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)

02.03.03 Математическое обеспечение и администрирование информационных систем
(код и наименование направления подготовки / специальности)

WEB-дизайн и мультимедиа
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему Модернизация архитектуры существующей распределенной информационной системы в ООО «Квартплата 24»

Обучающийся А.В. Свитов
(И.О. Фамилия) (личная подпись)

Руководитель к.т.н., В.С. Климов
(ученая степень, звание, И.О. Фамилия)

Консультант к.ф.н., М.М. Бажутина
(ученая степень, звание, И.О. Фамилия)

Тольятти 2022

Аннотация

Тема выпускной квалификационной работы - «Модернизация существующей распределенной информационной системы в ООО «Квартплата 24»».

Актуальность работы заключается в необходимости перехода от устаревших архитектур программного обеспечения к более современным.

Объектом исследования является экосистема сервисов в ООО «Квартплата 24».

Предметом исследования является микросервисная архитектура, позволяющая разрабатывать высоконагруженные, отзывчивые и отказоустойчивые информационные системы.

Цель выпускной квалификационной работы – разработка программного обеспечения, построенного по принципам реактивных и микросервисных архитектур.

Методы исследования – микросервисные и реактивные архитектуры, построение реактивных микросервисов.

Разработан модуль распределенной информационной системы, архитектура которого основана на принципах реактивной и микросервисной архитектур. Решена проблема обмена данными между сервисами экосистемы.

Результаты бакалаврской работы имеют практический интерес и могут быть рекомендованы разработчикам высоконагруженных распределенных информационных систем.

Выпускная квалификационная работа состоит из 54 страниц текста, 28 рисунков, 8 таблиц и 25 источников.

Abstract

The title of the bachelor’s thesis is «Modernization of the existing distributed information system in OOO Kvartplata 24».

The relevance of the work is due to the need to move from outdated software architectures to more modern architectures.

The object of the research is the ecosystem of services at OOO Kvartplata 24.

The subject of the research is a microservice architecture that allows developing highly loaded, responsive and fault-tolerant information systems.

The aim of the bachelor’s thesis is the development of software built on the principles of reactive and microservice architectures.

Research methods are microservice and reactive architectures, building reactive microservices.

A distributed information system module has been developed, the architecture of which is based on the principles of reactive and microservice architectures. The problem of data exchange between ecosystem services has been solved.

The results of the bachelor's thesis are of practical interest and can be recommended to developers of highly loaded distributed information systems.

The work consists of 54 pages of text, 28 figures, 8 tables and a list of 25 references.

Содержание

Введение

Широкое распространение информационных технологий способствует их скорейшему развитию, еще 15 лет назад мы могли видеть приложения, запущенные на нескольких серверах, которые обрабатывали гигабайты данных, находящихся в состоянии покоя. Нередким был случай, когда при попытке получения доступа к какому-либо ресурсу вы могли наблюдать сообщение о недоступности сервиса и ведущихся технических работах.

В наше время люди слишком сильно полагаются на различные технологии в своей повседневной жизни, поэтому современные системы должны удовлетворять требованиям современного пользователя, быть отзывчивыми и отказоустойчивыми. Как недоступность сервисов в течение пары часов повлияет на вашу жизнь? Это может не оказать существенного влияния на вас, если вы не пользуетесь веб-сервисами на постоянной основе, но что будет, если поисковая система, такая как Google, будет отвечать на ваш запрос не за секунду, а за тридцать секунд?

Пользователи неотзывчивого программного обеспечения со временем устанут от подобных проблем и начнут смотреть в сторону конкурентов. Для решения подобных проблем существуют микросервисы на основе реактивной архитектуры.

Актуальность данной работы заключается в необходимости отказа от классических принципов построения информационных систем в пользу современных архитектурных решений, позволяющих разрабатывать высоконагруженные, отказоустойчивые, масштабируемые и отзывчивые системы.

Целью данной выпускной квалификационной работы является модернизация архитектуры, существующей распределенной информационной системы путем внедрения микросервисных технологий на основе реактивной архитектуры, что послужит основой для последующего полного перехода к микросервисной архитектуре.

Для достижения поставленной цели необходимо выполнить промежуточные задачи:

- изучить текущее архитектурное решение экосистемы сервисов в ООО «Квартплата 24»;
- описать альтернативное архитектурное решение в виде микросервисной и реактивной архитектур;
- разработать микросервис, соблюдая принципы построения реактивных микросервисов;

Объект исследования – экосистема сервисов в ООО «Квартплата 24».

Предметом исследования является построение распределенной информационной системой, разделенной на модули-микросервисы, каждый из модулей сконцентрирован не на решении комплекса задач, а одной конкретной бизнес-задаче.

Методы исследования – микросервисные и реактивные архитектуры, построение реактивных микросервисов.

Первая глава содержит описание экосистемы сервисов в ООО «Квартплата 24», характеристику текущего архитектурного решения и преимуществ микросервисной, реактивной архитектур.

Вторая глава посвящена построению микросервисного приложения на основе команд и событий, а также решению проблемы доставки данных между микросервисами.

В третьей главе описаны инструменты разработки и процесс разработки микросервиса, тестирование работоспособности.

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

1 Характеристика организации и экосистемы сервисов

1.1 Характеристика организации

ООО «Квартплата 24» – Российская IT-компания, занимающаяся разработкой и сопровождением экосистемы из более чем 10 тесно интегрированных между собой облачных сервисов, решающих задачи расчёта и учета платы за ЖКХ, приема платежей и их распределение, а так же взыскание долгов в соответствии с законодательством РФ на всей её территории.

Компания предоставляет свои услуги на рынке услуг в сфере IT с 1996 года и обслуживает свыше 1 миллиона лицевых счетов более 800 ТСЖ, сервисных, ресурсоснабжающих организаций и управляющих компаний в 63 субъектах Российской Федерации.

Квартплата 24 обеспечивает более 30 способов электронной оплаты, принимает платежи по всей территории Российской федерации и за ее пределами, а так же является единственным в РФ платежным сервисом, который обеспечивает моментальный прием и расщепление платежей за жилищно-коммунальные услуги на всех этапах прохождения платежа.

Компания является участником проекта «Сколково» и резидентом технопарка в сфере высоких технологий «Жигулевская долина», а также включена Минстроем России в Банк решений Умного города и Банк эффективных технологий в ЖКХ.

1.2 Анализ экосистемы сервисов с точки зрения архитектуры

Состоящая из более 10 облачных сервисов, экосистема сервисов в ООО «Квартплата 24» представляет из себя комплексное решение для расчета платы за жилищно-коммунальные услуги, формирования платежных

документов, распределение платежей, работы с дебиторской задолженностью, контроля деятельности организаций.

Экосистема состоит из группы внутренних и внешних сервисов. Внешние сервисы предназначены для клиентского использования, внутренние же используются для поддержания технологических процессов.

Основными сервисами являются:

- сервис расчёта (Биллинг система);
- сервис платежей (процессинг);
- личный кабинет жителя;
- сервис по Работе с Должниками (СРД).

Сервисы, обеспечивающие интеграцию с внешними системами:

- взаимодействие с операторами фискальных данных;
- обмен данными с ОСЗН;
- обмен данными с ГИС ЖКХ;
- интеграция с телеметрическими системами;
- интеграция с внешними облачными сервисами и системами.

Визуальное представление основных сервисов экосистемы представлено на рисунке 1.

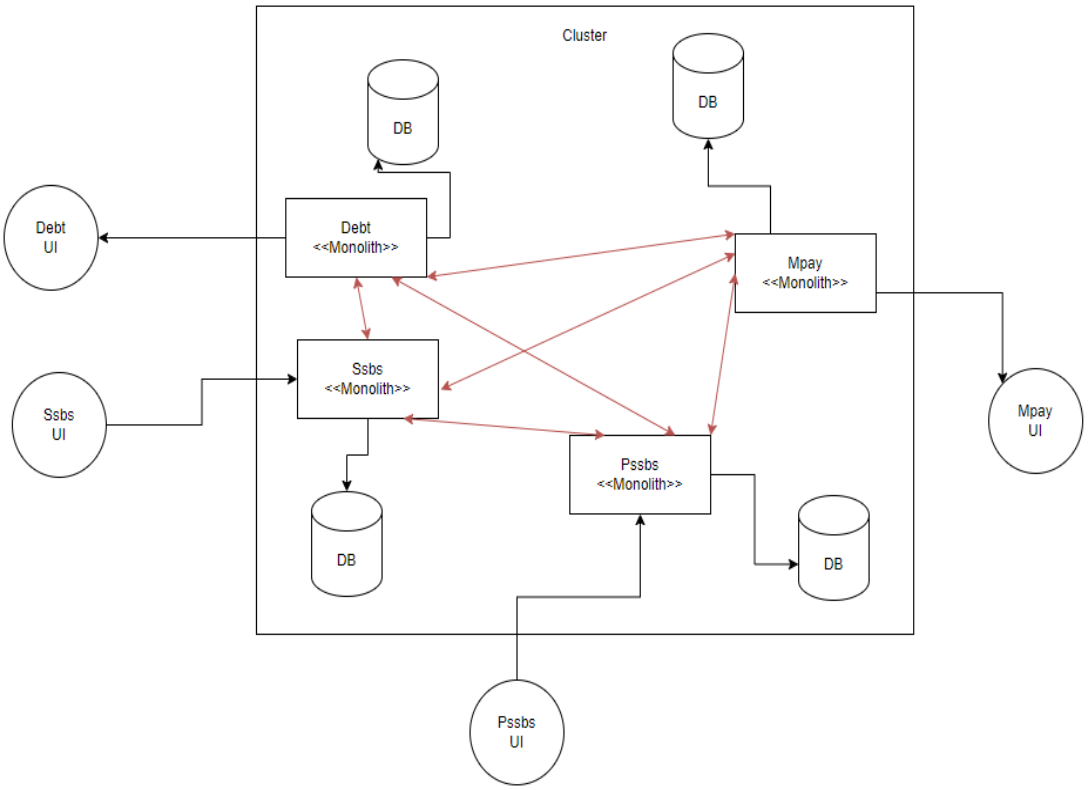

Рисунок 1 – Основные сервисы экосистемы в ООО «Квартплата 24»

Каждый из основных сервисов данной экосистемы является программным обеспечением, основанным на монолитной архитектуре. Данный подход к разработке и проектированию имеет некоторые преимущества перед другими архитектурами построения программного обеспечения, такие как:

– простое развертывание;
– единый мониторинг;
– упрощенное тестирование и устранение неполадок;
– переиспользование кода.

Но по мере того, как разрабатываемое приложение будет расти, будет появляться все больше трудностей в разработке и сопровождении данного программного обеспечения [2], [4]. Вне зависимости от изменяемого модуля, обновлять придется всё приложение, что может сопровождаться некоторыми трудностями, если монолит вырос до огромных масштабов. При нарастающей

нагрузке, чтобы обеспечить требуемую пропускную способность сервиса, единственным быстрым решением проблемы является вертикальное масштабирование, но вертикальное масштабирование ограничено мощностями сервера, на котором запущено приложение.

Добавление нового функционала в приложения подобного типа повлечет за собой изменения на всех уровнях приложения, клиентской части и серверной, в некоторых случаях изменением подвергается база данных. Внесение подобных изменений представлено на рисунке 2.

Рисунок 2 – Добавление нового функционала в приложение

Главной проблемой такой архитектуры является большое количество зависимостей, которыми обрастает монолит по мере разработки. Прекращение поддержки какого-либо фреймворка или библиотеки повлечет за собой колоссальные трудности в дальнейшей разработке, а именно проблемы с совместимостью версий различный технологий, используемых при разработке [5]. Подобная проблема повлечет за собой трудоемкую и дорогую переработку всего монолита. Запутанный монолит представлен на рисунке 3.

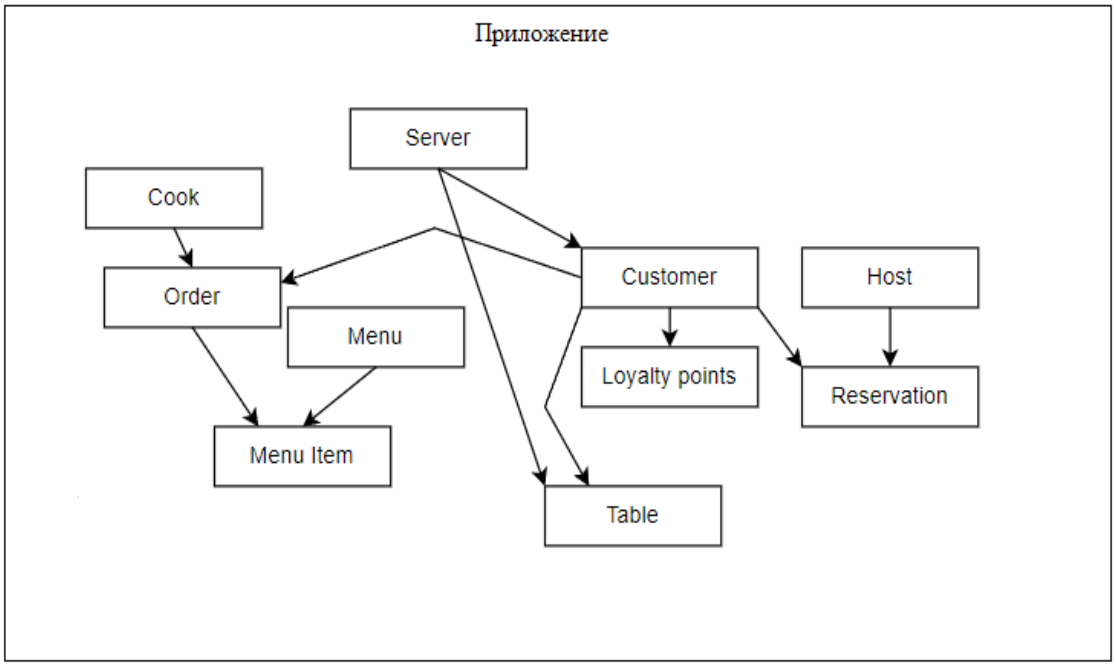

Рисунок 3 – Запутанный монолит

По мере роста запутанного монолита привлечение новых разработчиков с таким программным обеспечением становится всё менее возможным.

1.3 Микросервисная и реактивная архитектуры

В качестве альтернативы монолитной архитектуре и ее подвидам всё большую популярность получают микросервисная и реактивная архитектуры.

Системы, разработанные с соблюдением принципов реактивных систем, обеспечивают высокую отзывчивость, а также отличаются высокой гибкостью и способностью к масштабированию. В основе всех вышеперечисленных преимуществ перед другими системами лежит обмен данными при помощи неблокирующих сообщений.

Обмен данными, основанный на неблокирующем обмене сообщений, обеспечивает слабую связанность между модулями информационной системы. Такие сообщения позволяют организовать регулировку нагрузки,

отслеживание очередей сообщений при помощи различных инструментов для мониторинга и обеспечивает обратное давление в том случае, когда потребитель данных обрабатывает их медленнее, чем производитель данных их поставляет.

Обмен данных, основанный на средствах, предоставляющих неблокирующий обмен сообщениями, обеспечивает возможность для реализации гибких и устойчивых к нагрузкам систем [19]. Под гибкостью подразумевается способность системы реагировать на степень нагрузки, регулируя выделяемые вычислительные ресурсы. Устойчивость достигается путём изолирования подсистем в рамках одной информационной системы, различные модули такой системы могут прекращать свою работу и восстанавливаться самостоятельно, никак не влияя на другие модули. Это значит, что возникновение ошибки в одной части информационной системы никак не повлияет на работоспособность системы в целом, недоступным останется один из ее модулей.

Правильное соблюдение принципов построения реактивных систем позволит создавать отзывчивые системы с минимальным временем отклика при пользовательском запросе. Подобные системы упрощают обработку и исправление ошибок, повышают уверенность клиента в продукте и способствует скорейшему развитию продукта на рынке программного обеспечения на фоне конкурентов. Принципы, лежащие в основе реактивных систем изображены на рисунке 4.

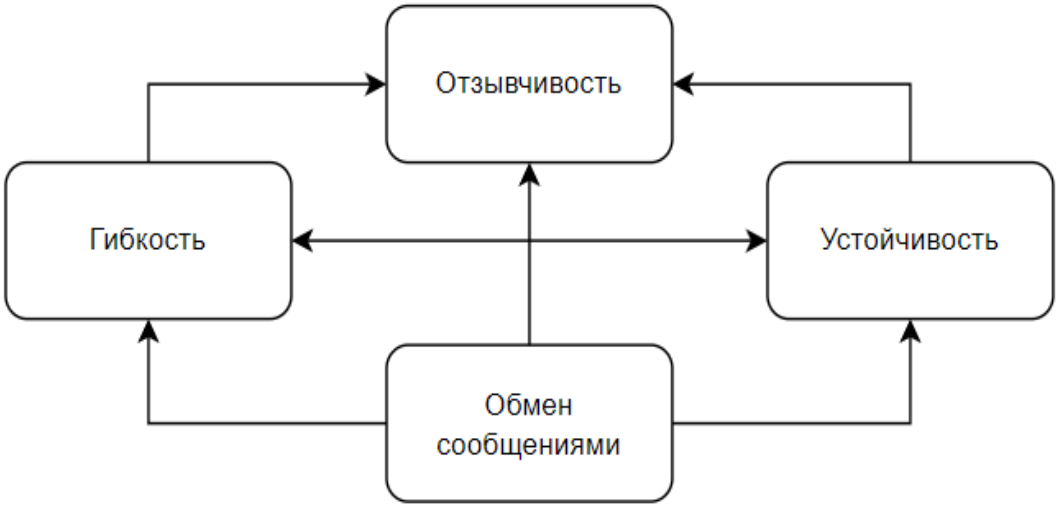

Рисунок 4 – Основа реактивных систем

Микросервисы – это приложение, представленное в виде отдельных сервисов, развертываемых независимо друг от друга. В связке группа микросервисов составляет распределенную информационную систему. Каждый из микросервисов владеет собственными данными, это значит, что сервисы системы не должны использовать какую-либо базу данных совместно. Подобная инкапсуляция данных внутри отдельных сервисов обеспечивает снижение связанности и позволяет использовать четкие интерфейсы сервисов для обмена данными. В зависимости от выполняемых задач, у микросервиса может и не быть базы данных вовсе.

Подобное архитектурное решение позволит распределять разработку между отдельными командами специалистов, что улучшит их понимание той части системы, которую они разрабатывают. Независимое развертывание открывает широкие возможности для масштабирования и снимает ограничение по части использование технологий [14]. Подобное микросервисное приложение изображено на рисунке 5.

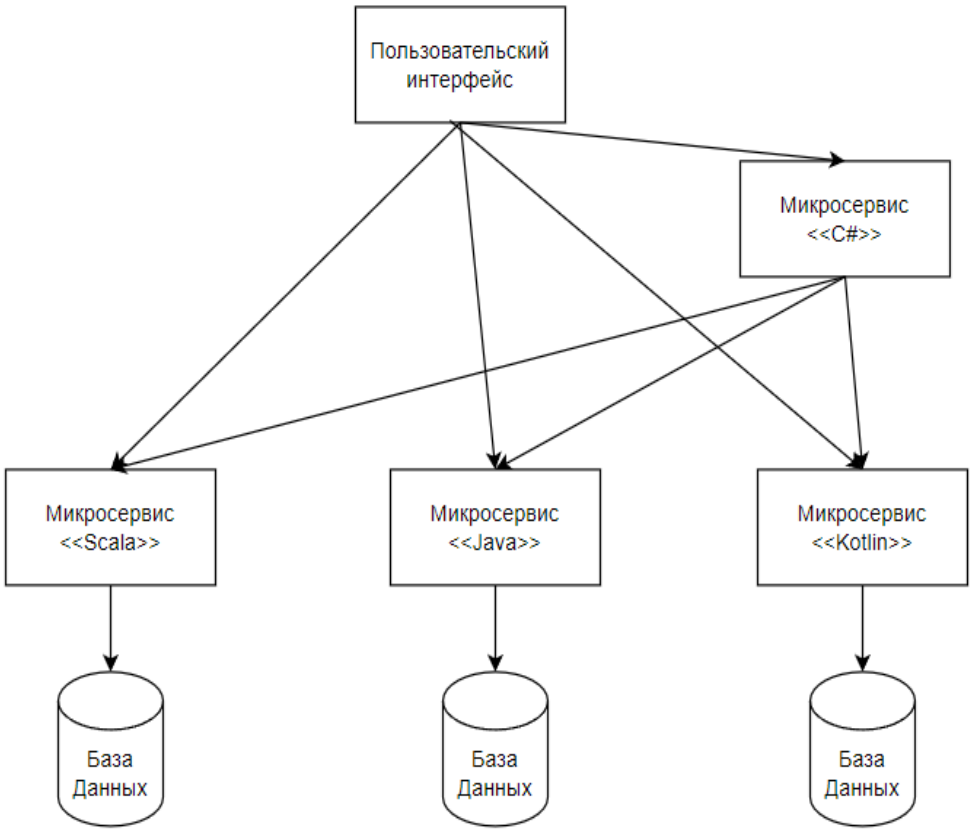

Рисунок 5 – Микросервисное приложение с использованием разных языков программирования

Микросервисы внутри одной распределенной информационной системы могут быть разработаны на разных языках программирования с использованием различных фреймворков и библиотек.

1.4 Постановка задачи на разработку микросервиса

В виду описанных выше особенностей микросервисных приложений и их преимуществами перед альтернативными вариантами архитектур было принято решение совершить переход к микросервисной архитектуре.

Основной целью данной выпускной квалификационной работы является модернизация архитектуры существующей распределенной информационной

системы, основанной на тесно интегрированных между собой монолитных приложений, путем интеграции в неё программного обеспечения, основанного на микросервисной архитектуре, что в последующем поможет проще декомпозировать монолиты и разделить их на отдельные микросервисы.

Основным функционалом разрабатываемого микросервиса является проверка на корректность и распространение данных о договорах, заключенных между ООО «Квартплата 24» с различными платежными системами и организациями-клиентами.

Для успешного выполнения данной выпускной квалификационной работы необходимо:

- спроектировать микросервисное приложение, являющееся сервисом для заключения договоров;
- описать инструмент реализации микросервиса, языка программирования и среды разработки;
- разработать диаграмму классов микросервиса;
- разработать микросервис, протестировать работоспособность;
- описать принципы работы разработанного микросервиса [1].

В данной главе были рассмотрены проблемы архитектурного решения существующей распределенной информационной системы, состоящей из тесно интегрированных между собой монолитных приложений, а также альтернативное решение на основе микросервисной архитектуры, позволяющее построить высоконагруженную систему, состоящую из независимых модулей – микросервисов. Была поставлена задача на модернизацию системы путем внедрения микросервисных технологий для упрощения последующей декомпозиции монолитов на отдельные сервисы и переработка распределенной информационной системы.

2 Построение микросервиса

2.1 Требования к разрабатываемому микросервису

На основании описанных в первой главе преимуществ реактивной и микросервисной архитектур над монолитной, было решено модернизировать существующую распределенную информационную систему, а именно внедрить программное обеспечение, разработанное с соблюдением принципов микросервисных и реактивных систем для последующей декомпозиции монолитов на микросервисы и упрощенного полного перехода к микросервисным технологиям.

Основной функционал сервиса, разрабатываемого в ходе выполнения данной выпускной квалификационной работы, заключается в проверке на корректность поступающих ему данных об организациях, с которыми ООО «Квартплата 24» заключила договор об оказании услуг. Согласно микросервисному подходу, разрабатываемый сервис должен быть сконцентрирован на одной бизнес-задаче, чтобы предотвратить последующее расширение функционала и разрастание микросервиса до запутанного монолита.

К функциональным требованиям данного сервиса относится прием данных об организации в виде команд, последующая проверка этих данных на основе текущего состояния системы и генерация событий в случае, если данные успешно прошли проверку. Разрабатываемый микросервис должен удовлетворять требованиям построения реактивных систем, для этого он должен быть реализован при помощи фреймворка Lightband Lagom.

Обмен данными должен происходить при помощи неблокирующего обмена сообщениями. Для этих целей будет задействован брокер сообщений, из которого сервисы, заинтересованные в этих данных, будут извлекать их по мере необходимости.

К нефункциональным требованиям данного сервиса можно отнести возможность к масштабированию, гибкости к нагрузкам и отказоустойчивость.

2.2 Проектирование архитектуры микросервиса, основанного на командах и событиях

Системы, использующие классический подход к работе с данными (CRUD – создание, чтение, обновление, удаление), имеют некоторые недостатки, которые являются критичными при разработке масштабируемого микросервисного приложения:

- такие приложения работают с различными сущностями и напрямую с базами данных, что замедляет производительность и может стать причиной проблем при масштабировании системы;
- при большом количестве пользователей, работающих одновременно, велика возможность возникновения конфликтов при работе с одним и тем же элементом данных;
- база данных хранит только итоговое состояние сущностей, при возникновении ошибки будет трудно или вовсе невозможно восстановить последовательность действий, которая привела к ошибке в данных [8], [12], [7].

Сервис, разрабатываемый в ходе данной выпускной квалификационной работы, не должен иметь вышеперечисленных недостатков, то есть должен хорошо масштабироваться, согласно микросервисному подходу, а также быть отказоустойчивым и отзывчивым. В виду специфики данных, оперируемых разрабатываемым микросервисом, немаловажным требованием является хранение данных не в виде итогового состояния, чтобы обеспечить возможность просмотра истории состояний различных сущностей.

Решением данной проблемы является шаблон, разделяющий работу сервиса на команды и события. Команда – это запрос, отправляемый сервису. Событие – действие, которое выполняет сервис при получении определенной команды. События являются списком изменений состояния той или иной сущности и служат источником итогового состояния, которое достигается путем последовательного проигрывания событий в том порядке, в котором они были созданы. Обычно такие приложения разделяются на две части: модель чтения, модель записи. Модель чтения служит для обработки команд - запросов на получение данных. Модель записи содержит в себе основную бизнес логику приложения. Архитектура сервиса, основанная на командах и событиях представлена на рисунке 6.

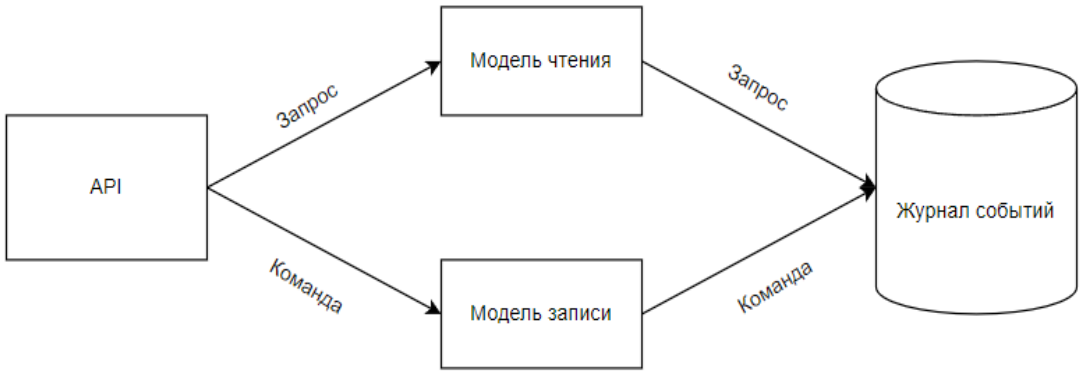

Рисунок 6 – Архитектура, основанная на командах и событиях

Шаблон команд и событий позволяет обрабатывать данные в виде набора событий, генерируемых на основании поступивших команд от пользователей или сторонних сервисов. События последовательно записываются в базу данных и в будущем могут быть использованы для получения итогового состояния или состояния в любой требуемый момент времени. Журнал событий может быть использован для повторного получения текущего состояния сущности, что упрощает поддержку сервиса и обработку

ошибок, а именно предоставляет возможность исправить алгоритмы обработки событий и проиграть их заново [6]. Журнал событий изображен на рисунке 7.

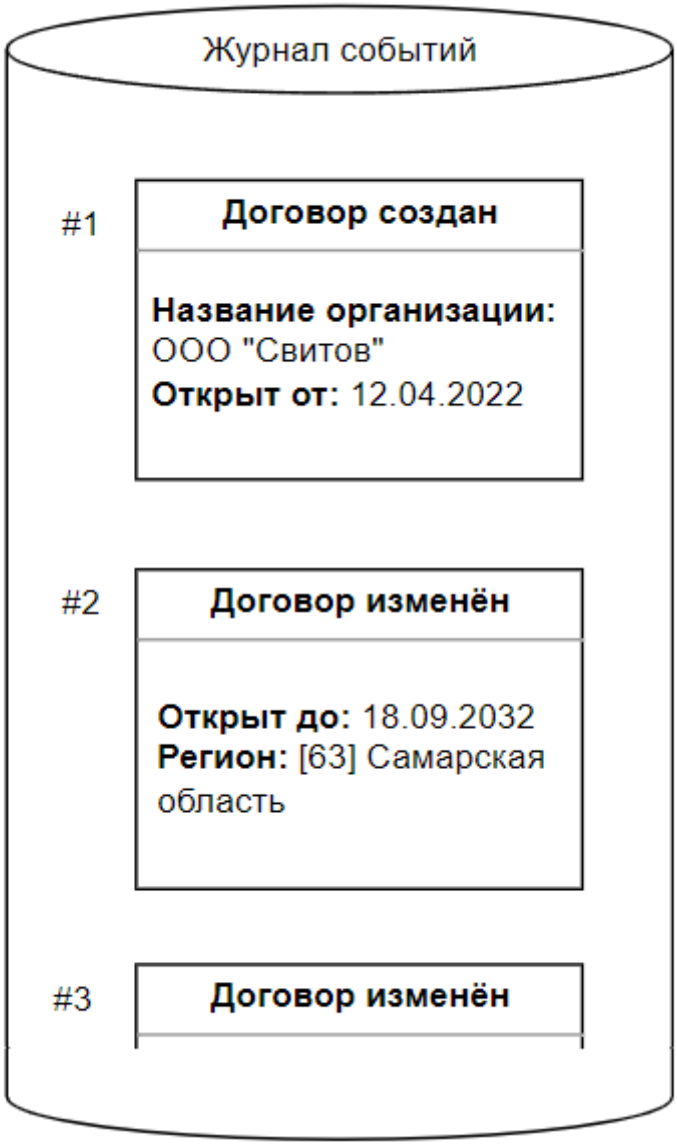

Рисунок 7 – Журнал событий

Общая архитектура микросервиса представлена на рисунке 8.

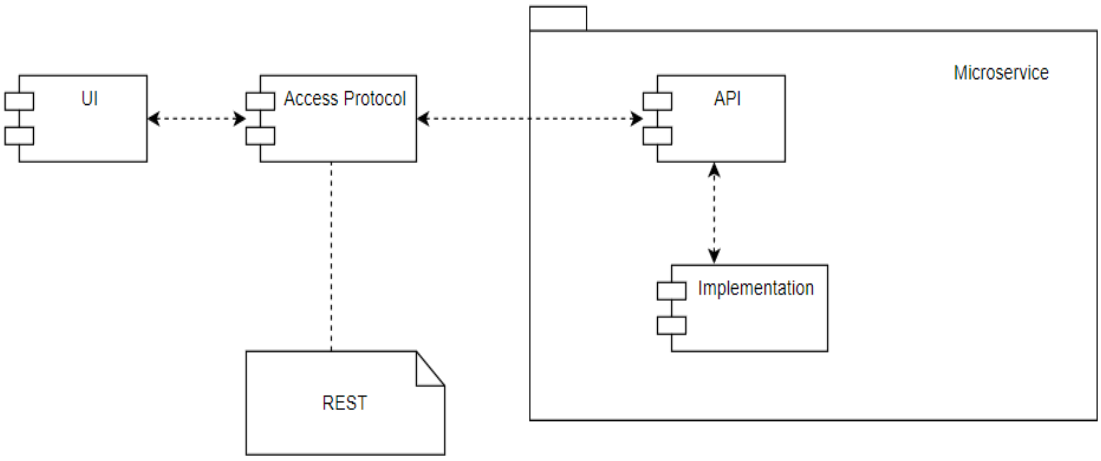

Рисунок 8 – Общая архитектура микросервиса

Сервис, разрабатываемый в ходе выполнения данной работы, будет выполнять три основные функции: создание договора, редактирование договора, закрытие договора. После достижения даты окончания договора он считается закрытым и изменить его данные будет невозможно. Таким образом мы имеем две основные команды для выполнения данных действий: создание договора, изменение договора. Команды, полученные системой, обрабатываются по следующим алгоритмам, представленным на рисунках 9 – 10.

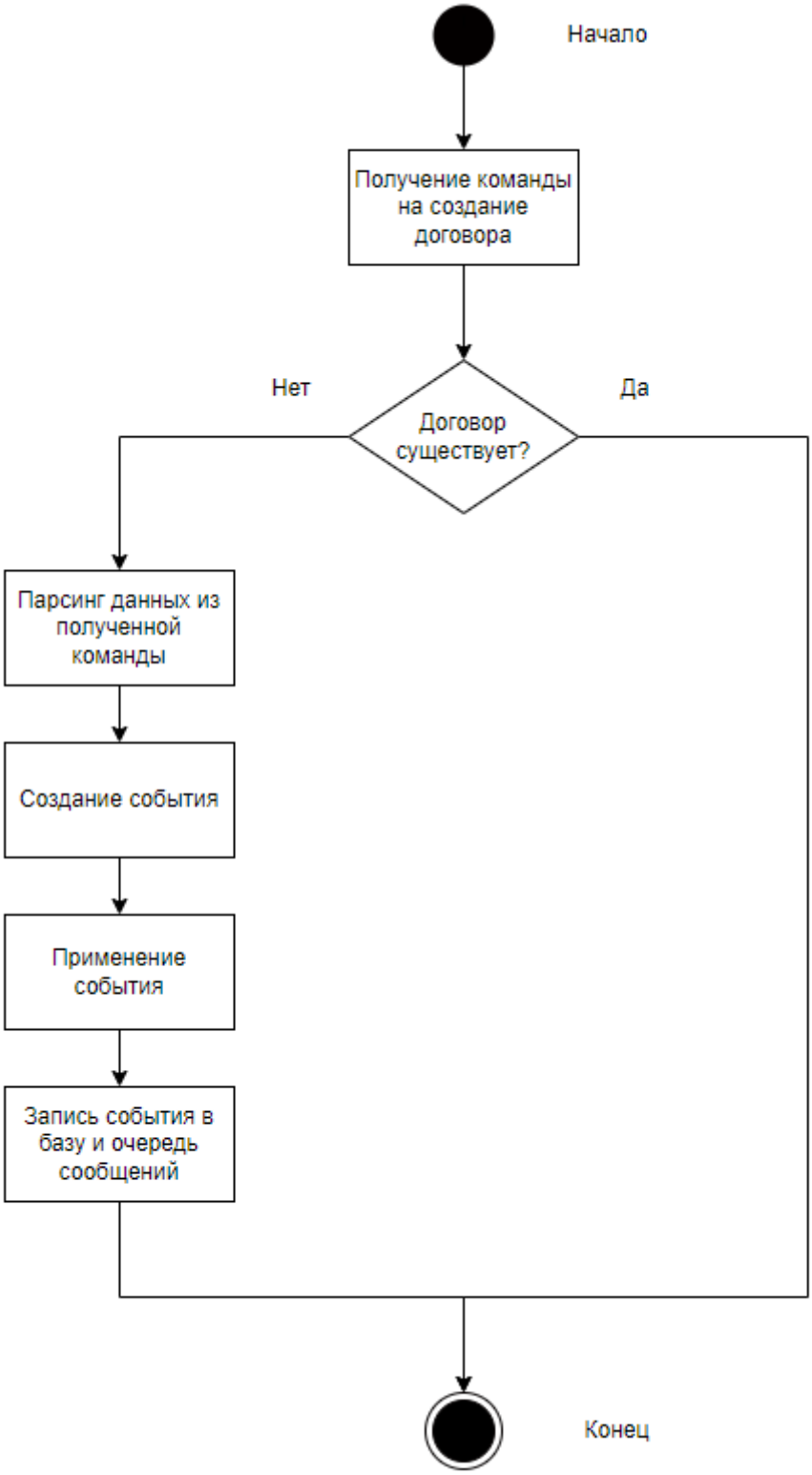

Рисунок 9 – Алгоритм обработки команды создания договора

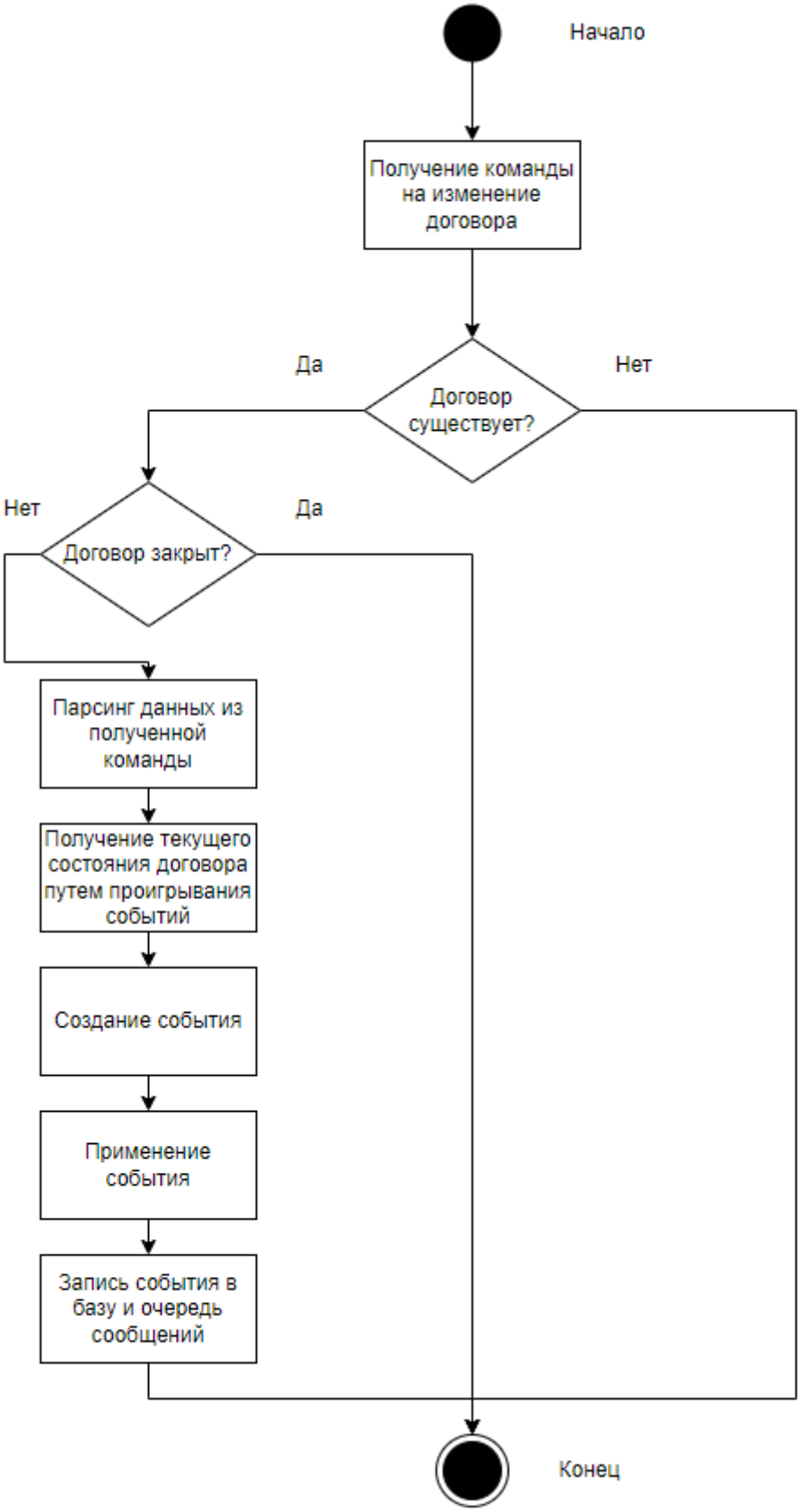

Рисунок 10 – Алгоритм обработки команды изменения договора

Диаграмма вариантов использования микросервиса на основе команд и событий представлена на рисунке 11.

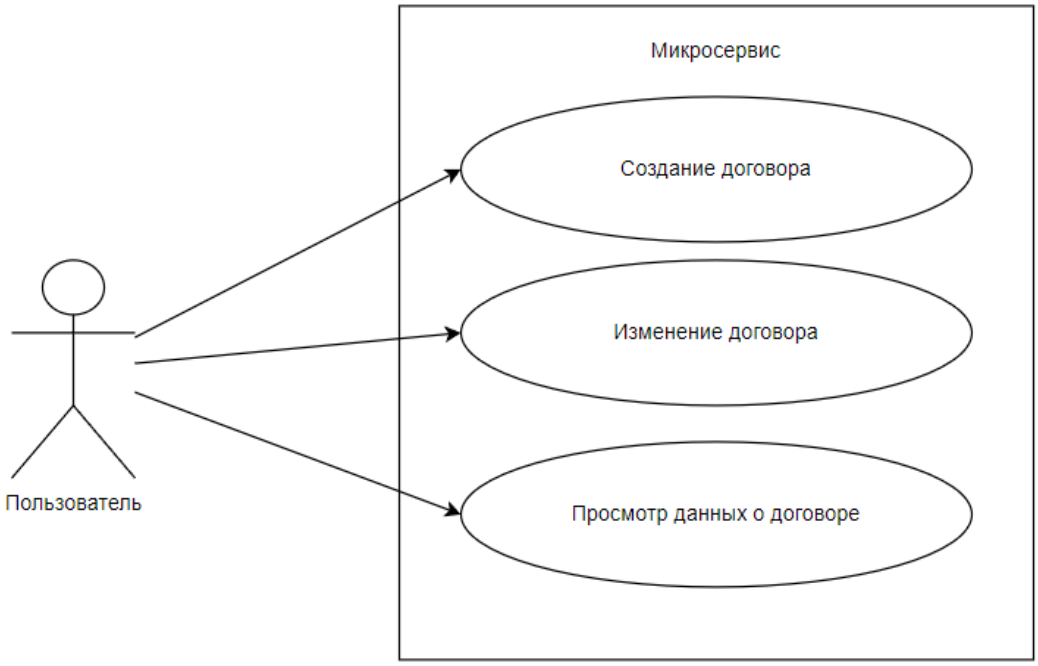

Рисунок 11 – Диаграмма вариантов использования микросервиса

Таким образом разрабатываемый микросервис предназначен для создания и изменения договоров. Сервис решает одну бизнес-задачу, что соответствует микросервисному подходу.

2.3 Описание инструмента разработки реактивных микросервисов

Lagom Framework – инструмент, созданный для разработки микросервисных приложений удовлетворяющих принципам построения реактивных систем, а значит основан на обмене данных путем отправки неблокирующих сообщений, что позволяет создавать отказоустойчивые, гибкие к нагрузкам и отзывчивые системы [21].

Для удовлетворения принципов построения таких систем в основе Lagom лежит Akka Framework, реализующий Модель Акторов, описанную

Карлом Хьюитом, Питером Бишопом и Ричардом Штайгером в 1973 году. В модели акторов всё является актором, что делает эту модель похожую на объектно-ориентированное программирование.

Актор – некоторая сущность, инкапсулирующая состояние и поведение [9]. Взаимодействие акторов в модели акторов основано на неблокирующем обмене сообщений. Получая сообщение актор способен изменять свое состояние, создавать новые акторы, отправлять сообщение другим акторам или изменять свое поведение для последующих полученных сообщений. Данные действия способны выполняться параллельно. Подобный обмен сообщениями позволяет не заботиться о синхронизации и позволяет разрабатывать системы, основанные на параллельных вычислениях.

Каждый актор имеет свой адрес, зная адрес актора можно отправить ему сообщения вне зависимость находится ли этот актор локально или удалённо. Элементами, связывающие акторы, являются их почтовые ящики. Нацеленные на обработку сообщений акторы имеют только один почтовый ящик, в который отправители помещают свои сообщения в очередь. Сообщения обрабатываются в том порядке, в котором они были помещены в почтовый ящик. Обмен сообщениями акторов изображен на рисунке 12.

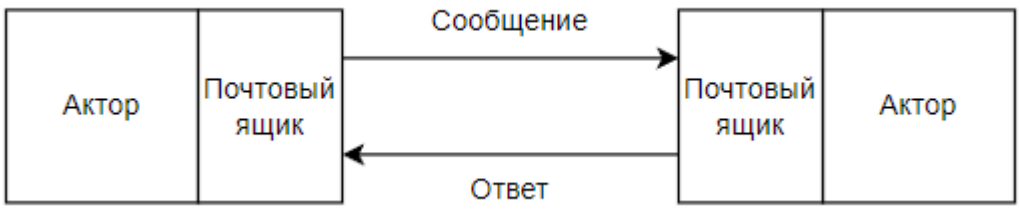

Рисунок 12 – Обмен сообщениями акторами

Акторы не могут существовать сами по себе, они существуют только в системе. При создании системы акторов автоматически создаются корневые

акторы, отвечающие за обработку ошибок своих дочерних акторов. Система акторов изображена на рисунке 13.

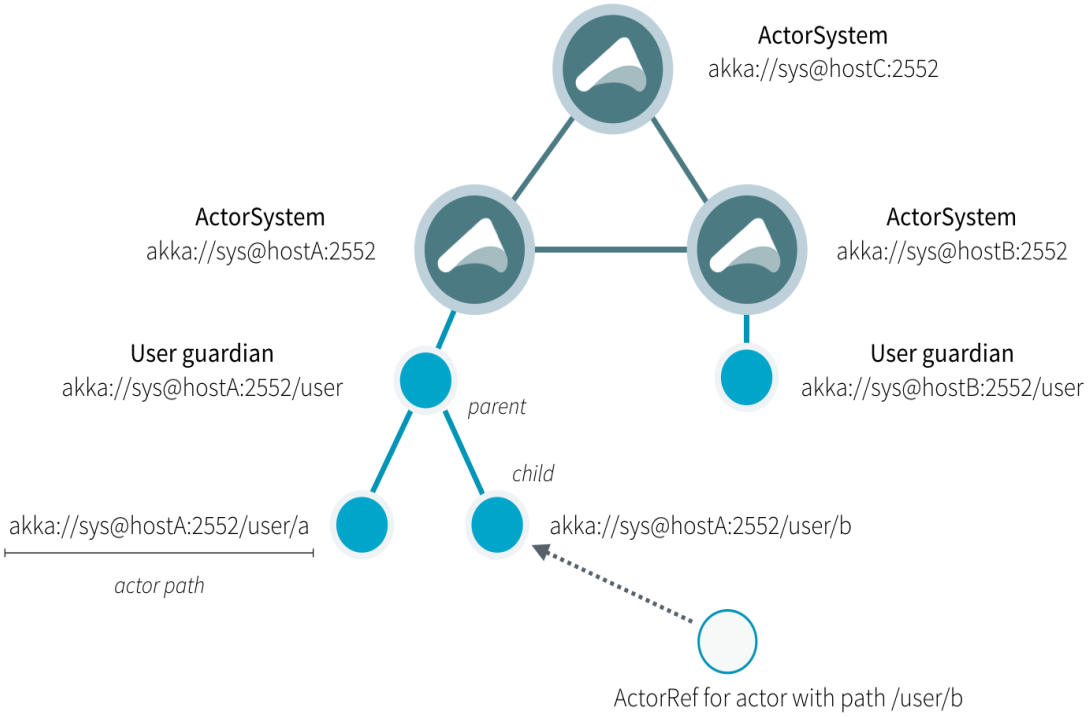

Рисунок 13 – Система акторов

Основа Lagom в виде Модели Акторов открывает широкие возможности для горизонтального масштабирования. При запуске приложение создает систему акторов и кластер, к которому в последствии, при горизонтальном масштабировании микросервиса, будут подключаться дополнительные экземпляры приложения. Несколько экземпляров приложения в кластере изображены на рисунке 14.

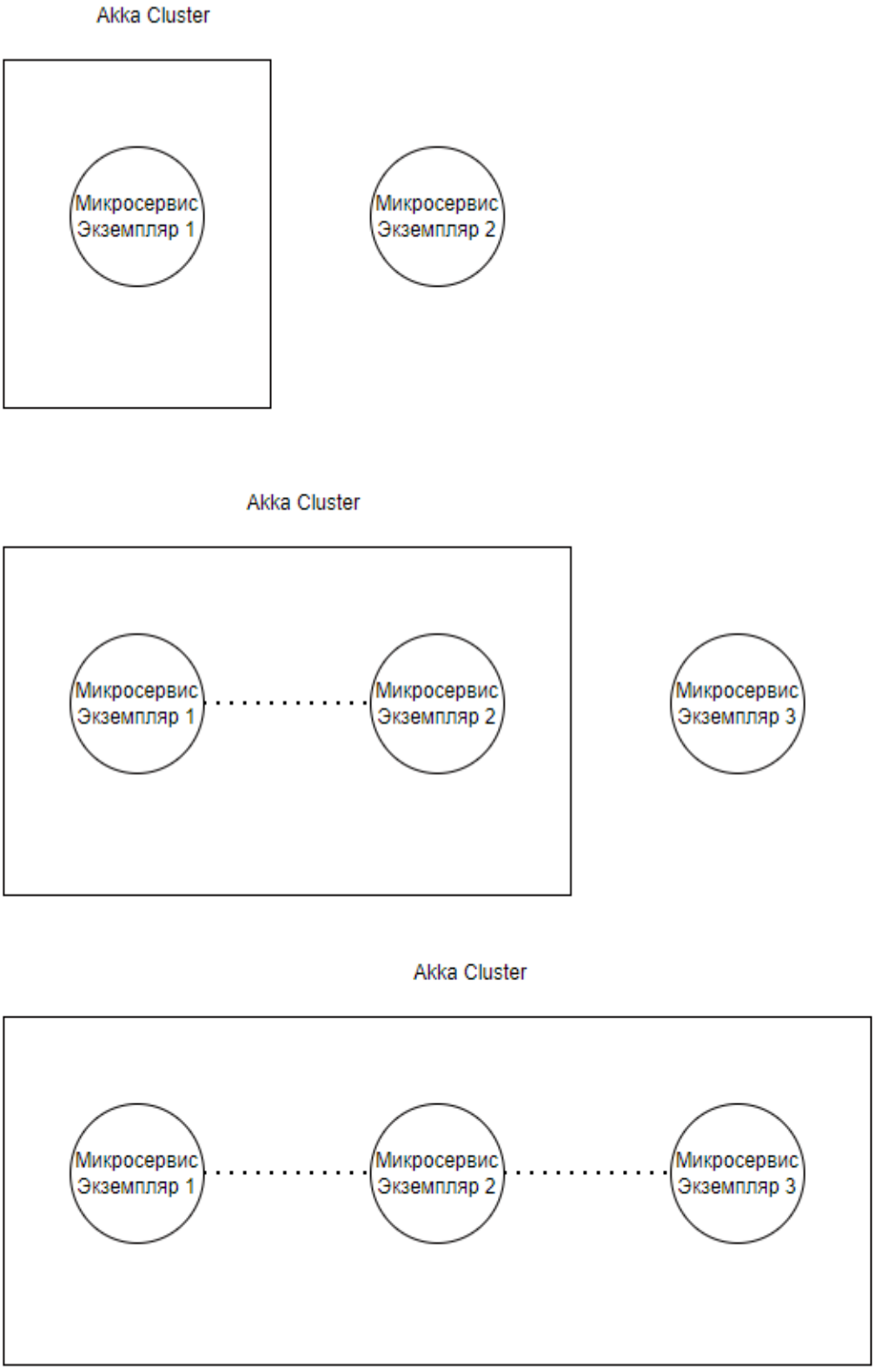

Рисунок 14 – Подключение дополнительных экземпляров приложения к кластеру

Получая команду, балансировщик Lagom делегирует ее обработку одному из экземпляров приложения в кластере. В случае успешного прохождения проверки, на основе команды генерируются события. Сущность, с которой связана полученная команда, будет выгружена в память приложения и храниться в ней в течение пяти минут в том случае, если она находится не в памяти. Во избежание ошибок при обработке события для сущности, которая не находится в памяти приложения, экземпляр приложения, получивший команду, последовательно отправит сообщение каждому из экземпляров приложения в кластере, чтобы проверить не находится ли эта сущность в памяти другого экземпляра. Если сущность находится в памяти одного из этих экземпляров, обработавшее данную команду приложение делегирует выполнение события тому экземпляру приложения, в чьей памяти находится данная сущность. Делегирование события на изменение договора, находящегося в памяти другого экземпляра микросервиса, представлено на рисунке 15.

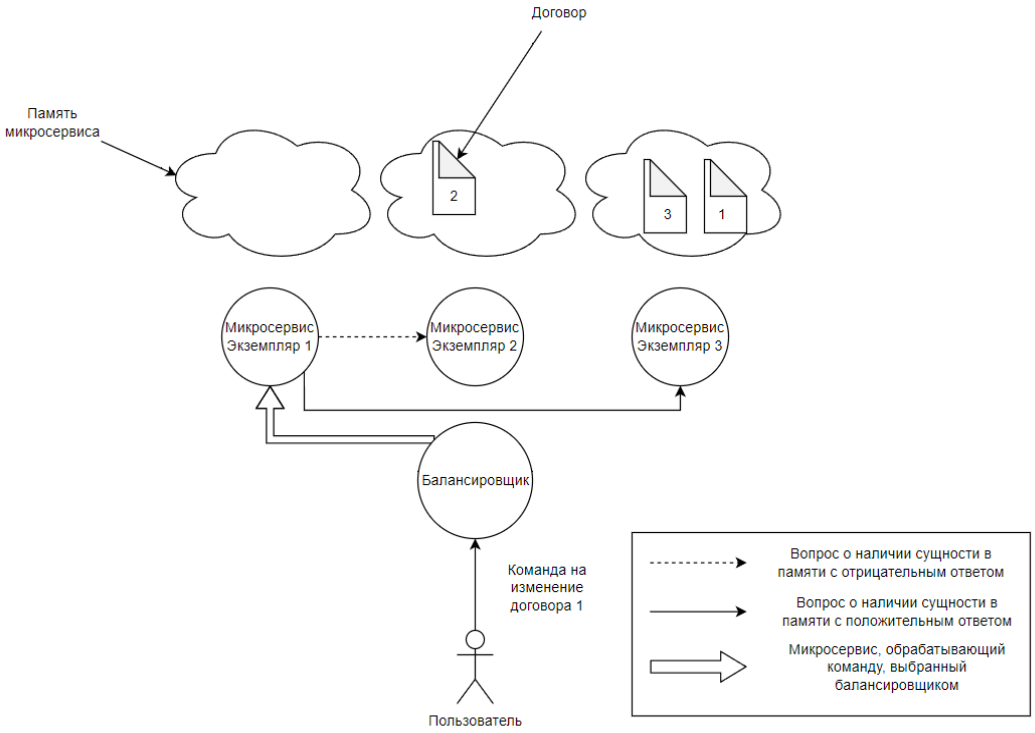

Рисунок 15 – Делегирования события на изменение договора

Договор хранится в памяти сервиса 5 минут, после выгружается из нее, если с ним не происходит никаких событий.

2.4 Доставка сообщений между сервисами экосистемы

Разработка микросервисов, не смотря на все свои преимущества, не лишены некоторых проблем, основной из которых является интеграция микросервиса с существующую систему. В основе микросервисной и реактивной архитектур лежит неблокирующий обмен сообщениями, позволяющий разрабатывать модули одной системы независимо друг от друга, что гарантирует высокую отказоустойчивость системы. Информационная система, элементы которой не обладают жесткими связями друг с другом, останется в работоспособном состоянии в том случае, если один из ее элементов выйдет из строя по каким-либо причинам. Подобного результата можно достичь, реализовав обмен данными между микросервисами при помощи очередей сообщений.

Микросервисы, чье общение реализовано путем обмена данными через очереди сообщений, можно представить в виде производителя данных, сервера очереди и потребителя. Модуль системы может быть как потребителем данных, так и производителем. Сервисы, использующие очередь сообщений, изображены на рисунке 16.

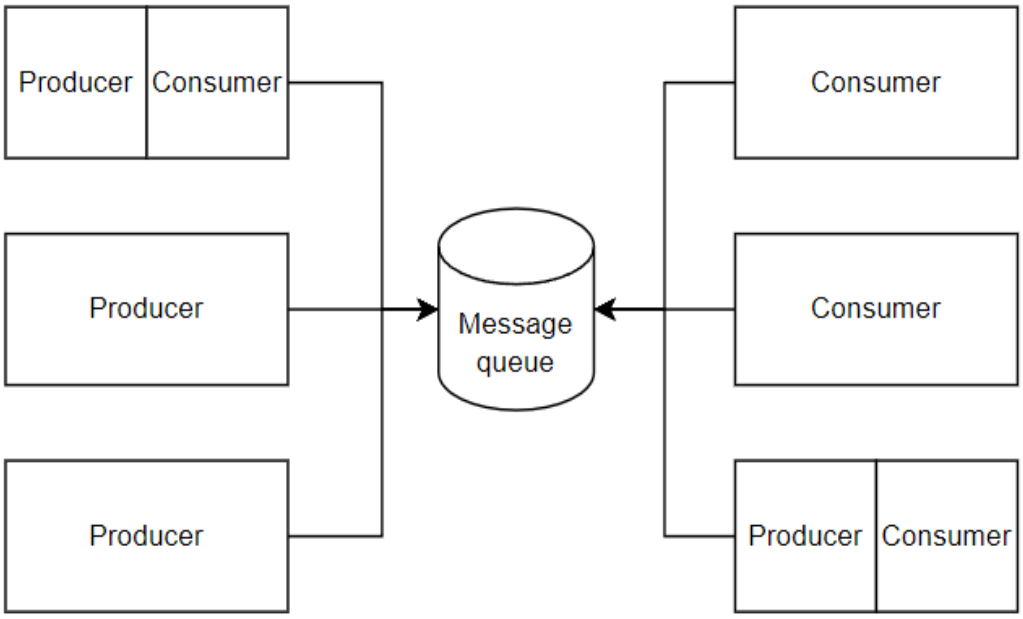

Рисунок 16 – Сервисы, передающие данные в очередь сообщений

Реализация основного функционала очередей сообщений представлена в виде двух моделей взаимодействия сервера и клиента:

- модель «Push» – данная модель взаимодействия клиента и сервера подразумевает непрерывную передачу данных от сервера к клиенту, что может негативно сказываться на работоспособности системы в том случае, если потребитель данных будет обрабатывать их медленнее, чем производитель их отправляет;
- модель «Pull» – данная модель предоставляет потребителю данных возможность самому управлять потоком получаемых сообщений в зависимости от своей нагрузки и скорости обработки данных.

Для интеграции разрабатываемого микросервиса в существующую распределенную информационную систему необходимо сравнить по наиболее важным критериям два наиболее популярных инструмента, позволяющих обеспечить обмен данными при помощи очередей сообщений, а именно RabbitMQ и Apache Kafka. Сравнение представлено на таблице 1.

Таблица 1 – Сравнение особенностей очередей сообщений

Критерии	**RabbitMQ**	**Apache Kafka**
Сохранение сообщений	нет	да
Автоматическая балансировка нагрузки	нет	да
Масштабируемость	нет	да
Маршрутизация	да	нет
Порядок сообщений	нет	да

Apache Kafka сохраняет сообщения в журнал и хранит их до тех пор, пока не будет запущена его очистка, что делает Apache Kafka достоверным источником ранее обработанных сообщений [10].

Apache Kafka производит автоматическую балансировку сообщений, распределяя нагрузку между получателями сообщений внутри одной темы.

Apache Kafka горизонтально масштабируемый, что позволяет запустить несколько её экземпляров внутри одного кластера и обеспечивает их совместную работу как одно целое.

RabbitMQ имеет 4 возможных способа маршрутизации для постановки сообщений в очереди, что позволяет производить более гибкие настройки их передачи и использование различных архитектурных шаблонов [23].

Одним из основных преимуществ Apache Kafka над RabbitMQ является работа по принципу модели «Pull», что позволяет потребителям данных доставать сообщения из очереди только тогда, когда они имеют возможность их обработать.

Apache Kafka обеспечивает получение данных потребителем в том порядке, в котором производитель данных отправил их через очередь сообщений.

На основе вышеперечисленных преимуществ Apache Kafka было принято решение использовать инструмент, предоставляемый Apache, для последующей передачи сообщений между сервисами.

Apache Kafka – разработанный в 2011 году распределенный и отказоустойчивый брокер сообщений. Используется для доставки сообщений в высоконагруженных системах, основными преимуществами над другими брокерами является высокая пропускная способность и возможность горизонтального масштабирования, такое масштабирование позволяет без проблем разворачивать несколько экземпляров брокера, нагрузка между которыми сбалансируется автоматически. Apache Kafka разработан на таких языках программирования как Java и Scala.

Внутренняя архитектура Apache Kafka состоит из нескольких элементов:

- Kafka Broker – сервер Apache Kafka, выполняющий функции посредника между потребителем данных и производителем данных, он получает сообщения от производителя, может хранить их какое то время, а потом предоставить их получателю;
- Apache Zookeeper – координатор, представленный в виде реляционной базы данных. Используется для синхронизации данных между несколькими брокерами;
- Kafka controller – в кластере, состоящем из нескольких Kafka Broker, выделяется один для обеспечения консистентности данных, для этих целей Kafka Controller использует Apache Zookeeper.

Элементы Apache Kafka представлены на рисунке 17.

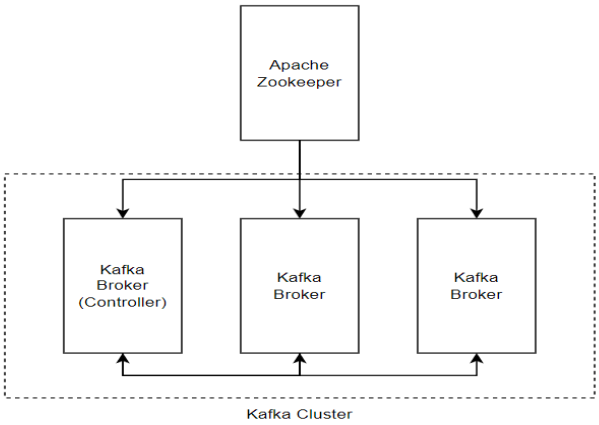

Рисунок 17 – Элементы Apache Kafka

Сообщения, с которыми работает Apache Kafka, представляют из себя набор пар ключ-значение. Элементы сообщения представлены на таблице 2.

Таблица 2 – Элементы сообщения Apache Kafka

Поле сообщения	**Описание**
Key	Ключ (необязательное поле) используется для распределения сообщения по кластеру.
Value	Сообщение, представленное в виде массива байт.
Timestamp	Время отправки сообщения. Если при отправке мы не указываем это поле, то оно будет автоматически проставлено во время обработки сообщения брокером.
Headers	Набор пользовательских атрибутов в формате ключ-значение.

Сообщения, передаваемые через Apache Kafka, помещаются в какой-либо Topic, объединяющий в себе сообщения одного типа. Topic представляет из себя поток данных-сообщений, данные из которого извлекаются

потребителем в том порядке, в котором были переданы в очередь сообщений. Topic изображен на рисунке 18. Для достижения лучшей производительности в параллельной работе данные внутри Topic разделены на части – Partition. Разделение на partition изображено на рисунке 19. Это же разделение обеспечивает повышенную отказоустойчивость Apache Kafka за счет дублирования сообщений на различных Kafka Broker. В случае, если один из Kafka Broker выйдет из строя, другой его экземпляр в Kafka кластере будет иметь копию partition. Дублирование данных изображено на рисунке 20.

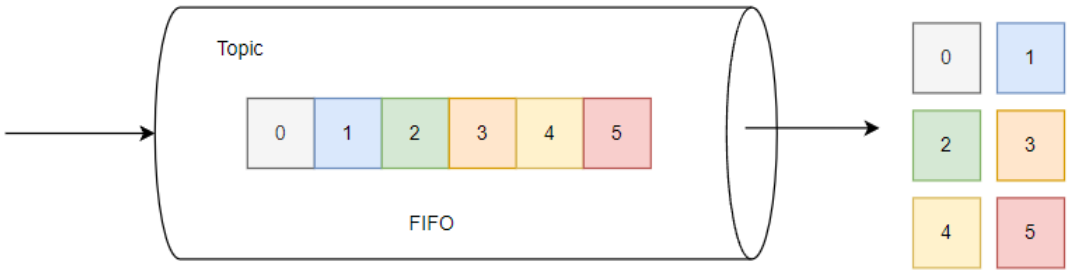

Рисунок 18 – Kafka topic, работающий по принципу очереди

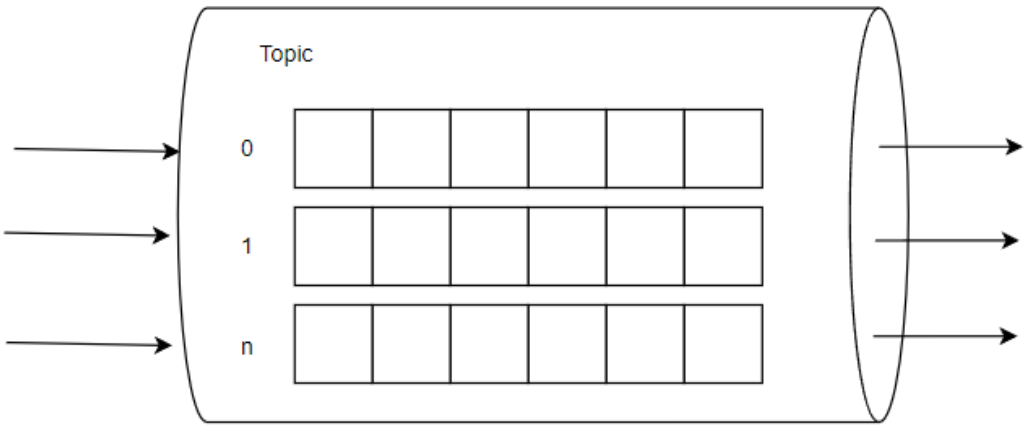

Рисунок 19 – Topic с разделением на partition

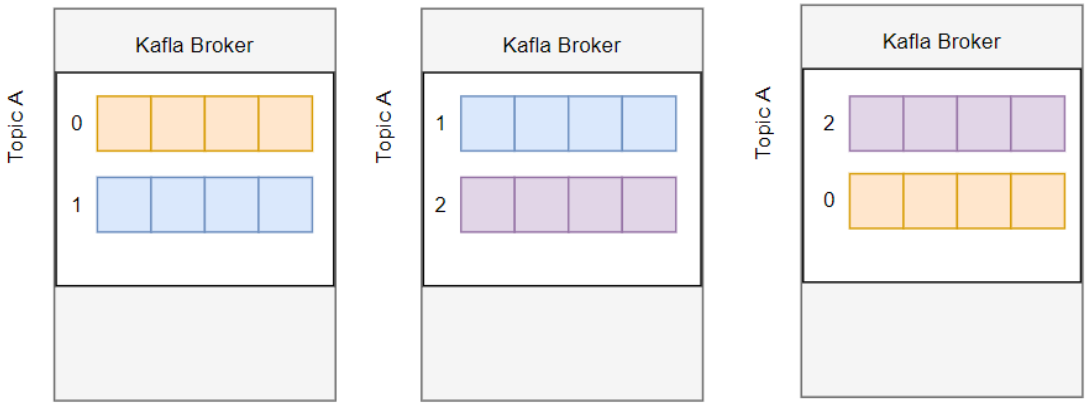

Рисунок 20 – Репликация данных

2.5 Разработка диаграммы классов микросервиса

Разрабатываемый микросервис будет разделен на два модуля, API и реализация. Команды будут поступать в виде POST и GET запросов.

В таблице 3 представлены основные интерфейсы, классы и их функция в модуле API.

Таблица 3 – Классы модуля API

Название класса	**Роль класса**
Command	Интерфейс, от которого наследуются все команды
APICommand	Описывает команды сервиса
APIEvent	Описывает события сервиса
AgreementService	Сопоставляет полученный HTTP запрос с методом его обработки

В таблице 4 представлены основные интерфейсы и классы основного модуля.

Таблица 4 – Классы основного модуля с реализацией

Название класса	Роль класса
Agreement	Описывает сущность “Договор”
AgreementCommands	Описывает команды договора
AgreementEvents	Описывает события договора
AgreementProcessor	Содержит реализацию событий договора
Tariff	Описывает сущность “Тариф”
AgreementServiceImpl	Реализация методов, определенных в AgreementService

Диаграмма классов, реализующих интерфейс Command, представлена на рисунке А.1.

В таблице 5 представлено назначение классов, реализующих интерфейс Command.

Таблица 5 - Классы, реализующий интерфейс Command

Название класса	Назначение класса
RegisterPaymentOperatorAgreement	Создание договора со сборщиком платежей.
UpdatePaymentOperatorChannels	Обновление набора каналов приема платежей
RemovePaymentOperatorChannels	Удаление каналов приема платежей
RewriteTariffExclusion	Настройка исключений в тарифе
UpdateTarrif	Настройка тарифа по договору
GetTariffAssignationStatus	Запрос списка договор использующих определенный тариф
ShowJournalEvents	Показать журнал событий
UpdatePaymentOperatorBankDetails	Обновление ссылок на банковские реквизиты
RegisterProcessingAgreement	Создание договора процессинга платежей
UpdateProcessingAgreementPrefixes	Обновление префиксов в договоре на прием платежей
RewriteProcessingChannels	Изменение условий по каналам приема средств для клиента
RemoveProcessingChannels	Удаление канала приема средств для клиента

Продолжение таблицы 5

UpdateSplitChannels	Изменение условий по каналам распределения средств
RemoveSplitChannels	Удаление каналов распределения средств
CreateTariff	Создание нового тарифа
RegisterTransferAgreement	Создание договора перевода средств
UpdateTransferSchedule	Обновление условий автоперевода
UpdateTransferBankDetails	Изменение банковских реквизитов по договору на перевод
SetParties	Изменить стороны по договору
AssignTarrif	Установить тариф по договору на процессинг

В таблице 6 представлены события, генерируемые на основе поступивших команд.

Таблица 6 – События сервиса

Название класса	**Назначение класса**
POAgreementCreated	Создан договор со сборщиком платежей
SourceChannelsAdded	Добавлены каналы приема платежей Сборщика Платежей
SourceChannelsUpdated	Обновлены каналы приема платежей Сборщика Платежей
SourceChannelsRemoved	Удалены каналы приема платежей Сборщика Платежей
POBankUpdated	Изменен расчетный банк Сборщика Платежей
TransferAgreementCreated	Создан Договор на Перевод Средств
ScheduleUpdated	Обновлено расписание перевода
TrBankUpdated	Обновлены банковские реквизиты в Договоре на Перевод
ProcessingAgreementCreated	Создан Договор на Прием Платежей
ProcessingAgreementPrefixesUpdated	Обновилось значение строки префиксов в договоре на приём платежей
ProcessingChannelsUpdated	Обновлены условия по каналам приема платежей Клиента
SplitChannelsAdded	Добавлены каналы расщепления платежей
SplitChannelsUpdated	Изменены каналы расщепления платежей
SplitChannelsRemoved	Удалены каналы расщепления платежей
TariffAssigned	По Договору установлен новый тариф
TariffReset	По Договору снят тариф

Продолжение таблицы 6

TariffUpdated	По Договору обновлен тариф
PeriodEndUpdated	Обновлена дата окончания договора
PartiesSet	Изменить Стороны(участники) по договору
TariffExclusionsUpdated	Добавлены исключения в каналы тарифа

Не смотря на количество команд, которые данный сервис может принимать, они относятся к трем типам команд: создание, изменение, удаление.

2.6 Архитектура микросервиса в контексте интеграции в существующую экосистему

На данный момент каждый из основных сервисов экосистемы в ООО «Квартплата 24» является приложением, построенным на основе монолитной архитектуры. Находясь в кластере, сервисы общаются друг с другом напрямую.

При разработке данного микросервиса использовались современные технологии доставки сообщений, повышающие отказоустойчивость системы. Интегрированный в экосистему микросервис представлен на рисунке Б.1.

Интеграция микросервиса, разработанного в ходе выполнения данной выпускной квалификационной работы, создаст твердую основу для последующей декомпозиции монолитных приложений и полного перехода к микросервисной архитектуре. Экосистема, полностью состоящая из микросервисов представлена на рисунке Б.2.

В данной главе были сформированы требования к разрабатываемому микросервису, выполняющему функцию сервиса для создания договоров с организациями-клиентами и различными платежными системами.

Описан принцип работы систем, построенных на разделении команд и событий, представлены алгоритмы обработки основных команд, принимаемых данным сервисом: создание договора, изменение договора и закрытие договора.

Рассмотрены недостатки и преимущества нескольких инструментов, созданных для обмена сообщения, что решает одну из основных проблем проектирования микросервисов – передача данных между сервисами.

Построена диаграмма классов команд, описаны основные команды, события и их назначения.

Представлена модернизированная архитектура экосистемы сервисов в ООО «Квартплата 24» с интегрированным в нее сервисом договоров, а также экосистема, полностью состоящая из микросервисов.

3 Разработка микросервиса

3.1 Выбор средств реализации микросервиса

При разработке программного обеспечения, основанного на микросервисной архитектуре, важно подобрать инструменты, которые в полной мере обеспечивают успешное выполнение поставленной цели, т.е. соответствуют принципам, лежащим в основе построения приложений-микросервисов.

Одним из условий выполнения данной выпускной квалификационной работы была реализация микросервисного приложения с использованием фреймворка Lagom. В виду того, что в основе данного фреймворка лежит модель акторов, реализованная Akka Framework, поддерживаемыми языками данного инструмента являются два языка программирования: Java, Scala.

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) [11], [13], [18].

Scala – мультипарадигмальный 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 [24].

Необходимо отметить, что Scala, как и Java, является языком, который компилируется в байт код и выполняется на Java Virtual Machine. По умолчанию Scala несёт в себе все основные Java библиотеки и имеет с ним полную совместимость.

Сравнение Java и Scala по основным критериям выбора языка программирования для выполнения данной выпускной квалификационной работы представлено в таблице 7.

Таблица 7 – Сравнение Java и Scala

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

Декларативность языка программирования Scala делает его более лаконичным и существенно сокращает объем кода.

Объектно-ориентированная парадигма Java является более распространенной среди большинства разработчиков программного обеспечения, императивный стиль Java делает код более прозрачным, что позволяет проще находить и устранять ошибки в алгоритмах [25].

Не смотря на мультипарадигмальность языка программирования Scala, больший уклон в сторону функционального стиля стимулирует писать рекурсии вместо привычных циклов. Все встроенные алгоритмы в Scala реализованы при помощи хвостовых рекурсий, что делает его более производительным на фоне Java [3], [15].

В виду схожести этих языков, сравнение проводилось по небольшой выборке особенностей данных языков, по итогу которого в качестве языка программирования для реализации был выбран язык программирования Scala.

Выбирая интегрированную среду разработки, нужно ориентироваться на выбранный язык программирования, существует три наиболее популярных среды разработки с поддержкой языка Scala: Eclipse, NetBeans и Intellij IDEA (Community Edition).

Eclipse – бесплатная среда разработки, позволяющая разрабатывать кроссплатформенное ПО. Разработана на Java компанией Eclipse Foundation.

NetBeans – бесплатная среда разработки для создания ПО на множестве языков, таких как: Java, Python, PHP, JavaScript, C, C++. Спонсируется и поддерживается компанией Oracle.

Intellij IDEA (Community Edition) – современная интегрированная среда разработки, поддерживающая множество языков программирования, имеющая интеграцию с системами управлениями версий [17], [20].

В виду схожих функциональных возможностей любой из современных сред среды разработки, проведем сравнение данных сред разработки по следующим критериям:

- удобство интерфейса;
- современность среды разработки;
- работа с плагинами и библиотеками – под работой с плагинами и библиотеками подразумевается удобство в их подключении;
- опыт использования – персональный опыт в использовании данной среды разработки.

Результат сравнения представлен в таблице 8.

Таблица 8 – Сравнение интегрированных сред разработки

Критерий выбора	**Eclipse**	**NetBeans**	**Intellij IDEA (Community Edition)**
Удобство интерфейса	6	7	8
Современность среды разработки	6	6	8
Работа с плагинами и библиотеками	7	7	9
Опыт использования	4	0	9
Итого:	23	20	34

В результате сравнение трех данных интегрированных сред разработки, была выбрана Intellij IDEA (Community Edition) разрабатываемая компанией JetBrains. Данная среда разработки имеет современный, функциональный интерфейс, удобный инструмент для подключения различных плагинов, в том числе Scala плагин, и библиотек. Наиболее важным преимуществом является большой опыт разработки именно в этой интегрированной среде.

В виду того, что существующие сервисы экосистемы используют PostgreSQL в качестве хранилища данных, для журнала событий будет использоваться именно эта СУБД.

PostgreSQL – базирующаяся на языке SQL объектно-реляционная СУБД. Основными ее особенностями являются: надежность, расширяемость, производительность [22].

3.2 Реализация основных модулей микросервиса

Функциональность данного сервиса заключается в создании, изменении и закрытии договоров с организациями-клиентами и различными платежными системами. Данный функционал будет реализовываться при помощи шаблона источника событий, то есть с разделением на команды и события. Выделим основные функции разрабатываемого микросервиса:

- получение команды;
- проверка команды на корректность;
- создание событий на основе команды;
- сохранение созданных событий в журнал событий;
- передача событий в брокер сообщений.

Для создания шаблонного проекта для последующей разработки используем SBT – автоматический сборщик проектов, написанных на Scala и Java. Создание шаблонного Lagom проекта производится путем выполнения консольной команды sbt new lagom/lagom-scala g8. Во время создания проекта сборщик спросит данные для сервиса: Название сервиса, инвертированное доменное имя организации для создания пакетов (com.company), версия приложения, желаемую версию фреймворка Lagom [16]. Данные проекта представлены на рисунке 21.

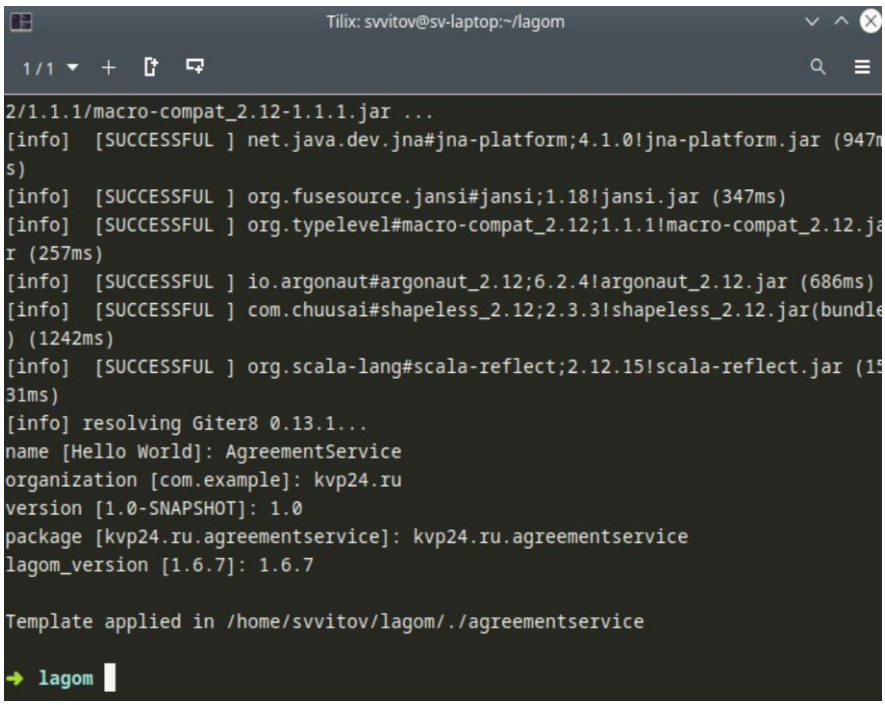

Рисунок 21 – Данные проекта

Созданный проект имеет двухмодульную структуру, модуль API и модуль имплементации методов, вызывающихся из API при получении команды. Структура проекта представлена на рисунке 22.

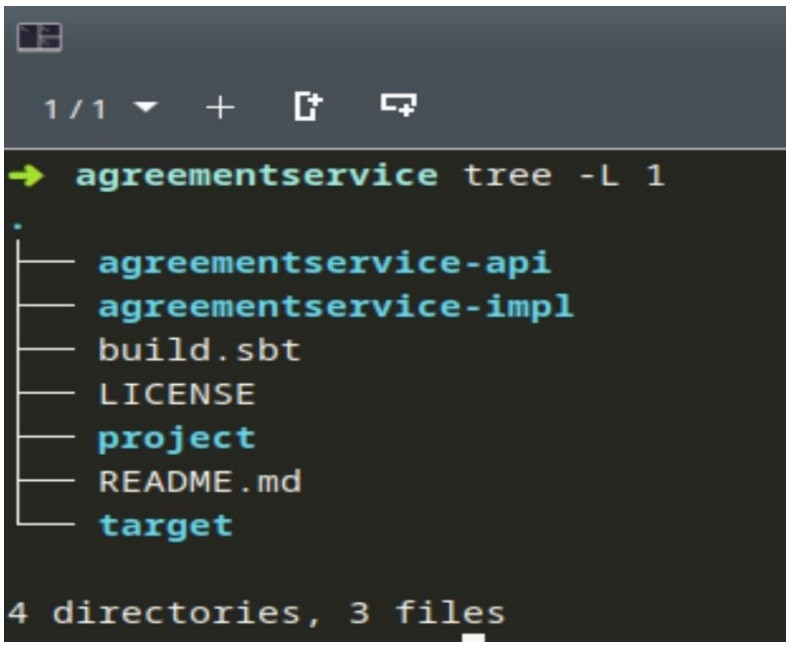

Рисунок 22 – Структура проекта

Данный шаблонный проект соответствует шаблону источника событий и, способный получать и обрабатывать команды, является полностью работоспособным. Для проверки успешности сборки проекта используем команду sbt runAll. Результат запуска проекта представлен на рисунке 23.

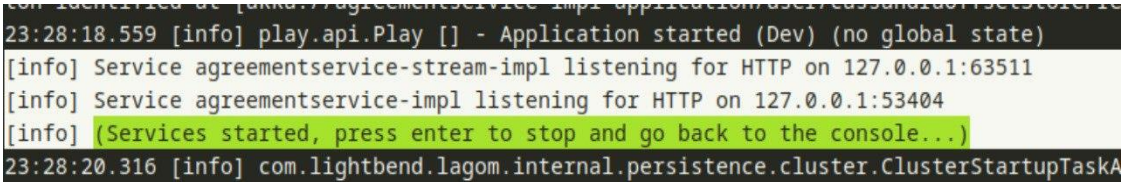

Рисунок 23 – Тестовый запуск проекта

Реализация модуля API начинается с описания команд, которые будет обрабатывать сервис договоров. Чтобы создать команды, нужно объединить их в группу, для этого создадим пустой интерфейс Command, который будут реализовывать все команды данного сервиса. Код с реализацией команд сервиса представлен в Приложении В.

Аналогично командам, события реализуют специальный интерфейс Event, который отличается от Command тем, что каждое событие должно иметь идентификационный номер, поэтому интерфейс Event содержит поле id. Код с реализацией команд представлен в Приложении Г.

Следующим шагом в реализации модуля API будет создание интерфейса, описывающего методы, которые в последствии будут вызваться при получении определенной команды. Для реализации такого интерфейса он должен наследоваться от встроенного интерфейса фреймворка Lagom – Service. Интерфейс Service предоставляет один метод – descriptor. В нем будут описаны запросы, полученные сервисом, их сопоставление с методами сервиса. Фрагмент кода с методами сервиса и их сопоставлением с запросами сервиса представлен в Приложении Д.

Модуль имплементации данного сервиса содержит методы, описанные ранее в модуле API. Класс имплементации принимает в конструкторе объект

класса ClusterSharding, это нужно для проверки того не работает ли один из экземпляров приложения в кластере с сущностью, на изменение или удаление которой пришла команда. После проводится проверка на возможность исполнения требуемых командой действий. В случае, когда команда не будет выполнена, метод из класса имплементации возвращает ответ Fail, ответ Done в случае успешной проверки команды. После проверки начинается процесс создания событий. После создания событий никаких проверок не производится. Фрагмент кода с реализацией интерфейса API представлен в Приложении Е.

В качестве журнала событий используется объектно-реляционная база данных PostgreSQL. Пропагандируя минимальную работу с базой данных, концентрируясь на событиях, создание журнала событий ограничивается созданием базы данных и подключение ее в конфиг-файле Lagom application.conf. Для создания базы данных необходимо ввести команду в терминале createdb agreement. Подключение базы данных к нашему сервису представлено в Приложении Ж. После первого запуска структура нашей базы данных будет создана автоматически в соответствии со спецификацией Lagom. В ней будут храниться события, логи и снапшоты. Снапшоты выступают в роли контрольных точек и нужны для того, чтобы не тратить ресурсы на проигрывание всех событий с самого начала существования сущности.

Общение между данным сервисом договоров и другими сервисами экосистемы производится при помощи Apache Kafka. При запуске приложения Lagom автоматически запускает Apache Kafka по адресу, указанному в конфигурационном файле application.conf. При первой отправке сообщений в брокер сообщений автоматически будет создан топик для сообщений данного сервиса в случае, если он не был создан ранее. При желании можно указать адрес другого брокера сообщений. Настройки Apache Kafka представлены в Приложении И. В последствии из этого брокера сообщений будут получать данные сервисы, заинтересованные в этих данных.

3.3 Тестирование работоспособности микросервиса

Для проверки работоспособности разработанного сервиса договоров проведем тестирование основных функций данного микросервиса.

Сервис договоров, разработанный в ходе выполнения выпускной квалификационной работы, выполняет следующие функции: создание договора, изменение договора, закрытие договора, передача информации о договорах другим сервисам экосистемы через Apache Kafka.

Создание договора представлено на рисунке 24.

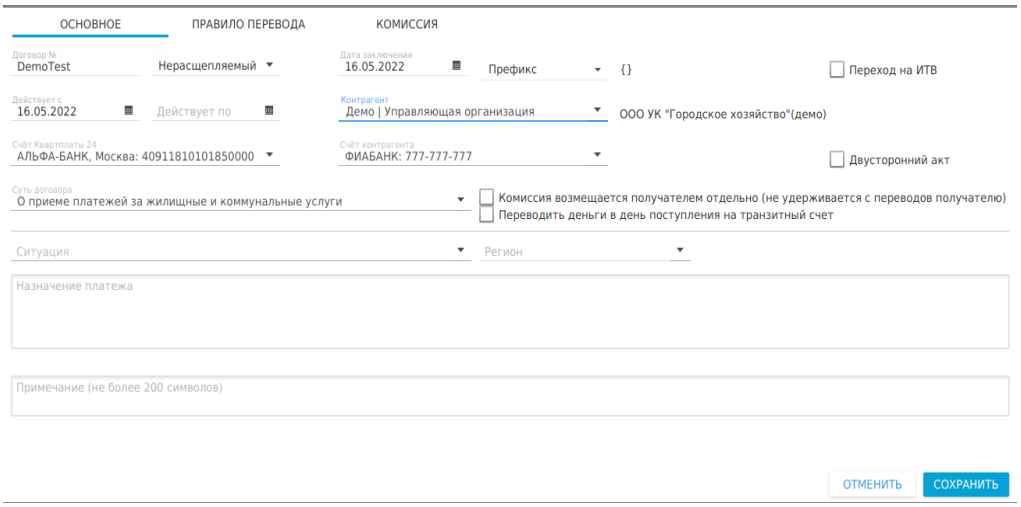

Рисунок 24 – Окно создания договора

Нажатие кнопки «Сохранить» создаст договор в том случае, если договора с таким номером не существует. В случае успешного сохранения договора мы увидим сообщение рядом с кнопкой сохранения. Сообщение об успешности действия представлено на рисунке 25.

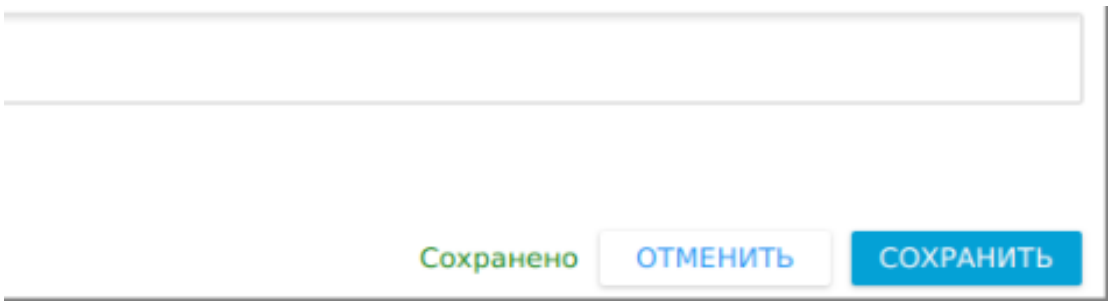

Рисунок 25 – Сообщение об успешно сохраненном договоре

Получая команду на создание договора, на основе данных, переданных в команде, создаются события, после записи в журнал событий они будут переданы в Apache Kafka для последующей передачи в сервисы, которые заинтересованы в этих данных. Сообщение, переданное в Apache Kafka, представлено на рисунке 26.

Рисунок 26 – Созданный договор, переданный в Apache Kafka

Изменение договора происходит в том же окне, после создания договора, введем новые данные – регион, дату закрытия договора. Измененный договор представлен на рисунке 27.

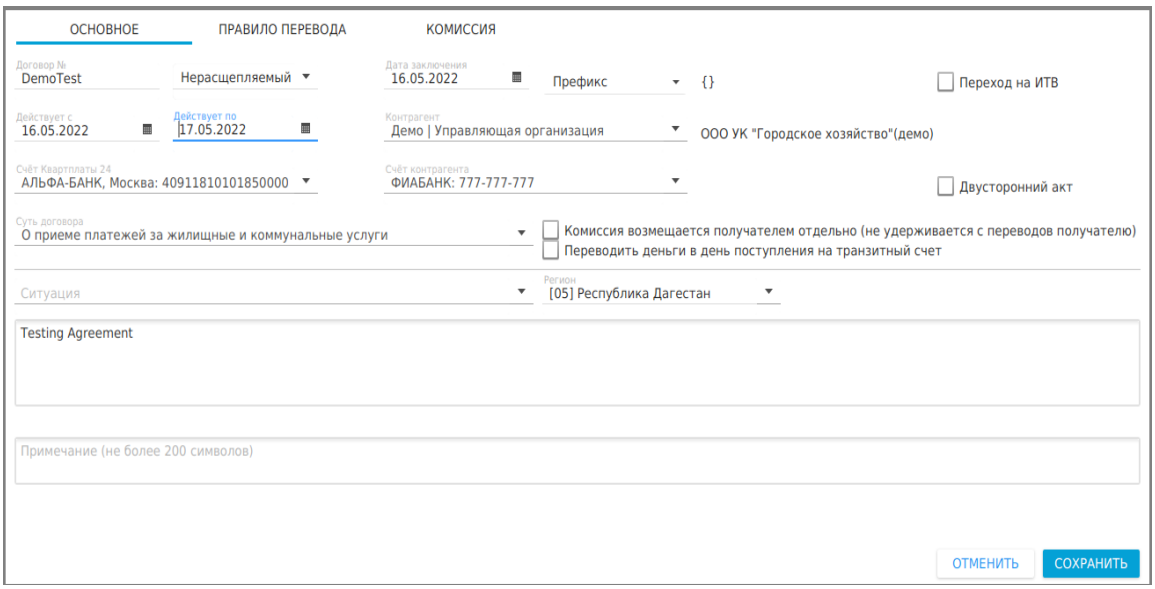

Рисунок 27 – Изменение договора

Процедура сохранения изменений происходит так же, как процедура сохранения. Получая команду на изменение договора происходит проверка на существование такого договора, актуальности договора и введенных данных на возможность изменения. Таким образом, например, будет невозможно изменить договор, не изменив никаких данных. Сообщение о изменении договора в Apache Kafka представлено на рисунке 28.

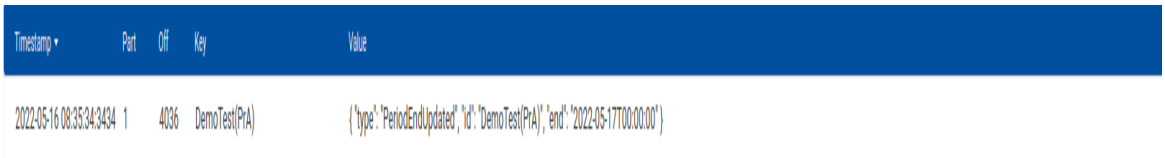

Рисунок 28 – Сообщение о изменении договора в Apache Kafka

После сохранения измененного договора обновляется не весь договор, а только измененная его часть. В Apache Kafka передается только измененная часть договора.

Заключение

Данная выпускная квалификационная работа посвящена построению высоконагруженных, отказоустойчивых распределенных систем при помощи микросервисных технологий, основанных на реактивной архитектуре.

Соблюдение принципов построения микросервисов, основывающихся на принципах реактивных систем, позволяют разрабатывать информационные системы, соответствующие требованиям современного пользователя.

В процессе работы над выпускной квалификационной работой были решены следующие задачи:

Произведен анализ и дана характеристика существующей распределенной информационной системы, представляющую из себя группу тесно интегрированных сервисов, являющихся приложениями, построенными по принципам монолитной архитектуры. Описаны преимущества и недостатки монолитной архитектуры.

Рассмотрен альтернативный вариант проектирования приложений, описаны основные принципы построения реактивных систем, приведены преимущества микросервисов над монолитами.

Подробно разобраны основные требования к разрабатываемому микросервису. Построена архитектура разрабатываемого микросервиса на основе шаблона источника событий и фреймворка Lagom, служащего для построения реактивных микросервисов. Решена проблема передачи данных между сервисами экосистемы, в результате сравнительного анализа в качестве инструмента для доставки сообщений был выбран брокер сообщений Apache Kafka, с помощью которого разработанный микросервис был интегрирован в существующую информационную распределенную систему. Были описаны алгоритмы, по которым обрабатываются полученные сервисом команды. Представлены основные команды и события разрабатываемого сервиса, диаграмма классов-команд, архитектура распределенной информационной системы с интегрированным в нее микросервисом.

Основываясь на требованиях к данной выпускной квалификационной работе, был разработан гибкий, отказоустойчивый, горизонтально масштабируемый, реактивный микросервис, выполняющий функцию создания, изменения и закрытия договоров компании ООО «Квартплата 24» с организациями клиентами и платежными системами. Языком программирования для реализации кодовой базы микросервиса был выбран Scala.

Выполнено тестирование основного функционала разработанного микросервиса, а именно создание договора, изменение договора.

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

Список используемых источников

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. Фаулер М. Рефакторинг. Улучшение существующего кода // М. Фаулер. – СПб : Символ Плюс, 2015. – 415с.
6. Шаблон источников событий [Электронный ресурс]. URL: https://docs.microsoft.com/ru-RU/azure/architecture/patterns/event-sourcing (дата обращения: 15.04.2022)
7. Шаблон материализованного представления [Электронный ресурс]. URL: https://docs.microsoft.com/ru-RU/azure/architecture/patterns/materialized-view (дата обращения: 16.04.2022)
8. Шаблон CQRS [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/azure/architecture/patterns/cqrs (дата обращения 15.04.2022)
9. Akka Documentation [Электронный ресурс]. URL: https://doc.akka.io/docs/akka/current/typed/actors.html (дата обращения: 10.02.2022)
10. Apache Kafka Introduction [Электронный ресурс]. URL: https://kafka.apache.org/intro (дата обращения 08.03.2022)

11. Baesens, B. Beginning Java Programming: The Object-Oriented Approach / B. Baesens, A. Backiel, S. Vanden Broucke. – 1st edition, Wrox, 2015.

12. CQRS, Event Sourcing and DDD FAQ [Электронный ресурс]. URL: https://cqrs.nu/Faq (дата обращения: 15.04.2022)

13. Deitel, H. Java How to Program / H. Deitel, P. Deitel. – 9th edition, Prentice Hall, 2015.

14. Duncan C. E. Winn. Cloud Foundry: The Definitive Guide: Develop, Deploy, and Scale 1st Edition, Kindle Edition. C. : O'Reilly Media, 2017. 478 c.

15. Functional Programming For The Rest of Us [Электронный ресурс]. – URL: https://www.defmacro.org/2006/06/19/fp.html (дата обращения: 02.02.2022)

16. Git Book [Электронный ресурс]. URL: https://git-scm.com/book/en/v2 (дата обращения: 02.02.2022)

17. Hudson O. Getting started with IntelliJ IDEA // O. Hudson, Birmingham: Packt Publishing, 2013. – 114p.

18. JavaFX Overview [Электронный ресурс]. URL: https://openjfx.io/javadoc/18/ (дата обращения: 15.04.2022)

19. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems Paperback. C. : O'Reilly Media, 2017. 616 c.

20. Krochmalski J. IntelliJ IDEA Essentials // J. Krochmalski. – Birmingham: Packt Publishing, 2014.-263p.

21. Lagom Documentation [Электронный ресурс]. URL: https://www.lagomframework.com/documentation/ (дата обращения: 15.04.2022)

22. PostgreSQL Documentation [Электронный ресурс]. URL: https://www.postgresql.org/docs/current/index.html (дата обращения: 10.04.2022)

23. RabbitMQ Tutorials [Электронный ресурс]. URL: https://www.rabbitmq.com/getstarted.html (дата обращения: 09.03.2022)

24. Scala Book [Электронный ресурс]. URL: https://docs.scala-lang.org/overviews/scala-book/introduction.html (дата обращения: 02.02.2022)

25. Tutorials Technology [Электронный ресурс]. URL: http://www.gwtproject.org/doc/latest/tutorial/index.html (дата обращения: 09.03.2022)

Приложение А

Классы, реализующие интерфейс Command

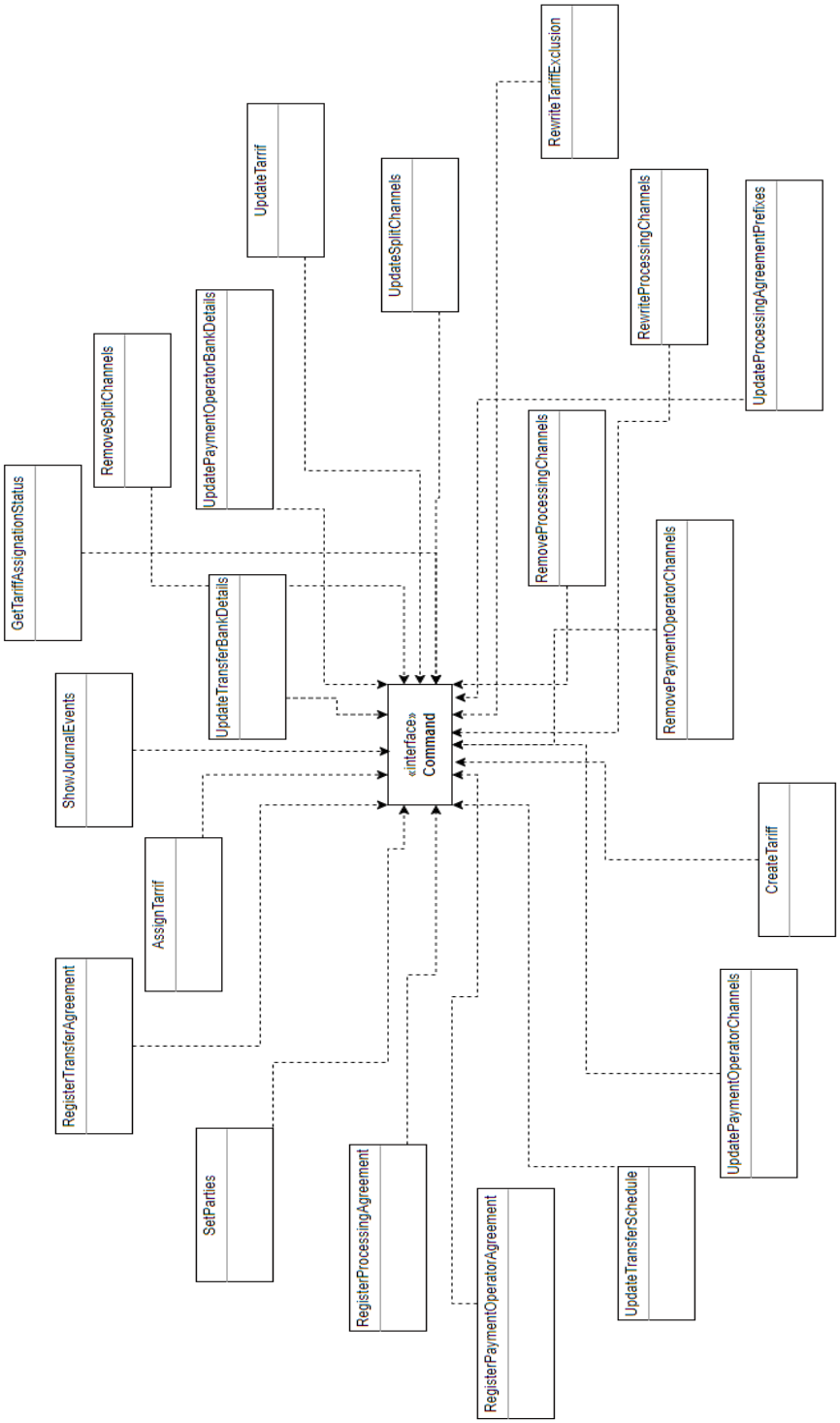

Рисунок А.1 – Классы, реализующие интерфейс Command

Приложение Б

Архитектура экосистемы сервисов

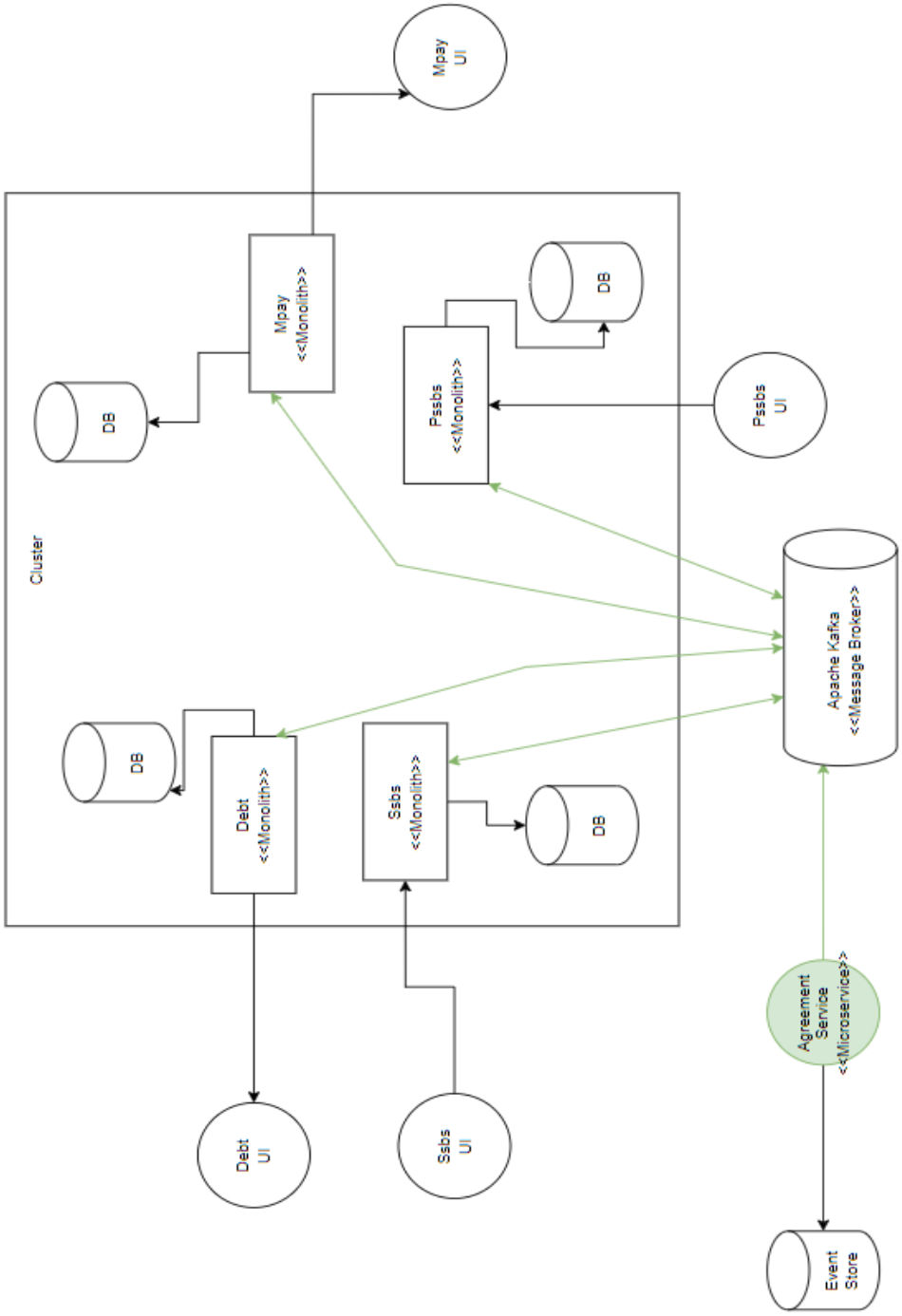

Рисунок Б.1 - Микросервис, интегрированный в экосистему сервисов

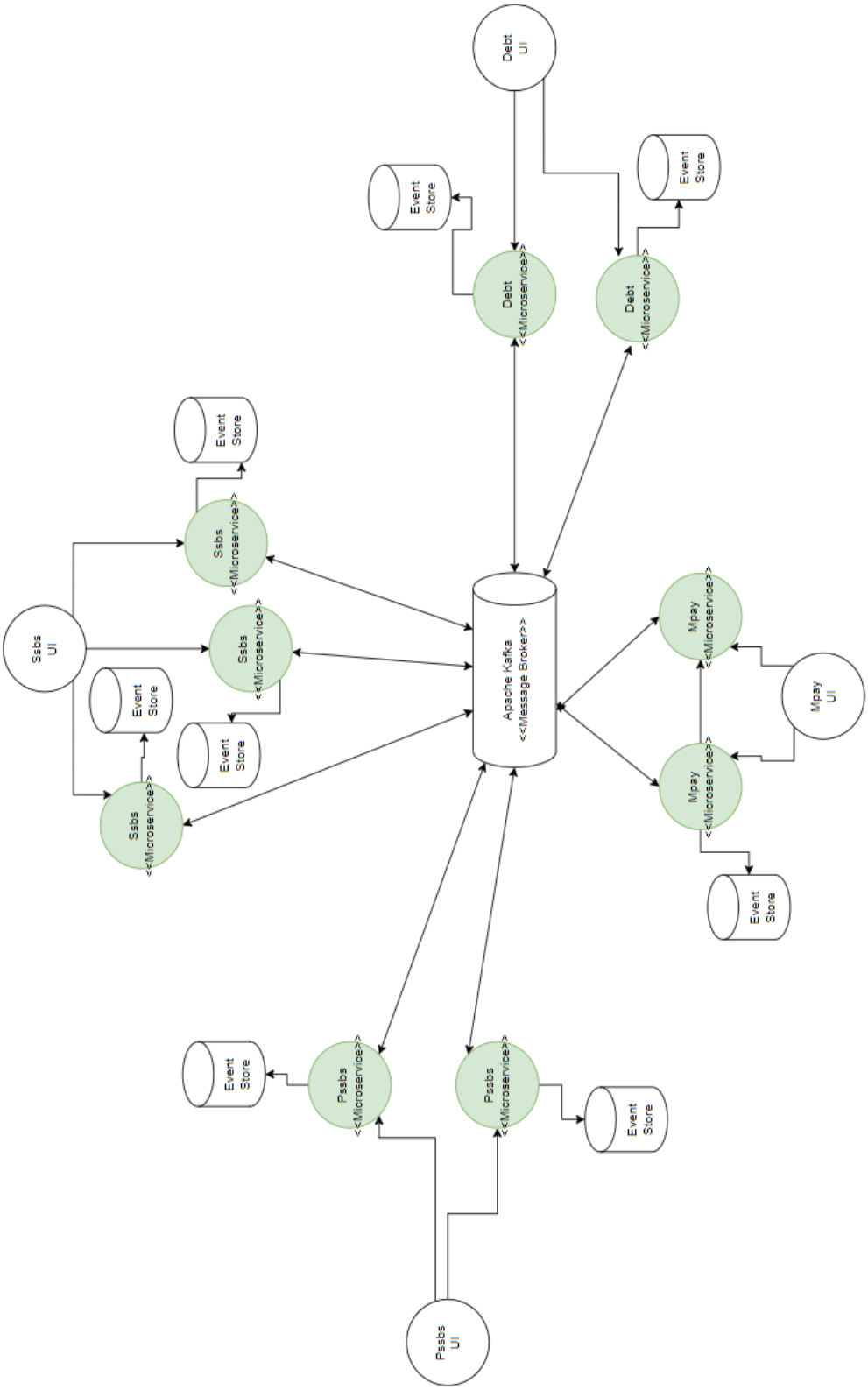

Рисунок Б.2 - Экосистема, полностью состоящая из микросервисов

Приложение В

Реализация команд, обрабатываемых микросервисом

```
object APICommand {
  sealed trait Command
  final case class RegisterPaymentOperatorAgreement(
     ba: BaseAgreementInfo,
     outChannels: immutable.Seq[SourceChannel],
     bankAccountId: Long
  ) extends Command
  final case class UpdatePaymentOperatorChannels(
     outChannels: immutable.Seq[SourceChannel]
  ) extends Command
  final case class RemovePaymentOperatorChannels(
     outChannels: immutable.Seq[ChannelId]
  ) extends Command
  final case class UpdatePaymentOperatorBankDetails(
     bankDetailsId: Long
  ) extends Command
  final case class RegisterProcessingAgreement(
     ba: BaseAgreementInfo,
     prefixes: Option[String],
     inChannels: Either[immutable.Seq[ProcessingChannel], TariffId],
     splitChannels: immutable.Seq[SplitChannel]
  ) extends Command
  final case class UpdateAgreementProperties(properties:
immutable.Seq[Property]) extends Command
  final case class RemoveAgreementProperties(keys: immutable.Seq[String])
extends Command
  final case class UpdateProcessingAgreementPrefixes(
```

```
prefixes: Option[String]
  ) extends Command
  final case class RewriteProcessingChannels(
     inChannels: immutable.Seq[ProcessingChannel],
     source: SourceOfChanges.Value = SourceOfChanges.Mpay
  ) extends Command
  final case class RemoveProcessingChannels(
     inChannels: immutable.Seq[ChannelId],
     source: SourceOfChanges.Value = SourceOfChanges.Mpay
  ) extends Command
  final case class UpdateSplitChannels(
     slitChannels: immutable.Seq[SplitChannel]
  ) extends Command
  final case class RemoveSplitChannels(
     slitChannels: immutable.Seq[AgreementId]
  ) extends Command
  final case class CreateTariff(
     tariff: Tariff
  ) extends Command
  final case class RegisterTransferAgreement(
     ba: BaseAgreementInfo,
     bankDetailsId: Long,
     minTransferAmount: Long,
     transferSchedule: String
  ) extends Command
  final case class UpdateTransferSchedule(
     minTransferAmount: Long,
     transferSchedule: String
```

```
    ) extends Command
      final case class UpdateTransferBankDetails(
        bankDetails: Long
      ) extends Command
      final case class SetParties(
        parties: Seq[PartyRef]
      ) extends Command
      final case class AssignTariff(
        tariffId: TariffId
      ) extends Command
      final case class RewriteTariffExclusion(
        ex: immutable.Seq[TariffExclusion]
      ) extends Command
      final case class UpdateTariff(
        tariffId: TariffId,
        tariffChannels: immutable.Seq[ProcessingChannel]
      ) extends Command
      case class GetTariffAssignationStatus(
        tariffId: TariffId
      ) extends Command
      case class ShowJournalEvents(fromOffset: Long, count: Long,
timeoutSeconds: Long = 60) extends Command
    }
```

Приложение Г

Реализация событий сервиса

```
sealed trait APIEvent {
  def id: String
}
final case class POAgreementCreated(
    _id: String,
    ba: BaseAgreementInfo,
    outChls: immutable.Seq[SourceChannel],
    bdId: Long
) extends APIEvent {
  override def id: String = _id
}
final case class SourceChannelsAdded(
    _id: String,
    newChs: immutable.Seq[SourceChannel]
) extends APIEvent {
  override def id: String = _id
}
final case class SourceChannelsUpdated(
    _id: String,
    updChs: immutable.Seq[SourceChannel]
) extends APIEvent {
  override def id: String = _id
}
final case class SourceChannelsRemoved(
    _id: String,
    rmdChs: immutable.Seq[ChannelId]
) extends APIEvent {
```

```
override def id: String = _id
}
final case class POBankUpdated(
    _id: String,
    bdId: Long
) extends APIEvent {
  override def id: String = _id
}
final case class TransferAgreementCreated(
    _id: String,
    ba: BaseAgreementInfo,
    bdId: Long,
    minTr: Long,
    trSchd: String
) extends APIEvent {
  override def id: String = _id
}
final case class ScheduleUpdated(
    _id: String,
    minTrAmt: Long,
    se: String
) extends APIEvent {
  override def id: String = _id
}
final case class TrBankUpdated(
    _id: String,
    bdId: Long
) extends APIEvent {
```

```
override def id: String = _id
}

final case class ProcessingAgreementCreated(
    _id: String,
    ba: BaseAgreementInfo,
    prefixes: Option[String],
    inChannels: immutable.Seq[ProcessingChannel],
    splitChannels: immutable.Seq[SplitChannel],
    tariff: Option[TariffDetails]
) extends APIEvent {
  override def id: String = _id
}
final case class ProcessingAgreementPrefixesUpdated(
    _id: String,
    prefixes: Option[String]
) extends APIEvent {
  override def id: String = _id
}
final case class AgreementPropertiesAdded(
    aggId: String,
    eventDate: LocalDateTime,
    properties: Seq[Property]
) extends APIEvent {
  override def id: String = aggId
}
final case class AgreementPropertiesUpdated(
    aggId: String,
```

```
eventDate: LocalDateTime,
    properties: Seq[Property]
) extends APIEvent {
  override def id: String = aggId
}

final case class AgreementPropertiesRemoved(
    aggId: String,
    eventDate: LocalDateTime,
    keys: Seq[String],
) extends APIEvent {
  override def id: String = aggId
}
final case class ProcessingChannelsUpdated(
    _id: String,
    added: Seq[ProcessingChannel],
    updated: Seq[ProcessingChannel],
    removed: Seq[String],
    source: SourceOfChanges.Value
) extends APIEvent {
  override def id: String = _id
}
final case class SplitChannelsAdded(
    _id: String,
    splitChs: immutable.Seq[SplitChannel]
) extends APIEvent {
  override def id: String = _id
}
```

```
final case class SplitChannelsUpdated(
    _id: String,
    splitChs: immutable.Seq[SplitChannel]
) extends APIEvent {
  override def id: String = _id
}

final case class SplitChannelsRemoved(
    _id: String,
    splitTrId: immutable.Seq[AgreementId]
) extends APIEvent {
  override def id: String = _id
}
final case class TariffAssigned(
    _id: String,
    tariff: TariffDetails
) extends APIEvent {
  override def id: String = _id
}
final case class TariffCreated(
    tariff: TariffDetails,
    processingChannels: Seq[ProcessingChannel]
) extends APIEvent {
  override def id: String = tariff.tariffId
}
final case class TariffUpdated(
    tariff: TariffDetails,
    added: Seq[ProcessingChannel],
```

```
    updated: Seq[ProcessingChannel],
       removed: Seq[String]
    ) extends APIEvent {
     override def id: String = tariff.tariffId
    }
    final case class TariffReset(
       _id: String
    ) extends APIEvent {
     override def id: String = _id
    }
    final case class PeriodEndUpdated(id: String, end: Option[LocalDateTime])
extends APIEvent
    final case class PartiesSet(
       _id: String,
       parties: Seq[PartyRef]
    ) extends APIEvent {
     override def id: String = _id
    }
    final case class TariffExclusionsUpdated(_id: String, exclusions:
Seq[TariffExclusionInfo]) extends APIEvent {
     override def id: String                              = _id
     implicit val tariffExclusionInfo: Format[TariffExclusionInfo] = Json.format
    }
```

Приложение Д

Фрагмент кода с методами и запросами сервиса

```
trait AgreementService extends Service {
  def registerPaymentOperatorAgreement(id: String):
ServiceCall[RegisterPaymentOperatorAgreement, Reply]
  def updatePaymentOperatorChannels(id: String):
ServiceCall[UpdatePaymentOperatorChannels, Reply]
  def removePaymentOperatorChannels(id: String):
ServiceCall[RemovePaymentOperatorChannels, Reply]
  final override def descriptor: Descriptor = {
    import APICommandJsonFormats._
    import APICommons._
    import APIEventJsonFormats._
    import Service._
    named("agrmt")
      .withCalls(
        restCall(Method.POST,"/agrmt/:id/crpoagr",
registerPaymentOperatorAgreement _),
        restCall(Method.POST, "/agrmt/:id/uppochs",
updatePaymentOperatorChannels _),
        restCall(Method.POST,"/agrmt/:id/rmpochs",
removePaymentOperatorChannels _),
        restCall(Method.GET, "agrmt/:id/rspch", reproduceSplitChannelsDev
_),
        restCall(Method.GET, "agrmt/allpa", getProcessingAgreements _),
      )
  }
}
```

Приложение Е

Фрагмент кода с реализацией методов API

```
class AgreementServiceImpl(
  clusterSharding: ClusterSharding,
  persistentEntityRegistry: PersistentEntityRegistry
)(implicit ec: ExecutionContext, repository: AgreementServiceRepository, materializer: Materializer)
  extends api.AgreementService {
 val log: Logger = LoggerFactory.getLogger(getClass)
 private def aggregateRef(id: String): EntityRef[AgreementCommand] =
  clusterSharding.entityRefFor(Agreement.typeKey, id)
 private def tariffRef(id: String): EntityRef[TariffCmd] =
  clusterSharding.entityRefFor(Tariff.typeKey, id)
 implicit val timeout: Timeout = Timeout(5.seconds)
 override def removeAgreementProperties(id: String): ServiceCall[APICommand.RemoveAgreementProperties, Reply] = {
  ServiceCall { prop =>
   if (prop.keys.nonEmpty)
    aggregateRef(id)
     .ask[Confirmation](reply => AgreementCommands.RemoveAgreementProperties(reply, prop.keys))
     .map(confirmation2reply)
   else Future.successful(api.Done(s"No keys found"))
  }
 }
 override def updateAgreementProperties(id: String): ServiceCall[UpdateAgreementProperties, Reply] = {
  ServiceCall { prop =>
   if (prop.properties.nonEmpty)
```

```
        aggregateRef(id)
              .ask[Confirmation](reply                                    =>
AgreementCommands.UpdateAgreementProperties(reply, prop.properties))
              .map(confirmation2reply)
            else Future.successful(api.Done(s”No props found”))
          }
        }
      }
```

Приложение Ж

Подключение базы данных к сервису

```
db.default {
  driver = “org.postgresql.Driver”
  url = “jdbc:postgresql://localhost/agreement”
  username = “postgres”
  password = “postgres”
   async-executor {
      queueSize = 10000
      numThreads = 2
      minConnections = 2
      maxConnections = 2
      registerMbeans = false
    }
}
```

Приложение И

Конфигурация брокера сообщений Apache Kafka

```
lagom.broker.kafka {
   service-name = “agreement”
   brokers = “127.0.0.1:9092”
}
}
```