МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет вычислительной техники

Кафедра МО и ПЭВМ

Направление подготовки 09.03.04 «Программная инженерия»

БАКАЛАВРСКАЯ РАБОТА

на тему

Серверные программные средства управления метаданными для сервиса потоковой обработки событий

Студент _____ Гусаров Александр Сергеевич
(подпись, дата) (ФИО полностью)

Руководитель _____ Шибанов С.В.
(подпись, дата) (фамилия, инициалы)

Нормоконтролер _____ Такташкин Д.В
(подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от _____№_____)

Заведующий кафедрой _____ Макарычев П.П.
(подпись) (фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____№____)

Секретарь ГЭК _____ Попова Н.А.
(подпись) (фамилия, инициалы)

Пенза, 2022

Федеральное государственное бюджетное образовательное учреждение высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет	*Кафедра*
Вычислительной техники	МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
_____П.П. Макарычев
«__» _____2022 г

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА

1. Студент Гусаров Александр Сергеевич гр. 18ВП2 факультета ВТ
<Фамилия, имя, отчество полностью>

2.Тема работы Серверные программные средства управления метаданными для сервиса потоковой обработки событий

Тема утверждена приказом ПГУ № 327/О от " 18 " апреля 2022 г.

3. Руководитель работы к.т.н., доцент кафедры «МО и ПЭВМ» Шибанов С.В.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

1) Назначение: серверные программные средства для управления метаданными в составе сервиса потоковой обработки событий

2) Среда выполнения: Linux

Языковые средства разработки: TypeScript

Инструментальные средства: VS Code, PostgreSQL 13, DBeaver, Docker, Draw.io.

Технологии разработки ООП, сервисно-ориентированный подход.

3) Основные функции разрабатываемого программного обеспечения:

а) взаимодействие с репозиторием метаданных;

б) регистрация, аутентификация и авторизация пользователей;

в) управление метаданными;

г) анализ корректности метаданных;

д) взаимодействие с клиентскими приложениями;

е) оповещение и доставка данных микросервисам;

ж) взаимодействие со средствами журнализации;

з) предотвращение несанкционированного доступа к механизмам управления;

5. Перечень подлежащих разработке вопросов:

1) *Анализ предметной области и постановка задачи;*
2) *План разработки бюджета;*
3) *Разработка требований и проектирование;*
4) *Реализация;*
5) *Контроль качества программного обеспечения;*
6) *Оформление пояснительной записки.*

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области и постановка задачи	10%	13.04.2022 – 22.04.2022	
2	План разработки бюджета	5%	23.04.2022 – 25.04.2022	
3	Проектирование	10%	26.04.2022 – 03.05.2022	
4	Реализация	50%	04.05.2022 – 02.06.2022	
5	Контроль качества программного обеспечения	10%	03.06.2022 – 08.06.2022	
6	Оформление пояснительной записки	15%	09.06.2022 – 17.06.2022	

Дата выдачи задания «22» ____декабря____2021 г.

Руководитель бакалаврской работы _____
(подпись, дата)

Задание к исполнению принял студент _____
(подпись, дата)

Работу к защите допустить
Декан факультета _____
(подпись, дата)

Реферат

Пояснительная записка содержит 122 листа, 37 рисунков, 8 таблиц, 20 использованных источника и 4 приложения.

СЕРВЕРНЫЕ ПРОГРАММНЫЕ СРЕДСТВА УПРАВЛЕНИЯ МЕТАДАННЫМИ ДЛЯ СЕРВИСА ПОТОКОВОЙ ОБРАБОТКИ СОБЫТИЙ, REST, UML, TYPESCRIPT, NEST, JEST, NODEJS, REDIS, POSTGRESQL, СЕРВЕР, МИКРОСЕРВИС, NGINX, DOCKER.

Объект исследования – сервисы свободного распространения наборов данных и основные методы визуализации данных.

Цель выпускной квалификационной работы – разработать серверные программные средства управления метаданными в составе сервиса потоковой обработки событий с возможностью обмена данными с клиентскими приложениями и микросервисами.

Технологии разработки – разработка серверных программных средств проводилась на языке программирования TypeScript с использованием фреймворка Nest и ORM Sequalize, программное обеспечение разворачивалось при помощи средств Docker, в качестве web-сервера был использован Nginx, для создания репозитория использовался PostgreSQL 13, обмен данными с микросервисами производился при помощи средств REDIS. В качестве среды разработки использовался редактор программного кода VS Code и менеджер баз данных DBeaver.

Среда исполнения – Linux.

Результаты работы – в ходе выполнения выпускной квалификационной работы был разработан программный комплекс управления метаданными в составе сервиса потоковой обработки событий. Все требования к разработке приложения соблюдены.

Содержание

Введение 7
1 Анализ предметной области и постановка задачи на разработку серверных программных средств 9
1.1 Сервис потоковой обработки событий на основе активных правил 9
1.1.1 Назначение и функциональные возможности сервиса потоковой обработки событий 9
1.1.2 Микросервисная событийно-ориентированная архитектура сервиса потоковой обработки событий 12
1.1.3 Сценарий работы сервиса потоковой обработки событий 14
1.2 Управление метаданными на сервисе потоковой обработки событий и исполнения активных правил 15
1.2.1 Состав метаданных 15
1.2.2 Жизненный цикл метаданных 16
1.2.3 Основные процессы управления метаданными 17
1.3 Постановка задачи на разработку серверных программных средств 20
1.3.1 Цели разработки и назначение программных средств 20
1.3.2 Роль и место серверных программных средств в процессе управления метаданными 21
1.3.3 Перечень автоматизируемых функций серверных программных средств 22
1.4 Обзор аналогичных работ в заданной предметной области 22
1.4.1 Отладчик активных правил DEAR и система визуализации исполнения активных правил Adela 22
1.4.2 Программное средство управления активными правилами и событиями IBM WebSphere ILOG JRules 26
1.4.3 Средство оптимизированной обработки и удобного операционного и стратегического анализа данных Nextbi Аналитика 29
1.4.4 Результаты обзора аналогичных работ в заданной предметной области 31

1.5 Планирование процесса разработки 32
1.5.1 Планирование процесса разработки и оценка длительности выполнения задач 32
1.5.2 Анализ ресурсов разработки 34
2 Проектирование и разработка требований к серверным программным средствам управления метаданными 38
2.1 Функциональные требования 38
2.2 Выявление требований и моделирование репозитория метаданных для сервиса потоковой обработки событий 40
2.2.1 Концептуальное моделирование 40
2.2.2 Логическое моделирование 42
2.2.3 Физическое моделирование 44
2.3 Требования к архитектуре программных средств 46
2.4 Проектирование модулей REST-сервера 48
2.5 Обоснование выбора технологий и средств разработки 51
2.6 Требования к аппаратно-программному обеспечению 53
2.6.1 Требования к аппаратно-техническому обеспечению 53
2.6.2 Требования к прикладному и системному программному обеспечению 53
3 Разработка серверных программных средств управления метаданными для сервиса потоковой обработки событий 55
3.1 Разработка архитектуры развертывания 55
3.2 Разработка программных средств взаимодействия с базой метаданных 56
3.2.1 Разработка физической структуры базы метаданных 56
3.2.2 Разработка программных средств доступа к базе метаданных 58
3.3 Разработка базовых программных средств 60
3.3.1 Разработка программных средств управления метаданными 60
3.3.2 Разработка программных управления пользовательскими аккаунтами 62
3.3.3 Разработка программных средств аутентификации пользователей 64
3.3.4 Разработка программных средств анализа корректности метаданных . 66

3.4 Разработка программных средств взаимодействия с внешними приложениями 69
3.4.1 Разработка программных средств взаимодействия с клиентским приложением 69
3.4.2 Разработка программных средств доставки метаданных к микросервисам 75
4 Контроль качества серверных программных средств 77
4.1 Планирование процесса контроля качества 77
4.1.1 Мероприятия по контролю качества программных средств 77
4.1.2 Обоснование выбора типов автоматизированного тестирования 79
4.1.3 Обоснование выбора метрик программного кода 80
4.2 Автоматизированное тестирование 81
4.2.1 Модульное тестирование 81
4.2.2 Сквозное тестирование 82
4.3 Оценка качества программного кода 83
4.3.1 Метрика Холстеда 83
4.3.2 Устойчивость и абстрактность 87
Заключение 90
Список использованных источников 92
Приложение А. Глоссарий 94
Приложение Б. Код сервисов у модулей Event, Active rule, Auth 96
Приложение В. Код компонентов обработки запроса по API 113
Приложение Г. Код тестирования классов модуля Role 119

Введение

Объемы собираемых и обрабатываемых данных постоянно растут, что является одной из основных причин бурного развития инфраструктуры обработки данных. Зачастую компании используют монолитные системы обработки данных, в которых компоненты тесно связаны между собой. Такой подход имеет ряд недостатков, основными из которых являются: высокая зависимость между компонентами, сложность модернизации, сложность поддержки кода при большом объеме функционала, единый набор технологий и программных средств. Проблема или ошибка в модуле такой системы может замедлить или разрушить все приложение. Со временем такие приложения сложно развивать и масштабировать.

В данном случае выручают средства потоковой обработки событий. Они приобретают все большую популярность и распространенность. Они способны моментально реагировать на события, которые поступают от сторонних наблюдаемых систем. Благодаря этому повышается скорость и правильность принятия решений.

Сервис потоковой обработки событий необходим для событийно-ориентированной обработки данных и событий, постигаемых от сторонней наблюдаемой системы. Обработка событий сопровождается механизмом активных правил, которые позволяют реагировать на события и их контекст соответствующим образом.

Микросервисная архитектура является хорошим выбором для подобных программных средств. Она дает множество преимуществ, основными из которых являются: удобная и простая масштабируемость, вывод в реальном времени, отказоустойчивость и высокая доступность. События вместе со специализированным контекстом поступают в сервис через очереди сообщений, которые позволяют балансировать нагрузку и повышают надежность. Сервис извлекает значимые события для конкретных задач и дает возможность динамического реагирования и воздействия на бизнес-приложения или службы в реальном времени. Во время всего

жизненного цикла сервиса присутствует огромное количество метаданных. Это может быть информация, связанная с активными правилами и событиями, или информация о конфигурации наблюдаемых компонентов и управляемых систем. Возникает необходимость управления всеми метаданными.

Целью выпускной квалификационной работы является разработка серверных программные средства управления метаданными для сервиса потоковой обработки событий. Работа включает в себя анализ требований, разработку требований, проектирование, реализацию и контроль качества программного обеспечения.

1 Анализ предметной области и постановка задачи на разработку серверных программных средств

1.1 Сервис потоковой обработки событий на основе активных правил

1.1.1 Назначение и функциональные возможности сервиса потоковой обработки событий

Обрабатывать большой поток данных с каждым годом становиться все сложнее и сложнее. Сервис потоковой обработки событий на основе активных правил выступает в роли платформы, которая получает поток событий от сторонних наблюдаемых систем. Такие наблюдаемые системы называют индустриальным интернетом вещей (IIoT). Примерами таких систем могут быть различные технические объекты или базы данных. Обработка событий сопровождается механизмом активных правил, который позволяет реагировать на события и их контекст соответствующим образом.

Сервис потоковой обработки событий в общем виде может представлять собой систему, на вход которой поступают определенные события со специализированным контекстом, а на выходе происходят соответствующие реакции.

Это позволяет проектировать алгоритмы для потоковой обработки данных и конструировать системы больших масштабов, вплоть до цифрового двойника сложного промышленного оборудования или целых предприятий. Данный сервис отлично подходит для таких отраслей как: промышленность, здравоохранение, розничная торговля, финансы и т.д.

Применительно к техническим объектам сервис может быть использован в промышленных предприятиях для контроля целой линии производства или в медицинских учреждениях для предупреждения врачей о состоянии пациентов. Упрощенным примером может служить система «Умный дом», где при срабатывании соответствующих датчиков происходят действия, необходимые для упрощения жизнедеятельности или для обеспечения безопасности помещения. Например, на входе в помещение

стоит датчик движения. Как тогда он отправит соответствующее событие на сервис, то он проанализирует метаданные события и исполнит заранее запрограммированное действие, например, в помещении включится свет.

В основе сервиса лежит модель Event-Condition-Active (ECA) [1]. Данная модель состоит из трех частей:

– событийная часть, которая подает в систему сигналы, исходящие от сторонних наблюдаемых систем;

– часть условия, которая является логическим блоком и при выполнении вызывает заранее запрограммированные действия;

– часть действий, которая состоит из различных операций по взаимодействию с наблюдаемой системой.

Механизм активных правил способствует не только быстрому реагированию, но и значительно снижает влияние на существующие прикладные системы. Функциональные возможности сервиса предназначены для решения следующих проблем: распределение потока информации между бизнес-доменами или целыми организациями, а также динамическое сотрудничество нескольких служб.

Данный сервис может извлекать значимые для наблюдаемой системы события и дает возможность динамического реагирования на бизнес-приложения или службы в реальном времени.

В рамках сервиса существует три группы репозиториев метаданных:

– глобальные репозитории, которые хранят информацию о всех наблюдаемых системах, событиях и активных правилах, пользователях и информацию конфигурации отдельных компонентов сервиса;

– оперативные хранилища компонентов, которые содержат информацию, необходимую для работы отдельно взятого компонента системы;

– репозиторий журнализации, необходимый хранения информации по работе сервиса.

На рисунке 1 представлена диаграмма вариантов использования сервиса потоковой обработки событий [2].

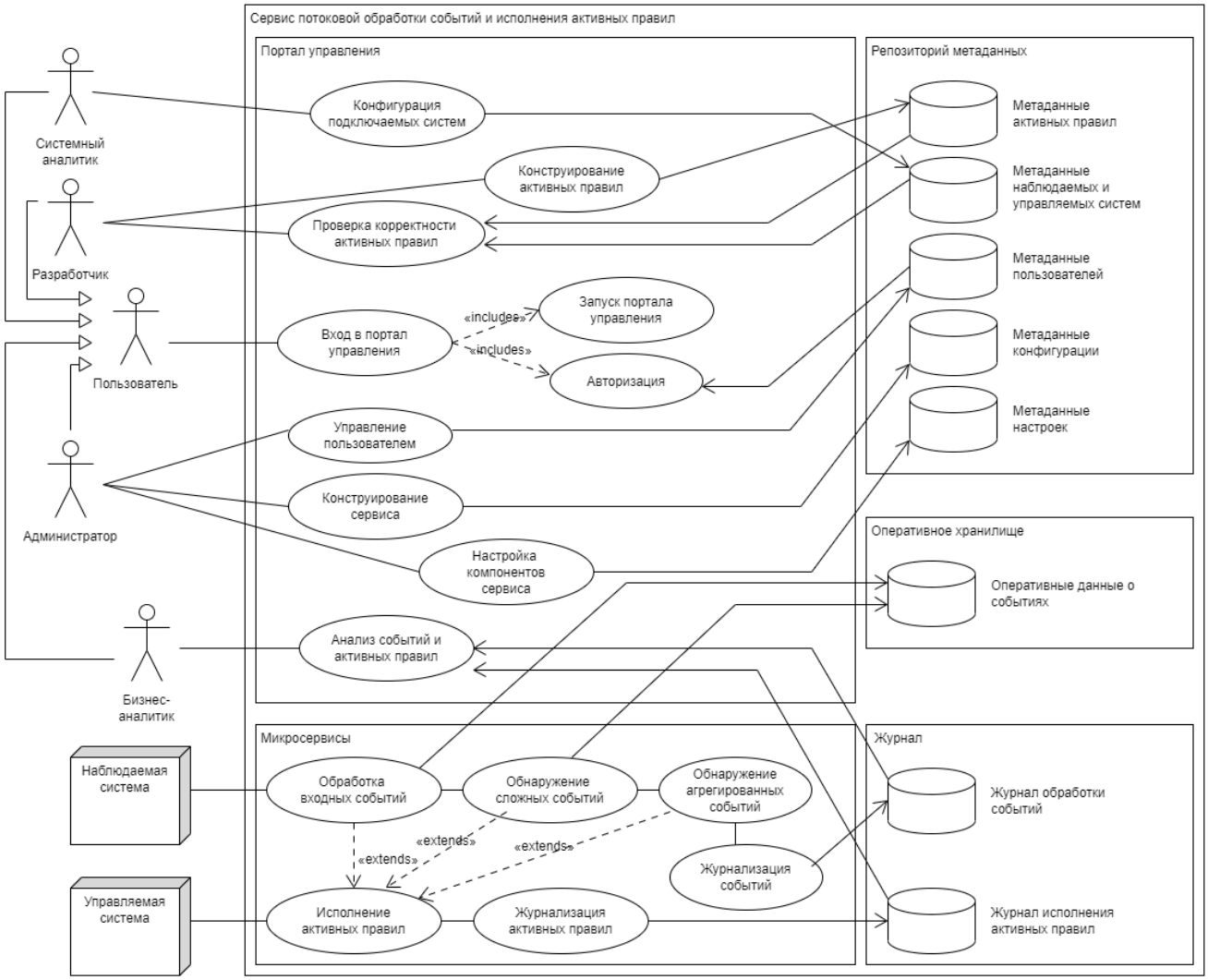

Рисунок 1 – Диаграмма вариантов использования сервиса потоковой обработки событий

Сервис потоковой обработки событий обладает следующими функциональными возможностями [3]:

– обработка входных событий от объектов мониторинга, обнаружение сложных и агрегированных событий на основе заданных шаблонов, доставка событий к соответствующим компонентам сервиса;

– создание активных правил через портал управления, настройка условий активации и входящих событий;

– управление созданными активными правилами, проверка их корректности, отключение и подключение к системам управления;

– вычисление условий активного правила и дальнейшее исполнение активных правил, в зависимости от поступивших событий;

– исполнение статических и динамических алгоритмов анализа активных правил и событий, поиск цикличности исполнения, выявление коллизий, моделирование новых событий на основе проанализированных данных;

– журнализация обрабатываемых событий и исполнения активных правил, последующее предоставление данных для анализа;

– поддержание актуальности метаданных во всех компонентах сервиса, синхронизация глобального и локальных репозиториев метаданных;

– масштабирование и настройка сервиса с учётом данных журнализации и результатов анализа.

Присутствует необходимость ведения учётных записей пользователей, их регистрация и аутентификация. Предоставление доступа к операциям сервиса защищено на основании пользовательских ролей.

1.1.2 Микросервисная событийно-ориентированная архитектура сервиса потоковой обработки событий

Архитектура сервиса потоковой обработки событий построена на основе микросервисов и основана на событийно-ориентированном подходе. Данная архитектура дает множества преимуществ, основными из которых являются: удобная и простая масштабируемость, вывод в реальном времени, отказоустойчивость и высокая доступность. Доставка событий в сервис производится посредством очередей и брокера сообщений, который занимается сбором и маршрутизацией сообщений на основе заранее предопределенной логики.

Архитектура позволяет увеличить автономность каждого элемента системы, тем самым отделяя логику независимых компонентов друг от друга. Брокер сообщений не анализирует передаваемые данные, он выступает в качестве маршрутизатора. Это позволяет настраивать взаимодействие между компонентами, написанными даже на разных языках программирования и работающими на разных платформах. Также благодаря этому повышается независимая масштабируемость микросервисов. Можно масштабировать

только те сервисы, на которые идет самая большая нагрузка. Выход из строя одного из микросервисов не скажется критично на работе всего сервиса.

Для взаимодействия с клиентскими приложениями для конфигурации сервиса и управления метаданными имеется REST-сервер. Данное программное средство может отправлять и получать сообщения от каждого микросервиса, а также управлять глобальным репозиторием метаданных.

Архитектура сервиса потоковой обработки событий представлена на рисунке 2.

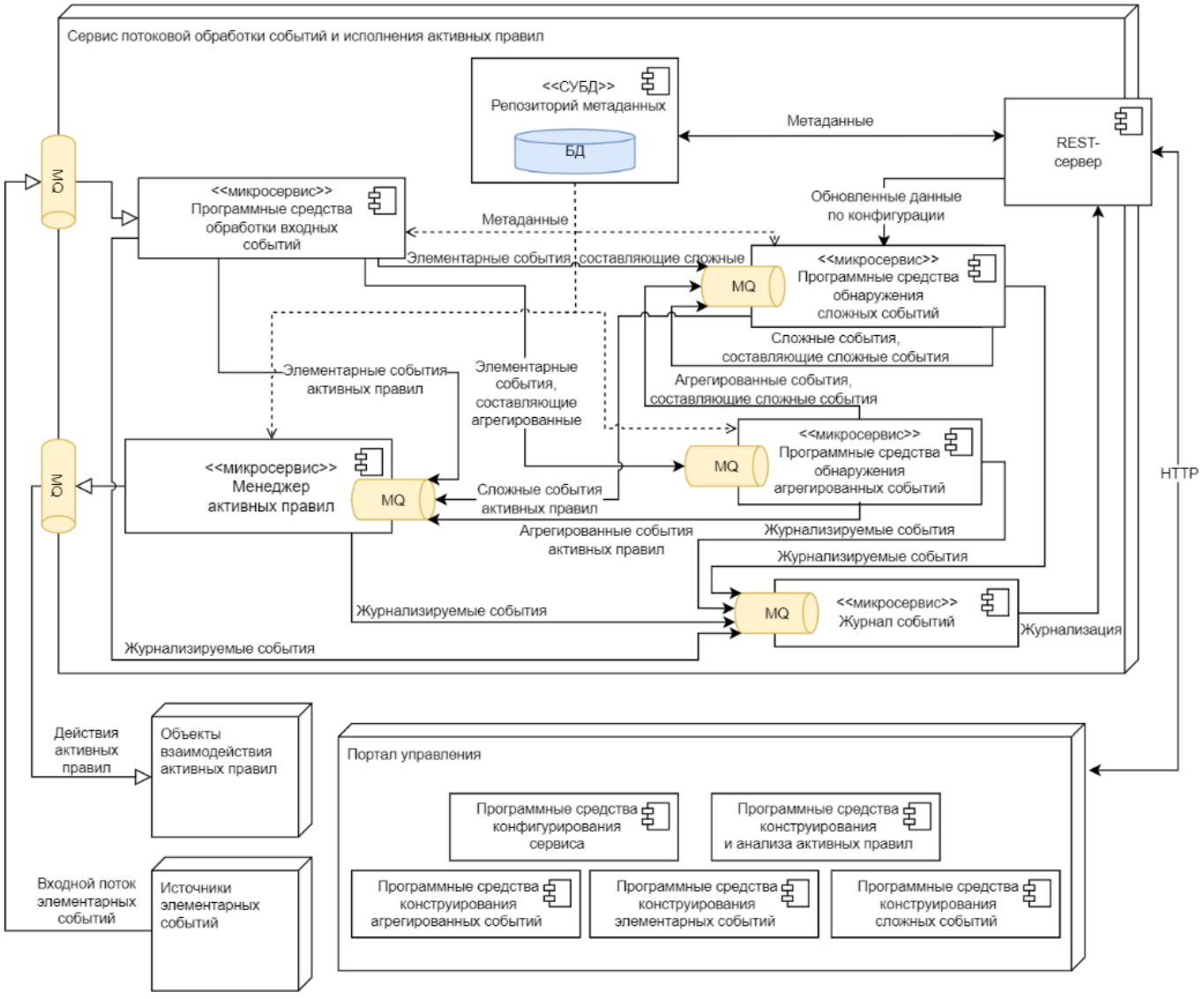

Рисунок 2 – Архитектура сервиса потоковой обработки событий

Центральный репозиторий необходим для хранения метаданных в процессе функционирования сервиса. Набор метаданных может включать в себя: спецификации событий и активных правил, информацию о

наблюдаемых системах и их объектах, информацию о системах управления, промежуточные результаты работы, правила обработки событий и правила формирования управляющих воздействий, результаты анализа событий и правил и многое другое [4].

Каждый микросервис реализует шаблон «Database Per Service» («База данных на сервис») [5]. Репозиторий в данном случае будет называться локальным. Это позволит повысить отказоустойчивость и автономность отдельно взятого сервиса. Задача синхронизации локальных репозиториев с глобальным решается по специальному сценарию, который снижает время ожидания запросов к метаданным без ущерба актуальности и минимальной задержкой в скорости работы сервиса.

1.1.3 Сценарий работы сервиса потоковой обработки событий

Создание различного рода алгоритмов, например в предприятии, позволяет удаленно контролировать процесс производства, заранее находить неисправности и собирать статистические данные по работе того или иного узла. Данный инструмент является эффективным средством управления жизненным циклом той или иной системы.

Обработка событий происходит по технологии ESP (Event Stream Processing) [6]. Она позволяет выявить часто встречаемые шаблоны поведения системы с целью обнаружения в них циклического повторения запрограммированного сценария, коллизий, иерархию событий и прочих проблемных случаев.

В ходе потоковой обработки событий сервис позволяет автоматически реагировать на запрограммированные ситуации, например, отказ оборудования или угроза пожарной безопасности.

Более широкое внедрение подобных технологий на предприятия позволило бы производить обслуживание аппаратуры «по состоянию», а не прибегать к жестким регламентационным срокам ремонтных работ, тем самым значительно сократив расходы.

1.2 Управление метаданными на сервисе потоковой обработки событий и исполнения активных правил

1.2.1 Состав метаданных

Основной набор метаданных представляет собой информацию, связанную с активными правилами, событиями и компонентами, от которых они поступают. Метаданные, связанные с активными правилами и событиями, располагаются в едином репозитории.

Присутствуют метаданные, описывающие конкретную систему, в которой происходит автоматизация процессов и управления жизненным циклом. К основным метаданным систем могут относиться:

– общие конфигурации реагирования на шаблоны поведений;

– уровень доступа пользователей по управлению и просмотру;

– иерархические связи с другими системами.

В каждую систему входит набор компонентов – источников, от которых поступают события. В состав метаданных компонентов можно отнести:

– перечень ролей, которые описывают функциональную возможность;

– уровень компонента, который характеризует важность и приоритет поступающих событий от заданного компонента;

– связь с перечнем систем, в которых происходит отслеживание событий;

– различные статистические данные.

События в сервисе могут быть представлены в виде следующих трех кортежей <ID, данные, временная метка>, идентификатор которого является уникальной меткой устройства, данные относятся к значениям наблюдения различных типов, а временная метка обозначает время, когда наблюдаемое устройство генерирует событие.

Сервис обрабатывает три различных вида событий: простые (атомарные), агрегированные и сложные (комплексные). Для

агрегированного события характерно наличие шаблона агрегации, а для сложного – перечень других событий, входящий в его состав.

Важным является предоставление ассоциаций между активными правилами и событиями. Имеется два типа ассоциаций: «От события к активному правилу» и «От активного правила к событию». Для распознавания алгоритмов взаимодействия сервису необходимо точно знать какое событие порождает активное правило или наоборот.

Как и в любом многопользовательском сервисе, данное программное обеспеченье обладает набором метаданных пользователей. К таким метаданным можно отнести: метаданные учетных записей и метаданные ролей доступа.

Важно заметить, что конфиденциальные данные пользователей, например пароль, необходимо хранить в зашифрованном виде. Шифрование может осуществляться при помощи криптографических хеш-функций формирования ключа. Хеш-функции позволяют безопасно хранить данные пользователей и авторизировать их во время взаимодействия с сервисом.

1.2.2 Жизненный цикл метаданных

Сервис имеет четыре основных уровня: уровень управления метаданными, уровень конструирования алгоритмов и систем взаимодействия, уровень обработки событий и исполнения активных правил, и уровень публикации/подписки бизнес-приложений. На каждом уровне метаданные играют свою ключевую роль.

Контроль и управление метаданными осуществляется через REST-сервер, который в свою очередь отвечает за своевременную передачу новых или обновленных метаданных микросервисам обработки событий и исполнения активных правил. REST-сервер имеет средства предоставления доступа к созданию новых метаданных, которые создаются при помощи администратора или автоматически в случае публикации/подписки новых бизнес-приложений.

Конструирование алгоритмов обработки событий и исполнения активных правил осуществляется при помощи портала управления. Он, в свою очередь, через протокол HTTP обращается к REST-серверу с метаданными, необходимыми для объявления новых событий и конструирования активных правил. На портале управления можно создавать как сложные, так и агрегированные события, объявлять новые правила и редактировать существующие, устанавливать различные ассоциации между событиями, правилами или системами в целом.

В данной архитектуре каждый микросервис выполняет свою задачу и, следовательно, работает со своим набором метаданных. При помощи брокера сообщений уникальный набор метаданных доставляется каждому элементу архитектуры. Микросервисы в ходе обработки событий и исполнения активных правил при необходимости отправляют сообщения бизнес-приложениям, посредством механизма подписчик/читатель. Команды управления могут использоваться в качестве автоматически генерируемых событий от самого сервиса и могут быть направлены на соответствующие датчики или устройства.

На завершающем этапе обработки событий метаданные журнализируются при помощи соответствующих сервисов. В дальнейшем это необходимо для предоставления статистических данный и выявления проблемных участков в существующих алгоритмах исполнения активных правил, например, обнаружение коллизий или зацикливаний.

1.2.3 Основные процессы управления метаданными

Взаимодействие сервиса потоковой обработки событий и исполнения активных правил с многопользовательским порталом управления организовано посредством REST API. Данная архитектура очень хорошо подходит для многопользовательской системы. В единой базе данных содержатся данные об аккаунтах пользователей и их роли.

В качестве способа аутентификации и передачи метаданных пользователя используется JWT [7]. JSON Web Token (JWT) – это открытый

стандарт (RFC 7519) для создания токенов доступа, основанный на формате JavaScript Object Notation (JSON). Токены создаются и подписываются секретным ключом на сервере, после чего передаются клиенту. Используются два токена: Access и Refresh.

Основные процессы управления метаданными представляют собой не только создание, редактирование и управления метаданных, а также определение наборов метаданных, необходимых каждому микросервису. Процессы управления имеют требования аутентификации и авторизации пользователя. Так, например, системный аналитик не должен иметь доступ к разработке новых активных правил и событий, генерируемых ими.

На рисунке 3 показаны основные процессы управления и роли доступа пользователей к данным процессам.

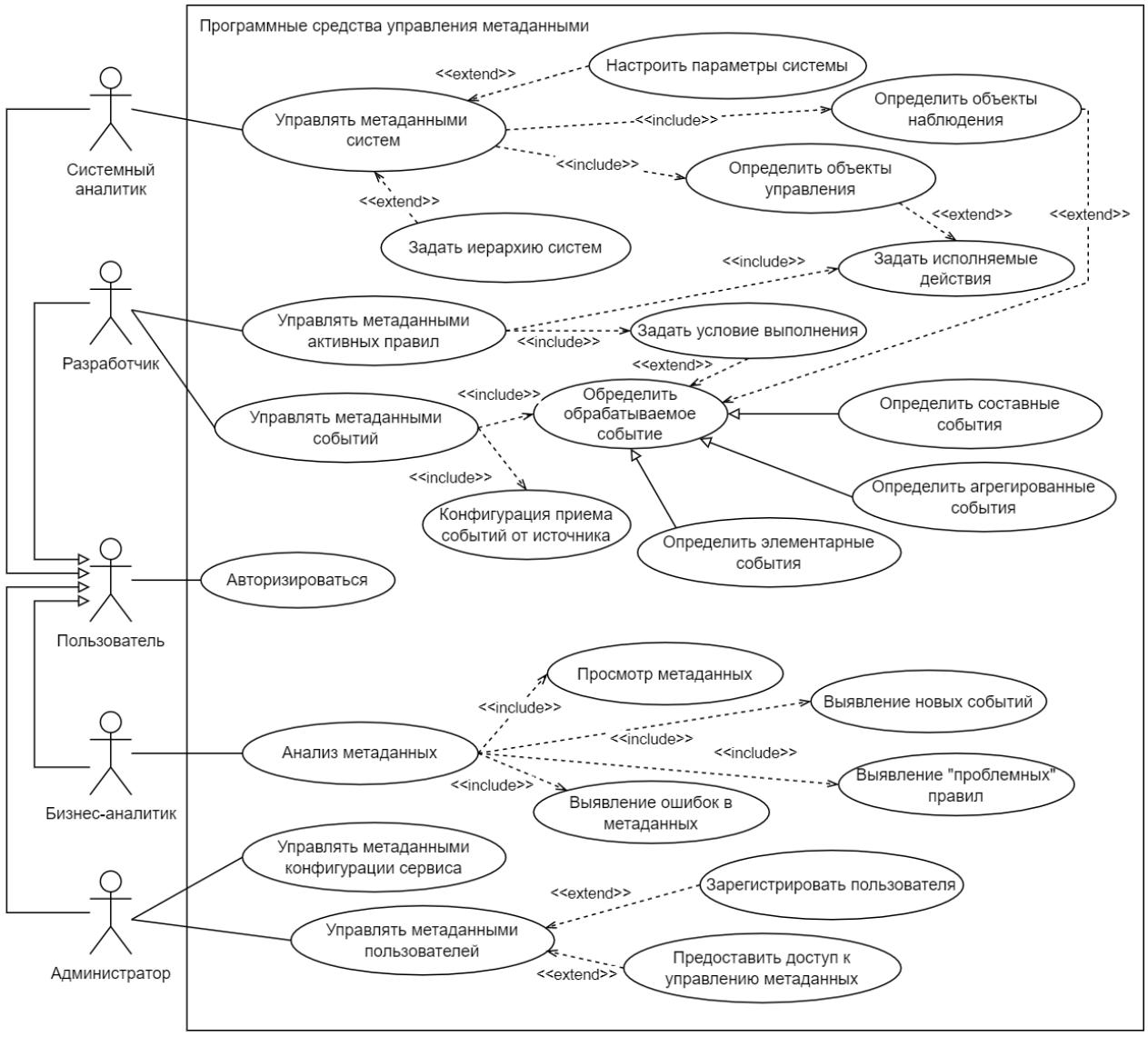

Рисунок 3 – Разграничение прав доступа пользователей

Существует четыре основные роли пользователей: разработчик, системный аналитик, бизнес-аналитик и администратор. Разработчик отвечает за весь процесс конструирования активных правил и событий, а также за управление всеми метаданными по ним. Системный аналитик конфигурирует наблюдаемые системы, настраивая их параметры и определяя объекты управления и наблюдения. Бизнес-аналитик работает со статическими данными, полученными от микросервисов, в которых содержится информация о произошедших событиях и активированных правилах. При помощи данной информации он выявляет ошибки, новые события и «проблемные» правила. Администратор, в свою очередь, работает с метаданными пользователей, регистрируя их и предоставляя возможность к различным интерфейсам управления. Вдобавок администратор обладает правами конфигурирования всех компонентов сервиса потоковой обработки событий и исполнения активных правил.

Все процессы управления происходят при помощи серверных программных средств. Под управлением метаданных подразумевается операции, основанные на CRUD (создание, чтение, обновление и удаление) запросах.

При создании и редактировании метаданных присутствуют алгоритмы первичной проверки данных. Создание одинаковых событий или активных правил недопустимо. Например, два одинаковых атомарных события могут негативно сказаться на агрегирующих запросах и в результате привести к нежелательным последствиям, так как данные, полученные от источника, будут дублироваться.

В результате использования механизмов управления метаданными сервис запускает алгоритмы обнаружения нарушений в логике исполнения активных правил. Так, если метаданные были изменены или удалены из системы, то как это повлияет на алгоритм и не приведет ли это к нарушению обработки событий и исполнению правил.

1.3 Постановка задачи на разработку серверных программных средств

1.3.1 Цели разработки и назначение программных средств

Программные средства управления необходимы для предоставления доступа пользователям к взаимодействию с метаданными, а также своевременного информирования микросервисов об создании, изменении или удалении метаданных.

Основными целями разработки являются:

– организация репозитория для хранения метаданных;

– создание механизмов регистрации, аутентификации и авторизации пользователей;

– разработка механизма предоставления доступа к управлению метаданными при помощи паттерна запрос-ответ;

– разраничение доступа к интерфейсам управления на основании ролей пользователей;

– разработка механизма взаимодействия с журнализацией метаданных;

– разработка механизма взаимодействия с микросервисами на основании паттернов подписчик/читатель.

В качестве серверных программных средств будут выступать четыре ключевых элемента: REST-сервер для непосредственного управления метаданными, Web-сервер, репозиторий для осуществления хранения, брокер сообщений для организации связи со сторонними модулями и микросервисами. Для клиентских приложений необходиму будет составить документацию, которая будет содержать перечень API взаимодействия и их краткие спецификации. В спецификации должны быть отражены данные, необходимые для осуществления запроса, а также уровень пользовательского доступа.

1.3.2 Роль и место серверных программных средств в процессе управления метаданными

Серверные программные средства играют ключевую роль в процессе обработки событий и исполнения активных правил. Именно они отвечают за механизмы управления метаданными, статические алгоритмы проверки исполнимости активных правил и обработки событий. В них содержатся конфигурации каждой наблюдаемой системы и наборы метаданных, необходимые всем микросервисам. Они гарантируют своевременную доставку актуальных метаданных на каждый элемент архитектуры и оповещение подключаемых бизнес-приложений. На рисунке 4 можно наблюдать взаимодействие компонентов архитектуры между собой.

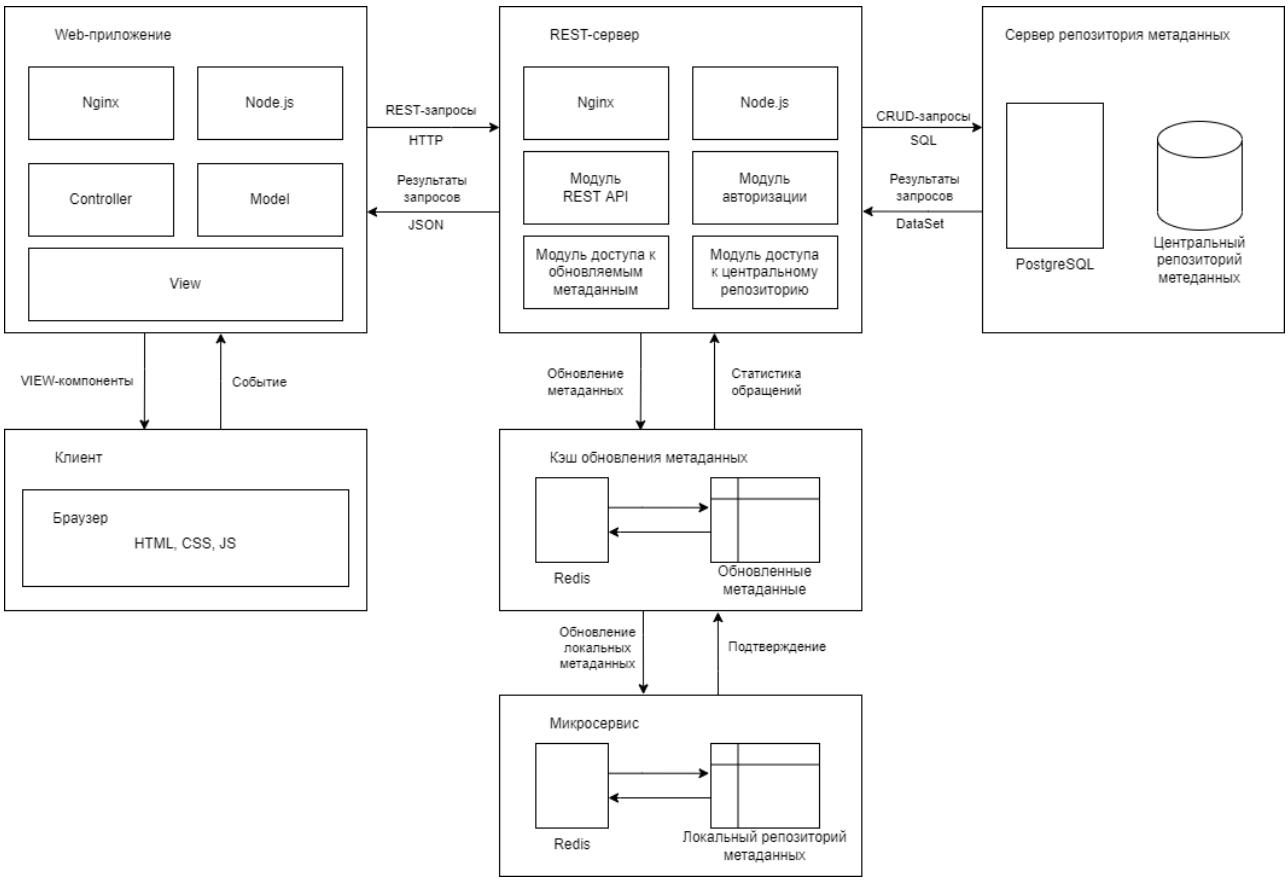

Рисунок 4 – Взаимодействие компонентов архитектуры сервиса потоковой обработки событий

Общение с клиентскими Web-приложениями происходит при помощи REST-запросов по протоколу HTTP. Передача данных осуществляется в формате JSON. При работе с метаданными REST-сервер автоматически

генерирует SQL запросы к серверу репозитория метаданных. Автоматическое оповещение микросервисов об обновлении метаданных происходит при помощи резидентной СУБД. Взаимодействие с микросервисами происходит по шаблону подписчик/читатель.

1.3.3 Перечень автоматизируемых функций серверных программных средств

Серверные программные средства должны обладать следующим перечнем автоматизируемых функций:

– автоматическая генерация запросов к репозиторию метаданных в зависимости от требований сервиса;

– контроль безопасности и предоставление доступа пользователей к механизмам управления метаданными;

– автоматическое оповещение бизнес-приложений об исполнении активных правил;

– автоматическое оповещение микросервисов об изменении, удалении или создании метаданных;

– контроль актуальности метаданных в локальных репозиториях каждого элемента архитектуры;

– журнализация операций управления метаданными;

– журнализация ошибок, вызванных во время эксплуатации.

В свою очередь, серверные программные средства должны обладать минимальным набором защиты от атак злонамеренных сторонних сервисов и программ, а также предотвращать несанкционированный доступ.

1.4 Обзор аналогичных работ в заданной предметной области

1.4.1 Отладчик активных правил DEAR и система визуализации исполнения активных правил Adela

DEAR является отладчиком для активных правил в объектно-ориентированном контексте [8]. Он реализован на основе EXACT, который

является менеджером правил, поддерживающим ECA правила в объектно-ориентированной системе баз данных ADAM. Данный отладчик позволяет отслеживать события и подсказывать пользователю о том, какие правила были запущены и что привело к запуску этих правил. Также, при возникновении события, он позволяет обнаружить параметры контекста, в котором возникло это событие.

Присутствует актуальная проблема отображения последовательности вызова активных правил. Зачастую недостаток отображения правил заключается в том, что система игнорирует тот факт, что правило должно конфликтовать с другими правилами во время их исполнения. Если присутствуют последовательно активируемые правила, то это может привести к незапланированному исходу. DEAR решил эту проблему путем отображения как цикла правил, так и цикла событий. В случае возникновения коллизий пользователь может сам определить, какие правила будут активироваться, т.е. система выдает перечень противоположных по бизнес-логике активных правил, которые потенциально будут вызваны после возникновения события. Также это позволило однозначно определять контекст вызванного активным правилом события или другого активного правила.

Явное определение набора коллизий одновременно инициированных правил отображается в дереве. Коэффициент правил, участвующих в коллизии полезен для рассмотрения правил, из-за которых могут возникать сложные взаимодействия. Программное средство предлагает несколько способов отображения данного дерева: отслеживание событий, отслеживание правил, а также отслеживание метаданных, определенных в ходе отладки.

У данного проекта есть ряд недостатков. Первый недостаток заключается в трудности отслеживания изменений данных в процессе активации цепочки активных правил, поскольку действия правил меняют их. В DEAR это не очень наглядно, так как в нем при визуализации правил не показывается, какие изменения произошли с данными. Невозможно определить, какие данные изменяются в любой момент исполнения активных

правил, а лишь известно какие данные будут изменены. Еще одна проблема связана с ограничениями целостности. Так как ограничение может быть не одиночным, а составным, то его визуализация становится сложной.

Попытка устранить недостатки DEAR была произведена при разработке системы Adela в 1994 году [9]. Данное программное средство обладает графическим интерфейсом. Оно позволяет отображать последовательность активаций правил, а также показывать последовательность возникновения и список вызванных событий. В приложении имеются: монитор базы данных, фильтр событий, графический менеджер и пользовательский механизм управления.

Монитор базы данных предназначен для показа текущего состояния метаданных. Фильтр событий получает информацию от монитора базы данных и обрабатывает его в тех случаях, которые может обработать графический менеджер. Графический менеджер получает информацию от фильтра и занимается необходимой анимацией. Пользовательский механизм предоставляет пользователю ряд команд для управления визуализацией.

Визуализация, реализованная в Adela, предусматривает два окна анимации: окно трассировки событий и правил и окно трассировки метаданных.

Окно трассировки событий и правил визуализирует их последовательность, а также состояние выполнения каждого правила при его активации (рисунок 5). События представляют собой закрашенные квадраты. Правила – эллипсы, содержащие три круга. Кружки используются для отображения, на каком этапе находится исполнение правила. Каждый элемент помечен идентификатором узла, рядом с которым располагается пояснительная информация: название элемента, уровень глубины дерева и количество раз, сколько это событие появлялось, или сколько раз исполнялось активное правило. Окно имеет возможность масштабировать все элементы.

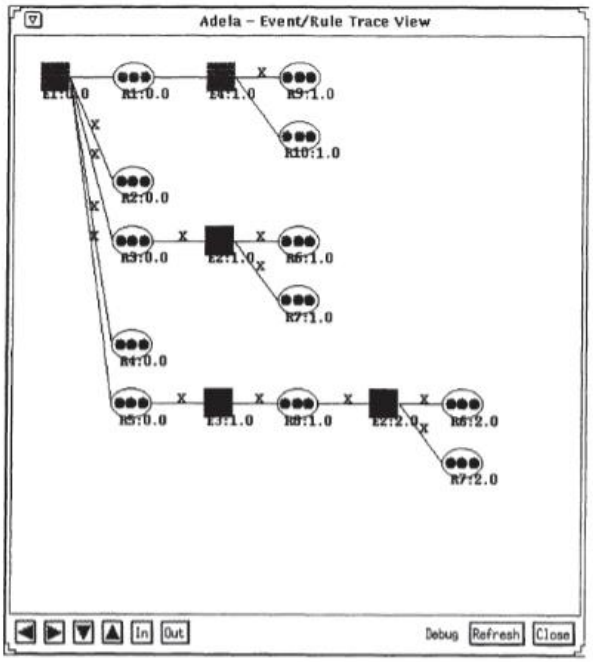

Рисунок 5 – Окно трассировки событий/правил

Окно данных визуализирует действие правил (рисунок 6). Правила представлены в виде сплошных черных эллипсов. Объекты визуализируются в виде сплошных черных прямоугольников. Связь между правилами не реализована полностью. Правила и объекты соединены только тогда, когда действие правила выполняет операцию над типом объекта. Изменения экземпляров объектов визуализируется в виде небольших кругов, которые имеют название типа объекта и идентификатор узла правила.

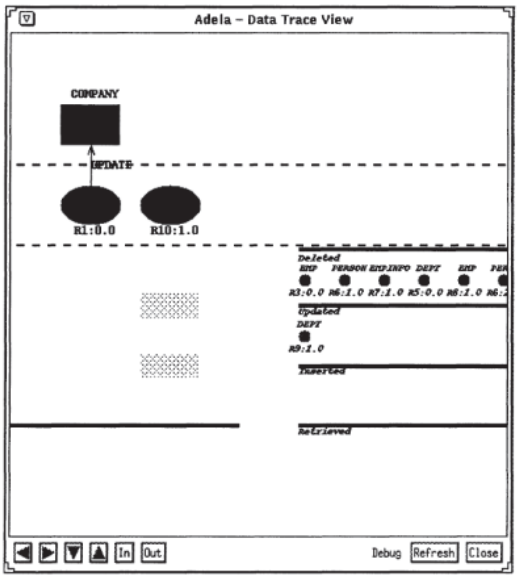

Рисунок 6 – Окно трассировки данных

Представленные графические окна отображают необходимую информацию для базового проектирования алгоритмов воздействия в системах обработки событий и исполнения активных правил. Adela была разработана в качестве прототипа на языке программирования C++, используя анимационную библиотеку Goofy.

Ряд преимуществ и недостатков отладчика активных правил DEAR и системы визуализации исполнения активных правил Adela будет учтен при разработке программного обеспечения.

1.4.2 Программное средство управления активными правилами и событиями IBM WebSphere ILOG JRules

Одним из современных решений реализации средства управления активными правилами и событиями на основе ECA модели является IBM WebSphere ILOG JRules [10]. В качестве ключевых нововведений можно отметить следующие механизмы: параметры контекста, связи и приоритеты.

Существует четыре контекста параметров: недавний, хронический, непрерывный и кумулятивный. Они были введены, чтобы обеспечить механизм захвата значимой семантики приложения и уменьшить пространство и вычислительные затраты для обнаружения составных событий. Контексты определяются с использованием понятий инициатора и терминатора для событий. Событие, которое инициирует возникновение составного события, называется инициатором составного события. Событие, которое завершает обнаружение составного события, обозначается как терминатор составного события. Например, составное событие (E1; E2; E3) имеет E1 в качестве инициатора и E3 в качестве терминатора.

В недавнем контексте используется самое последнее появление инициатора для любого элементарного события, которое запустило обнаружение сложного. Когда оно обнаруживается, то все события, которые не могут быть его инициаторами в будущем, удаляются. Инициатор события будет продолжать инициировать новые события до тех пор, пока не появится новый инициатор.

В контексте хроники пара инициатор-терминатор уникальна для события. Самый старый инициатор соединяется с самым старым терминатором для каждого события. После обнаружения сложного события элементарные, использованные для его обнаружения, удаляются. Событие может быть использовано не более одного раза для вычисления параметров составного события.

В непрерывном контексте каждый инициатор запускает отдельное обнаружение сложного события. Пока событие не будет найдено, инициатор не будет отброшен.

В кумулятивном контексте все элементарные события накапливаются как предположительные экземпляры составного события до тех пор, пока составное событие не будет обнаружено. После обнаружения все элементарные события, отбрасываются.

Режимы связи были введены, чтобы указать относительный момент времени, когда должны проверяться условия активного правила после возникновения события. Существует три режима связи: немедленный, отложенный, отдельный.

Немедленный: при обнаружении события транзакция приостанавливается, и условие, связанное с событием, оценивается немедленно. Выполнение инициирующей транзакции приостанавливается.

Отложенный: запускающая транзакция продолжается после обнаружения события. Оценка условий и выполнение действий выполняются в конце инициирующей транзакции перед ее фиксацией.

Отделенный: оценка состояния и выполнение действия выполняются в отдельной транзакции от инициирующей транзакции.

В дополнение к контексту параметра, каждому правилу также присваивается приоритет. Приоритетом правила по умолчанию является приоритет 1. Приоритет увеличивается с увеличением числового значения. То есть 2 имеет более высокий приоритет, чем 1, 3 имеет более высокий приоритет, чем 2, и так далее. Правила с одинаковым приоритетом выполняются одновременно, а правила с более высоким приоритетом всегда

выполняются перед правилами с более низким приоритетом. Вполне возможно, что правило вызывает события, которые, в свою очередь, могут вызвать дополнительные правила. Это приводит к каскадному выполнению правила. Кроме того, правила могут быть заданы либо в режиме немедленной связи, либо в режиме отложенной связи. При планировании правила для выполнения необходимо учитывать как приоритет, так и режим связи правила.

Сами правила и все соответствующие метаданные сохраняются в структуре, называемой «репозиторий» (рисунок 7). Хранилище является центральным местом для метаданных правил. Метаданные организованы в пакеты, которые могут быть вложены на любую произвольную глубину, что позволяет структурировать проект способом, который ближе к логике приложения. Кроме того, проект – это набор ссылок на пакеты и библиотеки репозитория. Библиотеки содержат данные, которые используются во всех бизнес-правилах, языке правил и шаблонах правил.

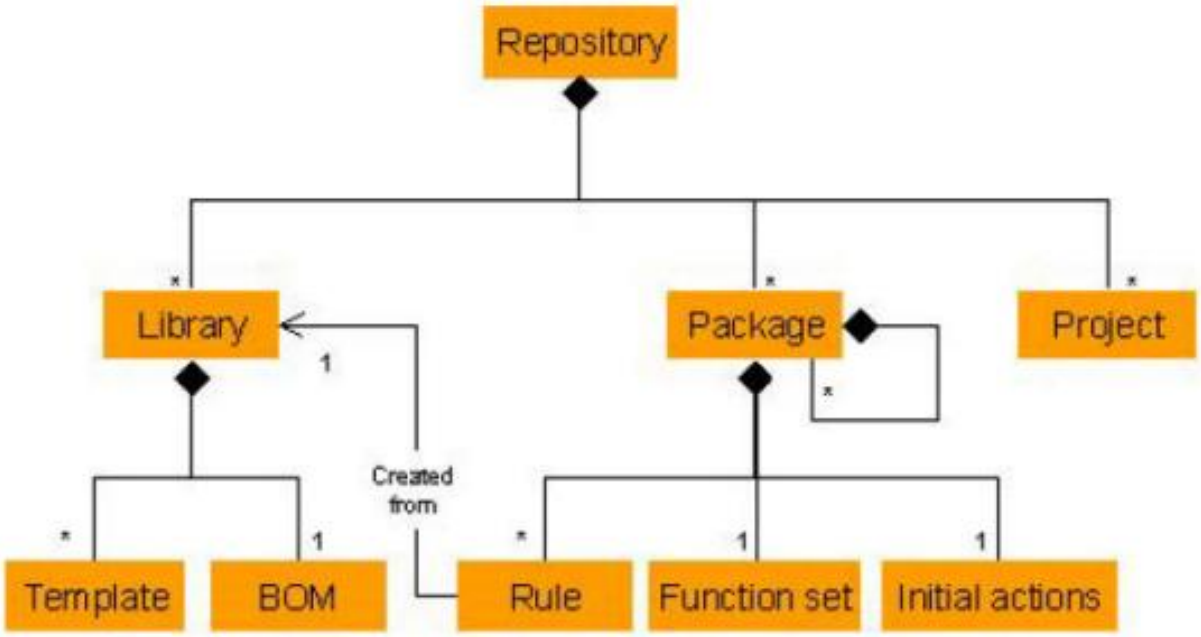

Рисунок 7 – «Репозиторий» правил и метаданных системы

Когда механизм правил ILOG интегрирован в приложение, частью его функции является мониторинг состояния различных объектов системы. Это необходимо, поскольку правила обычно ссылаются на объекты системы в

части условия их логики. Рабочей памятью является место, где ILOG JRules хранит ссылки на все объекты, с которыми он работает в данный момент. Правила сгруппированы в наборы. Правила в наборе и объекты приложения, на которые они ссылаются, связаны тем, что называется контекстом. То есть контекст связывает правила с рабочей памятью и реализует механизм активных правил, который управляет отношениями между ними.

1.4.3 Средство оптимизированной обработки и удобного операционного и стратегического анализа данных Nextbi Аналитика

Данное программное средство предназначено для отказоустойчивой обработки потоков событий и автоматического реагирования на запрограммированные ситуации.

Информационная система обработки событий в Nextbi построена на технологии Event Stream Processing (ESP). Технология позволяет обнаружить особо значимые шаблоны поведения наблюдаемых систем, корреляцию, иерархию событий и многое другое. ESP в Nextbi применяется для ряда целей: оценка исправности и производительности производственного оборудования, анализ активностей на биржевых торговых площадках, выявления мошеннических действий в финансовом секторе, аналитика различного рода информации, поступающих с датчиков и сенсоров. Также данное программное средство позволяет одновременно собирать анализировать маркетинговые показатели множества объектов: объем продаж, стоимость одного нового клиента, отток клиентов, прибыль и т.п.

Сервис обладает графическим редактором активных правил (рисунок 8). Редактирование происходит методами визуального программирования. У пользователя имеется возможность выбора обрабатываемого события и его контекста, а также условий и результата выполнения. Соединение правил с условиями происходит при помощи протягивания линии ассоциации между ними.

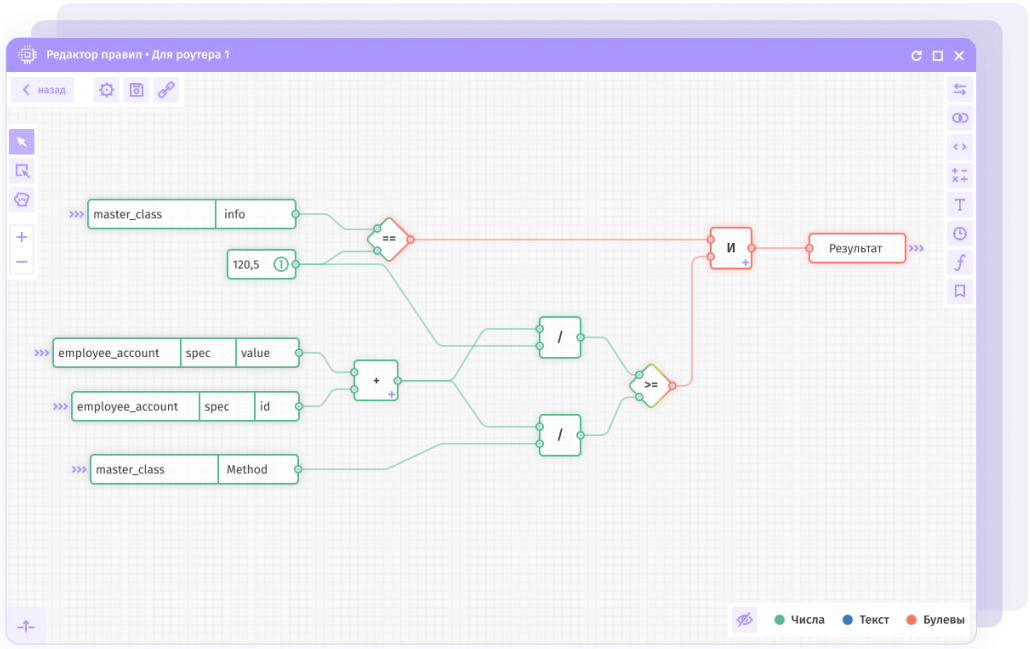

Рисунок 8 – Графический редактор активных правил Nextbi

Такой подход является абсолютно понятным для любого пользователя и не требует специфических знаний языков программирования.

Еще одним отличительным преимуществом от ранее рассмотренных работ является наличие развитой системы визуального проектирования алгоритмов обработки событий (рисунок 9) и автоматизированного мониторинга работы компонентов наблюдаемых систем (рисунок 10).

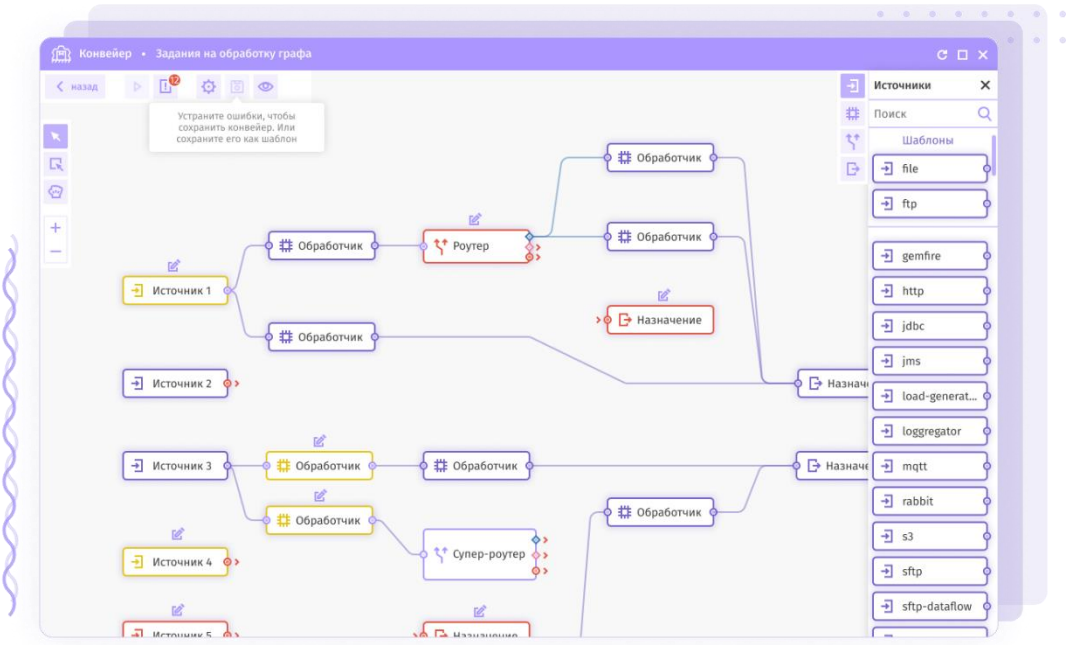

Рисунок 9 – Средство визуального проектирования алгоритмов обработки событий

Сервис позволяет настраивать обработчики событий прямо через графический редактор. События и обработчики отображаются в виде графа, ребра которого отображают взаимосвязь элементов.

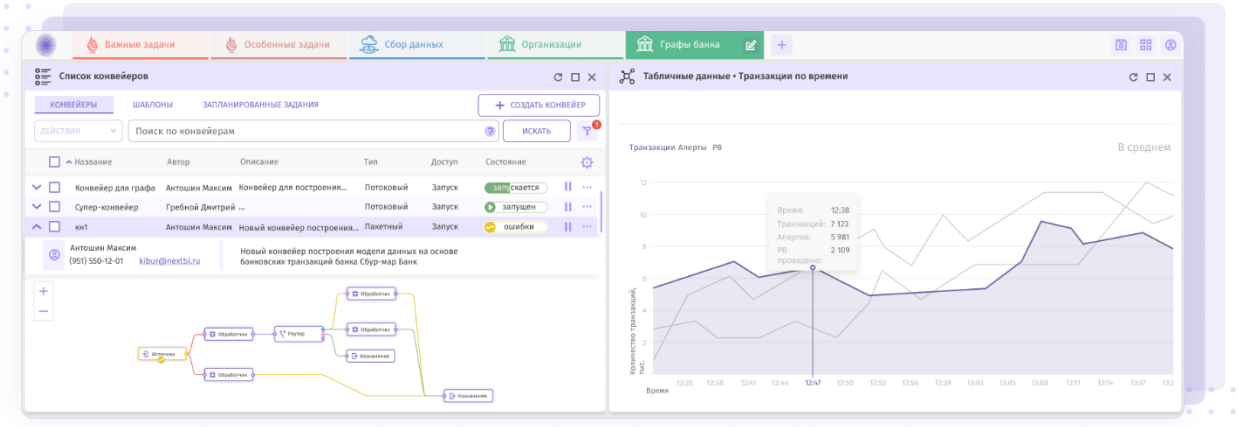

Рисунок 10 – Средство мониторинга работы компонентов наблюдаемых систем

Инструменты для визуального мониторинга работы облегчают работу отслеживания большого потока обрабатываемых данных. Также они сигнализируют о реагировании на внештатные ситуации, например, отказ рабочего узла, повышение нагрузки на него или выход из строя отдельного компонента.

1.4.4 Результаты обзора аналогичных работ в заданной предметной области

В результате обзора можно выделить ряд особенностей, которыми также должны обладать серверные программные средства управления метаданными для сервиса потоковой обработки событий:

– предоставление возможности пользователю определять, какие правила активируются и становятся причиной активации других правил, а также возможность обнаружить контекст, который инициировал события;

– статические алгоритмы явного определения набора коллизий активных правил;

– предоставление метаданных клиентским приложениям для визуализации активных правил в виде сетей петри и древовидного графа;

– предоставление метаданных клиентским приложениям о том, какие правила срабатывают, а также последствия этих срабатываний;

– дополнительные метаданные, для удобства организации систем: параметры контекста, различные виды связей между событиями и приоритеты исполнения активных правил.

Данные достоинства в дальнейшем будут учтены при разработке программного обеспечения.

1.5 Планирование процесса разработки

1.5.1 Планирование процесса разработки и оценка длительности выполнения задач

Планирование процесса разработки является первоначальной частью создания программного продукта. Оно включает в себя определение основных этапов работ, расчёт средств и времени, необходимых для реализации поставленных задач.

Исходя из анализа предметной области были определены основные задачи, необходимые для разработки серверных программных средств. В таблице 1 представлено описание задач проекта, последовательности и длительности их выполнения. Выбрана шестидневная рабочая неделя.

Таблица 1 – Задачи проекта

№	Задача	Тип	Длительность (дни)	Предшественники
1	2	3	4	5
1	Начало проекта	Веха	0	-
2	Анализ предметной области и постановка задачи на разработку	Фаза	11	-
3	Изучение работ-аналогов	Задача	3	1
4	Анализ состава и жизненного цикла метаданных	Задача	2	3
5	Анализ процессов управления метаданными	Задача	2	4
6	Определение перечня автоматизируемых функций	Задача	2	5
7	Планирование процесса разработки	Задача	2	6

Продолжение таблицы 1

1	2	3	4	5
8	Анализ предметной области и постановка задачи завершены	Веха	0	7
9	Разработка требований к серверным программным средствам управления метаданными	Фаза	7	2
10	Разработка функциональных требований	Задача	3	8
11	Разработка требований к репозиторию	Задача	2	10
12	Выбор технологий и разработка архитектуры ПС	Задача	1	11
13	Определение требований к аппаратно-программному обеспечению	Задача	1	12
14	Разработка требований завершена	Веха	0	13
15	Разработка серверных программных средств	Фаза	28	9
16	Разработка архитектуры развертывания ПС	Задача	2	13
17	Разработка ПС взаимодействия с базой метаданных	Задача	3	16
18	Разработка ПС аутентификации пользователей	Задача	4	17
19	Разработка ПС управления метаданными	Задача	6	18
20	Разработка ПС анализа корректности метаданных	Задача	3	19
21	Разработка ПС конфигурирования и настройки компонентов	Задача	3	20
22	Разработка ПС взаимодействия с клиентскими приложениями	Задача	3	21
23	Разработка ПС доставки метаданных к микросервисам	Задача	2	22
24	Разработка ПС взаимодействия со средствами журнализации	Задача	2	23
25	Разработка серверных программных средств завершена	Веха	0	24
26	Контроль качества серверных программных средств	Фаза	6	15
27	Планирование процесса контроля качества	Задача	2	24
28	Тестирование серверных программных средств	Задача	4	27
29	Оценка качества программного кода	Задача	2	27
30	Контроль качества завершен	Веха	0	28;29
31	Завершение работы над проектом	Фаза	5	26
32	Оформление документации	Задача	4	30
33	Сдача проекта	Задача	1	32
34	Проект завершен	Веха	0	33
35	Конец проекта	Веха	0	34

На следующем этапе необходимо проанализировать ресурсы проекта и их распределение по каждой задаче.

1.5.2 Анализ ресурсов разработки

После определения основных задач были оценены ресурсы проекта. Они включают в себя все необходимое для завершения процесса разработки, например оборудование, материалы и трудовые затраты. Перечень ресурсов представлен в таблице 2.

Таблица 2 – Ресурсы проекта

Название ресурса	Тип ресурса	Затраты		Начисление
		Стандартная ставка	Ставка сверхурочных	
Программист	Трудовой	300,00 р/ч	400,00 р/ч	Пропорциональное
Аналитик	Трудовой	150,00 р/ч	200,00 р/ч	Пропорциональное
Тестировщик	Трудовой	200,00 р/ч	250,00 р/ч	Пропорциональное
Интернет	Затраты	-	-	Пропорциональное
Лицензия ПС	Затраты	-	-	Пропорциональное

Информация об распределении ресурсов по каждой задаче представлена в таблице 3.

Таблица 3 – Распределение ресурсов по задачам

№	Задача	Ресурсы	Единицы (затраты)
1	2	3	4
1	Начало проекта	-	-
2	Анализ предметной области и постановка задачи на разработку	-	-
3	Изучение работ-аналогов	Аналитик Интернет	100% 400р.
4	Анализ состава и жизненного цикла метаданных	Аналитик	100%
5	Анализ процессов управления метаданными	Аналитик	100%
6	Определение перечня автоматизируемых функций	Аналитик	100%
7	Планирование процесса разработки	Аналитик Лицензия ПС	100% 1,000р
8	Анализ предметной области и постановка задачи завершены	-	-
9	Разработка требований к серверным программным средствам управления метаданными	-	-

Продолжение таблицы 3

1	2	3	4
10	Разработка функциональных требований	Программист	100%
11	Разработка требований к репозиторию	Программист	100%
12	Выбор технологий и разработка архитектуры ПС	Программист	100%
13	Определение требований к аппаратно-программному обеспечению	Программист	100%
14	Разработка требований завершена	-	-
15	Разработка серверных программных средств	-	-
16	Разработка архитектуры развертывания ПС	Программист Лицензия ПС	100% 800р
17	Разработка ПС взаимодействия с базой метаданных	Программист	100%
18	Разработка ПС аутентификации пользователей	Программист	100%
19	Разработка ПС управления метаданными	Программист	100%
20	Разработка ПС анализа корректности метаданных	Программист	100%
21	Разработка ПС конфигурирования и настройки компонентов	Программист	100%
22	Разработка ПС взаимодействия с клиентскими приложениями	Программист	100%
23	Разработка ПС доставки метаданных к микросервисам	Программист	100%
24	Разработка ПС взаимодействия со средствами журнализации	Программист	100%
25	Разработка серверных программных средств завершена	-	-
26	Контроль качества серверных программных средств	-	-
27	Планирование процесса контроля качества	Тестировщик	100%
28	Тестирование серверных программных средств	Тестировщик Лицензия ПС	100% 2,000р
29	Оценка качества программного кода	Программист Лицензия ПС	100% 500р
30	Контроль качества завершен	-	-
31	Завершение работы над проектом	-	-
32	Оформление документации	Программист Лицензия ПС Интернет	100% 1,500р 400р
33	Сдача проекта	Программист	100%
34	Проект завершен	-	-
35	Конец проекта	-	-

Одним из лучших средств планирования и анализа временных и финансовых затрат разработки является программное обеспеченье Microsoft Project. Для планирования процесса разработки использовалась версия Microsoft Project 2007 года. При помощи данного средства на рисунке 11 представлен план разработки в виде диаграммы Ганта. Диаграмма Ганта состоит из полос, ориентированных вдоль оси времени. Каждая полоса представляет собой отдельную задачу в составе проекта. Края полосы представляют собой моменты начала и завершения работ.

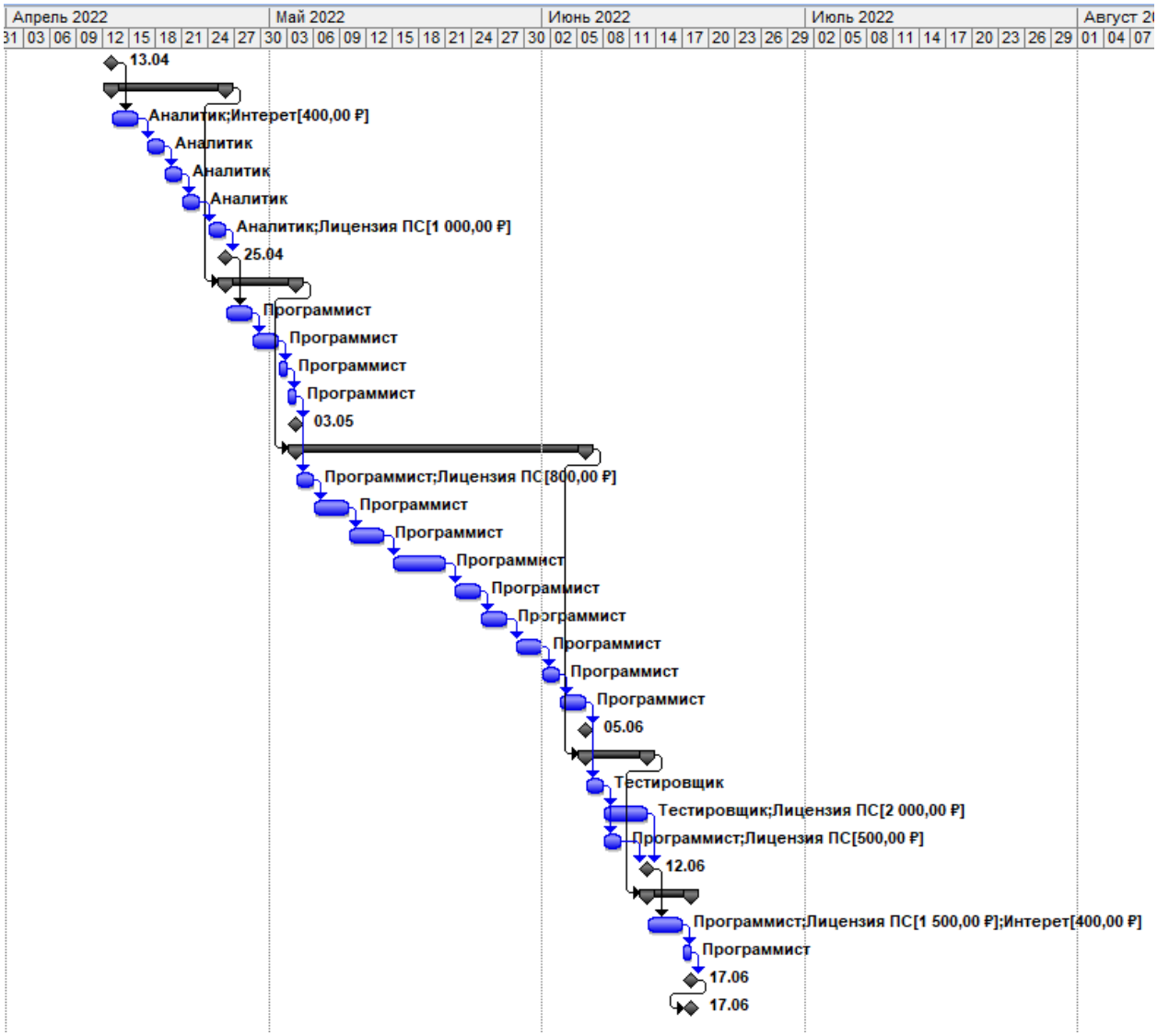

Рисунок 11 – Диаграмма Ганта

На данной диаграмме были отображены необходимые ресурсы, основные этапы разработки, основные задачи и их последовательность выполнения, а также сроки выполнения каждого этапа. Была выбрана шестидневная рабочая неделя. Выполнение задач имеет последовательный характер, так как трудовые ресурсы ограничены.

После обозначения основных задач и распределения ресурсов по ним были составлены: отчеты «Обзор затрат» (рисунок 12), который показывает фактические затраты на реализацию проекта, и «Обзор ресурсов» (рисунок 13), отображающий необходимые трудозатраты для осуществления разработки.

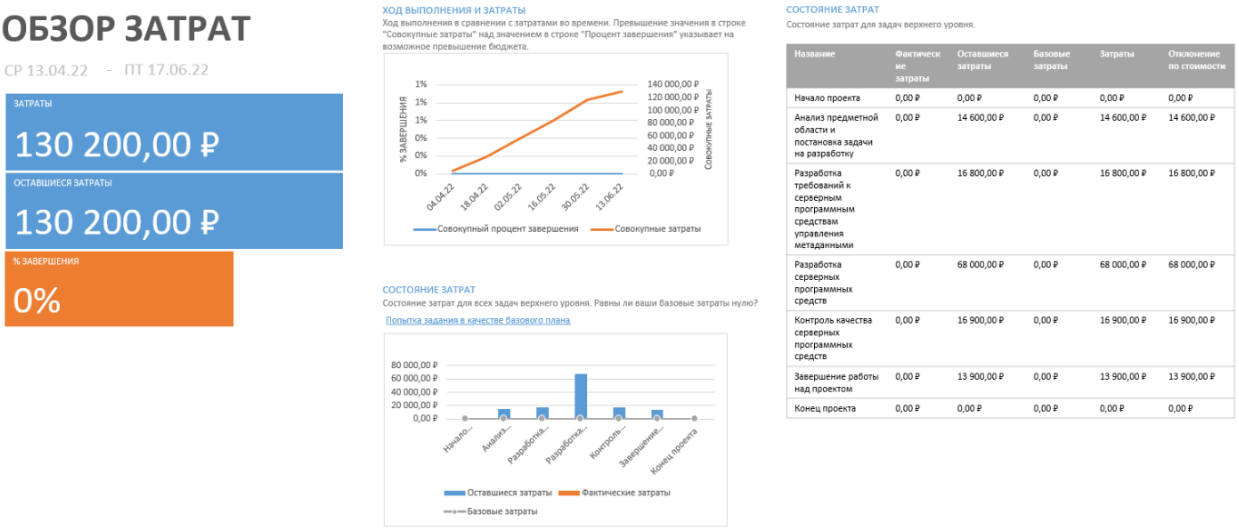

Рисунок 12 – Отчет «Обзор затрат»

В результате составления отчета «Обзор затрат» было подсчитано, что общая сумма, необходимая для разработки серверных программных средств, составляет 130200 рублей.

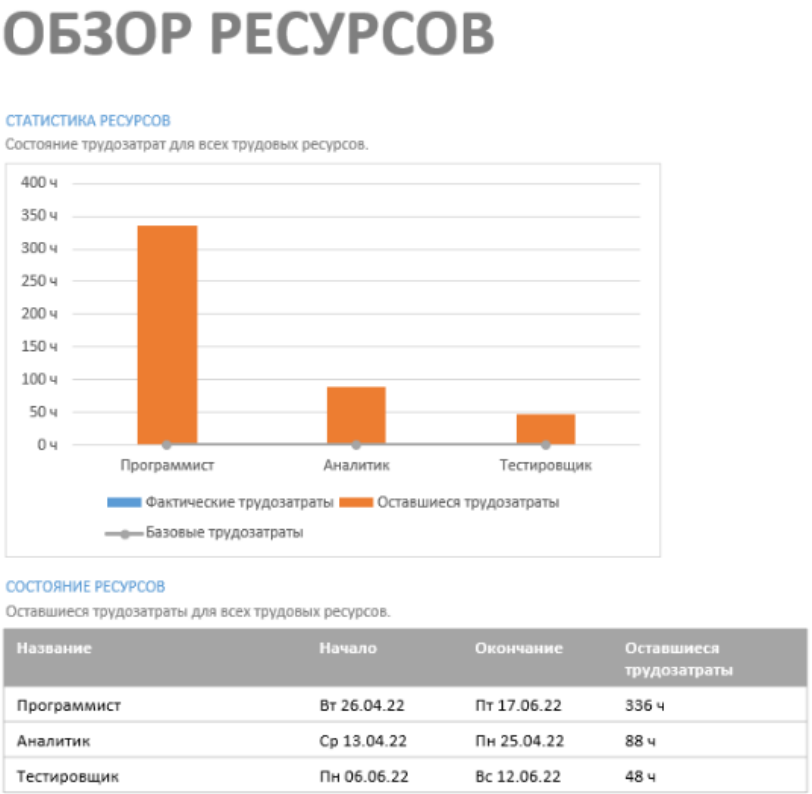

Рисунок 13 – Отчет «Обзор ресурсов»

Срок выполнения всех задач, необходимых для разработки, с учетом выходных дней составил 56 дня.

2 Проектирование и разработка требований к серверным программным средствам управления метаданными

2.1 Функциональные требования

Помимо базовых механизмов управления серверные программные средства должны обладать следующими возможностями:

– взаимодействовать с клиентскими приложениями и с репозиторием метаданных;

– выполнять алгоритмы по обнаружению нарушений;

– контролировать актуальность метаданных;

– взаимодействовать с микросервисами по шаблону подписчик/читатель и оповещать их об обновленных и новых метаданных;

– обрабатывать данные журнализации.

Ключевым звеном является REST-сервер, который будет удовлетворять данные требования. Сервер должен иметь возможность работы с репозиторием метаданных при помощи ORM. Это позволит автоматически генерировать SQL запросы. Представления таблиц БД необходимо сделать при помощи конкретных моделей-классов, с указанием информации о них: название, атрибуты, типы данных, PK, FK и т.д. Для оптимизации запросов необходимо реализовать представления (View) средствами ORM.

Взаимодействие с микросервисами необходимо организовать при помощи брокера сообщений. Заранее должны быть определены заголовки и содержания сообщений.

Взаимодействие с клиентскими приложениями должно происходить по HTTP протоколу. Предусмотреть защиту от несанкционированного доступа к механизмам управления.

На основании вышеупомянутых требований была разработана диаграмма вариантов использования (рисунок 14).

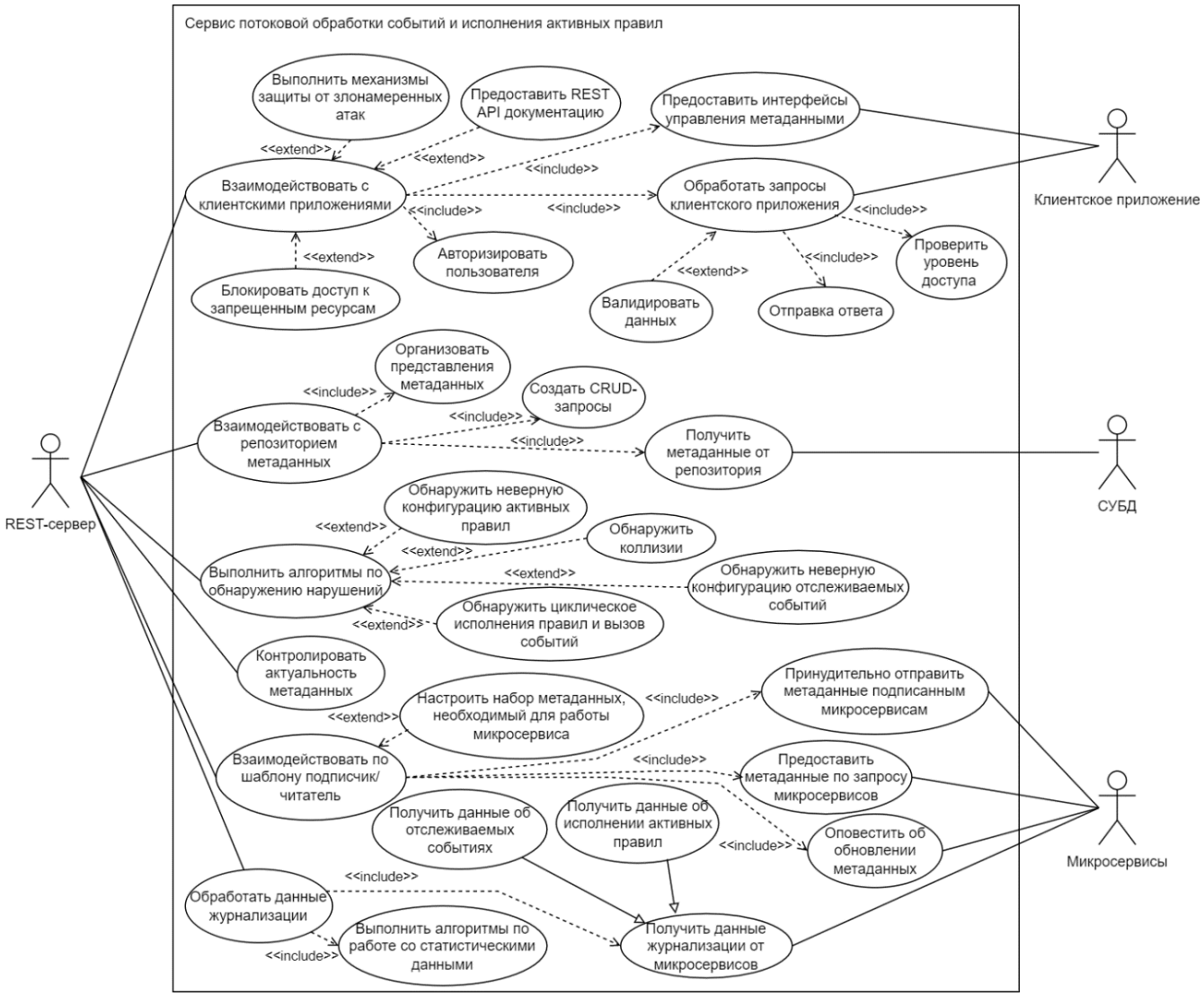

Рисунок 14 – Функциональные возможности REST-сервера

Доступ пользователей к определенному функционалу сервиса будет осуществляться при помощи токенов доступа – JSON Web Token (JWT). Будет присутствовать Access и Refresh токены. Access токен необходим для авторизации пользователя на REST сервере. Каждый JWT в целях безопасности имеет определенный срок жизни. Для Access токена время жизни составит 120 минут. Refresh токен – специальный JWT, при помощи которого автоматически происходит обновление Access токена. Благодаря этому пользователю нет необходимости постоянно вводить учетные данные и авторизироваться на портале управления повторно. Refresh токен будет иметь больший срок жизни – 14 дней. Если пользователь не будет заходить в систему дольше указанного периода, то потребуется повторная авторизация, так как жизненный цикл Refresh токена закончится.

2.2 Выявление требований и моделирование репозитория метаданных для сервиса потоковой обработки событий

2.2.1 Концептуальное моделирование

Репозиторий должен обладать всеми необходимыми сущностями для хранения метаданных сервиса потоковой обработки событий.

Для моделирования применялась нотация «сущность-связь», а конкретно нотация Мартина. В данной нотации атрибуты записываются внутри диаграммы сущности. Отношения записываются в виде текста над стрелкой. Нотация предлагает четыре вида стрелок, которые отображают арность отношений.

Моделирование репозитория проводилось в среде case-средства Visual Paradigm 16.2 [11]. Данное программное средство имеет интуитивно понятный и удобный интерфейс. Оно позволяет осуществлять моделирование в три уровня: концептуальный, логический и физический.

Концептуальное моделирование репозитория метаданных выполнялось на основе результатов анализа предметной области. Данный вид моделирования затрагивает смысловую структуру рассматриваемой предметной области, и разрабатываемая модель является начальным прототипом репозитория метаданных.

Данный этап модулирования включал в себя следующие действия:

1) выделение сущностей, соответствующих классам объектов метаданных;

2) выделение бинарных отношений между сущностями (отношения наследования, отношения ассоциации 1:M, M:M), соответствующих отношениям между классами объектов метаданных;

3) определение атрибутов сущностей, соответствующих свойствам классов объектов метаданных.

На рисунке 15 представлена диаграмма «сущность-связь» концептуальной модели репозитория метаданных.

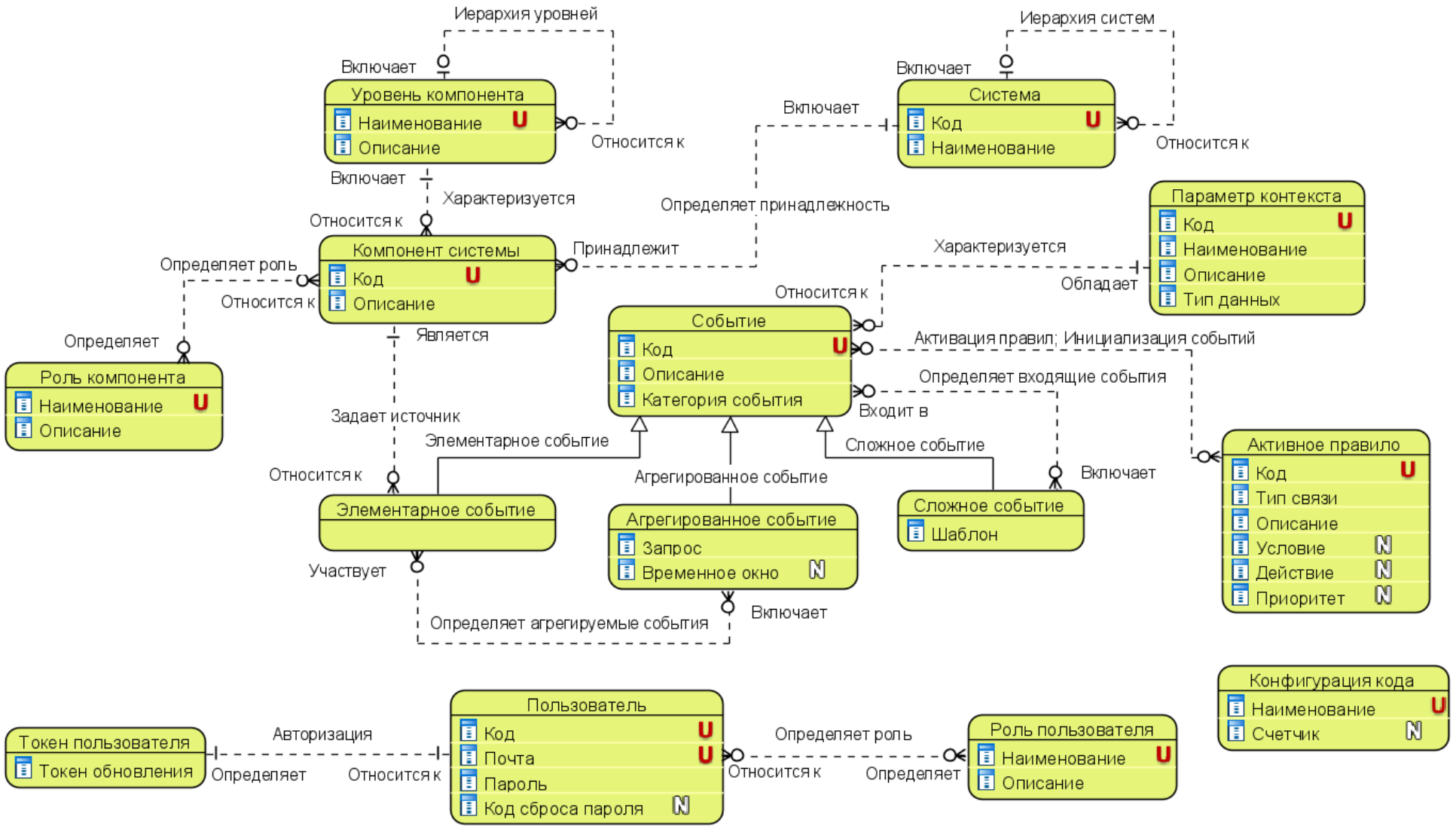

Рисунок 15 – Концептуальная модель репозитория метаданных

На дальнейшем этапе необходимо было осуществить логическое моделирование.

2.2.2 Логическое моделирование

Логическое моделирование реляционной базы данных репозитория метаданных выполнялось на основе результатов концептуального моделирования и включало следующие действия:

1) отображение сущностей, соответствующих классам объектов метаданных, в отношения реляционной схемы базы данных;

2) отображение бинарных отношений между сущностями (отношения наследования, отношения ассоциации 1:M, M:M), соответствующих отношениям между классами объектов метаданных, в бинарные связи между отношениями реляционной схемы базы данных;

3) отображение атрибутов сущностей, соответствующих свойствам объектов метаданных, в атрибуты отношений реляционной схемы БД.

При логическом моделировании применялись UUID версии 4 в качестве первичного ключа. Основными достоинствами являются:

– UUID гарантирует независимость, и программные средства могут быть перенесены между различными базами данных без каких либо конфликтов;

– возможность удобного и безопасного разделения и слияния баз данных, так как UUID обеспечивает «уникальность» не только внутри таблиц, но и внутри разных БД;

– отличная поддержка ORM и возможность генерации в автономном режиме.

Четвертая версия UUID использует генератор псевдослучайных чисел для создания идентификатора, в котором каждый бит строки генерируется полностью «случайным образом».

На рисунке 16 представлена диаграмма «сущность-связь» логической модели репозитория метаданных.

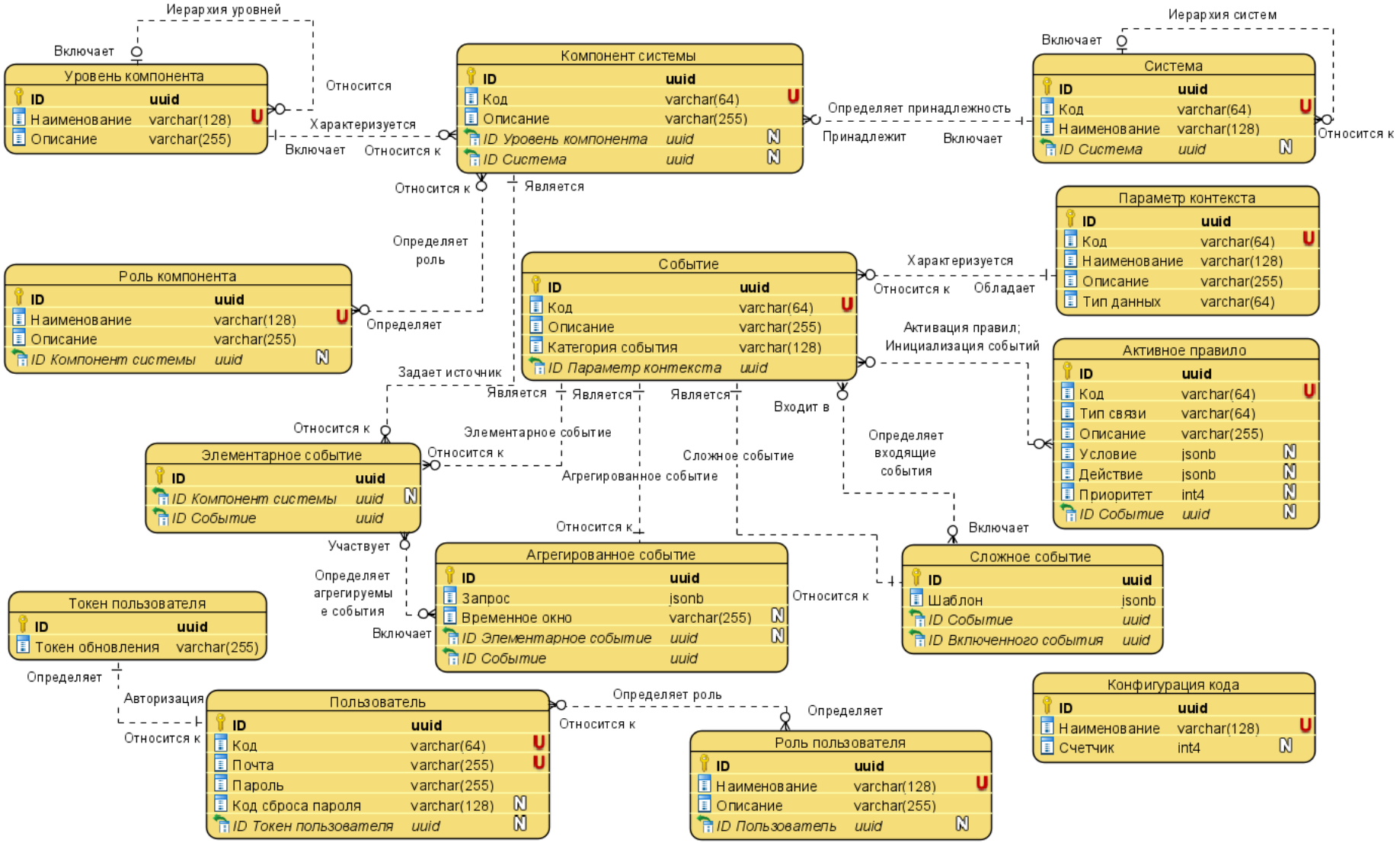

Рисунок 16 – Логическая модель репозитория метаданных

В данной версии UUID существует вероятность того, что идентификатор может быть продублирован, но он крошечный из-за большого количества возможных комбинаций. Например, на 10 триллионов сгенерированных UUID вероятность появления двух одинаковых значений равна 0,00000006 %.

Связи между отношениями реляционной базы данных репозитория метаданных реализуются с помощью внешних ключей, для которых заданы соответствующие ограничения ссылочной целостности. Отношения M:M в логической модели не декомпозированы с помощью добавления промежуточных таблиц.

2.2.3 Физическое моделирование

Физическое моделирование реляционной базы данных репозитория метаданных выполнялось на основе результатов логического моделирования и включало следующие действия:

1) отображение отношений логической схемы в таблицы физической схемы реляционной базы данных;

2) отображение бинарных связей между отношениями логической схемы в отношения между таблицами реляционной базы;

3) отображение атрибутов отношений логической схемы в столбцы таблиц физической схемы реляционной базы данных.

Связи M:M между отношениями логической модели реляционной базы данных декомпозированы с помощью добавления промежуточных таблиц. В качестве идентификатора той или иной сущности для сторонних приложения будет использоваться уникальный строковый код, вместо UUID четвертой версии. Код будет генерироваться для каждой сущности по собственному алгоритму.

Физическая модель репозитория метаданных представлена на рисунке 17.

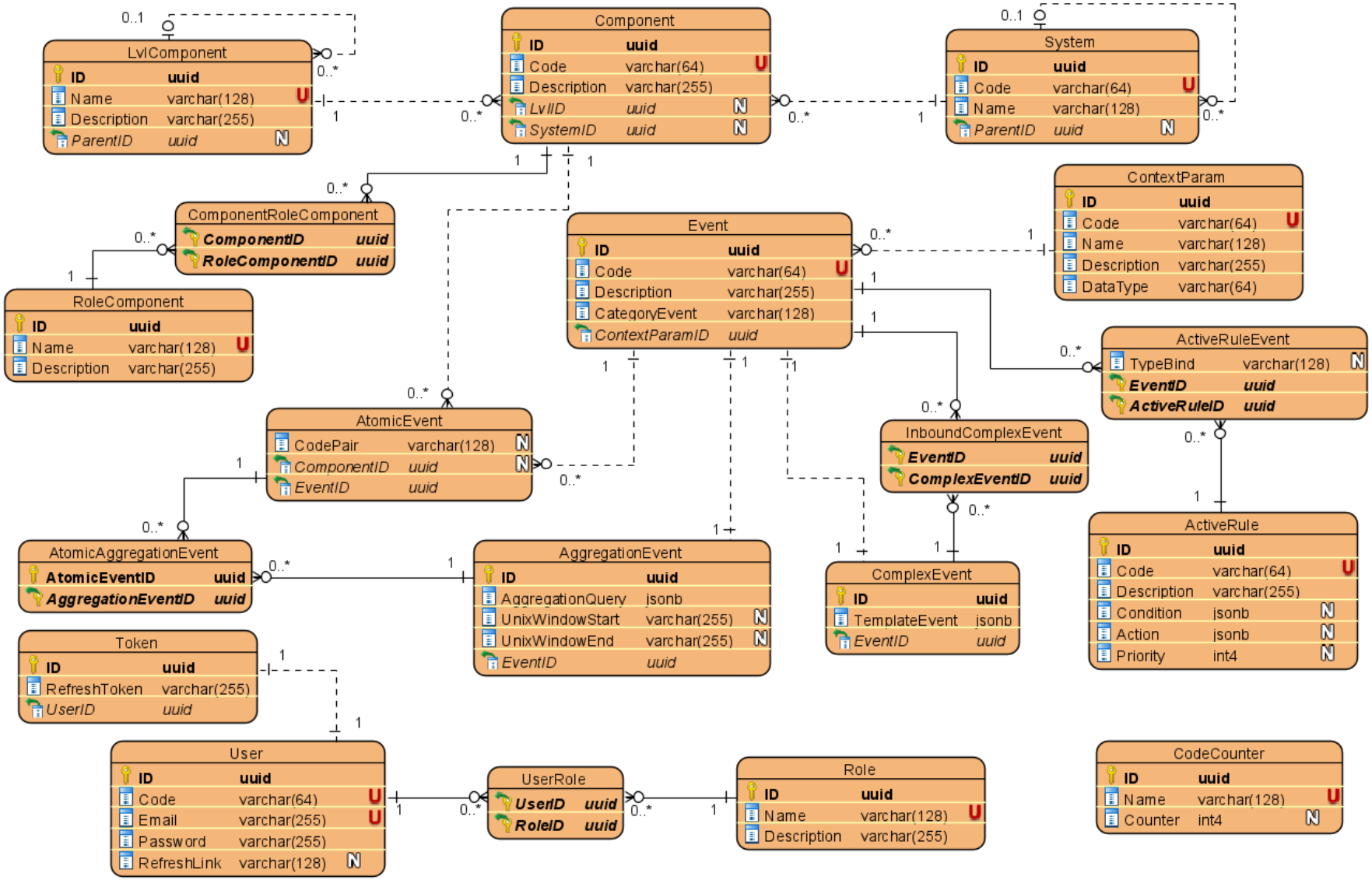

Рисунок 17 – Физическая модель репозитория метаданных

На основе физической модели средствами Visual Paradigm были сгенерированы SQL-скрипты для создания базы данных. Это позволит быстро развернуть и создать базу данных на стороне СУБД PostgreSQL.

2.3 Требования к архитектуре программных средств

Базовым элементом архитектуры является модуль. Модульная архитектура приложения – это подход к проектированию и разработке, при котором программное обеспечение разбивается на независимые элементы архитектуры, которые могут взаимодействовать между собой только через четко описанные интерфейсы. Модули содержат в себе логику, которая удовлетворяет принципу единой ответственности. В архитектуре приложения будут выделены следующие модули: модуль регистрации и авторизации пользователей, модуль работы с активными правилами, модуль работы с событиями, модуль работы с компонентами систем, модуль работы с ролями доступа пользователей и т.д.

Структура модулей сервера представляется в виде графа, в основе которого лежит «корневой» модуль. В нем происходит группирование остальных модулей. Отдельно взятый модуль организует работу контроллеров и провайдеров.

Контроллер отвечает за обработку входящих REST запросов и возврат ответа на клиент. Так как контроллер принадлежит определенному модулю, то он удовлетворяет принцип единой ответственности. То есть каждый контроллер отвечает исключительно на только запросы клиента. Например, контроллер модуля ролей обрабатывает только те запросы, которые касаются операций по работе с ролями. Nest изначально содержит функционал, облегчающий работу со всеми HTTP-методами.

Провайдеры (сервисы) служат для инкапсулирования логики приложения. Например, бизнес-логика или инфраструктурная логика. На уровне провайдеров используется один из популярнейших паттернов – внедрение зависимостей. Таким образом можно внедрять провайдеры в

контроллеры или другие высокоуровневые провайдеры. Также это позволяет держать код более поддерживаемым и читаемым.

В дополнение к основным элементам архитектуры REST-сервер должен обладать рядом других функциональных элементов. Для REST важно наличие промежуточного ПО (Middleware). Middleware – это метод, который вызывается перед обработчиком маршрута. Он имеет доступ к объектам запроса и ответа. В сервисе потоковой обработки событий и исполнения активных правил это хорошо подходит для журнализации данных.

В дополнении к промежуточному ПО для авторизации и аутентификации пользователей по ролям будет использоваться Guard («Охранник»). Он по принципу единственной ответственности разрешает или запрещает контроллеру обрабатывать запрос, в зависимости от уровня доступа пользователя.

В серверных средствах управления метаданными присутствует такой важный структурный элемент как Interceptor (Перехватчик). Перехватчики обладают набором полезных возможностей, которые основаны на технике аспектно-ориентированного программирования (AOP). Они позволяют:

– привязать дополнительную логику до / после выполнения метода;
– преобразовать результат, возвращаемый функцией;
– преобразовать исключение, создаваемое функцией;
– расширить базовое поведение функции;
– полностью переопределить функцию в зависимости от конкретных условий.

В сервисе потоковой обработки событий перехватчики выполняют множество задач. Например, кэширование метаданных или обработка ошибок запроса и возврат соответствующего ответа на клиент.

Архитектура была спроектирована таким образом, чтобы запрос клиента проходил все необходимые стадии жизненного цикла при максимальной эффективности. На основании описанных требований была составлена диаграмма жизненного цикла HTTP запроса (рисунок 18).

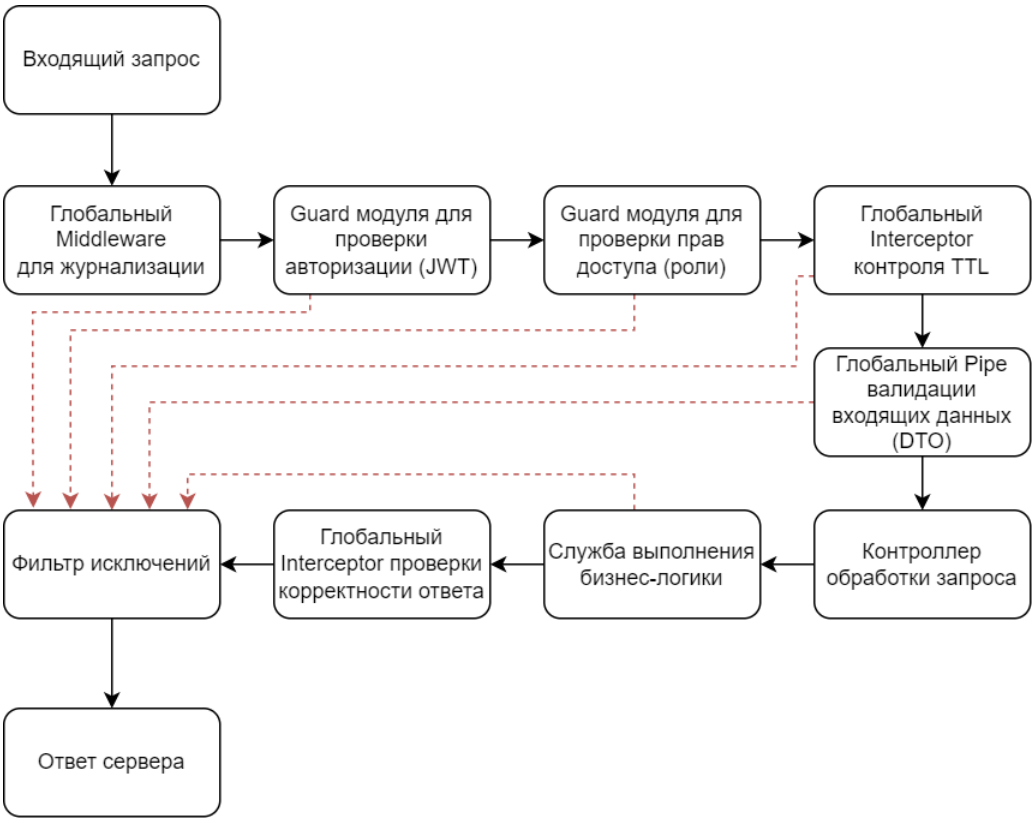

Рисунок 18 – Жизненный цикл запроса от клиентского приложения

В случае возникновения исключений на каком-либо этапе процесс обработки запроса обрывается. Сервис обладает глобальным программным средством обработки исключений, который обрабатывает исключения незамедлительно и возвращает соответствующий ответ клиенту.

2.4 Проектирование модулей REST-сервера

REST-сервер для управления метаданными в сервисе потоковой обработки событий имеет модульную архитектуру. Спроектированные модули должны инкапсулировать в себе логику по управлению собственной категорией метаданных. Также основные методы взаимодействия с репозиторием метаданных и организация моделей посредством ORM должны быть вынесены в отдельный модуль, чтобы его можно было использовать многократно без дублирования бизнес-логики. Данный подход упрощает масштабирование и поддержку приложения в долгосрочной перспективе. Диаграмма зависимостей модулей представлена на рисунке 19.

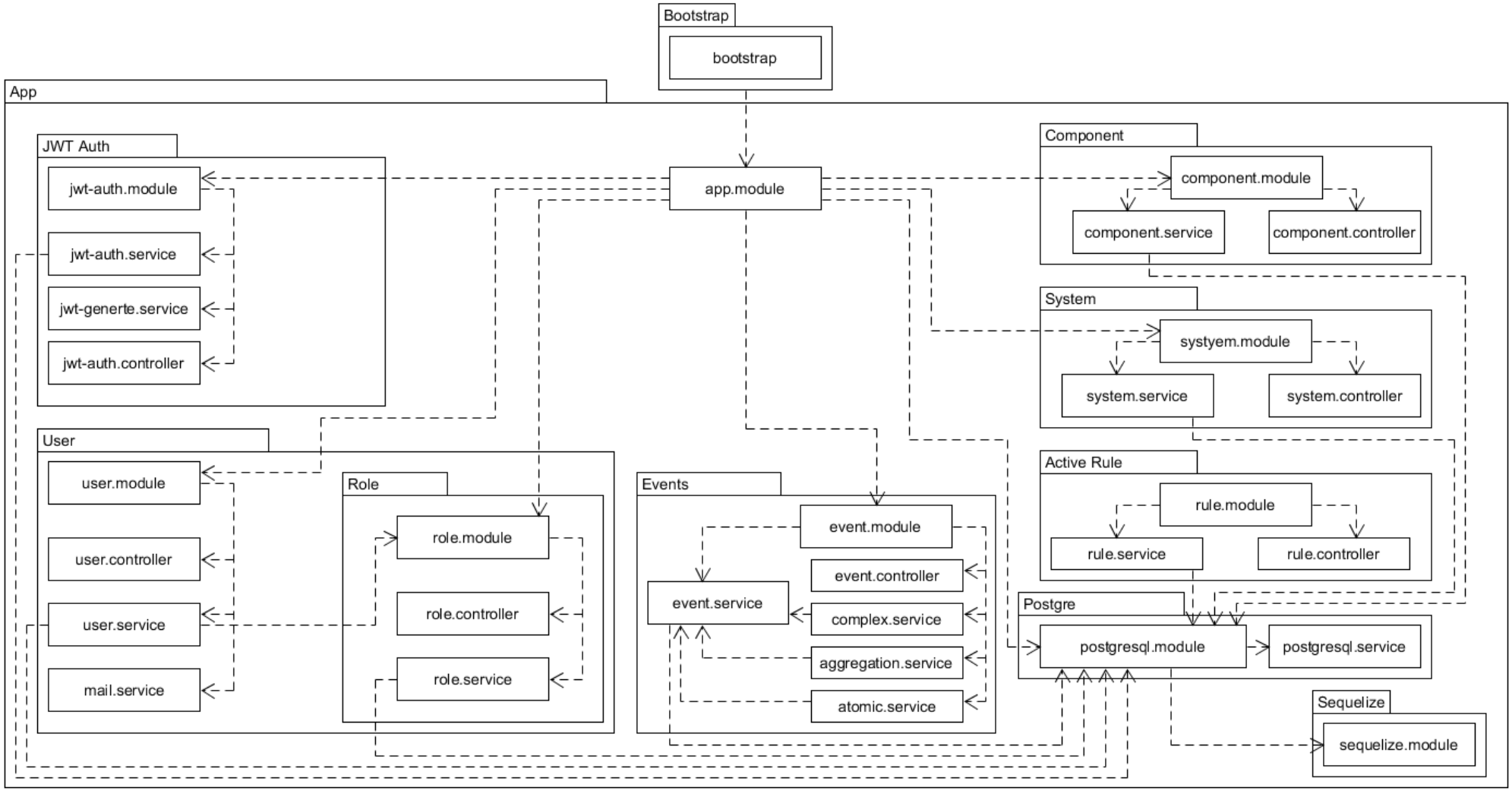

Рисунок 19 – Диаграмма пакетов модулей приложения

Более детальное описание каждого модуля представлено в таблице 4.

Таблица 4 – описание компонентов REST-сервера.

Модуль	Назначение
AppModule	Корневой модуль проекта. Отвечает за сборку всего проекта и запуск REST-сервера.
JwtAuthModule	Представляет метаданные для организации структуры приложения. Реализует зависимости компонентов группы JWT
JwtGenerateSevice	Отвечает за генерацию, обновление, удаление JWT токенов.
JwtAuthService	Отвечает за верификацию JWT токенов, авторизацию и аутентификацию пользователей.
UserModule	Представляет метаданные для организации структуры приложения. Реализует зависимости компонентов группы User
UserService	Реализует CRUD операции для аккаунтов пользователей, операции по активации аккаунта.
MailService	Отвечает за верификацию email-адреса пользователя, подтверждение регистрации.
RoleModule	Представляет метаданные для организации структуры приложения. Реализует зависимости компонентов группы Role
RoleService	Отвечает за роли, доступные пользователям. Их присвоение конкретного аккаунту. Реализует CRUD операции.
EventsModule	Представляет метаданные для организации структуры приложения. Реализует зависимости компонентов группы Events
EventsService	Отвечает за метаданные событий. Предоставляет CRUD операции.
PostgresqlModule	Отвечает за объектно-ориентированное представление сущностей, данных, связей в БД. Отвечает за работу с ними.
PostgresqlService	Отвечает за начальную инициализацию данных.
ComponentModule	Представляет метаданные для организации структуры приложения. Реализует зависимости компонентов группы Component
ComponentService	Реализует CRUD операции для компонентов системы.
SystemModule	Представляет метаданные для организации структуры приложения. Реализует зависимости компонентов группы System
SystemService	Реализует CRUD операции для систем сервиса
RuleModule	Отвечает за метаданные активных правил. Организует зависимости с сервисами по конфигурации активных правил.
RuleService	Реализует CRUD операции. Производит привязку событий к правилам с указанием типа связи.

Также необходимо будет разработать дополнительный набор модулей для организации связи с микросервисами. Его основными задачами будут являться: оповещение об обновлении метаданных и их транспортировка.

2.5 Обоснование выбора технологий и средств разработки

Процесс разработки серверных программных средств управления метаданными будет основан на методологии Rational Unified Process (RUP). Данная методология получила свою широкую известность благодаря своей эффективности. Она основывается на интерактивной модели разработки, которая подразумевает разбиение проекта на несколько подпроектов, которые разрабатываются по порядку [12]. Каждому подпроекту назначается определенный перечень задач, бюджет и сроки выполнения. Особое внимание акцентируется на начальных этапах разработки, что позволяет значительно снизить коммерческие и технические риски. Одной из особенностей данной методологии является то, что в результате работы над проектом создается множество моделей, которые позволяют заранее рассмотреть будущее программное обеспечение.

REST-сервер будет разработан с помощью фреймворка Nest [13]. Фреймворк предлагает заранее предопределенную модульную архитектуру, которая заточена под максимально удобную поддержку и простое масштабирование.

По результатам разработки будет проведено тестирование программного кода. Автоматизированное тестирование будет проведено при помощи фреймворка Jest [14]. Фреймворк обладает рядом преимуществ, например, во избежание вмешательство в работу сервера и вставки тестовых данных в репозиторий он позволяет работать с Stub и Mock-объектами. Stub-объекты подменяют реальные дынные из других модулей системы, необходимые для тестов. Mock-объекты обладают таким же интерфейсом, что и реальные типы данных, но они заданы сценарием для возврата заранее запрограммированных ответов. Вдобавок данный фреймворк позволяет просматривать покрытие кода тестами в процентном и количественном вариантах. Он тесно связан с фреймворком Nest и предоставляется в расширенном пакете инструментальных средств.

Функциональное тестирование HTTP запросов будет производиться при помощи ПО Postman. Он не требует какой-либо сложной предварительной настройки и является абсолютно бесплатным. Postman поддерживает все ключевые элементы HTTP запросов: от выбора метода запроса до настройки cookies файлов и токенов авторизации.

REST-сервер для доступа к единому репозиторию метаданных сервиса потоковой обработки событий будет разворачиваться внутри Docker-контейнеров [15]. Контейнер – это способ упаковать приложение и все его зависимости в единый образ. Образ запускается в изолированной среде (Linux), не влияющей на основную операционную систему. Тем самым Docker-контейнер не зависит от окружения, присутствия там необходимых зависимостей и настроек. Параллельно с REST-сервером разворачивается PostgreSQL при помощи Docker-compose механизма, который позволяет работать с несколькими контейнерами одновременно.

Одним из основных преимуществ контейнеров является эфемерность. Docker позволяет разрабатывать приложения по принципу Stateless, когда приложение не полагается на хранилище данных в контейнере. Все данные хранятся в специальных томах (Docker volumes). Поэтому контейнер, без риска потери данных, в любой момент времени может быть остановлен, перезапущен или уничтожен.

Наборов инструментов Swagger [16] предоставляет широкую возможность документирования API сервера. Swagger позволяет быстро создать документацию без углубления в тонкости разметки HTML страниц. Данный инструмент имеет широкий спектр аннотаций. Например, аннотации для описания операций и методов HTTP, описания параметров и тела запроса, описания выходных данных и так далее. Интерфейс документации интуитивно понятен и предоставляет необходимые данные для настройки запросов на клиентском приложении.

В качестве web-сервера был выбран Nginx [17]. Он демонстрирует высокую скорость обработки сторонних подключений при низких

аппаратных требованиях, в отличие от web-сервера Apache. Также Nginx обладает высокой масштабируемостью.

2.6 Требования к аппаратно-программному обеспечению

2.6.1 Требования к аппаратно-техническому обеспечению

Серверные программные средства управления метаданными обладают минимальными и рекомендуемыми требованиями к аппаратно-техническому обеспечению. Требования представлены в таблице 5.

Таблица 5 – Требования к аппаратно-техническому обеспечению

Требование	Минимальные	Рекомендуемые
Процессор	Тактовая частота 1,6 ГГц	2,8 ГГЦ
Количество вычислительных потоков процессора	4	8
Оперативная память	1 ГБ	4 ГБ
Свободное дисковое пространство	15 ГБ	15 ГБ
Тип дисковых накопителей	HDD	SSD /NVMe

Оборудование должно быть обеспечено бесперебойным энергоснабжением. Емкость ИБП должна обеспечивать непрерывную работу оборудования при отсутствии энергоснабжения в течение минимум 30 минут. Необходимо обеспечить возможность корректного завершения работы сервиса в автономном режиме.

2.6.2 Требования к прикладному и системному программному обеспечению

Основными требованиями к прикладному программному обеспечению является наличие Docker ПО и web-сервера Nginx. Docker – это платформа для запуска приложений в виртуальных окружениях, которые называются контейнерами. Виртуализация позволяет быстро разворачивать программное обеспечение вне зависимости от архитектуры и без каких-либо требований

для самих приложений к операционной системе. Docker обладает требованиями, которые описаны далее.

Для развертывания на Windows:

– Windows 10 64-бит: Pro, Enterprise или Education (сборка 15063 или более поздняя);

– функции Hyper-V и Containers Windows должны быть активны.

Минимальные требования для запуска клиента Hyper-V на Windows 10:

– 64 битный процессор с функцией Second Level Address Translation (SLAT);

– 4 GB оперативной памяти;

– на уровне BIOS должна быть включена поддержка виртуализации.

Для развертывания на системах на базе ОС Linux:

– разрядность 64 бит;

– ядро версии 3.10 или новее.

Для обеспечения производительности и стабильности работы не рекомендуется устанавливать на сервер стороннее программное обеспечение.

3 Разработка серверных программных средств управления метаданными для сервиса потоковой обработки событий

3.1 Разработка архитектуры развертывания

На основании требований к программным средствам была разработана архитектура развертывания (рисунок 20).

В качестве хранилища данных использовалась СУБД PostgreSQL. Для связи REST-сервера с микросервисами была настроена резидентная система управления базами данных Redis. Она при помощи шаблона публикации/подписки выполняет функции брокера сообщений.

Основное приложение работает на платформе NodeJS. Был написан скрипт, который позволяет данному приложению при компиляции минимизировать исходные файлы, чтобы те были готовы к производству и быстрее читались системой. Важно отметить, что REST-сервер не имеет тесную связь с Redis или PostgreSQL. Все три решения были настроены и помещены в отдельные Docker-контейнеры.

При помощи средств Docker были настроены механизмы перезапуска контейнеров. Контейнеры будут автоматически запускаться в аварийных случаях, например, незапланированная перезагрузка сервера или кратковременный выход из строя сетевого оборудования. Политика перезапуска гарантирует, что связанные контейнеры будут запущены в правильном порядке.

В качестве web-сервера был использован Nginx, который можно гибко конфигурировать под нужны инфраструктуры. Также данный web-сервер имеет малое потребление оперативной памяти и ресурсов процессора. Он отвечает за принятие HTTP-запросов от клиентский приложений. Для повышения отказоустойчивости сервиса он не был помещен в Docker-контейнер, конфигурация производилась напрямую под ОС Ubuntu.

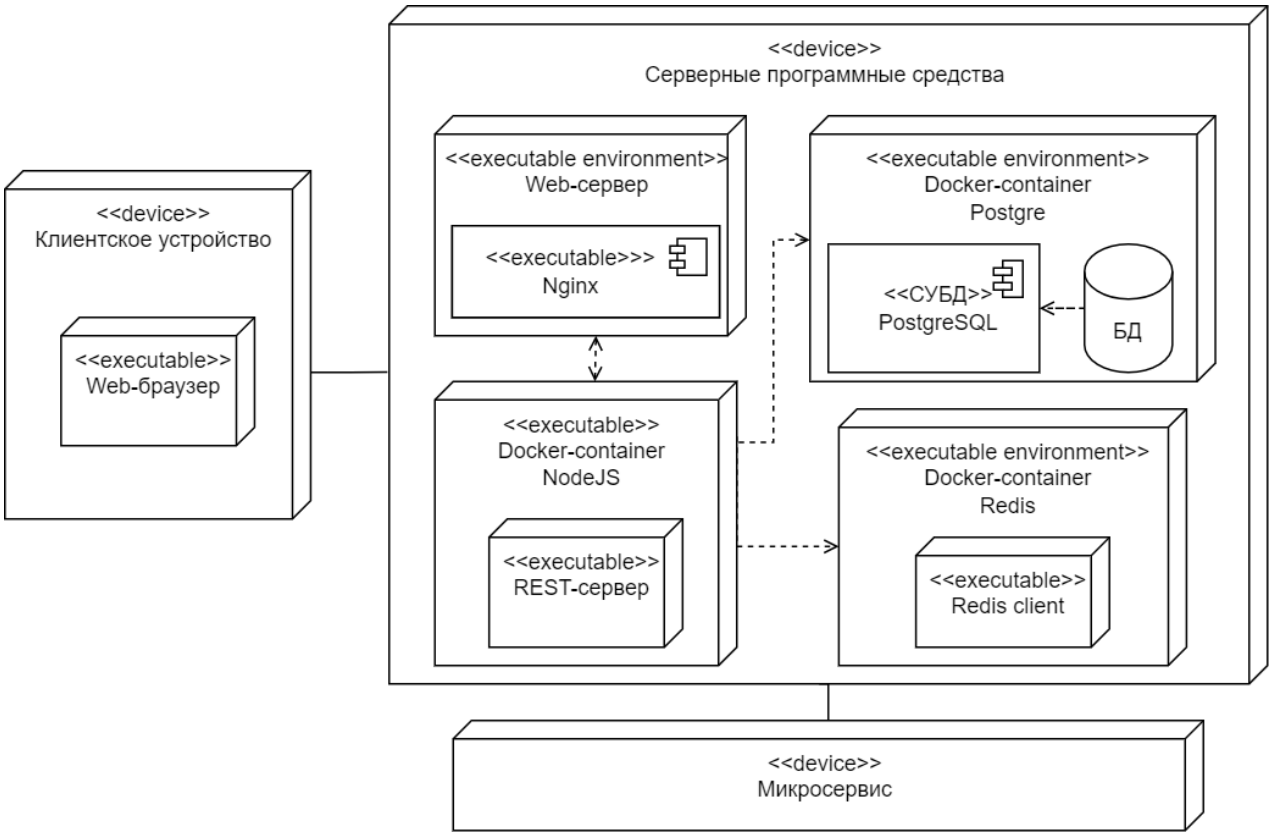

Рисунок 20 – Диаграмма развертывания

Между сервисными программными средствами была организована внутренняя сеть для коммуникации между собой. Вдобавок для PostgreSQL настраивалась отдельная сеть для доступа извне.

3.2 Разработка программных средств взаимодействия с базой метаданных

3.2.1 Разработка физической структуры базы метаданных

На основании смоделированной модели репозитория метаданных была разработана база данных при помощи средств PostgreSQL. Структура разработанной базы данных представлена на рисунке 21. Она удовлетворяет всем требованиям, предъявленным на этапе проектирования.

Репозиторий содержит 19 сущностей, 2 из которых реализуют иерархическую связь. Связи M:M были реализованы при помощи промежуточных таблиц.

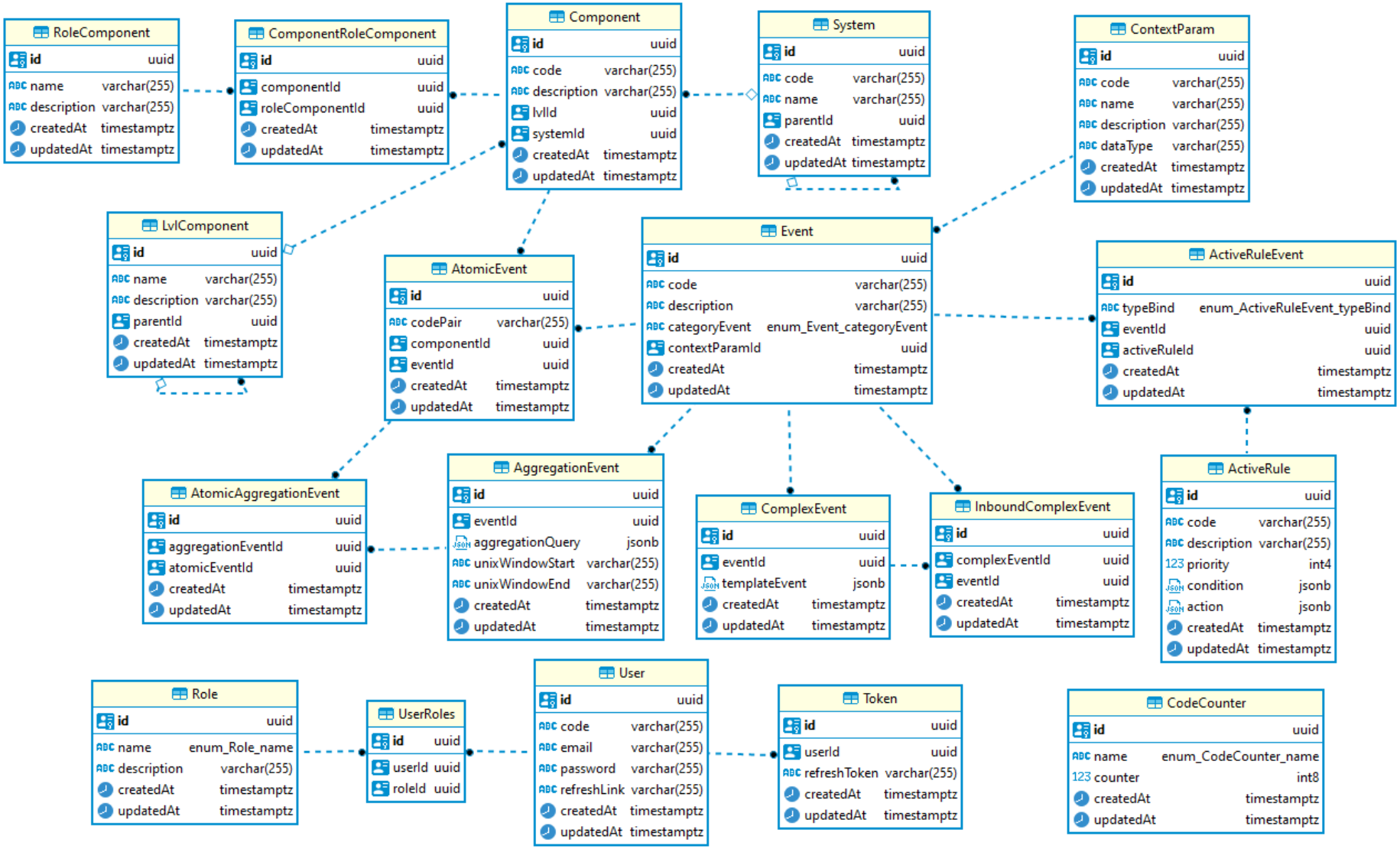

Рисунок 21 – Структура репозитория метаданных

Дальнейшее администрирование репозитория метаданных производилось при помощи средств DBeaver.

3.2.2 Разработка программных средств доступа к базе метаданных

Nest фреймворк имеет полноценную интеграцию с SQL и NoSQL базами данных. Из коробки имеются сразу несколько ORM. Работа с PostgreSQL производится посредством использования SequelizeORM. На листинге 1 представлено подключение к базе данных.

Листинг 1 – Подключение сервиса потоковой обработки событий к PostgreSQL

```
@Module({
    imports: [
        SequelizeModule.forFeature([
            User, Token, Role, UserRole,
            Component, RoleComponent,
  ComponentRoleComponent, LvlComponent,
            System, Event, ParamEvent
        ]),
        SequelizeModule.forRoot({
            dialect: 'postgres',
            port: +process.env.POSTGRES_PORT,
            host: process.env.POSTGRES_HOST,
            username: process.env.POSTGRES_USER,
            password: process.env.POSTGRES_PASSWORD,
            database: process.env.POSTGRES_DB,
            autoLoadModels: true,
            synchronize: true,
        }),
    ],
    controllers: [],
    providers: [PostgresqlService],
    exports: [SequelizeModule]
})
export class PostgresqlModule {}
```

SequelizeORM позволяет взаимодействовать с базами данных на языке приложения (TypeScript), автоматически генерируя SQL запросы. Представления таблиц БД сделано при помощи некой абстракции – модели. Модель Sequelize предоставляет информацию о таблице: название, атрибуты,

типы данных, PK, FK и т.д. Ниже приведен код одной из моделей репозитория (листинг 2).

Листинг 2 – Модель таблицы Event

```
@Table({ tableName: 'Event' })
export class Event extends Model<Event, IEventCreate> {
      @Column({
            type: DataType.UUID,
            defaultValue: DataType.UUIDV4,
            primaryKey: true,
      })
      id: string;

      @Column({
            type: DataType.STRING,
            unique: true,
            allowNull: false,
      })
      code: string;

      @Column({
            type: DataType.STRING,
            allowNull: false,
      })
      description: string;

      @Column({
            type: DataType.ENUM(...Object.values(CategoryEvent)),
            allowNull: false, })
      categoryEvent: CategoryEvent;

      @HasMany(() => AtomicEvent)
      atomicEvent: AtomicEvent[];

      @HasOne(() => ComplexEvent)
      complexEvent: ComplexEvent;

      @HasOne(() => AggregationEvent)
      aggregationEvent: AggregationEvent;

      @ForeignKey(() => ContextParam)
      @Column({ type: DataType.UUID, allowNull: false })
      contextParamId: string;

      @BelongsTo(() => ContextParam)
      contextParam: ContextParam;

      @BelongsToMany(() => Component, () => AtomicEvent)
      component: Component[];

      @BelongsToMany(() => ComplexEvent, () => InboundComplexEvent)
      inboundComplexEvent: ComplexEvent[];

      @BelongsToMany(() => ActiveRule, () => ActiveRuleEvent)
      activeRule: ActiveRule[];
}
```

Данные модели внедряются в соответствующий модуль на основании паттерна программирования «Внедрение зависимостей» (Dependency Injection). Перечень модулей соответствует тем, которые были описаны на этапе проектирования.

3.3 Разработка базовых программных средств

3.3.1 Разработка программных средств управления метаданными

Основными объектами управления являются события и активные правила, а также связанное с ними окружение. Бизнес-логику управления метаданными инкапсулируют сервисы REST приложения. Для создания SQL запросов посредством ORM заранее определены представления (View). Это способствует оптимизации запросов, так как метод запрашивает только те данные, которые необходимы ему для работы. Разработано 18 сервисов, основные сервисы управления метаданными были подробно изображены на диаграмме компонентов и расписаны на этапе проектирования.

Для каждого набора метаданных обязательно существуют четыре базовых сценария: создание, чтение, удаление, изменение. Некоторый сервисы обладают дополнительной логикой по управлению метаданными. Дополнительные сценарии описаны в таблице 6. Код сервисов модулей событий и активных правил приведен в приложении Б.

Таблица 6 – Дополнительные сценарии управления метаданными активных правил, событий и их окружения.

Сервис	Сценарий	Пояснение
1	2	3
Actve Rule	Bind Event	Назначение события активному правило по одному из двух видов связи: «Событие порождает активное правило», «Активное правило порождает событие»
Actve Rule	Unbind Event	Отсоединение события от активного правила
Actve Rule	Find Events	Поиск присоединенных событий по всем типам связи.
Component	Add Role	Добавление роли компоненту
Component	Remove Role	Удаление роли у компонента
Component	Add Lvl	Добавление уровня компоненту
Component	Remove Lvl	Удаление уровня у компонента
Component	Add System	Присоединение компонента к системе

Продолжение таблицы 6

1	2	3
Component	Remove System	Отсоединение компонента от системы
Event	Add Atomic To Aggregation	Добавление элеметарного события в аггрегированное
Event	Remove Atomic To Aggregation	Удаление элеметарного события из аггрегированного
Event	Add Inbound	Добавление элементарного события в состав сложного
Event	Remove Inbound	Удаление элементарного события из сложного
Counter	Create Code	Создание уникального кода для объекта метаданных
System	Attach	Управление иерархией. Назначение дочерней системы родительской.
System	Detach	Управление иерархией. Удаление дочерней системы из родительской.

Преимущество сервисов в данной архитуктуре – удобное внедрение зависимостей (Dependency injection). Сервисы могут взаимодействовать между собой без какой либо жесткой привязки друг к другу. Взаимодействие сервисов для осуществления сценария привязки активных правил к событию изображено на диаграмме последовательности (рисунок 22).

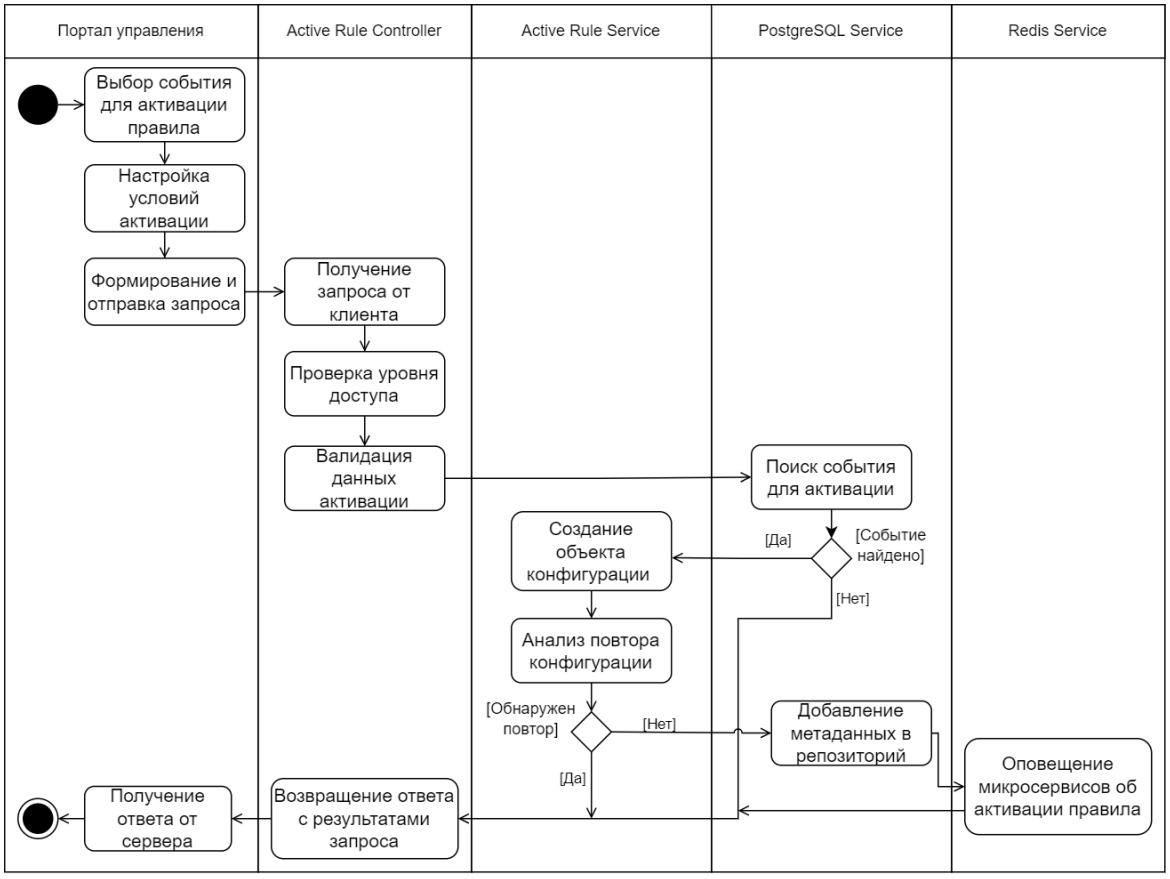

Рисунок 22 – Диаграмма последовательности для привязки активного правила к событию

От портала управления поступает запрос на привязку активного правила. Запрос обрабатывается соответствующим контроллером и делегирует выполнение бизнес-логики следующим компонентам: сервису активных правил, сервису взаимодействия с репозиторием и сервису взаимодействия с микросервисами.

Микросервисы получают оповещение об изменении или создании новых метаданных. Это необходимо для корректного отслеживания наблюдаемых систем и оперативного взаимодействия на их компоненты.

Если пользователь не имеет требуемый уровень доступа или тело запроса имеет неверную структуру, то запрос будет сразу отклонен с соответствующей ошибкой. Более детально алгоритм аутентификации пользователей и валидации данных будет рассмотрен на последующих этапах.

3.3.2 Разработка программных управления пользовательскими аккаунтами

Регистрация пользователя производится через панель администрирования. Для регистрации необходимо передать на сервер email пользователя и его пароль.

REST-сервер обладает Mail сервисом, который работает с транспортером электронных сообщений Google. Код сервиса приведен в приложении Б. Данный сервис отвечает за рассылку email-сообщений пользователям при активации аккаунта, смены пароля и его сброса.

Сброс пароля может осуществляться как администратором, так и пользователем напрямую. Для смены пароля сервер запрашивает следующие данные: email пользователя, его старый и новый пароль. Для сброса пароля требуется лишь email пользователя. В данном случае сервис отправляет два сообщения. Первое из которых является подтверждением того, что действие сделано намерено и старый пароль необходимо сбросить. Данное сообщение изображено на рисунке 23.

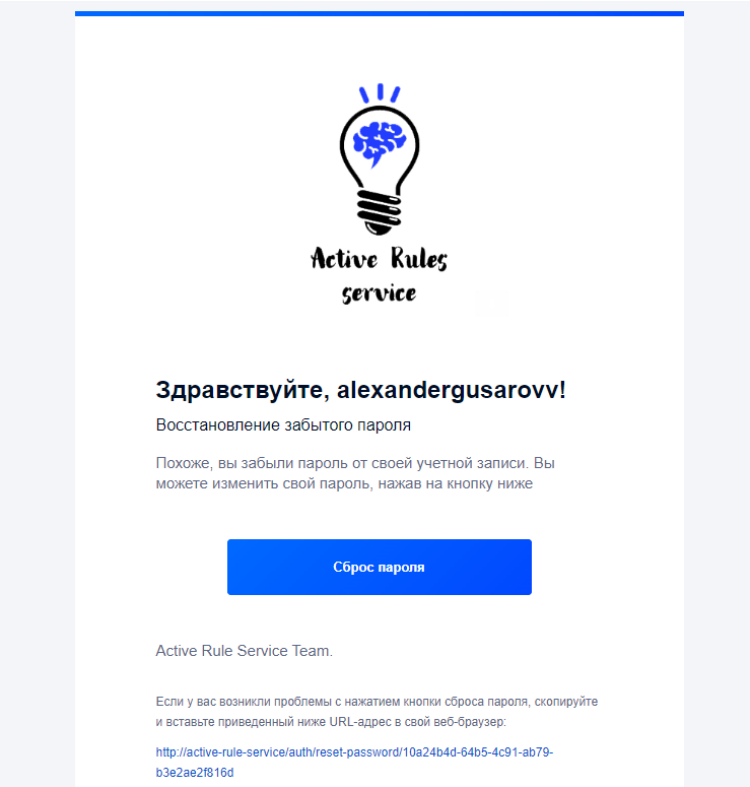

Рисунок 23 – Сообщение с подтверждением сброса пароля пользователя

Второе письмо (рисунок 24), в случае подтверждения сброса, отправляет новый пароль. Пароль автоматически генерируется серверными программными средствами.

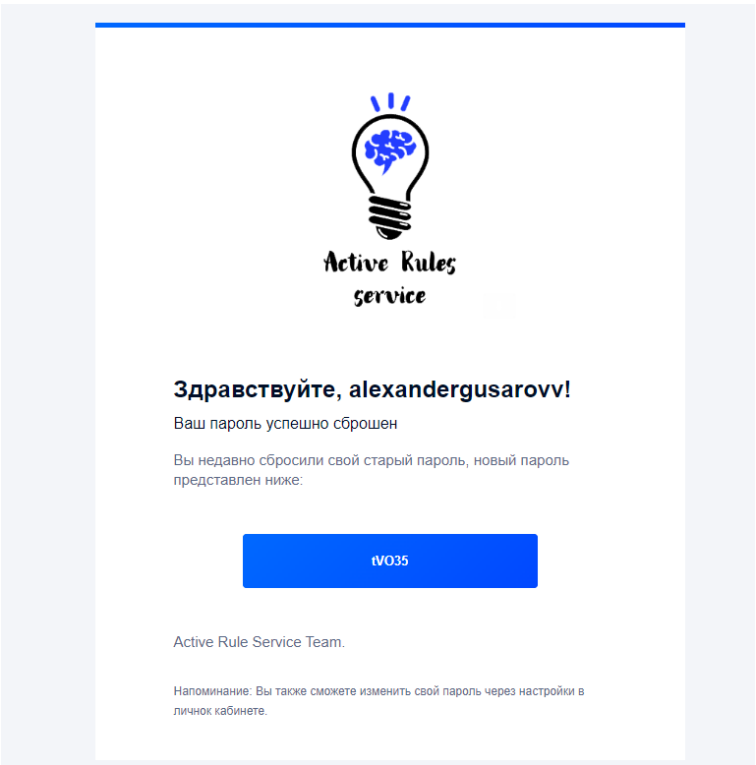

Рисунок 24 – Сообщение с новым паролем пользователя.

Если автоматически сгенерированный пароль не устроит пользователя, то он всегда сможет сменить его в личном кабинете на портале управления.

3.3.3 Разработка программных средств аутентификации пользователей

Доступ пользователей к определенному функционалу сервиса осуществляется при помощи токенов доступа – JSON Web Token (JWT). Работа токенов была подробна рассмотрена на этапе проектирования.

Хранение Access токена реализовано в локальном хранилище портала (LocalStorage). Refresh токен содержится в httpOnly cookie. Важно, чтобы присутствовало наличие флага httpOnly, чтобы через стороннее клиентское ПО было невозможно изменить cookie браузера. Их установка должна производиться исключительно сервером. Refresh токен, в свою очередь, также хранится на сервере со вспомогательными метаданными. Например, запись токена в паре с IP адресом клиента. Если при обновлении токена доступа обнаружится новый IP адрес, то можно высылать email-уведомление пользователю о том, что был произведен вход с другого устройства.

За основную работу с JWT токенами отвечают компоненты: JwtAuthController, JwtAuthService и JwtGenerateService. Первых отвечает за принятие HTTP-запросов, два других – за выполнение бизнес-логики. Диаграммы первых двух классов представлены на рисунках 25 и 26. Код сервисов представлен в приложении Б. Рассмотрим их спецификацию более подробно.

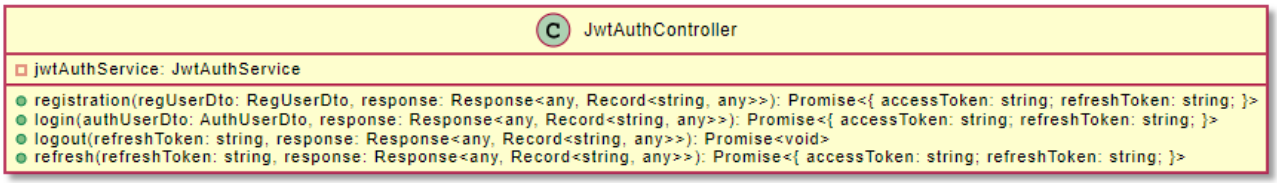

Рисунок 25 – Диаграмма класса JwtAuthController

Внедрение зависимости: jwtAuthService – сервис, инкапсулирующий бизнес-логику по работе с авторизацией. Контроллеры, требующие

авторизованных пользователей, после обработки HTTP-запроса передают выполнение бизнес-логики этому сервису.

Методы класса:

– registration – обработка HTTP-запроса, валидация формата данных для регистрации пользователя;

– login – обработка HTTP-запроса, валидация формата данных для авторизации пользователя;

– logout – обработка HTTP-запроса для выхода из системы пользователя;

– refresh – обработка HTTP-запроса для обновления Access токена, работа с cookie.

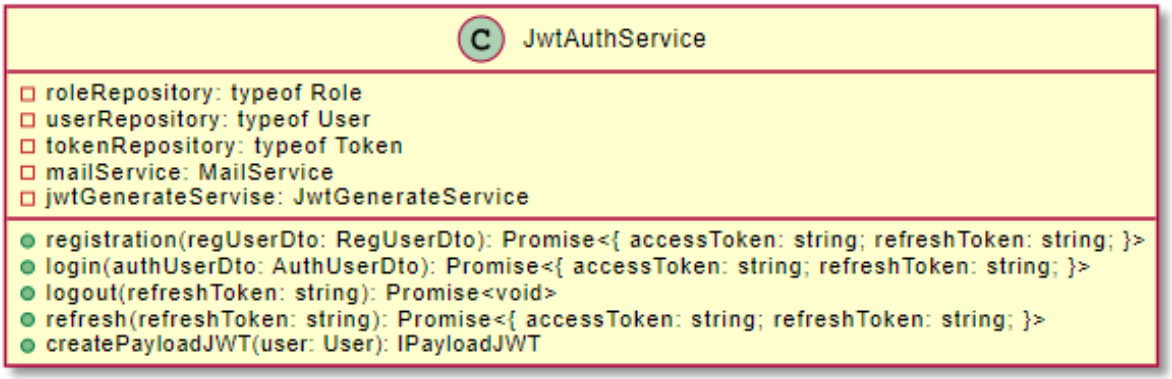

Рисунок 26 – Диаграмма класса JwtAuthService

Внедрения зависимостей:

– roleRepository – модель репозитория для работы с ролями пользователей;

– userRepository – модель репозитория для работы с данными пользователей;

– tokenRepository – модель репозитория для работы с Refresh токенами пользователей;

– mailService – сервис, отвечающий за верификацию почты путем отправки сообщения пользователю;

– jwtGenerateService – сервис, инкапсулирующий логику создания JWT токенов, шифрование, а также их валидацию.

Методы класса:

– registration – регистрация пользователя в системе;

– login – авторизация пользователя в системе, предоставление токенов доступа;

– logout – выход пользователя из системы, уничтожение токенов доступа;

– refresh – обновление токена доступа пользователя;

– createPayloadJWT – создание полезной нагрузки для токена доступа.

Access Token содержит «полезную нагрузку» (Payload). Payload – это «полезные» данные, которые хранятся внутри JWT и необходимы клиентским приложениям для осуществления внутренней бизнес-логике. Среди данных присутствует информация о владельце токена: уникальный идентификатор пользователя, email, последняя дата авторизации, список ролей доступа.

3.3.4 Разработка программных средств анализа корректности метаданных

Сервис обладает двумя типами анализа корректности метаданных: статический и динамический. Статический анализ выполняется на этапе управления метаданными, а динамический на этапах процессов обработки событий и исполнения активных правил. Ответственность за статический анализ берет на себя REST-сервер.

Существуют следующие категории проблемных ситуации: избыточность, коллизии активных правил, цикличность, смысловая несогласованность [18].

Избыточность означает, что могут быть события или правила, которые идентичны друг другу по определенным признакам. Это может произойти в больших системах, где события или правила создаются разными людьми. К признакам дублирования событий можно отнести: идентичная категория, одинаковый контекст исполнения, одинаковый источник (в случае

элементарных событий). Присутствуют дополнительные критерии, связанные с конфигурацией сложных и агрегированных, например, шаблон исполнения сложного события или выборка элементарных событий для агрегирования. Поиск избыточных событий осуществляется при помощи алгоритмов на этапах создания и редактирования событий, а также на этапе проектирования системы.

Сценарий создания сложного события при получении запроса с клиентского приложения рассмотрен на рисунке 27. В дополнение к алгоритмам проверки при создании активного правила на диаграмме отображено состояние первичной проверки корректности запроса. Оно будет подробнее описано на следующем этапе разработки.

К признакам избыточности активных правил можно отнести: одинаковое условие активации и схожие исполняемые действия, например, порождение одних и тех же событий. Избыточные правила перегружают систему, так как одно и то же действие будет выполнено минимум два раза. Поиск избыточных правил осуществляется на схожих с событиями этапах.

У каждого правила имеется приоритет исполнения. Приоритет – числовое значение в диапазоне от 1 до 10. Система автоматически выставляет приоритет равным 5, если пользователь не указал иного. Изменить приоритет можно при редактировании активного правила. Коллизии исполнения активных правил возникают тогда, когда одним событием порождается несколько противоречивых друг другу правил с одинаковым приоритетом. В таком случае возникает ситуация «гонки», когда любое из событий может активироваться раньше другого. В зависимости от порядка выполнения правил, конечное состояние системы может быть разным. Разработан алгоритм, который может обнаружить коллизии на основании параметров исполняемых действий. Например, если в одном активном правиле действие «alarm» имеет значение «true» (включение сигнализации), а в другом правиле данное действие имеет значение «false», то алгоритм посчитает эти правила как противоречивые.

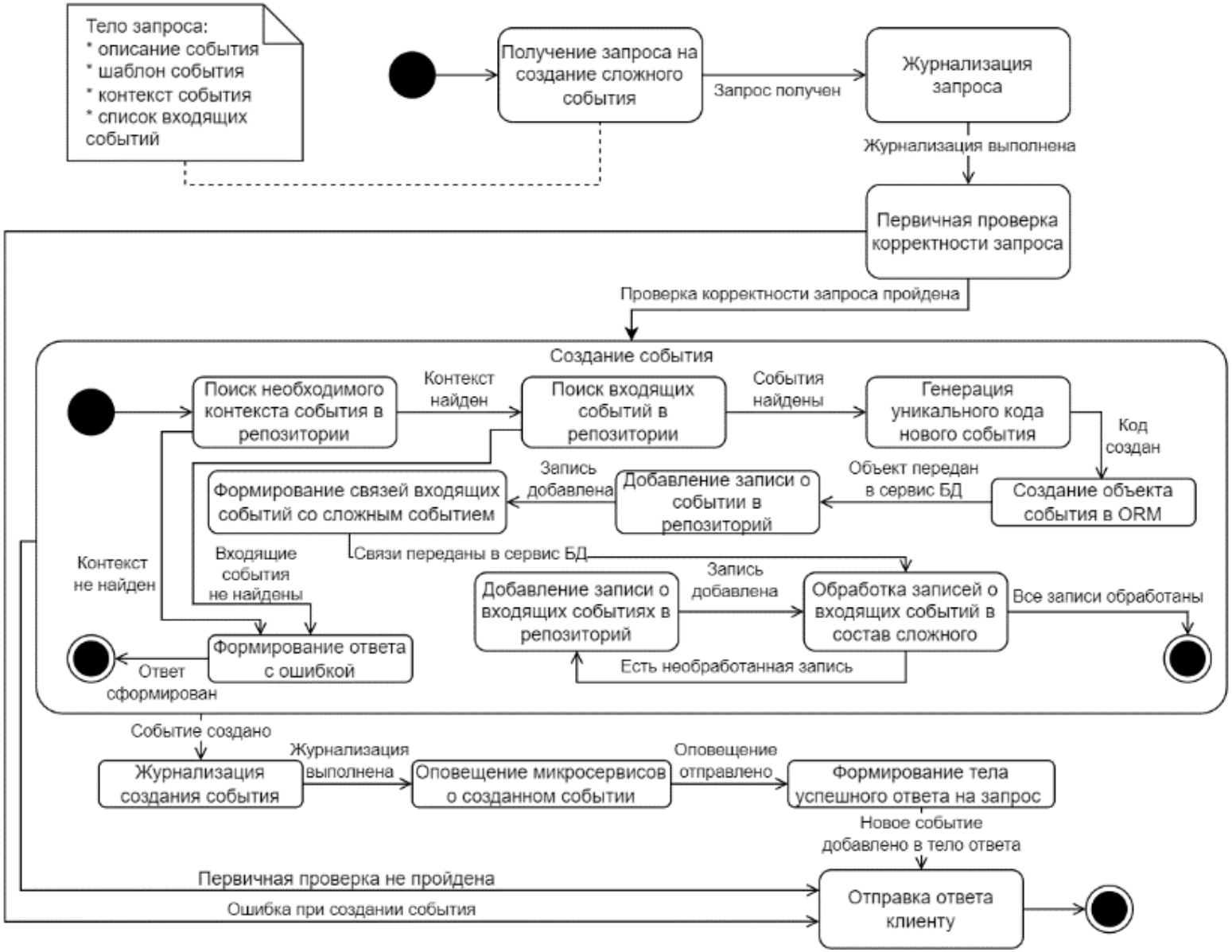

Рисунок 27 – Диаграмма состояний создания сложного события

Цикличность означает бесконечный повтор исполнения одних и тех же активных правил. Сущетсвует два вида циклов: гарантированный и негарантированный. В гарантированном цикле всегда порождается действие, которое приводит к повтору состояния. Например, последовательное исполнение двух правил с условиями «станок выключен»/«станок включен» и с действиями «включить станок»/«выключить станок» приведет к повтору первоначального состояния. В негарантированном цикле присутствуют активные правила, которые только при выполнении определенного условия могут приводить к повтору состояния. Реализованы алгоритмы, которые методом обхода конфигураций могут обнаружить данную конфликтную ситуацию.

Смысловая несогласованность правил подразумевает под собой цепочку противоречащих друг другу действий. Например, действия открыть окна и включить обогреватель. Чтобы узнать, какие действия противоречивы, необходимо иметь конкретную информацию о внешней среде. Смысловая несогласованность определяется на этапе динамического анализа при помощи микросервисов.

3.4 Разработка программных средств взаимодействия с внешними приложениями

3.4.1 Разработка программных средств взаимодействия с клиентским приложением

Было разработано 10 контроллеров, которые отвечают за принятие HTTP запроса и возврат ответа. Они делегируют всю бизнесс-логику соответсвующим сервисам. Каждый контроллер обрабатывает свои тематические точки доступа. Существуют следующие контроллеры:

- User;
- User Role;
- System;
- Component;

– Component Role;

– Component Lvl;

– JWT Auth and Registration;

– Event;

– Context Param Event;

– Active Rule.

Все точки доступа REST-сервера поведены в документации. В ней отражаются основные моменты, необходимые для запроса, а также возможные варианты ответа. Документация написана при помощи case-средств Swagger. Рассмотрим все точки доступа для взаимодействия с событиями сервиса (рисунок 28).

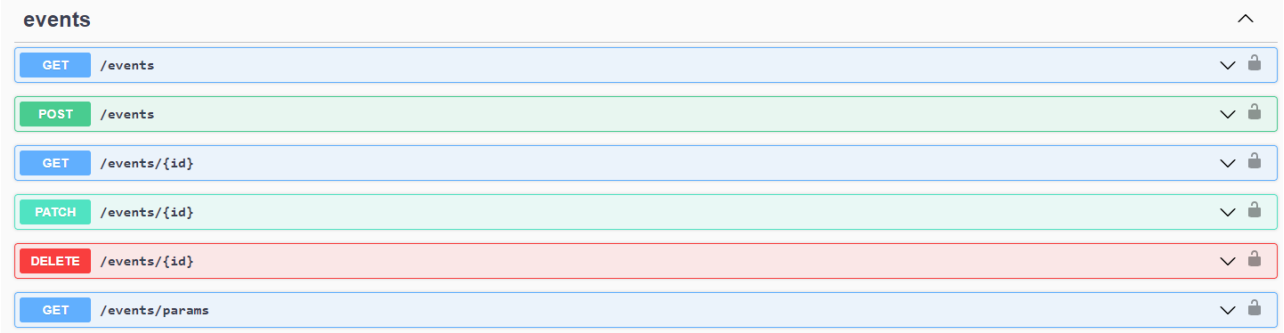

Рисунок 28 – Описание точек доступа для работы с событиями сервиса

Каждый элемент документации можно раскрыть для более детального представления. На примере (рисунок 29) работы с точкой доступа /events/{id} для обновления событий видно:

– запрос имеет метод PATCH, указывающий на возможное частичное обновления данных;

– идентификация объекта типа event определяется при помощи динамического параметра;

– схема возможных полей обновления объекта (DTO) с указанием типов данных;

– возможные ответы сервера (200 – объект успешно изменен, 403 – доступ запрещен, с указанием роли, необходимой для осуществления запроса.

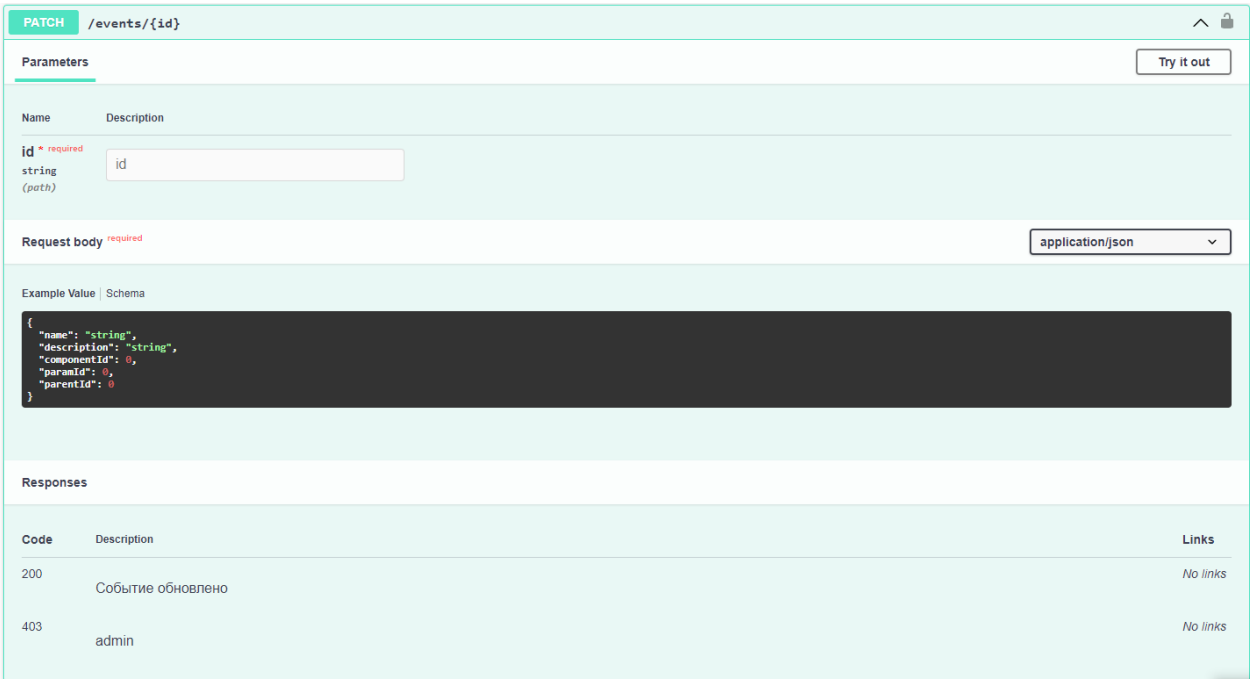

Рисунок 29 – Описание точки доступа /events/{id}

Полная документация доступна по адресу сервера с точкой доступа /api. Вдобавок на указанной странице можно напрямую отправлять запрос на сервер.

На диаграмме последовательности (рисунок 30) изображен запрос к REST-серверу с использованием токена доступа.

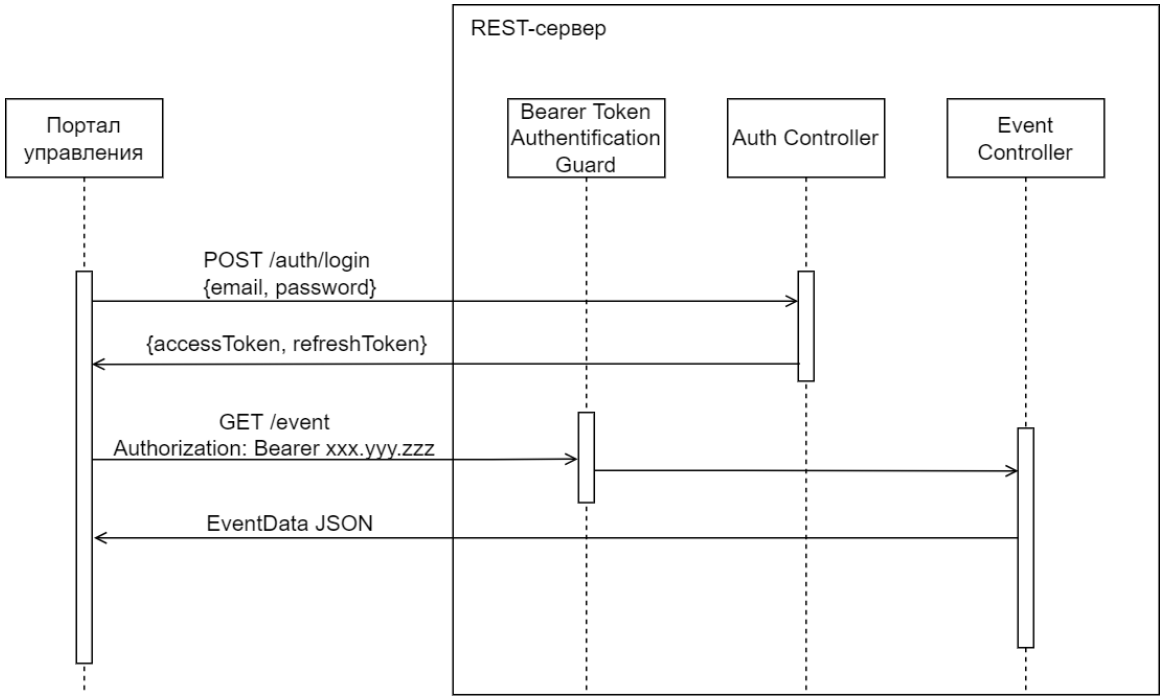

Рисунок 30 – Диаграмма последовательности взаимодействия с клиентским приложением

Важно отметить, что к программным средствам взаимодействия с клиентским приложением был разработан Guard, который отвечает за уровень доступа пользователя к той или иной точке API. Механизм Guard был подробно рассмотрен на этапе проектирования архитектуры. Токен доступа отправляется вместе с запросом в заголовке Authorization. Он имеет формат «Bearer TOKEN». Если уровень доступа не соответствует требуемому, то ответ возвращается с ошибкой «403 Forbidden».

Жизненный цикл запроса соответствует всем требованиям, поставленным на этапе проектирования. Он включает в себя: 2 промежуточных ПО (Middleware), 2 «защитника» (Guard), средства валидации данных запроса (Pipe DTO), 3 «перехватчика», глобальный фильтр исключений. Код данных программных средств представлен в приложении В.

Пример ошибки валидации представлен на рисунке 31.

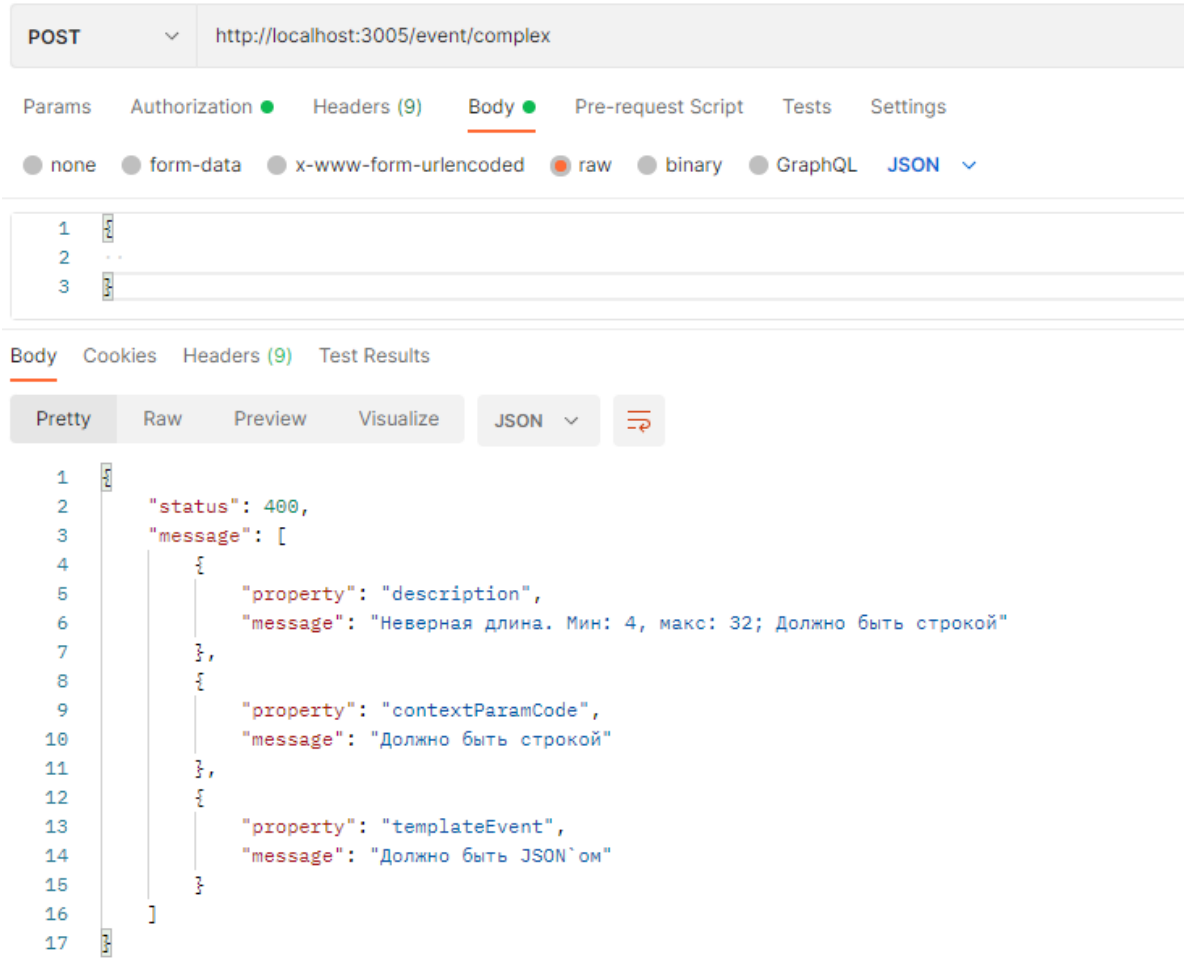

Рисунок 31 – Ответ от сервера с ошибочной первичной валидацией

Каждый контроллер обладает методами валидации тела или параметров запроса (DTO). Data Transfer Object (DTO) – один из шаблонов проектирования, используется для передачи данных. Валидация DTO была реализована при помощи декораторов, которые отвечают за валидацию входящих данных и в случае ошибки валидации генерируют исключение. Исключение декораторов обрабатывается фильтром исключений, который в свою очередь возвращает ответ клиенту с кодом 400 (Bad Request).

Объявление декораторов представлено в приложении А. За основу реализации декораторов была взята библиотека Class-Validator [19]. На листинге 3 показано использование декораторов для объявления параметров класса. Данные параметры входят в тело запроса создания активного правила.

Листинг 3 – Использование реализованных декораторов валидации

```
export class CreateRuleDto {
    @ApiProperty({
      type: String, description: 'Role description'
    })
    @IsStringRus()
    @LengthRus(4, 32)
    description: string;

    @ApiProperty({
      type: String, description: 'Condition json'
    })
    @IsJSONRus()
    condition: JSON;

    @ApiProperty({
      type: String, description: 'Action json'
    })
    @IsJSONRus()
    action: JSON;
}
```

Диаграмма состояний первичной проверки корректности запроса изображена на рисунке 32. За основу взят сценарий создания сложного события, который был рассмотрен на одном из прошлых этапов разработки.

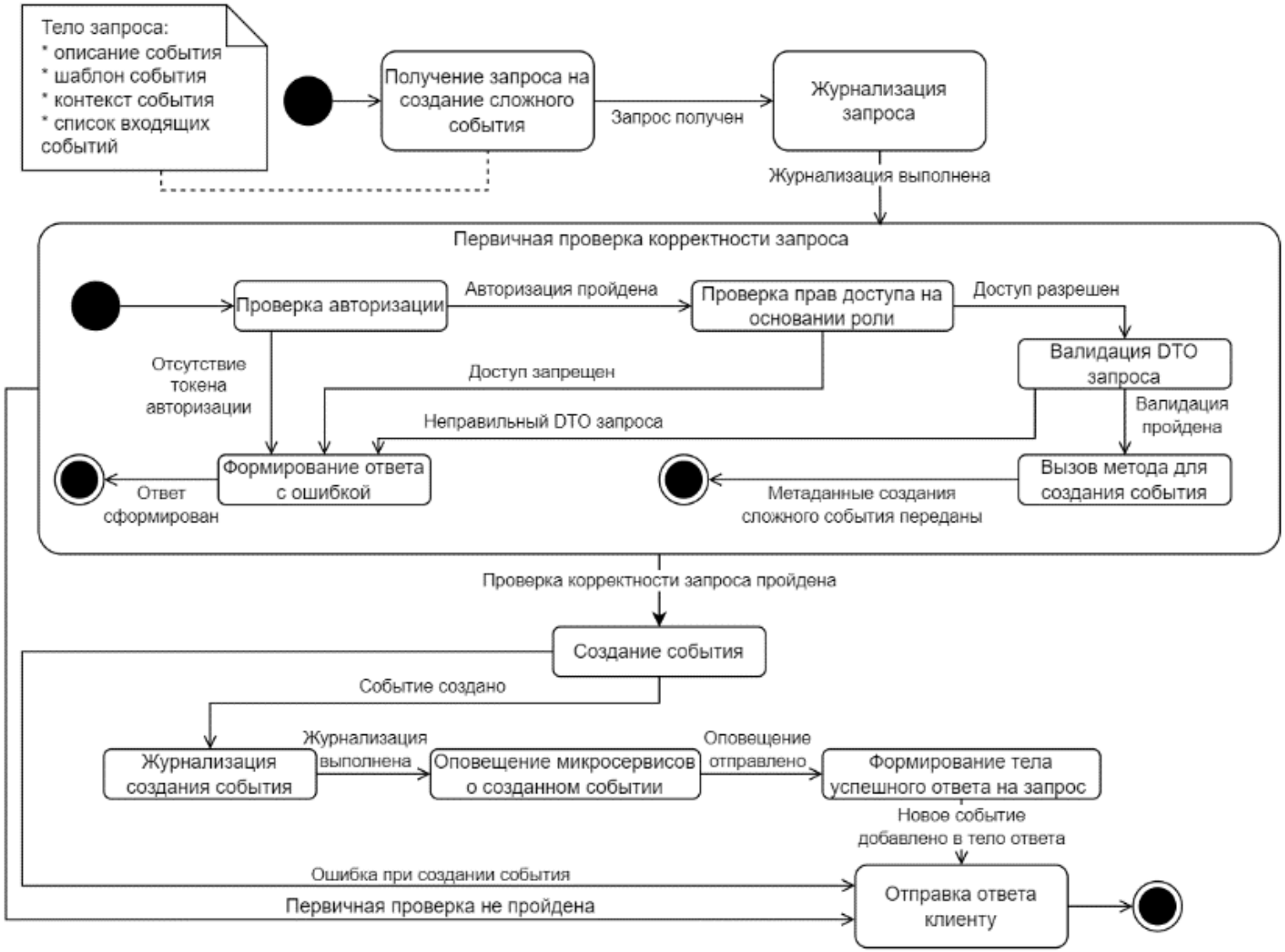

Рисунок 32 – Диаграмма создания сложного события с первичной проверкой запроса от клиентского приложения

Первичная проверка включает в себя три основных этапа: проверка авторизации, проверка прав доступа на основании роли, валидация DTO запроса.

3.4.2 Разработка программных средств доставки метаданных к микросервисам

Взаимодействие REST-сервера с микросервисами реализовано при помощи средств Redis, который разворачивается внутри Docker-контейнера.

Redis предоставляет возможность пересылки данных при помощи брокера сообщения. Данный механизм называется Pub/Sub. Механизм реализует парадигму публикации и читателя сообщений (рисунок 33).

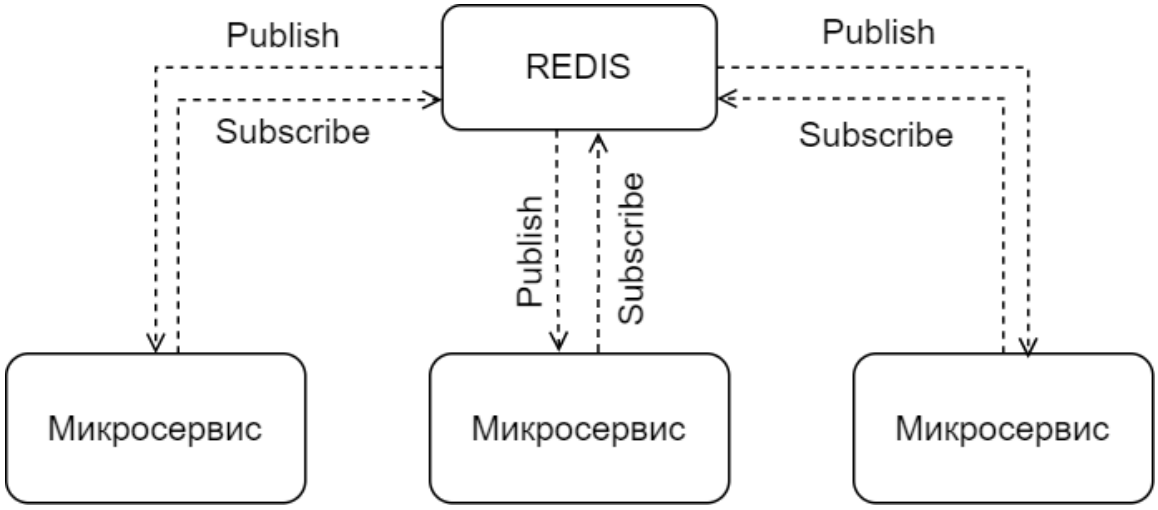

Рисунок 33 – Диаграмма коммуникации Redis Pub/Sub.

Необходимые для передачи данные должны быть опубликованы в канал брокера сообщений. Опубликованные сообщения доставляются только тем клиентам, кто был подписан на данный канал. Оъект сообщения состоит минимум из двух базовых элементов: данные полученные в результате пабликации другим клиентом. и имя исходного канала.

Реализация Redis Pub/Sub поддерживает сопоставление шаблонов. Клиенты могут подписаться на шаблоны в стиле glob, чтобы получать все сообщения, отправленные на названия каналов, соответствующие заданному шаблону. Также клиент может получить одно сообщение несколько раз, если он подписан на несколько идентичных шаблонов.

Для начала взаимодействия необходимо подключиться к транспортеру Redis. На листинге 4 показано подключение сервера к средствам Redis.

Листинг 4 – Подключение к Redis

```
this.client = ClientProxyFactory.create({
     transport: Transport.REDIS,
     options: {
          url: 'redis://localhost:6379',
     },
});
```

Каждый микросервис может подписаться на любое количество каналов. Стоит отметить, что Redis Pub/Sub используется в тех случаях, когда события могут быть потеряны. Если в момент публикации не будет ни одного подписчика, событие уйдет в никуда.

4 Контроль качества серверных программных средств

4.1 Планирование процесса контроля качества

4.1.1 Мероприятия по контролю качества программных средств

Контроль качества программного продукта – это набор мероприятий, которые гарантируют, что процессы, процедуры, а также стандарты подходят для проекта и реализуются правильно. Данный процесс выполняется параллельно с разработкой программного обеспечения. Он фокусируется на улучшении процесса разработки программного обеспечения, для предотвращения проблем.

Преимущества проведения данных мероприятий:

– помогают производить высококачественное программное обеспечение и экономить время и средства разработки;

– способствуют повышению надежности;

– выгодны в условиях отсутствия обслуживания в течение длительного времени.

Мероприятия по контролю качества серверных приложений можно разделить на две основные группы: анализ кода и подсчет метрик, и тестирование программных средств.

Метрики программного кода позволяют изучить сложность и стилистику разрабатываемого проекта, оценить возможность дальнейшей поддержки и масштабируемости ПО [20]. Метрики подразделяются на 7 основных классов:

– количественные метрики, основанные на количественных характеристиках исходного кода программ;

– метрики сложности потока управления программы, основанные на анализе управляющего графа программы;

– метрики сложности потока управления данными;

– метрики сложности потока управления и данных программы, которые устанавливают сложность структуры ПО как на основе количественных подсчетов, так и на основе анализа управляющих структур;

– объектно-ориентированные метрики;

– метрики надежности, которые основаны на количестве ошибок в программе;

– гибридные метрики, состоящие из двух или более ранее упомянутых метрик.

Тестирование программных средств бывает двух видов: автоматизированное и ручное. Автоматизированное тестирование – это метод тестирования программного обеспечения, который выполняется с использованием специальных программных средств, которые, в свою очередь необходимы для выполнения набора тестовых примеров. Напротив, ручное тестирование выполняется человеком, использующим разработанное программное средство и тщательно выполняющим каждый шаг теста «руками». Написанные автоматизированные тесты экономят множество времени и упрощают работу, помогая находить малозаметные ошибки. Также они помогают на этапах рефакторинга кода, давая уверенность в том, что код останется работоспособным. Можно выделить следующие типы автоматизированного тестирования:

– смоук тестирование;

– модульное тестирование;

– интеграционное тестирование;

– функциональное тестирование;

– проверка ключевых слов;

– регрессионное тестирование;

– тестирование на основе данных;

– тестирование черного ящика.

Выбор того или иного типа тестирования зависит от конкретного разрабатываемого программного обеспечения.

4.1.2 Обоснование выбора типов автоматизированного тестирования

Для серверных программных средств было проведено автоматическое тестирование. Были созданы Unit (модульные) и End-to-end (сквозные) тесты.

В коде разрабатываемого ПО много зависимостей: один метод ссылается на другой и влияет на разные части программы. Возникает необходимость проверять различные методы класса «в вакууме», отдельно от остального кода. Каждый из тестов является изолированным и независимым друг от друга. Данные автоматические сценарии позволяют убедиться, что отдельные части нашего приложения работают должным образом. К основным достоинствам модульных тестов можно отнести:

– позволяют обнаружить ошибки на ранних этапах разработки и исправить их;

– позволяют лучше понимать кодовую базу проекта и проще вносить изменения в готовое программное обеспечение;

– хорошие модульные тесты могут служить проектной документацией;

– помогают с миграцией кода.

End-to-end тестирование позволяет имитировать HTTP-запросы клиентских приложении. При помощи них могут быть обнаружены проблемы, возникающие при реальном взаимодействии. В отличие от модульного тестирования, которое фокусируется на отдельных модулях и классах, сквозное (end-to-end) тестирование охватывает взаимодействие классов и модулей на более совокупном уровне. Оно обеспечивает возможность выполнения повторяемых тестов при различных процессах приложений, возникающих в разных точках. По мере роста приложения становится трудно вручную тестировать поведение каждой конечной точки API. Автоматизированные сквозные тесты помогают убедиться, что общее поведение системы является правильным и соответствует требованиям проекта.

4.1.3 Обоснование выбора метрик программного кода

Для контроля качества были использованы метрики: анализ абстрактности и устойчивости программного обеспечения, метрика Холстеда для анализа программного кода.

Метрика устойчивости отражает количественную меру трудозатрат, которые необходимо произвести при изменении зависимых компонентов от анализируемого компонента. Метрика абстрактности показывает отношение количества абстрактных классов к общему числу классов в компоненте. На основании данных метрик может быть вычислено нормированное расстояние до главной последовательности, которое является результирующим показателем. Расстояние до главной последовательности – показатель, который отображает степень отдаленности компонента от данной последовательности. Чем значение ближе к нулю, тем качество компонента лучше. На рисунке 34 отображена главное последовательность, где на вертикальной оси располагается мера абстрактности, а на горизонтальной – мера устойчивости.

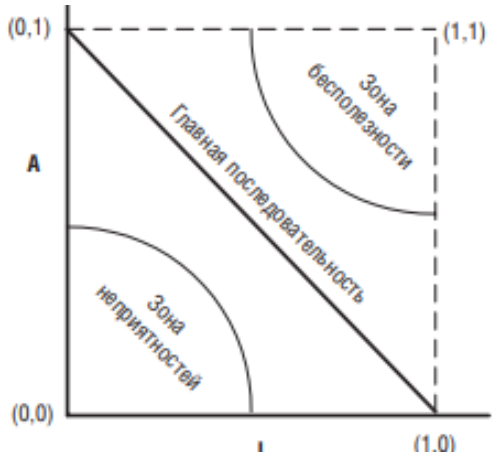

Рисунок 34 – Главная последовательность и зоны неприятностей

Метрика Холстеда относится к метрикам, вычисляемым на основании анализа числа строк и синтаксических элементов исходного кода программы. Она позволяет компенсировать недостатки, связанные с возможностью записи одной и той же бизнес-логики разным количеством строк и

операторов. Также позволяет оценить теоретический уровень и сложность программного обеспечения.

4.2 Автоматизированное тестирование

4.2.1 Модульное тестирование

Автоматизированное тестирование производилось при помощи фреймворка Jest. Он служит в качестве средства запуска тестов, а также предоставляет функции проверки и утилиты двойного тестирования (test-double), которые помогают имитировать отдельные классы и методы. Имитация происходит путем переопределения результатов вызова, утверждения корректности передаваемых объектов и т.п. Модульное (Unit) тестирование будет рассмотрено на примере компонента ролей пользователя, а конкретно на классах RoleController и RoleService. Первый из них отвечает за принятие HTTP-запросов от клиентских приложений, второй – за исполнение соответствующей бизнес-логики. В данном тестировании были созданы экземпляры классов с гарантией того, что контроллер и сервис выполняют свой контракт API.

В результате тестирования было проверено:

– корректное объявление методов в классах;

– корректность возвращаемых данных;

– создание новых объектов;

– поиск объектов по идентификатору;

– удаление объектов по идентификатору;

– обработчика ошибок при указании недействительных параметров поиска или параметров создания;

– корректное число вызовов необходимых методов.

Все компоненты прошли модульное тестирование. Результаты тестирования представлены на рисунке 35.

```
PASS src/user/role/test/role.service.spec.ts (19.87 s)
  RoleService
    √ should be defined (10 ms)
    √ should get the roles (3 ms)
    √ should add a new role (4 ms)
    √ should get a role (2 ms)
    √ should remove a role (2 ms)

PASS src/user/role/test/role.controller.spec.ts (24.22 s)
  RoleController
    √ should be defined (11 ms)
    √ should get the roles (4 ms)
    √ should make a new role (3 ms)
    √ should find a role (3 ms)
    √ should remove the role (3 ms)

Test Suites: 2 passed, 2 total
Tests:       10 passed, 10 total
Snapshots:   0 total
Time:        26.166 s
Ran all test suites.
```

Рисунок 35 – Результаты модульного тестирования

Код тестирования приведен в приложении Г.

4.2.2 Сквозное тестирование

Для выполнения сквозных тестов была использована конфигурация, аналогичная той, которая рассматривалась в модульном тестировании.

Код e2e тестирования приведен в приложении Г. REST-сервер использует защиту, которая обрабатывает запросы только от авторизированных пользователей. Для обхода этой защиты был реализован компонент MockAuthGuard, который позволяет осуществлять все запросы без авторизации.

Тестирование состоит из множества различных сценариев использования. Были использованы инструментальные средства для создания экземпляра полной среды выполнения, чтобы использовать ее для имитации HTTP-запросов.

В результате тестирования было проверено:

– наличие end-points (маршруты для отдельных HTTP запросов к REST-серверу);

– возможность модулей обрабатывать запросы;

– возвращаемые объекты от модулей сервера;

– тестирование Interceptor (перехватчика), который в результате возвращения пустого объекта генерирует ошибку 404 с поясняющей информацией;

– status-cod который возвращает REST-API приложение при успешном запросе.

Вдобавок к вышеперечисленному End-to-end тестирование помогло проверять поток приложения, увеличивая охват тестирования и уверенность в общей производительности серверных программных средств. Пример тестирования модуля ролей представлен на рисунке 36.

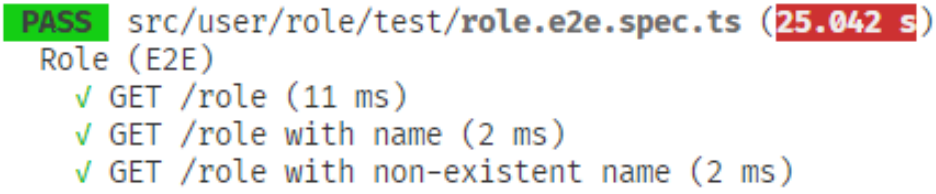

Рисунок 36 – Результаты тестирования

Модуль тестирования инициирует запрос к приложению, который идентичен фактическому HTTP-запросу, например GET /role, поступающему по сети. Все тесты были успешно пройдены.

4.3 Оценка качества программного кода

4.3.1 Метрика Холстеда

Для модулей REST-сервера были подсчитаны метрики Холстеда. Ниже будет рассмотрен анализ кода на примере модуля ролей пользователей. Он работает в совокупности с модулем JWT Auth, который отвечает за авторизацию пользователей в сервисе и разграничение доступа к различным методам. Также он отвечает за управление метаданных ролей через API.

Основу метрики Холстеда составляют четыре измеряемые характеристики программы:

– NUOprtr (Number of Unique Operators) – число уникальных операторов программы, включая символы-разделители, имена процедур и знаки операций (словарь операторов);

– NUOprnd (Number of Unique Operands) – число уникальных операндов программы (словарь операндов);

– Noprtr (Number of Operators) – общее число операторов в программе;

– Noprnd (Number of Operands) – общее число операндов в программе.

Код класса сервиса для модуля приведен в листинге 5. Операторы для модуля приведены в таблице 7. В данном программном коде имеется: 54 физических строк кода, 21 логических строк кода, 18 строк комментариев.

Листинг 5 – Класс RoleService

```
@Injectable()
export class RoleService {
    // DI сущности репозитория метаданных пользователя
    constructor(@InjectModel(Role) private roleRepository: typeof
Role) {}

    findAll(): Promise<Role[]> {
        return this.roleRepository.findAll();
    }

    findOne(id: number): Promise<Role> {
        return this.roleRepository.findByPk(id);
    }

    /**
     * Поиск роли по названию
     * @param value Название роли
     */
    findOneByValue(value: string): Promise<Role> {
        return this.roleRepository.findOne({ where: { value } });
    }

    /**
     * Создание роли
     * @param createRoleDto Параметры роли
     */
    async create(createRoleDto: CreateRoleDto): Promise<Role> {
        const candidate = await this.roleRepository.findOne({
            where: { value: createRoleDto.value },
        });
        if (candidate) {
            throw new BadRequestException('Данная роль уже
существует');
        }

        return this.roleRepository.create(createRoleDto);
    }
    /**
```

```
     * Обновление существующей роли
     * @param id Идентификатор роли
     * @param updateRoleDto Параметры роли
     */
    async update(id: number, updateRoleDto: UpdateRoleDto):
Promise<Role> {
        const role = await this.roleRepository.findByPk(id);
        if (!role) return;
        return role.update(updateRoleDto);
    }
    /**
     * Удаление роли по идентификатору
     * @param id Идентификатор роли
     */
    async remove(id: number): Promise<{ message: string }> {
        if (id <= 2)
          throw new BadRequestException('Данную роль удалять
запрещено');
        const destroyCount = await this.roleRepository.destroy({
            where: { id },
        });
        if (destroyCount === 0) return;
        return { message: 'Удалено объектов: ${destroyCount}' };
    }
}
```

Таблица 7 – Операторы

Операторы	Число вхождений
@Injectable()	1
return	8
this	7
.	22
!	1
update	1
findByPk	2
findOne	2
findAll	1
===	1
{}	13
;	18
,	4
if	4
await	3
new	2
throw	2

Продолжение таблицы 7

Операторы	Число вхождений
=	3
<=	1
$	1
n1 = 20	N1 = 97

Операнды для модуля приведены в таблице 8.

Таблица 8 – Операнды

Операнды	Число вхождений
createRoleDto	3
updateRoleDto	2
0	1
Id	7
2	1
destroyCount	3
`Удалено объектов:`	1
'Данную роль удалять запрещено'	1
role	3
where	3
value	3
n2=11	N2=28

Длина программы рассчитывается по формуле:

$$N = N1 + N2 = 50 + 76 = 125.$$

Словарь программы рассчитывается по формуле:

$$n = n1 + n2 = 17 + 21 = 31.$$

Объем программы рассчитывается по формуле:

$$HV = N * log2(n) = 125 * 4.95 = 618.75.$$

Потенциальный объем программы:

$$V^* = (2 + n`) * log2(2 + n`) = 7 * \log2(7) = 19.65.$$

Теоретический уровень программы рассчитывается по формуле:

$$L` = V^* / HV = 19.65 / 618.75 = 0.031.$$

Сложность программы рассчитывается по формуле:

$$HD = (n1 / 2) * (N2 / n2) = (17 / 2) * (28 / 11) = 21.6.$$

Все показатели находятся в норме. Поддержка кода в дальнейшем не вызовет трудностей.

4.3.2 Устойчивость и абстрактность

По результату разработки REST-сервера для доступа к единому репозиторию метаданных сервиса потоковой обработки событий были рассчитаны соотношения между абстрактностью и устойчивостью. Для вычисления устойчивости кода используется следующая формула:

$$I = \frac{Ce}{(Ca + Ce)},$$

где Ce – исходящие связи из компонента;

Ca – входящие связи в компонент.

Абстрактностью A называется величина

$$A = \frac{Na}{Nc},$$

где Nc – количество классов в компоненте;

Na – количество абстрактных в компоненте.

Метрика A изменяется от 0 до 1, причем 0 означает, что в компоненте вообще нет абстрактных классов, а 1 – что компонент содержит только абстрактные классы. Для вычисления данных соотношения основные программные классы были сгруппированы. Такие элементы соответствуют диаграмме пакетов, которая была представлена на этапе проектирования. Рассмотрим данные показатели.

Компонент User:

$$I_1 = \frac{Ce}{Ca + Ce} = \frac{2}{2+2} = \frac{1}{2};\ A_1 = \frac{Na}{Nc} = \frac{2}{7}.$$

Компонент имеет хорошую степень устойчивости, содержит 2 абстрактных класс.

Компонент JWT Auth:

$$I_2 = \frac{Ce}{Ca + Ce} = \frac{1}{1+1} = \frac{1}{2};\ A_2 = \frac{Na}{Nc} = \frac{1}{4}.$$

Компонент имеет хорошую степень устойчивости, содержит 1 абстрактный класс.

Компонент Events:

$$I_3 = \frac{Ce}{Ca + Ce} = \frac{1}{1+1} = \frac{1}{2};\ A_3 = \frac{Na}{Nc} = \frac{1}{6}.$$

Компонент имеет хорошую степень устойчивости, содержит 1 абстрактный класс.

Компоненты System, Active Rule, Component:

$$I_4 = \frac{Ce}{Ca + Ce} = \frac{1}{1+1} = \frac{1}{2};\ A_4 = \frac{Na}{Nc} = \frac{1}{3}.$$

Компоненты имеют хорошую степень устойчивости. Каждый из них содержит 1 абстрактный класс.

Компонент Postgre:

$$I_5 = \frac{Ce}{Ca + Ce} = \frac{1}{8+1} = \frac{1}{9};\ A_5 = \frac{Na}{Nc} = \frac{1}{2}.$$

Компоненты имеет не очень хорошую степень устойчивости, это обусловлено тем, что он часто импортируется в другие компоненты и играет важную роль. Реструктуризировать данный компонент не имеет смысла, так как он не содержит в себе какую-либо бизнес-логику и является средством взаимодействия с СУБД PostgreSQL. Компонент содержит 1 абстрактный класс.

На основе вычисленных метрик абстрактности и устойчивости для каждого компонента приложения было найдено нормированное расстояние до главной последовательности по формуле:

$$D = \left|\frac{(A + I - 1)}{\sqrt{2}}\right|,$$

где A – мера абстрактности;

I – мера неустойчивости.

Чем ближе точка располагается к главной последовательности (показатель ближе к 0), тем лучше. Расстояние до главной последовательности:

– $D_1 = \left|\frac{(0.29+0.5-1)}{\sqrt{2}}\right| = 0{,}15;$

– $D_2 = \left|\frac{(0.25+0.5-1)}{\sqrt{2}}\right| = 0{,}18;$

– $D_3 = \left|\frac{(0.17+0.5-1)}{\sqrt{2}}\right| = 0{,}26;$

– $D_4 = \left|\frac{(0.33+0.5-1)}{\sqrt{2}}\right| = 0{,}12;$

– $D_5 = \left|\frac{(0.5+0.11-1)}{\sqrt{2}}\right| = 0{,}27.$

Произведя расчеты, можно отобразить компоненты на графике (рисунок 37). На вертикальной оси отображен показатель абстрактности A, а на горизонтальной оси мера устойчивости I.

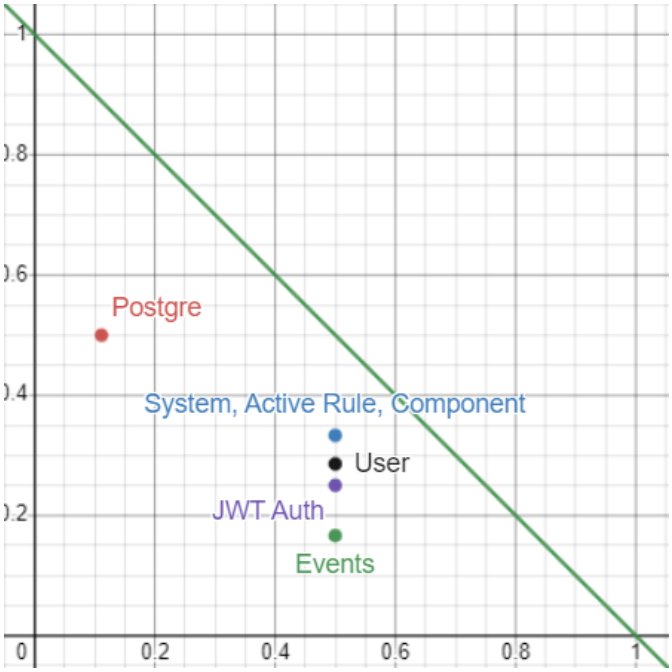

Рисунок 37 – Отображение метрик устойчивости и абстрактности

Все компоненты лежат близко к главной последовательности. Это означает, что отношение устойчивости и абстрактности очень хорошее.

Заключение

В ходе выполнения выпускной квалификационной работы были разработаны серверные программные средства управления метаданными для сервиса потоковой обработки событий.

Проведен анализ предметной области и выделены функциональные возможности сервиса потоковой обработки событий. Была описана его архитектура и сценарии работы. Также проведен обзор аналогичных работ в заданной предметной области, выделены достоинства и недостатки описываемых ПО, которые были учтены на этапах проектирования и разработки.

Были описаны механизмы управления метаданными для сервиса потоковой обработки событий, их жизненный цикл и состав. Разработаны требования к программному обеспечению. Проведено планирование и оценка бюджета проекта при помощи средств MS Project. Затраты на проект составили 130200 рублей. Время разработки 56 рабочих дней.

Проведен этап проектирования, разрабатываемых программных средств, который включал в себя: проектирование архитектуры, разработку функциональных требований, трехуровневое моделирование репозитория метаданных, проектирование модулей REST-сервера. На этапе проектирования был обоснован выбор технологий и средств разработки. Были выделены требования к аппаратно-программного обеспечению.

Разработаны серверные программные средства. Основными этапами разработки стали: разработки архитектуры развертывания, разработка базы данных и программных средств взаимодействия с ней, разработка механизмов управления метаданными и алгоритмы анализа их корректности, разработка средств регистрации и аутентификации пользователей, разработка средств взаимодействия с клиентскими приложениями и микросервисами. Разработка проводилась в редакторе исходного кода Visual Studio Code на языке программирования TypeScript с использованием фреймворка Nest. В качестве исполняемой платформы была выбрана Node.js, основанная на

движке V8. В качестве web-сервера был использован Nginx. Программные средства разворачивались внутри Docker-контейнеров, что добавляло им быструю переносимость и относительную независимость к прикладному и системному программному обеспечению.

Проведен контроль качества серверных программных средств. Были составлены автоматизированные тесты. Также были подсчитаны метрики устойчивости и абстрактности и метрики Холстеда.

Все этапы сопровождались диаграммами и пояснительным текстом к ним. Разработанное программное обеспечение отвечает всем требованиям технического задания.

Список использованных источников

1. Adaikkalavan R. Active Authorization Rules for Enforcing Role-Based Access Control and its Extensions / R. Adaikkalavan, S. Chakravarthy // 21st International Conference on Data Engineering Workshops – Tokyo : Изд-во IEEE, 2005. – С. 220 – 222.

2. Прошкина Е. Н. Язык моделирования информационных процессов UML 2.0 / Прошкина Е. Н., Балашова И. Ю. – Пенза : Изд-во ПГУ, 2018. – 112 с.

3. Adaikkalavan R., Chakravarthy S., When to Trigger Active Rules? // DBLP Computer Science Bibliography [Электронный ресурс]. URL: https://dblp.uni-trier.de/rec/conf/comad/AdaikkalavanC09.html (дата обращения 05.03.2022).

4. Шибанов С. В., Скоробогатько А. А., Э. В. Лысенко. Интегрированная модель активных правил / С. В. Шибанов, А. А. Скоробогатько, Э. В. Лысенко // Математическое и программное обеспечение систем в промышленной и социальной сферах: междунар. сб. науч. трудов – Магнитогорск : Изд-во Магнитогорск. гос. техн. ун-та им. Г.И. Носова, 2011. – С. 41 – 46.

5. Pattern: Database per service // Microservice Architecture [Электронный ресурс]. URL: https://microservices.io/patterns/data/database-per-service.html (дата обращения 05.03.2022).

6. Towards an Integrated Model for Event and Stream Processing / Q. Jiang, R. Adaikkalavan, S. Chakravarthy // Technical Report CSE – Texas : Изд-во CSE, 2004. – 35 с.

7. Sebastian Peyrott JWT Handbook // Auth0 Portal [Электронный ресурс]. URL: https://auth0.com/resources/ebooks/jwt-handbook (дата обращения 10.11.2021).

8. Diaz O., Jaime A. and Paton, N.W. DEAR: a debugger for active rules in an object-oriented context / O. Díaz, A. Jaime, N. Paton // Rules in Database Systems – Scotland : Изд-во Springer, 1993. – С. 180 – 193.

9. Fors T. ADELA: animated debugging and explanation of active database rules / T. Fors // MSc. dissertation, department of computer science – Sweden : Изд-во University of Skovde, 1995. – С. 215 – 231.

10. Kung D., Fegaras L. Design and implementation of a dynamic programming environment for active rules / D. Kung, L. Fegaras – Texas, Изд-во: The University of Texas at Arlington, 2003. – 106 с.

11. Visual Paradigm Quick Start // [Электронный ресурс]. URL: https://cdn.visual-paradigm.com/quickstart/quickstart.pdf (дата обращения: 20.01.2021).

12. Kruchten P. The Rational Unified Process: An Introduction 3rd Edition / P. Kruchten – Boston, Издательство: Addison-Wesley, 2003. – 332 с.

13. Documentation | NestJS // Docs Nestjs [Электронный ресурс]. URL: https://docs.nestjs.com (дата обращения 18.09.2021).

14. Documentation Jest // Jest Io [Электронный ресурс]. URL: https://jestjs.io/ru/ (дата обращения 11.10.2021).

15. Overview of Docker // Docker docs [Электронный ресурс]. URL: https://docs.docker.com (дата обращения 02.02.2022).

16. Swagger for Everyone // Swagger Io [Электронный ресурс]. URL: https://swagger.io (дата обращения 02.02.2022).

17. NGINX Cookbook, 2019 Edition // Nginx [Электронный ресурс]. URL: https://www.nginx.com/resources/library/nginx-cookbook-2019-edition (дата обращения 07.11.2021).

18. Cano J., Delaval G. Correct coordination of ECA rules by verification and control / J. Cano, G. Delaval, E. Rutten // Coordination of ECA Rules by Verification and Control – Berlin : Изд-во Springer, 2014. – С. 180 – 193.

19. Class-validator, Decorator-based property validation for classes // GitHub [Электронный ресурс]. URL: https://github.com/typestack/class-validator (дата обращения 14.04.2022).

20. Intel. Программный код и его метрики // Habr [Электронный ресурс]. URL: https://habr.com/ru/company/intel/blog/106082 (дата обращения 07.11.2021).

Приложение А

ГЛОССАРИЙ

(справочное)

Промышленный интернет вещей (англ. Industrial Internet of Things) – многоуровневая система, включающая в себя датчики и контроллеры, установленные на узлах и агрегатах промышленного объекта, средства передачи собираемых данных и их визуализации, мощные аналитические инструменты интерпретации получаемой информации и многие другие компоненты.

База данных (БД) – это информативная модель, позволяющая организованно поддерживать данные о группе объектов, имеющих один и тот же набор параметров.

Unified Modeling Language (UML) – графический унифицированный язык моделирования, использующий объектно-ориентированный подход для моделирования в сфере разработки программного обеспечения.

Event condition action (ECA) – модель структуры активных правил, в которой содержатся три главных элемента: событие, условие, действие.

Обработка потоков событий (англ. Event Stream Processing, ESP) – набор технологий, предназначенных для построения информационных систем обработки событий, которые включают в себя визуализацию событий, их хранение, связующее программное обеспечение и языки программирования обработки событий.

Data Transfer Object (DTO) – один из шаблонов проектирования, используется для передачи данных между подсистемами приложения.

Приложение Б

КОД СЕРВИСОВ У МОДУЛЕЙ EVENT, ACTIVE RULE, AUTH

(рекомендуемое)

Листинг Б.1 – Файл event.service.ts

```
@Injectable()
export class EventsService {
    constructor(
        @InjectModel(Event) private eventRepository: typeof Event,
        @InjectModel(ContextParam) private paramRepository: typeof
ContextParam,
        @InjectModel(Component) private componentRepository: typeof
Component,
    ) {}

    readonly findModel = {
        attributes: ['code', 'description', 'categoryEvent'],
        include: [
            {
                model: ContextParam,
                attributes: ['code', 'name', 'description', 'dataType'],
            },
            {
                model: ActiveRule,
                attributes: ['code', 'description', 'condition', 'action'],
                through: { attributes: [] },
            },
            {
                model: ComplexEvent,
                as: 'complexEvent',
                attributes: ['templateEvent'],
                include: [
                    {
                        model: Event,
                        as: 'inboundEvent',
                        attributes: ['code', 'description', 'categoryEvent'],
                        through: { attributes: [] },
                    },
                ],
            },
            {
                model: ComplexEvent,
                as: 'inboundComplexEvent',
                attributes: ['templateEvent'],
                through: { attributes: [] },
                include: [
                    {
                        model: Event,
                        as: 'event',
                        attributes: ['code', 'description', 'categoryEvent'],
                    },
                ],
            },
            {
                model: AggregationEvent,
                attributes: [
                    'aggregationQuery',
                    'unixWindowStart',
                    'unixWindowEnd',
                ],
                include: [
                    {
                        model: AtomicEvent,
                        as: 'atomicEvent',
                        include: [
                            {
```

```
                                model: Component,
                                as: 'component',
                                attributes: ['code', 'description'],
                            },
                            {
                                model: Event,
                                attributes: [
                                    'code',
                                    'description',
                                    'categoryEvent',
                                ],
                            },
                        ],
                        attributes: ['codePair'],
                        through: { attributes: [] },
                    },
                ],
            },
        ],
    };

    private clearOut(event: Event) {
        let clearEvent = event.toJSON() as any;
        if (!clearEvent.complexEvent) delete clearEvent.complexEvent;
        if (!clearEvent.aggregationEvent) delete clearEvent.aggregationEvent;
        if (clearEvent.categoryEvent === CategoryEvent.AGGREGATION) {
            clearEvent = { ...clearEvent, ...clearEvent.aggregationEvent };
            delete clearEvent.aggregationEvent;
        }
        if (clearEvent.categoryEvent === CategoryEvent.COMPLEX) {
            clearEvent = { ...clearEvent, ...clearEvent.complexEvent };
            delete clearEvent.complexEvent;
        }
        return clearEvent;
    }

    async findAll() {
        const events = await this.eventRepository.findAll(this.findModel);
        const clearOut = [];
        events.forEach((ev) => clearOut.push(this.clearOut(ev)));
        return clearOut;
    }

    async findOne(code: string) {
        const event = await this.eventRepository.findOne({
            where: { code },
            ...this.findModel,
        });
        return this.clearOut(event);
    }

    async findContextParam(code: string) {
        const param = await this.paramRepository.findOne({ where: { code }
});
        if (!param) throw new NotFoundException('Параметр не найден');
        return param;
    }

    async findComponent(code: string) {
        const component = await this.componentRepository.findOne({
            where: { code },
        });
        if (!component) throw new NotFoundException('Компонент не найден');
        return component;
    }
}
```

Листинг Б.2 – Файл event-complex.service.ts

```
@Injectable()
export class EventsComplexService {
    constructor(
        @InjectModel(Event) private eventRepository: typeof Event,
        @InjectModel(InboundComplexEvent)
        private inboundRepository: typeof InboundComplexEvent,
        @InjectModel(ComplexEvent)
        private complexRepository: typeof ComplexEvent,
        private readonly counterService: CounterService,
        private readonly eventService: EventsService,
    ) {}

    private async findComplex(code: string, context?: boolean) {
        const modelCompl = [
            { model: ComplexEvent, as: 'complexEvent' },
            { model: ComplexEvent, as: 'inboundComplexEvent' },
        ];
        const includeModel = [...modelCompl];
        const includeModelContext = [...modelCompl, { model: ContextParam }];
        const event = await this.eventRepository.findOne({
            where: { code },
            include: context ? includeModelContext : includeModel,
        });
        if (!event) throw new NotFoundException('Сложное событие не
найдено');
        if (event.categoryEvent != CategoryEvent.COMPLEX)
            throw new BadRequestException('Данное событие не является
сложным');
        return event;
    }

    async createComplex(dto: CreateComplexEventDto) {
        const param = await this.eventService.findContextParam(
            dto.contextParamCode,
        );
        const codeEvent = await this.counterService.getNewCode(
            CounterName.EVENT,
        );
        const createdTypeEvent = await this.eventRepository.create({
            code: codeEvent,
            categoryEvent: CategoryEvent.COMPLEX,
            contextParamId: param.id,
            description: dto.description,
        });
        await this.complexRepository.create({
            eventId: createdTypeEvent.id,
            templateEvent: dto.templateEvent,
        });
        return this.eventService.findOne(codeEvent);
    }

    async updateComplex(dto: UpdateComplexEventDto) {
        const event = await this.findComplex(dto.code, true);
        if (dto.contextParamCode) {
            const param = await this.eventService.findContextParam(
                dto.contextParamCode,
            );
            event.contextParamId = param.id;
        }
        if (dto.contextParamCode) {
            if (dto.contextParamCode === event.contextParam.code) {
                throw new BadRequestException(
```

```
                    'Данный параметр контекста уже установлен',
                );
            }
            const param = await this.eventService.findContextParam(
                dto.contextParamCode,
            );
            event.contextParamId = param.id;
        }
        if (dto.description) event.description = dto.description;
        if (dto.templateEvent)
            event.complexEvent.templateEvent = dto.templateEvent;
        await event.complexEvent.save();
        await event.save();
        return this.eventService.findOne(event.code);
    }

    async removeComplex(code: string) {
        const event = await this.findComplex(code);
        if (
            await this.inboundRepository.findOne({
                where: { eventId: event.id },
            })
        ) {
            throw new BadRequestException(
                'Данное сложное сообытие входит в состав другого сложного',
            );
        }
        await event.complexEvent.destroy();
        await event.destroy();
        return 'Сложное событие удалено';
    }

    async addInbound(dto: AddInboundEvent) {
        if (dto.codeAnother == dto.codeComplex)
            throw new BadRequestException(
                'Событие запрещено включать в самого себя',
            );
        const event = await this.findComplex(dto.codeComplex);
        const eventAdd = await this.eventRepository.findOne({
            where: { code: dto.codeAnother },
        });
        if (!eventAdd)
            throw new NotFoundException('Добавляемое событие не найдено');
        if (
            await this.inboundRepository.findOne({
                where: {
                    complexEventId: event.complexEvent.id,
                    eventId: eventAdd.id,
                },
            })
        ) {
            throw new BadRequestException(
                'Добавляемое событие уже входит в состав данного сложного
события',
            );
        }
        await this.inboundRepository.create({
            complexEventId: event.complexEvent.id,
            eventId: eventAdd.id,
        });
        return this.eventService.findOne(dto.codeComplex);
    }

    async removeInbound(dto: RemoveInboundEvent) {
        const event = await this.findComplex(dto.codeComplex);
        const eventRem = await this.eventRepository.findOne({
```

```
            where: { code: dto.codeAnother },
        });
        if (!eventRem)
            throw new NotFoundException('Удаляемое событие не найдено');
        const inbound = await this.inboundRepository.findOne({
            where: {
                complexEventId: event.complexEvent.id,
                eventId: eventRem.id,
            },
        });
        if (!inbound) {
            throw new BadRequestException(
                'Удаляемое событие не входит в состав данного сложного
события',
            );
        }
        await inbound.destroy();
        return this.eventService.findOne(dto.codeComplex);
    }
}
```

Листинг Б.3 – Файл event-atomic.service.ts

```
@Injectable()
export class EventsAtomicService {
    constructor(
        @InjectModel(Event) private eventRepository: typeof Event,
        @InjectModel(AtomicEvent) private atomicRepository: typeof
AtomicEvent,
        @InjectModel(InboundComplexEvent)
        private inboundRepository: typeof InboundComplexEvent,
        private readonly counterService: CounterService,
        private readonly eventService: EventsService,
    ) {}

    async findAtomic(code: string) {
        const event = await this.eventRepository.findOne({
            where: { code },
            include: [{ model: Component }, { model: AtomicEvent }],
        });
        if (!event) throw new NotFoundException('Атомарное событие не
найдено');
        if (event.categoryEvent != CategoryEvent.ATOMIC)
            throw new BadRequestException(
                'Данное событие не является атомарным',
            );
        return event;
    }

    async createAtomic(dto: CreateAtomicEventDto) {
        const component = await this.eventService.findComponent(
            dto.codeComponent,
        );
        const param = await this.eventService.findContextParam(
            dto.contextParamCode,
        );
        // Поиск повтора объявления
        const candidateAll = await this.eventRepository.findAll({
            where: { contextParamId: param.id },
            include: [{ model: Component }],
        });
        for await (const candidate of candidateAll)
            if (candidate.component.find((el) => el.id === component.id)) {
                throw new BadRequestException(
```

```
                    'Данное событие уже существует. Code: ${candidate.code}',
                );
            }
        const codeEvent = await this.counterService.getNewCode(
            CounterName.EVENT,
        );
        const created = await this.eventRepository.create({
            code: codeEvent,
            categoryEvent: CategoryEvent.ATOMIC,
            contextParamId: param.id,
            description: dto.description,
        });
        await this.atomicRepository.create({
            componentId: component.id,
            eventId: created.id,
            codePair: '${component.code}-${codeEvent}',
        });
        return this.eventService.findOne(codeEvent);
    }

    async updateAtomic(dto: UpdateAtomicEventDto) {
        const event = await this.eventRepository.findOne({
            where: { code: dto.code },
            include: [{ model: Component }, { model: ContextParam }],
        });
        if (!event) throw new NotFoundException('Событие не найдено');
        if (dto.codeComponent)
            throw new BadRequestException(
                'Запрещено напрямую редактировать список компонентов',
            );
        if (dto.contextParamCode) {
            if (dto.contextParamCode === event.contextParam.code) {
                throw new BadRequestException(
                    'Данный параметр контекста уже установлен',
                );
            }
            const param = await this.eventService.findContextParam(
                dto.contextParamCode,
            );
            event.contextParamId = param.id;
        }
        if (dto.description) event.description = dto.description;
        await event.save();
        return this.eventService.findOne(event.code);
    }

    async addComponentAtomic(dto: AddComponentAtomic) {
        const event = await this.findAtomic(dto.codeEvent);
        const component = await this.eventService.findComponent(
            dto.codeComponent,
        );
        if (
            event.atomicEvent &&
            event.atomicEvent.find((el) => el.componentId === component.id)
        ) {
            throw new BadRequestException(
                'Данное атомарное событие уже существует, повтор
конфигурации',
            );
        }
        await this.atomicRepository.create({
            componentId: component.id,
            eventId: event.id,
            codePair: '${component.code}-${event.code}',
        });
        return this.eventService.findOne(dto.codeEvent);
```

```
    }

    async removeComponentAtomic(dto: RemoveComponentAtomic) {
        const event = await this.findAtomic(dto.codeEvent);
        if (!event.atomicEvent || event.atomicEvent.length < 2)
            throw new BadRequestException(
                'Элементарное событие не может быть без источника',
            );
        const component = await this.eventService.findComponent(
            dto.codeComponent,
        );
        const atomic = await this.atomicRepository.findOne({
            where: { componentId: component.id, eventId: event.id },
            include: [{ model: AggregationEvent }],
        });
        if (!atomic)
            throw new BadRequestException(
                'Данное событие исходит не от заданного компонента',
            );
        if (atomic.aggregationEvent && atomic.aggregationEvent.length > 0) {
            throw new BadRequestException(
                'Данное атомарное сообытие входит в состав агрегирующего',
            );
        }
        await atomic.destroy();
        return this.eventService.findOne(dto.codeEvent);
    }

    async removeAtomic(dto: RemoveAtomicEvent) {
        const event = await this.findAtomic(dto.codeEvent);
        const component = await this.eventService.findComponent(
            dto.codeComponent,
        );
        const atomic = await this.atomicRepository.findOne({
            where: { componentId: component.id, eventId: event.id },
            include: [{ model: AggregationEvent }],
        });
        if (!atomic)
            throw new BadRequestException(
                'Данное событие исходит не от заданного компонента',
            );
        if (atomic.aggregationEvent && atomic.aggregationEvent.length > 0) {
            throw new BadRequestException(
                'Данное атомарное сообытие входит в состав агрегирующего',
            );
        }
        if (
            await this.inboundRepository.findOne({
                where: { eventId: event.id },
            })
        ) {
            throw new BadRequestException(
                'Данное атомарное сообытие входит в состав сложного',
            );
        }
        await event.destroy();
        return 'Атомарное событие удалено';
    }
}
```

Листинг Б.4 – Файл event-aggregation.service.ts

```
@Injectable()
export class EventsAggregationService {
```

```
    constructor(
        @InjectModel(Event) private eventRepository: typeof Event,
        @InjectModel(InboundComplexEvent)
        private inboundRepository: typeof InboundComplexEvent,
        @InjectModel(AggregationEvent)
        private aggregationRepository: typeof AggregationEvent,
        private readonly counterService: CounterService,
        private readonly eventService: EventsService,
        private readonly atomicService: EventsAtomicService,
    ) {}

    private async findAggregation(code: string, context?: boolean) {
        const modelAggr = {
            model: AggregationEvent,
            include: [{ model: AtomicEvent }],
        };
        const includeModel = [modelAggr];
        const includeModelContext = [modelAggr, { model: ContextParam }];
        const event = await this.eventRepository.findOne({
            where: { code },
            include: context ? includeModelContext : includeModel,
        });
        if (!event)
            throw new NotFoundException('Агрегирующее событие не найдено');
        if (event.categoryEvent != CategoryEvent.AGGREGATION)
            throw new BadRequestException(
                'Данное событие не является агрегирующим',
            );
        return event;
    }

    async createAggregation(dto: CreateAggregationEventDto) {
        const param = await this.eventService.findContextParam(
            dto.contextParamCode,
        );
        const codeEvent = await this.counterService.getNewCode(
            CounterName.EVENT,
        );
        const createdTypeEvent = await this.eventRepository.create({
            code: codeEvent,
            categoryEvent: CategoryEvent.AGGREGATION,
            contextParamId: param.id,
            description: dto.description,
        });
        const unixWindow = {};
        if (dto.unixWindowStart)
            unixWindow['unixWindowStart'] = dto.unixWindowStart;
        if (dto.unixWindowEnd) unixWindow['unixWindowEnd'] =
dto.unixWindowEnd;
        await this.aggregationRepository.create({
            eventId: createdTypeEvent.id,
            aggregationQuery: dto.aggregationQuery,
            ...unixWindow,
        });
        return this.eventService.findOne(codeEvent);
    }

    async updateAggregation(dto: UpdateAggregationEventDto) {
        const event = await this.findAggregation(dto.code, true);
        if (dto.contextParamCode) {
            const param = await this.eventService.findContextParam(
                dto.contextParamCode,
            );
            console.log('test');
            event.contextParamId = param.id;
        }
```

```
        if (dto.contextParamCode) {
            if (dto.contextParamCode === event.contextParam.code) {
                throw new BadRequestException(
                    'Данный параметр контекста уже установлен',
                );
            }
            const param = await this.eventService.findContextParam(
                dto.contextParamCode,
            );
            event.contextParamId = param.id;
        }
        if (dto.description) event.description = dto.description;
        if (dto.unixWindowStart)
            event.aggregationEvent.unixWindowStart = dto.unixWindowStart;
        if (dto.unixWindowEnd)
            event.aggregationEvent.unixWindowEnd = dto.unixWindowEnd;
        if (dto.aggregationQuery)
            event.aggregationEvent.aggregationQuery = dto.aggregationQuery;
        await event.aggregationEvent.save();
        await event.save();
        return this.eventService.findOne(event.code);
    }

    async removeAggregation(code: string) {
        const event = await this.findAggregation(code);
        if (
            await this.inboundRepository.findOne({
                where: { eventId: event.id },
            })
        ) {
            throw new BadRequestException(
                'Данное аггрегирующее сообытие входит в состав сложного',
            );
        }
        await event.aggregationEvent.destroy();
        await event.destroy();
        return 'Аггрегирующее событие удалено';
    }

    async addAtomicToAggregation(dto: AddAtomicToAggregationDto) {
        const event = await this.atomicService.findAtomic(dto.codeAtomic);
        const component = event.component.find(
            (el) => el.code === dto.codeComponentAtomic,
        );
        if (!component)
            throw new NotFoundException(
                'Компонент у данного события не найден',
            );
        const atomic = event.atomicEvent.find(
            (el) => el.eventId === event.id && el.componentId ===
component.id,
        );
        const aggregation = await this.findAggregation(dto.codeAggregation);
        if (
            aggregation.aggregationEvent.atomicEvent.find(
                (el) => el.id === atomic.id,
            )
        ) {
            throw new BadRequestException(
                'Данное атомарное сообытие уже входит в состав
аггрегирующего',
            );
        }
        await aggregation.aggregationEvent.$add('AtomicEvent', atomic.id);
        return this.eventService.findOne(aggregation.code);
    }
```

```
    async removeAtomicToAggregation(dto: RemoveAtomicToAggregationDto) {
        const event = await this.atomicService.findAtomic(dto.codeAtomic);
        const component = event.component.find(
            (el) => el.code === dto.codeComponentAtomic,
        );
        if (!component)
            throw new NotFoundException(
                'Компонент у данного события не найден',
            );
        const atomic = event.atomicEvent.find(
            (el) => el.eventId === event.id && el.componentId ===
component.id,
        );
        const aggregation = await this.findAggregation(dto.codeAggregation);
        if (
            !aggregation.aggregationEvent.atomicEvent.find(
                (el) => el.id === atomic.id,
            )
        ) {
            throw new BadRequestException(
                'Данное атомарное сообытие не входит в состав
аггрегирующего',
            );
        }
        await aggregation.aggregationEvent.$remove('AtomicEvent', atomic.id);
        return this.eventService.findOne(aggregation.code);
    }
}
```

Листинг Б.5 – Файл active-rule.service.ts

```
@Injectable()
export class ActiveRuleService {
    constructor(
        @InjectModel(ActiveRule) private ruleRep: typeof ActiveRule,
        @InjectModel(ActiveRuleEvent)
        private ruleEventRep: typeof ActiveRuleEvent,
        @InjectModel(Event) private eventRep: typeof Event,
        private readonly counterService: CounterService,
    ) {}

    readonly findModel = {
        attributes: ['code', 'description', 'condition', 'action'],
        include: {
            model: Event,
            attributes: [
                'code',
                'description',
                'categoryEvent',
                'contextParamId',
            ],
            through: { attributes: [] },
        },
    };

    findAll() {
        return this.ruleRep.findAll(this.findModel);
    }

    findOne(code: string) {
        return this.ruleRep.findOne({ where: { code }, ...this.findModel });
    }
```

```
    async create(dto: CreateRuleDto) {
        const ruleCode = await
this.counterService.getNewCode(CounterName.RULE);
        await this.ruleRep.create({ ...dto, code: ruleCode });
        return this.findOne(ruleCode);
    }

    async update(dto: UpdateRuleDto) {
        const rule = await this.ruleRep.findOne({ where: { code: dto.code }
});
        if (!rule) throw new NotFoundException('Активное правило не
найдено');
        await rule.update(dto);
        return this.findOne(rule.code);
    }

    async remove(code: string) {
        const rule = await this.ruleRep.findOne({
            where: { code },
            include: { all: true },
        });
        if (!rule) throw new NotFoundException('Активное правило не
найдено');
        await rule.destroy();
        return 'Активное првило удалено';
    }

    async bindEvent(dto: BindEventDto) {
        const rule = await this.ruleRep.findOne({
            where: { code: dto.codeRule },
        });
        if (!rule) throw new NotFoundException('Активное правило не
найдено');

        const event = await this.eventRep.findOne({
            where: { code: dto.codeEvent },
        });
        if (!event) throw new NotFoundException('Событие не найдено');

        const bind = await this.ruleEventRep.findOne({
            where: {
                eventId: event.id,
                activeRuleId: rule.id,
                typeBind: dto.typeBind,
            },
        });
        if (bind)
            throw new NotFoundException('Данная конфигурация уже
существует');

        await this.ruleEventRep.create({
            eventId: event.id,
            activeRuleId: rule.id,
            typeBind: dto.typeBind,
        });
        return this.findOne(rule.code);
    }

    async unbindEvent(dto: UnbindEventDto) {
        const rule = await this.ruleRep.findOne({
            where: { code: dto.codeRule },
            include: [{ model: Event }],
        });
        if (!rule) throw new NotFoundException('Активное правило не
найдено');
        const event = rule.event.find((ev) => ev.code === dto.codeEvent);
```

```
        if (!event) {
            throw new BadRequestException(
                'Правило не привязанно к данному событию',
            );
        }
        const bind = await this.ruleEventRep.findOne({
            where: {
                eventId: event.id,
                activeRuleId: rule.id,
                typeBind: dto.typeBind,
            },
        });
        if (!bind)
            throw new NotFoundException(
                'Данное событие не связано с правилом по такому типу',
            );

        await bind.destroy();
        return this.findOne(rule.code);
    }
}
```

Листинг Б.6 – Файл mail.service.ts

```
@Injectable()
export class MailService implements OnModuleInit {
    private transporter: nodemailer.Transporter<SentMessageInfo>;
    private readonly logger = new Logger(MailService.name);

    constructor(@InjectModel(User) private userRepository: typeof User) {}

    onModuleInit() {
        this.transporter = nodemailer.createTransport(
            {
                host: process.env.SMTP_HOST,
                port: +process.env.SMTP_PORT,
                secure: false,
                auth: {
                    user: process.env.SMTP_USER,
                    pass: process.env.SMTP_PASSWORD,
                },
            },
            {
                from: 'Active Rule Service <${process.env.SMTP_USER}>',
            },
        );

        this.transporter.verify((error, success) => {
            error
                ? this.logger.error(error)
                : this.logger.log(
                        'MailService настроен верно. Verify success:
                                        ${success}',
                  );
        });
    }

    async sendResetPasswordMail(emailUser: string) {
        this.logger.log('Send reset password Mail to ${emailUser}');

        const uuid = v4(); // ⇨ '9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d'
        const activationLink = '${process.env.API_URL}/auth/reset-
password/${uuid}';
```

```
        await this.transporter.sendMail({
            to: emailUser,
            subject: 'Сброс пароля',
            text: '',
            html: mailTemplateReset(emailUser.split('@')[0], activationLink),
        });

        const user = await this.userRepository.findOne({
            where: { email: emailUser },
        });
        user.refreshLink = uuid;
        user.save();
        return 'Письмо со сбросом пароля было отправлено на почту';
    }

    async sendNewPasswordMail(emailUser: string, newPassword: string) {
        this.logger.log(`Send new password Mail to ${emailUser}`);

        await this.transporter.sendMail({
            to: emailUser,
            subject: 'Новый пароль',
            text: '',
            html: mailTemplateNewPassword(emailUser.split('@')[0],
              newPassword),
        });

        const user = await this.userRepository.findOne({
            where: { email: emailUser },
        });
        user.refreshLink = null;
        user.save();
        return 'Письмо с новым паролем было отправлено на почту';
    }
}
```

Листинг Б.7 – Файл jwt-auth.service.ts

```
@Injectable()
export class JwtAuthService {
    constructor(
        @InjectModel(User) private userRepository: typeof User,
        @InjectModel(Token) private tokenRepository: typeof Token,
        private readonly counterService: CounterService,
        private readonly jwtGenerateServise: JwtGenerateService,
    ) {}

    async registration(regUserDto: RegUserDto) {
        const candidate = await this.userRepository.findOne({
            where: { email: regUserDto.email },
        });
        if (candidate) {
            throw new BadRequestException(
                'Пользователь с указанным адресом электронной почты уже
зарегистрирован',
            );
        }

        const hashPassword = await bcrypt.hash(
            regUserDto.password,
            +process.env.BCRYPT_SOLT,
        );

        const codeUser = await
this.counterService.getNewCode(CounterName.USER);
```

```
        const user = await this.userRepository.create({
            ...regUserDto,
            password: hashPassword,
            code: codeUser,
        });

        const tokens = await this.jwtGenerateServise.generateTokens(
            this.createPayloadJWT(user),
            user.id,
        );

        return tokens;
    }

    async login(authUserDto: AuthUserDto) {
        const user = await this.userRepository.findOne({
            where: { email: authUserDto.email },
            include: { all: true },
        });
        if (!user)
            throw new BadRequestException('Непрвильный логин или пароль');

        // Сравнение введенного пароля и пароля из БД
        const passwordEquals = await bcrypt.compare(
            authUserDto.password,
            user.password,
        );
        if (!passwordEquals)
            throw new BadRequestException('Непрвильный логин или пароль');

        const tokens = await this.jwtGenerateServise.generateTokens(
            this.createPayloadJWT(user),
            user.id,
        );

        return tokens;
    }

    async logout(refreshToken: string) {
        await this.tokenRepository.destroy({ where: { refreshToken } });
    }

    async refresh(refreshToken: string) {
        const userTokenData =
            this.jwtGenerateServise.validateRefreshToken(refreshToken);
        const tokenFromBD = await this.tokenRepository.findOne({
            where: { refreshToken: refreshToken },
        });
        if (!userTokenData || !tokenFromBD) {
            throw new UnauthorizedException();
        }

        const user = await this.userRepository.findByPk(tokenFromBD.userId, {
            include: { all: true },
        });
        const tokens = await this.jwtGenerateServise.generateTokens(
            this.createPayloadJWT(user),
            user.id,
        );

        return tokens;
    }

    createPayloadJWT(user: User) {
        const payloadJWT: IPayloadJWT = {
```

```
            email: user.email,
            code: user.code,
            currentDate: Date.now(),
            roles: [],
        };
        if (!user.role) {
            return payloadJWT;
        }
        user?.role.forEach((role) => payloadJWT.roles.push(role.name));
        return payloadJWT;
    }

    async resetPassword(link: string) {
        const user = await this.userRepository.findOne({
            where: { refreshLink: link },
        });
        if (!user) {
            return 'Неверная ссылка для сброса пароля';
        }

        let randomPassord = '';
        const passwordTemplate =
            'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
        for (let i = 0; i < 5; i++)
            randomPassord += passwordTemplate.charAt(
                Math.floor(Math.random() * passwordTemplate.length),
            );
        const hashPassword = await bcrypt.hash(
            randomPassord,
            +process.env.BCRYPT_SOLT,
        );

        user.password = hashPassword;
        await user.save();
        return { email: user.email, password: randomPassord };
    }

    async changePassword(dto: ChangePasswordDto) {
        const user = await this.userRepository.findOne({
            where: { email: dto.email },
        });
        if (!user)
            throw new BadRequestException('Указанный пользователь не
найден');

        // Сравнение введенного пароля и пароля из БД
        const passwordEquals = await bcrypt.compare(
            dto.oldPassword,
            user.password,
        );
        if (!passwordEquals)
            throw new BadRequestException('Непрвильно указан прошлый
пароль');

        // Применение нового пароля
        const hashPassword = await bcrypt.hash(
            dto.newPassword,
            +process.env.BCRYPT_SOLT,
        );
        user.password = hashPassword;
        await user.save();
        return 'Пароль успешно изменен';
    }
}
```

Листинг Б.8 – Файл jwt-generte.service.ts

```
@Injectable()
export class JwtGenerateService {
    constructor(@InjectModel(Token) private tokenRepository: typeof Token) {}

    async generateTokens(payload: IPayloadJWT, userId: string) {
        const accessToken = jwt.sign(payload, process.env.JWT_ACCESS_SECRET,
{
            expiresIn: '30m',
        });
        const refreshToken = jwt.sign(payload,
process.env.JWT_REFRESH_SECRET, {
            expiresIn: '14d',
        });

        await this.saveRefreshToken(userId, refreshToken);

        return {
            accessToken,
            refreshToken,
        };
    }

    async saveRefreshToken(
        userId: string,
        refreshToken: string,
    ): Promise<Token> {
        const tokenData = await this.tokenRepository.findOne({
            where: { userId },
        });
        if (tokenData) {
            tokenData.refreshToken = refreshToken;
            return tokenData.save();
        }
        return this.tokenRepository.create({ userId, refreshToken });
    }

    validateAccessToken(token: string) {
        try {
            const userData = jwt.verify(token,
process.env.JWT_ACCESS_SECRET);
            return userData;
        } catch (err) {
            return null;
        }
    }

    validateRefreshToken(token: string) {
        try {
            const userData = jwt.verify(token,
process.env.JWT_REFRESH_SECRET);
            return userData;
        } catch (err) {
            return null;
        }
    }
}
```

Приложение В

КОД КОМПОНЕНТОВ ОБРАБОТКИ ЗАПРОСА ПО API

(рекомендуемое)

Листинг В.1 – Файл validation.pipe.ts

```
@Injectable()
export class RuValidationPipe
    extends ValidationPipe
    implements PipeTransform<any>
{
    async transform(value: any, metadata: ArgumentMetadata) {
        const obj = plainToClass(metadata.metatype, value);
        const errors = await validate(obj, {
            forbidNonWhitelisted: true,
            transform: true,
            whitelist: true,
        });

        if (errors.length) {
            /**
                property - название поля в котором ошибка
                constraints - сами ошибки в объектах (Пример: {isLength:
'Некорректная длина'})
            */
            const messages = errors.map((err) => {
                if (err.constraints.whitelistValidation)
                    err.constraints.whitelistValidation =
                        'Поле не должно существовать';
                return {
                    property: err.property,
                    message: `${Object.values(err.constraints).join('; ')}`,
                    error: [...Object.values(err.constraints)],
                };
            });
            throw new BadRequestException({
                status: HttpStatus.BAD_REQUEST,
                message: messages,
            });
        }

        return obj;
    }
}
```

Листинг В.2 – Файл logger.middleware.ts

```
@Injectable()
export class LoggerMiddleware implements NestMiddleware {
    use(req: Request, res: Response, next: NextFunction) {
        Logger.log(
            `Request ${req.baseUrl} with method ${req.method}`,
            LoggerMiddleware.name,
        );
        next();
    }
}
```

Листинг В.3 – Файл message.interceptor.ts

```
@Injectable()
export class OneMessageInterceptor implements NestInterceptor {
    intercept(
        context: ExecutionContext,
```

```
        stream$: CallHandler,
    ): Observable<any> {
        return stream$.handle().pipe(
            map((data) => {
                if (typeof data === 'string') {
                    return { message: data };
                } else {
                    return data;
                }
            }),
        );
    }
}
```

Листинг В.4 – Файл notfound.interceptor.ts

```
@Injectable()
export class NotFoundInterceptor implements NestInterceptor {
    constructor(private errorMessage: string = 'Не найдено') {}

    intercept(
        context: ExecutionContext,
        stream$: CallHandler,
    ): Observable<any> {
        return stream$.handle().pipe(
            tap((data) => {
                if (data === undefined || data === null) {
                    throw new NotFoundException(this.errorMessage);
                }
            }),
        );
    }
}
```

Листинг В.5 – Файл timeout.interceptor.ts

```
@Injectable()
export class TimeoutInterceptor implements NestInterceptor {
    intercept(context: ExecutionContext, next: CallHandler):
Observable<any> {
        return next.handle().pipe(
            timeout(5000),
            catchError((err) => {
                if (err instanceof TimeoutError) {
                    return throwError(() => new
RequestTimeoutException());
                }
                return throwError(() => err);
            }),
        );
    }
}
```

Листинг В.6 – Файл jwt-auth.guard.ts

```
@Injectable()
export class JwtAuthGuard implements CanActivate {
    canActivate(
```

```
        context: ExecutionContext,
    ): boolean | Promise<boolean> | Observable<boolean> {
        const req = context.switchToHttp().getRequest();
        Logger.log('JwtAuthGuard', JwtAuthGuard.name);
        try {
            if (!req.headers.authorization) throw new
              UnauthorizedException();

            const authHeader = req.headers.authorization;
            const bearer = authHeader.split(' ')[0];
            const token = authHeader.split(' ')[1];

            if (bearer != 'Bearer' || !token) {
                throw new UnauthorizedException();
            }

            const account = jwt.verify(
                token,
                process.env.JWT_ACCESS_SECRET,
            ) as IPayloadJWT;
            if (!account) throw new UnauthorizedException();

            req.account = account;

            return true;
        } catch (e) {
            throw new UnauthorizedException();
        }
    }
}
```

Листинг В.7 – Файл role-auth.guard.ts

```
/**
 * Перекрывает JWT авторизацию
 * Если у пользователя есть хотя бы одна необходимая роль -> доступ разрешён
 */
@Injectable()
export class RoleGuard implements CanActivate {
    constructor(private reflector: Reflector) {}

    canActivate(
        context: ExecutionContext,
    ): boolean | Promise<boolean> | Observable<boolean> {
        const req = context.switchToHttp().getRequest();
        try {
            const requiredRoles = this.reflector.getAllAndOverride<string[]>(
                ROLE_KEY,
                [context.getHandler(), context.getClass()],
            );

            // Если не указаны роли
            if (!requiredRoles) {
                return true;
            }

            if (!requiredRoles.find((el) => el === 'admin'))
                requiredRoles.push('admin');

            if (!req.headers.authorization) throw new
              UnauthorizedException();
            const authHeader = req.headers.authorization;
            const bearer = authHeader.split(' ')[0];
            const token = authHeader.split(' ')[1];
```

```
            if (bearer != 'Bearer' || !token) {
                throw new UnauthorizedException();
            }
            const account = jwt.verify(
                token,
                process.env.JWT_ACCESS_SECRET,
            ) as IPayloadJWT;
            if (!account) throw new UnauthorizedException();

            req.account = account;

            if (!account.roles.some((role) => requiredRoles.includes(role)))
                throw new HttpException(
                    'Доступ запрещен',
                    HttpStatus.FORBIDDEN,
                );
            return true;
        } catch (e) {
            if (e.response == 'Пользователь не авторизован') {
                throw new HttpException(
                    'Пользователь не авторизован',
                    HttpStatus.UNAUTHORIZED,
                );
            }
            throw new HttpException('Доступ запрещен', HttpStatus.FORBIDDEN);
        }
    }
}
```

Листинг В.8 – Файл validate.decorator.ts

```
export const IsStringRus = () => IsString({ message: 'Должно быть строкой'});

export const IsUnixStringRus = () =>
    applyDecorators(
        IsString({ message: 'Должно быть строковым значением unix time' }),
        Length(10, 10, { message: 'UnixTime должен содержать 10 чисел' }),
    );

export const IsIntRus = () => IsInt({ message: 'Должно быть числом' });

export const IsJSONRus = () => IsObject({ message: 'Должно быть JSON'ом' });

export const LengthRus = (min: number, max: number) =>
    Length(min, max, { message: 'Неверная длина. Мин: ${min}, макс: ${max}'
});

export const IsEnumRus = (obj: object) =>
    IsEnum(obj, { message: 'Недопустимое значение из перечисления' });

export const IsEmailRus = () =>
    IsEmail({}, { message: 'Неверный формат e-mail' });

export const Cookies = createParamDecorator(
    (data: string, ctx: ExecutionContext) => {
        const request = ctx.switchToHttp().getRequest();
        const obj = data ? request.cookies?.[data] : request.cookies;
        if (!obj)
            throw new BadRequestException('Cookie '${data}' не была
              передана');
        return obj;
    },
);
```

```
export const Role = (...role: string[]) =>
    applyDecorators(
        ApiForbiddenResponse({
            description: `Доступ: ${role.join(', ')}${
                role.find((el) => el === RoleName.ADMIN)
                    ? ''
                    : ', ${RoleName.ADMIN}`
            }`,
        }),
        ApiBearerAuth(),
        SetMetadata(ROLE_KEY, role),
    );
```

Приложение Г

КОД ТЕСТИРОВАНИЯ КЛАССОВ МОДУЛЯ ROLE

(рекомендуемое)

Листинг Г.1 – Исходный код тестирования RoleController

```
const roleStub = { id: 1, value: 'admin', description: 'Администратор' };

describe('RoleController', () => {
    let roleController: RoleController;
    let roleService: RoleService;

    beforeEach(async () => {
        const module: TestingModule = await Test.createTestingModule({
            controllers: [RoleController],
            providers: [
                {
                    provide: RoleService,
                    useValue: {
                        findAll: jest.fn(() => [roleStub]),
                        findOne: jest.fn().mockImplementation((id: number) =>
                            Promise.resolve({
                                ...roleStub,
                                id,
                            }),
                        ),
                        create: jest.fn(() => roleStub),
                        update: jest.fn(() => roleStub),
                        remove: jest.fn(),
                    },
                },
            ],
        }).compile();

        roleController = module.get(RoleController);
        roleService = module.get<RoleService>(RoleService);
    });

    it('should be defined', () => {
        expect(roleService).toBeDefined();
    });

    it('should get the roles', async () => {
        expect(await roleController.findAll()).toEqual([roleStub]);
    });

    it('should make a new role', async () => {
        expect(await roleController.create({ value: roleStub.value,
description: roleStub.description })).toEqual(roleStub);
    });

    it('should find a role', async () => {
        await roleController.findOne(1);
        expect(roleService.findOne).toHaveBeenCalledWith(1);
        expect(roleController.findOne(1)).resolves.toEqual({
            ...roleStub,
            id: 1,
        });
    });

    it('should remove the role', async () => {
        await roleController.remove(1);
        expect(roleService.remove).toBeCalledWith(1);
    });
});
```

Листинг Г.2 – Исходный код тестирования RoleService

```
const roleStub = { value: 'admin', description: 'Администратор' };

describe('RoleService', () => {
    let service: RoleService;
    let model: typeof Role;

    beforeEach(async () => {
        const moduleRef = await Test.createTestingModule({
            providers: [
                RoleService,
                {
                    provide: getModelToken(Role),
                    useValue: {
                        findAll: jest.fn(() => [roleStub]),
                        findOne: jest.fn(),
                        findByPk: jest.fn(),
                        create: jest.fn(() => roleStub),
                        destroy: jest.fn()
                    }
                }
            ],
        }).compile();

        service = moduleRef.get(RoleService);
        model = moduleRef.get<typeof Role>(getModelToken(Role));
    });

    it('should be defined', () => {
        expect(service).toBeDefined();
    });

    it('should get the roles', async () => {
        expect(await service.findAll()).toEqual([roleStub]);
    })

    it('should add a new role', async () => {
        expect(await service.create(roleStub)).toEqual(roleStub);
    })

    it('should get a role', async () => {
        const findSpy = jest.spyOn(model, 'findByPk');
        expect(service.findOne(1));
        expect(findSpy).toBeCalledWith(1);
    })

    it('should remove a role', async () => {
        const destroySpy = jest.spyOn(model, 'destroy');

expect(service.remove(1)).rejects.toBeInstanceOf(BadRequestException);

expect(service.remove(2)).rejects.toBeInstanceOf(BadRequestException);
        expect(service.remove(3))
        expect(destroySpy).toBeCalledWith({where: {id: 3}})
    })
});
```

Листинг 4 – Исходный код e2e тестирования

```
describe('Role (e2e)', () => {
    let app: INestApplication;
```

```
    const roleTest = [
        { id: 1, name: 'user' },
        { id: 2, name: 'user' },
    ]

    const roleService = {
        findAll: () => [...roleTest],
        findOne: (id: number) => roleTest.find(el => el.id === id)
    };

    beforeAll(async () => {
        const moduleRef = await Test.createTestingModule({
            imports: [RoleModule],
        })
            .overrideProvider(RoleService)
            .useValue(roleService)
            .overrideProvider(RoleGuard)
            .useClass(MockAuthGuard)
            .compile();

        app = moduleRef.createNestApplication();
        app.useGlobalInterceptors(new NotFoundInterceptor())
        await app.init();
    });

    it(`/GET role`, () => {
        return request(app.getHttpServer()).get('/role')
            .expect(200)
            .expect( roleService.findAll());
    });

    it(`/GET role with id`, () => {
        const testId = 1;
        return request(app.getHttpServer()).get(`/role/${testId}`)
            .expect(200)
            .expect( roleService.findOne(testId));
    });

    it(`/GET role with non-existent id`, () => {
        const testId = 5;
        return request(app.getHttpServer()).get(`/role/${testId}`)
            .expect(404);
    });

    afterAll(async () => {
        await app.close();
    });
});

export class MockAuthGuard implements CanActivate {
    canActivate(context: ExecutionContext) {
        return true;
    }
}
```