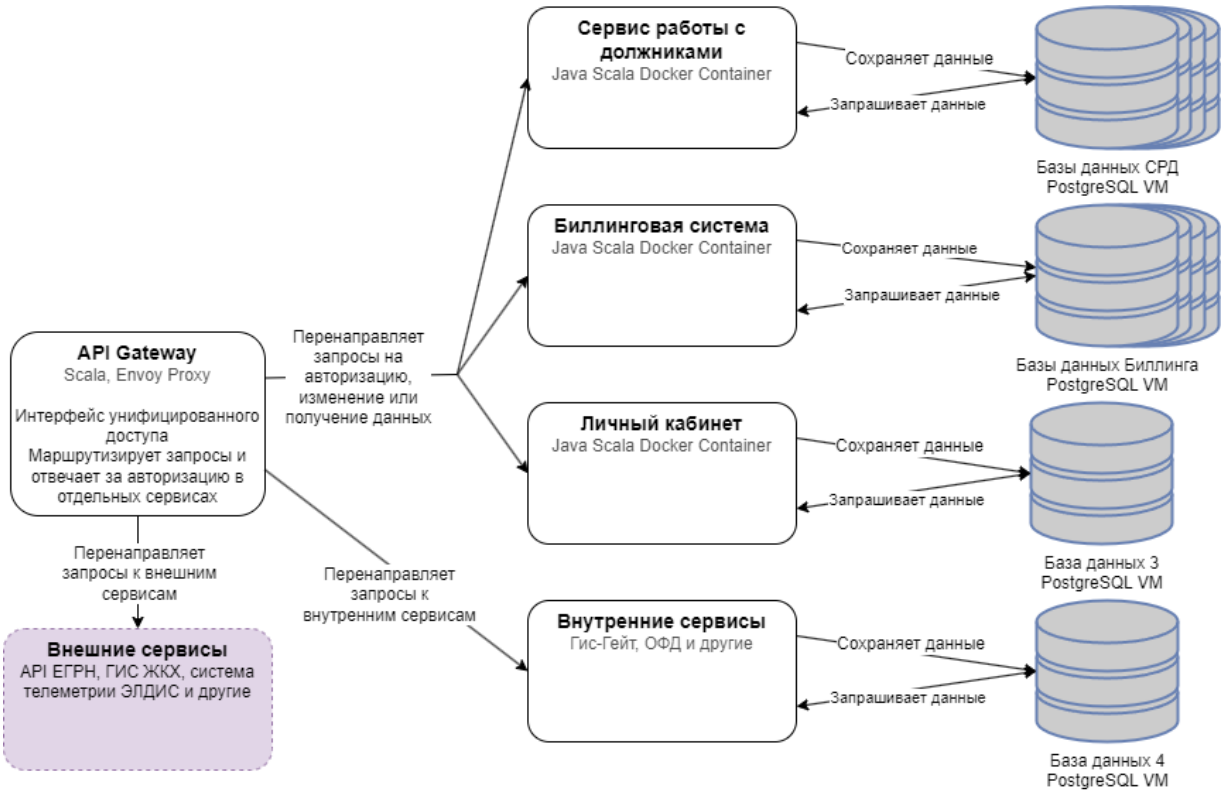

Рисунок 16 – Диаграмма контейнеров, демонстрирующая интегрированный в экосистему сервисов интерфейс унифицированного доступа

Интеграция сервиса, реализованного в рамках данной выпускной квалификационной работы, создаст твердую основу для дальнейшего внедрения микросервисов и переработки монолитных приложений, и, в итоге, полного перехода к микросервисной архитектуре.

В данной главе были сформулированы требования к разрабатываемому интерфейсу унифицированного доступа к распределенной реактивной системе, выполняющего функции маршрутизации запросов с клиента на сервер, агрегацию нескольких ответов от сервисов, а также единой точки авторизации для клиентов.

Рассмотрены недостатки и преимущества нескольких инструментов, созданных для реализации описанных функций.

Построена диаграмма классов сервиса, описаны основные команды и их назначение.

Представлена модернизированная архитектура экосистемы сервисов ООО «Квартплата 24» с интегрированным в нее интерфейсом унифицированного доступа к распределенной реактивной информационной системе.

Глава 3 Разработка интерфейса унифицированного доступа

3.1 Выбор средств реализации

При разработке программного обеспечения, соответствующего принципам реактивной архитектуры, важно подобрать такой набор инструментов, которые в полной мере обеспечивают успешное выполнение поставленной цели, т.е. соответствуют принципам, лежащим в основе построения реактивных приложений.

Java – объектно-ориентированный язык программирования, созданный компанией Sun Microsystems в 1995 году и являющийся вторым по популярности языком программирования в мире, в таких рейтингах как: IEEE Spectrum (2020), TIOBE (2021) [5], [6], [10].

Scala – мультипарадигменный язык программирования, разработанный в 2004 году командой специалистов из Федеральной политехнической школы Лозанны под руководством Мартина Одерски. Совмещает в себе функциональную и объектно-ориентированную парадигмы [16].

Как и Java, Scala является языком, компилирующимся в байт-код и выполняемым на Java Virtual Machine (JVM). По умолчанию, Scala имеет в себе все основные библиотеки Java и имеет с ним полную совместимость, что дает возможность, например, описать поля класса в Java, а потом написать его имплементацию на Scala [3].

В таблице 2 представлено сравнение двух языков программирования по основным критериям.

Таблица 2 – Сравнение языков Java и Scala

Критерии	Java	Scala
Лаконичность	нет	да
Простота кода	да	нет
Производительность	нет	да

По сравнению с Java, язык Scala более лаконичен, и его использование существенно сокращает объем кода.

Поскольку был выбран язык программирования Scala, необходимо подобрать подходящую для работы с ними среду интегрированной разработки. На данный момент существуют три наиболее популярных среды разработки, подходящие для языка Scala: NetBeans, Eclipse и IntelliJ IDEA (Community Edition).

Краткое описание данных сред разработки:

— NetBeans – это интегрированная среда разработки на Java и других языках программирования, которая предоставляет удобный редактор кода, поддержку различных фреймворков и платформ, а также инструменты для профилирования и отладки приложений.

— Eclipse – это популярная среда разработки на Java и других языках программирования, которая поддерживает множество плагинов и инструментов для совместной работы, тестирования и развертывания приложений.

— IntelliJ IDEA – это мощная среда разработки на Java и других языках программирования, которая обладает интеллектуальными возможностями анализа кода, рефакторинга и авто дополнения, а также поддерживает множество технологий и фреймворков для создания современных приложений [7], [8].

Поскольку по имеющемуся функционалу все три описанные среды разработки примерно схожи, было решено провести их сравнение по следующим критериям:

— простота и удобство интерфейса;

— соответствие современным требованиям;

— удобство подключения и работы с плагинами и библиотеками;

— личный опыт использования.

Результаты сравнения по описанным критериям представлен в таблице 3, оценки выставлены от 0 до 2.

Таблица 3 – Сравнение интегрированных сред разработки

Критерии оценивания/название IDE	NetBeans	Eclipse	IntelliJ IDEA (Community Edition)
Удобство интерфейса	1	1	2
Соответствие современным требованиям	1	0	2
Работа с плагинами и библиотеками	2	2	2
Личный опыт	1	1	2
Итого:	5	4	8

Таким образом, после проведения сравнительного анализа выбор пал на на IntelliJ IDEA (Community Edition). Данная среда по всем параметрам обходит своих конкурентов, так как имеет достаточно удобный, дружелюбный к пользователю интерфейс, удобный набор инструментов для работы с плагинами и библиотеками, включая Scala Plugin, необходимый для написания программ на данном языке, базами данных большинства видов (PostgreSQL, Cassandra и пр.). Также, не последним аргументом послужил тот факт, что самый большой опыт работы именно с этой средой разработки.

Таким образом, были выбраны необходимые для разработки интерфейса унифицированного доступа инструменты, библиотеки и фреймворки.

3.2 Реализация основных модулей сервиса и интегрирование его с другими сервисами экосистемы

Функциональное назначение данного сервиса заключается маршрутизация запросов между клиентами и сервисами, а также между самими сервисами, авторизации пользователей с выдачей временных токенов для доступа к ресурсам других сервисов, а также в балансировке нагрузки.

Для создания шаблона проекта будущего сервиса использовался Scala Build Tool (SBT) – автоматический сборщик проектов. Проект, использующий фреймворк Akka gRPC создается с помощью команды sbt new akka/akka-grpc-quickstart-scala.g8. В процессе создания проекта необходимо указать название

проекта, версии основных фреймворков, таких как Akka, Akka gRPC и Akka HTTP, а также версию используемого языка Scala [9].

Также необходимо создать классы, описанные в главе 2-й пункте 2.4. Итоговая структура проекта представлена на рисунке 17.

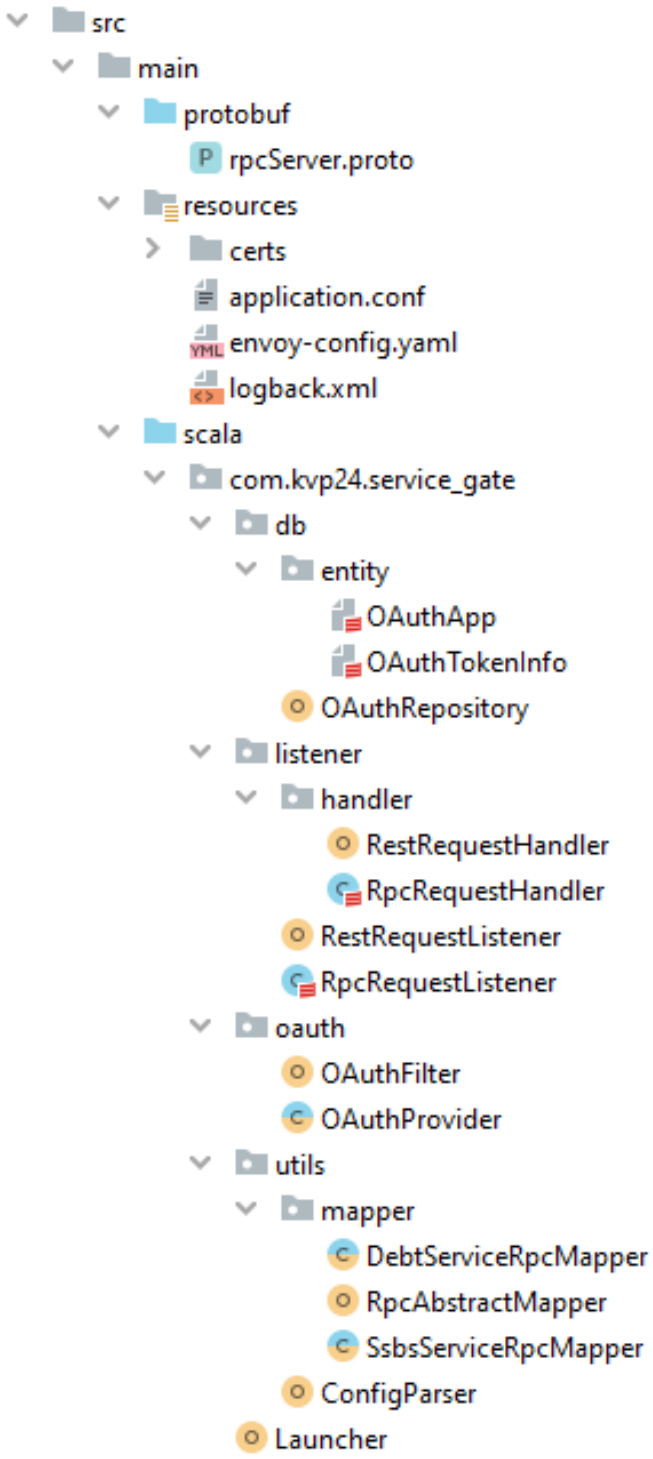

Рисунок 17 – Скриншот, демонстрирующий структуру проекта

После того, как был создан проект и его основные сущности, необходимо настроить Envoy-прокси, который будет отвечать за балансировку нагрузки, поиск сервисов. Фрагмент его конфигурации представлен на рисунке 18.

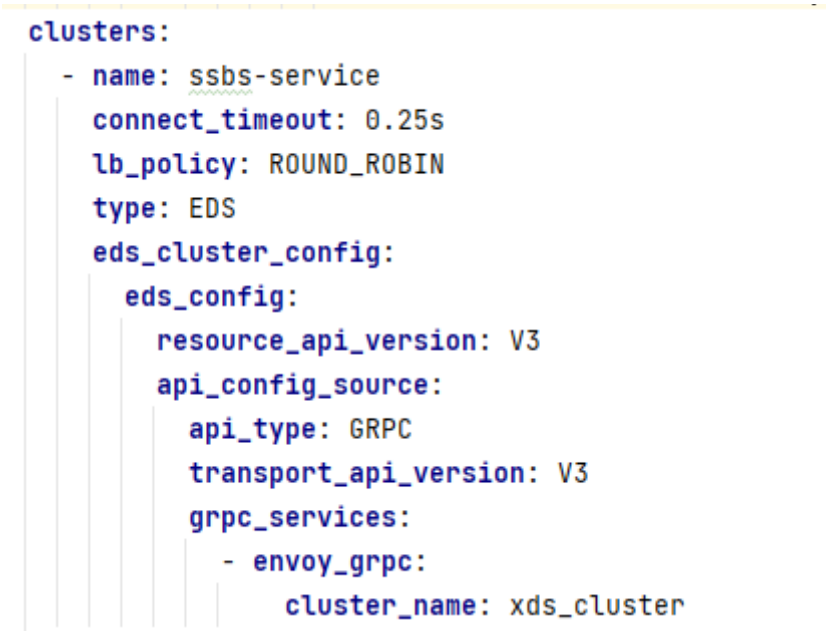

Рисунок 18 – Скриншот с листингом конфигурации envoy-proxy для обнаружения ssbs-сервиса

За балансировку отвечает параметр lb_policy, он может принимать несколько значений:

— Round Robin – равномерно распределяет запросы между доступными узлами в круговом порядке;

— Least Request – выбирает узел с наименьшим числом активных запросов в конкретно данный момент времени;

— Random – как следует из названия, узел выбирается случайным образом.

В данном случае, для биллингового сервиса выбрана политика Round Robin, поскольку он обычно запущен в двух экземплярах.

Кроме того, gRPC-вызовы автоматически добавляются в конфигурацию прокси из файлов с расширением «.proto». В случае с обычными REST-

запросами, то их приходится прописывать вручную. На рисунке 19 представлен листинг конфигурации энд-поинтов:

```
filter_chains:
  - filters:
      - name: envoy.filters.network.http_connection_manager
        typed_config:
          "@type": type.googleapis.com/envoy.extensions.filters.network.http_connection_mana
          stat_prefix: ingress_http
          codec_type: AUTO
          route_config:
            name: local_route
            virtual_hosts:
              - name: local_service
                domains: ["*"]
                routes:
                  - match: { prefix: "/" }
                    route: { cluster: ssbs-service }
                  - match: { prefix: "/" }
                    route: { cluster: debt }
                  - match: { prefix: "/" }
                    route: { cluster: pssbs }
          http_filters:
            - name: envoy.filters.http.router
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.http.router.v3.Router
```

Рисунок 19 – Скриншот, демонстрирующий листинг конфигурации энд-поинтов в Envoy

В данном случае Envoy слушает все запросы из Биллинга, СРД и личного кабинета.

Полный листинг конфигурации Envoy-прокси для интерфейса унифицированного доступа представлен в приложении А.

Далее необходимо описать основные классы: Laucher, HttpRequestListener и RpcRequestListener. В последних двух силами Akka HTTP и Akka gRPC создаются слушатели, которые будут активны на протяжении всего времени работы интерфейса унифицированного доступа. При получении каких-либо запросов они будут либо перенаправлять их в

соответствующие сервисы, либо, как в случае с запросами на авторизацию, осуществлять дополнительные действия.

Листинг кода описанных классов представлен в приложениях Б, В и Г соответственно.

Следующим шагом необходимо с помощью библиотеки Slick описать сущности базы данных в проекте. Сначала необходимо создать case class, и описать в нем поля таблицы, после чего надо создать еще один класс, который будет наследоваться от класса Table, реализованного в Slick. Внутри класса необходимо создать поля, идентичные тем, что были описаны в case class-е ранее, и задать им значения, используя метод из библиотеки Slick – column с указанием типа. В параметры функции передается название столбца из таблицы БД в виде строки. На рисунке 20 представлен фрагмент кода, демонстрирующий описание сущности с использованием Slick.

```
final case class OAuthApp(
    id: String,
    created_at: LocalDateTime = LocalDateTime.now(),
    deleted_at: Option[LocalDateTime] = None,
    name: String
)

class OAuthAppTable(tag: Tag) extends Table[OAuthApp](tag, _tableName = "oauth_app") {
  val id = column[String]("id", O.PrimaryKey)
  val created_at = column[LocalDateTime]("created_at")
  val deleted_at = column[Option[LocalDateTime]]("deleted_at")
  val name = column[String]("name")

  def *: ProvenShape[OAuthApp] = (
    id,
    created_at,
    deleted_at,
    name
  ) <> ((OAuthApp.apply _).tupled, OAuthApp.unapply _)
}
```

Рисунок 20 – Скриншот с листингом кода описания сущности OAuthApp с использованием Slick

По такому же принципу описывается сущность и для таблицы «OAuthTokenInfo».

Теперь, когда основные сущности описаны, можно приступить к реализации выдачи JWT-токенов. В момент, когда будет осуществляться вызов команды на авторизацию, запрос будет перенаправляться на серверную часть определенного приложения (например, в биллинговую систему или СРД), где будет выполнена процедура идентификации пользователя и послан ответ, который так же будет обработан интерфейсом унифицированного доступа, и, в зависимости от результатов идентификации, будут либо созданы токены временного доступа, либо возвращено сообщение с отказом в авторизации.

Токен создается следующим образом:

— осуществляется поиск существующего токена. Если токен уже есть, то ему проставляется дата закрытия, делая его недействительным;

— далее заполняются поля сущности OAuthTokenInfo, проставляется дата создания, уникальный идентификатор пользователя, с помощью стандартных утилит языка Java генерируются токены доступа и обновления, проставляется идентификатор клиента;

— в конце полученная сущность записывается в базу.

При любых действиях пользователя в приложении, когда требуется взаимодействие с серверной частью, в запросе будет передаваться его user_id. По этому идентификатору будет осуществляться поиск актуального токена доступа для конкретного сервиса, в котором он работает. Если токен не будет найден, то на клиент будет отправлен соответствующий ответ, который приведет к выходу из системы.

Выданный токен доступа будет действителен до момента выхода пользователя из приложения. Когда приложение будет закрыто, будет отправлен соответствующий запрос, при получении которого у существующего токена доступа будет проставлена дата закрытия deleted_at. Таким образом, при следующей попытке авторизации токен этот токен не будет учитываться, и пользователь получит новый токен.

Поскольку некоторые сервисы запрашивают данные из внешних систем, ответы на эти запросы зачастую приходят в REST-формате. Поскольку таких запросов относительно немного, и известна их структура, на каждый такой запрос были написаны конвертеры, преобразующий ответ в виде json в rpc-вызов, который далее будет направлен в соответствующий сервис.

Таким образом, были описаны основные моменты в реализации интерфейса унифицированного доступа.

3.3 Тестирование работоспособности интерфейса унифицированного доступа

Для проверки работоспособности разработанного интерфейса унифицированного доступа проведем тестирование его основных функций.

Было решено провести тестирование двумя способами: ручным, т.е. запросы отправляются вручную через специальный инструмент – Postman, и, второй способ – через запущенный клиент приложения сервиса работы с должниками.

Для демонстрации тестирования будет использован запрос на авторизацию. Так можно будет проверить сразу несколько пунктов: то, что запросы корректно перенаправляются в сервис с клиента, работоспособность механизма выдачи токенов доступа, а также процесс выдачи данных.

Для тестирования надо создать в Postman новый запрос, указать сервер, и выбрать из списка необходимый запрос, в данном случае, getUserAuth. В теле сообщения необходимо указать логин пользователя, тип сервиса, и хешированный пароль. На рисунке 21 представлен скриншот Postman с результатом выполнения запроса.

В ответе был возвращен токен доступа, используя его, можно осуществить другие запросы. Таким образом, можно сказать, что авторизация работает корректно.

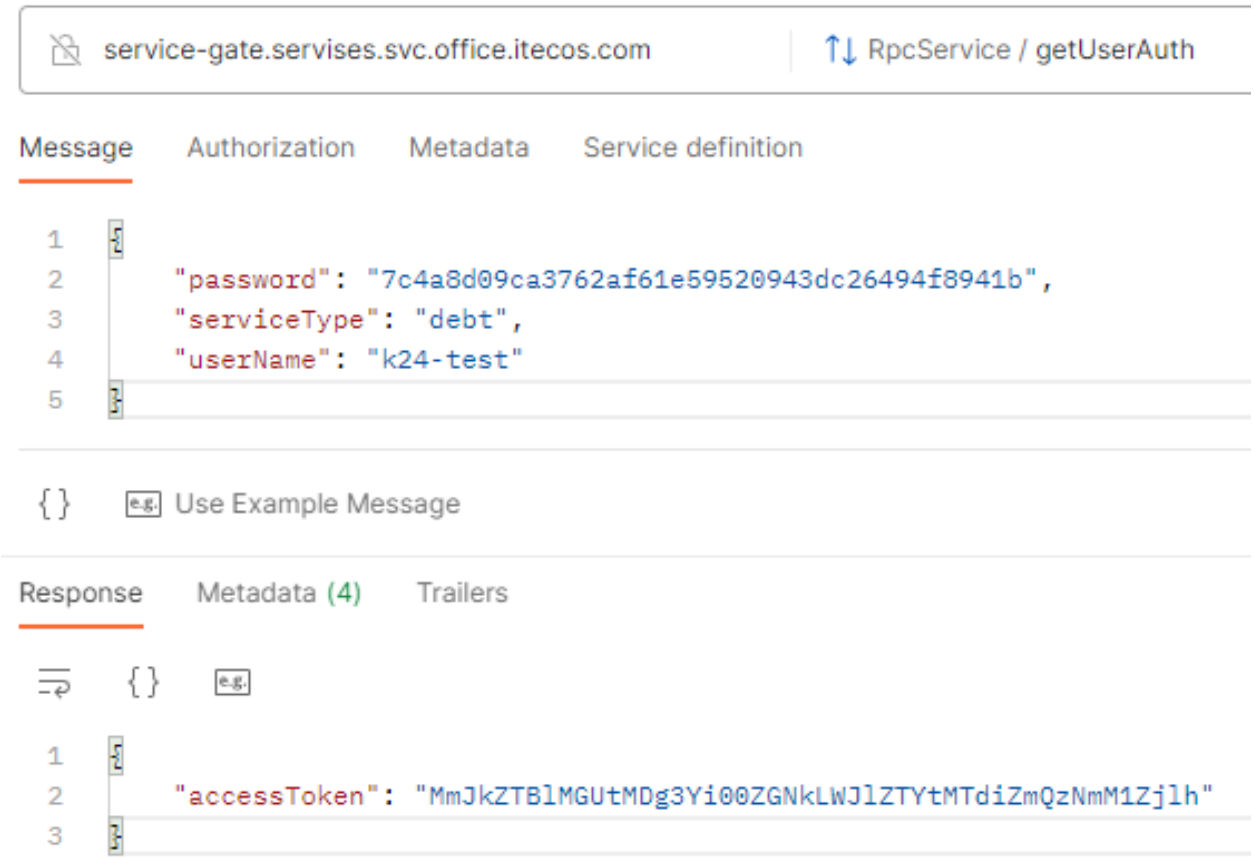

Рисунок 21 – Скриншот, демонстрирующий результат выполнения запроса на авторизацию

Теперь необходимо протестировать возможность запроса информацию о конкретном лицевом счете из сервиса. Для этого также создается новый запрос, в теле которого указывается токен доступа и номер л/с. На рисунке 22 представлен скриншот, демонстрирующий результаты запроса.

В результате была возвращена базовая информация о лицевом счете: плательщик, площадь жилья, старый и новый номер лицевого счета и управляющая компания, что означает, что запрос был успешно обработан.

Таким образом можно сказать, что сервис корректно выполняет свои основные функции, а именно, успешно авторизует пользователя в системе и корректно переадресует запросы со стороны клиента на соответствующие сервисы.

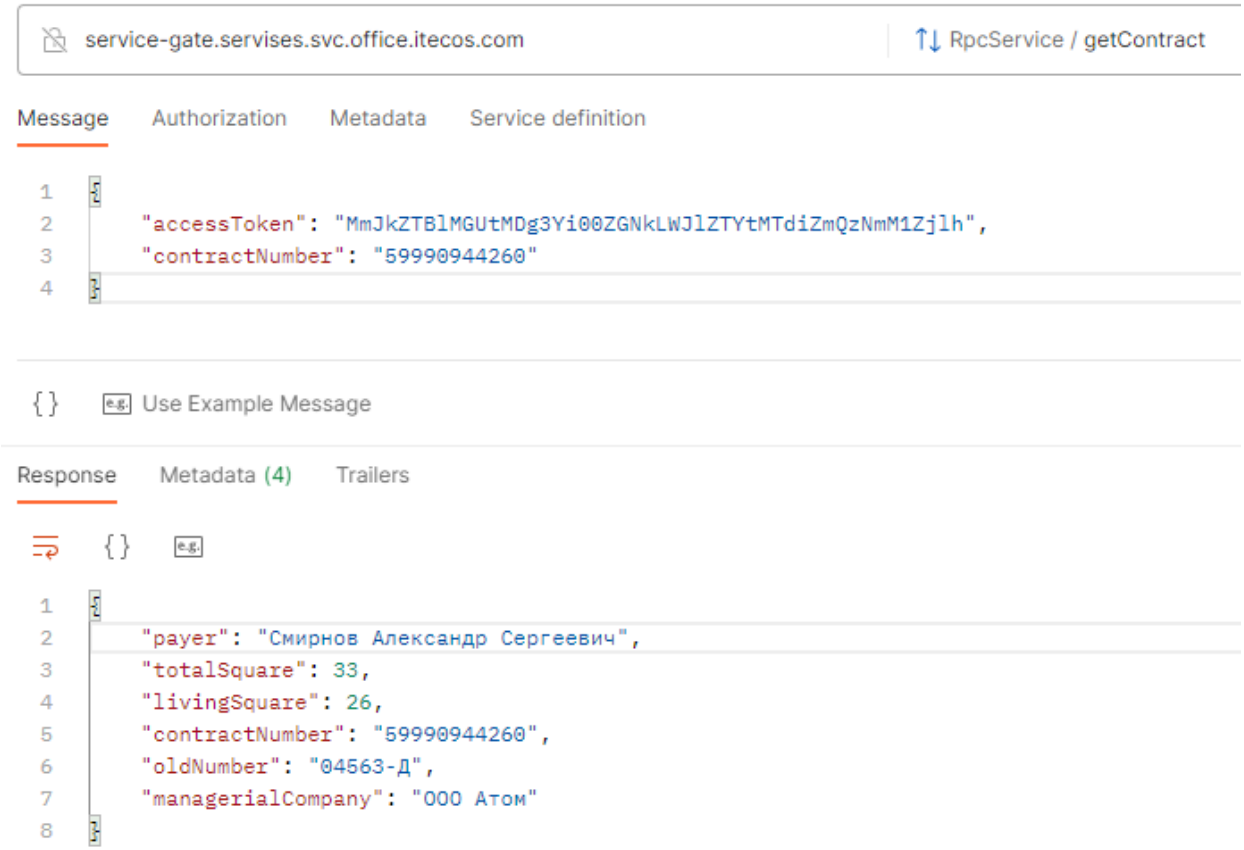

Рисунок 22 – Скриншот, демонстрирующий результат запроса на получение информации о лицевом счете

В данной главе были описаны инструменты, необходимые для реализации интерфейса унифицированного доступа к распределенной реактивной информационной системе. Также были рассмотрены основные моменты его реализации, и проведено тестирование с целью убедиться в корректной работе разработанного сервиса.

Заключение

Данная выпускная квалификационная работа посвящена построению интерфейса унифицированного доступа к высоконагруженным, отказоустойчивым информационным системам, с использованием микросервисных технологий, основанных на реактивной архитектуре.

В процессе выполнения выпускной квалификационной работы был выполнен ряд задач.

Был произведен анализ и дана характеристика существующей распределенной реактивной информационной системе, представляющую из себя группу тесно интегрированных сервисов, представляющих собой смесь монолитных и микросервисных приложений.

Подробно разобраны основные требования к разрабатываемому сервису, спроектирована его архитектура на основе паттерна API Gateway. Были подобраны инструменты разработки – прокси Envoy и OAuth 2.0 для авторизации пользователей с использованием JWT-токенов.

Основываясь на описанных требованиях, был разработан гибкий и отказоустойчивый сервис, выполняющий функцию единой точкой входа для API и авторизации и аутентификации пользователей для отдельных сервисов экосистемы ООО «Квартплата 24». Языком программирования для реализации сервиса был выбран Scala, а в качестве среды разработки была выбрана IntelliJIdea.

Выполнено тестирование основного функционала разработанного сервиса.

Результаты выполнения данной выпускной квалификационной работы имеют практический интерес и могут быть рекомендованы разработчикам высоконагруженных, производительных, отказоустойчивых информационных систем.

Список используемой литературы

1. Вигерс К. Разработка требований к программному обеспечению // К. Вигерс, Д. Битти. – Спб:BHV,2014.-736с
2. Галимянов А.Ф., Галимянов Ф.А. Архитектура информационных систем. – Казань: Казан. ун-т, 2019. – 117 с
3. Ездаков А.Л. Функциональное и логическое программирование. – М.: Бином. Лаборатория знаний, 2009. – 120 с.
4. Трутнев Д. Р. Архитектуры информационных систем. Основы проектирования: Учебное пособие. – СПб.: НИУ ИТМО, 2012. – 66 с.
5. Baesens, B. Beginning Java Programming: The Object-Oriented Approach / B. Baesens, A. Backiel, S. Vanden Broucke. – 1st edition, Wrox, 2015.
6. Deitel, H. Java How to Program / H. Deitel, P. Deitel. – 9th edition, Prentice Hall, 2015.
7. Hudson O. Getting started with IntelliJ IDEA // O. Hudson, Birmingham: Packt Publishing, 2013. – 114p.
8. Krochmalski J. IntelliJ IDEA Essentials // J. Krochmalski. – Birmingham: Packt Publishing, 2014.-263p.
9. Akka gRPC Quickstart with Scala [Электронный ресурс]. URL: https://developer.lightbend.com/guides/akka-grpc-quickstart-scala/. (дата обращения: 28.04.2023).
10. JavaFX Overview [Электронный ресурс]. URL: https://openjfx.io/javadoc/18/ (дата обращения: 21.04.2023)
11. JWT – как безопасный способ аутентификации и передачи данных [Электронный ресурс]. URL: https://vc.ru/dev/106534-jwt-kak-bezopasnyy-sposob-autentifikacii-i-peredachi-dannyh (дата обращения: 22.04.2023)
12. Hibernate ORM [Электронный ресурс]. URL: https://hibernate.org/orm/ (дата обращения: 22.04.2023)
13. Kong API Gateway FAQ [Электронный ресурс]. URL: https://konghq.com/faqs#:~:text=Kong%20Gateway%20is%20the%20world%27s,

for%20microservices%20and%20distributed%20architectures. (дата обращения: 22.04.2023).

14. Microsoft Руководство по архитектуре приложений .NET gRPC [Электронный ресурс]. URL: https://learn.microsoft.com/ru-ru/dotnet/architecture/cloud-native/grpc (дата обращения: 31.03.2023).

15. PostgreSQL Documentation [Электронный ресурс]. URL: https://www.postgresql.org/docs/current/index.html (дата обращения: 21.04.2023)

16. Scala Book [Электронный ресурс]. URL: https://docs.scalalang.org/overviews/scala-book/introduction.html (дата обращения: 22.04.2023).

17. Slick [Электронный ресурс]. URL: https://scala-slick.org (дата обращения: 22.04.2023)

18. What is Envoy [Электронный ресурс]. URL: https://www.envoyproxy.io/docs/envoy/v1.26.2/intro/what_is_envoy (дата обращения: 22.04.2023).

19. What is Nginx [Электронный ресурс]. URL: https://www.nginx.com/resources/glossary/nginx/#:~:text=NGINX%20is%20open%20source%20software,for%20maximum%20performance%20and%20stability. (дата обращения: 22.04.2023).

20. You’re Not Actually Building Microservices [Электронный ресурс]. URL: https://www.simplethread.com/youre-not-actually-building-microservices/ (дата обращения: 31.03.2023).

Приложение А

Фрагмент конфигурации Envoy-proxy

```
static_resources:
  listeners:
    - address:
        socket_address:
          address: 0.0.0.0
          port_value: 80
      filter_chains:
        - filters:
            - name: envoy.http_connection_manager
              typed_config:
                "@type":
type.googleapis.com/envoy.config.filter.network.http_connection_manager.v2.Htt
pConnectionManager
                codec_type: http1
                stat_prefix: ingress_http
                route_config:
                  name: local_route
                  virtual_hosts:
                    - name: backend
                      domains:
                        - "*"
                      routes:
                        - match:
                            prefix: "/api/"
                          route:
                            cluster: api
                        - match:
                            prefix: "/"
                          route:
                            cluster: frontend
                http_filters:
                  - name: envoy.router
                    typed_config: { }
  clusters:
    - name: ssbs_client
      connect_timeout: 1.00s
      type: strict_dns
      lb_policy: round_robin
      load_assignment:
```

```
      cluster_name: ssbs_client
      endpoints:
       - lb_endpoints:
          - endpoint:
             address:
              socket_address:
                address: debt_client
                port_value: 9090
                ipv4_compat: true
  - name: debt_client
    connect_timeout: 1.00s
    type: strict_dns
    lb_policy: round_robin
    eds_cluster_config:
     eds_config:
      resource_api_version: V3
      api_config_source:
       api_type: GRPC
       transport_api_version: V3
       grpc_services:
        - envoy_grpc:
            cluster_name: xds_cluster
  - name: xds_cluster
    connect_timeout: 0.25s
    type: STATIC
    lb_policy: ROUND_ROBIN
    typed_extension_protocol_options:
     envoy.extensions.upstreams.http.v3.HttpProtocolOptions:
      "@type":
type.googleapis.com/envoy.extensions.upstreams.http.v3.HttpProtocolOptions
      explicit_http_config:
       http2_protocol_options:
        connection_keepalive:
         interval: 30s
         timeout: 5s
       upstream_connection_options:
        # configure a TCP keep-alive to detect and reconnect to the admin
        # server in the event of a TCP socket half open connection
        tcp_keepalive: { }
       load_assignment:
        cluster_name: xds_cluster
        endpoints:
         - lb_endpoints:
```

```
              - endpoint:
                  address:
                    socket_address:
                      address:
                      port_value:
  - name: service_gate
    connect_timeout: 0.25s
    type: strict_dns
    lb_policy: round_robin
    load_assignment:
      cluster_name: terminal
      endpoints:
        - lb_endpoints:
            - endpoint:
                address:
                  socket_address:
                    address: service_gate
                    port_value: 9090
                    ipv4_compat: true
admin:
  access_log_path: "/dev/null"
  address:
    socket_address:
      address: 0.0.0.0
      port_value: 8001
```

Приложение Б

Листинг кода класса Launcher

```
package com.kvp24.service_gate
import akka.actor.typed.ActorSystem
import akka.actor.typed.scaladsl.Behaviors
import com.kvp24.service_gate.listener.RestRequestListener
import com.kvp24.service_gate.listener.RpcRequestListener
import org.slf4j.LoggerFactory
import scala.concurrent.ExecutionContextExecutor
import scala.util.{Failure, Success, Try}
object Launcher {
  private val log = LoggerFactory.getLogger("ServiceGateLauncher")
  private val defaultActorSystem: ActorSystem[Nothing] =
    ActorSystem(Behaviors.empty, "service-gate-default-ac")
  private implicit val executionContext: ExecutionContextExecutor =
    defaultActorSystem.executionContext
  def main(args: Array[String]): Unit = {
    Try {
      RestRequestListener.run(defaultActorSystem)
    } match {
      case Failure(exception) =>
        log.error(
          s"Can't start RestRequestListener: ${exception.getMessage}",
          exception
        )
      case Success(_) =>
        log.info("RestRequestListener started successfully")
    }
    Try {
      RpcRequestListener.run(defaultActorSystem)
    } match {
      case Failure(exception) =>
        log.error(
          s"Can't start RpcRequestListener: ${exception.getMessage}",
          exception
        )

      case Success(_) =>
        log.info("RpcRequestListener started successfully")
    }
  }
```

Приложение В

Фрагмент кода класса RestRequestListener

```
package com.kvp24.service_gate.listener

import akka.actor.typed.ActorSystem
import akka.http.scaladsl.Http
import akka.stream.Materializer
import akka.stream.scaladsl.Sink
import com.kvp24.service_gate.listener.handler.RestRequestHandler
import com.kvp24.service_gate.utils.ConfigParser
import org.slf4j.LoggerFactory

import scala.concurrent.ExecutionContextExecutor

object RestRequestListener {

  private val log = LoggerFactory.getLogger(RestRequestListener.getClass)

  def run(
      actorSystem: ActorSystem[Nothing]
  )(implicit exc: ExecutionContextExecutor): Unit = {
    implicit lazy val materializer: Materializer = Materializer(actorSystem)
    val settingsOption = ConfigParser.extractRestSettings
    settingsOption match {
      case Some(settings) =>
        Http()(actorSystem)
          .newServerAt(settings.host, settings.port)
          .connectionSource

.to(Sink.foreach(_.handleWithSyncHandler(RestRequestHandler.startRestHandler)
))
          .run()

      case None =>
        log.error("Can't start RestRequestListener - config not found")
    }
  }

…
}
```

Приложение Г

Фрагмент кода класса RpcRequestListener

```
package com.kvp24.service_gate.listener

import akka.actor.typed.ActorSystem
import akka.http.scaladsl.ConnectionContext
import akka.http.scaladsl.HttpsConnectionContext
import akka.http.scaladsl.Http
import akka.http.scaladsl.model.HttpRequest
import akka.http.scaladsl.model.HttpResponse
import akka.pki.pem.DERPrivateKeyLoader
import akka.pki.pem.PEMDecoder
import com.kvp24.service_gate.listener.handler.RpcRequestHandler
import com.kvp24.service_gate.utils.ConfigParser
import com.kvp24.service_gate.RpcServiceHandler
import org.slf4j.LoggerFactory

import java.security.KeyStore
import java.security.SecureRandom
import java.security.cert.Certificate
import java.security.cert.CertificateFactory
import javax.net.ssl.KeyManagerFactory
import javax.net.ssl.SSLContext
import scala.concurrent.ExecutionContext
import scala.concurrent.Future
import scala.io.Source
import scala.util.{Failure, Success}

class RpcRequestListener(actorSystem: ActorSystem[_]) {

  private val log = LoggerFactory.getLogger(this.getClass)

  implicit lazy val system: ActorSystem[_] = actorSystem
  implicit lazy val ec: ExecutionContext = actorSystem.executionContext

  def run(): Future[Http.ServerBinding] = {
    ConfigParser.extractRpcSettings match {
      case Some(config) =>
        val service: HttpRequest => Future[HttpResponse] =
          RpcServiceHandler(new RpcRequestHandler(actorSystem))

        val bound: Future[Http.ServerBinding] = Http(system)
```

```
      .newServerAt(interface = config.host, port = config.port)
      .enableHttps(serverHttpContext)
      .bind(service)

    bound.onComplete {
      case Success(binding) =>
        val address = binding.localAddress
        log.info(
          s"gRPC server bound to ${address.getHostString}:${address.getPort}"
        )

      case Failure(exception) =>
        log.error(
          "Failed to bind gRPC endpoint, terminating system",
          exception
        )
        system.terminate()
    }
    bound

  case None =>
    val message = "Can't start RpcRequestListener - config not found"
    log.error(message)
    Future.failed(new Exception(message))
 }
}

private def serverHttpContext: HttpsConnectionContext = {
 val privateKey =
  DERPrivateKeyLoader.load(PEMDecoder.decode(readPrivateKeyPem()))
 val fact = CertificateFactory.getInstance("X.509")
 val cer = fact.generateCertificate(
  classOf[RpcRequestListener].getResourceAsStream("/certs/server1.pem")
 )
 val ks = KeyStore.getInstance("PKCS12")
 ks.load(null)
 ks.setKeyEntry(
  "private",
  privateKey,
  new Array[Char](0),
  Array[Certificate](cer)
 )
 val keyManagerFactory = KeyManagerFactory.getInstance("SunX509")
```

```
    keyManagerFactory.init(ks, null)
    val context = SSLContext.getInstance("TLS")
    context.init(keyManagerFactory.getKeyManagers, null, new SecureRandom)
    ConnectionContext.https(context)
  }

  private def readPrivateKeyPem(): String =
    Source.fromResource("certs/server1.key").mkString
}
```