МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

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

Факультет	*Кафедра*
вычислительной техники	МОиПЭВМ

Направление подготовки 09.03.04 «Программная инженерия»

Направленность Программная инженерия

БАКАЛАВРСКАЯ РАБОТА

на тему

RESTful API сервер управления метаданными в составе информационно-управляющей платформы

Студент _____ Сурин Константин Олегович
(подпись, дата) (ФИО полностью)

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

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

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

Заведующий кафедрой _____ Козлов А.Ю.
(подпись) (фамилия, инициалы)

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

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

Пенза, 2025

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

Факультет	*Кафедра*
Вычислительной техники	МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
_____А.Ю. Козлов «20»
декабря 2024 г

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

1. Студент____Сурин Константин Олегович_____гр._21ВП1___факультета___ВТ_____
<Фамилия, имя, отчество полностью>

2.Тема работы___RESTful API сервер управления метаданными в составе___
_информационно-управляющей платформы_____
_____
_____

Тема утверждена приказом ПГУ №_____от "_____"_____20_____

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: RESTful API сервер управления метаданными в составе Информационно-управляющей платформы*
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) управление метаданными;
   б) анализ корректности метаданных;
   в) версионирование метаданных;
   г) регистрация, аутентификация и авторизация пользователей;
   д) взаимодействие с клиентскими приложениями;
   е) оповещение и доставка данных микросервисам;
   ж) взаимодействие со средствами журнализации;
   з) обработка ошибок;

5. Перечень подлежащих разработке вопросов:

1) *Анализ предметной области и постановка задачи на разработку;*
2) *Разработка требований и проектирование;*
3) *Реализация;*
4) *Контроль качества инструментальных средств;*
5) *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 на разработку	10%	08.04.2025 – 14.04.2025	
2	Проектирование программных средств	20%	15.04.2025 – 25.04.2025	
3	Реализация программных средств	30%	26.04.2025 – 12.05.2025	
4	Тестирование программных средств	20%	13.05.2025 – 21.05.2025	
5	Оформление пояснительной записки	20%	22.05.2025 – 03.06.2025	

Дата выдачи задания «20» декабря 2024г.

Руководитель бакалаврской работы _____
(подпись, дата)

Задание к исполнению принял студент _____
(подпись, дата)

Работу к защите допустить
Декан ФВТ _____ Л.Р. Фионова
(подпись, дата)

Реферат

Пояснительная записка содержит 128 листов, 23 рисунка, 21 таблицу, 20 использованных источников и 5 приложений.

RESTFUL API, УПРАВЛЕНИЕ МЕТАДАННЫМИ, ИНФОРМАЦИОННО-УПРАВЛЯЮЩАЯ ПЛАТФОРМА, TYPESCRIPT, NEST, ORM SEQUALIZE, DOCKER, NGINX, SQL, REDIS, VS CODE, DBEAVER, LINUX.

Объект исследования – RESTful API сервер управления метаданными в составе информационно-управляющей платформы.

Цель выпускной квалификационной работы – спроектировать и разработать REST-серверное приложение для управления метаданными в составе информационно-управляющей платформы.

Технологии разработки – язык 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 было спроектировано, разработано и протестировано REST-серверное приложение управления метаданными в составе информационно-управляющей платформы.

Оглавление

Введение 8

1 Анализ предметной области и планирование процесса разработки RESTful API сервера управления метаданными 10

1.1 Low-code платформа для развертывания, эксплуатации и сопровождения информационно-управляющих систем 10

1.1.1 Назначение и функциональные возможности платформы 10

1.1.2 Обобщенная программная архитектура платформы 13

1.1.3 Концептуальная модель информационно-управляющей платформы 16

1.2 Управление метаданными в информационно-управляющей платформе . 18

1.2.1 Объектная модель метаданных информационно-управляющей платформы 18

1.2.2 Жизненный цикл метаданных в информационно-управляющей платформе 19

1.2.3 Основные процессы управления метаданными в информационно-управляющей платформе 21

1.3 Сравнительный анализ систем управления метаданными в составе аналогичных платформ 22

1.3.1 Обоснование сравнительного анализа 22

1.3.2 Обзор приложений для анализа 22

1.3.3 Результаты сравнительного анализа и выводы 26

1.4 Планирование процесса разработки приложения 27

1.4.1 Цели разработки и назначение приложения 27

1.4.2 Классы пользователей приложения 29

1.4.3 План-график разработки и планирование бюджета 29

2 Анализ требований на разработку RESTful API сервера управления метаданными 34

2.1 Функциональные требования 34

2.2 Выявление требований и концептуальное моделирование репозитория метаданных для RESTful API сервера 36
2.3 Требования к архитектуре RESTful API сервера 38
2.4 Обоснование выбора технологий и средств разработки 41
2.5 Требования к аппаратно-программному обеспечению 43
2.5.1 Требования к аппаратно-техническому обеспечению 43
2.5.2 Требования к прикладному и системному программному обеспечению 44
3 Разработка серверных программных средств управления метаданными для RESTful API сервера 45
3.1 Разработка архитектуры развертывания 45
3.2 Разработка программных средств RESTful API сервера 47
3.3 Разработка программных средств взаимодействия с базой метаданных .. 50
3.3.1 Физическое моделирование 50
3.3.2 Разработка структуры базы метаданных 52
3.3.3 Разработка программных средств доступа к базе метаданных 54
3.4 Разработка базовых программных средств 56
3.4.1 Разработка программных средств управления метаданными 56
3.4.2 Разработка программных средств анализа корректности метаданных 59
3.4.3 Разработка программных средств управления пользователями 62
3.5 Разработка программных средств взаимодействия с клиентскими приложениями 65
4 Контроль качества серверных программных средств 72
4.1 Планирование процесса контроля качества 72
4.1.1 Мероприятия по контролю качества программных средств 72
4.1.2 Обоснование выбора типов автоматизированного тестирования 75
4.1.3 Обоснование выбора метрик программного кода 75
4.2 Автоматизированное тестирование 76

4.2.1 Модульное тестирование компонентов сервера 76
4.2.2 Интеграционное тестирование компонентов сервера 78
4.3 Оценка качества программного кода 80
4.3.1 Оценка на основе метрики Холстеда 80
4.3.2 Оценка на основе метрики устойчивости и абстрактности 83
Заключение 86
Список использованных источников 88
Приложение А (справочное) Глоссарий 90
Приложение Б (обязательное) Листинг сервисов моделей 92
Приложение В (обязательное) Листинг вспомогательных сервисов 111
Приложение Г (обязательное) Листинг средств авторизации 118
Приложение Д (рекомендуемое) Листинг модульных и интеграционных тестов 124

Введение

В современном мире информационные технологии играют ключевую роль в автоматизации бизнес-процессов и управления данными. Объемы собираемых и обрабатываемых данных стремительно растут, что стимулирует активное развитие инфраструктуры их обработки. Часто компании применяют монолитные системы, где компоненты тесно взаимосвязаны. Такой подход имеет ряд недостатков: сильная зависимость компонентов, сложность обновления и модернизации, трудности с поддержкой кода при большом объеме функционала, а также ограничение единым набором технологий и инструментов.

Развитие платформы low-code [1], позволяющей минимизировать ручное программирование и ускорять разработку приложений, открывает новые возможности для создания информационно-управляющих систем (ИУС). Одним и важных компонентов таких платформ является управление метаданными, которые обеспечивают гибкость, масштабируемость и адаптивность системы к изменяющимся требованиям. Для эффективного взаимодействия между компонентами платформы необходима разработка серверных программных средств, обеспечивающих надежный доступ к метаданным через стандартизированные интерфейсы.

RESTful API является стандартом для построения масштабируемых и поддерживаемых серверных приложений [2]. Его использование в составе платформы low-code позволяет обеспечить удобный и единообразный доступ к данным и функциональности, упрощая интеграцию с другими системами и разработку пользовательских интерфейсов. Разработка RESTful API сервера для управления метаданными в составе платформы low-code представляет собой актуальную задачу, направленную на повышение эффективности создания и эксплуатации ИУС.

Актуальность микросервисной архитектуры обусловлена способностью разделять сложные системы на независимые, легко масштабируемые компоненты [3]. Это важно для платформ low-code, где требуется высокая гибкость и быстрая адаптация к новым требованиям. Микросервисная архитектура

позволяет повысить отказоустойчивость, упростить обновление отдельных модулей и оптимизировать использование ресурсов, что делает ее предпочтительным подходом для реализации сервера управления метаданными.

Целью данной выпускной квалификационной работы является разработка RESTful API сервера управления метаданными в составе информационно-управляющей платформы low-code. В рамках работы будет выполнен анализ и разработка требований, проектирование, реализация, тестирование и оценка качества серверных программных средств.

1 Анализ предметной области и планирование процесса разработки RESTful API сервера управления метаданными

1.1 Low-code платформа для развертывания, эксплуатации и сопровождения информационно-управляющих систем

1.1.1 Назначение и функциональные возможности платформы

Запрос на Low-Code-инструменты возник не случайно – бизнес всегда стремился получать готовые IT-решения, которые полностью соответствуют его нуждам, но при этом требуют минимальных затрат. Однако долгое время совместить эти два критерия было невозможно. Готовые коробочные продукты, хотя и доступные по цене, часто оказывались слишком ограниченными, а кастомная разработка под конкретные задачи неизбежно вела к высоким расходам и длительным срокам реализации.

Сначала ситуацию попытались улучшить визуальные языки программирования и IDE [4], которые упростили создание приложений, но по-прежнему требовали квалифицированных разработчиков. Позже появились универсальные платформы, предлагающие гибкость в настройке под нужды бизнеса. Однако их адаптация либо ограничивалась встроенными возможностями, либо снова вынуждала привлекать технических специалистов.

Новый стандарт у разработчиков корпоративных решений – встроенные инструменты Low-Code, позволяющие самостоятельно развивать продукт без поддержки вендора или программистов в штате. Согласно исследованию low-code-платформ (LCAP) от агентства Gartner, к 2024 году более 65% разрабатываемых приложений будет построено с помощью инструментов Low-Code, а многие всемирно известные разработчики сложных информационных систем уже заняли сектор «Лидеры» Magic Quadrant LCAP [5].

Low-Code-разработка позволяет сократить объем ручного кодирования, используя готовые компоненты и шаблоны. Такой подход ускоряет создание приложений и снижает нагрузку на разработчиков, поскольку большую часть работы выполняет сама платформа. Суть Low-Code – в минимизации написания

кода, заменяя его визуальным моделированием, drag-and-drop интерфейсами и предустановленными функциями. Это делает процесс разработки доступнее даже для тех, у кого нет глубоких технических знаний.

Low-code занимает промежуточное положение между традиционным программированием и no-code решениями. Подход сочетает визуальные инструменты с возможностью добавления кода для расширения базовой функциональности и подходит как аналитикам с базовыми знаниями программирования, так и опытным разработчикам, которые хотят ускорить процесс создания приложений.

Low-code платформы – это инструменты, позволяющие создавать информационно-управляющие системы (ИУС) с минимальным использованием традиционного программирования. Они предлагают ряд средств, облегчающих разработку приложений без глубоких знаний кодирования:

- визуальные компоненты (блоки, элементы интерфейса, базы данных) для построения ИУС;
- настраиваемая конфигурация компонентов вместо написания кода и их взаимодействие через графический интерфейс, определяющих логику бизнес-процессов;
- встроенные модули для работы с данными, интеграции с внешними системами, автоматизации задач и управления пользователями.

Преимущества использования Low-code платформ для ИУС:

- снижение затрат на разработку, так как не требуется привлекать дорогостоящих программистов;
- существенное сокращение времени разработки приложения;
- возможность быстро адаптировать ИУС к меняющимся потребностям бизнеса;
- автоматизация многих бизнес-процессов;
- единая платформа, объединяющая различные функциональности в единое решение (CRM, BPM, ECM и т. д.);

– интеграция ИУС с другими системами, как существующими, так и новыми.

На рисунке 1 показана диаграмма use-case Low-Code-платформы [6], отражающая основные сценарии взаимодействия пользователей с системой.

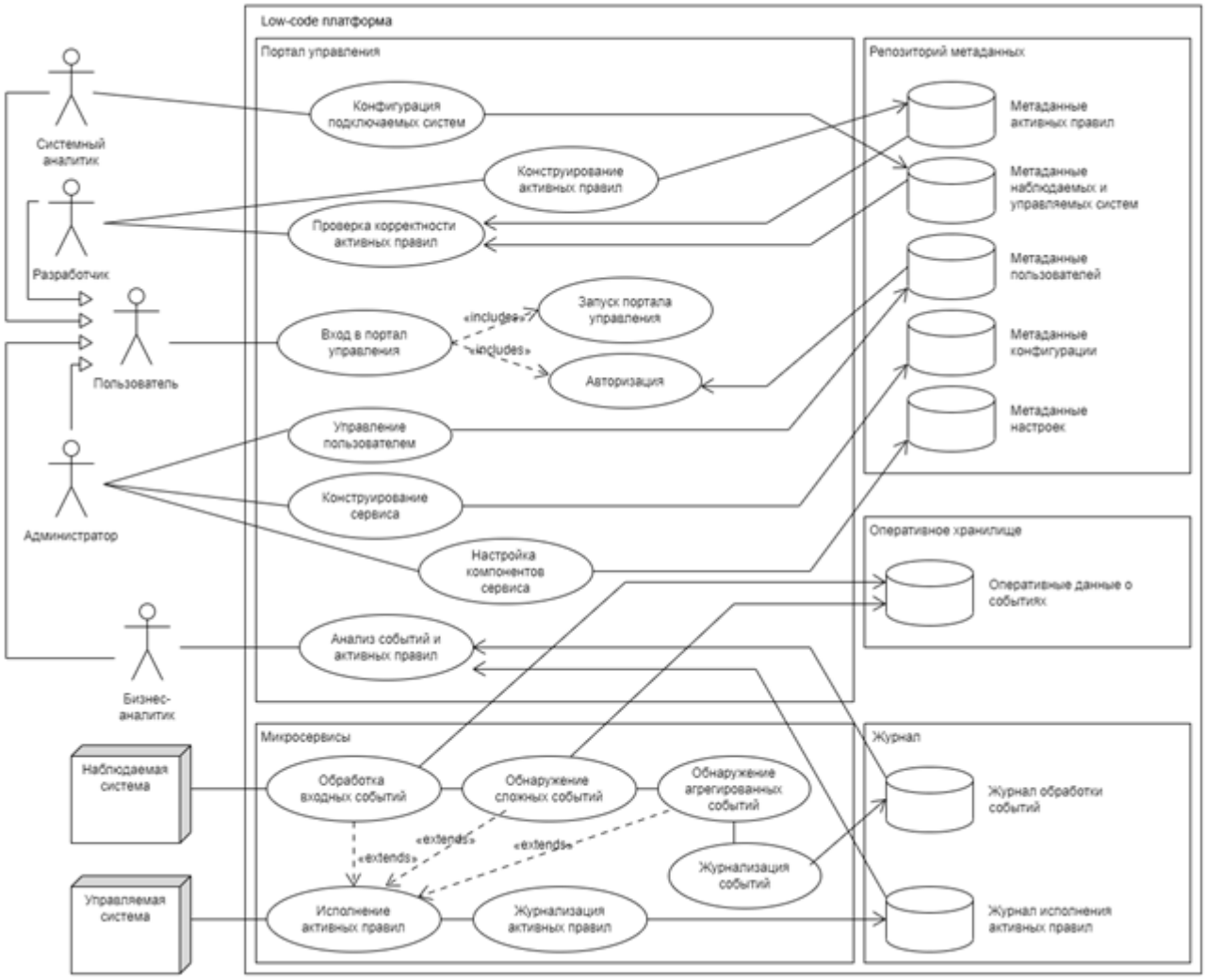

Рисунок 1 – Диаграмма вариантов использования RESTful API сервера управления метаданными

Данная схема иллюстрирует архитектуру системы обработки событий и управления метаданными. Она включает пользователей (системных аналитиков, разработчиков, пользователей и администраторов), взаимодействующих через REST-сервер с порталом управления. Система разделена на блоки: управление событиями (включая настройку правил, сбор и маршрутизацию), обработку событий (анализ, настройка сервисов) и хранение метаданных (активных правил, пользователей, настроек). Брокер сообщений обеспечивает маршрутизацию, а

базы данных хранят данные для различных операций и конфигураций. Архитектура поддерживает масштабируемость и отказоустойчивость.

1.1.2 Обобщенная программная архитектура платформы

Общая архитектура low-code платформы является многокомпонентной, что положительно сказывается на расширяемости и масштабируемости.

Клиентская часть платформы отвечает за взаимодействие с пользователем и включает в себя:

- веб-интерфейс для разработчиков и конечных пользователей;
- мобильные приложения (если поддерживается).

Серверная часть отвечает за обработку данных и бизнес-логику приложения. В ее составе обычно представлены следующие компоненты:

- API Gateway предоставляет интерфейсы для взаимодействия с клиентской частью, отвечает за обработку запросов от клиентов и маршрутизацию их к соответствующим сервисам;
- сервис управления данными, выполняющий обработку запросов к базам данных, кэширование, управление транзакциями;
- сервис бизнес-логики, реализующий бизнес-процессы и логику приложения;
- сервис безопасности, управляющий аутентификацией и авторизацией пользователей.

Хранилище данных:

- реляционные и нереляционные базы данных для хранения информации;
- системы хранения файлов и документов.

Low-code платформа может включать в себя различные интеграционные компоненты, модули для интеграции с внешними API, сервисами и другими системами, определяющими функционал конкретной ИУС, адаптированной под конкретные бизнес-требования.

Платформа построена на основе микросервисной архитектуры, которая

обеспечивает модульность, независимость и высокую масштабируемость системы. Для поддержания асинхронного взаимодействия между компонентами применяется событийно-ориентированный подход, обеспечивающим значительные преимущества в производительности и надежности. Основу системы составляет распределенная сеть независимых сервисов, каждый из которых отвечает за определенный функционал системы.

Для взаимодействия компонентов и поддержания слабой связанности используется брокер сообщений [7]. Он ключевую роль в архитектуре, обеспечивая доставку событий между компонентами. Брокер действует как интеллектуальный маршрутизатор, передавая сообщения в соответствующие очереди без анализа их содержимого.

Преимуществами такой архитектуры является:

- универсальность системы, позволяющая разрабатывать различные компоненты на разных языках программирования и работать на различных платформах, используя схожие принципы взаимодействия;
- гибкое масштабирование, дающее возможность увеличить мощность отдельных модулей без необходимости перестройки всей системы;
- отказоустойчивость, при которой выход из строя одного компонента не приводит к остановке всей системы.

Для организации взаимодействия клиентских приложений с платформой используется REST-сервер, который является ее центральным компонентом. Он обеспечивает взаимодействие клиентских приложений с микросервисами и управляет централизованным репозиторием метаданных, что значительно упрощает процесс конфигурации и мониторинга всей платформы.

Данное архитектурное решение сочетает в себе высокую производительность, надежность и гибкость, делая платформу эффективным решением для работы с динамичными нагрузками и сложными бизнес-процессами.

На рисунке 2 отображена архитектурная схема платформы, предназначенной для развертывания информационно-управляющей платформы.

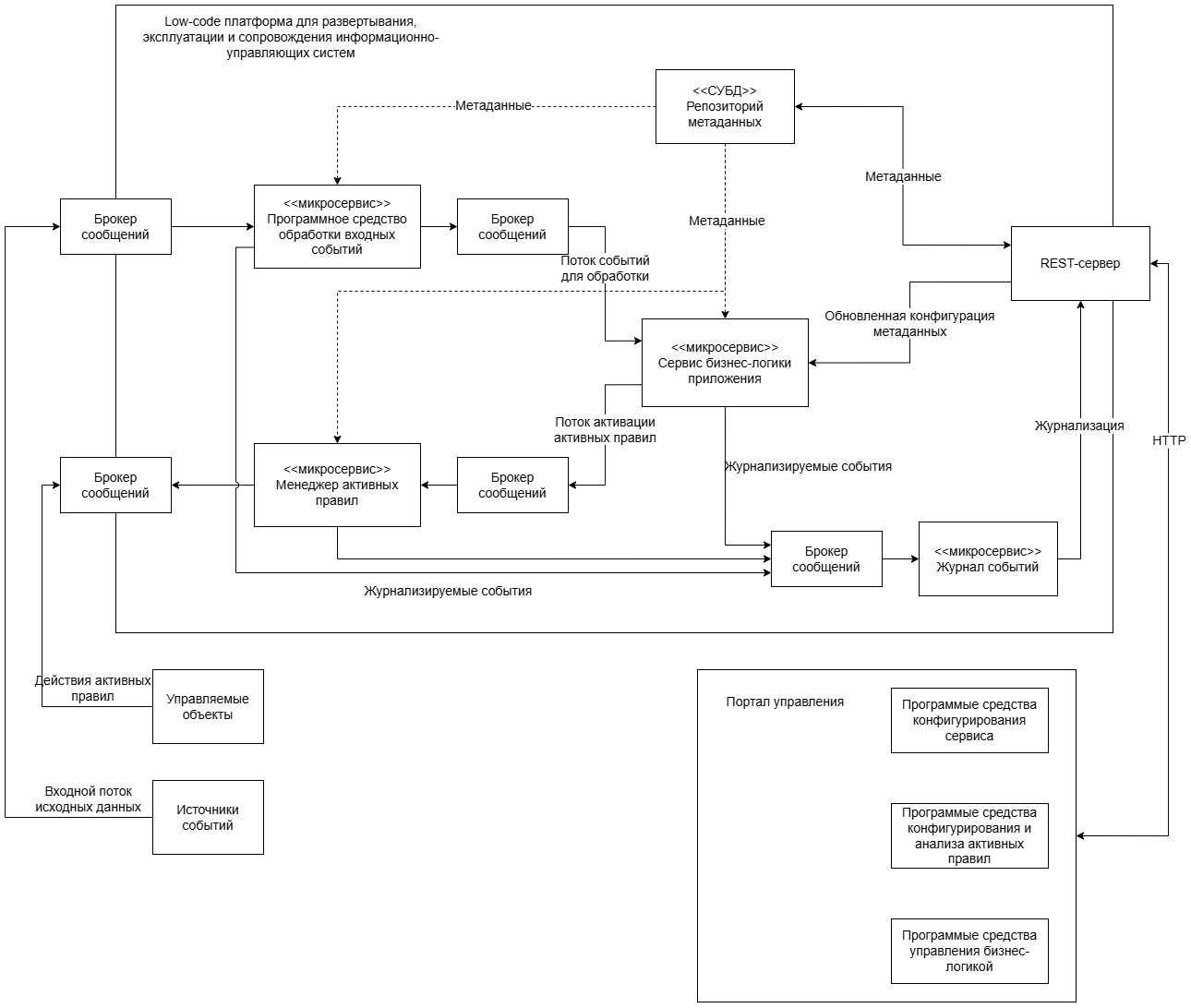

Рисунок 2 – Архитектура платформы

В данной архитектуре клиентской частью является портал управления, который обеспечивает все взаимодействие пользователей с возможностями платформы с помощью различных средств конфигурации и управления.

Серверная часть представлена программными средствами обработки входных данных, сервисом бизнес-логики и менеджером активных правил, отвечающих за весь процесс обработки информации, поступающей в систему. За управление отвечает REST-сервер, обеспечивающий доступ к метаданным и координирующий взаимодействие между компонентами.

Вся информация принимается и передается посредством брокеров сообщений.

Данные, информация о структуре событий, конфигурациях и правилах

обработки в системе хранятся прежде всего в центральном репозитории метаданных. Архитектура платформы реализует принцип изолированного хранения данных, означающий что каждый микросервис обладает собственной независимой базой данных в соответствии с паттерном «Database per Service» [8]. Для его реализации может быть организовано локальное хранилище данных для каждого микросервиса. При этом система должна поддерживать средства синхронизации метаданных, обеспечивающие поддержания целостности и актуальности данных во всей системе.

Также в данной системе используются средства журнализации для анализа и отладки.

1.1.3 Концептуальная модель информационно-управляющей платформы

Концептуальная модель информационно-управляющей платформы на основе low-code представляет собой абстрактное представление структуры, компонентов и взаимодействий в системе, которая позволяет пользователям создавать приложения с минимальным написанием кода.

Основными компонентами модели являются:

- платформа low-code, являющаяся центральным элементом, обеспечивающим инструменты для разработки, развертывания и сопровождения ИУС, которая включает визуальные редакторы, готовые шаблоны и интеграционные возможности;
- пользователи, взаимодействующие с системой (администраторы, операторы, разработчики, конечные пользователи);
- данные, необходимые для функционирования системы;
- автоматизированные и управляемые рабочие процессы;
- интеграции и подключения к внешним системам;
- пользовательский интерфейс, созданный с использованием low-code инструментов.

На рисунке 3 представлена диаграмма сущность-связь, описывающая

структуру данных для информационно-управляющей платформы на основе low-code для управления активными правилами.

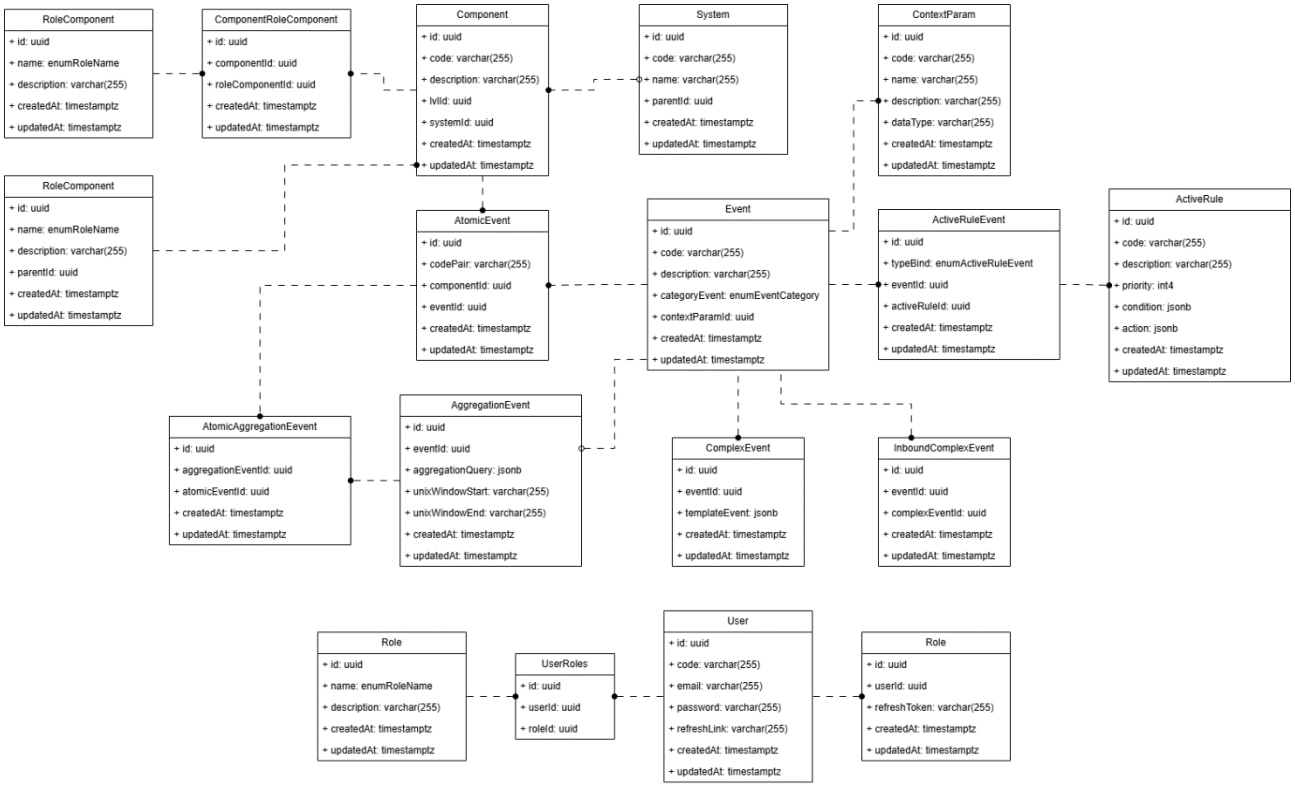

Рисунок 3 – Диаграмма сущность-связь

Основные сущности:

- RoleComponent описывает компоненты ролей в системе, такие как типы или категории ролей;
- ComponentRoleComponent – связующая сущность между компонентами и ролями, реализующая отношение многие-ко-многим;
- Component представляет компоненты системы, связанные с определенными системами;
- System описывает системы или подсистемы, которые могут иметь иерархическую структуру (parentId);
- ContextParam хранит параметры контекста, такие как коды, имена и типы данных с приоритетом;
- AtomicEvent описывает атомарные события, связанные с компонентами и событиями;

- Event представляет события с категориями и связанными параметрами контекста;
- ActiveRuleEvent связывает активные правила с событиями;
- ActiveRule описывает активные правила с условиями и действиями в формате JSON;
- AtomicAggregationEvent связывает атомарные события с агрегированными событиями;
- AggregationEvent описывает агрегированные события с запросами и временными окнами;
- ComplexEvent представляет сложные события с шаблонами;
- InboundComplexEvent связывает входящие сложные события с их компонентами;
- User хранит информацию о пользователях системы;
- Role описывает роли пользователей;
- UserRoles – связующая сущность для отношения многие-ко-многим между пользователями и ролями.

1.2 Управление метаданными в информационно-управляющей платформе

1.2.1 Объектная модель метаданных информационно-управляющей платформы

Метаданные играют важную роль в процессе функционирования платформы, так как они описывают структуру, свойства и контекст данных, а также определяют, как эти данные должны использоваться и обрабатываться в системе [9].

Объектная модель метаданных описывает структуру и взаимосвязи между различными элементами метаданных в системе. Основные компоненты могут включать:

- метаданные объектов: описания сущностей, таких как таблицы, поля, представления, процедуры и т. д., их атрибуты;

– связи, определяющие, как объекты взаимодействуют друг с другом, связи между таблицами;

– семантические метаданные: информация о значении и контексте данных, включающая словари терминов, бизнес-правила и контексты использования;

– процессы и правила обработки данных и правила валидации, которые применяются к данным;

– исторические метаданные: информация о том, как данные изменялись со временем (журнал изменений, версии).

Система обеспечивает безопасное хранение конфиденциальных данных пользователей за счет применения криптографического хеширования с использованием «соли». Данный подход преобразует пароли в необратимый формат, исключая возможность их восстановления из хранимых данных. Механизм аутентификации построен на сравнении хешированных значений, что позволяет верифицировать пользователей без необходимости хранения исходных паролей в открытом виде. Такая реализация соответствует актуальным требованиям информационной безопасности и промышленным стандартам защиты персональных данных.

1.2.2 Жизненный цикл метаданных в информационно-управляющей платформе

Для корректного управления метаданными в платформе необходимо понимать их жизненный цикл в работе системы.

Жизненный цикл метаданных включает несколько этапов:

1) создание метаданных;
2) хранение метаданных;
3) использование метаданных;
4) обновление метаданных;
5) архивирование метаданных;
6) удаление метаданных.

Схема жизненного цикла представлена на рисунке 4.

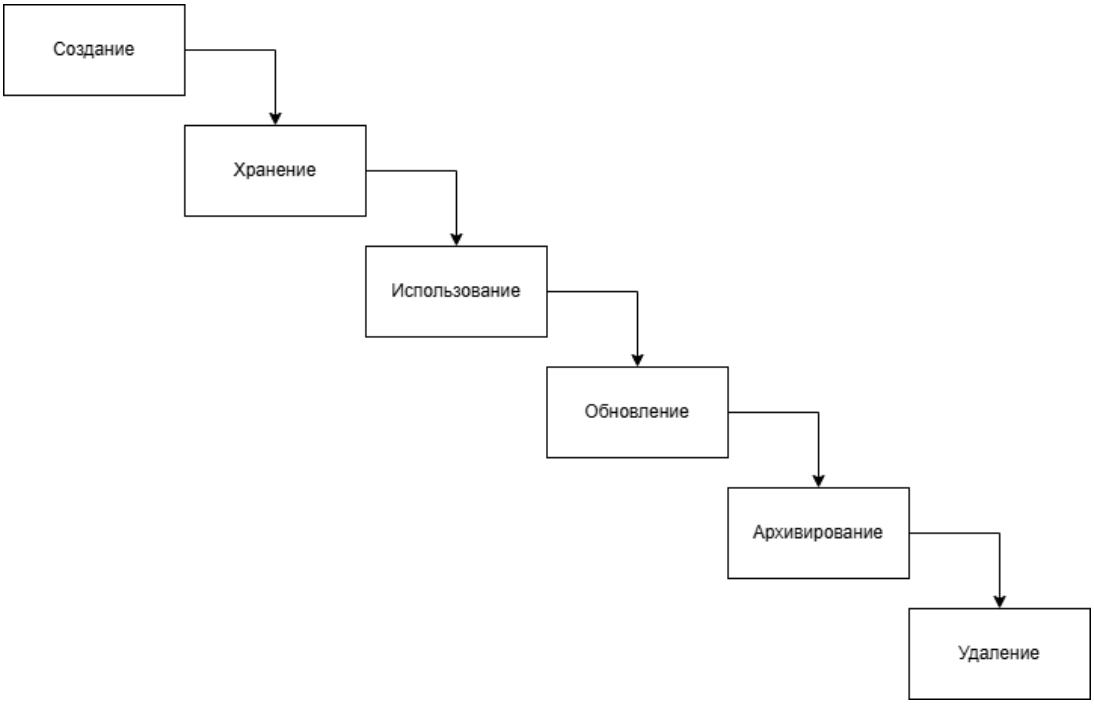

Рисунок 4 – Жизненный цикл метаданных в информационно-управляющей платформе

Этап создания метаданных заключается в формировании новых объектов метаданных в результате проектирования системы, определение их атрибутов и связей. На этом этапе происходит проверка корректности создаваемых объектов.

На этапе хранения созданные метаданные сохраняются в центральном репозитории, а также рассылаются в локальные репозитории микросервисов.

На следующем этапе происходит использование метаданных в работе различных компонент платформы для обработки данных.

Обновление метаданных происходит в процессе работы при изменении бизнес-требований или структуры системы, которое может быть связано с добавлением новых объектов или изменением существующих.

Для снижения нагрузки на систему устаревшие или неиспользуемые метаданные могут архивироваться. При этом сохраняется история их изменений, которая используется для мониторнга работы системы.

Ненужные или устаревшие наборы метаданных могут быть удалены из системы при необходимости оптимизации платформы.

Система реализует механизм логирования метаданных с помощью сервисов мониторинга. Данный функционал обеспечивает непрерывный контроль за работоспособностью системы, фиксируя все операции с метаданными в хронологическом порядке.

1.2.3 Основные процессы управления метаданными в информационно-управляющей платформе

Процессы управления метаданными связаны с обеспечением интеграции, доступа, совместного использования и анализа корректности работы платформы с данными. Основными процессами управления метаданными в информационно-управляющей платформе являются:

- каталогизация – организация и классификация метаданных для упрощения их поиска и использования;
- управление версиями – отслеживание изменений в метаданных для обеспечения целостности и возможности отката к предыдущим версиям при необходимости;
- контроль доступа – обеспечения безопасности и защиты конфиденциальной информации;
- поддержание качества данных – обеспечение точности, полноты и актуальность метаданных с помощью валидации и очистки данных;
- интеграция метаданных – объединение метаданных из различных источников;
- мониторинг и аудит – отслеживание изменений в метаданных.

Для передачи пользовательских метаданных и аутентификации в информационно-управляющей платформе используется JSON Web Token (JWT). JSON Web Token – это открытый стандарт (RFC 7519) для создания токенов доступа, основанный на формате JavaScript Object Notation (JSON) [10]. JWT создаются на сервере с применением криптографической подписи, что гарантирует их подлинность и защиту от подделки. Каждый токен содержит необходимую информацию о пользователе и его правах доступа, позволяя системе проводить авторизацию

без необходимости постоянного обращения к центральной базе данных.

1.3 Сравнительный анализ систем управления метаданными в составе аналогичных платформ

1.3.1 Обоснование сравнительного анализа

Выбор действительно эффективной low-code платформы для enterprise-сегмента требует глубокого анализа и понимания специфики бизнес-процессов компании. Ключевое отличие настоящего low-code решения от кастомизируемой коробочной системы заключается в его гибкости и возможности создавать универсальные приложения без ограничений предустановленных шаблонов.

При оценке платформ следует обратить внимание на гибкость архитектуры, возможности интеграции, масштабируемость, инструменты разработки и управление жизненным циклом приложений. Важно, чтобы платформа позволяла свободно проектировать бизнес-логику, легко интегрировалась с существующими системами, обеспечивала высокую производительность при больших нагрузках и предоставляла удобные визуальные инструменты для разработки.

1.3.2 Обзор приложений для анализа

1. ELMA365 – российская low-code платформа для автоматизации бизнес-процессов и управления эффективностью организации [11].

Особенности:

- интегрированный BPM-движок;
- система электронного документооборота;
- интуитивно понятный интерфейс.

Плюсы:

- единая среда для моделирования, исполнения и мониторинга процессов;
- библиотека готовых шаблонов бизнес-процессов;
- низкий порог входа для бизнес-пользователей;
- интеграция с популярными российскими системами;

– полная локализация на русский язык.

Минусы:

– ограниченные возможности кастомизации по сравнению с некоторыми конкурентами;

– может потребовать дополнительных настроек для сложных корпоративных сценариев;

– ограниченная функциональность бесплатной версии.

ELMA365 подходит для средних и малых компаний, ищущих готовое решение для управления бизнес-процессами с акцентом на документооборот и проектное управление.

2. SimpleOne – low-code платформа для разработки и управления сложными корпоративными приложениями.

Особенности:

– архитектура живой кастомизации;

– быстрое внедрение и доработка системы;

– многоуровневая система разработки (No-code, Low-code, Pro-code).

Плюсы:

– внесение изменений в систему без прерывания ее работы;

– набор инструментов для разработки, включая редактор рабочих процессов;

– детальная настройка прав доступа, вплоть до уровня отдельных полей;

– встроенные функции автоматизации на основе искусственного интеллекта;

– встроенные сервисные инструменты ESM-платформы.

Минусы:

– стоимость решения ориентирована на крупный и средний бизнес;

– ограниченное количество готовых отраслевых решений по сравнению с некоторыми конкурентами.

SimpleOne оптимально подходит для крупных и средних предприятий с

комплексной IT-инфраструктурой, требующих высокой производительности, гибкости и безопасности в процессе цифровой трансформации.

3. Comindware – Российская BPM-платформа для автоматизации бизнес-процессов и создания корпоративных приложений.

Особенности:

- визуальные инструменты для разработки бизнес-процессов;
- библиотека готовых процессов;
- гибкая настройка пользовательского интерфейса.

Плюсы:

- редакторы форм и бизнес-процессов;
- поддержка C# для сложной кастомизации;
- коннекторы и внешние интеграции по API;
- возможность работы в облаке и на собственном сервере;
- предоставление демоверсии продукта.

Минусы:

- высокая стоимость решения;
- может потребовать значительных ресурсов для внедрения;
- сложность освоения для неподготовленных пользователей.

Comindware подходит для средних и крупных предприятий, нуждающихся в гибкой платформе для автоматизации бизнес-процессов с возможностью глубокой кастомизации.

4. Citeck ECOS платформа для разработки корпоративных приложений с акцентом на управление бизнес-процессами и контентом.

Особенности:

- открытый исходный код (базовая версия);
- управление документами и бизнес-процессами;
- возможность создания индивидуальных интерфейсов.

Плюсы:

- управление версиями документов;
- гибкое построение отчетов;

- готовые решения для различных бизнес-процессов;
- интеграции с 1С, Oracle, КриптоПро и другими сервисами.

Минусы:

- отсутствие открытых цен для Enterprise версии;
- может потребовать дополнительных настроек для сложных сценариев;
- ограниченный функционал в бесплатной версии.

Citeck ECOS подходит для организаций, ищущих гибкое решение для управления документами и бизнес-процессами с возможностью глубокой кастомизации, особенно если важна возможность работы с открытым исходным кодом.

5. Digital Q экосистема low-code платформ для цифровой трансформации и импортозамещения, ориентированная на финансовый сектор.

Особенности:

- комплексный подход к цифровой трансформации;
- специализация на финансовом секторе.

Плюсы:

- ускорение разработки и сокращение затрат;
- интегрированные инструменты DevOps и CI/CD;
- поддержка микросервисной архитектуры;
- автоматическая генерация кода и тестов;
- широкий набор инструментов для работы с данными и интеграции.

Минусы:

- более узкая специализация на финансовом секторе;
- может требовать значительных ресурсов для полного внедрения;
- потенциально высокая стоимость для небольших организаций.

Digital Q оптимально подходит для крупных финансовых организаций, особенно банков, стремящихся к комплексной цифровой трансформации и импортозамещению IT-систем.

1.3.3 Результаты сравнительного анализа и выводы

Для детального сравнения аналогичных платформ была составлена таблица (таблица 1).

Таблица 1 – Сравнение low-code платформ

Критерий	**SimpleOne**	**ELMA**	**Comindware**	**Citeck ECOS**	**Digital Q**
Целевая аудитория	Крупные и средние предприятия	Средние и малые предприятия	Средние и крупные предприятия	Организации различного масштаба	Крупные финансовые организации
Специализация	Универсальная ITSM/ESM	BPM и документооборот	BPM и корпоративные приложения	ECM и BPM	Финансовый сектор
Архитектура	Микросервисная	Монолитная	Модульная	Модульная	Микросервисная
Возможности кастомизации	Высокие	Средние	Высокие	Высокие (open-source)	Средние
Скорость разработки	Высокая	Средняя	Средняя	Высокая	Высокая
Интеграционные возможности	Широкие (API, коннекторы)	Интеграция с российскими системами	Широкие (API)	Широкие (1С, Oracle, КриптоПро)	Средние
Работа с данными	Продвинутая аналитика	Базовая аналитика	Визуализация связей между процессами	Управление контентом и метаданными	Продвинутая аналитика и ETL
Поддержка AI/ML	Есть (ИИ автоматизация)	Ограниченная	Нет данных	Нет данных	Есть
DevOps инструменты	Продвинутые (CI/CD)	Базовые	Встроенные	Базовые	Продвинутые (CI/CD)
Мобильная разработка	Адаптивные веб-приложения	Веб-приложения	Адаптивные веб-приложения	Адаптивные веб-приложения	Омниканальные решения
Сложность освоения	Средняя	Низкая	Средняя	Средняя	Высокая
Особенности лицензирования	Ориентация на крупный и средний бизнес	Ограниченная бесплатная версия	О499 000 руб. за простое решение	Бесплатная open-source версия	Нет данных
Ключевые преимущества	Скорость разработки, ESM-платформа	Интуитивный интерфейс, готовые шаблоны	Визуальные инструменты, готовые бизнес-процессы	Открытый исходный код, управление документами	Финансовая специализация, ускорение разработки

В результате обзора можно выделить ряд особенностей, которыми также должна обладать платформа:

- интуитивно понятный интерфейс разработки, включающий визуальные шаблоны и готовые компоненты для ускорения разработки;
- автоматизированные процессы установки и настройки приложений;
- широкие интеграционные возможности для подключения к внешним системам;
- инструменты мониторинга и эксплуатации для отслеживания производительности, ошибок и нагрузки;
- генерация отчетов и аналитика в реальном времени для операторов;
- управление пользователями и доступом, настройка ролей и прав доступа для разных пользователей (администраторы, операторы, конечные пользователи);
- аутентификация и авторизация;
- диагностика и устранение ошибок через анализ логов;
- поддержка горизонтального и вертикального масштабирования для обработки растущих нагрузок;
- возможность совместной работы нескольких пользователей над одним приложением;
- управление версиями и контроль изменений;
- шифрование данных на уровне хранения.

Указанные преимущества будут интегрированы в архитектуру разрабатываемого программного обеспечения.

1.4 Планирование процесса разработки приложения

1.4.1 Цели разработки и назначение приложения

Целью разработки является создание RESTful API сервера управления метаданными в составе универсальной low-code платформы, которая упрощает процессы развертывания, эксплуатации и сопровождения информационно-управляющих систем (ИУС), минимизируя необходимость глубоких знаний

программирования, сокращая время разработки и повышая гибкость адаптации системы под изменяющиеся бизнес-требования.

RESTful API сервер предназначен для автоматизации жизненного цикла информационно-управляющей платформы, предоставляя пользователям (администраторам, операторам, разработчикам и конечным пользователям) инструменты для быстрого создания, настройки, развертывания и эксплуатации приложений, а также их сопровождения, включая мониторинг, обновление и интеграцию с внешними системами. Платформа обеспечивает интуитивно понятный интерфейс управления данными, генерации отчетов и диагностики, что позволяет эффективно управлять ИУС в условиях ограниченных ресурсов и высокой динамики бизнес-процессов.

Сервер должен обладать следующим функционалом:

- управление метаданными, включая их хранение, обработку и анализ, через интуитивно понятный интерфейс;
- простая и надежная интеграция платформы с внешними системами и сторонними приложениями;
- набор готовых API и шаблонных решений по работе с метаданными для ускорения процесса разработки;
- обеспечение безопасности и контроля доступа, реализация механизмов аутентификации и авторизации для защиты данных и управления доступом.

Целевая аудитория такой системы может быть:

- малый и средний бизнес (SMB), компании, нуждающиеся в быстром создании и настройке ИУС без привлечения крупных ИТ-отделов;
- ИТ-администраторы и операторы, ответственные за развертывание, эксплуатацию и мониторинг систем;
- разработчики с ограниченным опытом, желающие создавать приложения с минимальными навыками программирования;
- сотрудники организаций, использующие ИУС для повседневных задач (например, менеджеры, аналитики);

– интеграторы систем, компании или специалисты, работающие над подключением ИУС к существующим корпоративным системам (CRM, ERP).

1.4.2 Классы пользователей приложения

В платформе существуют четыре основные роли пользователей, каждая из которых выполняет свои функции в работе системы: разработчик, системный аналитик, бизнес-аналитик и системный администратор. Спецификация ролей пользователей представлена в таблице 2.

Таблица 2 – Спецификация ролей пользователей

Роль	Назначение
Разработчик	Проектирование и поддержка бизнес-логики, создание активных правил, обработка событий и управление метаданными
Системный аналитик	Настройка наблюдаемых систем, определение ключевых компонентов мониторинга и управления, конфигурация параметров взаимодействия
Бизнес-аналитик	Анализ статистических данных
Администратор	Управление пользовательскими учетными записями, распределением ролей и глобальная конфигурация системных компонентов

Четкое распределение обязанностей в информационно-управляющей платформе позволяет повысить производительность и качество данных, ускорить принятие решений, улучшить безопасность.

1.4.3 План-график разработки и планирование бюджета

Проведенный анализ предметной области позволил выделить основные задачи, необходимые для создания серверных компонентов платформы. В таблице 3 представлен детализированный план разработки, включающий последовательность выполнения работ, их продолжительность и взаимные зависимости.

Таблица 3 – Задачи проекта

№	Задача	Тип	Длительность (дни)	Предшественники
1	**2**	**3**	**4**	**5**
1	Старт проекта	Веха	0	-
2	Исследование и постановка целей проекта	Фаза	11	-
3	Анализ существующих решений	Задача	3	1

Продолжение таблицы 3

1	2	3	4	5
4	Изучение структуры данных конфигураций	Задача	2	3
5	Оценка процессов управления конфигурациями	Задача	2	4
6	Формирование списка автоматизируемых задач	Задача	2	5
7	Разработка плана проекта	Задача	2	6
8	Завершение анализа и постановки целей	Веха	0	7
9	Формирование требований к серверной части	Фаза	7	2
10	Разработка функциональных спецификаций	Задача	3	8
11	Определение требований к хранилищу данных	Задача	2	10
12	Выбор технологий и проектирование архитектуры	Задача	1	11
13	Уточнение требований к инфраструктуре	Задача	1	12
14	Завершение разработки требований	Веха	0	13
15	Создание серверной части системы	Фаза	28	9
16	Проектирование схемы развертывания	Задача	2	13
17	Разработка модуля интеграции с базой данных	Задача	3	16
18	Создание модуля авторизации	Задача	4	17
19	Разработка модуля управления метаданными	Задача	6	18
20	Создание модуля проверки корректности метаданных	Задача	3	19
21	Разработка модуля настройки системы	Задача	3	20
22	Создание интерфейса для клиентских приложений	Задача	3	21
23	Разработка модуля передачи данных сервисам	Задача	2	22
24	Интеграция с системой логирования	Задача	2	23
25	Завершение разработки серверной части	Веха	0	24
26	Проверка качества системы	Фаза	6	15
27	Подготовка плана тестирования	Задача	2	24
28	Проведение тестирования системы	Задача	4	27
29	Анализ качества кода	Задача	2	27
30	Завершение проверки качества	Веха	0	28;29
31	Финализация проекта	Фаза	5	26
32	Подготовка проектной 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 планирования был проведен детальный анализ

требуемых ресурсов для реализации проекта. Полный перечень необходимых ресурсов с указанием спецификаций и количественных показателей систематизирован в таблице 4.

Таблица 4 – Ресурсы проекта

Название ресурса	**Тип ресурса**	**Затраты**		**Начисление**
		Стандартная ставка	**Ставка сверхурочных**	
Разработчик	Трудовой	400,00 р/ч	450,00 р/ч	Пропорциональное
Бизнес-аналитик	Трудовой	250,00 р/ч	300,00 р/ч	Пропорциональное
Тестировщик	Трудовой	220,00 р/ч	270,00 р/ч	Пропорциональное
Сетевые ресурсы	Затраты	-	-	Пропорциональное
Программное обеспечение	Затраты	-	-	Пропорциональное

Детализированные данные о распределении ресурсов между отдельными задачами разработки систематизированы в таблице 5.

Таблица 5 – Распределение ресурсов по задачам

№	**Задача**	**Ресурсы**	**Единицы (затраты)**
1	**2**	**3**	**4**
1	Старт проекта	-	-
2	Исследование и постановка целей проекта	-	-
3	Анализ существующих решений	Бизнес-аналитик Сетевые ресурсы	100% 500р.
4	Изучение структуры данных конфигураций	Бизнес-аналитик	100%
5	Оценка процессов управления конфигурациями	Бизнес-аналитик	100%
6	Формирование списка автоматизируемых задач	Бизнес-аналитик	100%
7	Разработка плана проекта	Бизнес-аналитик Программное обеспечение	100% 1,200р
8	Завершение анализа и постановки целей	-	-
9	Формирование требований к серверной части	-	-
10	Разработка функциональных спецификаций	Разработчик	100%
11	Определение требований к хранилищу данных	Разработчик	100%
12	Выбор технологий и проектирование архитектуры	Разработчик	100%
13	Уточнение требований к инфраструктуре	Разработчик	100%
14	Завершение разработки требований	-	-

Продолжение таблицы 5

1	2	3	4
15	Создание серверной части системы	-	-
16	Проектирование схемы развертывания	Разработчик Программное обеспечение	100% 1000р
17	Разработка модуля интеграции с базой данных	Разработчик	100%
18	Создание модуля авторизации	Разработчик	100%
19	Разработка модуля управления метаданными	Разработчик	100%
20	Создание модуля проверки корректности метаданных	Разработчик	100%
21	Разработка модуля настройки системы	Разработчик	100%
22	Создание интерфейса для клиентских приложений	Разработчик	100%
23	Разработка модуля передачи данных сервисам	Разработчик	100%
24	Интеграция с системой логирования	Разработчик	100%
25	Завершение разработки серверной части	-	-
26	Проверка качества системы	-	-
27	Подготовка плана тестирования	Тестировщик	100%
28	Проведение тестирования системы	Тестировщик Программное обеспечение	100% 2,000р
29	Анализ качества кода	Разработчик Программное обеспечение	100% 700р
30	Завершение проверки качества	-	-
31	Финализация проекта	-	-
32	Подготовка проектной документации	Разработчик Программное обеспечение Сетевые ресурсы	100% 1,500р 500р
33	Сдача проекта	Разработчик	100%
34	Проект завершен	-	-
35	Конец проекта	-	-

Для планирования и анализа временных и финансовых затрат на разработку использовалось специализированное программное обеспечение Microsoft Project 2007. Этот инструмент позволяет эффективно управлять проектом, визуализировать сроки выполнения задач и контролировать ресурсы.

Диаграмма Ганта служит эффективным инструментом визуализации хода выполнения проекта. Данное представление включает:

- перечень всех работ проекта с их детализацией;
- временные рамки реализации каждой задачи;
- логические взаимосвязи между различными этапами работ;

– выделение критического пути, определяющего общую продолжительность проекта.

Каждая задача представлена в виде горизонтальной полосы, расположенной вдоль временной шкалы. Положение и длина полосы соответствуют дате начала, продолжительности и дате завершения работы. Диаграмма представлена на рисунке 5.

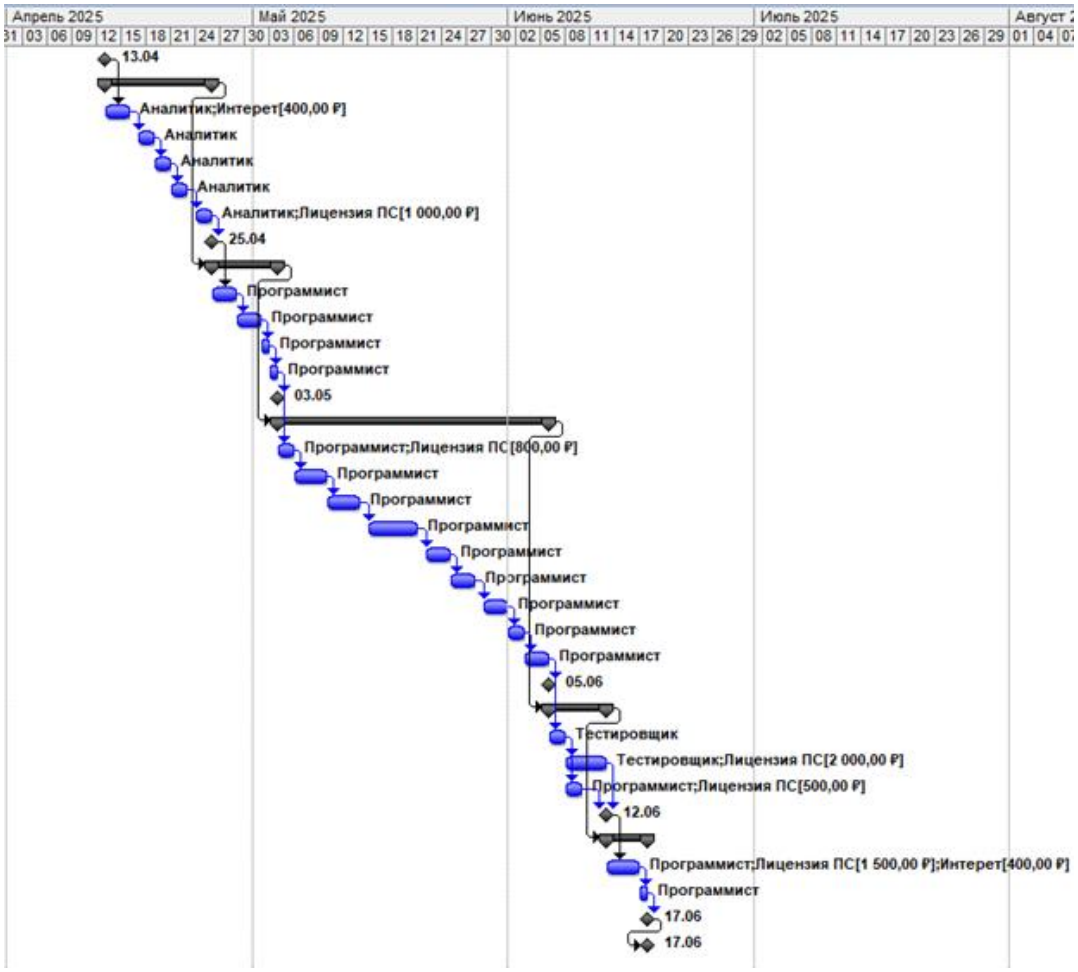

Рисунок 5 – Диаграмма Ганта

Диаграмма Ганта наглядно демонстрирует все ключевые аспекты проекта: необходимые ресурсы, основные этапы разработки, последовательность выполнения задач и сроки реализации каждого этапа. Установлена рабочая неделя длительностью шесть дней. Задачи выполняются последовательно из-за ограниченности трудовых ресурсов.

Планируемый срок выполнения всех работ с учетом выходных дней – 70 календарных дней. Денежные затраты составили 135500 рублей.

2 Анализ требований на разработку RESTful API сервера портала управления метаданными

2.1 Функциональные требования

Сервер управления метаданными должен обеспечивать комплексный функционал для эффективной работы с системой. Основу составляет поддержка стандартных CRUD-операций, позволяющих создавать, читать, обновлять и удалять метаданные. Для удобства работы реализованы механизмы поиска и фильтрации информации.

Важной особенностью сервера является его интеграционная способность – он обеспечивает синхронное и асинхронное взаимодействие с другими модулями платформы. Система безопасности включает многоуровневые механизмы аутентификации и авторизации.

Сервер поддерживает постоянную связь с микросервисами, оперативно оповещая их об изменениях в метаданных. Реализованы функции контроля актуальности данных и автоматического выявления возможных нарушений. Для обеспечения надежности работы система включает механизмы обработки ошибок и детализированного логирования всех операций.

RESTful сервер, отвечающий предъявленным требованиям, должен реализовывать взаимодействие с репозиторием метаданных посредством объектно-реляционного отображения (ORM). Данная технология выступает промежуточным слоем между реляционной базой данных и объектно-ориентированной моделью приложения, автоматизируя генерацию SQL-запросов и трансляцию результатов в объекты предметной области.

Для организации взаимодействия между компонентами системы применяется архитектура на основе брокера сообщений, выполняющего роль центрального посредника. Данный подход обеспечивает асинхронный обмен данными между микросервисами через унифицированный интерфейс с четко определенными форматами сообщений и служебными заголовками.

Ключевые особенности реализации:

– брокер сообщений выступает буфером между сервисами, принимая на себя функции маршрутизации и очереди сообщений;

– стандартизированные форматы обмена гарантируют совместимость между разнородными компонентами системы;

– HTTP-интерфейс для клиентских приложений с обязательной аутентификацией и авторизацией запросов.

На основе указанных принципов проектирования была разработана диаграмма вариантов использования (рисунок 6), отражающая ключевые сценарии взаимодействия участников системы с ее основными функциональными возможностями.

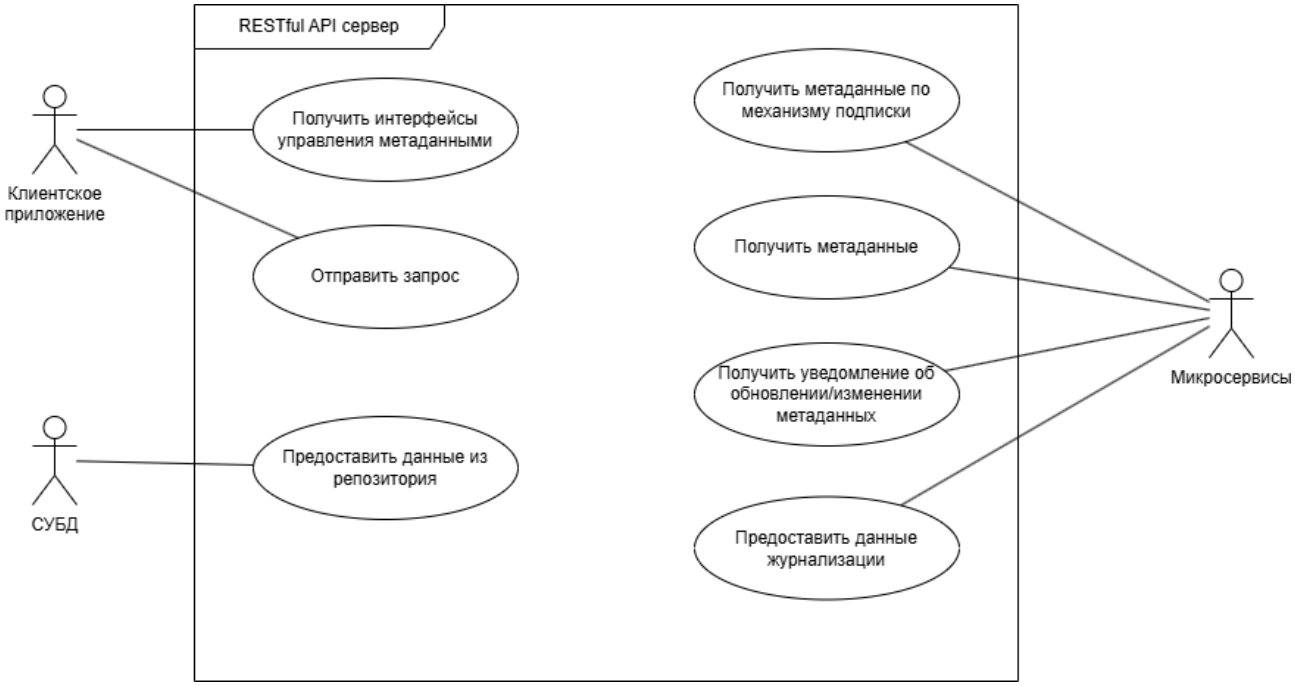

Рисунок 6 – Функциональные возможности REST-сервера

RESTful API сервер является ключевым компонентом платформы для информационно-управляющих систем (ИУС), обеспечивающим интеграцию и управление данными в сложных корпоративных средах. Его функциональность специально адаптирована для работы с автоматизированными системами управления и мониторинга, где критически важны надежное управление метаданными, высокая производительность обработки, безопасный обмен данными и бесшовная интеграция модулей.

2.2 Выявление требований и концептуальное моделирование репозитория метаданных для RESTful API сервера

Для корректной работы с метаданными, сервер должен иметь представление об используемых в работе сущностях.

Проектирование репозитория выполнялось с использованием CASE-средства Visual Paradigm 16.2 [12], которое предоставляет мощный функционал для многоуровневого моделирования данных. Этот инструмент поддерживает все ключевые этапы проектирования: от создания концептуальной модели, отражающей бизнес-сущности и их взаимосвязи до физической модели с учетом особенностей конкретной СУБД.

Концептуальное моделирование репозитория метаданных выполнялось на основе результатов анализа предметной области. Концептуальная модель – это высокоуровневое, абстрактное представление данных, описывающее ключевые сущности системы, их атрибуты и связи между ними с точки зрения бизнес-логики или предметной области.

Для моделирования применялся алгоритм концептуального моделирования в контексте нотации «сущность-связь», на основе нотации Мартина, представляющий собой структурированный процесс создания модели данных, которая описывает сущности, их атрибуты и связи между ними.

Процесс моделирования данных реализуется через последовательное выполнение ключевых этапов. Первоначально осуществляется формулировка целей и границ моделирования, что позволяет установить рамки будущей структуры данных. Затем выполняется выявление и классификация ключевых сущностей предметной области с последующим установлением взаимосвязей между ними. Завершающим этапом становится спецификация характеристик каждой сущности через определение набора атрибутов.

На рисунке 7 изображена ER-диаграмма (Entity-Relationship), отражающая концептуальную модель репозитория метаданных.

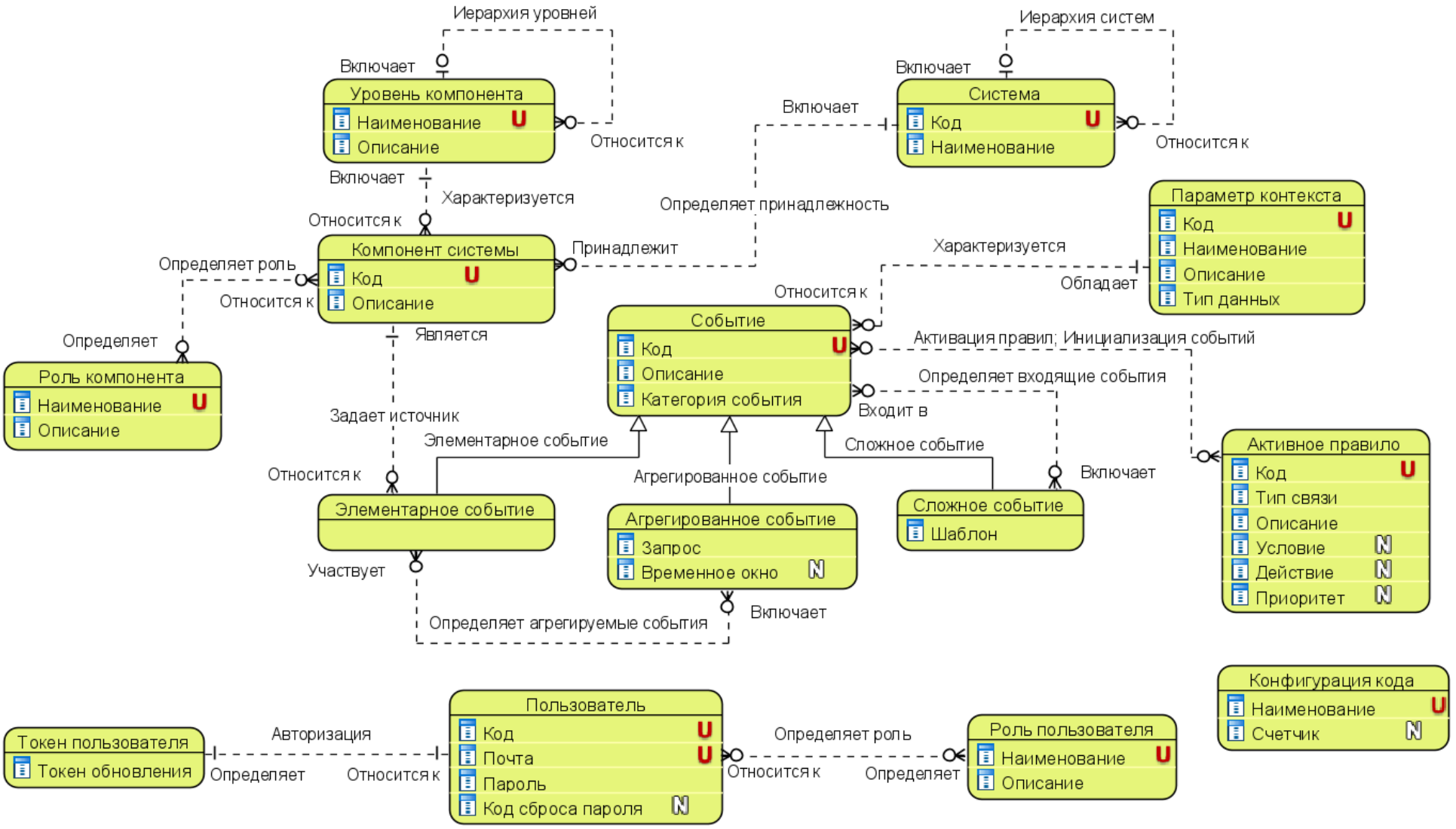

Рисунок 7 – Концептуальная модель репозитория метаданных

В результате проделанного анализа можно сделать следующие выводы. На основе построенной концептуальной модели будет создана физическая модель базы данных.

2.3 Требования к архитектуре RESTful API сервера

REST API (Representational State Transfer Application Programming Interface) – это архитектурный стиль взаимодействия между клиентом и сервером через HTTP. Он определяет принципы построения API, обеспечивая стандартизированный и эффективный обмен данными между различными системами [13].

Ключевые особенности RESTful-архитектуры включают:

– отсутствие состояния, означающее что каждый запрос содержит полный набор информации, необходимой для его обработки;

– поддержку кэширования для оптимизации работы с часто запрашиваемыми данными;

– многоуровневую структуру с четким разделением ответственности между компонентами;

– унифицированный интерфейс взаимодействия, обеспечивающий стандартизированный доступ к ресурсам.

RESTful API сервер для работы с репозиторием метаданных в ИУС реализует описанные принципы REST, поддерживая функциональные требования, такие как CRUD-операции, поиск, фильтрация, интеграция с микросервисами, управление доступом, обработка ошибок, производительность и документацию.

Основой архитектурного стиля REST является клиент-серверная модель. В данной модели клиент отвечает за инициацию запросов, которые обрабатываются сервером, после чего формируется ответ.

В рамках данной работы используется модульная архитектура приложения – это подход к проектированию программного обеспечения, при котором приложение разделяется на независимые, слабо связанные модули, каждый из которых

выполняет определённую функцию или решает конкретную задачу. Каждый модуль представляет собой обособленную единицу с чётко определённым интерфейсом, что упрощает разработку, тестирование, масштабирование и сопровождение системы. В контексте информационно-управляющей платформы и RESTful API сервера, модульная архитектура обеспечивает гибкость, интеграцию с микросервисами и поддержку функциональных требований, таких как работа с метаданными.

Для того чтобы сделать архитектуру более чистой и масштабируемой логика разделена между контроллерами и провайдерами. Это разделение способствует соблюдению принципов SOLID и облегчает дальнейшее развитие приложения.

Контроллеры в NestJS играют ключевую роль в обработке входящих REST-запросов и формировании ответов для клиента. Они действуют как посредники между клиентом и сервером, обеспечивая четкое разделение ответственности. Каждый контроллер привязан к конкретному модулю и обрабатывает только определенный набор запросов, соответствующий его функциональной области. Например, контроллер модуля ролей занимается исключительно операциями, связанными с ролями, что соответствует принципу единой ответственности (Single Responsibility Principle). NestJS предоставляет встроенные декораторы для работы с HTTP-методами, что упрощает маршрутизацию и обработку запросов.

Провайдеры, или сервисы, служат для инкапсуляции бизнес-логики и инфраструктурных задач приложения. Они позволяют вынести сложную логику из контроллеров, делая код более модульным и удобным для тестирования. Провайдеры могут включать в себя работу с базами данных, внешними API, валидацию данных и другие процессы, не связанные напрямую с HTTP-запросами. Благодаря этому контроллеры остаются легковесными и сосредоточены только на взаимодействии с клиентом [14].

Одним из ключевых преимуществ NestJS является поддержка внедрения зависимостей (Dependency Injection), что значительно упрощает управление провайдерами. Этот паттерн позволяет легко внедрять сервисы в контроллеры или другие провайдеры, что делает код более гибким и поддерживаемым.

Для работы REST-сервера необходимы вспомогательные компоненты, которые обеспечивают гибкость, безопасность и эффективность обработки запросов. Одним из таких элементов является Middleware (ПО промежуточного слоя) – функции, выполняемые до обработки запроса в контроллере. Они имеют доступ к объектам запроса (request) и ответа (response), что делает их идеальными для таких задач, как логирование, предварительная валидация данных или модификация запросов. Например, в сервисе потоковой обработки событий Middleware может использоваться для журнализации входящих запросов перед их дальнейшей обработкой.

Для контроля доступа к API критически важны Guards («Охранники»). Они действуют как фильтры, проверяющие права пользователя перед тем, как запрос достигнет контроллера. В системе, где реализована ролевая модель доступа, Guards анализируют токен аутентификации и разрешают или блокируют выполнение запроса на основе ролей пользователя.

Перехватчики (Interceptors) в NestJS представляют собой мощный инструмент, основанный на концепции аспектно-ориентированного программирования (AOP). Они позволяют внедрять дополнительную логику в процесс выполнения методов, не изменяя их исходный код. Перехватчики обеспечивают гибкость и модульность, позволяя разработчикам централизованно управлять сквозной функциональностью приложения. Их возможности включают:

– выполнение действий до или после вызова метода;

– изменение возвращаемых значения метода, например, форматирование данных в определённый вид (JSON, XML);

– обработка ошибок;

– кэширование;

– изменение логики выполнения метода, что полезно для реализации специфических сценариев.

Разработанная архитектура оптимизирует обработку клиентских запросов, обеспечивая их прохождение через все этапы жизненного цикла с максимальной производительностью. Для наглядного представления этого процесса была разработана специализированная диаграмма (рисунок 8).

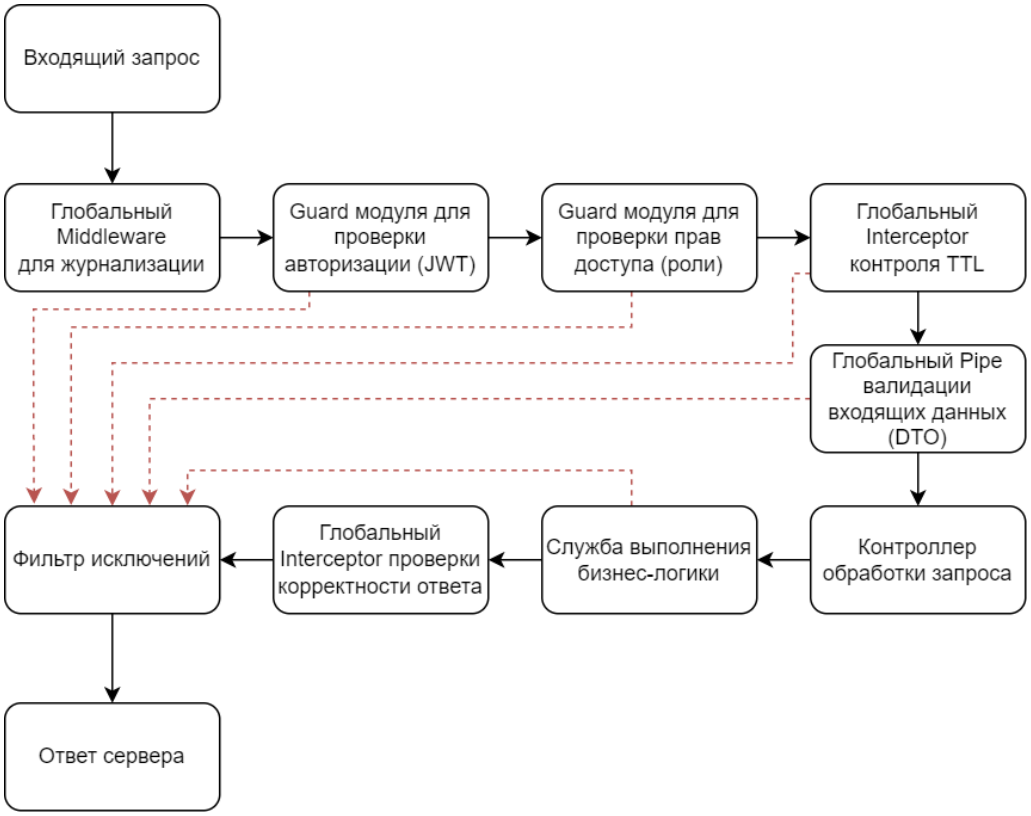

Рисунок 8 – Схема исполнения запроса от клиентского приложения

При возникновении ошибки на любом этапе обработки запроса стандартный поток выполнения прерывается, и управление передается глобальному обработчику исключений. Этот механизм гарантирует, что клиент всегда получит структурированный и понятный ответ, даже если в процессе работы сервера произошла непредвиденная ошибка.

2.4 Обоснование выбора технологий и средств разработки

Разработка серверной части системы управления метаданными будет осуществляться в соответствии с методологией Rational Unified Process (RUP) [15],

обеспечивающей итеративный и контролируемый подход к созданию программного обеспечения. В качестве технологической основы выбран фреймворк NestJS, предоставляющий готовую модульную архитектуру, что значительно упрощает поддержку и дальнейшее расширение системы.

Для обеспечения качества кода предусмотрено комплексное тестирование. Модульные тесты будут реализованы с использованием фреймворка Jest, который интегрирован с NestJS и предлагает мощные возможности тестирования, включая:

- подмену реальных зависимостей через Stub и Mock-объекты;
- измерение покрытия кода тестами в различных метриках;
- изоляцию тестов от рабочей среды сервера.

Функциональное тестирование API будет проводиться в Postman – удобном инструменте для проверки HTTP-запросов, поддерживающем все необходимые функции:

- работу с различными HTTP-методами;
- настройку авторизации и cookies;
- создание коллекций тестов.

Система будет развертываться в Docker-окружении [16], включающем:

- отдельный контейнер для REST-сервера;
- контейнер с PostgreSQL, управляемый через Docker-compose;
- возможность легкого масштабирования компонентов.

Для развертывания серверной инфраструктуры параллельно с REST-сервером используется база данных PostgreSQL, которая запускается через механизм Docker Compose. Этот инструмент позволяет одновременно управлять несколькими контейнерами, упрощая настройку и координацию сервисов.

Для документирования API используется Swagger [17], предоставляющий возможность автоматической генерации документации.

В качестве веб-сервера был выбран Nginx [18] благодаря его оптимальным характеристикам для работы с высокими нагрузками. Его асинхронная

архитектура обеспечивает стабильную производительность при одновременной обработке тысяч соединений, сохраняя при этом минимальное потребление ресурсов. В отличие от традиционных решений, Nginx демонстрирует более эффективную работу со статическим контентом и гибкие возможности балансировки нагрузки.

Описанный технологический стек создаёт прочную основу для серверной части приложения. Он обеспечивает высокую производительность, надежность и гибкость, позволяя разработчикам сосредоточиться на бизнес-логике, а не на инфраструктурных задачах. Такая архитектура подходит как для стартапов, так и для крупных систем, требующих масштабирования и долгосрочной поддержки.

2.5 Требования к аппаратно-программному обеспечению

2.5.1 Требования к аппаратно-техническому обеспечению

RESTful API сервер для управления метаданными предъявляет определённые требования к аппаратной части системы. Эти требования включают минимальные и рекомендуемые характеристики, которые обеспечивают стабильную работу приложения. Подробные параметры приведены в таблице 6.

Таблица 6 – Требования к аппаратно-техническому обеспечению

Требование	**Минимальные**	**Рекомендуемые**
Процессор	Частота 1,6 ГГц	2,8 ГГЦ
Количество потоков процессора	4	8
Оперативная память	1 ГБ	4 ГБ
Свободное дисковое пространство	15 ГБ	15 ГБ
Тип накопителя	HDD	SSD /NVMe

Обязательным условием является использование источников бесперебойного питания, способных поддерживать работу оборудования не менее 30 минут при отключении основного энергоснабжения. Такое решение гарантирует не только непрерывность функционирования сервиса при кратковременных перебоях, но и позволяет выполнить все необходимые процедуры для безопасного завершения работы системы в случае длительного отсутствия питания.

2.5.2 Требования к прикладному и системному программному обеспечению

Система требует обязательного наличия двух ключевых технологических компонентов: платформы контейнеризации Docker и веб-сервера Nginx. Docker обеспечивает изолированное выполнение приложений в рамках специальных контейнеров – легковесных виртуальных сред, которые включают все необходимые зависимости. Эта технология обеспечивает кроссплатформенность, быстрое развертывание без сложной настройки, изоляцию окружения для каждого модуля. Высокопроизводительный веб-сервер Nginx был выбран благодаря своей надежности, низкому потреблению ресурсов и широким возможностям конфигурации.

Для установки на операционной системе Windows необходимо соблюдение следующих условий:

– использование 64-разрядной версии Windows 10 (редакции Pro, Enterprise или Education) с номером сборки 15063 или выше;

– активация функций Hyper-V и Windows Containers.

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

– 64-разрядный процессор с поддержкой технологии Second Level Address Translation (SLAT);

– не менее 4 ГБ оперативной памяти;

– включение поддержки аппаратной виртуализации в настройках BIOS/UEFI.

Для систем, работающих под управлением Linux, требуются:

– 64-разрядная архитектура процессора;

– ядро операционной системы версии 3.10 или новее.

Для обеспечения высокой производительности и стабильной работы системы не рекомендуется устанавливать на сервер постороннее программное обеспечение, не связанное с основными задачами.

3 Разработка серверных программных средств управления метаданными для RESTful API сервера

3.1 Разработка архитектуры развертывания

Система реализована по принципам микросервисной архитектуры, где каждый компонент выполняет строго определенную функцию: PostgreSQL отвечает за хранение данных, Redis обеспечивает обмен сообщениями, Node.js обрабатывает бизнес-логику, Docker управляет изоляцией и развертыванием сервисов, а Nginx распределяет нагрузку и обрабатывает клиентские запросы. Такой подход гарантирует масштабируемость, отказоустойчивость и удобство поддержки системы.

PostgreSQL выступает в качестве основной системы управления базами данных, обеспечивая надежное и масштабируемое хранение информации.

Для организации взаимодействия между REST-сервером и микросервисами используется Redis, работающий в режиме резидентной in-memory базы данных с поддержкой шаблона публикация/подписка. Этот подход позволяет эффективно обмениваться сообщениями между компонентами системы без создания жестких зависимостей, выполняя функции брокера сообщений.

Основное приложение представляет собой REST-сервер. Разработка приложения выполнена на основе платформы Node.js.

При компиляции приложения происходит оптимизация кода с использованием скрипта. Это позволяет улучшить производительность и снизить нагрузку на систему.

Все ключевые компоненты, включая REST-сервер, Redis и PostgreSQL, развернуты в изолированных Docker-контейнерах, что обеспечивает гибкость развертывания и управляемость. Docker настроен на автоматический перезапуск контейнеров в случае аварийных ситуаций, таких как неожидан-ная перезагрузка сервера или сбои в сети, с соблюдением правильного порядка инициализации зависимых сервисов.

Для обработки входящих HTTP-запросов используется Nginx, развернутый напрямую в операционной системе Ubuntu без контейнеризации. Это решение обеспечивает высокую производительность, гибкость конфигурации и повышенную отказоустойчивость, поскольку Nginx работает вне Docker-инфраструктуры, минимизируя возможные точки отказа.

На рисунке 9 представлена архитектура развёртывания платформы.

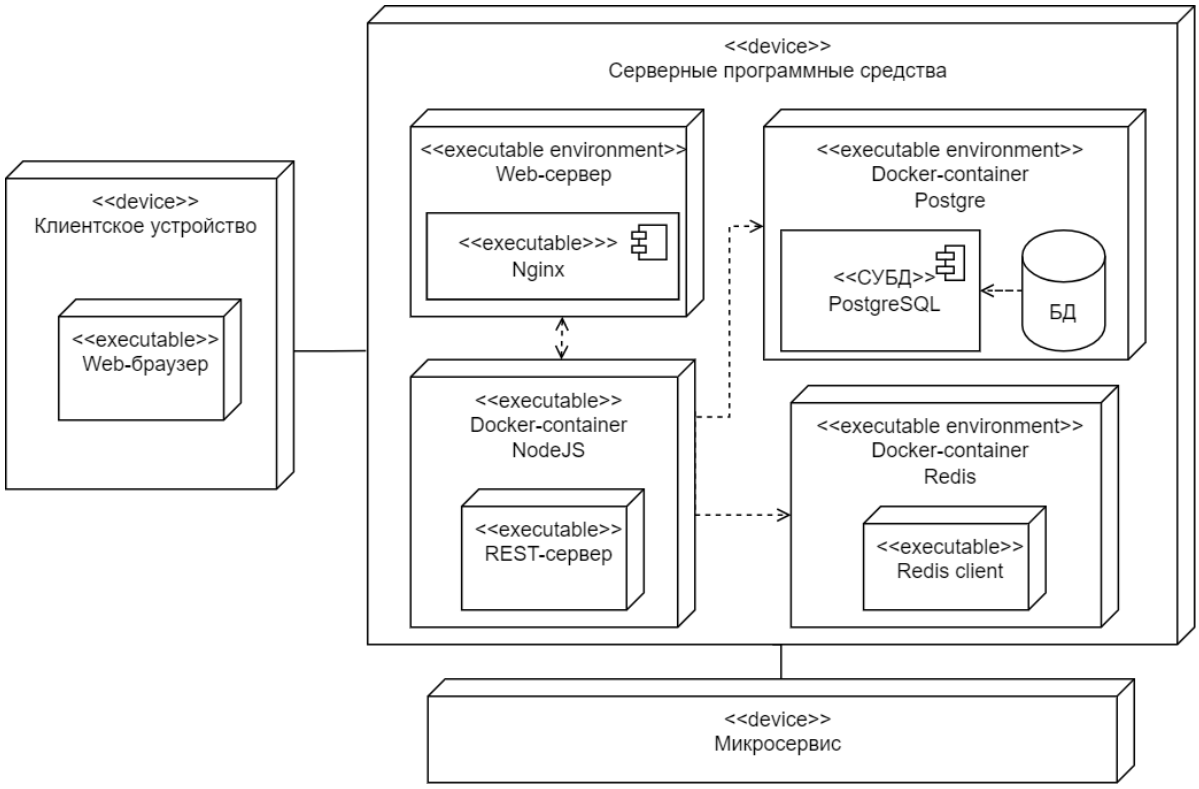

Рисунок 9 – Диаграмма развертывания информационно-управляющей платформы

Системные компоненты взаимодействуют через выделенную внутреннюю сеть, обеспечивающую защищенный обмен данными между сервисами. При этом для PostgreSQL была организована дополнительная сетевая изоляция – СУБД доступна через отдельно настроенный сетевой интерфейс, что позволяет контролировать внешние подключения к базе данных.

Сетевая сегментация реализована с учетом принципа минимальных привилегий, когда каждый компонент получает ровно те сетевые доступы, которые необходимы для его функционирования.

3.2 Разработка программных средств RESTful API сервера

REST-сервер управления метаданными в составе информационно-управляющей платформы спроектирован на основе модульной архитектуры, где каждый модуль отвечает за работу с определенной категорией метаданных. Такой подход обеспечивает четкое разделение ответственности: каждый модуль инкапсулирует собственную бизнес-логику, включая валидацию данных, обработку запросов и взаимодействие с репозиторием. Например, модуль управления ролями обрабатывает только операции, связанные с ролями пользователей, таких как создание, обновление, удаление и проверка прав доступа, а модуль работы с событиями – исключительно логику, относящуюся к событиям, включая их регистрацию, классификацию и обработку в реальном времени.

Модульная структура позволяет поддерживать порядок в кодовой базе и упрощает тестирование отдельных компонентов системы. Каждый модуль изолированно выполняет свои функции, практически не пересекаясь с другими частями системы, что повышает читаемость кода.

Каждый модуль служит фундаментом для других модулей системы: они наследуют базовую функциональность, такую как общие методы валидации, обработка ошибок или взаимодействие с общей инфраструктурой и дополняют ее специфической логикой по мере необходимости. Подобная архитектура не только ускоряет разработку новых модулей, снижая необходимость дублирования кода, но и облегчает поддержку приложения, поскольку изменения в базовой логике автоматически применяются ко всем зависимым модулям, обеспечивая согласованность работы системы в целом.

Модульность также упрощает масштабирование: при необходимости можно добавить новый модуль для работы с дополнительной категорией метаданных, не затрагивая существующие компоненты. Это делает систему гибкой и готовой к расширению в будущем.

Диаграмма зависимостей модулей представлена на рисунке 10.

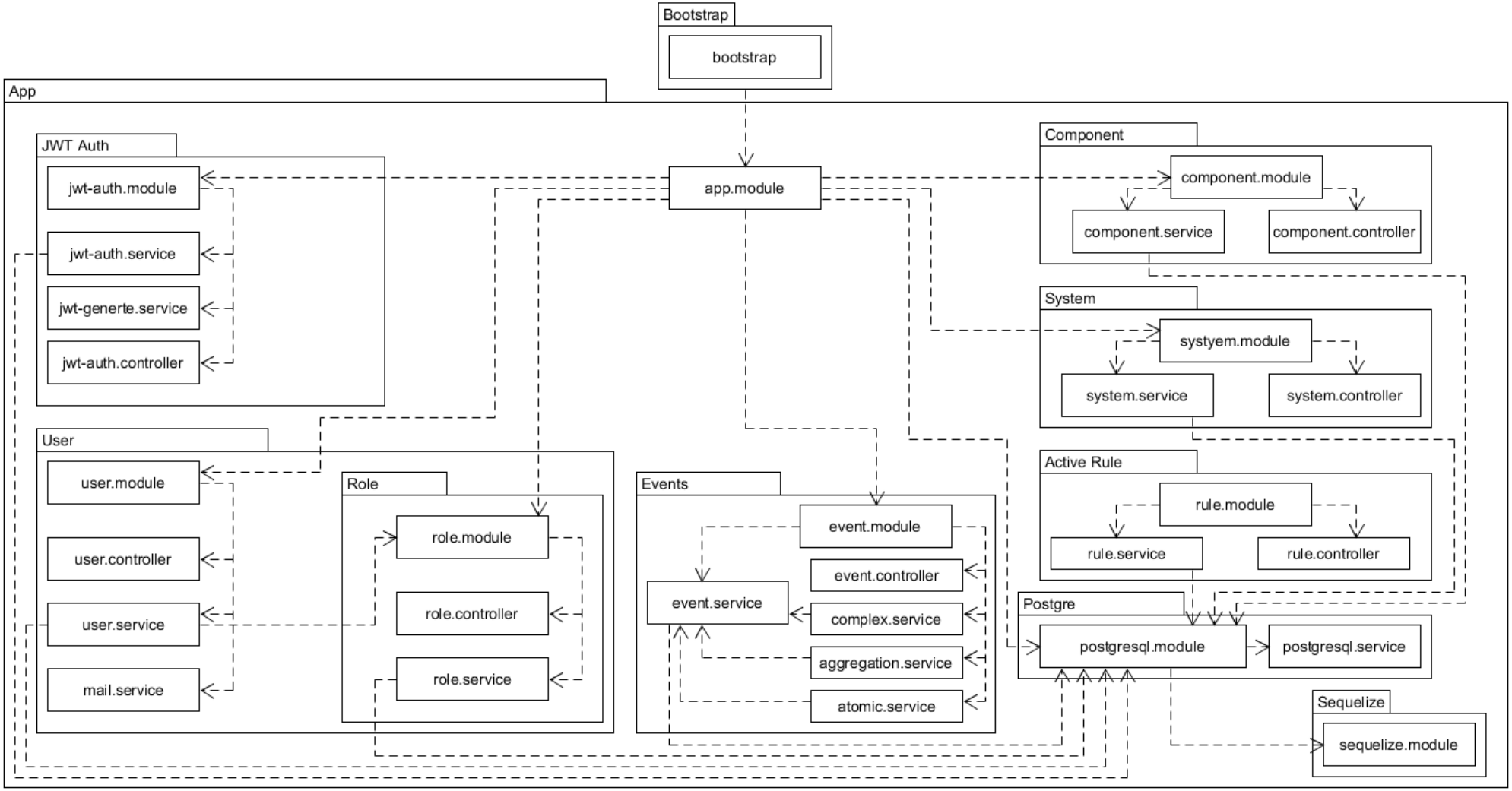

Рисунок 10 – Диаграмма пакетов модулей приложения

В таблице 7 представлено детальное описание модулей, используемых в платформе.

Таблица 7 – Описание компонентов REST-сервера

Модуль	**Назначение**
AppModule	Основной модуль приложения, отвечающий за его сборку и запуск REST-сервера
JwtAuthModule	Определяет структуру приложения, управляет зависимостями компонентов JWT
JwtGenerateSevice	Генерирует, обновляет и удаляет JWT-токены
JwtAuthService	Проверяет JWT-токены, выполняет аутентификацию и авторизацию пользователей
UserModule	Определяет структуру приложения, управляет зависимостями компонента User
UserService	Выполняет CRUD операции для пользовательских аккаунтов
MailService	Отвечает за проверку email адресов и подтверждение регистрации
RoleModule	Определяет структуру приложения, управляет зависимостями компонента Role
RoleService	Управляет ролями пользователей, их назначением
EventsModule	Определяет структуру приложения, управляет зависимостями компонента Event
EventsService	Работает с метаданными событий
PostgresqlModule	Обеспечивает объектно-ориентированное представление данных в БД
PostgresqlService	Инициирует начальные данные в БД
ComponentModule	Определяет структуру приложения, управляет зависимостями компонента Component
ComponentService	Выполняет CRUD операции для компонентов сервиса
SystemModule	Определяет структуру приложения, управляет зависимостями компонента System
SystemService	Выполняет CRUD операции для систем сервиса
RuleModule	Управляет метаданными активных правил и их связями с сервисами
RuleService	Выполняет CRUD операции, связывает события с правилами, определяя тип связи

Отдельный модуль отвечает за взаимодействие основных компонентов системы с различными микросервисами. К его функциям относится уведомление сервисов об изменениях в используемых ими наборах метаданных и обеспечение быстрой передачи сообщений между компонентами системы. Это необходимо для поддержки согласованного состояния данных всей системы.

3.3 Разработка программных средств взаимодействия с базой метаданных

3.3.1 Физическое моделирование

Финальным этапом моделирования репозитория портала управления является создание физической модели БД.

Физическая модель – это детализированное описание структуры базы данных, адаптированное к конкретной СУБД (например, PostgreSQL, MySQL). Она включает технические аспекты хранения, индексации и оптимизации производительности.

Для создания физической модели базы данных на основе созданной ранее логической модели сначала выполняется преобразование отношений логической схемы в таблицы физической схемы реляционной базы данных. После того как основные сущности будут сформированы необходимо преобразовать бинарные связи между отношениями логической схемы в связи между таблицами реляционной базы. Далее выполняется преобразование атрибутов отношений логической схемы в столбцы физической модели реляционной базы данных.

В реляционных базах данных нет явного представления отношения «многие-ко-многим». Поэтому при физическом моделировании данные отношения необходимо представить с помощью дополнительной промежуточной таблицы, образующей связи «один-ко-многим» с исходными таблицами.

В качестве идентификатора сущностей для поддержки взаимодействия со сторонними приложениями будет использоваться строковый код, который создается на основе UUID, используемого внутри основного серверного приложения. Данный код генерируется по уникальному алгоритму для каждой сущности, используемой в программе.

Физическая модель базы метаданных представлена на рисунке 11.

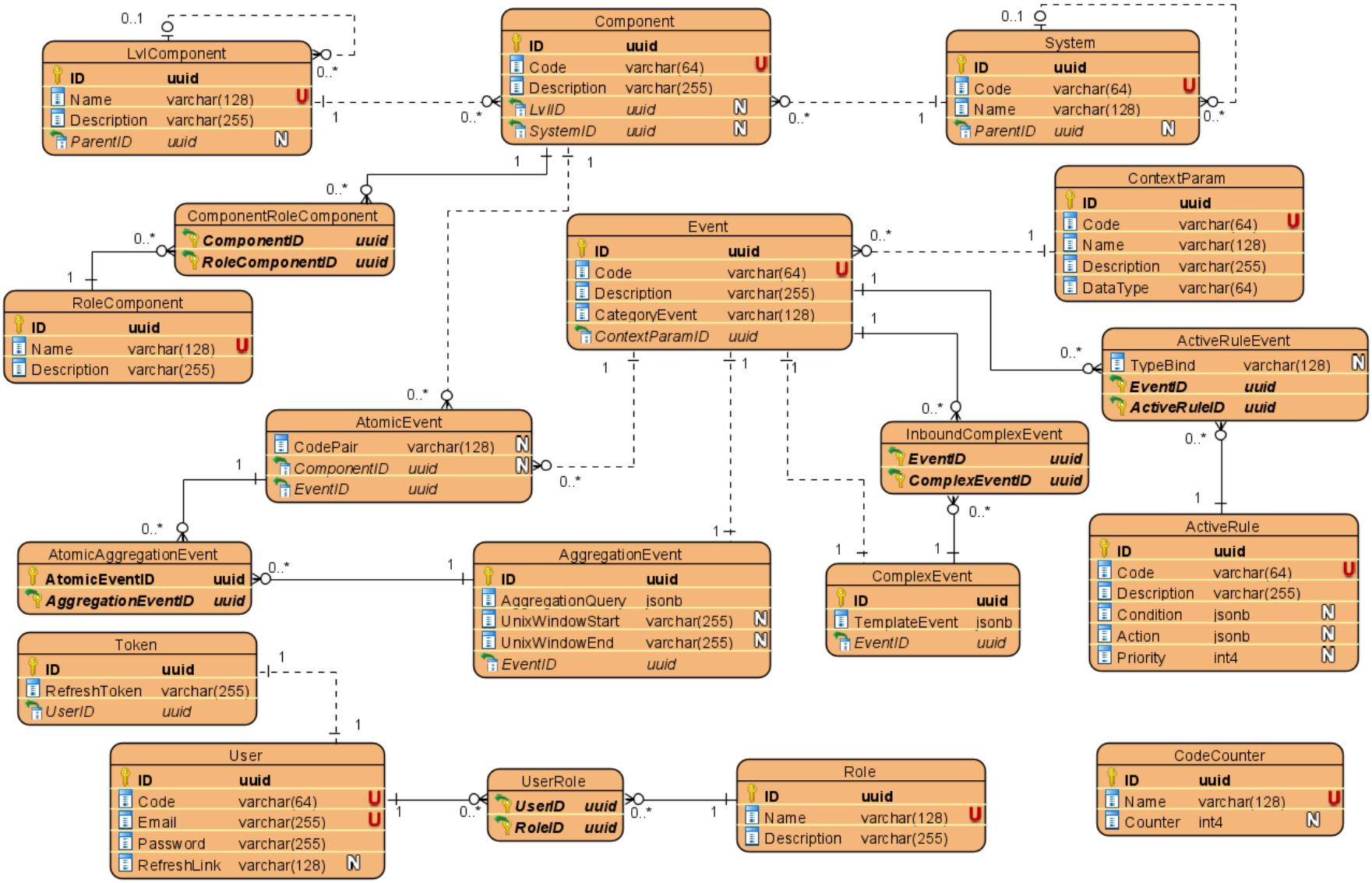

Рисунок 11 – Физическая модель репозитория метаданных

Visual Paradigm позволяет автоматически создавать SQL-скрипты, которые можно использовать для создания объектов базы данных. Данная функциональность CASE-инструмента значительно ускоряет процесс развертывания, позволяя мгновенно создать полную структуру базы данных в СУБД PostgreSQL без необходимости ручного написания SQL-кода.

3.3.2 Разработка структуры базы метаданных

Структура базы метаданных была реализована как реляционная база данных с помощью средства PostgesQL на основе разработанной физической модели.

Архитектура базы данных включает 19 сущностей, среди которых 2 сущности с иерархической организацией. Спецификация сущности Event представлена в таблице 8.

Таблица 8 – Спецификация сущности Event

Наименование сущности: Event
Назначение сущности: хранение информации о событиях
Атрибуты: id (uuid) – уникальный идентификатор события; codePair (varchar(255)) – кодовая пара, связанная с событием; componentId (uuid) – внешний ключ, ссылающийся на сущность LuComponent; eventId (uuid) – внешний ключ, ссылающийся на сущность AtomicEvent; createdAt (timestamptz) – метка времени создания события; updatedAt (timestamptz) – метка времени последнего обновления события.

Спецификация сущности User представлена в таблице 9.

Таблица 9 – Спецификация сущности User

Наименование сущности: User
Назначение сущности: хранение информации о пользователях
Атрибуты: id (uuid) – уникальный идентификатор пользователя; uid (uuid) – уникальный идентификатор пользователя; code (varchar(255)) – код пользователя; email (varchar(255)) – электронная почта пользователя; password (varchar(255)) – пароль пользователя; roleId (uuid) – внешний ключ, ссылающийся на сущность Role; createdAt (timestamptz) – метка времени создания пользователя; updatedAt (timestamptz) – метка времени последнего обновления пользователя.

На рисунке 12 представлена схема готовой базы данных.

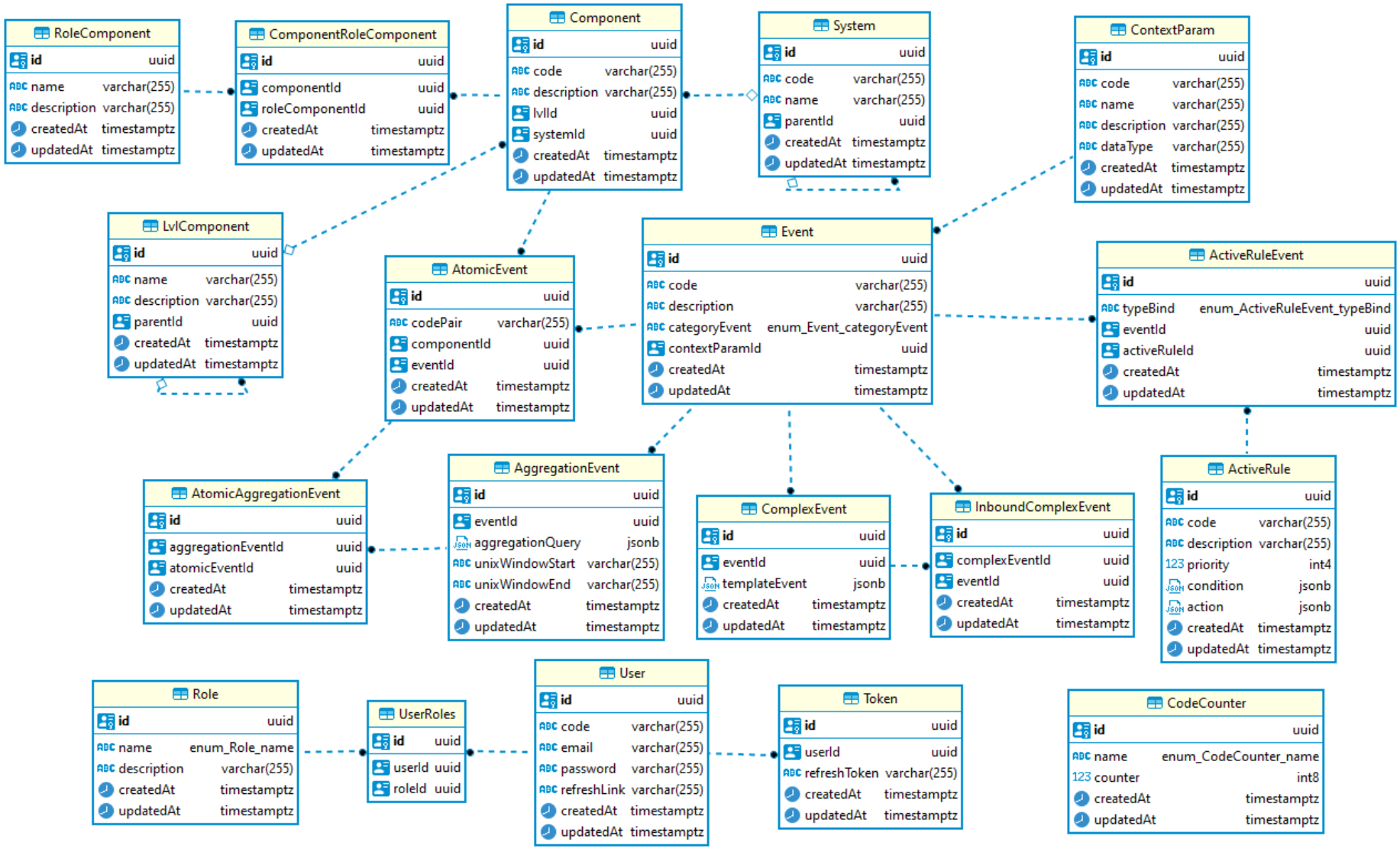

Рисунок 12 – Структура репозитория метаданных

Для управления и поддержки разработанной базы данных использовались средства DBeaver. Это средство дает возможность использования редактора SQL, управления соединениями БД и доступа к импорту/экспорту данных.

3.3.3 Разработка программных средств доступа к базе метаданных

Фреймворк NestJS обладает широкими возможностями для интеграции с различными типами баз данных, включая как реляционные (SQL), так и не-реляционные (NoSQL) решения. В рамках проекта для работы с PostgreSQL был выбран ORM-инструмент Sequelize, который предоставляет удобный интерфейс для взаимодействия с базой данных. Настройка подключения включает определение параметров соединения, аутентификационных данных и конфигурации пула подключений. На листинге 1 представлено подключение к базе данных.

Листинг 1 – Настройка подключения к PostgreSQL

```
import { Module } from '@nestjs/common';
import { SequelizeModule } from '@nestjs/sequelize';
import { DatabaseService } from './database.service';
import { User, Token, Role, UserRole } from '../models/user-re-
lated.models';
import { Component, RoleComponent, ComponentRoleComponent, Lvl-
Component } from '../models/component-related.models';
import { System, Event, ParamEvent } from '../models/system-re-
lated.models';

@Module({
    imports: [
        SequelizeModule.forRoot({
            dialect: 'postgres',
            host: process.env.POSTGRES_HOST,
            port: parseInt(process.env.DB_PORT, 10),
            username: process.env.POSTGRES_USER,
            password: process.env.POSTGRES_PASSWORD,
            database: process.env.POSTGRES_DB,
            autoLoadModels: true,
            synchronize: true,
        }),
        SequelizeModule.forFeature([
            User, Token, Role, UserRole,
            Component, RoleComponent,
            ComponentRoleComponent, LvlComponent,
            System, Event, ParamEvent
        ]),
```

```
    ],
    providers: [PostgresqlService],
    controllers: [],
    exports: [SequelizeModule]
})
export class PostgresqlModule {}
```

Sequelize обеспечивает надежную работу с PostgreSQL, поддерживая сложные запросы, транзакции и миграции. Благодаря встроенной TypeScript-поддержке достигается строгая типизация моделей и запросов, что повышает надежность кода.

Sequelize использует модели как объектное представление таблиц БД. Каждая модель описывает структуру таблицы – поля, типы данных, ключи и связи. Это позволяет работать с базой через JavaScript-объекты, в то время как Sequelize автоматически генерирует соответствующие SQL-запросы. Модели поддерживают строгую типизацию TypeScript и удобную работу с отношениями между таблицами. Ниже приведен код одной из моделей репозитория (листинг 2).

Листинг 2 – Модель таблицы ActiveRule

```
@Table({ tableName: 'ActiveRule' })
export class ActiveRule extends Model<ActiveRule, IActiveRu-leCre-
ate> {
    @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.INTEGER,
        defaultValue: 5,
        validate: {
            min: 1,
            max: 10,
        },
    })
    priority: number;

    @Column({
        type: DataType.JSONB,
        defaultValue: {},
    })
    condition: JSON;

    @Column({
        type: DataType.JSONB,
        defaultValue: {},
    })
    action: JSON;

    @BelongsToMany(() => Event, () => ActiveRuleEvent)
    event: Event[];
}
```

Модели данных внедряются в модули системы через механизм Dependency Injection, следуя заранее спроектированной архитектуре, сохраняя строгое соответствие между реализацией и первоначальным проектом.

3.4 Разработка базовых программных средств

3.4.1 Разработка программных средств управления метаданными

Основными объектами управления в системе являются события, активные правила и их окружение. События представляют собой действия, инициируемые пользователями или другими сервисами. Активные правила определяют логику обработки событий, задавая условия и последовательность действий, которые должны быть выполнены. Окружение включает дополнительные параметры и конфигурации, необходимые для настройки системы.

Для эффективного взаимодействия с базой данных через ORM используются заранее подготовленные представления (Views), что позволяет оптимизировать запросы – каждый метод получает только необходимые данные.

Архитектура включает 18 специализированных сервисов, ключевые из которых были детально проработаны на этапе проектирования и отображены на диаграмме компонентов.

Базовый функционал всех сервисов метаданных включает четыре стандартных операции: создание, чтение, обновление и удаление (CRUD). Помимо базовых операций, отдельные сервисы дополнены расширенной логикой обработки метаданных. Реализация сервисов для модулей событий и активных правил доступна в приложении Б. Сценарий управления метаданными сервиса ActiveRule приведен в таблице 10.

Таблица 10 – Сценарии управления метаданными сервиса ActiveRule

Сценарий	**Описание**
Bind Event	Связывание события с активным правилом по одному из двух типов: «Событие инициирует правило» или «Правило генерирует событие»
Unbind Event	Разрыв связи между событием и активным правилом
Find Events	Выявление всех событий, связанных с активными правилами

Сценарий управления метаданными сервиса Component приведен в таблице 11.

Таблица 11 – Сценарии управления метаданными сервиса Component

Сценарий	**Описание**
Add Role	Присвоение новой роли компоненту
Remove Role	Удаление назначенной роли с компонента
Add Lvl	Установка уровня для компонента
Remove Lvl	Снятие уровня компонента
Add System	Интеграция компонента в заданную систему
Remove System	Исключение компонента из состава системы

Сценарий управления метаданными сервиса Component приведен в таблице 12.

Таблица 12 – Сценарии управления метаданными сервиса Event

Сценарий	**Описание**
Add Atomic To Aggregation	Включение простого события в состав агрегированного
Remove Atomic To Aggregation	Удаление простого события из агрегированного
Add Inbound	Добавление простого события в структуру сложного
Remove Inbound	Исключение простого события из состава сложного

Сценарий управления метаданными сервиса Component приведен в таблице 13.

Таблица 13 – Сценарии управления метаданными сервиса System

Сценарий	**Описание**
Attach	Установление иерархической связи путем назначения дочерней системы родительской
Detach	Разрыв иерархической связи путем удаления дочерней системы из родительской

Для настройки взаимодействия модулей в сервисной архитектуре без жестких связей используется принцип внедрения зависимостей (Dependency Injection), позволяющий упростить процесс предоставления внешних зависимостей компоненту. Это помогает обеспечить слабую связанность системы.

На рисунке 13 изображена схема взаимодействия сервисов с использованием сервисной архитектуры на примере функции привязки активного правила к событию.

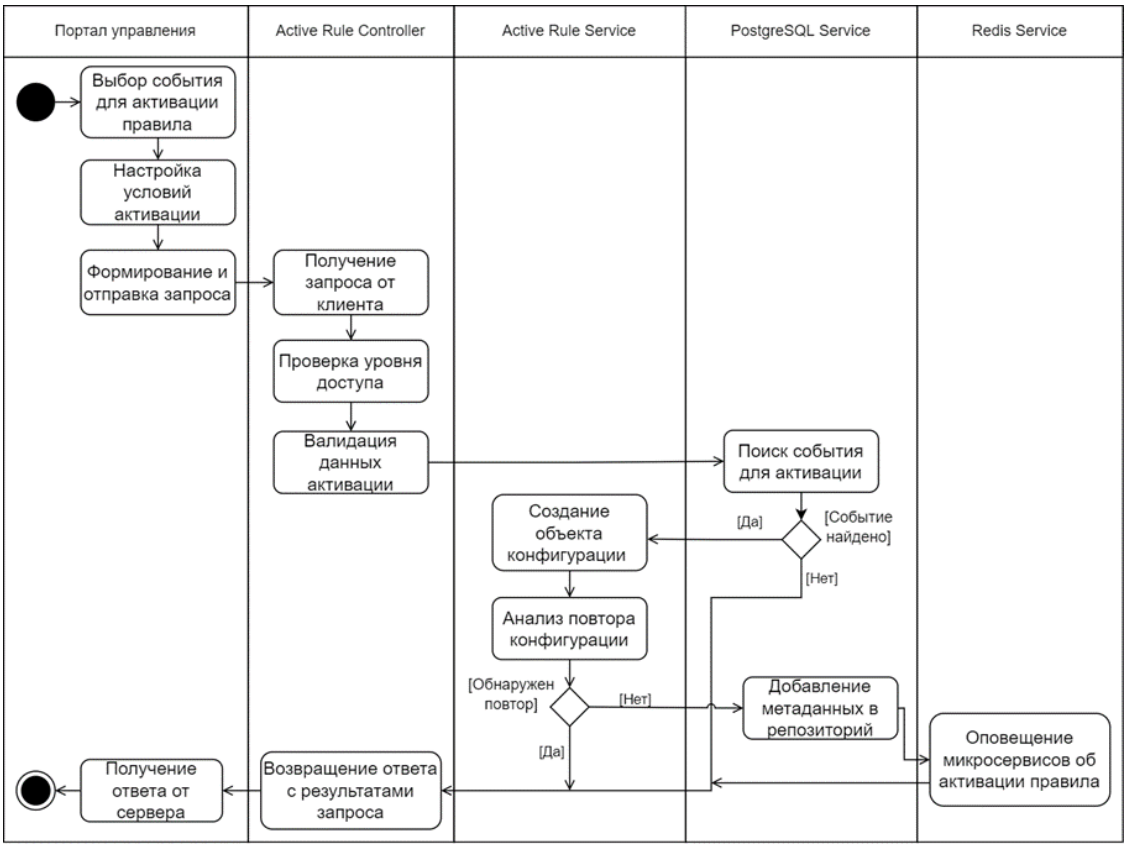

Рисунок 13 – Схема выполнения функции привязки активного правила к событию

Такой подход дает возможность сервисам общаться через четко определенные интерфейсы, сохраняя независимость реализации. Архитектура обеспечивает прозрачность взаимодействий, простоту модификации отдельных компонентов и легкость тестирования, при этом вся бизнес-логика инкапсулирована внутри сервисов, а внешние вызовы выполняются исключительно через их публичные API.

При инициации порталом запроса на привязку активного правила, система обрабатывает его через цепочку взаимодействующих компонентов. Входящий запрос сначала поступает в контроллер, который проверяет права доступа пользователя и корректность структуры данных. При обнаружении несоответствий запрос сразу отклоняется с соответствующей ошибкой. Более детально алгоритм аутентификации пользователей и валидации данных будет рассмотрен на последующих этапах.

После успешной проверки управление передается сервисному слою, где сервис активных правил выполняет основную бизнес-логику, сервис репозитория сохраняет изменения в базе данных, а сервис интеграции с микро-сервисами рассылает соответствующие уведомления. Эти уведомления позволяют микро-сервисам своевременно отслеживать изменения метаданных и поддерживать согласованность данных в распределенной системе.

3.4.2 Разработка программных средств анализа корректности метаданных

Для обеспечения надежности и целостности данных в портале управления метаданными на REST-сервере реализовано два подхода к валидации данных: статический анализ и динамический анализ.

Статическая проверка осуществляется непосредственно REST-сервером на этапе создания или редактирования с помощью валидации входящих параметров перед их сохранением в базе данных или использованием в других компонентах системы. Это уменьшает количество ошибок и обеспечивает согласованность

данных внутри портала управления.

Динамический анализ проводится во время выполнения операций и обработки бизнес-правил. Он помогает поддерживать корректность данных при их изменении в процессе работы программы.

Основными проблемами, связанными с корректностью и согласованностью метаданных являются:

- дублирование метаданных;
- конфликты бизнес-правил;
- циклические зависимости;
- логическая несогласованность.

Дублирование метаданных возникает, когда в системе создаются метаданные с одинаковыми характеристиками. Это может быть повторяющиеся записи или правила с аналогичной логикой. Дублирование может приводить к увеличению нагрузки на систему, затруднять ее масштабирование.

Для решения проблемы дублирования данных существуют алгоритмы, позволяющие обнаружить данные со схожим контекстом. Для этого проверяется их структура, источник, категория. Такие алгоритмы используются при вводе новых данных и при проектировании системы. В их основе лежат функции хэширования и сравнения ключевых атрибутов.

Более сложные алгоритмы используются при проверках сложных событий, так как создание данного типа событий включает несколько этапов, включающих в себя первичную валидацию структуры, входных данных, применение алгоритмов проверки привязки активных правил. Сценарий создания сложного события представлен на рисунке 14.

Еще одной проблемой согласованности данных является конфликты бизнес-правил. Они могут возникать, когда одно событие активирует несколько активных правил с противоречивыми результатами. Это приводит непредсказуемому поведению системы и способно привести к ошибке.

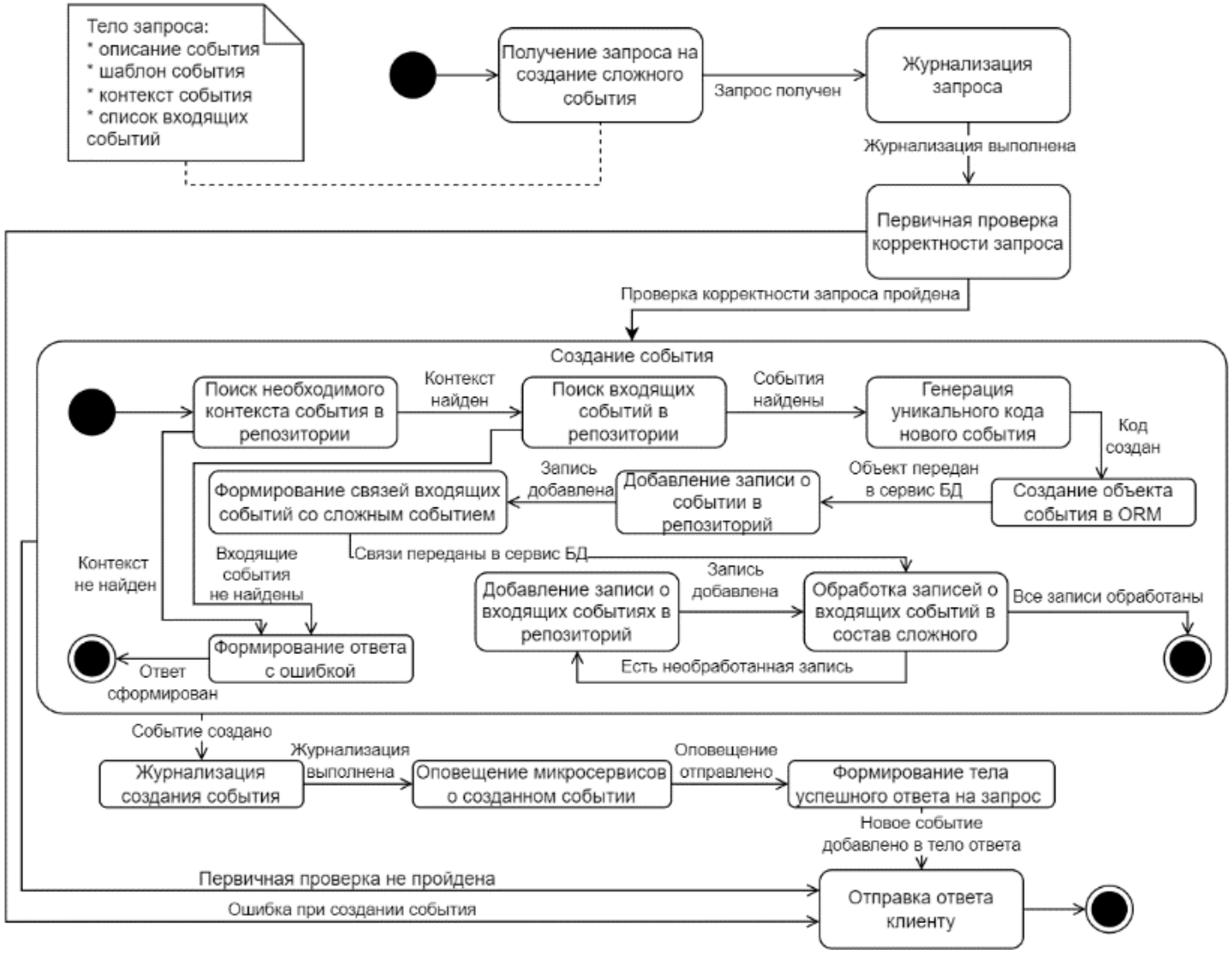

Рисунок 14 – Диаграмма состояний создания сложного события

Для предотвращения таких ситуаций реализован алгоритм, анализирующий условия и действия активных правил. Для выявления конфликтов проводится сравнение параметров действий, чтобы не допустить противоположных. Еще одним механизмом управления выполнением активных правил, способным предотвратить конфликты, является использование приоритетов. Их можно настроить при создании и редактировании правил. При выполнении правил алгоритм проверяет их приоритет и, если возникает ситуация совпадения приоритета для противоречивых правил, оповещает о возникновении коллизии.

Циклические зависимости возникают, когда выполнение одного правила приводит к активации другого. Это может вызвать ситуацию, при котором при любых действиях система будет возвращаться к исходному состоянию или уходить в бесконечный цикл, что вызывает ее перегрузку. Выделяют два типа циклов: гарантированный и негарантированный.

Гарантированный цикл – состояние, при котором выполнение правил неизбежно приводит к повтору исходных условий.

Негарантированный цикл – возникает, если повтор состояния возможен, но не обязателен.

Для обнаружения циклов реализованы алгоритмы, анализирующие конфигурации правил и выявляющие потенциальные конфликты.

Смысловая несогласованность означает, что действия, выполняемые системой, логически противоречат друг другу. Поскольку такие конфликты зависят от контекста внешней среды, их выявление происходит на этапе динамического анализа с помощью специализированных микросервисов.

3.4.3 Разработка программных средств управления пользователями

Регистрация новых пользователей выполняется через административную панель, где необходимо указать email и пароль. В системе реализован Mail-сервис, интегрированный с почтовым сервисом Google, который автоматически отправляет уведомления при регистрации, изменении и сбросе пароля. Данный

сервис обеспечивает надежную доставку всех системных уведомлений, связанных с управлением учетными записями.

Для обеспечения безопасности доступа к системе используются JWT-токены (JSON Web Token), которые прикрепляются к аккаунту пользователя.

Access-токен хранится в локальном хранилище браузера (LocalStorage), тогда как refresh-токен размещается в защищенной httpOnly cookie, что предотвращает его изменение через клиентские скрипты. Серверная реализация дополнительно сохраняет refresh-токены вместе с метаданными (например, IP-адресом клиента) для обеспечения безопасности. Их установка должна производиться исключительно сервером.

Refresh токен сохраняется на сервере вместе с дополнительными метаданными, такими как IP-адрес пользователя. Если при обновлении токена доступа выявляется другой IP-адрес, пользователю отправляется Email-уведомление о входе с другого устройства.

За обработку JWT-токенов отвечают следующие компоненты: JWTAuthController, JWTAuthService и JWTGenerateService. Их код представлен в приложении Г.

JWTAuthController является ключевым компонентом, отвечающим за обработку входящих запросов, связанных с аутентификацией и авторизацией. Этот контроллер служит точкой входа для клиентских приложений, принимая запросы на вход в систему, сброс пароля или обновление токенов доступа. В таблице 14 представлены методы данного класса.

Таблица 14 – Методы класса JWTAuthController

Метод	**Описание**
registration	Отвечает за обработку HTTP-запросов, валидацию формата данных для регистрации пользователей
login	Отвечает за обработку HTTP-запросов, валидацию формата данных для авторизации пользователей
logout	Отвечает за обработку HTTP-запросов выхода из системы
refresh	Отвечает за обработку HTTP-запросов обновления Access-токена, осуществляет работу с cookie

Спецификация класса контроллера представлена на рисунке 15.

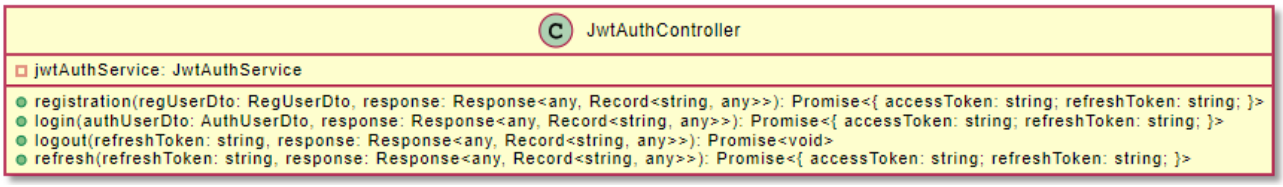

Рисунок 15 – Спецификация класса JwtAuthController

JWTAuthService реализует основную бизнес-логику процессов аутентификации и авторизации. Сервис отвечает за проверку пользовательских данных, их сопоставление с записями в базе данных и управление процессами входа и выхода из системы.

В таблице 15 представлены методы данного класса.

Таблица 15 – Методы класса JWTAuthService

Метод	**Описание**
registration	Отвечает за обработку HTTP-запросов, валидацию формата данных для регистрации пользователей
login	Отвечает за обработку HTTP-запросов, валидацию формата данных для авторизации пользователей
logout	Отвечает за обработку HTTP-запросов выхода из системы
refresh	Отвечает за обработку HTTP-запросов обновления Access-токена
createPayloadJWT	Отвечает за добавление полезной нагрузки для токена доступа

На рисунке 16 представлена спецификация этого сервиса.

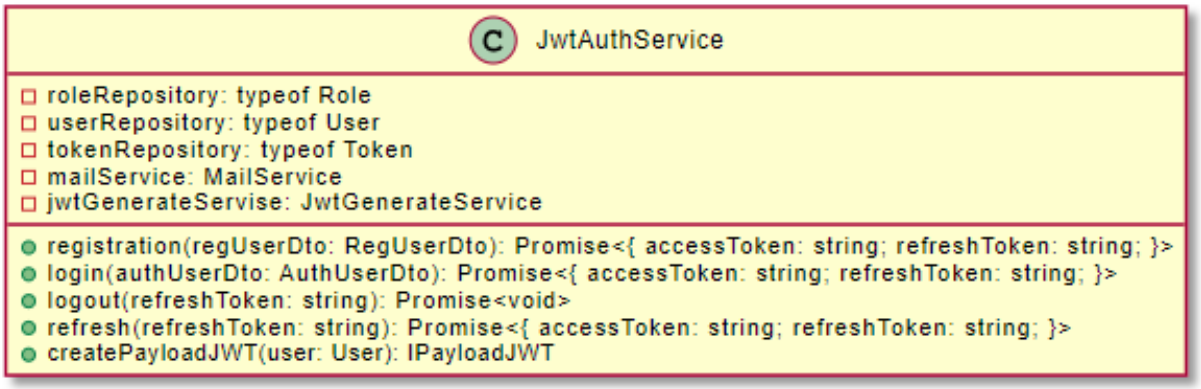

Рисунок 16 – Спецификация класса JwtAuthService

Для обеспечения доступа к данным пользователей, ролям, токенам, а также функциональности Email-сервиса и работы с JWT используется механизм

внедрения зависимостей. В таблице 16 представлены зависимости, необходимые для функционирования сервиса.

Таблица 16 – Зависимости класса JwtAuthService

Метод	**Описание**
roleRepository	Модель для работы с ролями пользователей
userRepository	Модель для работы с данными пользователей
tokenRepository	Модель для работы с Refresh токенами
mailService	Сервис верификации почты
jwtGenerateService	Сервис, отвечающий за логику создания JWT токенов, шифрование и валидацию

JWTGenerateService отвечает за генерацию, обновление и валидацию JWT-токенов. Сервис создает Access и Refresh токен, включая в них полезные данные (Payload) – специальный блок данных, содержащий необходимую информацию для работы клиентской части приложения. В этом блоке хранятся ключевые сведения о пользователе: его уникальный ID, email, дата последнего входа в систему и перечень назначенных ролей с соответствующими правами доступа.

3.5 Разработка программных средств взаимодействия с клиентскими приложениями

Система включает 10 специализированных контроллеров, каждый из которых отвечает за обработку HTTP-запросов по определённой тематике. Контроллеры выполняют роль промежуточного слоя: принимают входящие запросы, передают данные на обработку соответствующим сервисам бизнес-логики и возвращают клиенту сформированные ответы.

В таблице 17 отображен перечень используемых контроллеров и описание их назначения в системе.

Таблица 17 – Контроллеры взаимодействия с клиентскими приложениями

Название	**Описание**
User	Управление данными пользователей, включая создание, обновление, удаление и получение информации о пользователях
UserRole	Управление ролями пользователей, их назначение и удаление
System	Управление системами, их созданием, настройкой, подключением и отключением компонентов

Продолжение таблицы 17

Название	Описание
Component	Управление компонентами системы, их добавление, удаление, настройка свойств
ComponentRole	Назначение и удаление ролей для компонентов, определяющих их функциональные возможности
ComponentLvl	Управление уровнями компонентов, включая их добавление или удаление для настройки приоритетов или свойств
JWTAuthAndRegis-tration	Обработка аутентификации и регистрации пользователей с использованием JWT-токенов для обеспечения безопасности
Event	Управление событиями, включая их создание, удаление, привязку к правилам и обработку
ContextParamEvent	Управление контекстом событий, настройка и интеграция в процессы
ActiveRule	Управление активными правилами, создание, изменение и связывание с событиями

Все конечные точки REST API сервера подробно описаны в документации, которая содержит полную информацию о структуре запросов, поддерживаемых параметрах, форматах данных и возможных вариантах ответов. Документация охватывает как успешные сценарии работы с API, так и возможные ошибки с пояснением их причин. Для создания и поддержания документации используется Swagger – инструмент, который автоматически генерирует спецификацию API в соответствии со стандартом OpenAPI. Swagger предоставляет интерактивный интерфейс, позволяющий тестировать конечные точки непосредственно в браузере. Рассмотрим все точки доступа для взаимодействия с событиями сервиса (рисунок 17).

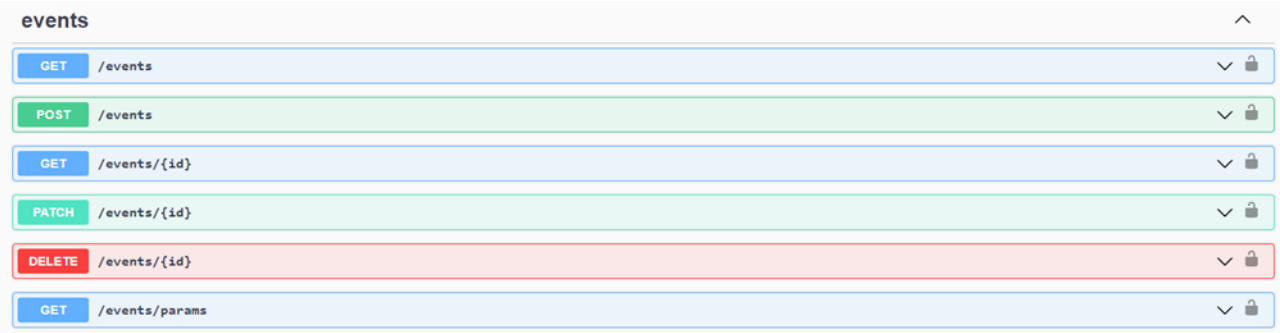

Рисунок 17 – Описание точек доступа для работы с событиями сервиса

Для понятного и эффективного взаимодействия с элементами API была составлена документация, в которой описывается метод, его функциональность,

тело запроса и возможные ответы сервера.

На рисунке 18 представлен пример документации эндпоинта «events», который отвечает за обновление объекта событий.

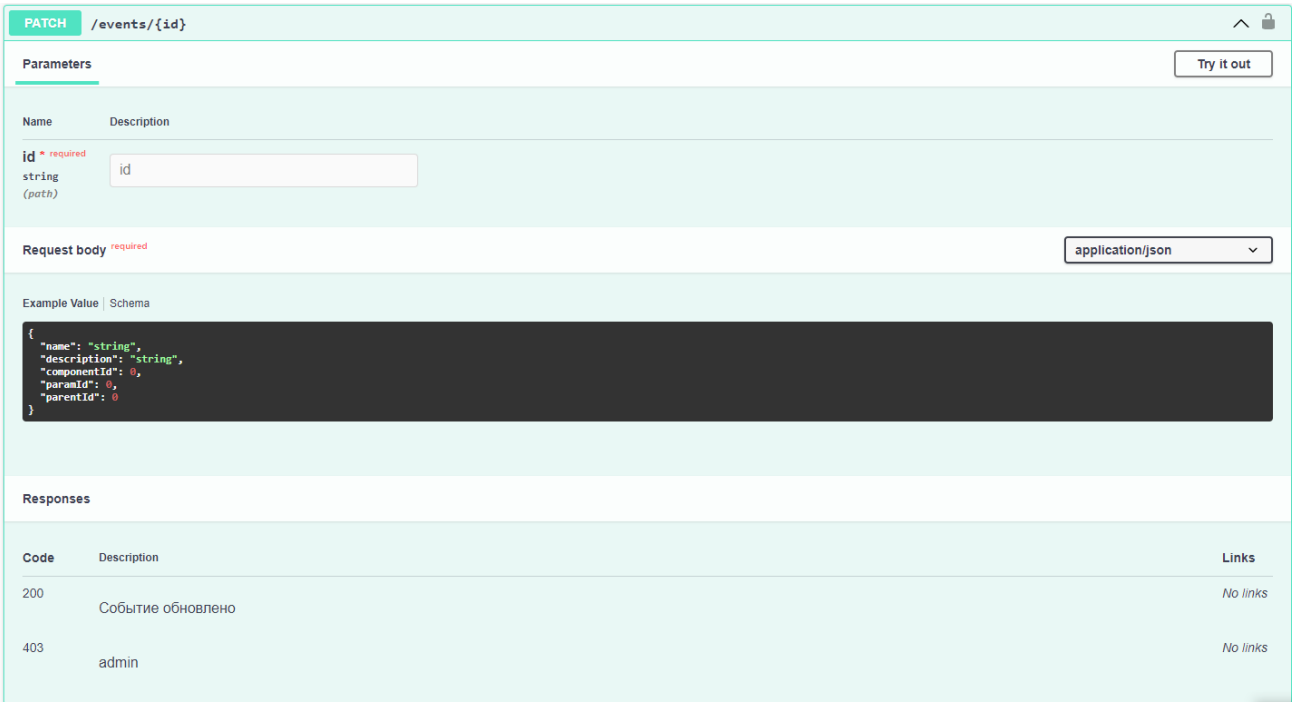

Рисунок 18 – Описание точки доступа /events/{id}

Данный фрагмент отображает следующие характеристики эндпоинта:

– метод PATCH, использующийся для частичного изменения данных объекта;

– объект event идентифицируется с помощью динамического параметра id в URL;

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

– возможные ответы сервера с кодами (200 – успешное изменение объекта, 403 – доступ запрещен, с указанием требуемой роли для доступа к данному запросу).

Для получения всей документации API необходимо обратиться по адресу сервера к точке доступа «api», предоставляющей информацию обо всех возможных запросах к серверу и допустимых параметрах. На данной странице также можно напрямую отправлять запросы к серверу при необходимости обратиться

к API из браузера.

На рисунке 19 представлена схема взаимодействия портала управления с клиентским приложением. При отправке запроса используется токен доступа.

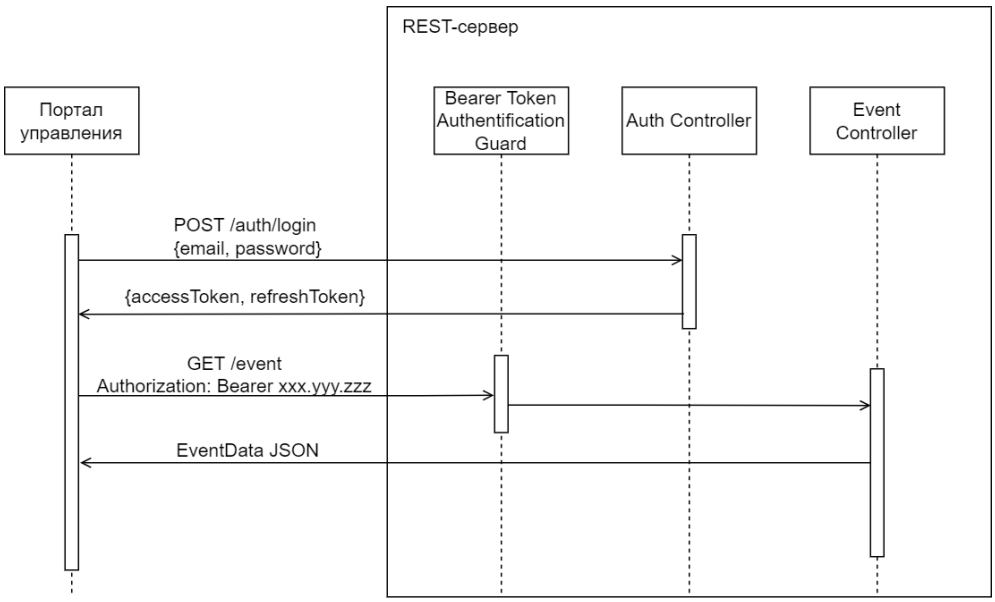

Рисунок 19 – Диаграмма последовательности взаимодействия с клиентским приложением

Для обеспечения безопасного взаимодействия с API был разработан механизм авторизации на основе механизма Guard, контролирующий доступ пользователей к защищенным конечным точкам. Как предусмотрено архитектурой системы, авторизация осуществляется через JWT-токен, который клиентское приложение передает в заголовке Authorization в стандартном формате Bearer TOKEN. В случае отсутствия необходимых прав система возвращает статусный код «403 Forbidden», блокируя выполнение запроса.

Обработка каждого запроса реализована в строгом соответствии с проектной документацией и включает многоуровневую систему обработки: промежуточное ПО (Middleware) для первичной валидации, двойной механизм контроля доступа (Guard), систему преобразования и проверки входных данных (DTO-валидация), набор перехватчиков (Interceptor) для дополнительной обработки данных, а также единый механизм обработки исключений. Полная реализация этих компонентов представлена в приложении В.

Пример ошибки валидации представлен на рисунке 20.

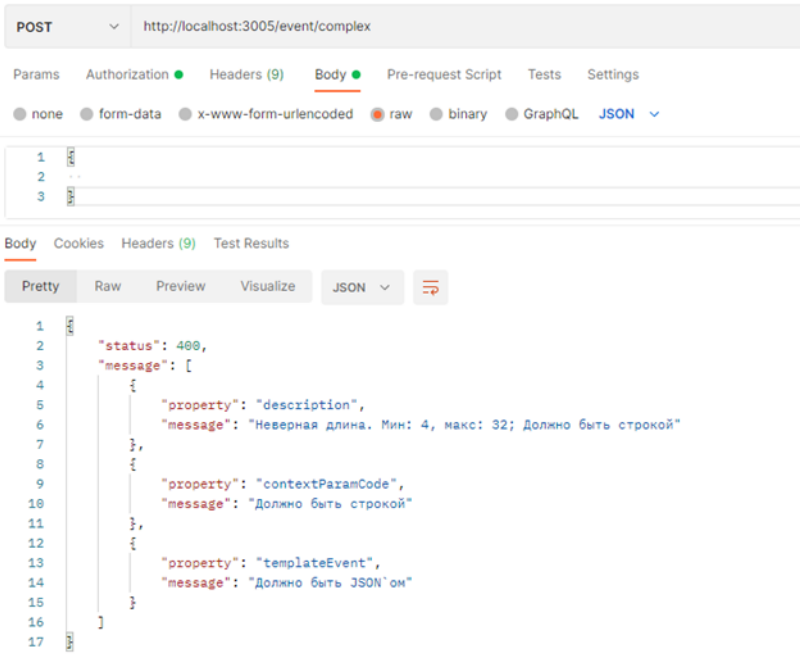

Рисунок 20 – Ответ от сервера с ошибочной первичной валидацией

Для валидации данных запроса в платформе используется механизм Data Transfer Object (DTO), представляющих специальный шаблон проектирования для структурированной передачи данных. Контроллеры при получении запроса сравнивают полученную информацию с данным шаблоном. Валидация реализована с применением декораторов, которые автоматически проверяют соответствие входящих данных заданным критериям. При обнаружении несоответствий система генерирует исключение, которое перехватывается глобальным фильтром исключений. Проверяющий модуль создает исключение и преобразует в ответ с HTTP-статусом 400 Bad Request, содержащий информацию об ошибках валидации.

Объявление декораторов представлено в приложении В. Для реализации используется готовая библиотека Class-Validator, которая включает множество готовых декораторов для проверки различных типов данных. Пример использования этих декораторов для описания параметров класса, участвующих в теле запроса при создании активного правила, приведен в листинге 3.

Листинг 3 – Использование реализованных декораторов валидации

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

    @ApiProperty({
      type: String,
      description: 'JSON condition for the rule'
    })
    @IsJSONRus()
    condition: JSON;

    @ApiProperty({
      type: String,
      description: 'JSON action for the rule'
    })
    @IsJSONRus()
    action: JSON;
}
```

Схема алгоритма первичной проверки корректности запроса изображена на рисунке 21.

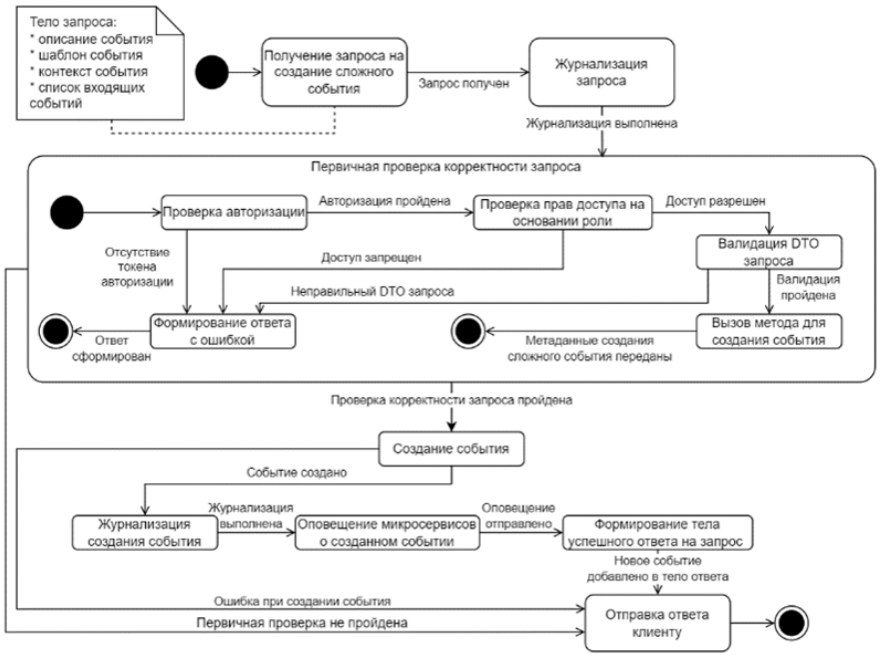

Рисунок 21 – Диаграмма создания сложного события с первичной проверкой запроса от клиентского приложения

Процесс первичной проверки состоит из трех этапов. На первом этапе осуществляется проверка авторизации, которая подтверждает, что пользователь имеет действительные учетные данные. На втором этапе проводиться проверка прав доступа на основе роли. На последнем этапе происходит валидация DTO запроса, при которой проверяется соответствие структуры и типов данных в теле запроса.

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 комплекс мероприятий, направленный на поддержание соответствия разрабатываемого продукта предъявляемым требованиям. Он ведется на всех этапах разработки программного обеспечения, таких как определение требований, проектирование, написание программы, контроль исходного кода и тестирование.

Основными критериями качества программного обеспечения являются:

- функциональность – способность программного обеспечения выполнять установленные требованиями функции;
- надежность – способность программного обеспечения работать без ошибок;
- удобство использования – доступность программного обеспечения для эксплуатации пользователями;
- эффективность – оптимальное использование ресурсов для решения поставленных задач;
- удобство сопровождения – возможность поддержки, расширения и доработки программного обеспечения;
- портативность – доступность программного обеспечения на разных платформах;
- совместимость – возможность для взаимодействия с различными сторонними программами;
- защищенность – уровень защиты конфиденциальных данных.

Существует большое количество подходов к контролю качества серверных приложений. Они направлены на оценку и обеспечение основных критериев качества. К таким мероприятиям относят:

- планирование качества;

- тестирование;
- анализ кода;
- мониторинг и логирование;
- автоматизация процессов;
- составление документации;
- проверка масштабируемости.

Среди основных мероприятий чаще всего используют тестирование и анализ кода, так как они наиболее приближены непосредственно к процессу разработки.

Тестирование программного обеспечения заключается в непосредственной проверке на соответствие предъявляемым требованиям. Выделяют тестирование двух видов: автоматизированное и ручное. Автоматизированное тестирование заключается в использовании различных программ для выполнения тестов. Такой вид тестов используется для часто повторяющихся тестов или каких-либо типовых проверок, что уменьшает возможность человеческой ошибки. Это помогает обнаружить незначительные ошибки. Ручное тестирование проводится непосредственно человеком. При этом тщательно проверяется функционал приложения, визуальные компоненты. В этом случае могут быть выявлены серьезные ошибки [19].

К видам автоматического тестирования относятся:

- дымовое тестирование (smoke testing) – проверка базовых функций программного обеспечения;
- модульное тестирование (unit testing) – проверка корректной работы модуля;
- интеграционное тестирование – проверка взаимодействия отдельных компонентов программы;
- нагрузочное тестирование – проверка работоспособности программы при большой нагрузке;

– регрессионное тестирование – проверка работоспособности кода при обновлении.

Видами ручного тестирования являются:

– модульное тестирование – проверка отдельных компонентов программы;

– интеграционное тестирование – проверка взаимной работы частей приложения;

– системное тестирование – проверка на соответствие исходным требованиям;

– приемочное тестирование – проверка бизнес-требований.

При оценке качества программных средств обычно заранее происходит выбор видов тестирования в зависимости от сложности программного обеспечения и его типа.

Анализ кода позволяет оценить качество кода, возможность его поддержки и расширения, выявить ошибки и уязвимости. К анализу кода также относятся метрики, которые позволяют изучить сложность и надежность оцениваемого программного кода. Метрики разделяются на классы:

– метрики сложности – необходимы для оценки сложности понимания кода программы;

– метрики сопровождаемости – используются для оценки возможности поддержки;

– метрики покрытия тестами – оценка использования тестов;

– метрики безопасности – используются для выявления потенциальных угроз безопасности;

– метрики производительности – оценка скорости работы программы и использования ресурсов;

– метрики стиля кода – соответствие стандартам написания кода.

4.1.2 Обоснование выбора типов автоматизированного тестирования

Особенностью разработанного приложения является его модульная архитектура и слабая связь отдельных компонентов. Поэтому для тестирования приложения были выбраны модульное и интеграционное тестирование. Это обеспечит большое покрытие кода тестами, гарантируя надежность отдельных сервисов и слаженность их работы.

Модульные тесты позволят проверить работоспособность отдельных компонентов программы независимо от остальной части приложения. Это не допускает влияния потенциальных ошибок межсервисного взаимодействия или проблем в других модулях. К преимуществам модульного тестирования относится:

- простота написания отдельных тестов;
- расширение понимания кода и назначения функций;
- возможность регрессионного тестирования.

Интеграционное тестирование необходимо для проверки правильности работы модулей при взаимодействии друг с другом. Интеграционные тесты непосредственно имитируют клиентские запросы к серверу, так как в обработке каждого запроса участвует множество сервисов. Такие тесты позволяют выявлять ошибки, которые невозможно обнаружить при модульном тестировании, такие как несоответствие форматов данных или нарушения конфигурации зависимостей.

4.1.3 Обоснование выбора метрик программного кода

Для обеспечения контроля качества разработанной системы использовались метрики анализа абстрактности и устойчивости программного обеспечения и метрика Холстеда для анализа исходного кода.

Метрика абстрактности необходима для определения степени использования абстрактных классов и интерфейсов. Для ее вычисления необходимо найти отношение числа абстрактных классов к общему числу классов в компоненте. Показатель абстрактности отображает гибкость и модульность системы, чем он

выше. Метрика устойчивости показывает какие трудозатраты необходимы для изменения зависимых компонентов при изменении анализируемого сервиса. Эта метрика основана на количестве внешних зависимостей, которые затрагивает изменение, и позволяет оценить изолированность компонента.

Метрика Холстеда используется для анализа исходного кода и основана на подсчете синтаксических элементов, таких как операторы и операнды, а также общего числа строк кода. Эта метрика используется для оценки сложности реализации конкретного фрагмента кода. Метрика предоставляет следующие показатели: объем программы, уровень сложности и трудозатраты на разработку.

4.2 Автоматизированное тестирование

4.2.1 Модульное тестирование компонентов сервера

Для проведения автоматизированного модульного тестирования был выбран фреймворк Jest, обеспечивающий удобную настройку и запуск тестов [20]. Jest предоставляет встроенные функции для создания тестов. С помощью данного фреймворка можно имитировать поведение классов и методов, проверяя передаваемые параметры и совпадение результата с ожидаемым. В таблице 18 приведена спецификация модульных тестов, которые проводились в системе.

Таблица 18 – Спецификация модульных тестов

Контроллер	**Метод**	**Тестовая ситуация**	**Ожидаемый результат**
UserController	GET /users	Получение списка пользователей	Код 200 JSON с массивом пользователей
UserController	GET /users/{id}	Получение пользователя по ID	Код 200 JSON с данными пользователя
UserController	POST /users	Создание пользователя	Код 201 JSON с данными нового пользователя
ComponentController	GET /components	Получение списка компонентов	Код 200 JSON с массивом компонентов
ComponentController	POST /components	Проверка обработки некорректных данных	Код 400 JSON с описанием ошибки

Продолжение таблицы 18

Контроллер	Метод	Тестовая ситуация	Ожидаемый результат
ComponentController	DELETE /components/{id}	Удаление компонента	Код 204 Пустой ответ
ActiveRoleController	POST /activeRoles	Создание активного правила	Код 201 JSON с данными нового активного правила
ActiveRoleController	PUT /activeRoles/{id}	Обновление активного правила	Код 200 JSON с обновленными данными
ActiveRoleController	DELETE /activeRoles/{id}	Удаление активного правила	Код 204 Пустой ответ

Все модульные тесты были успешно пройдены, ошибок в работе компонентов программы не выявлено.

Для примера приведены модульные тесты компонентов управления ролями: RoleController, который отвечает за обработку HTTP-запросов от клиентских приложений, и RoleService, реализующего бизнес-логику управления ролями. Для тестирования были созданы экземпляры данных классов, чтобы гарантировать изолированную проверку их функциональности. Код модульных тестов представлен в приложении Д.

При тестировании проверялись следующие параметры:

- корректность объявления и реализации методов классов;
- правильность возвращаемых данных для успешных сценариев и обработки ошибок;
- создание новых объектов ролей с валидацией входных данных;
- поиск объектов по уникальному идентификатору;
- удаление ролей по идентификатору;
- обработка исключительных ситуаций при некорректных входных параметрах;
- количество вызовов методов и их последовательность.

Для эмулирования зависимостей, таких как взаимодействие с базой данных и внешними сервисами, использовались моки (mocks) – искусственные объекты, имитирующие функции реальных объектов.

Результаты тестов представлены на рисунке 22.

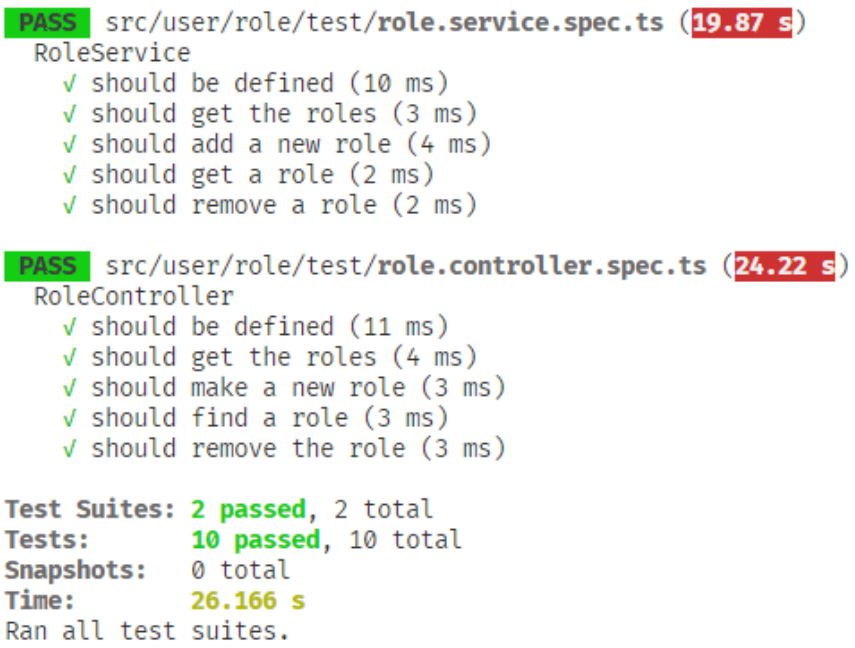

Рисунок 22 – Результаты тестирования RoleService и RoleController

Проведенные тесты показали корректность работы тестируемых модулей.

4.2.2 Интеграционное тестирование компонентов сервера

Для проведения интеграционного тестирования использовалась конфигурация, аналогичная той, что применялась при модульном тестировании. Из-за использования на сервере механизмов защиты от неавторизованного доступа был реализован компонент MockAuthGuard, позволяющий осуществлять запросы в тестовой среде без проверки прав доступа. В таблице 19 приведена спецификация интеграционных тестов.

Таблица 19 – Спецификация интеграционных тестов

Тестовый сценарий	**Метод**	**Входные данные**	**Ожидаемый результат**
Назначение роли пользователю	POST /users/{id}/roles	ID пользователя JSON с ID роли	Код 200 JSON обновленного пользователя с ролью
Назначение несуществующей роли пользователю	POST /users/{id}/roles	ID пользователя JSON с ID несуществующей роли	Код 404 JSON с сообщением об ошибке

Продолжение таблицы 19

Тестовый сценарий	Метод	Входные данные	Ожидаемый результат
Удаление роли у пользователя	DELETE /users/{id}/roles/{roleId}	ID пользователя ID роли	Код 204 Пустой ответ
Удаление несуществующей роли у пользователя	DELETE /users/{id}/roles/{roleId}	ID пользователя ID несуществующей роли	Код 404 JSON с сообщением об ошибке

Все интеграционные тесты были успешно пройдены, ошибок во взаимодействии компонентов не выявлено.

В качестве примера приведено интеграционное тестирование функционала работы с пользователями, затрагивающего компоненты RoleController и RoleService. В рамках данного тестирования проверялось взаимодействие компонентов друг с другом и с базой данных при выполнении CRUD-операций. Код интеграционных тестов представлен в приложении Д.

В ходе тестирования было проверено:

– доступность маршрутов для обработки HTTP-запросов;

– корректность обработки входящих запросов модулями;

– правильность структуры и содержания объектов, возвращаемых модулями;

– работа компонента перехватчика (Interceptor), ответственного за обработку ошибок;

– корректность возвращаемых статус-кодов при успешной обработке запроса.

Пример тестирования модуля ролей представлен на рисунке 23.

```
PASS src/user/role/test/role.e2e.spec.ts (25.042 s)
  Role (E2E)
    √ GET /role (11 ms)
    √ GET /role with name (2 ms)
    √ GET /role with non-existent name (2 ms)
```

Рисунок 23 – Результаты тестирования

Проведенные тесты показали корректность исполнения запросов в тестируемых модулях.

4.3 Оценка качества программного кода

4.3.1 Оценка на основе метрики Холстеда

Для оценки сложности и качества кода разработанного серверного приложения были рассчитаны метрики Холстеда. Для примера рассмотрен JwtGenerateService, который отвечает за создание и валидацию токенов доступа и обновления.

Для расчета метрики учитываются следующие параметры программного кода:

- число уникальных операторов программы;
- число уникальных операндов программы;
- общее число операторов в программе;
- общее число операндов в программе.

В листинге 4 представлен исходный код класса JwtGenerateService. Данный класс содержит 48 физических и 20 логических строк кода.

Листинг 4 – Класс JwtGenerateService

```
@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_AC-
CESS_SECRET, {
            expiresIn: '30m',
        });
        const refreshToken = jwt.sign(payload, process.env.JWT_RE-
FRESH_SECRET, {
            expiresIn: '14d',
        });
        await this.saveRefreshToken(userId, refreshToken);
        return {
            accessToken,
        };
    }
```

```
    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_AC-
CESS_SECRET);
            return userData;
        } catch (err) {
            return null;
        }
    }

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

Для оценки сложности необходимо подсчитать количество уникальных операторов и операндов и их общее число. Операторы данного класса представлены в таблице 20.

Таблица 20 – Операторы

Операторы	**Число вхождений**
@Injectable()	1
export	1
class	1
constructor	1
@InjectModel	1
private	1
async	2

Продолжение таблицы 20

Операторы	**Число вхождений**
await	3
=>	4
try	2
catch	2
return	6
if	1
.	7
=	5
{}	8
()	10
:	6
jwt.sign	2
jwt.verify	2
findOne	1
save	1
create	1
where	1

Операнды представлены в таблице 21.

Таблица 21 – Операнды

Операнды	**Число вхождений**
JwtGenerateService	1
tokenRepository	3
Token	2
payload	3
IPayloadJWT	1
userId	4
String	4
accessToken	2
refreshToken	4
jwt	4
’30m’	1
‘14d’	1
expiresIn	2
tokenData	3
userData	2
err	2

Данный код всего содержит 70 операторов (N1), 24 уникальных оператора (n1), 41 операнд (N2) и 17 уникальный операндов (n2). Для получения метрики необходимо провести расчеты по формулам, представленным ниже.

Для начала необходимо рассчитать длину программы (N) по формуле:

$$N = N1 + N2 = 70 + 41 = 111.$$

Далее необходимо рассчитать длину словаря (n) по формуле:

$$n = n1 + n2 = 24 + 17 = 41.$$

Объем программы (HV) рассчитывается по формуле:

$$HV = N * log2(n) = 111* 5.36 = 594.69.$$

Сложность программы (HD) рассчитывается по формуле:

$$HD = (n1 / 2) * (N2 / n2) = 24 / 2 * 41 / 17 = 28.9.$$

На основе полученных данных можно сделать вывод, что показатели находятся в норме. Это означает, что код является легко поддерживаемым.

4.3.2 Оценка на основе метрики устойчивости и абстрактности

Оценка на основе метрик устойчивости и абстрактности показывает качество проектирования модулей приложения с точки зрения гибкости, удобства поддержки и устойчивости к изменениям. Для оценки на основе этих метрик необходимо рассмотреть архитектуру сервера, его модули, классы и их зависимости.

Чтобы рассчитать соотношения между абстрактностью и устойчивостью необходимо выполнить ряд вычислений. 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 абстрактности (A) варьируется от 0 до 1, где 0 указывает на отсутствие абстрактных классов в компоненте, а 1 – на наличие исключительно абстрактных классов. Ниже представлены полученные значения.

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 класса.

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:

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

Компонент обладает невысокой устойчивостью, поскольку часто импортируется другими компонентами и выполняет ключевую функцию. Реструктуризация этого компонента нецелесообразна, так как он не включает бизнес-логику и служит для взаимодействия с СУБД PostgreSQL.

На основе рассчитанных метрик абстрактности и устойчивости для каждого компонента сервера было определено нормированное расстояние до главной последовательности с использованием формулы:

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

где A – мера абстрактности;

I – мера неустойчивости.

Чем ближе точка располагается к главной последовательности (показатель ближе к 0), тем лучше. Расстояния до главной последовательности для различных компонентов представлены в таблице 22.

Таблица 22 – Результаты расчета нормированного расстояния

Компонент	**Нормированное расстояние**
User	0,15
JWT Auth	0,18
Events	0,26
System, Active Rule, Component	0,12
Postgre	0,27

После выполнения расчетов компоненты можно представить на графике (рисунок 24). На вертикальной оси отложена метрика абстрактности (A), а на горизонтальной – метрика устойчивости (I).

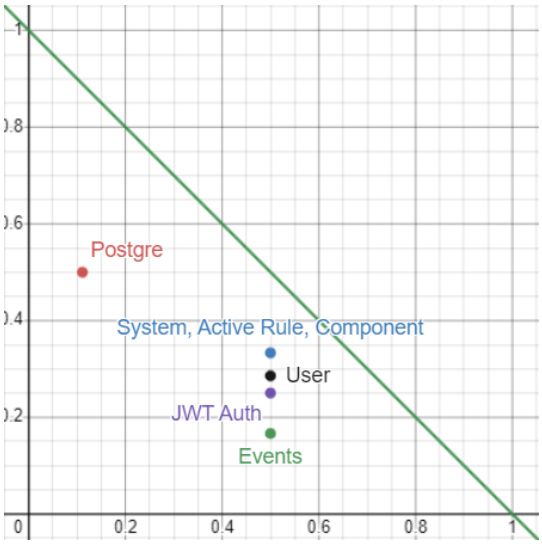

Рисунок 24 – Отображение метрик устойчивости и абстрактности

Все компоненты лежат близко к главной последовательности, что означает оптимальный баланс между устойчивостью и абстрактностью. Это указывает на высокое качество разработанного программного обеспечения, которое легко поддерживать и расширять.

Заключение

В рамках выполнения выпускной квалификационной работы был разработан RESTful API сервер управления метаданными в составе информационно-управляющей платформы.

В ходе разработки программных средств был проведен анализ предметной области, определены ключевые функциональные требования. Выполнен обзор существующих решений в данной области, выявлены их преимущества и недостатки, которые были учтены на стадиях проектирования и реализации системы.

Было выполнено описание механизмов управления метаданными. Сформированы требования к программному обеспечению. По результатам анализа составлен план проведения этапов разработки программного обеспечения. Проведено планирование и оценка бюджета при помощи средств MS Project. В результате планирования затраты времени составили 70 часов, а денежные затраты 135500 рублей.

На этапе проектирования была разработана архитектура, сформированы основные функциональные требования, проведено логическое и физическое моделирование репозитория метаданных, спроектированы модули REST-сервера. На этом этапе был определен и обоснован технологический стек.

Разработан RESTful API сервер портала управления. Основные этапы разработки включали: проектирование архитектуры развертывания, создание базы данных и инструментов для работы с ней, разработку механизмов управления метаданными и алгоритмов проверки их корректности, реализацию систем регистрации и аутентификации пользователей, а также разработку инструментов для взаимодействия с клиентскими приложениями и микросервисами. Разработка велась в среде Visual Studio Code с использованием языка TypeScript и фреймворка NestJS. В качестве среды выполнения выбрана платформа Node.js, основанная на движке V8. Для работы веб-сервера применялся Nginx. Программные компоненты развертывались в Docker-контейнерах, что обеспечивало их высокую мобильность и независимость от прикладного и системного ПО.

Проведена оценка качества серверных программных компонентов. Были разработаны автоматизированные тесты, а также рассчитаны метрики устойчивости, абстрактности и метрики Холстеда. Все этапы разработки сопровождались схемами и пояснительной документацией. Разработанное ПО полностью соответствует требованиям технического задания.

Список использованных источников

1. Nitesh Upadhyaya / Low-Code/No-Code Platforms and Their Impact on Traditional Software Development: A Literature Review / California State University, Long Beach, 2023.

2. Leonard Richardson, Mike Amundsen, Sam Ruby / RESTful Web APIs / O'Reilly Media, Inc., 2013. – 379.

3. Newman Sam / Building Microservices: Designing Fine-Grained Systems / Wiley, 2018. – 250.

4. Что такое визуальный язык программирования и почему он меняет разработку приложений? [Электронный ресурс]. URL: https://appmaster.io/ru/blog/vpl-izmenenie-razrabotki-prilozhenii (дата обращения 18.05.2025).

5. Возможности Low-Code платформы: программирование без специальных навыков [Электронный ресурс]. URL: https://simpleone.ru/blog/low-code-capabilities-programming-without-special-skills (дата обращения 20.05.2025).

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

7. Jakub Korab / Understanding Message Brokers / O'Reilly Media, Inc., 2017.

8. Pattern: Database per service // Microservice Architecture [Электронный ресурс]. URL: https://microservices.io/patterns/data/database-per-service.html (дата обращения 10.05.2025).

9. Что такое метаданные. Объясняем простыми словами [Электронный ресурс]. URL: https://secretmag.ru/enciklopediya/chto-takoe-metadannye-obyasnyaem-prostymi-slovami.htm (дата обращения 17.05.2025).

10. Что такое JSON Web Token (JWT), как он работает и зачем нужен [Электронный ресурс]. URL: https://blog.skillfactory.ru/chto-takoe-json-web-token-jwt/ (дата обращения 21.05.2025).

11. Low-code платформы 2025: сравнительный обзор [Электронный ресурс]. URL: https://habr.com/ru/articles/867032/ (дата обращения 23.05.2025).

12. Visual Paradigm Quick Start // [Электронный ресурс]. URL: https://cdn.visual-paradigm.com/quickstart/quickstart.pdf (дата обращения: 30.04.2025).

13. REST API: что это такое и как работает [Электронный ресурс]. URL: https://practicum.yandex.ru/blog/chto-takoe-rest-api-i-kak-rabotaet/#chto-takoye-rest-api (дата обращения 30.05.2025).

14. NestJS [Электронный ресурс]. URL: https://insayt.github.io/nestjs.ru.com (дата обращения 15.05.2025).

15. Philippe Kruchten / The Rational Unified Process: An Introduction / Addison-Wesley, 2003.

16. Overview of Docker // Docker docs [Электронный ресурс]. URL: https://docs.docker.com (дата обращения 02.05.2025).

17. API Development forEveryone [Электронный ресурс]. URL: https://swagger.io/ (дата обращения 02.05.2025).

18. Веб-сервер Nginx и что о нём должен знать каждый разработчик [Электронный ресурс]. URL: https://practicum.yandex.ru/blog/chto-takoe-nginx/ (дата обращения 28.05.2025).

19. Сэм Канер, Джек Фолк, Хунг Куок Нгуен / Тестирование программного обеспечения / Издательство «ДиаСофт», 2001. – 544 с.

20. Использование с TypeScript [Электронный ресурс]. URL: https://v1.test-utils.vuejs.org/ru/guides/using-with-typescript.html (дата обращения 07.06.2025).

Приложение А

(справочное)

Глоссарий

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.

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

Representational State Transfer (REST) – это набор правил того, как программисту организовать написание кода серверного приложения, чтобы все системы легко обменивались данными и приложение можно было масштабировать.

Application programming interface (API) – это набор способов и правил, по которым различные программы общаются между собой и обмениваются данными.

Integrated Development Environment (IDE) – интегрированная среда разработки, набор ПО для создания кода.

Информационно-управляющая система (ИУС) – цифровая система контроля или управления техническим устройством.

JavaScript Object Notation (JSON) – это формат передачи данных, который используется при взаимодействии веб-сервера и браузера.

JSON Web Token (JWT) – это открытый стандарт (RFC 7519) для создания токенов доступа, основанный на формате JSON.

Object Relational Mapping (ORM) – это технология, которая позволяет работать с базами данных так, как если бы это были объекты из языков программирования.

Приложение Б

(обязательное)

Листинг сервисов моделей

Листинг Б.1 – Файл active-rule.service.ts

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

    readonly findModel = {
        attributes: ['code', 'description', 'condition', 'ac-
tion'],
        include: {
            model: Event,
            attributes: ['code', 'description', 'categoryEvent'],
            through: { as: 'association', attributes: ['typeBind']
},
        },
    };

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

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

    findAllRulesSystem(code: string) {
        const findModelBySystem = (codeSystem: string) => ({
            attributes: ['code', 'description', 'condition', 'ac-
tion'],
            include: {
                model: Event,
                attributes: ['code', 'description', 'catego-
ryEvent'],
                duplicating: false,
                required: true,
                include: [
                    {
                        model: Component,
                        attributes: ['code', 'description'],
                        duplicating: false,
                        required: true,
                        include: [
                            {
```

```
                                model: System,
                                where: { code: codeSystem },
                                attributes: [],
                                duplicating: false,
                                required: true,
                            },
                        ],
                        through: { attributes: [] },
                    },
                ],
                through: { as: 'association', attributes:
['typeBind'] },
            },
        });

        return this.ruleRep.findAll(findModelBySystem(code));
    }

    async create(dto: CreateRuleDto) {
        const ruleCode = await this.counterService.getNew-
Code(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 правило 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 найде-
но');

        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 привязано 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.2 – Файл component.service.ts

```
@Injectable()
export class ComponentService {
    constructor(
        @InjectModel(Component) private componentRepository:
typeof Component,
        @InjectModel(RoleComponent) private roleRepository: typeof
RoleComponent,
        @InjectModel(LvlComponent) private lvlRepository: typeof
LvlComponent,
        @InjectModel(System) private sysRepository: typeof System,
        private readonly counterService: CounterService,
    ) {}

    readonly findModelDef = {
        attributes: ['code', 'description'],
        include: [
            { model: LvlComponent, as: 'lvl', attributes: ['name',
'description'] },
            { model: System, as: 'system', attributes: ['code',
'name'] },
            {
                model: RoleComponent,
                as: 'role',
                attributes: ['name', 'description'],
                through: { attributes: [] },
            },
            {
                model: Event,
                as: 'event',
                attributes: ['code', 'description', 'catego-
ryEvent'],
                through: { attributes: [] },
                include: [{ model: ContextParam, attributes:
['code', 'name', 'description', 'dataType'] }], // TODO проверить
включаемость
            },
        ],
    };

    findAll(): Promise<Component[]> {
        return this.componentRepository.findAll(this.findModel-
Def);
    }

    findOne(code: string) {
        return this.componentRepository.findOne({ where: { code },
...this.findModelDef });
    }
```

```
    async create(dto: CreateComponentDto) {
        const componentCode = await this.counterService.getNew-
Code(CounterName.COMPONENT);
        const created = await this.componentRepository.create({
...dto, code: componentCode });
        return { code: created.code, name: created.description };
    }

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

    async remove(dto: CodeConfirmDto) {
        const candidate = await this.componentRepository.findOne({
            where: { code: dto.code },
            include: { all: true },
        });
        if (!candidate) throw new NotFoundException('Компонент не
найден');
        if (candidate.event && candidate.event.length > 0 &&
!dto.confirm) {
            throw new BadRequestException('К компоненту присоеди-
нены события');
        }
        // TODO проверка удаления компонента если присоединены
эвенты
        // Удаляться ли автоматом связи или эвент?
        await candidate.destroy();
        return 'Компонент удален';
    }

    async addRole(dto: AddRoleDto) {
        const component = await this.componentRepository.findOne({
            where: { code: dto.code },
            include: [{ model: RoleComponent }],
        });
        if (!component) throw new NotFoundException('Компонент не
найден');
        if (component.role.find((el) => el.name === dto.name)) {
            throw new BadRequestException('Компонент уже имеет
указанную роль');
        }
        const role = await this.roleRepository.findOne({ where: {
name: dto.name } });
```

```
        if (!role) throw new NotFoundException('Роль не найден');
        await component.$add('role', role.id);
        return this.findOne(dto.code);
    }

    async removeRole(dto: RemoveRoleDto) {
        const component = await this.componentRepository.findOne({
            where: { code: dto.code },
            include: [{ model: RoleComponent }],
        });
        if (!component) throw new NotFoundException('Компонент не
найден');
        const role = await this.roleRepository.findOne({ where: {
name: dto.name } });
        if (!role) throw new NotFoundException('Роль не найден');
        if (!component.role.find((el) => el.name === dto.name)) {
            throw new BadRequestException('Компонент не имеет ука-
занную роль');
        }
        await component.$remove('role', role.id);
        return this.findOne(dto.code);
    }

    async addLvl(dto: AddLvlDto) {
        const component = await this.componentRepository.findOne({
            where: { code: dto.code },
            include: [{ model: LvlComponent }],
        });
        if (!component) throw new NotFoundException('Компонент не
найден');
        if (component.lvl && component.lvl.name === dto.name) {
            throw new BadRequestException('Компонент уже имеет
указанный уровень');
        }
        const lvl = await this.lvlRepository.findOne({ where: {
name: dto.name } });
        if (!lvl) throw new NotFoundException('Уровень не
найден');
        component.lvlId = lvl.id;
        await component.save();
        return this.findOne(dto.code);
    }

    async removeLvl(dto: RemoveLvlDto) {
        const component = await this.componentRepository.findOne({
            where: { code: dto.code },
            include: [{ model: RoleComponent }],
        });
        if (!component) throw new NotFoundException('Компонент не
найден');
```

```
        const lvl = await this.lvlRepository.findOne({ where: {
name: dto.name } });
        if (!lvl) throw new NotFoundException('Уровень не
найден');
        if (!component.lvl || component.lvl.name != dto.name) {
            throw new BadRequestException('Компонент не имеет ука-
занный уровень');
        }
        component.lvlId = null;
        await component.save();
        return this.findOne(dto.code);
    }

    async addSystem(dto: AddSystemDto) {
        const component = await this.componentRepository.findOne({
            where: { code: dto.codeComponent },
            include: [{ model: System }],
        });
        if (!component) throw new NotFoundException('Компонент не
найден');
        if (component.system && component.system.code ===
dto.codeSystem) {
            throw new BadRequestException('Компонент уже содер-
жится в указанной системе');
        }
        const sys = await this.sysRepository.findOne({
            where: { code: dto.codeSystem },
            include: { all: true },
        });
        if (sys.children && sys.children.length > 0)
            throw new BadRequestException('Система не является ко-
нечной');
        if (!sys) throw new NotFoundException('Система не найде-
на');
        component.systemId = sys.id;
        await component.save();
        return this.findOne(dto.codeComponent);
    }

    async removeSystem(dto: RemoveSystemDto) {
        const component = await this.componentRepository.findOne({
            where: { code: dto.codeComponent },
            include: [{ model: System }],
        });
        if (!component) throw new NotFoundException('Компонент не
найден');
        const sys = await this.sysRepository.findOne({ where: {
code: dto.codeSystem } });
        if (!sys) throw new NotFoundException('Система не найде-
на');
```

```
        if (!component.system || component.system.code !=
dto.codeSystem) {
            throw new BadRequestException('Компонент не содержится
в указанной системе');
        }
        component.systemId = null;
        await component.save();
        return this.findOne(dto.codeComponent);
    }
}
```

Листинг Б.3 – 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: Component,
                attributes: ['code', 'description'],
                include: [
                    {
                        model: System,
                        attributes: ['code', 'name'],
                    },
                ],
                through: { attributes: [] },
            },
            {
                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', 'cate-
goryEvent'],
                        through: { attributes: [] },
                    },
                ],
            },
            {
                model: ComplexEvent,
                as: 'inboundComplexEvent',
                attributes: ['templateEvent'],
                through: { attributes: [] },
                include: [
                    {
                        model: Event,
                        as: 'event',
                        attributes: ['code', 'description', 'cate-
goryEvent'],
                    },
                ],
            },
            {
                model: AggregationEvent,
                attributes: ['aggregationQuery', 'timestampBegin',
'timestampEnd'],
                include: [
                    {
                        model: AtomicEvent,
                        as: 'atomicEvent',
                        include: [
                            {
                                model: Component,
                                as: 'component',
                                attributes: ['code', 'descrip-
tion'],
                            },
                            {
                                model: Event,
                                attributes: ['code', 'descrip-
tion', 'categoryEvent'],
                            },
                        ],
                        attributes: ['codePair'],
                        through: { attributes: [] },
                    },
                ],
            },
        ],
```

```
    };

    private clearOut(event: Event) {
        let clearEvent = event.toJSON() as any;
        if (!clearEvent.complexEvent) delete clearEvent.com-
plexEvent;
        if (!clearEvent.aggregationEvent) delete clearEvent.aggre-
gationEvent;
        if (clearEvent.categoryEvent === CategoryEvent.AGGREGA-
TION) {
            clearEvent = { ...clearEvent, ...clearEvent.aggrega-
tionEvent };
            delete clearEvent.aggregationEvent;
        }
        if (clearEvent.categoryEvent === CategoryEvent.COMPLEX) {
            clearEvent = { ...clearEvent, ...clearEvent.com-
plexEvent };
            delete clearEvent.complexEvent;
        }
        return clearEvent;
    }

    async findAll() {
        const events = await this.eventReposi-
tory.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,
        });
        if (!event) throw new NotFoundException();
        return this.clearOut(event);
    }

    findAllEventsSystem(code: string) {
        const findModelBySystem = (codeSystem: string) => ({
            attributes: ['code', 'description', 'categoryEvent'],
            duplicating: false,
            required: true,
            include: [
                {
                    model: Component,
                    attributes: ['code', 'description'],
                    duplicating: false,
                    required: true,
                    include: [
```

```
                        {
                            model: System,
                            where: { code: codeSystem },
                            attributes: [],
                            duplicating: false,
                            required: true,
                        },
                    ],
                    through: { attributes: [] },
                },
                {
                    model: ContextParam,
                    attributes: ['code', 'name', 'description',
'dataType'],
                },
            ],
        });

        return this.eventRepository.findAll(findModelBySys-
tem(code));
    }

    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.4 – Файл system.service.ts

```
@Injectable()
export class SystemService {
    constructor(
        @InjectModel(System) private systemRepository: typeof Sys-
tem,
        private readonly counterService: CounterService,
    ) {}

    readonly findModelDef = {
        attributes: ['code', 'name'],
```

```
        include: [
            { model: System, as: 'parent', attributes: ['code',
'name'] },
            { model: System, as: 'children', attributes: ['code',
'name'] },
        ],
    };

    readonly findModelForRec = {
        attributes: ['id', 'code', 'name', 'parentId'],
        raw: true,
    };

    async findAll(attached?: boolean) {
        if (attached) {
            const systems = await this.systemReposi-
tory.findAll(this.findModelForRec);
            const systemsWithChildren = [];
            for await (const el of systems) {
                systemsWithChildren.push(await this.addChildren-
Recursive(el));
            }
            return systemsWithChildren;
        } else {
            return this.systemRepository.findAll(this.findModel-
Def);
        }
    }

    async findOne(code: string, attached?: boolean) {
        if (attached) {
            const system = await this.systemRepository.findOne({
where: { code }, ...this.findModelForRec });
            return await this.addChildrenRecursive(system);
        }
        return this.systemRepository.findOne({ where: { code },
...this.findModelDef });
    }

    async create(createSystemDto: CreateSystemDto) {
        const systemCode = await this.counterService.getNew-
Code(CounterName.SYSTEM);
        const created = await this.systemRepository.create({
...createSystemDto, code: systemCode });
        return { code: created.code, name: created.name };
    }

    async update(dto: UpdateSystemDto) {
        const system = await this.systemRepository.findOne({
where: { code: dto.code } });
```

```
        if (!system) throw new NotFoundException('Система не
найдена');
        await system.update(dto);
        return this.systemRepository.findOne({ where: { code:
dto.code }, ...this.findModelDef });
    }

    async remove(dto: DeleteSystemDto) {
        const candidate = await this.systemRepository.findOne({
            where: { code: dto.code },
            include: { all: true },
        });
        if (!candidate) throw new NotFoundException('Система не
найдена');
        if (candidate.children && candidate.children.length > 0) {
            if (!dto.confirmAttach)
                throw new BadRequestException('На удаляемую си-
стему ссылаются другие системы');

            for await (const ch of candidate.children) {
                ch.parentId = null;
                await ch.save();
            }
        }
        if (candidate.component && candidate.component.length > 0)
{
            if (!dto.confirmComponent) throw new BadRequestExcep-
tion('Система содержит компоненты');
            for await (const comp of candidate.component) {
                comp.systemId = null;
                await comp.save();
            }
        }
        await candidate.destroy();
        return 'Система удалена';
    }

    /**
     * Рекурсивный возврат всех потомков
     * @param candidate объект в RAW с id и parentId
     * @returns объект системы, а не сущность
     */
    private async addChildrenRecursive(candidate: System) {
        const children = await this.systemRepository.findAll({
            where: { parentId: candidate.id },
            attributes: ['id', 'code', 'name', 'parentId'],
            raw: true,
        });

        if (children.length > 0) {
            const promises = [];
```

```
            children.forEach((el) => {
                el['parentCode'] = candidate.code;
                promises.push(this.addChildrenRecursive(el));
            });
            candidate['children'] = await Promise.all(promises);
        } else candidate['children'] = [];
        delete candidate['id'];
        delete candidate['parentId'];
        return candidate;
    }

    async attach(dto: AttachSystemDto) {
        const systems = await this.findAttachingSystems(dto.code-
Parent, dto.codeAttach);
        if (systems.parent.component && systems.parent.compo-
nent.length > 0) {
            throw new BadRequestException(
                'Родительская система уже имеет компоненты, невоз-
можно присоединить',
            );
        }
        if (systems.attach.parent) {
            throw new BadRequestException(
                `Присоединяемая система уже присоединена к системе
${systems.attach.parent.code}`,
            );
        }
        systems.attach.parentId = systems.parent.id;
        await systems.attach.save();
        return this.systemRepository.findOne({ where: { code:
dto.codeAttach }, ...this.findModelDef });
    }

    async detach(dto: DetachSystemDto) {
        const systems = await this.findAttachingSystems(dto.code-
Parent, dto.codeAttach);
        if (!systems.attach.parentId)
            throw new BadRequestException('Отсоединяемая система
не имеет родителя');
        if (systems.attach.parentId != systems.parent.id) {
            throw new BadRequestException(`Отсоединяемая система
не имеет родителя ${dto.codeParent}`);
        }
        systems.attach.parentId = null;
        await systems.attach.save();
        return this.systemRepository.findOne({ where: { code:
dto.codeAttach }, ...this.findModelDef });
    }

    private async findAttachingSystems(codeParent: string,
codeAttach: string) {
```

```
        const parent = await this.systemRepository.findOne({
            where: { code: codeParent },
            include: { all: true },
        });
        if (!parent) throw new NotFoundException(`Родительская си-
стема ${codeParent} не найдена`);
        const attach = await this.systemRepository.findOne({
            where: { code: codeAttach },
            include: { all: true },
        });
        if (!attach) throw new NotFoundException(`Присоединяемая
система ${codeParent} не найдена`);
        return { parent, attach };
    }
}
```

Листинг Б.5 – Файл user.service.ts

```
@Injectable()

export class UserService {

    constructor(

        @InjectModel(User) private userRepository: typeof User,

        @InjectModel(Role) private roleRepository: typeof Role,

    ) {}



    readonly findModel = {

        attributes: ['code', 'email'],

        include: [{ model: Role, attributes: ['name', 'description'],
through: { attributes: [] } }],

    };



    findAll() {

        return this.userRepository.findAll(this.findModel);

    }



    findOne(code: string) {
```

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

    async update(dto: UpdateUserDto) {
        const user = await this.userRepository.findOne({ where: { code: dto.code } });
        if (!user) throw new NotFoundException('Пользователь не найден');
        await user.update(dto);
        return this.userRepository.findOne({ where: { code: dto.code }, ...this.findModel });
    }

    async remove(code: string) {
        if (code === '00001') throw new BadRequestException('Изменять данные параметры запрещено');
        const destroyCount = await this.userRepository.destroy({ where: { code } });
        if (destroyCount === 0) throw new NotFoundException('Пользователь не найден');
        return 'Пользователь удален';
    }

    async addRole(dto: AddRoleDto) {
        const user = await this.userRepository.findOne({
            where: { code: dto.code },
            include: [{ model: Role }],
        });
```

```
        if (!user) throw new NotFoundException('Пользователь не найден');

        if (user.role.find((el) => el.name === dto.name)) {
            throw new BadRequestException('Пользователю уже принадлежит эта роль');
        }

        const role = await this.roleRepository.findOne({ where: { name: dto.name } });
        if (!role) throw new NotFoundException('Роль не найден');

        await user.$add('role', role.id);
        return await this.userRepository.findOne({ where: { code: dto.code }, ...this.findModel });
    }

    async removeRole(dto: RemoveRoleDto) {
        if (dto.code === '00001' && dto.name === RoleName.ADMIN) {
            throw new BadRequestException('Изменять данные параметры запрещено');
        }

        const user = await this.userRepository.findOne({
            where: { code: dto.code },
            include: [{ model: Role }],
        });
        if (!user) throw new NotFoundException('Пользователь не найден');
```

```
        const role = await this.roleRepository.findOne({ where: { name: dto.name } });
        if (!role) throw new NotFoundException('Роль не найден');

        if (!user.role.find((el) => el.name === dto.name)) {
            throw new BadRequestException('Пользователю не принадлежит эта роль');
        }

        await user.$remove('role', role.id);
        return this.userRepository.findOne({ where: { code: dto.code }, ...this.findModel });
    }
}
```

Приложение В

(обязательное)

Листинг вспомогательных сервисов

Листинг В.1 – 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 <${pro-
cess.env.SMTP_USER}>`,
            },
        );

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

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

        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], acti-
vationLink),
        });

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

    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 В.2 – 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.con-
straints).join('; ')}`,
                    error: [...Object.values(err.constraints)],
                };
            });
            throw new BadRequestException({
                status: HttpStatus.BAD_REQUEST,
                message: messages,
            });
        }

        return obj;
    }
}
```

Листинг В.3 – 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.4 – 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.5 – 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.er-
rorMessage);
                }
            }),
        );
    }
}
```

Листинг В.6 – Файл timeout.interceptor.ts

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

Листинг В.7 – 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.8 – Файл Файл role-auth.guard.ts

```
@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.getAllAnd-
Override<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.in-
cludes(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);
        }
    }
}
```

Приложение Г

(обязательное)

Листинг средств авторизации

Листинг Г.1 – 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 по-
чты уже зарегистрирован`,
            );
        }

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

        const codeUser = await this.counterService.getNew-
Code(CounterName.USER);

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

        const tokens = await this.jwtGenerateServise.generate-
Tokens(
            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.generate-
Tokens(
            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(re-
freshToken);
        const tokenFromBD = await this.tokenRepository.findOne({
            where: { refreshToken: refreshToken },
        });
        if (!userTokenData || !tokenFromBD) {
            throw new UnauthorizedException();
        }

        const user = await this.userRepository.findByPk(token-
FromBD.userId, {
            include: { all: true },
        });
        const tokens = await this.jwtGenerateServise.generate-
Tokens(
            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) => pay-
loadJWT.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 =
            'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstu-
vwxyz0123456789';
        for (let i = 0; i < 5; i++)
            randomPassord += passwordTemplate.charAt(
                Math.floor(Math.random() * passwordTem-
plate.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 про-
шлый 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 Г.2 – 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_AC-
CESS_SECRET, {
            expiresIn: '30m',
        });
        const refreshToken = jwt.sign(payload, process.env.JWT_RE-
FRESH_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_AC-
CESS_SECRET);
            return userData;
        } catch (err) {
            return null;
        }
    }

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

Приложение Д

(рекомендуемое)

Листинг модульных и интеграционных тестов

Листинг Д.1 – Исходный код модульного 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.createTestingMod-
ule({
            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()).toE-
qual([roleStub]);
    });

    it('should make a new role', async () => {
        expect(await roleController.create({ value:
roleStub.value, description: roleStub.description })).toE-
qual(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 Д.2 – Исходный код модульного 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(BadRe-
questException);
        expect(service.remove(2)).rejects.toBeInstanceOf(BadRe-
questException);
        expect(service.remove(3))
        expect(destroySpy).toBeCalledWith({where: {id: 3}})
    })
});
```

Листинг Д.3 – Исходный код интеграционного тестирования RoleService и RoleController

```
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;
    }
}
```