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 модуля верификации данных информационной системы учета измерительных приборов при переходе на микросервисную архитектуру

Обучающийся П. О. Уланов
(Инициалы Фамилия) (личная подпись)

Руководитель канд.пед.наук, доцент, Е.А. Ерофеева
(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 модуля верификации данных информационной системы учета измерительных приборов при переходе на микросервисную архитектуру».

Актуальность работы заключается в необходимости проверки корректности данных при миграции от существующей архитектуры программного обеспечения к микросервисной.

Объектом исследования является информационная система учета измерительных приборов в ООО «Квартплата 24».

Предметом исследования является верификация данных с помощью отдельного микросервиса.

Цель выпускной квалификационной работы – разработка микросервиса верификации данных для перехода на микросервисную архитектуру.

Разработан модуль распределенной информационной системы, архитектура которого основана на принципах реактивной и микросервисной архитектур. Решена проблема верификации данных для перехода от устаревшей архитектуры программного обеспечения к микросервисной в рамках домена информационной системы учета измерительных приборов.

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

Выпускная квалификационная работа состоит из 50 страницы текста, 24 рисунков, 3 таблиц и 21 источника.

Abstract

The theme of the bachelor’s thesis is «Development of a module for verifying the data of a measuring instrument accounting system during the transition to a microservice architecture.»

The relevance of the work lies in the lack of validation of data during the transition from software architecture to microservice architecture.

The object of the study is the information system for accounting for measuring instruments at OOO Kvartplata 24.

The subject of the study is data verification using a separate microservice.

The purpose of the final qualification work is the development of a data verification microservice for the transition to a microservice architecture.

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 verification for the transition from an outdated software architecture to a microservice one within the domain of the information system for measuring instrument accounting has been solved.

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

The final qualifying work consists of 50 pages of text, 24 figures, 3 tables and a list of 25 references.

Оглавление

Введение……... 5

Глава 1 Характеристика организации. Анализ существующей системы сервисов…… 7

1.1 Характеристика организации……... 7

1.2 Анализ существующей экосистемы сервисов с точки зрения архитектуры…….. 8

1.3 Микросервисная и реактивная архитектура……... 13

1.4 Анализ информационной системы учета измерительных приборов…….. 16

1.5 Постановка задачи по разработке микросервиса……... 25

Глава 2 Проектирование микросервиса верификации…….. 27

2.1 Требования по разрабатываемому микросервису…….. 27

2.2 Проектирование архитектуры микросервиса верификации……. 28

2.3 Описание инструментов разработки микросервиса…….. 32

2.4 Разработка диаграммы классов микросервиса…….. 35

Глава 3 Разработка микросервиса верификации…… 39

3.1 Выбор средств для разработки микросервиса…… 39

3.2 Реализация основных модулей микросервиса…… 43

3.3 Тестирование разработанного микросервиса…… 46

Заключение……49

Список используемых источников…….. 51

Приложение А Код модуля для обобщенного сравнения данных……53

Приложение Б Код модуля для для сравнения представлений конкретно для Точки учета…… 56

Приложение В Код модуля взаимодействия с базой данных……... 60

Введение

Информационные технологии являются важным аспектом нашей жизни в современном мире. Они позволяют нам хранить, обрабатывать и передавать информацию с помощью различных устройств, таких как компьютеры, смартфоны и планшеты. Они также используются для поддержания связи, работы в онлайн-режиме, управления бизнес-процессами и решения различных задач.

Ввиду широкого распространения информационных технологий и доступа к ним, требуются механизмы проверки корректности, полноты и точности данных. Это важно, потому что неверные данные могут привести к неправильным решениям и ошибкам в работе системы. Одним из таких механизмов является процесс верификации данных.

Верификация данных - это процесс проверки данных различных типов по критериям поступление из доверенного источника, точность, согласованность и соответствие формату представления после выполнения операций миграции, трансформации и других операций с данными. Особую важность верификация данных приобретает в условиях автоматизированной обработки данных в информационных системах, внесении данных в базы данных и при применении технологий машинного обучения при подготовке обучающих и тестовых датасетов.

В качестве методов верификации можно применять проверку идентичности исходных и производных наборов данных путем побайтового сравнения, подсчета контрольных сумм, вычитки текстов и другие методы.

Верификация данных в микросервисной архитектуре важна, потому что компоненты-сервисы обычно разрабатываются, выпускаются и управляются независимо друг от друга. Это может привести к тому, что данные, передаваемые между ними, не будут соответствовать ожидаемым форматам или содержать недопустимые значения. Чтобы избежать этого, необходимо выполнять верификацию данных, чтобы убедиться, что они соответствуют

требованиям. Это может снизить риск нежелательных ошибок и улучшить надежность и производительность системы в целом. Верификация данных связана с валидацией данных, но между ними есть существенное отличие. В процессе верификации проверяется формальное соответствие заданным критериям результатов операций с данными, в то время как в процессе валидации проверяется корректность самого набора данных и применимость данных для решения конкретных вычислительных и других задач.

Актуальность работы заключается в необходимости проверки корректности данных при миграции от существующей архитектуры программного обеспечения к микросервисной.

Объектом исследования является информационная система учета измерительных приборов в ООО «Квартплата 24».

Предметом исследования является верификация данных с помощью отдельного микросервиса.

Цель выпускной квалификационной работы – разработка микросервиса верификации данных для перехода на микросервисную архитектуру.

Глава 1 Характеристика организации. Анализ существующей системы сервисов

1.1 Характеристика организации

ООО «Квартплата 24» - это российская компания в сфере информационных технологий, которая разрабатывает и поддерживает экосистему из более чем 10 облачных сервисов, взаимодействие которых тесно связано друг с другом. Эти облачные сервисы предназначены для решения задач, связанных с расчетом и учетом оплаты за ЖКХ, приемом платежей и их распределением, а также взысканием долгов в соответствии с законодательством Российской Федерации на всей ее территории [4] .

С 1996 года компания занимается разработкой и внедрением информационных систем для управления жилищно-коммунальным хозяйством. ООО «Квартплата 24» обслуживает более трех миллионов лицевых счетов для более чем 800 организаций, включая товарищества собственников жилья, сервисные и ресурсоснабжающие компании, а также управляющие компании в 67 регионах России [4] .

С помощью Квартплаты 24 можно произвести более 30 видов электронных платежей, которые принимаются как в России, так и за ее пределами. Этот сервис объединяет более 40 банков и платежных систем, что делает его удобным для пользователей. Кроме того, Квартплата 24 является единственным платежным сервисом в РФ, который позволяет мгновенно принимать и делить платежи за жилищно-коммунальные услуги на всех этапах прохождения платежа.

Структура организации представлена на рисунке №1.

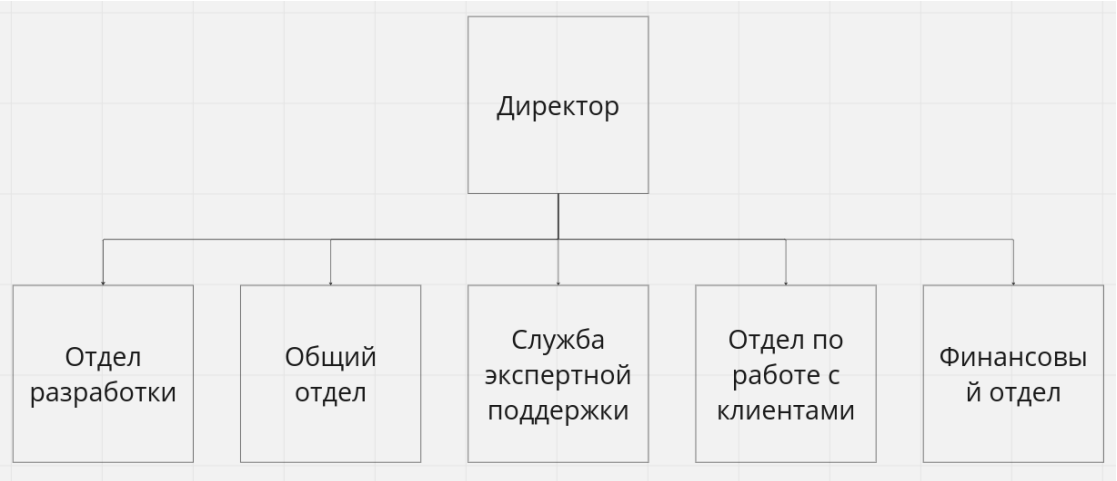

Рисунок 1 - Структура организации ООО «Квартплата 24»

Компания также является участником проекта «Сколково», резидентом технопарка «Жигулевская долина» в области высоких технологий и включена в каталог решений «Карта инновационных решений». Она также входит в «Банк решений Умного города» и «Банк эффективных технологий в ЖКХ», которые поддерживает Минстрой России.

1.2 Анализ существующей экосистемы сервисов с точки зрения архитектуры

Насчитывающая 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 [4] .

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 расчета (SSBS);
- сервис платежей (Мрау);

- личный кабинет жителя(PSSBS);
- сервис по работе с должниками(Debt).

Визуальное представление основных сервисов ООО «Квартплата 24» представлено на рисунке №2.

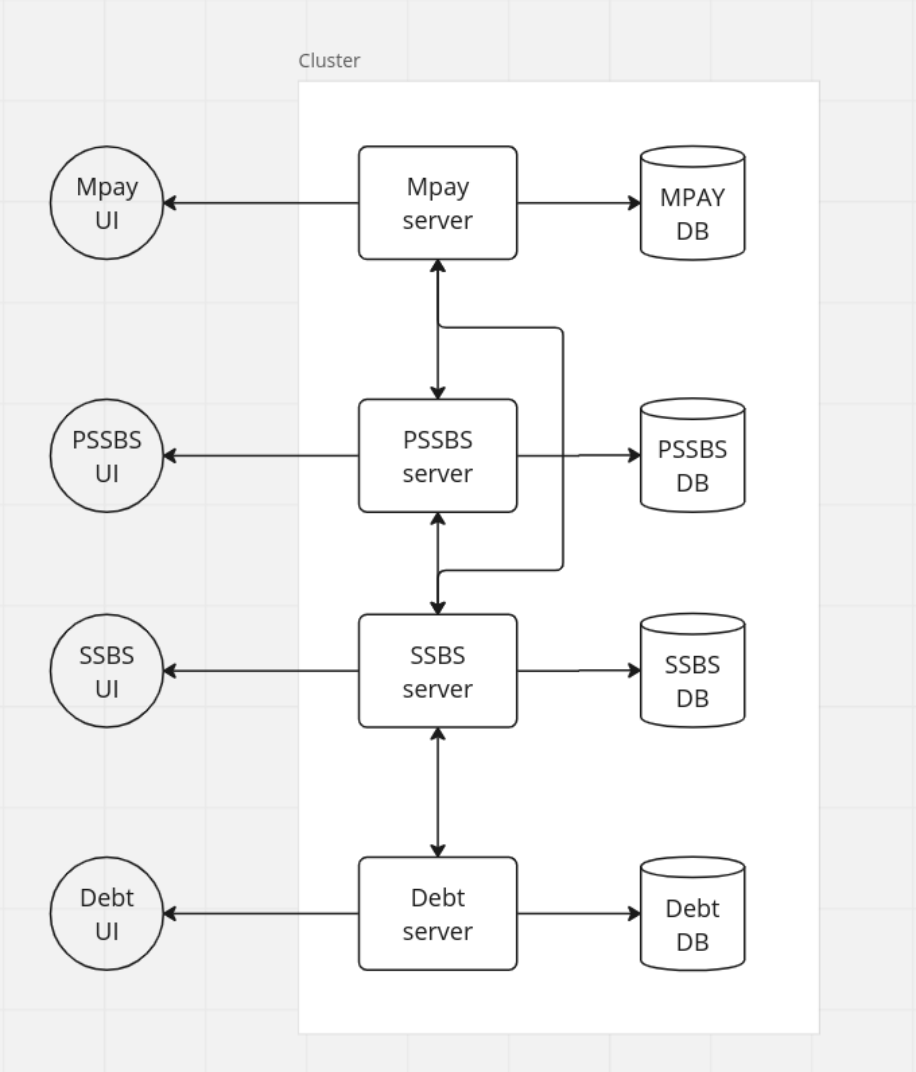

Рисунок 2 – Основные сервисы ООО «Квартплата 24»

Сервисы, обеспечивающие интеграцию с внешними системами:

- взаимодействие с операторами фискальных данных (Фискализация платежей);
- обмен данными с ОСЗН;
- обмен данными с ГИС ЖКХ;

- интеграция с телеметрическими системами;
- интеграция с внешними облачными сервисами и системами.

Визуальное представление экосистемы сервисов ООО «Квартплата 24» представлено на рисунке 3.

Рисунок 3 – Экосистема сервисов 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 роста и дальнейшего развития разрабатываемого приложения, будет появляться все больше трудностей в разработке и сопровождении данного программного обеспечения. 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 4.

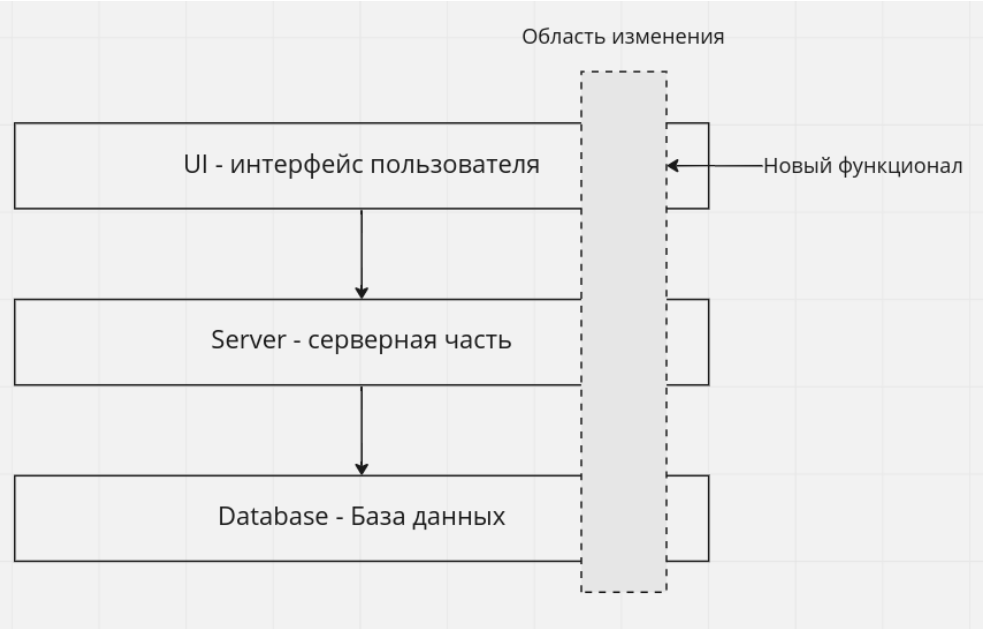

Рисунок 4 - Добавление нового функционала в монолитное приложение

Ключевой проблемой монолитной архитектуры является большое количество зависимостей, которые добавляются в приложение по мере разработки в нем новых модулей. Прекращение поддержки фреймворка или библиотеки, уже использующиеся в нем через зависимость, принесет колоссальные трудности в последующей разработке и поддержке существующего кода. Подобного рода проблема повлечет за собой трудоемкую и долгую переработку затронутых модулей, что в дальнейшем может вылиться в переработку всего монолита. Вариант запутанного монолита представлен на рисунке №5.

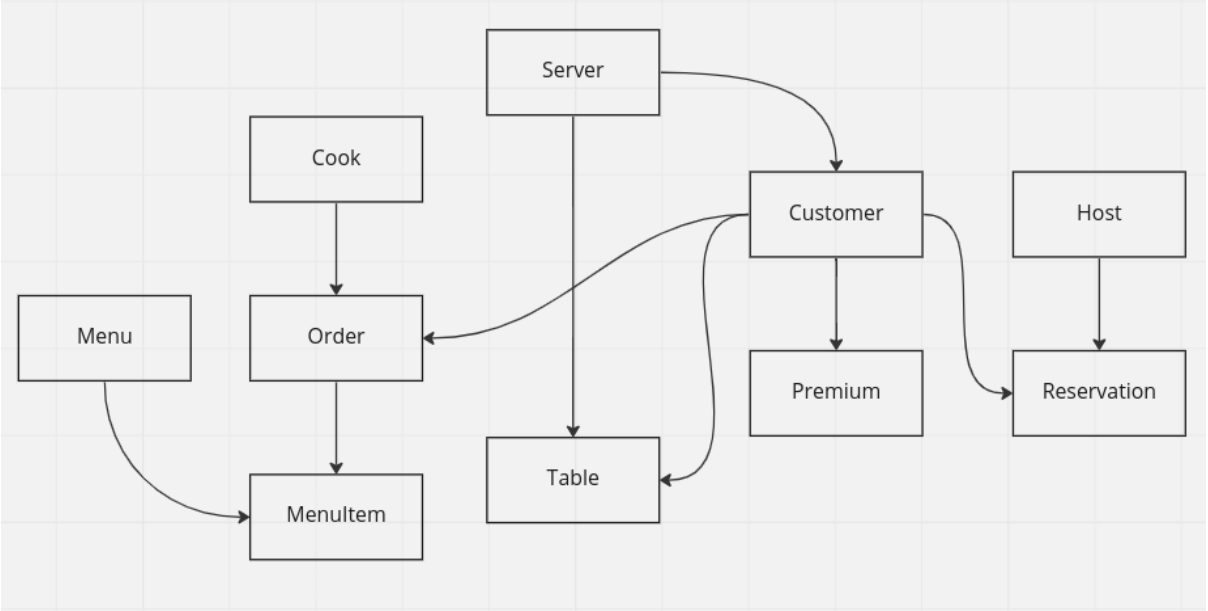

Рисунок 5 - Запутанный монолит

По мере обрастания запутанного приложения основанного на монолитной архитектуре его улучшение, поддержка или дальнейшая разработка становится все сложнее и сложнее, затрудняя также привлечение новых разработчиков.

1.3 Микросервисная и реактивная архитектура

Альтернативой монолитной архитектуры и её вариациям, набирая всё большую популярность, выступает микросервисная и реактивная архитектура.

Системы, конструируемые с использованием паттернов и принципов реактивного подхода к архитектуре, обеспечивают высокую гибкость и способность к масштабированию, а также отличаются высокой отказоустойчивостью. Основой всех вышеперечисленных преимуществ по сравнению с другими подходами к разработке программного обеспечения лежит обмен данными с использованием неблокирующих сообщений.

Обмен данными, основой которого является отсутствие блокировки обмена сообщениями, гарантирует слабую связанность между элементами информационной системы. Сообщения такого рода позволяют организовать регулирование нагрузки, отслеживание так называемых очередей сообщений с использованием различных инструментов для мониторинга и обеспечивает обратное давление, если потребитель получает и обрабатывает данные медленнее, чем производитель их поставляет.

Обмен данными, основанный на средствах, предоставляющих неблокирующий обмен сообщениями, дает возможность для реализации гибких и устойчивых к нагрузкам информационных систем. Гибкость подразумевает под собой способность системы реагировать на степень нагрузки, регулируя используемый объем вычислительных мощностей. Устойчивость же достигается за счет изолирования подсистем в рамках одной информационной системы, различные элементы такой системы имеют возможность прекращать свою работу и восстанавливаться самостоятельно, никаким образом не влияя на другие части этой системы. Это означает, что возникновение ошибки в одной части информационной системы никак не

сможет повлиять на работоспособность системы в целом, неработоспособным будет только один её модуль.

Построение информационных систем с соблюдением принципов и подходов микросервисно-реактивной архитектуры позволяет разрабатывать отзывчивые системы с минимальным временем отклика при пользовательском запросе. Подобного рода системы упрощают разработку, обработку и исправление ошибок, повышают положительный опыт использования системы пользователем и способствует быстрому развитию программного обеспечения на фоне конкурентов на рынке. Принципы, лежащие в основе микросервисно-реактивной архитектуры изображены на рисунке №6

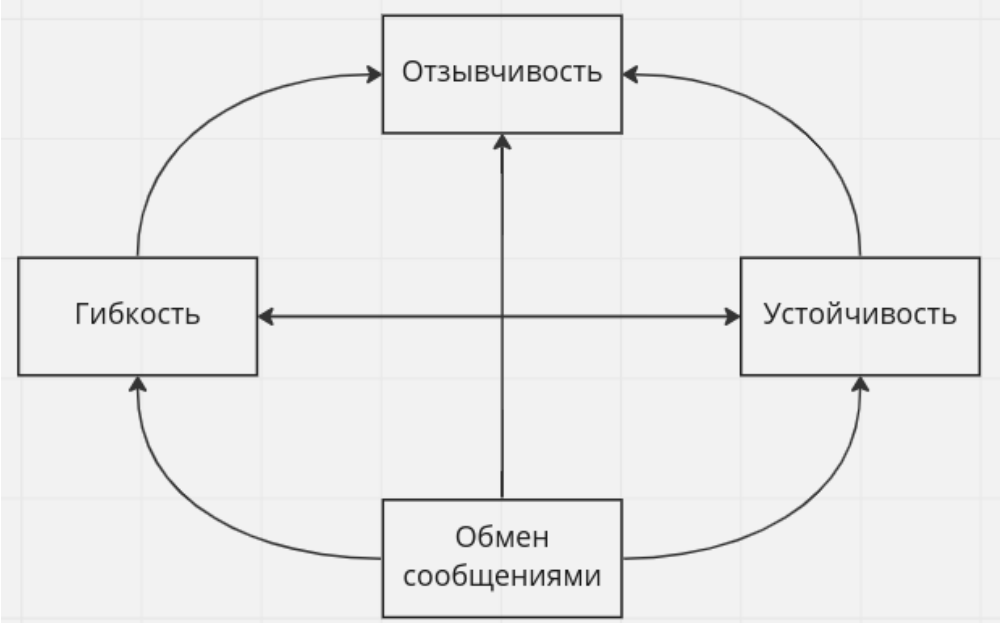

Рисунок 6 - Принципы микросервисно-реактивной архитектуры

Микросервисы - это программное обеспечение, представленное в виде отдельных сервисов, имеющих возможность развертываться независимо друг от друга. В связке группа микросервисов образует распределенную информационную систему. Каждый микросервис имеет контроль над собственными данными, из этого следует, что сервисы информационной системы не должны иметь какие-либо базы данных для совместного

использования [7] . Подобного рода сокрытие данных внутри отдельных сервисов обеспечивает уменьшение связанности и позволяет использовать конкретные интерфейсы для обмена данными. Исходя из выполняемых задач, у микросервиса вполне может отсутствовать база данных как таковая.

Архитектурное решение подобного рода позволит распределить разработку между отдельными командами разработчиков, что позволит им иметь ограниченный контекст, что в свою очередь положительно скажется на их понимании той части системы, которую им придется разрабатывать. Независимое развертывание дает широкие возможности для масштабирования информационной системы и убирает ограничение на использование различного рода технологий. Пример такого микросервисного приложения изображен на рисунке №7.

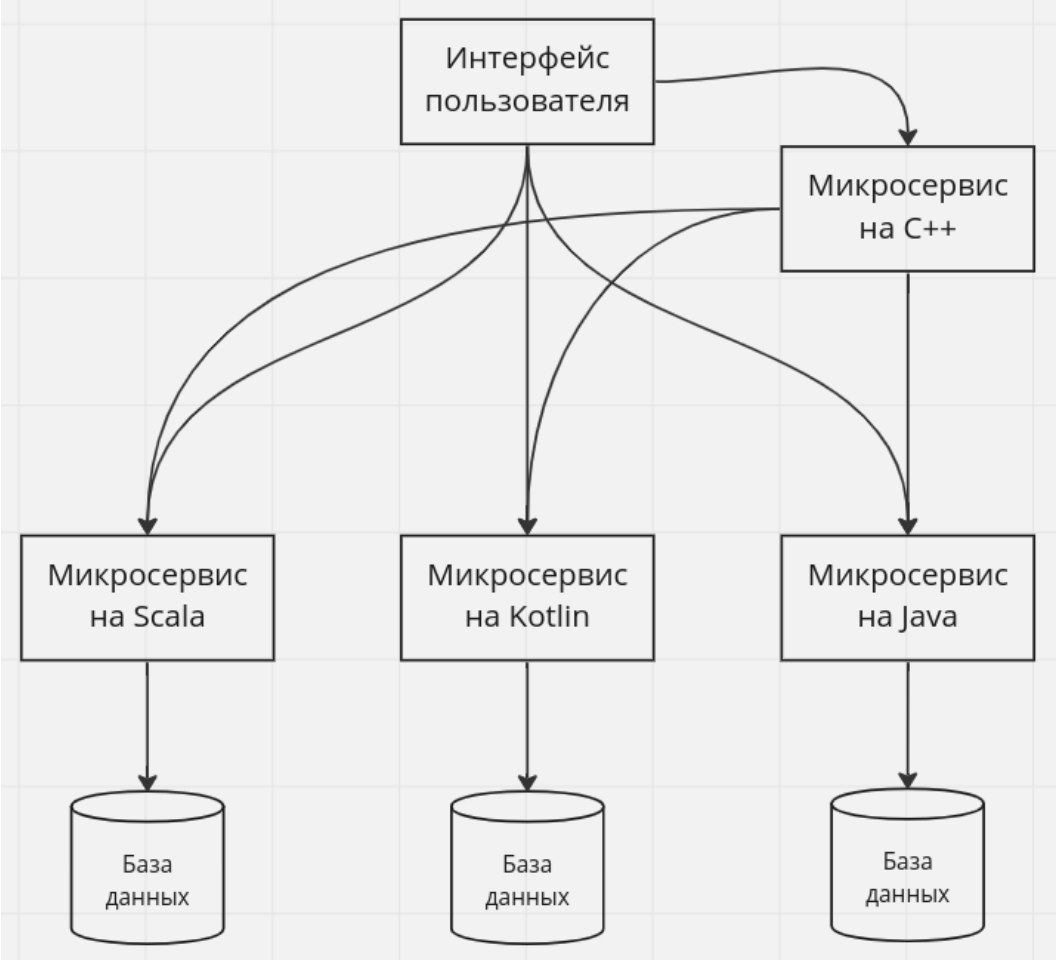

Рисунок 7 - Микросервисное приложение с модулями написанными с использованием разных языков программирования

Микросервисы внутри одной распределенной информационной системы могут быть разработаны с использованием разных языков программирования с соответствующими для них актуальными фреймворками и библиотеками.

1.4 Анализ информационной системы учета измерительных приборов

Информационная система учета измерительных приборов является частью сервиса расчета, одного из самых больших монолитов экосистемы ООО “Квартплата 24”. В функционал данной системы входит:

- учет показаний по шкалам приборам учета;
- место установки приборов учета;
- процесс установки нового прибора учет;
- процесс снятия прибора учета;
- процесс изменения прибора учета;
- валидация входящих показаний.

Компонентами данной информационной системы в сервисе расчета являются:

Точка учета - является центральной сущностью домена связанного с приборами учета. В представлении сервиса расчета является темпоральной, то есть имеет дату начала и окончание действия. В ней находятся вводы на которые происходит установка приборов учета, также указан код услуги, он является определяющим критерием для списка групп услуг, в рамках которых на данную точку учета могут быть установлены приборы учета. В базе данных представлена таблицей metering_point, параметрами которой являются:

- id - уникальный идентификатор, является первичным ключом;
- aggregate_root_id - уникальный идентификатор агрегата;
- classifier - код услуги, является ограничением для услуг, в рамках которого на данную точку учета могут быть установлены приборы учета;
- habitable_fk - вторичный ключ, ссылка на жилье, в котором установлена точка учета;
- aggregate_created - поле, отображающие создан ли агрегат в микросервисе точки учета;
- period_date_start - дата начала действия точки учета;
- period_date_end - дата окончания действия точки учета;
- date_create - дата создания точки учета;
- date_change - дата изменения чего-либо в точке учета.

Визуально таблица точки учета представлена на рисунке №8.

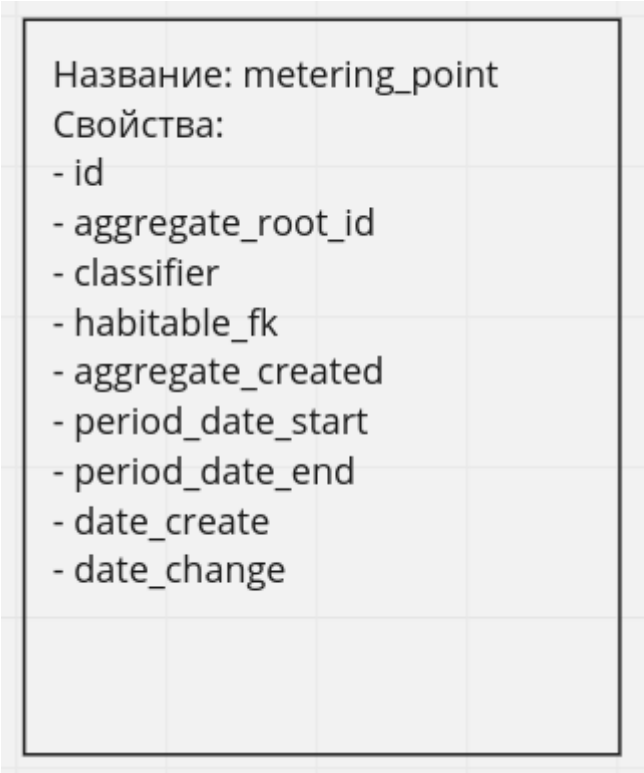

Рисунок 8 - Визуальное представление таблицы точки учета

Ввод - место установки определенного прибора учета. Находится в точке учета, имеет дату начала и окончания действия, в рамках которого на него можно установить прибор учета. Так же имеет порядковый номер, который вычисляется исходя из количества других вводов, уже созданных на точке учета. В базе данных представлен таблицей inlet, параметрами которой являются:

- id - уникальный идентификатор, является первичным ключом;
- description - описание, задается пользователем;
- period_date_start - дата начала действия точки учета;
- period_date_end - дата окончания действия точки учета;
- metering_point_fk - вторичный ключ, ссылка на точку учета;
- number - номер ввода, вычисляется автоматически, исходя из уже имеющхся на точке учат вводах.

Визуально таблица ввода представлена на рисунке №9.

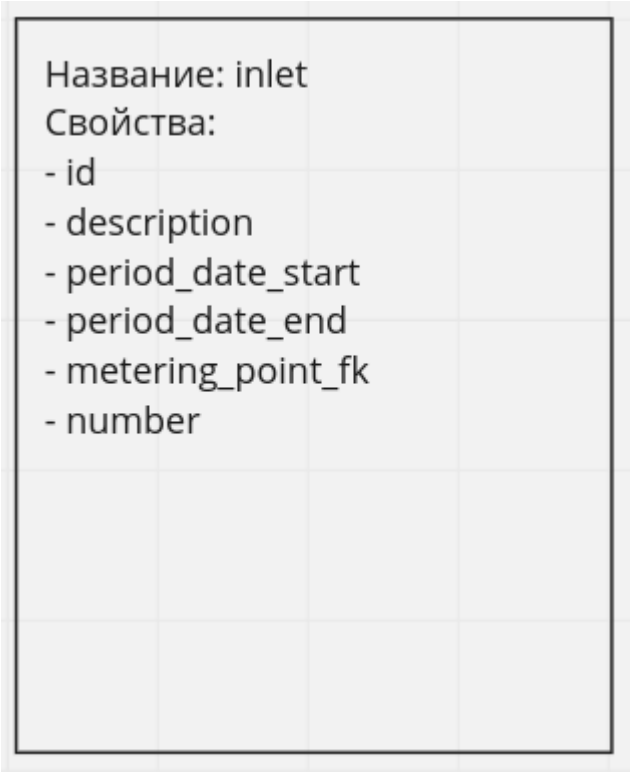

Рисунок 9 - Визуальное представление таблицы ввода

Прибор учета - это счетчик, у которого указывается серийный номер, задается определенный тип прибора, от которого зависит ресурс, для измерения которого он предназначен. В базе данных представлен таблицей measure_unit, параметрами которой являются:

- id - уникальный идентификатор, является первичным ключом;
- serial_number - серийный номер прибора учета;
- description - описание, задается пользователем;
- device_type - тип прибора учета, задается исходя из ресурса под который рассчитан прибор учета;
- old_code - старый номер;
- accuracy - точность, количество знаков после запятой;
- scale - размер шкалы;
- verified_until - дата поверки прибора учета;
- original_verified_until - оригинальная дата поверки прибора учета.

Визуально таблица прибора учета представлена на рисунке №10.

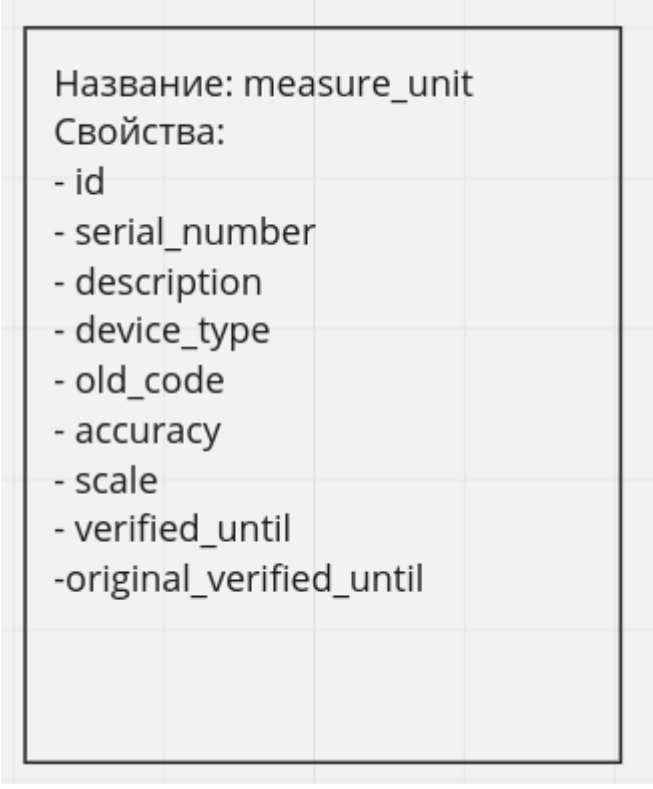

Рисунок 10 - Визуальное представление таблицы прибора учета

Прибор учета в жилье - это счетчик, измеряющий расход определенного ресурса, установленный в квартире, в офисе или нежилом помещении. В информационной системе сервиса расчета является представлением шкалы определенного прибора учета, то есть если прибор учета, установленный в жилье является двухтарифным, принимающим показания по электричеству раздельно за день и за ночь, то в базе будет создано две сущности прибора учета в жилье, имеющих ссылку на один прибор учета, одинаковый период действия, ссылку на один ввод и одну точку учета, но имеющие ссылки на разные услуги, соответственно для дня и ночи. Он имеет период действия, в рамках которого на данный прибор учета в жилье регистрируются показания и который должен быть в рамках периода действия ввода, на который устанавливается прибор учета в жилье. Нельзя создать прибор учета в жилье без начального показания. В базе данных представлен таблицей measure_unit_in_habitable, параметрами которой являются:

- id - уникальный идентификатор, является первичным ключом;
- period_date_start - дата начала действия прибора учета;
- period_date_end - дата снятия прибора учета;
- date_create - дата создания прибора учета;
- date_change - дата какого либо изменения, затрагивающее данный прибор учета;
- measure_unit - вторичный ключ, ссылка на прибор учета;
- facility_group_fk - вторичный ключ, ссылка на группу услуг;
- metering_point_fk - вторичный ключ, ссылка на точку учета;
- description - описание прибора учета;
- inlet_fk - вторичный ключ, ссылка на ввод.

Визуально таблица прибора учета представлена на рисунке №11.

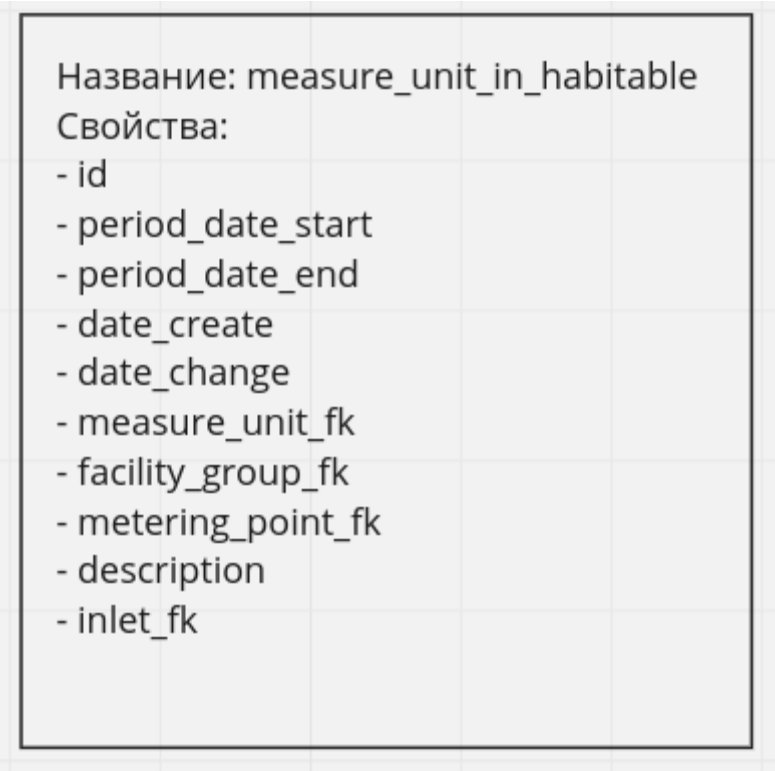

Рисунок 11 - Визуальное представление таблицы прибора учета в жилье

Показания приборов учета - показания, которые регистрируются на конкретный прибор учета по конкретной услуге. Они необходимы для ежемесячного расчета начислений по конкретному контракту, исходя из правила расчета и тарифа установленных на услугу. В базе данных представлен таблицей measure_unit_in_habitable, параметрами которой являются:

- id - уникальный идентификатор, является первичным ключом;
- date - дата регистрации показания;
- date_create - дата создания прибора учета;
- date_change - дата какого либо изменения, затрагивающее данный прибор учета;
- indications - показание;
- measure_unit_in_habitable_fk - вторичный ключ, ссылка на прибор учета в жилье;
- facility_group_fk - вторичный ключ, ссылка на группу услуг;

- changed_by - поле, показывающие, кем внесено показание в информационную систему;
- description - описание прибора учета;
- from_organisation - поле, показывающие, что показание внесено управляющей компанией, а не жителем;
- accounting_month - месяц расчета.

Визуально таблица показаний прибора учета представлена на рисунке №12.

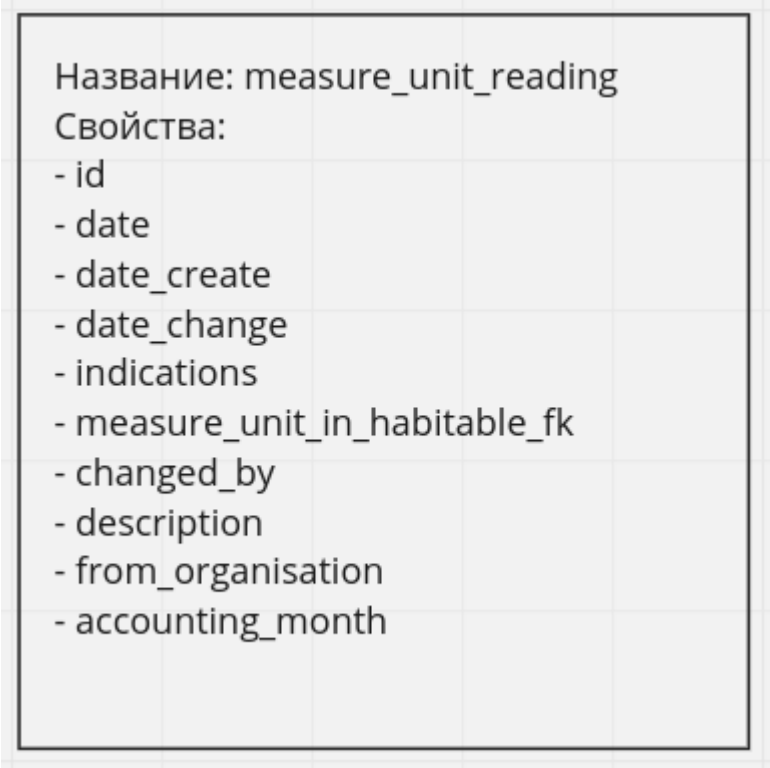

Рисунок 12 - Визуальное представление таблицы показаний приборов учета

Версия экосистемы ООО “Квартплата 24” в части информационной системы измерительных приборов при отсутствии микросервиса Точки Учета являлась крайне зависимой от работоспособности сервиса расчета, так как в заинтересованные сервисы данные отправлялись через синхронизацию, в которой сервис расчета принимал непосредственное участие. Любое создание новой сущности или изменение уже существующей в части домена

информационной системы измерительных приборов порождало задачи на синхронизацию данных между личным кабинетом, сервисом интеграции с ГИС ЖКХ и сервисом выбора протокола для оплаты. Визуально циркуляция потоков данных между сервисом расчета и сервисом личного кабинета жителя через обработчик очереди сообщений без микросервиса Точки Учета представлена на рисунке №13.

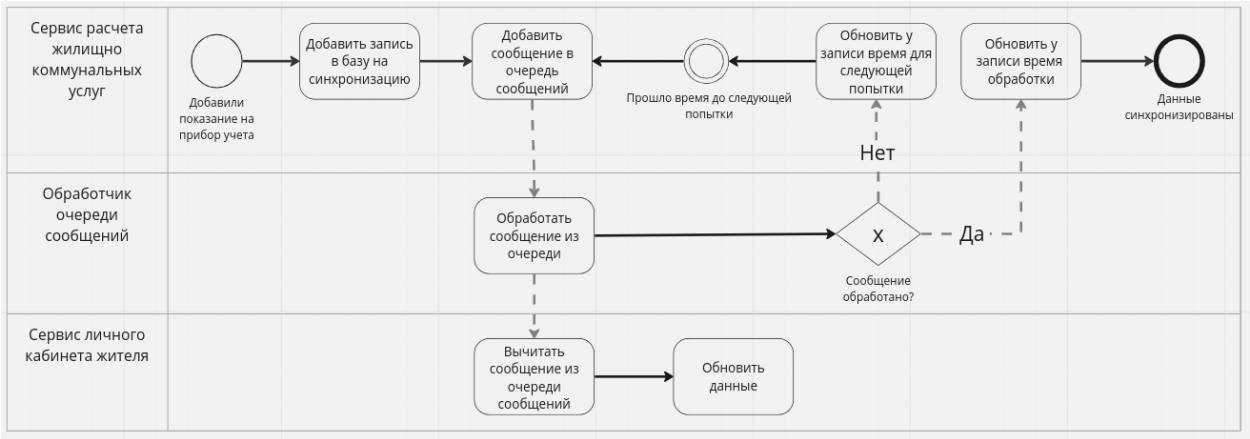

Рисунок 13 - Циркуляция потоков данных между сервисом расчета и сервисом личного кабинета жителя через обработчик очереди сообщений без микросервиса Точки Учета

При наличии микросервиса Точки Учета задача распространения данных для заинтересованных сервисов делегируется микросервису Точки Учета. И в таком случае задачей сервиса расчета будет являться использование ограниченного набора команд, предоставляемых публичным API микросервиса Точки Учета. В свою же очередь Точка Учета в процессе обработки команд будет порождать события, которые будут распространяться в заинтересованные сервисы, обновляя локальное представление, находящееся в них. Исходя из этого и вышеописанных преимуществ микросервисной архитектуры программного обеспечения возрастет надежность и отзывчивость экосистемы в целом. Визуально циркуляция потоков данных между сервисом расчета и сервисом личного кабинета жителя

через обработчик очереди сообщений с микросервисом Точка Учета представлена на рисунке №14.

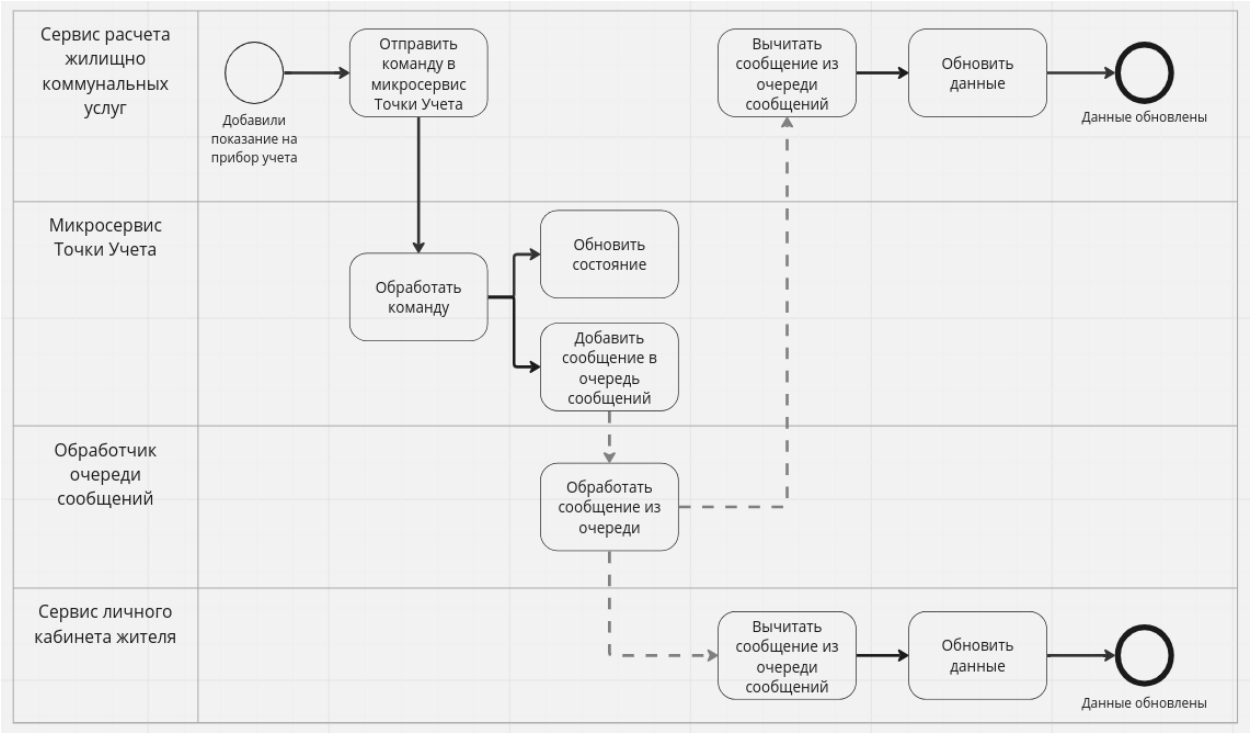

Рисунок 14 - Циркуляция потоков данных между сервисом расчета и сервисом личного кабинета жителя через обработчик очереди сообщений с микросервисом Точка Учета

Беря во внимание все вышеописанное в данной главе, руководством компании совместно со старшими разработчиками было принято решение начать переход на микросервисную архитектуру в разрезе информационной системы учета измерительных приборов. Ввиду того, что домен информационной системы учета измерительных приборов достаточно обширно представлен в сервисе расчета, для более стабильного перехода от уже существующего варианта информационной системы к варианту ориентированному на микросервисную архитектуру потребовался микросервис верификации данных.

1.5 Постановка задачи по разработке микросервиса

Принимая в расчет описанные выше особенности микросервисной архитектуры и её преимущества над альтернативными вариантами архитектур было принято решение начать переработку обозначенного выше монолита в экосистеме ООО “Квартплата 24” - сервиса расчета, с последующим разделением его доменных частей на отдельные микросервисы.

Первым на очереди из таких микросервисов является сервис Точки Учета, который должен отвечать за хранение данных о приборах учета, вводах на которых эти приборы учета установлены, показания по приборам учета, а также распространение необходимой информации для таких сервисов как: сервис расчета, личный кабинет и ГИС ЖКХ.

Основной целью данной выпускной квалификационной работы является разработка микросервиса верификации данных при выделении из сервиса расчета домена для микросервиса Точки Учета, а также для корректного перехода на работу с микросервисом Точки Учета.

Основным функционалом разрабатываемого микросервиса является проверка данных на корректность находящихся в базе данных сервиса расчета и базе данных Точки Учета.

Для успешного выполнения данной квалификационной работы необходимо:

- спроектировать микросервисное приложения для верификации данных;
- описать инструменты для реализации микросервиса, среды разработки и языка программирования;
- разработать способ верификации данных;
- разработать диаграмму классов микросервиса;
- разработать микросервис;
- протестировать его работоспособность.

В данной главе были рассмотрены различия монолитной и микросервисных архитектур, какие они имеют положительные и отрицательные стороны.

Был проведен анализ существующей экосистемы сервисов и информационной системы учета измерительных приборов ООО «Квартплата 24», описаны существующие элементы системы и таблицы базы данных, отвечающие за хранение данных по части системы учета измерительных приборов, обоснована потребность наличия микросервиса верификации данных.

Также была поставлена задача на разработку микросервиса верификации данных для корректного выделения из существующего монолита определенной его части в отдельный микросервис.

Глава 2 Проектирование микросервиса верификации

2.1 Требования по разрабатываемому микросервису

Исходя из описанных в первой главе недостатков монолитной архитектуры и преимуществ микросервисной архитектуры над монолитной, была начата переработка обозначенного в первой главе монолита экосистемы ООО “Квартплата 24” - сервиса расчета, с последующим разделением его доменных частей на отдельные микросервисы.

Основным функционалом микросервиса, разрабатываемого в ходе данной выпускной квалификационной работы, является проверка корректности данных находящихся в базе данных сервиса расчета и базе данных Точки Учета. Беря в расчет микросервисный подход к разработке программного обеспечения, разрабатываемый микросервис должен быть сосредоточен на выполнении одной бизнес-задачи, чтобы предотвратить в дальнейшем разрастание его функционала и обязанностей до монолита. Бизнес-задачей микросервиса верификации является только проверка объекта имеющего представление как в базе данных Точки Учета, так и в базе данных сервиса расчета.

Функциональными требованиями данного микросервиса являются запрос данных из микросервиса Точки Учета и сервиса расчета по конкретной Точке учета, их сравнение, нахождение расхождений между двумя объектами по определенным правилам, сохранение результата сравнения в базу. Разрабатываемый микросервис должен удовлетворять требованиям построения архитектуры с использованием микросервисного подхода, для этого было решено разрабатывать его с помощью фреймворка Lightband Lagom.

К нефункциональным требованиям данного микросервиса можно отнести его отказоустойчивость, возможность к масштабированию и гибкость по отношению к нагрузкам.

2.2 Проектирование архитектуры микросервиса верификации

Информационные системы, проектируемые с использованием классических подходов к работе с данными, например, такими как CRUD, имеют определенные недостатки, являющиеся критичными при разработке масштабируемого и отказоустойчивого программного обеспечения, основанного на микросервисной архитектуре.

CRUD:

- create - создание;
- read- чтение;
- update- обновление;
- delete - удаление.

Одними из таких недостатков являются:

- если в программным обеспечении одновременно работает большое количество пользователей, то велика вероятность возникновения конфликтов при изменении одной и той же сущности в базе данных;
- в базе данных хранится только итоговый вариант сущности, которую как либо модифицировали. Что в свою очередь ведет к тому. что если произойдет какая-либо ошибка при работе с данными, то её практически невозможно будет вернуть в состояние до возникновения ошибки;
- в подобном программном обеспечении работа ведется с сущностями и напрямую с базами данных, что в свою очередь сказывается на отзывчивости и производительности, а также может негативно повлиять на способность информационной системы к масштабированию.

Микросервис, разрабатываемый в рамках данной выпускной квалификационной работы, не должен иметь подобного рода недостатков, то есть должен иметь возможность хорошо масштабироваться, быть

отказоустойчивым и отзывчивым, что является основой подхода к построению микросервисной архитектуры.

Классический вариант шаблона, удовлетворяющего вышеописанным требованиям, представляет собой разделение кодовой базы на команды и события [17] . Командой является запрос, отправляемый напрямую сервису через его публичное API. Событие - это то, что порождает микросервис в результате обработки команды [10] .

Беря в расчет бизнес-задачу микросервиса верификации данных, ему не требуется реализация событий, порождаемых командами которые он обрабатывает. Таким образом из классического шаблона исключаются события и остаются только команды.

Основными командами разрабатываемого микросервиса будут являться: сравнение двух конкретных представлений Точки Учета, сравнение всех имеющихся представлений Точки Учета в конкретной схеме. Сравнение должно происходить по определенному набору правил, которые допустимо комбинировать в произвольном количестве.

Результатом выполнения данных команд будет являться запись в базе данных, которую потом можно будет просмотреть через вспомогательную команду для просмотра результата сравнения по уникальному идентификатору.

Также потребуется реализовать команду для отображения доступных команд, так как данный микросервис не подразумевает наличие у себя интерфейса пользователя.

Алгоритмы обработки команды на сравнение представлен на рисунке № 15

Рисунок 15 - Алгоритмы обработки команды на сравнение

Диаграмма вариантов использования микросервиса представлена на рисунке №16.

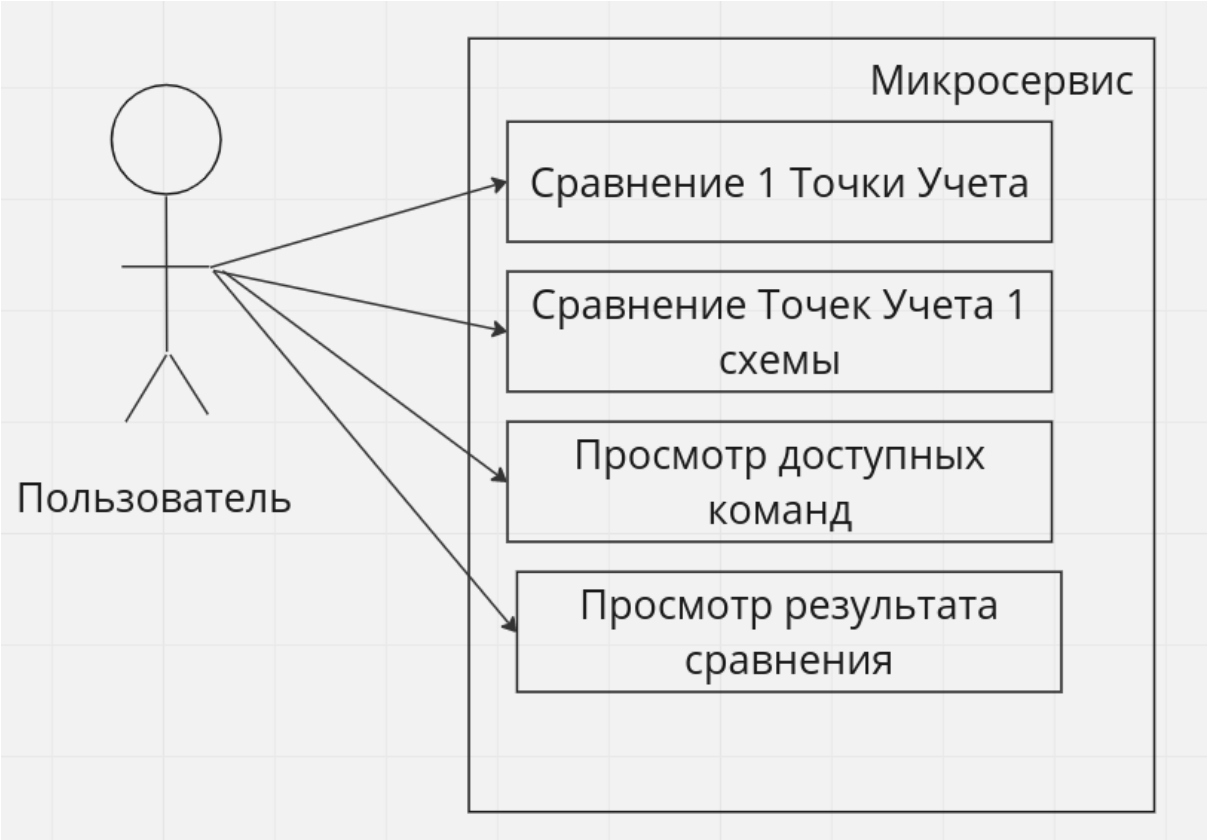

Рисунок 16 - Диаграмма вариантов использования микросервиса

Таким образом микросервис предназначен для сравнения одной точки учета, сравнения всех точек учета одной схемы. Также у пользователя есть возможность просмотра доступных команд и просмотр результата работы конкретной задачи.

2.3 Описание инструментов разработки микросервиса

Lagom Framework - инструмент, предназначенный для разработки приложений на основе микросервисной архитектуры. Это в свою очередь означает, что он основан на обмене данными через отправку неблокирующих сообщений, что позволяет создавать отзывчивые, гибкие и отказоустойчивые информационные системы [16].

Для удовлетворения принципов построения систем, основанных на неблокирующих операциях, в основе Lagom Framework лежит Akka Framework, который в свою очередь реализует Модель Акторов, описанную Карлом Хьитом, Ричардом Штайгером и Питером Бишопом в 1973 году. В модели акторов всё является актором, что делает данную модель похожей на объектно-ориентированную парадигму программирования, в которой всё является объектом.

Актор — это фундаментальная единица вычисления, реализующая обработку, хранение и коммуникацию, инкапсулирующая свое состояние и поведение. Модель акторов не говорит о том, как именно эта сущность должна быть реализована. Взаимодействие акторов в акторной модели происходит посредством неблокирующего обмена сообщениями. Получая какое-либо сообщения актор, реагируя на него, способен изменять свое состояние, отправлять сообщение другим акторам, создавать новые акторы или изменять свое поведение для обработки последующих полученных сообщений [7] . Вышеописанные действия способны выполняться параллельно. Обмен сообщениями подобного рода позволяет не думать о синхронизации и позволяет проектировать и разрабатывать системы, в основе которых лежат параллельные вычисления.

Каждому актору присваивается свой адрес, чем и гарантирована инкапсуляция состояние и поведения актора. Все взаимодействие с актором

происходит через присвоенную ему адрес. Зная его, например, можно отправить сообщение конкретному актору, независимо от того, находится ли он удаленно или локально. Элементами, связывающие акторы, являются их почтовые ящики. Акторы, целью которых является обработка сообщений, имеют только один почтовый ящик, в которых хранится очередь полученных актором сообщений. Гарантирована обработка сообщений в том порядке, в котором они попали в почтовый ящик. Обмен сообщениями между двумя акторами изображен на рисунке №17.

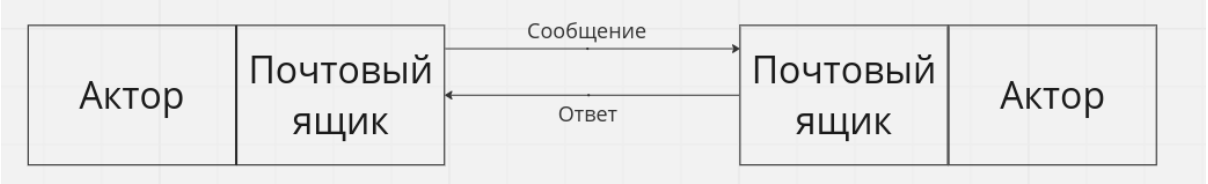

Рисунок 17 - Обмен сообщениями между двумя акторами

Актор может существовать только в рамках акторной системы. При создании акторной системы автоматически создаются корневые акторы, отвечающие за обработку ошибок своих дочерних акторов. Система акторов визуально представлена на рисунке №18.

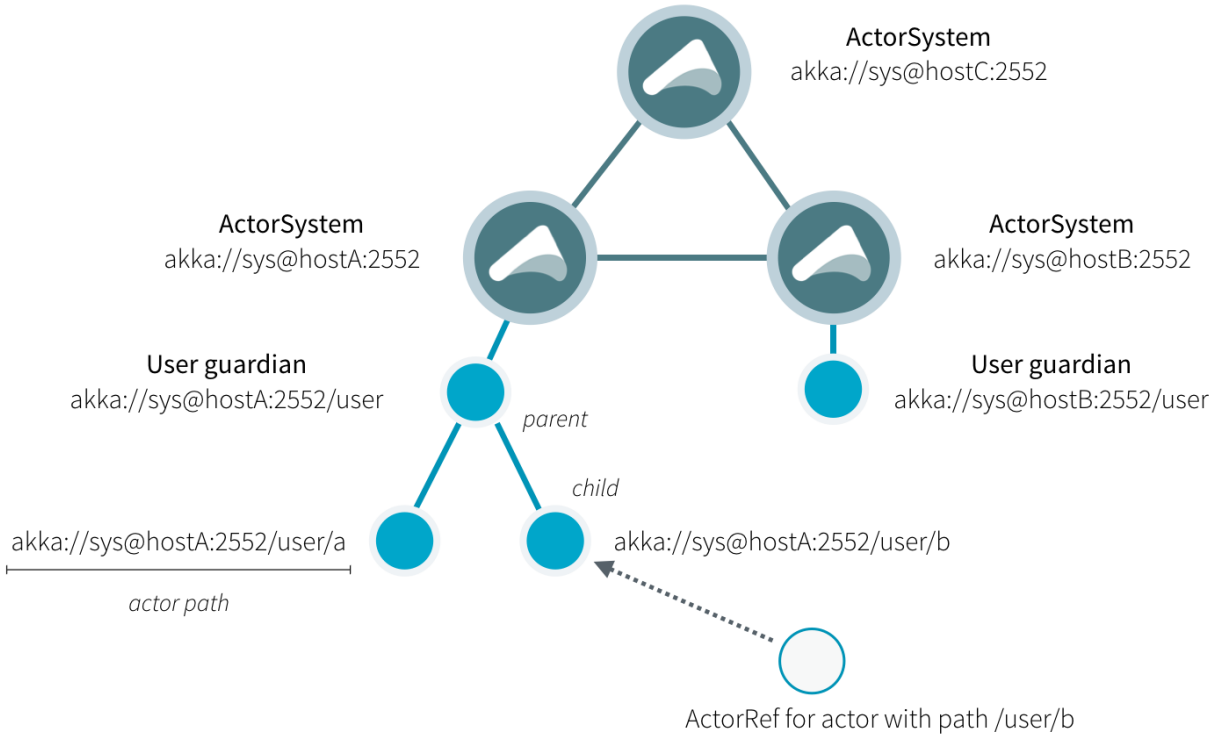

Рисунок 18 - Акторная система

Модель акторов, являющаяся основой Lagom Framework дает возможность для безболезненного и качественного горизонтального масштабирования. На старте приложение создает акторную систему и кластер, к которому, впоследствии, будут подключаться дополнительные экземпляры приложения при горизонтальном масштабировании [16]. Кластер, к которому последовательно подключаются экземпляры приложения изображен на рисунке № 19.

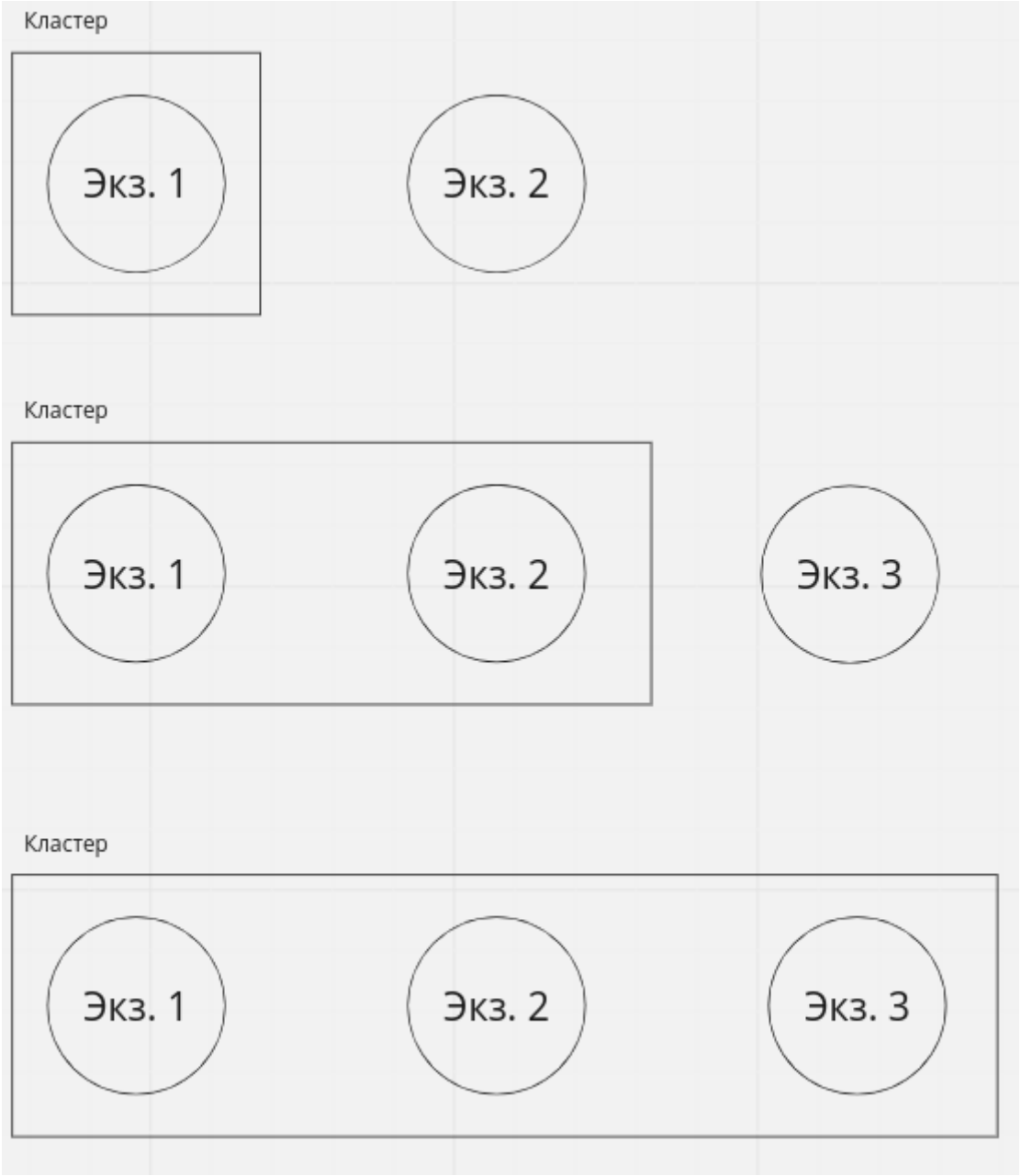

Рисунок 19 - Кластер, к которому последовательно подключаются экземпляры приложения

2.4 Разработка диаграммы классов микросервиса

Разрабатываемый микросервис будет разделен на два модуля: API и имплементация API.

В таблице 1 представлены основные классы, интерфейсы и их роли в модуле API.

Таблица 1 - Классы модуля API

Название класса	Роль класса
MeteringPointComponentCalls	Интерфейс, от которого наследуются команды
VerificationService	Находит для полученного HTTP запроса метод его обработки

Схема классов, являющихся имплементацией интерфейса MeteringPointComponentCalls, представлена на рисунке №20

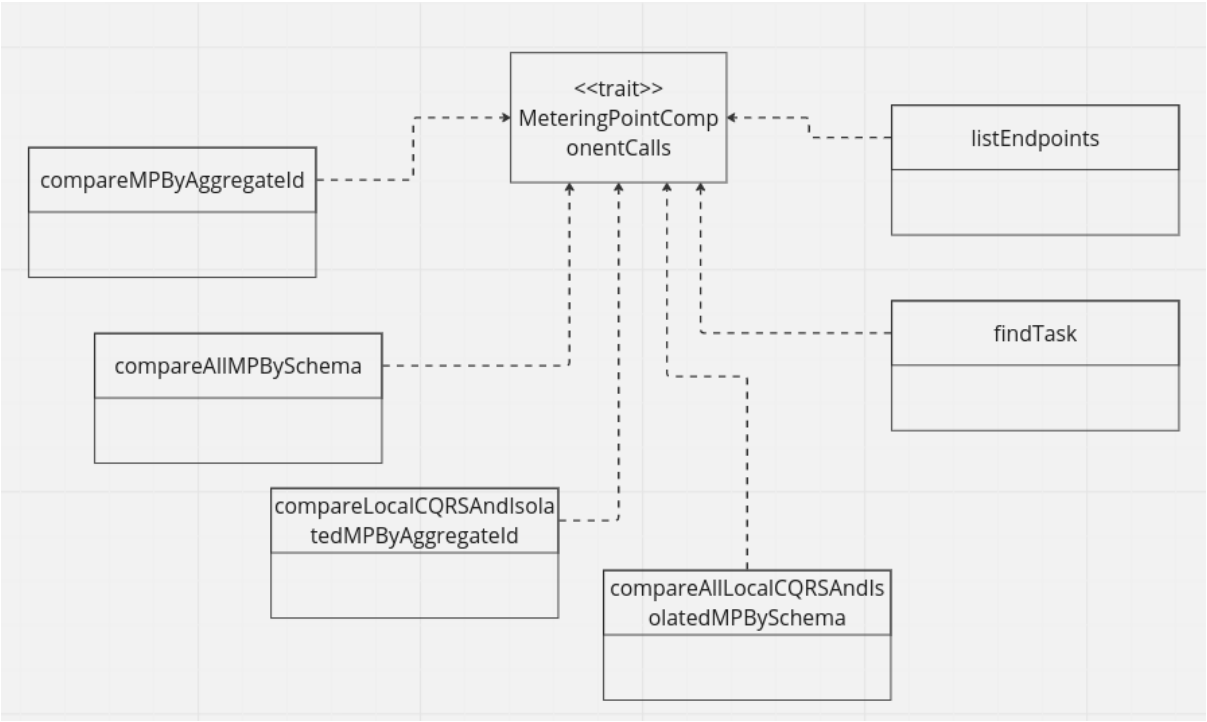

Рисунок 20 - Схема классов, являющихся имплементацией интерфейса MeteringPointComponentCalls

В таблице 2 представлены основные классы, интерфейсы и их роли в модуле, являющимся имплементация модуля API.

Таблица 2 - Классы модуля, являющиеся имплементацией модуля API

Название класса	Роль класса
MeteringPointComponents	Реализует команды из MeteringPointComponentCalls
Stage	Стадия сравнения данных из двух источников
StageResult	Хранит результат сравнения данных из двух источников по правилам за стадию
Rule	Правило сравнения данных из двух источников
RuleResult	Хранит результат сравнения данных из двух источников по правилу
VerificationJob	Запускает сравнение данных из двух источников исходя из заданных стадий
JobResult	Результат сравнение данных из двух источников
MeteringPointCompareComponents	Реализует правила и стадии для сравнения данных из двух источников Точек Учета

В данной главе был описан принцип работы систем основанных на разделении команд и событий. Представлен алгоритм обработки основной команды, принимаемой данным микросервисом. Построена диаграмма классов команд, описаны основные классы и их роли в рамках модулей. Сформированы требования к разрабатываемому микросервисному программному обеспечению, основной задачей которого является верификация данных.

Глава 3 Разработка микросервиса верификации

3.1 Выбор средств для разработки микросервиса

Начиная разработку программного обеспечения, архитектура которого основывается на микросервисном подходе, важно выбрать инструменты, которые полностью обеспечивают гарантированное выполнение поставленной цели, то есть в моем случае соответствующие принципам построения микросервисного приложения.

Основным условием выполнения данной выпускной квалификационной работы является реализация микросервиса верификации данных с использованием фреймворка Lagom. Принимая во внимание то, что основой фреймворка Lagom является модель акторов, реализация которой достигается с помощью фреймворка Akka, то выбор языка программирования для разработки микросервиса сводится к двум вариантам: Java, Scala.

Java - широко используемый язык программирования для написания приложений. Язык Java широко использовался на протяжении более двух десятилетий. Миллионы приложений Java используются и сегодня. Java – это многоплатформенный, объектно-ориентированный и сетецентрический язык, который сам по себе может использоваться как платформа. Это быстрый, безопасный и надежный язык программирования для всего: от мобильных приложений и корпоративного программного обеспечения до приложений для работы с большими данными и серверных технологий [8] .

Scala - это язык программирования, основанный на Java. Его можно использовать для разных целей, он лаконичнее и упорядоченнее, чем Java. Scala является мультипарадигмальным языком программирования, сочетающий возможности функционального и объектно-ориентированного программирования [19] . Спроектированный кратким и типобезопасным для простого и быстрого создания компонентного программного обеспечения.

При использовании Java разработчикам необходимо писать длинные строки кода, чтобы выполнить простые задачи. Scala позволяет выполнить те

же задачи, используя более лаконичный код. Задача, которая требует 20 строк Java-кода, на Scala займет всего 6. Такое сжатие кода позволяет сделать его более организованным, а также легким для чтения и дальнейшего использования. Меньшее количество строк также облегчает нахождение и исправление ошибок. Функциональная природа Scala позволяет более естественно использовать лямбда-функции и цепочки, требуя задействования меньшего количества кода [19] .

Так как оба языка работают на JVM, перед запуском их код должен одинаковым образом компилироваться в байт-код. Scala выигрывает в производительности благодаря такой технике оптимизации, как “хвостовая рекурсия” (“tail call recursion”), которая выполняется на этапе компиляции, где если последним выражением в методе является рекурсивный вызов, то он может быть опущен и заменен на итерационное решение. Кроме этой оптимизации, фактически, Java и Scala обладают близкими по своим показателям характеристиками производительности, так как оба языка работают на JVM.

В целом, Java и Scala имеют много схожестей и оба могут быть использованы для создания самых различных приложений, однако каждый из этих языков программирования имеет свои сильные и слабые стороны.

По итогу выбора между Java и Scala, для разработки микросервиса верификации данных был выбран язык программирования Scala за лаконичное написание кода и выигрыш в производительности в рамках разрабатываемого микросервисного приложения.

Выбор интегрированной среды разработки основывается на выбранном языке программирования. Наиболее популярными средами разработки, поддерживающими разработку на языке программирования Scala являются: Intellij IDEA, NetBeans и Eclipse.

IntelliJ IDEA — интегрированная среда разработки программного обеспечения для многих языков программирования, в частности Java, JavaScript, Python, разработанная компанией JetBrains.

NetBeans — интегрированная среда разработки приложений на языках программирования Java, Python, PHP, JavaScript, C, C++, Ада и ряда других.

Eclipse — интегрированная среда разработки модульных кроссплатформенных приложений. Развивается и поддерживается Eclipse Foundation.

Критерии для выбора интегрированной среды разработки:

- Удобство использования: насколько легко и интуитивно понятно использовать среду разработки;
- Функциональность: наличие необходимых инструментов и возможностей для разработки конкретного проекта;
- Интеграция с другими инструментами: наличие возможности интеграции с другими инструментами, такими как системы контроля версий, сборщики проектов и т.д;
- Скорость работы: быстродействие и отзывчивость среды разработки;
- Надежность: стабильность работы и отсутствие ошибок при использовании.

Удобство использования: Intellij IDEA и NetBeans имеют более интуитивно понятный интерфейс, чем Eclipse.

Функциональность: Intellij IDEA имеет наиболее широкий набор инструментов и возможностей для разработки проектов, в то время как NetBeans и Eclipse имеют более узкую специализацию.

Интеграция с другими инструментами: все три среды разработки имеют возможность интеграции с системами контроля версий, сборщиками проектов и другими инструментами.

Скорость работы: Intellij IDEA и NetBeans работают быстрее, чем Eclipse.

Надежность: все три среды разработки достаточно стабильны и надежны в работе.

Результат сравнения интегрированных сред разработки представлен в таблице 3.

Таблица 3 - Результат сравнения интегрированных сред разработки

Критерий	Intellij IDEA	NetBeans	Eclipse
Удобство использования	1	1	0
Функциональность	1	0	0
Интеграция с другими инструментами	1	1	1
Скорость работы	1	1	0
Надежность	1	1	1
Итог	5	4	2

В итоге сравнения Intellij IDEA, NetBeans и Eclipse была выбрана интегрированная среда разработки от JetBrains - Intellij IDEA.

Сервисы экосистемы ООО Квартплата 24 в качестве места для хранения данных используют PostgreSQL, ввиду этого она же и будет использоваться для хранения результата работы задач верификации данных.

PostgreSQL — это объектно-реляционная система управления базами данных, наиболее развитая из открытых СУБД в мире. Имеет открытый исходный код и является альтернативой коммерческим базам данных. Данная СУБД позволяет гибко управлять базами данных, с её помощью можно создавать, модифицировать или удалять записи, отправлять транзакцию — набор из нескольких последовательных запросов на особом языке запросов SQL [18] .

Фреймворком для взаимодействия с базой данных из кода была выбрана библиотека Slick.

Slick — это прогрессивная, комплексная библиотека доступа к базам данных для Scala со строго типизированными API с широкими возможностями компоновки [20] . Slick упрощает использование базы данных естественным для нее образом. Это позволяет работать с реляционными базами данных почти так же, как если бы использовались структуры данных языка программирования Scala, и в то же время дает полный контроль над тем, когда осуществляется доступ к базе данных и сколько данных передается. Запросы, написанные с использованием Slick извлекают максимальную выгоду из безопасности во время компиляции и отличной композиционности языка программирования Scala, сохраняя при этом возможность перехода к необработанному SQL, когда это необходимо для пользовательских или расширенных функций базы данных.

3.2 Реализация основных модулей микросервиса

Функциональностью данного микросервиса является создание задач на верификацию данных двух представлений Точек учета, сохранение результата верификации данных в базу данных. Основными функциями разрабатываемого микросервиса являются:

- получение команды на инициализацию верификации данных;
- верификация данных с помощью определенного набора правил;
- сохранение результата работы задачи верификации.

Создание проекта для разработки используется Scala Build Tool - автоматический сборщик проектов. Lagom проект создается с помощью выполнения команды sbt new lagom/lagom-scala, во время создания проекта разработчик должен ввести данные для проекта: название проекта, версия приложения, версия фреймворка Lagom [16].

Проект созданный с помощью команды sbt new lagom/lagom-scala имеет структуру состоящую из двух модулей: модуль API и модуль имплементации методов, обозначенных в модуле API.

Разработка начинается с описания команд в модуле API. Для создания команд необходимо объединить их в группу, для этого используется интерфейс MeteringPointComponentCalls, в котором объявлены функции:

def compareMPByAggregateId(aggregateId: String, isolated: Boolean, schema: String): ServiceCall[NotUsed, String] - сравнение представлений Точек учета из сервиса расчета и микросервиса Точки учета по идентификатору агрегата;

def compareAllMPBySchema(schema: String, isolated: Boolean, limit: Option[Int] = None): ServiceCall[NotUsed, UUID] - сравнение представлений Точек учета из сервиса расчета и микросервиса Точки учета всех Точек учета по конкретной схеме базы данных сервиса расчета;

def compareLocalCQRSAndIsolatedMPByAggregateId(aggregateId: String): ServiceCall[NotUsed, String] - сравнение представлений локального и изолированного контура Точек учета из сервиса расчета учета по идентификатору агрегата;

def compareAllLocalCQRSAndIsolatedMPBySchema(schema: String, limit: Option[Int] = None): ServiceCall[NotUsed, UUID] - сравнение представлений локального и изолированного контура Точек учета из сервиса расчета учета по конкретной схеме базы данных сервиса расчета.

Далее необходимо создать интерфейс в котором будут описаны пути для выше обозначенных методов. Данный интерфейс должен наследоваться от встроенного в фреймворк Lagom интерфейса - Service, который предоставляет один метод - descriptor. В данном методе будет описано сопоставление команд, которые получил сервис с их функциями-обработчиками.

Функция сопоставления команд с функциями-обработчиками представлена на рисунке №21.

```
def meteringPointComponentCalls: Seq[Descriptor.Call[_, _]] = Seq(
  pathCall("/api/compareMPByAggregateId/:aggregateId?isolated&schema",
    compareMPByAggregateId _),
  pathCall("/api/compareAllMPBySchema/:schema?isolated&limit",
    compareAllMPBySchema _),
  pathCall("/api/compareLocalCQRSAndIsolatedMPByAggregateId/:aggregateId",
    compareLocalCQRSAndIsolatedMPByAggregateId _),
  pathCall("/api/compareAllLocalCQRSAndIsolatedMPBySchema/:schema?limit",
    compareAllLocalCQRSAndIsolatedMPBySchema _),
)
```

Рисунок 21 - Функция сопоставления команд с функциями-обработчиками

Модуль имплементации методов из модуля API содержит в себе реализацию обозначенных выше функций. Так же для верификации данных был разработан модуль для обобщенного сравнения данных и его реализация для сравнения представлений конкретно для Точки учета. Для сохранения результата работы задач сравнения был реализован модуль взаимодействия с базой данных с использованием библиотеки Slick.

Код модуля для обобщенного сравнения данных представлен в приложении А.

Код модуля для для сравнения представлений конкретно для Точки учета представлен в приложении Б.

Код модуля взаимодействия с базой данных представлен в приложении В.

Функция compareMPByAggregateId имеет в своей реализации неблокирующие друг друга запросы данных из сервиса расчета и микросервиса Точки Учета, с последующим сравнением ответов из сервисов с помощью реализованной в компоненте сравнения функции сравнения представления из сервиса расчета и микросервиса Точки учета, после того как задача на сравнение выполнена - результат её работы записывается в базу данных.

Функция compareAllMPBySchema отличается от compareMPByAggregateId тем, что реализована с использованием поточной обработкой данных.

Функция compareLocalCQRSAndIsolatedMPByAggregateId имеет в своей реализации запросы данных из сервиса расчета, с последующим сравнением данных из изолированного и локального представления Точки учета сервиса расчета с помощью реализованной в компоненте сравнения функции сравнения изолированного и локального представления Точки учета сервиса расчета, после того как задача на сравнение выполнена - результат её работы записывается в базу данных.

Функция compareAllLocalCQRSAndIsolatedMPBySchema отличается от compareLocalCQRSAndIsolatedMPByAggregateId тем, что реализована с использованием поточной обработкой данных.

При первом запуске структура базы данных будет создана автоматически, в соответствии со спецификацией Lagom. В ней будут храниться результаты работы задач сравнения.

3.3 Тестирование разработанного микросервиса

Для проверки работоспособности микросервиса верификации данных проведем тестирование его основных функций. Тестирование будет осуществляться с помощью Postman.

Postman — это инструмент для создания, тестирования, документирования, публикации и обслуживания API. Он позволяет создавать коллекции запросов к любому API, применять к ним разные окружения, настраивать мок-серверы, писать автотесты на JavaScript, анализировать и визуализировать результаты запросов. Программа поддерживает разные виды архитектуры API: HTTP, REST, SOAP, GraphQL и WebSockets.

Микросервис верификации данных, разработанный в процессе выполнения данной выпускной квалификационной работы, способен обрабатывать следующие команды:

- предоставление списка команд;
- поиск результа завершенной задачи в базе;
- сравнение представлений Точек учета;
- сравнение представлений Точек учета по конкретной схеме базы данных сервиса расчета;
- сравнение представлений локального и изолированного контура Точек учета из сервиса расчета учета;
- сравнение представлений локального и изолированного контура Точек учета из сервиса расчета учета по конкретной схеме базы данных сервиса расчета.

Результат запроса на представление списка команд представлен на рисунке №22.

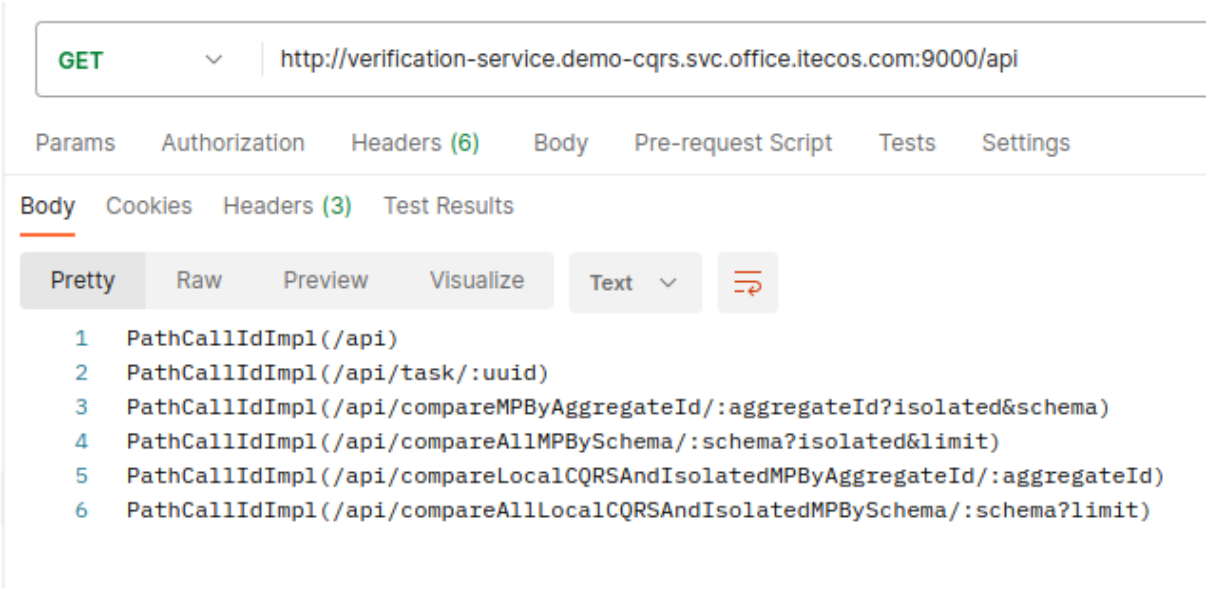

Рисунок 22 - Результат запроса на представление списка команд

Результат выполнения команды на поиск результата в базе данных микросервиса верификации данных представлен на рисунке №23.

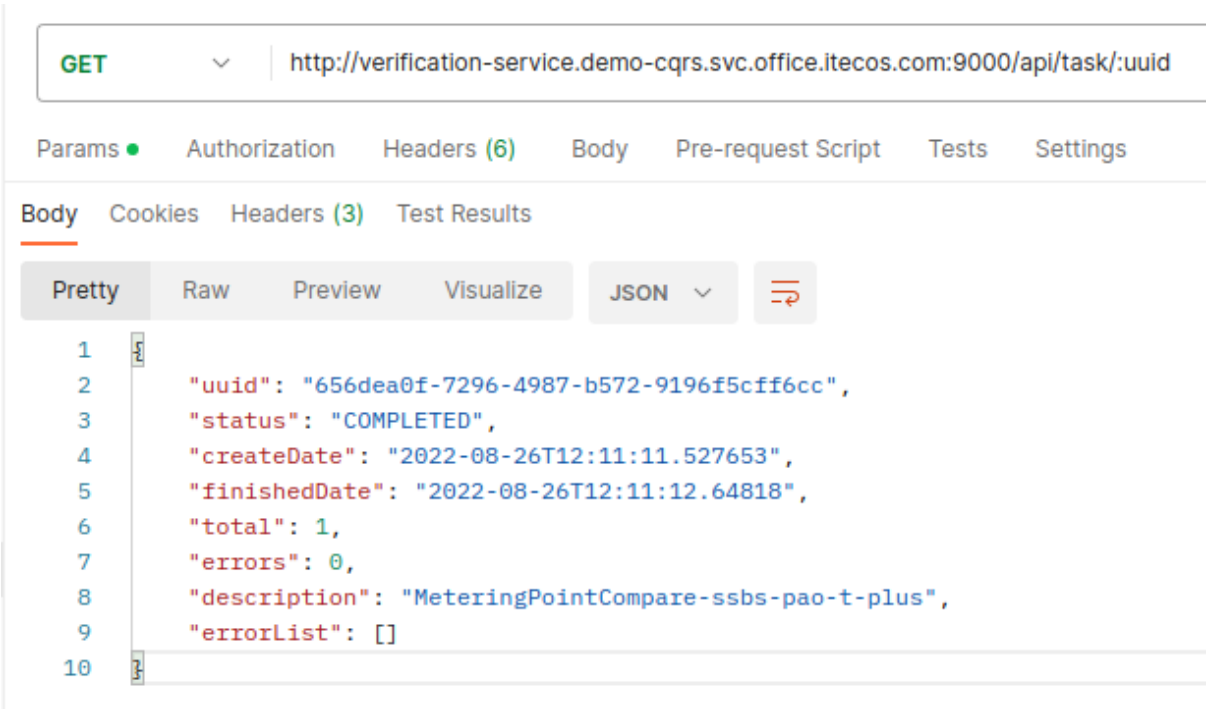

Рисунок 23 - Результат выполнения команды на поиск результата в базе данных микросервиса верификации данных

Результат работы задачи на верификацию данных Точки учета с идентификатором агрегата равным ssbs-40223,865963,01:03 представлен на рисунке №24.

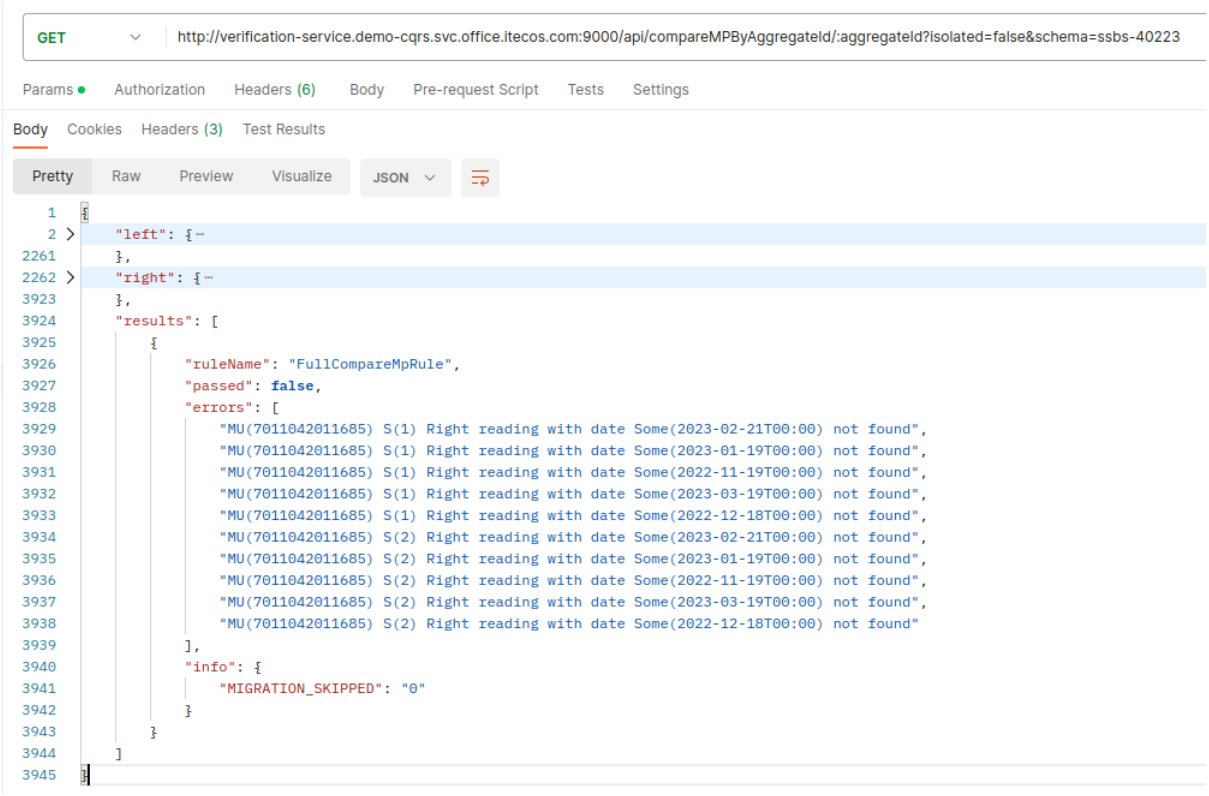

Рисунок 24 - Результат работы задачи на верификацию данных

Из результатов тестов видно, что микросервис обрабатывает команды и отображает верный результат.

Заключение

Данная выпускная квалификационная работа выполнена с целью разработки отказоустойчивых, высоконагруженных и распределенных систем с использованием технологий построения микросервисов, основанных на реактивной архитектуре.

Информационные системы, разработанные с соблюдением принципов построения микросервисов, в свою очередь основанных на принципах реактивных систем, гарантируют соответствие требованиям современного пользователя.

Выполняя данную выпускную квалификационную работу, мной были решены следующие задачи:

- Выполнен анализ существующих сервисов экосистемы ООО Квартплата 24, представляющими из себя сервисы, спроектированными по принципам монолитной архитектуры;
- Рассмотрены преимущества и недостатки построения сервиса, основанного на монолитной архитектуре;
- Описана альтернатива, в виде проектирования приложения с использованием микросервисной архитектуры;
- Проведено сравнение микросервисной и монолитной архитектуры, обозначены преимущества и недостатки;
- Подробно рассмотрены ключевые требования к разрабатываемому микросервису;
- Спроектирована архитектура разрабатываемого микросервиса с использованием фреймворка Lagom, языком программирования для разработки был выбран Scala;
- Найдено решение для верификации данных через выделенный для этого микросервис;
- Описаны команды, которые в состоянии обрабатывать данный микросервис.

Основываясь на требованиях к данной выпускной квалификационной работы, был спроектирован и реализован горизонтально масштабируемый, отказоустойчивый, гибкий микросервис, основной функцией которого является верификация данных, находящихся в разных информационных системах.

Выполнено тестирование функционала разработанного микросервиса верификации данных.

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

Список используемых источников

1. Вигерс К. Разработка требований к программному обеспечению // К. Вигерс, Д. Битти. – Спб:BHV,2014. - 736 с.

2. Галимянов Ф.А., Галимянов А.Ф. Архитектура информационных систем. – Казань: Казан. ун-т, 2019. - 117 с.

3. Ездаков А.Л. Функциональное и логическое программирование. – М.: Бином. Лаборатория знаний, 2009. - 120с.

4. Официальный сайт ООО Квартплата 24 [Электронный ресурс]. URL: https://www.kvp24.ru/ (дата обращения: 01.02.2023)

5. Трутнев Д. Р. Архитектуры информационных систем. Основы проектирования: Учебное пособие. – СПб.: НИУ ИТМО, 2012. - 66 с.

6. Фаулер М. Рефакторинг. Улучшение существующего кода // . Фаулер. – СПб : Символ Плюс, 2015. - 415 с.

7. Akka Documentation [Электронный ресурс]. URL: https://doc.akka.io/docs/akka/current/typed/actors.html (дата обращения: 03.03.2023)

8. Baesens, B. Beginning Java Programming: The Object-Oriented Approach / B. Baesens, A. Backiel, S. Vanden Broucke. – 1st edition, Wrox, 2015. 620 с.

9. CQRS template [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/azure/architecture/patterns/cqrs (дата обращения: 10.02.2023)

10. CQRS, Event Sourcing and DDD FAQ [Электронный ресурс]. URL: https://cqrs.nu/Faq (дата обращения: 01.02.2023)

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

12. Event sourcing template [Электронный ресурс]. URL: https://docs.microsoft.com/ru-RU/azure/architecture/patterns/event-sourcing (дата обращения: 03.03.2023)

13. Functional Programming For The Rest of Us [Электронный ресурс]. URL: https://www.defmacro.org/2006/06/19/fp.html (дата обращения: 13.03.2023)

14. Hudson O. Getting started with IntelliJ IDEA // O. Hudson, Birmingham: Packt Publishing, 2013. 114 с.

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

16. Lagom Documentation [Электронный ресурс]. URL: https://www.lagomframework.com/documentation/ (дата обращения: 03.03.2023)

17. Materialized view template [Электронный ресурс]. URL: https://docs.microsoft.com/ru-RU/azure/architecture/patterns/materialized-view (дата обращения: 10.02.2023)

18. PostgreSQL Documentation [Электронный ресурс]. URL: https://www.postgresql.org/docs/current/index.html (дата обращения: 03.03.2023)

19. Scala Book [Электронный ресурс]. URL: https://docs.scala-lang.org/overviews/scala-book/introduction.html (дата обращения: 01.02.2023)

20. Slick [Электронный ресурс]. URL: https://scala-slick.org/ (дата обращения: 13.03.2023)

21. Tutorials Technology [Электронный ресурс]. URL: http://www.gwtproject.org/doc/latest/tutorial/index.html (дата обращения: 03.03.2023)

Приложение А

Код модуля для обобщенного сравнения данных

```
package ru.kvp24.verification.compare

import akka.stream.Materializer
import akka.stream.scaladsl.Sink
import akka.stream.scaladsl.Source
import play.api.libs.json.Format
import play.api.libs.json.Json
import play.api.libs.json.OFormat

import scala.concurrent.ExecutionContext
import scala.concurrent.Future

object CompareCoreComponents {
  object CompareCoreFormats {
    implicit val ruleResultJsonFormat: Format[RuleResult] = Json.format[RuleResult]
    implicit def jobResultJsonFormat[L, R](implicit leftFormat: Format[L], rightFormat: Format[R]): Format[JobResult[L, R]] = {
      Json.format[JobResult[L, R]]
    }
  }
  case class StageResult(results: Iterable[RuleResult]) {
    override def toString: String = results.mkString("\n")
  }

  case class Stage[L, R](rules: Seq[Rule[L, R]], description: String = "", parallelism: Int = 5) {
    def run(
      left: L,
      right: R,
      errorProcessor: RuleResult => Future[RuleResult] = Future.successful
    )(implicit
      materializer: Materializer,
      executionContext: ExecutionContext
    ): Future[StageResult] = {
      Source(rules)
        .mapAsyncUnordered(parallelism) { rule =>
          rule.compare(left, right)
        }
        .mapAsyncUnordered(parallelism) {
```

```
          errorProcessor
        }
        .runWith(Sink.collection)
        .map { results =>
          StageResult(results)
        }
      }
    }

    case class RuleResult(ruleName: String, passed: Boolean = true, errors: Seq[String] = Seq.empty, info: Map[String, String] = Map.empty) {
      override def toString: String =
        s"""$ruleName has ${if (passed) "passed" else "failed"} ${if (errors.isEmpty) ""
        else errors.mkString("errors are:\n\t", "\n\t", "")}
        info: ${info}
        """.stripMargin

    }

    trait Rule[L, R] {
      def ruleName: String

      def compare(left: L, right: R)(implicit executionContext: ExecutionContext): Future[RuleResult]

      def ok: RuleResult = RuleResult(ruleName)

      def errors(errors: Seq[String]): RuleResult = RuleResult(ruleName, passed = false, errors)
    }

    case class JobResult[L, R](left: L, right: R, results: Seq[RuleResult] = Seq()) {
      def addResults(res: Iterable[RuleResult]): JobResult[L, R] = copy(results = results ++ res)

      def isPassed: Boolean = results.forall(_.passed)

      def liftToOption: JobResult[Option[L], Option[R]] = JobResult(Some(left), Some(right), results)
    }

    trait VerificationJob[L, R] {
      def stages: Seq[Stage[L, R]]
```

```
        def run(left: L, right: R, errorProcessor: RuleResult => Future[RuleResult]
= Future.successful)(implicit
            materializer: Materializer,
            executionContext: ExecutionContext
          ): Future[JobResult[L, R]] = {
           stages
            .foldLeft(Future.successful(JobResult(left, right))) { (acc, el) =>
             acc.flatMap { job =>
              if (job.isPassed) {
               el.run(left, right, errorProcessor).map { stageResult =>
                job.addResults(stageResult.results)
               }
              } else Future.successful(job)
             }
            }
          }

        def runLifted(left: Option[L], right: Option[R], errorProcessor: RuleResult
=> Future[RuleResult] = Future.successful)(implicit
            materializer: Materializer,
            executionContext: ExecutionContext
          ): Future[JobResult[Option[L], Option[R]]] = {
           (left, right) match {
            case (Some(l), Some(r)) =>
             run(l, r, errorProcessor).map(_.liftToOption)
            case _ =>
             val ruleResult = RuleResult("Arguments check", passed = false, errors
= Seq("One of arguments is missing"))
             val jobResult  = JobResult(left, right, Seq(ruleResult))
             errorProcessor(ruleResult).map(_ => jobResult)
           }
          }
         }
        }
```

Приложение Б

Код модуля для для сравнения представлений конкретно для Точки учета

```
class MeteringPointCompareComponents {
  val compareStages: Seq[Stage[MeteringPointV2,
MeteringPointMaterializedView.MeteringPoint]] = Seq(
    Stage(
      rules = Seq(FullCompareMpRule),
      description = "FullCompareMpRule"
    )
  )

  private val verificationJob: MeteringPointVerificationJob =
MeteringPointVerificationJob(compareStages)

  def compareSingle(
     left: Option[MeteringPointV2],
     right: Option[MeteringPointMaterializedView.MeteringPoint],
  )(implicit
     materializer: Materializer,
     executionContext: ExecutionContext
  ): Future[JobResult[Option[MeteringPointV2],
Option[MeteringPointMaterializedView.MeteringPoint]]] = {
    verificationJob.runLifted(left, right)
  }
  def compareInTask(
     repository: TaskRepository,
     uuid: UUID,
     left: Option[MeteringPointV2],
     right: Option[MeteringPointMaterializedView.MeteringPoint]
  )(implicit
     materializer: Materializer,
     executionContext: ExecutionContext
  ): Future[JobResult[Option[MeteringPointV2],
Option[MeteringPointMaterializedView.MeteringPoint]]] = {
    val aggregateId                              =
left.map(_.tmpAggregateId).orElse(right.map(_.aggregateId))
    def errorProcessor: RuleResult => Future[RuleResult] =
saveError(repository, uuid, aggregateId, _)
    verificationJob.runLifted(left, right, errorProcessor)
  }
```

```
        def saveError(repository: TaskRepository, uuid: UUID, aggregateId:
Option[String], ruleResult: RuleResult)(implicit
            executionContext: ExecutionContext
          ): Future[RuleResult] = {
            if (!ruleResult.passed) {
              repository
                .insertError(taskUuid = uuid, error = ruleResult.errors.mkString("; "),
ruleName = ruleResult.ruleName, aggregateId = aggregateId, info =
ruleResult.info.mkString("; "))
                .map(_ => ruleResult)
            } else {
              Future.successful(ruleResult)
            }
          }

          case class MeteringPointVerificationJob(stages:
Seq[Stage[MeteringPointV2, MeteringPointMaterializedView.MeteringPoint]])
              extends VerificationJob[MeteringPointV2,
MeteringPointMaterializedView.MeteringPoint]

          case object MUCountRule extends Rule[MeteringPointV2,
MeteringPointMaterializedView.MeteringPoint] {
            override def ruleName: String = "Compare Measure Units count in
Metering Point"

            override def compare(left: MeteringPointV2, right:
MeteringPointMaterializedView.MeteringPoint)(implicit
                executionContext: ExecutionContext
            ): Future[RuleResult] = Future {
              val meteringPointFromSSBSMuSize    = left.measureUnits.size
              val meteringPointFromServiceMuSize =
right.inlets.values.flatMap(_.measureUnits).size

              if (meteringPointFromSSBSMuSize ==
meteringPointFromServiceMuSize) ok
              else
                errors(
                  Seq(
                    s"SSBS MU count != Service MU count <=>
${meteringPointFromSSBSMuSize} != ${meteringPointFromServiceMuSize}"
                  )
                )
            }
          }
```

```
        case object UninstallDateMURule extends Rule[MeteringPointV2,
MeteringPointMaterializedView.MeteringPoint] {
          override def ruleName: String = "Compare Measure Unit uninstall date"

          override def compare(left: MeteringPointV2, right:
MeteringPointMaterializedView.MeteringPoint)(implicit
              executionContext: ExecutionContext
            ): Future[RuleResult] = Future {
            val muFromSSBS: Map[UUID, Option[LocalDateTime]] =
left.measureUnits.map { mu =>
              (mu.mPMeasureUnitUUID, mu.uninstallDate)
            }.toMap

            val muFromService: Map[UUID, Option[LocalDateTime]] =
right.inlets.values
              .flatMap(_.measureUnits)
              .map { mu =>
                (mu.uuid, mu.uninstallDate)
              }
              .toMap

            val errs: Iterable[String] = muFromSSBS.keys.flatMap { uuid =>
              val uDSSBS    = muFromSSBS.get(uuid).flatten
              val uDService = muFromService.get(uuid).flatten
              if (uDSSBS == uDService) Seq()
              else
                Seq(s"MU UUID = $uuid, SSBS MU uninstallDate != Service MU
uninstallDate <=> ${uDSSBS} != ${uDService}")
            }

            if (errs.isEmpty) ok
            else errors(errs.toSeq)

          }
        }

        case object FullCompareMpRule extends Rule[MeteringPointV2,
MeteringPointMaterializedView.MeteringPoint] {
          override def ruleName: String = "FullCompareMpRule"

          override def compare(left: MeteringPointV2, right:
MeteringPointMaterializedView.MeteringPoint)(implicit

executionContext: ExecutionContext
```

```
            ): Future[RuleResult] = Future {
              val (resultErrors, infoMap) =
MeteringPointComparator.fullCompare(left,
MeteringPointConverter.convert(right))
              val result =
                if (resultErrors.isEmpty) {
                  ok
                } else {
                  errors(resultErrors)
                }
              result.copy(info = infoMap)
            }
          }
        }
```

Приложение В

Код модуля взаимодействия с базой данных

```
package ru.kvp24.verification.impl.repository

import akka.actor.ActorSystem
import com.typesafe.config.Config
import ru.kvp24.verification.api.Replies.TaskError
import ru.kvp24.verification.api.Replies.TaskStatus
import
ru.kvp24.verification.impl.repository.TaskSlickRepository.TaskStatusType.TaskSt
atusType
import ru.kvp24.verification.impl.repository.TaskSlickRepository._
import slick.ast.BaseTypedType
import slick.basic.DatabaseConfig
import slick.dbio.Effect._
import slick.jdbc.JdbcBackend.Database
import slick.jdbc.JdbcProfile
import slick.jdbc.JdbcType
import slick.jdbc.meta.MTable

import java.time.LocalDateTime
import java.util.UUID
import scala.concurrent.ExecutionContext
import scala.concurrent.Future

trait TaskRepository {
  def startTask(uuid: UUID, description: String): Future[Int]
  def finishTask(uuid: UUID, total: Long, errors: Long): Future[Int]
  def failTask(uuid: UUID): Future[Int]
  def updateTask(uuid: UUID, total: Long, errors: Long): Future[Int]
  def insertError(taskUuid: UUID, aggregateId: Option[String], ruleName:
String, error: String, info: String = ""): Future[Int]
  def findTask(taskUuid: UUID): Future[Option[TaskStatus]]
  def findRunningTask(description: String): Future[Option[UUID]]
}

object TaskSlickRepository {
  object TaskStatusType extends Enumeration {
    type TaskStatusType = Value
    val IN_PROGRESS, COMPLETED, FAILED = Value
  }
  case class TaskRow(
```

```
            uuid: UUID,
            status: TaskStatusType,
            createDate: LocalDateTime = LocalDateTime.now(),
            finishedDate: Option[LocalDateTime] = None,
            total: Long,
            errors: Long,
            description: String
        )

        case class ErrorRow(
            id: Long,
            taskUuid: UUID,
            aggregateId: Option[String],
            ruleName: String,
            date: LocalDateTime = LocalDateTime.now(),
            error: String,
            info: String = ""
        )

    }

    class TaskSlickRepository(database: Database, actorSystem:
ActorSystem)(implicit val executionContext: ExecutionContext) extends
TaskRepository {
        val readSideConfig: Config =
actorSystem.settings.config.getConfig("lagom.persistence.read-side.jdbc")
        val profile: JdbcProfile   = DatabaseConfig.forConfig[JdbcProfile]("slick",
readSideConfig).profile

        import profile.api._

        val taskTable  = TableQuery[TaskTable]
        val errorTable = TableQuery[ErrorTable]

        private val tables = Vector(
          taskTable,
          errorTable
        )

        def createTables(): DBIOAction[Vector[Unit], NoStream, Read with
Schema] = {
          MTable.getTables
            .map(r => r.map(_.name.name))(database.ioExecutionContext)
            .flatMap { existing =>
              DBIO.sequence(
```

```
            tables.filter(t =>
!existing.contains(t.baseTableRow.tableName)).map(_.schema.create)
          )
        }(database.ioExecutionContext)
      }

      implicit val accountTypeMapper: JdbcType[TaskStatusType] with
BaseTypedType[TaskStatusType] =
        MappedColumnType.base[TaskStatusType, String](
          e => e.toString,
          s => TaskStatusType.withName(s)
        )

      class TaskTable(tag: Tag) extends Table[TaskRow](tag, "task") {
        def uuid: Rep[UUID]                    = column[UUID]("uuid",
O.PrimaryKey)
        def status: Rep[TaskStatusType]        =
column[TaskStatusType]("status")
        def createDate: Rep[LocalDateTime]     =
column[LocalDateTime]("create_date")
        def finishedDate: Rep[Option[LocalDateTime]] =
column[Option[LocalDateTime]]("finished_date")
        def total: Rep[Long]                   = column[Long]("total")
        def errors: Rep[Long]                  = column[Long]("errors")
        def description: Rep[String]           = column[String]("description")
        def * = (
          uuid,
          status,
          createDate,
          finishedDate,
          total,
          errors,
          description
        ) <> (TaskRow.tupled, TaskRow.unapply)
      }

      class ErrorTable(tag: Tag) extends Table[ErrorRow](tag, "error") {
        def id: Rep[Long]              = column[Long]("id", O.PrimaryKey,
O.AutoInc)
        def taskUuid: Rep[UUID]        = column[UUID]("task_fk")
        def aggregateId: Rep[Option[String]] =
column[Option[String]]("aggregate_id")
        def ruleName: Rep[String]      = column[String]("rule_name")
        def date: Rep[LocalDateTime]   = column[LocalDateTime]("date")
        def error: Rep[String]         = column[String]("error")
```

```
    def info: Rep[String]              = column[String]("info")
    def * = (
      id,
      taskUuid,
      aggregateId,
      ruleName,
      date,
      error,
      info
    ) <> (ErrorRow.tupled, ErrorRow.unapply)
  }

  override def startTask(uuid: UUID, description: String): Future[Int] = {
    val row = TaskRow(uuid = uuid, status =
TaskStatusType.IN_PROGRESS, total = 0, errors = 0, description = description)
    saveTask(row)
  }

  override def finishTask(uuid: UUID, total: Long, errors: Long): Future[Int]
= {
    updateTask(uuid, TaskStatusType.COMPLETED, total, errors)
  }

  override def findTask(uuid: UUID): Future[Option[TaskStatus]] = {
    println(uuid)
    val query = for {
      task   <- taskTable.filter(_.uuid === uuid).result.headOption
      errors <- errorTable.filter(_.taskUuid === uuid).result
    } yield (task, errors)
    val action = query.map { case (taskOpt, errors) =>
      val taskErrors = errors.map { err =>
        TaskError(aggregateId = err.aggregateId, ruleName = err.ruleName,
date = err.date, error = err.error, info = err.info)
      }
      taskOpt.map { task =>
        TaskStatus(
          uuid = uuid,
          status = task.status.toString,
          createDate = task.createDate,
          finishedDate = task.finishedDate,
          total = task.total,
          errors = task.errors,
          description = task.description,
          errorList = taskErrors
        )
```

```
      }
    }
    database.run(action)
  }

  override def findRunningTask(description: String): Future[Option[UUID]]
= {
    val action = taskTable
      .filter(task => task.description === description && task.status ===
TaskStatusType.IN_PROGRESS)
      .map(_.uuid)
      .result
      .headOption
    database.run(action)
  }
  def saveTask(task: TaskRow): Future[Int] = {
    database.run(taskTable.insertOrUpdate(task))
  }
  def updateTask(uuid: UUID, status: TaskStatusType, total: Long, errors:
Long): Future[Int] = {
    val action = taskTable
      .filter(_.uuid === uuid)
      .map(task => (task.total, task.errors, task.status, task.finishedDate))
      .update((total, errors, status, Some(LocalDateTime.now())))
    database.run(action)
  }

  def getTask(uuid: UUID): Future[Option[TaskRow]] = {
    val query = taskTable.filter(_.uuid === uuid)
    database.run(query.result.headOption)
  }
  def deleteTask(uuid: UUID): Future[Int] = {
    val action = taskTable.filter(_.uuid === uuid).delete
    database.run(action)
  }
  def getError(id: Long): Future[Option[ErrorRow]] = {
    val query = errorTable.filter(_.id === id)
    database.run(query.result.headOption)
  }
  def getErrors(taskUuid: UUID): Future[Seq[ErrorRow]] = {
    val query = errorTable.filter(_.taskUuid === taskUuid)
    database.run(query.result)
  }
  override def insertError(taskUuid: UUID, aggregateId: Option[String],
ruleName: String, error: String, info: String = ""): Future[Int] = {
```

```
      val action = errorTable.map(err =>
        (err.taskUuid, err.aggregateId, err.ruleName, err.date, err.error, err.info)
      ) += (taskUuid, aggregateId, ruleName, LocalDateTime.now(), error,
info)
      database.run(action)
    }
    def deleteErrorByTask(taskUuid: UUID): Future[Int] = {
      val action = errorTable.filter(_.taskUuid === taskUuid).delete
      database.run(action)
    }

    object RepositoryMappers {}

    override def failTask(uuid: UUID): Future[Int] = {
      val action = taskTable
        .filter(_.uuid === uuid)
        .map(x => (x.status, x.finishedDate))
        .update((TaskStatusType.FAILED, Some(LocalDateTime.now())))
      database.run(action)
    }

    override def updateTask(uuid: UUID, total: Long, errors: Long):
Future[Int] = {
      val action = taskTable
        .filter(_.uuid === uuid)
        .map(x => (x.total, x.errors))
        .update((total, errors))
      database.run(action)
    }
  }
```