Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

«Программно-алгоритмический комплекс с многоцелевой и масштабируемой архитектурой для совместной работы и управления проектами»

Студент	ИУ7-86Б		К. М. Шматко
	(группа)	(подпись)	(инициалы, фамилия)
Руководитель ВКР			А. М. Никульшин
		(подпись)	(инициалы, фамилия)
Нормоконтролер			
		(подпись)	(инициалы, фамилия)

2025 год

РЕФЕРАТ

Расчетно-пояснительная записка 92 с., 18 рис., 2 табл., 20 источн., 8 прил.

В данной работе рассматривается проектирование и разработка программно-алгоритмического комплекса с многоцелевой и масштабируемой архитектурой, предназначенного для организации совместной работы пользователей и управления проектами.

Проведен анализ предметной области, включающий обзор и сравнение современных реляционных и нереляционных систем хранения данных. Рассмотрены характеристики распределенных систем хранения и методы организации сетевого многопользовательского взаимодействия.

Разработаны основные положения и структура предлагаемого программно-алгоритмического комплекса. Описана его модульная архитектура, ключевые компоненты и принципы их взаимодействия. Определены основные структуры данных, используемые для хранения пользовательской информации, проектных данных и создаваемого контента.

Обоснован выбор средств программной реализации. Выполнено тестирование разработанного комплекса для проверки корректности его работы. Описаны основные сценарии взаимодействия пользователя с программным обеспечением.

Проведено исследование потенциальной применимости разработанного комплекса в различных сценариях управления проектами и командной работы. Исследованы ключевые характеристики системы.

Ключевые слова: совместная работа, управление проектами, программный комплекс, масштабируемая архитектура, распределенное хранение данных, MongoDB, PostgreSQL, WebSockets, многопользовательское взаимодействие, гибридная база данных.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ

В настоящей расчетно-пояснительной записке применяют следующие термины с соответствующими определениями.

Системы хранения данных — совокупность языковых и программных средств, предназначенных для создания, ведения и совместного использования БД многими пользователями.

Распределенная система хранения данных — система хранения, в которой данные физически расположены на нескольких узлах (серверах), связанных сетью, но представляются пользователю или приложению как единое целое.

Масштабируемость — способность системы, сети или процесса увеличивать свою производительность и пропускную способность пропорционально увеличению нагрузки (количества пользователей, объема данных, транзакций) путем добавления ресурсов (аппаратных или программных).

PostgreSQL — объектно-реляционная система управления базами данных, соответствующая стандарту SQL, расширяемая и поддерживающая сложные запросы и транзакции (ACID).

MongoDB — документоориентированная NoSQL система управления базами данных, использующая для хранения данных JSON-подобные документы (BSON).

WebSocket — сетевой протокол, обеспечивающий постоянное полнодуплексное (двустороннее) соединение между клиентом и сервером поверх одного TCP-соединения, предназначенный для интерактивного обмена данными в реальном времени.

Mind map (интеллект-карта, диаграмма связей) — метод структуризации и визуализации концепций с использованием графической записи в виде диаграммы.

Endpoint (конечная точка) — интерфейс или URL-адрес в REST API, предоставляющий доступ к определенному ресурсу или функциональности системы через HTTP-методы (GET, POST, PUT, DELETE).

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей расчетно-пояснительной записке применяют следующие сокращения и обозначения.

БД — База Данных

СУБД — Система Управления Базами Данных

ACID — Atomicity, Consistency, Isolation, Durability

API — Application Programming Interface

GraphQL — Graph Query Language

HDFS — Hadoop Distributed File System

HTTP — HyperText Transfer Protocol

HTTPS — HyperText Transfer Protocol Secure

JSON — JavaScript Object Notation

NoSQL — Not only SQL

P2P — Peer-to-Peer

REST — Representational State Transfer

S3 — Simple Storage Service

SQL — Structured Query Language

TCP — Transmission Control Protocol

UDP — User Datagram Protocol

WebRTC — Web Real-Time Communication

XML — Extensible Markup Language

ФИО — Фамилия, Имя, Отчество

ВВЕДЕНИЕ

В условиях динамично развивающейся цифровой экономики и роста популярности удаленных и гибридных форматов работы, организация совместной деятельности и управления проектами становится критически важной для успеха команд и организаций. Существующие инструменты часто либо специализированы на узком круге задач, либо представляют собой разрозненные сервисы, что усложняет интеграцию рабочих процессов и управление информацией. Возникает потребность в универсальных платформах, обеспечивающих комплексную поддержку командной работы. Более того, данные, генерируемые пользователями в рамках работы с подобным комплексом (текстовые записки, созданные схемы, файлы проектов, история коммуникаций в чатах и др.), могут послужить основой для формирования уникальных наборов данных (датасетов). Эти датасеты в перспективе могут быть использованы для проведения исследований и выполнения выпускных квалификационных работ студентами последующих курсов.

Целью данной работы является разработка программно-алгоритмического комплекса с многоцелевой и масштабируемой архитектурой для совместной работы и управления проектами.

Для достижения поставленной цели необходимо решить следующие задачи:

1) Провести анализ предметной области, обзор и сравнение существующих систем хранения данных и методов сетевого многопользовательского взаимодействия, а также формализовать требования к разрабатываемому комплексу.

2) Разработать архитектуру программно-алгоритмического комплекса, структуру приложения, описание его компонентов, их взаимодействия и ключевых структур данных.

3) Обосновать выбор средств программной реализации, разработать программное обеспечение, реализующее основные функции комплекса, и выполнить его тестирование для проверки корректности работы.

4) Провести исследование применимости и оценить ключевые характеристики разработанного программно-алгоритмического комплекса.

1 Аналитический раздел

1.1 Обзор современных систем хранения данных

1.1.1 Определение и классификация систем управления базами данных

Системы хранения данных, рассматриваемые в данном контексте, представляют собой совокупность языковых и программных средств, предназначенных для создания, ведения и совместного использования БД многими пользователями [1]. Эти системы, известные также как системы управления базами данных (СУБД), обеспечивают структурированное хранение данных и поддерживают операции над ними, такие как манипуляции, индексация, выполнение сложных запросов и поддержание целостности данных.

Существуют три основные группы моделей баз данных, различающихся структурой организации данных: дореляционные, реляционные и постреляционные базы данных.

Дореляционные базы данных

Основной особенностью дореляционных моделей является то, что взаимосвязи между данными управляются явно, и структура данных часто определяется физическим расположением данных, а не логическими ассоциациями.

Дореляционные модели включают в себя:

1) иерархическую модель, где структура данных представлена в виде дерева, информация в котором разбита на сегменты, имеющие строгое родитель–дочернее отношение;

2) сетевую модель, позволяющую создавать множество связей между узлами, что делает возможным иметь несколько родителей у одного узла;

3) инвертированные списки, строящиеся на основе терминов (слов или фраз), которые встречаются в документах или текстовых полях базы данных.

Реляционные базы данных

Основной особенностью реляционных баз данных является использование структурированных таблиц, где данные организованы в столбцы и строки [2]. Каждая таблица представляет собой коллекцию кортежей, общая структура которых описана в схеме таблицы. Таблицы могут быть связаны друг с другом через идентификаторы и ключи, что позволяет структурировать и интегрировать данные из различных источников.

Реляционные базы данных обладают набором свойств ACID:

1) атомарность – транзакция будет выполнена как одно целое;

2) согласованность – транзакция переводит базу данных из одного согласованного состояния в другое согласованное состояние;

3) изолированность – транзакция исполняется изолированно от других транзакций;

4) устойчивость – результаты успешно завершенной транзакции будут устойчивы к будущим сбоям.

Постреляционные базы данных

Основной особенностью постреляционных баз данных является их гибкость в обработке разноплановых и менее структурированных данных, а также поддержка более производительных способов распределения данных и масштабирования [2].

Виды постреляционных моделей:

1) ключ-значение хранилища;

2) графы;

3) документоориентированные базы данных.

1.1.2 Неоднородные данные и их особенности

Неоднородные данные представляют собой различные типы данных, которые могут отличаться по структуре, формату и источнику. Среди основных характеристик неоднородных данных можно выделить следующие:

1) форматы – неоднородные данные включают разнообразные форматы, такие как текст, изображения, аудио, видео, JSON, XML и таблицы SQL;

2) источники – неоднородные данные поступают из различных источников, включая социальные сети, системы транзакций, веб-логи и корпоративные информационные системы;

3) структурированность – неоднородные данные подразделяются на структурированные, полуструктурированные и неструктурированные, в зависимости от степени их организации.

Работа с неоднородными данными требует специального подхода, учитывающего их разнородность и сложность. Рассмотрим основные задачи, связанные с обработкой и управлением такими данными [3].

1) **Интеграция данных.** Объединение данных из различных источников требует нормализации, очистки и преобразования данных для обеспечения их согласованности и целостности.

2) **Хранение данных.** Разнообразие форматов может требовать использования гибридных решений для хранения, включая в себя как реляционные, так и NoSQL системы для обработки различных типов данных.

3) **Обработка и анализ.** Обработка разнообразных типов данных требует применения различных аналитических методов, таких как обработка естественного языка (Natural Language Processing) для текстов и алгоритмы компьютерного зрения для изображений.

4) **Управление метаданными.** Метаданные играют важную роль в интерпретации данных и обеспечении их доступности, предоставляя информацию о происхождении, структуре и содержании данных.

1.2 Анализ предметной области систем распределенного хранения данных

1.2.1 Характеристики современных систем хранения

Современные системы хранения данных являются фундаментальной основой для построения масштабируемых приложений, особенно в условиях растущих объемов информации и разнообразия типов данных. Они должны обеспечивать высокую производительность, надежность и гибкость для удовлетворения потребностей как разработчиков, так и конечных пользователей.

Ключевые характеристики современных систем хранения данных включают [4]:

1) масштабируемость – способность системы увеличивать свои ресурсы (объем хранения, производительность) по мере роста данных и нагрузки без значительной деградации производительности. Масштабируемость может быть вертикальной (увеличение ресурсов на одном сервере) и горизонтальной (добавление новых серверов в кластер);

2) устойчивость к отказам – обеспечение непрерывной работы системы даже при сбоях отдельных компонентов. Это достигается посредством репликации данных и избыточности серверов;

3) производительность – высокая скорость операций чтения и записи данных, низкая задержка при обработке запросов, что критически важно для приложений в реальном времен;

4) гибкость модели данных – поддержка различных моделей данных (реляционная, документоориентированная, графовая и др.) позволяет оптимально хранить и обрабатывать разные типы данных;

5) совместимость – возможность интеграции с другими системами и сервисами, поддержка стандартных протоколов и API;

6) безопасность – механизмы аутентификации, авторизации, шифрования данных и управления доступом для защиты данных от несанкционированного доступа;

7) консистентность данных – обеспечение целостности и согласованности данных во всех узлах распределенной системы, особенно в условиях одновременных операций чтения и записи;

8) скалируемая архитектура – возможность распределения нагрузки и хранения данных в масштабируемой архитектуре, такой как кластер или облако;

9) управление транзакциями – поддержка атомарности, согласованности, изоляции и долговечности (ACID-свойства) для гарантии правильности и надёжности операций с данными;

10) поддержка неоднородных данных – способность хранить и обрабатывать структурированные, полуструктурированные и неструктурированные данные.

Современные системы хранения данных, такие как NoSQL-базы (MongoDB, Cassandra) и реляционные базы данных (PostgreSQL, MySQL), обладают различными комбинациями этих характеристик, что позволяет выбирать оптимальную систему для конкретных задач [5—8].

1.2.2 Влияние типов данных на выбор системы хранения

Типы данных, с которыми предстоит работать системе, играют ключевую роль при выборе подходящей системы хранения. Каждая система оптимизирована для определённых видов данных и операций над ними [4].

Структурированные данные

Структурированные данные имеют строгую и фиксированную структуру. Они организованы в виде таблиц, где каждая запись соответствует определённому набору полей с предопределёнными типами данных. Примеры включают транзакционные данные, инвентаризационные списки, финансовые записи.

Для работы со структурированными данными оптимальны реляционные СУБД, такие как PostgreSQL или MySQL. Они обеспечивают:

1) Строгую схему данных и целостность при использовании ограничений и связей между таблицами.

2) Возможность выполнения сложных запросов с использованием языка SQL.

3) Поддержку транзакций с ACID-свойствами.

Полуструктурированные данные

Полуструктурированные данные не обладают строгой схемой, но имеют внутреннюю организацию и метаданные. Примеры включают XML, JSON-файлы, логи приложений.

Документоориентированные СУБД, такие как MongoDB или Cassandra, подходят для работы с полуструктурированными данными, предоставляя:

1) Гибкую схему данных, позволяющую хранить документы с различной структурой.

2) Высокую производительность при операциях с большими объёмами документов.

3) Масштабируемость и отказоустойчивость в распределённых средах.

Неструктурированные данные

Неструктурированные данные не имеют предопределённой модели и включают текстовые документы, изображения, аудио и видео файлы. Их обработка требует специального подхода.

Системы хранения данных для неструктурированных данных, такие как Hadoop HDFS или Amazon S3, позволяют:

1) Хранить большие объёмы данных в распределённой среде.

2) Обрабатывать данные с использованием параллельных вычислений (например, MapReduce).

3) Использовать специализированные инструменты для анализа и поиска в данных.

Графовые данные

Графовые данные состоят из узлов и связей между ними, что позволяет моделировать сложные взаимоотношения. Применяются в социальных сетях, рекомендационных системах, анализе связей.

Графовые СУБД, такие как Neo4j или OrientDB, предлагают:

1) Хранение и обработку графовых структур данных.

2) Быстрый доступ к связанным данным без необходимости сложных JOIN-операций.

3) Специфичные языки запросов для работы с графами (например, Cypher).

Выбор системы хранения

Таким образом, при выборе системы хранения необходимо учитывать:

1) Типы данных и их структуру.

2) Объём данных и прогнозируемый рост.

3) Требования к производительности и масштабируемости.

4) Необходимость в транзакционной целостности и консистентности данных.

5) Возможность интеграции с существующими системами и инструментами.

Понимание типов данных и их особенностей позволяет выбрать оптимальную систему хранения для конкретных задач.

1.2.3 Особенности хранения текстов и структурированных данных на примере структуры *Пользователь*

В системе совместной работы над проектами структура *Пользователь* является одной из ключевых и включает в себя как структурированные данные, так и связи с другими сущностями системы.

Структура *Пользователь* может включать следующие поля:

1) **ФИО** – строковые поля для хранения имени, фамилии и отчества пользователя.

2) **Логин** – уникальный идентификатор пользователя в системе.

3) **Пароль** – хеш пароля для аутентификации пользователя.

4) **Фото** – ссылка на изображение профиля пользователя.

5) **Связи с проектами** – информация о проектах, к которым пользователь имеет доступ, а также о проектах, созданных пользователем.

Кроме того, система хранит неструктурированные данные, которые включают такие элементы как тексты и другие мультимедийные данные, представленные в формате JSON.

Хранение структурированных данных

Реляционная база данных является оптимальным выбором для хранения структурированных данных пользователей по следующим причинам:

1) **Строгая схема данных**. Реляционные СУБД позволяют задать фиксированную структуру данных, обеспечивая строгую организацию информации о пользователях, что критично для таких атрибутов, как идентификатор, логин, и хеш пароля.

2) **Связи между таблицами**. Возможность моделирования сложных связей между таблицами позволяет управлять отношениями между пользователями и другими сущностями, такими как проекты.

3) **Безопасность и транзакционность**. Реляционные базы данных поддерживают ACID-транзакции, что гарантирует надёжность и консистентность данных.

Пример создания таблицы пользователей показывает, как можно определить строгую схему для хранения ключевой информации о пользователях 1.1:

Листинг 1.1 – Создание таблицы пользователей

```
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    full_name VARCHAR(255) NOT NULL,
    login VARCHAR(100) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    photo_url VARCHAR(255),
);
```

Для управления связями пользователей с проектами используется дополнительная таблица, что позволяет организовать реляцию «многие ко многим» и управлять ролями и правами доступа 1.2:

Листинг 1.2 – Создание таблицы связей пользователей с проектами

```
CREATE TABLE user_projects (
    user_id INTEGER REFERENCES users(user_id),
    project_id INTEGER REFERENCES projects(project_id),
    role VARCHAR(50),
    PRIMARY KEY (user_id, project_id)
);
```

Хранение текстов в формате JSON

Для хранения текстов, представленных в формате JSON, выбрана нереляционная база данных по следующим причинам:

1) **Отсутствие фиксированной схемы**. Нереляционные базы данных позволяют работать с динамически изменяющимися данными, такими как текстовые блоки, и их метаданные. Это позволяет адаптировать структуру хранения в соответствии с изменяющимися требованиями без необходимости изменения схемы.

2) **Гибкость и масштабируемость**. Такие системы плавно подстраиваются под изменяющиеся объёмы данных и требования, позволяя легко хранить и манипулировать информацией в формате JSON.

3) **Высокая производительность при работе с документами**. Оптимизация под работу с JSON-документами обеспечивает быструю обработку и выборку данных, что важно при хранении и доступе к текстовым описаниям и метаданным.

Пример документа в нереляционной базе данных представлен в листинге 1.3:

Листинг 1.3 – Пример документа в нереляционной базе данных

```
{
    "_id": ObjectId("..."),
    "user_id": 123,
    "content": "Текстовый контент...",
    "metadata": {
        "tags": ["example", "text"],
        "created_at": ISODate("2024-12-01T12:00:00Z")
    },
    "additional_fields": {
        "field1": "value1",
        "field2": "value2"
    }
}
```

Поле `user_id` связывает текст с конкретным пользователем из реляционной базы данных, что позволяет осуществлять интеграцию между реляционной и нереляционной системами хранения на уровне приложения.

Хранение схем объектов для расширяемости системы

Чтобы обеспечить возможность расширения системы и добавления новых типов объектов без необходимости изменения существующего кода или структуры базы данных, необходимо хранить схемы объектов в базе данных. Это касается таких объектов, как тексты, изображения, графики и другие пользовательские сущности.

Нереляционная база данных, благодаря своей гибкой схеме и документноориентированной структуре, подходит для хранения как данных, так и соответствующих им схем. Схемы объектов могут быть сохранены в специальной коллекции, например, `objectSchemas`, где каждый документ описывает структуру определённого типа объекта.

Пример документа-схемы показан в листинге 1.4:

Листинг 1.4 – Пример документа-схемы

```
{
    "_id": ObjectId("..."),
    "object_type": "text",
    "schema": {
        "fields": [
            "title": "string",
            "required": "boolean",
            "type": "string"
        ],
        "properties": "array",
        "handlers": "array"
    },
    "created_at": ISODate("2024-12-01T12:00:00Z")
}
```

1.3 Сравнение систем распределенного хранения данных

1.3.1 Сравнение систем хранения

Для сравнения MongoDB, PostgreSQL, MySQL и Cassandra использованы следующие критерии:

1) **Горизонтальная масштабируемость**: возможность расширения системы путём добавления новых серверов.

2) **Вертикальная масштабируемость**: возможность улучшения производительности через увеличение мощности существующих серверов.

3) **Изменяемость схемы данных**: простота адаптации к изменяющейся структуре данных.

4) **Поддержка различных форматов данных**: работа с JSON и другими современными форматами.

5) **Консистенция данных**: гарантии целостности данных.

6) **Поддержка транзакций**: наличие и применение транзакций с ACID-свойствами.

7) **Поддержка полуструктурированных и неструктурированных данных**: возможность работы с менее структурированными данными.

8) **Поддержка структурированных данных**: работа с классической реляционной моделью.

9) **Активность и поддержка сообщества**: включая доступность документации, форумов и специалистов.

10) **Инструменты и интеграции**: доступность инструментов для разработки и интеграции с другими технологиями.

На основе этих критериев составлена таблица сравнения MongoDB, PostgreSQL, MySQL и Cassandra (таблица 1.1) [9, 10]. При оценке критериев

был применен весовой коэффициент, где «плюс» оценивается в 1 балл, «минус» в 0 баллов, а «плюс-минус» в 0.5 балла.

Таблица 1.1 – Сравнение MongoDB, PostgreSQL, MySQL и Cassandra

Критерий сравнения	**MongoDB**	**PostgreSQL**	**MySQL**	**Cassandra**
Горизонтальная масштабируемость	+	+–	–	+
Вертикальная масштабируемость	–	+	+	–
Изменяемость схемы данных	+	+–	–	+
Поддержка различных форматов данных	+	–	–	+
Консистенция данных	–	+	+	–
Поддержка транзакций	–	+	+	–
Поддержка полуструктурированных и неструктурированных данных	+	–	–	+
Поддержка структурированных данных	–	+	+	–
Активность и поддержка сообщества	+	+	+	+–
Инструменты и интеграции	+	+	+	+–
Сумма баллов	**6**	**7**	**6**	**5**

Из таблицы видно, что MongoDB и PostgreSQL лучше удовлетворяют ключевые критерии по сравнению с Cassandra и MySQL, однако ни одна из баз данных не охватывает все требования полностью. В связи с этим для выполнения всех задач системы необходимо их совместное использование. Это позволит:

1) Использовать **MongoDB** для хранения полуструктурированных и неструктурированных данных, таких как тексты, документы, мультимедиа и схемы объектов.

2) Применять **PostgreSQL** для хранения структурированных данных, требующих строгой консистентности и поддержки транзакций, таких как информация о пользователях, правах доступа, связях между объектами системы.

3) Объединить сильные стороны обеих систем, создавая гибкую, масштабируемую и надежную систему хранения, способную эффективно по времени и памяти обрабатывать неоднородные данные.

1.3.2 Реализация распределённых баз данных на примере MongoDB и PostgreSQL

Распределённые базы данных позволяют хранить и обрабатывать данные на нескольких серверах, обеспечивая масштабируемость, отказоустойчивость и высокую доступность системы.

MongoDB

MongoDB изначально спроектирована как распределённая документно-ориентированная база данных. Её основные механизмы реализации распределённости включают:

1) **Балансировка нагрузки** – распределение запросов между узлами кластера для обеспечения оптимальной производительности.

2) **Шардинг** – горизонтальное разделение данных по нескольким узлам (шардам). Позволяет масштабировать базу данных горизонтально, добавляя новые серверы для обработки увеличивающихся объёмов данных и нагрузки [11].

3) **Репликация** – процесс копирования и поддержания синхронных копий данных на нескольких узлах. Обеспечивает отказоустойчивость и высокую доступность системы, позволяя автоматически переключаться на резервные узлы в случае сбоя основного [11].

PostgreSQL

PostgreSQL изначально не является распределённой СУБД, однако существуют инструменты и расширения, позволяющие реализовать распределённость:

1) **Репликация**. PostgreSQL поддерживает потоковую репликацию на уровне основного и репликантных серверов, что обеспечивает отказоустойчивость и балансировку нагрузки на чтение [12].

2) **Логическая репликация**. Позволяет реплицировать отдельные таблицы и данные, что обеспечивает большую гибкость при настройке репликации [12].

3) **Расширения для горизонтального масштабирования**:

1) **Citus** —расширение для PostgreSQL, которое превращает его в распределённую систему путем горизонтального масштабирования таблиц по узлам кластера [13]. Citus позволяет автоматически распределять данные по узлам и параллельно обрабатывать запросы, увеличивая производительность и масштабируемость.

2) **Postgres-XL** – масштабируемая распределённая база данных на основе PostgreSQL, обеспечивающая параллельную обработку запросов и распределение данных [14]. Предоставляет как масштабирование на чтение, так и на запись, поддерживает транзакции и обеспечивает согласованность данных в кластере.

3) **pg_pool и pg_bouncer** – инструменты для управления пулом подключений и балансировки нагрузки, которые помогают улучшить производительность и масштабируемость системы. Позволяют распределять запросы между основным сервером и репликами для оптимального использования ресурсов [15, 16].

Эти инструменты позволяют масштабировать PostgreSQL **горизонтально**, то есть увеличивать производительность системы путем добавления новых серверов или узлов. Горизонтальное масштабирование позволяет обрабатывать больший объем данных и нагрузку благодаря распределению данных и запросов по нескольким узлам. Это делает PostgreSQL пригодным для использования в распределённых системах.

1.3.3 Особенности сочетания MongoDB и PostgreSQL

Совместное использование MongoDB и PostgreSQL позволяет работать с разными типами данных и соблюдать требования к ним.

Основные преимущества совместного использования:

1) **Оптимизация по типу данных**. Каждая СУБД используется для тех типов данных, с которыми она работает наиболее производительно.

2) **Масштабируемость**. Возможность масштабировать компоненты системы независимо друг от друга.

3) **Гибкость разработки**. Ускорение разработки благодаря использованию гибких инструментов для разных задач.

Однако при совместном использовании стоит учитывать следующие сложности:

1) **Интеграция данных**. Требуется обеспечить согласованность данных между двумя СУБД, что может быть реализовано через уровень приложения и чёткое разделение ответственности.

2) **Управление системой**. Необходимость поддержки и администрирования двух различных систем.

3) **Безопасность и авторизация**. Разработка единой системы аутентификации и авторизации для доступа к данным в обеих базах данных.

1.4 Методы сетевого многопользовательского взаимодействия

При создании системы для совместной работы нужно помнить, что пользователи могут одновременно редактировать одни и те же данные (замет-

ки, задачи, схемы), поэтому система должна обеспечивать согласованность данных и актуальность их отображения для всех участников. Рассмотрим ключевые аспекты и методы, обеспечивающие такое взаимодействие.

Архитектуры взаимодействия

Существуют две основные архитектуры для построения сетевого взаимодействия:

1) Централизованная (Клиент-Серверная).

2) Децентрализованная (Peer-to-Peer, P2P).

Централизованная архитектура является наиболее распространенной для веб-приложений. Все клиенты (браузеры пользователей) подключаются к центральному серверу или кластеру серверов. Сервер хранит основные данные, обрабатывает запросы клиентов, управляет логикой приложения и обеспечивает синхронизацию данных между пользователями.

Из преимуществ можно выделить относительную простоту управления состоянием и согласованностью данных, так как сервер является единым источником истины, и простоту реализации контроля доступа и безопасности. Но сервер может стать узким местом по производительности при отсутствии масштабирования. Еще одним недостатком является единая точка отказа.

В децентрализованной архитектуре узлы (пользователи) взаимодействуют напрямую друг с другом, без центрального сервера, или с минимальным его участием, например, для обнаружения узлов. Каждый узел хранит копию данных или ее часть и обменивается изменениями с другими узлами.

К достоинствам можно отнести высокую отказоустойчивость, так как нет единой точки отказа, потенциально лучшую масштабируемость для некоторых задач и меньшую нагрузку на центральную инфраструктуру. Однако в такой системе значительно сложнее обеспечить согласованность данных, разрешать конфликты и гарантировать безопасность.

Для большинства систем совместной работы клиент-серверная архитектура является более предпочтительной из-за упрощения задач синхронизации и управления. Однако элементы P2P могут использоваться для специфических функций, например, WebRTC для видеозвонков в чатах.

Протоколы и подходы к обмену данными

На прикладном уровне для взаимодействия клиента и сервера используются различные протоколы и подходы:

1) **HTTP/S с REST API или GraphQL.** Клиент отправляет запросы (GET, POST, PUT, DELETE и т.д.) на сервер для получения или изменения данных. REST является стандартным архитектурным стилем. GraphQL предлагает более гибкий подход к запросу данных клиентом. Подходит для операций, не требующих немедленного отклика у других пользователей, например, сохранение профиля, создание нового проекта.

2) **WebSockets**. Протокол, обеспечивающий постоянное двунаправленное соединение между клиентом и сервером. Подходит для функций, требующих обмена данными в реальном времени, таких как совместное редактирование документов, мгновенные уведомления, чаты, отображение статуса присутствия пользователей. Сервер может отправлять данные клиенту по своей инициативе, без запроса от клиента, что критично для real-time функциональности.

3) **Протоколы транспортного уровня.** Как правило, для надежной передачи данных в клиент-серверных веб-приложениях используется TCP, который гарантирует доставку пакетов в правильном порядке и контроль целостности, что важно для синхронизации данных. UDP используется реже, в основном там, где допустима потеря пакетов ради скорости, например, в некоторых играх или потоковом видео.

Учитывая необходимость обеспечения функций совместной работы, чата и мгновенных уведомлений в реальном времени, для разрабатываемого комплекса в качестве основного механизма синхронизации состояния между клиентами и сервером выбран протокол WebSockets. Для стандартных операций с данными, не требующих немедленной синхронизации, будет использоваться HTTP/S с применением REST-подхода.

1.5 Формализация задачи создания комплекса

1.5.1 Исходные задачи программно–алгоритмического комплекса

Исходные задачи программно–алгоритмического комплекса включают:

1) **Регистрация и аутентификация пользователей** – предоставление возможности пользователям создавать учетные записи, входить в систему и управлять своими профилями.

2) **Создание и управление проектами** – пользователи могут создавать проекты, настраивать их параметры, а также управлять доступом к ним.

3) **Добавление пользователей в проекты** – возможность приглашать других пользователей в проекты, назначать роли и права доступа.

4) **Совместное редактирование документов** – предоставление инструментов для создания и редактирования различных типов документов внутри проектов:

   1) **Записки** – создание текстовых документов с возможностью форматирования и вставки элементов.

   2) **Схемы** – создание и редактирование диаграмм и схем в режиме онлайн.

   3) **Онлайн-презентации** – создание и демонстрация презентаций внутри проекта.

   4) **Mind map** – создание интеллектуальных карт для структурирования идей.

5) **Дополнительные сервисы проекта**:

   1) **Календарь** – интеграция календаря для планирования мероприятий и событий проекта.

   2) **Трекер задач** – управление задачами и отслеживание их выполнения.

3) **Чат** – реализация коммуникации между участниками проекта в режиме реального времени.

6) **Обеспечение безопасности данных** – защита данных пользователей и проектов от несанкционированного доступа.

7) **Масштабируемость системы** – возможность системы эффективно по времени работать при увеличении количества пользователей и проектов.

8) **Модульная система для добавления новых сущностей и сервисов** – разработка и интеграция модульного подхода, который позволит расширять функциональность системы. Данный подход обеспечивает возможность:

1) **Добавления новых сущностей в базу данных** – гибкое изменение схемы данных для интеграции новых типов данных и сущностей без необходимости модификации основной структуры базы данных.

2) **Создания новых сервисов** – разработка дополнительных инструментов и служб, что позволяет адаптировать систему под изменяющиеся требования пользователей и проектов.

1.5.2 Диаграмма вариантов использования

Диаграмма вариантов использования представляет основные функции системы и взаимодействия пользователей с ней. Опишем основные варианты использования программно-алгоритмического комплекса:

1) Регистрация в системе.
2) Вход в систему (аутентификация).
3) Просмотр и редактирование своего профиля.
4) Создание проекта.
5) Приглашение других пользователей в проект.
6) Создание и редактирование:
    1) Записок.
    2) Схем.
    3) Презентаций.
    4) Mind map.
7) Использование дополнительных сервисов:
    1) Календарь.
    2) Трекер задач.
    3) Чат.
8) Управление правами доступа участников проекта.
9) Просмотр и участие в проектах, в которые он добавлен.

На рисунке 1.1 представлена общая диаграмма вариантов использования сервиса.

Рисунок 1.1 – Диаграмма вариантов использования сервиса

На рисунке 1.2 представлена диаграмма вариантов использования записок.

Рисунок 1.2 – Диаграмма вариантов использования записок

На рисунке 1.3 представлена диаграмма вариантов использования схем и mind map.

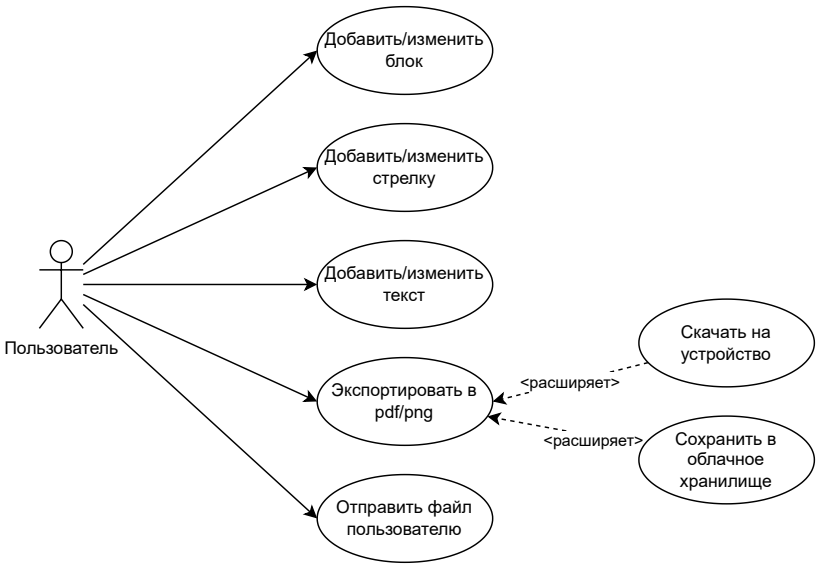

Рисунок 1.3 – Диаграмма вариантов использования схем и mind map

На рисунке 1.4 представлена диаграмма вариантов использования календаря.

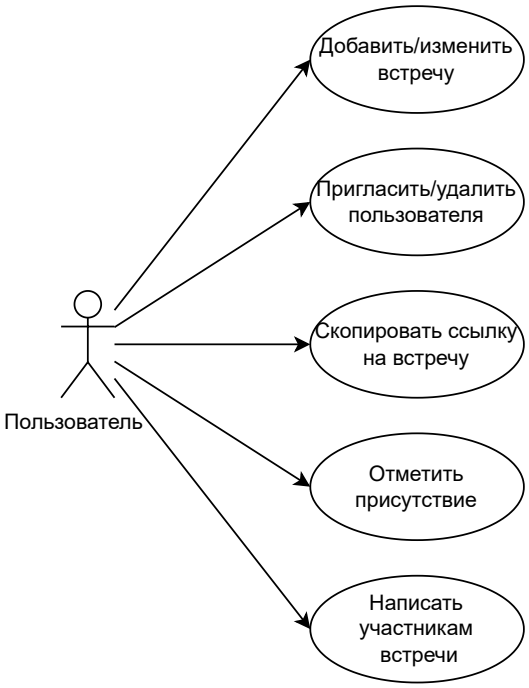

Рисунок 1.4 – Диаграмма вариантов использования календаря

На рисунке 1.5 представлена диаграмма вариантов использования трекера задач.

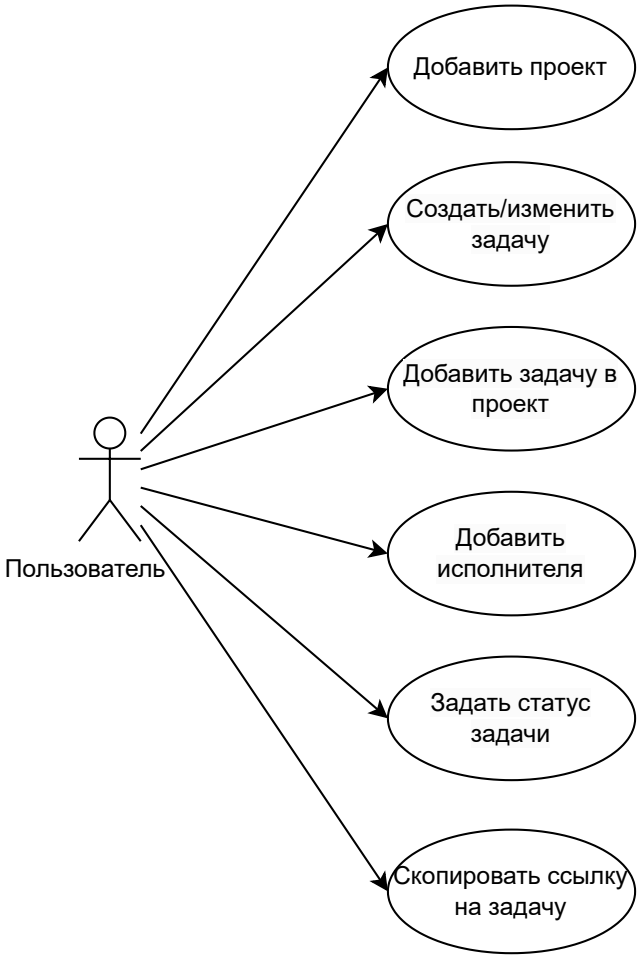

Рисунок 1.5 – Диаграмма вариантов использования трекера задач

На рисунке 1.6 представлена диаграмма вариантов использования чата.

Рисунок 1.6 – Диаграмма вариантов использования чата

На рисунке 1.7 представлена диаграмма вариантов использования презентаций.

Рисунок 1.7 – Диаграмма вариантов использования презентаций

1.5.3 Анализ требований пользователей и ролей

Для реализации системы необходимо определить основные роли пользователей и их требования к системе.

Роли пользователей

Первично предполагаются следующие роли в системе:

1) **Гость** — пользователь, не зарегистрированный в системе. Имеет ограниченный доступ, может просматривать публичную информацию о системе.

2) **Зарегистрированный пользователь** имеет учетную запись в системе. Может создавать проекты, присоединяться к проектам, использовать доступные инструменты.

3) **Администратор проекта** — пользователь, создавший проект или назначенный администратором. Имеет расширенные права на управление проектом и его участниками.

4) **Системный администратор** отвечает за общую поддержку и администрирование системы, управление пользователями, настройку прав доступа.

Требования пользователей

1. Гость

1) Возможность просматривать информацию о системе и её возможностях.

2) Возможность зарегистрироваться в системе.

2. Зарегистрированный пользователь

1) Возможность входа в систему с использованием учетных данных.

2) Создание новых проектов.

3) Просмотр и участие в проектах, в которые пользователь добавлен.

4) Использование инструментов для создания и редактирования контента (записки, схемы, презентации и т.д.).

5) Участие в коммуникациях проекта (чат).

6) Просмотр календаря и задач проекта.

3. Администратор проекта

1) Все возможности зарегистрированного пользователя.

2) Управление участниками проекта (добавление/удаление пользователей).

3) Назначение ролей и прав доступа участникам проекта.

4) Настройка параметров проекта.

4. Системный администратор

1) Управление пользователями системы (блокировка/разблокировка учетных записей, восстановление доступа).

2) Настройка глобальных параметров системы.

Вывод

Современные системы управления базами данных (СУБД) подразделяются на дореляционные, реляционные и постреляционные, каждая из которых отвечает различным требованиям к хранению данных — от строгих структурированных до более гибких моделей. Для хранения неоднородных данных, таких как в системах совместной работы над проектами, выбор конкретной модели определяется специфическими потребностями. Например, реляционные базы данных подходят, когда требуются строгая схема и сложные связи между сущностями, как в случае данных о пользователях. В то же время, нереляционные базы, такие как MongoDB, обеспечивают гибкость и масштабируемость при работе с полуструктурированными данными в формате JSON. Сравнительный анализ показал, что сочетание MongoDB и PostgreSQL позволяет использовать преимущества каждой модели, обеспечивая надежность и

целостность структурированных данных, что является важным для адаптации системы к нуждам проекта. Однако это требует тщательного планирования, интеграции и управления сложностями.

Для разрабатываемого комплекса была выбрана клиент-серверная архитектура как обеспечивающая большую простоту управления состоянием и безопасностью. Были определены основные протоколы взаимодействия: HTTP/S с REST для стандартных запросов и WebSockets для обеспечения функциональности в реальном времени (совместное редактирование, чаты, уведомления).

В результате формализации задачи были определены исходные задачи программно-алгоритмического комплекса, охватывающие основные функциональные возможности системы совместной работы над проектами, а также был проведен анализ ролей пользователей.

2 Конструкторский раздел

2.1 Основные положения предлагаемого ПАКа

При проектировании программно-алгоритмического комплекса были заложены следующие основные положения, направленные на обеспечение его функциональности, масштабируемости и многоцелевого использования:

1) **Гибридная модель хранения данных:** Учитывая необходимость работы с неоднородными данными, применяется гибридный подход к хранению:

1) **PostgreSQL** используется для хранения структурированных, реляционных данных, требующих строгой схемы, целостности и поддержки транзакций. Сюда относятся данные о пользователях, проектах, файлах (метаданные), типах файлов, а также связи между ними (участие пользователей в проектах, принадлежность файлов проектам, роли и права доступа).

2) **MongoDB** используется для хранения полуструктурированных и неструктурированных данных, требующих гибкости схемы и горизонтальной масштабируемости. В MongoDB хранится фактическое содержимое файлов, состоящее из различных блоков данных, а также информация о стилях.

2) **Комбинированное использование REST API и WebSockets:** Для взаимодействия между клиентом и сервером будут применяться два основных механизма:

1) **REST API** будет использоваться для выполнения стандартных CRUD-операций (создание, чтение, обновление, удаление) над основными сущностями системы (пользователи, проекты, файлы и т.д.), а также для запроса или модификации данных, не требующих немедленной синхронизации у других пользователей.

2) **WebSocket** будет использоваться для обеспечения взаимодействия в реальном времени важного для функций совместной работы: одновременное редактирование документов, обмен сообще-

ниями в чате, доставка мгновенных уведомлений, отображение статуса присутствия пользователей.

3) **Клиент-серверная архитектура:** Комплекс реализуется в рамках клиент-серверной архитектуры, где пользователь взаимодействует с системой через клиентское приложение (веб-браузер), а основная бизнес-логика, управление данными и синхронизация выполняются на серверной стороне.

4) **Управление неоднородными данными:** Для управления разнообразным контентом (текстовые заметки, схемы, презентации, задачи и т.д.) каждый документ будет представлен как совокупность таких блоков различных типов (например, параграф, заголовок, изображение, узел схемы, задача). Центральным элементом этой модели является суперобъект (SuperObject), хранящийся в MongoDB. Использование сущности суперобъекта вызвано необходимостью унифицированного подхода к хранению и обработке различных типов контента внутри системы. В условиях, когда пользователь может создавать документы с разной внутренней структурой (заметки, схемы, календари, задачи и т.д.), требуется обобщённая сущность, способная представлять такой файл в MongoDB вне зависимости от его специфики. SuperObject выполняет роль обобщённого контейнера, объединяющего метаинформацию о документе, его типе и связи с соответствующими специализированными коллекциями. Это позволяет:

1) использовать единый механизм идентификации и доступа к документу на стороне MongoDB;

2) реализовать добавление новых типов сервисов без изменения общей структуры базы;

3) обеспечить согласованность между реляционной и документной моделями за счёт жёсткой связи между files в PostgreSQL и super_objects в MongoDB.

5) **Модульность серверного приложения:** Серверное приложение проектируется с учетом модульности, где каждая основная функциональная область (управление пользователями, проектами, контентом

и т.д.) выделяется в логический компонент, что упрощает разработку, тестирование и дальнейшее развитие системы.

6) **Масштабируемость:** Архитектура и выбор технологий (в частности, использование MongoDB и возможность репликации/масштабирования PostgreSQL) закладывают основу для потенциальной горизонтальной и вертикальной масштабируемости системы при росте нагрузки.

2.2 Структура и компоненты программного приложения

Программно-алгоритмический комплекс строится на основе многоуровневой архитектуры, близкой к принципам чистой или луковой архитектуры, с четким разделением ответственности между слоями. Общая структура представлена на рисунке 2.1.

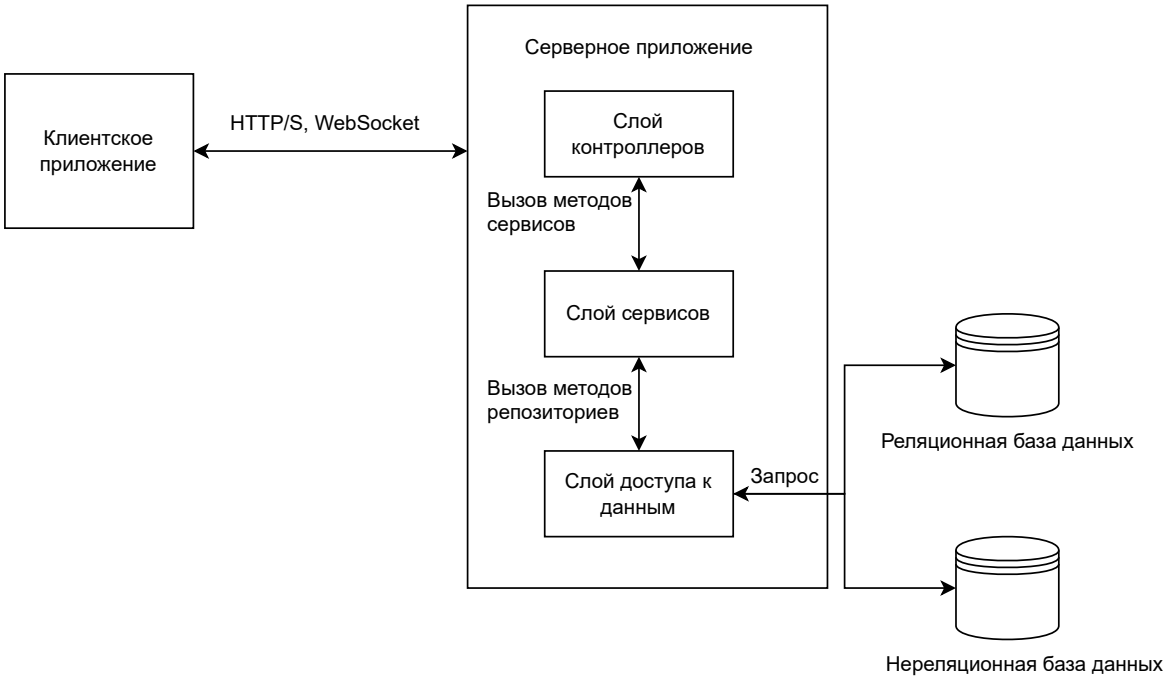

Рисунок 2.1 – Общая архитектура программно-алгоритмического комплекса

Основные компоненты и слои архитектуры:

1) **Клиентское приложение** будет реализовано как одностраничное веб-приложение (SPA), отвечающее за пользовательский интерфейс, визуализацию данных и взаимодействие с пользователем. Обмен данными с серверной частью будет осуществляется через REST API и WebSocket.

2) **Серверное приложение** реализует бизнес-логику и будет состоять из следующих слоев:

- **Слой Контроллеров**, который отвечает за обработку входящих HTTP-запросов и вызов методов соответствующего сервиса. В этот же слой логически входит обработка WebSocket-соединений для real-time взаимодействия.
- **Слой Сервисов**, который инкапсулирует основную бизнес-логику приложения. Сервисы координируют работу репозиториев, выполняют преобразование данных, реализуют правила предметной области и управляют транзакциями.
- **Слой Доступа к Данным**, включающий интерфейсы репозиториев и абстрагирующий детали взаимодействия с базами данных, предоставляя методы для CRUD-операций.

3) **Система хранения реляционных данных** хранит метаданные и связи между основными сущностями системы.

4) **Система хранения документных данных:** хранит гибкое содержимое файлов и связанные с ним стили.

Разработанная архитектура позволяет создать стандартизированный программный интерфейс, обеспечивающий единообразный доступ к различным типам данных и возможность простого добавления новых сервисов без изменения базовой структуры системы. Это достигается за счет типизации обработки запросов через REST endpoints и использования общих паттернов для работы с данными на уровне контроллеров и сервисов.

На рисунке 2.2 представлена более подробная диаграмма компонентов.

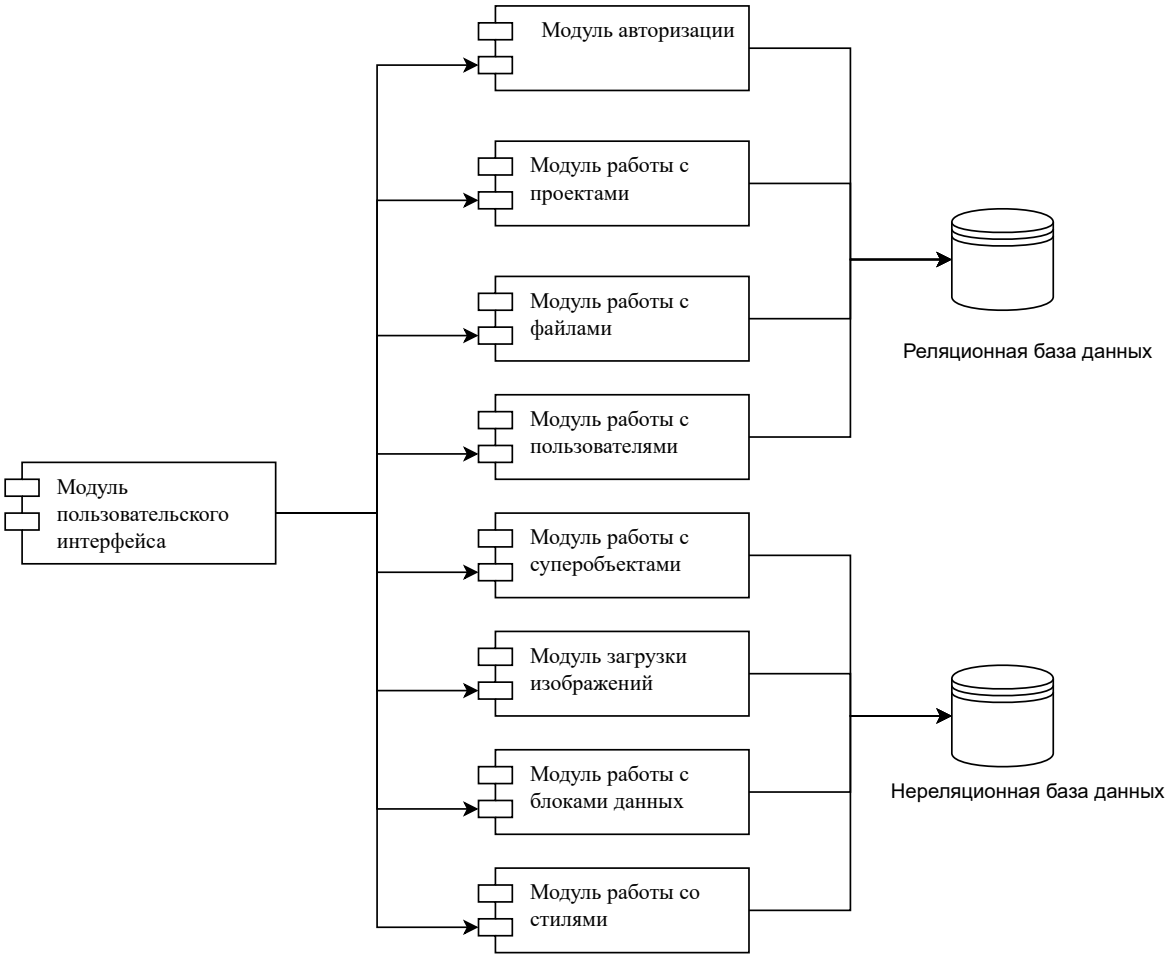

Рисунок 2.2 – Диаграмма компонентов приложения

На рисунке 2.3 приведена диаграмма классов основных бизнес-сущностей системы и их взаимодействия. Она фокусируется на ключевых сущностях User (Пользователь) и Project (Проект), а также на компонентах, отвечающих за их обработку (контроллеры, сервисы, репозитории).

Выбор для демонстрации именно этих классов и связанных с ними компонентов обусловлен следующими причинами:

— Сущности User и Project являются основополагающими в разрабатываемой системе управления проектами и совместной работы. Практически вся функциональность так или иначе связана с пользователями и проектами, к которым они имеют доступ.

— Структура этих классов и паттерны их обработки являются типичными и применяются для большинства других сущностей системы (на-

пример, для файлов, типов файлов, блоков контента). Таким образом, данная диаграмма иллюстрирует общий подход к проектированию и организации кода на серверной стороне.

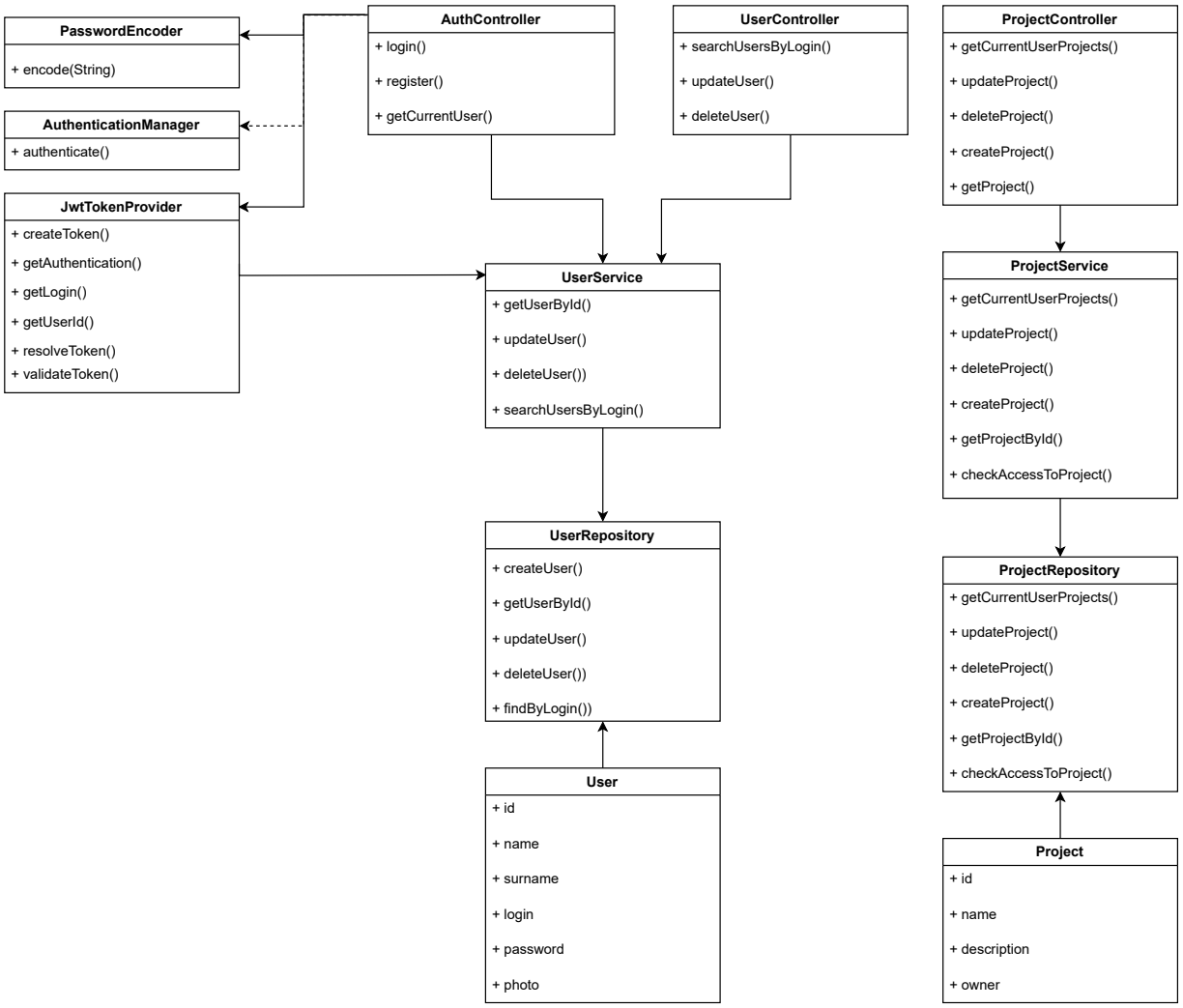

Рисунок 2.3 – Диаграмма классов для сущностей User, Project и связанных компонентов

2.3 Ключевые структуры данных

Для хранения информации выбраны две модели данных: реляционная модель в PostgreSQL и документная модель в MongoDB.

Реляционная модель данных

Структура реляционной базы данных предназначена для хранения основной метаинформации о сущностях системы и связей между ними.

Основные таблицы:

— `users`: информация о пользователях системы (id, name, surname, login, password(hash), photo).

— `projects`: описание проектов, создаваемых пользователями. Каждый проект связан с пользователем-владельцем (id, name, date, owner_id).

— `file_types`: типы создаваемых в системе файлов или документов (id, name, например, «заметка», «схема», «презентация»).

— `files`: метаинформация о каждом файле или документе, независимо от его внутреннего содержания. Включает общие атрибуты (id, name, type_id, author_id, date) и ссылку `superObjectId` на соответствующий документ в MongoDB, где хранится сам контент.

— `projects_users`: связующая таблица для реализации ролевой модели доступа пользователей к проектам, определяющая роль каждого участника в конкретном проекте (id, project_id, user_id, role).

— `projects_files`: связующая таблица, указывающая на принадлежность файлов (из таблицы `files`) к проектам (из таблицы `projects`) (id, project_id, file_id).

Эта структура обеспечивает ссылочную целостность, транзакционность и возможности для сложных запросов с объединением данных.

Полная реляционная схема базы данных представлена на рисунке 2.4.

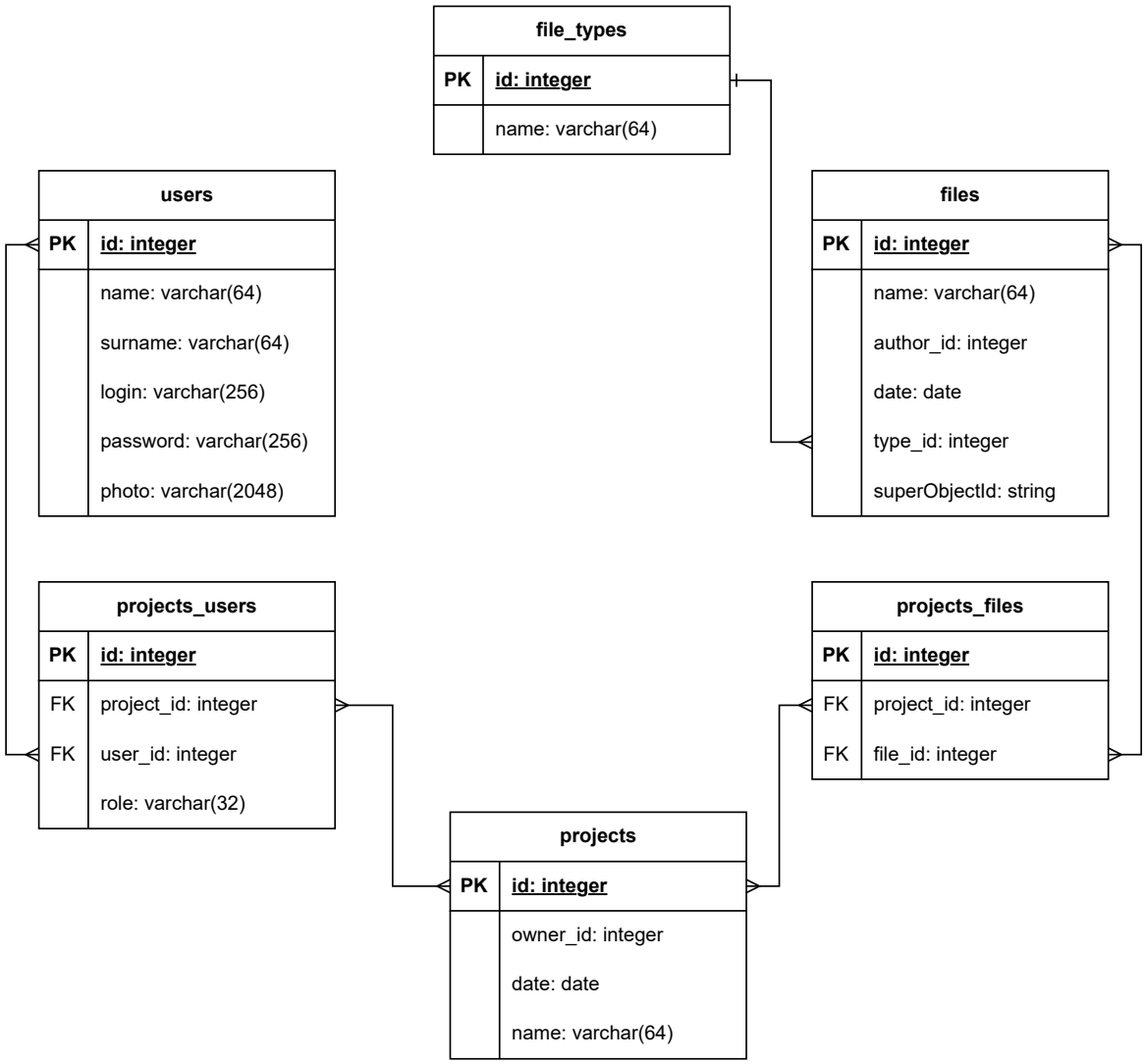

Рисунок 2.4 – Диаграмма реляционной базы данных

Документная модель данных (MongoDB)

Документо-ориентированная модель данных в MongoDB используется для хранения непосредственно контента создаваемых сущностей, а также сложных, гибко структурированных данных, связанных с их представлением и стилизацией.

Рассмотрим основные коллекции.

`super_objects:` коллекция, содержащая основную информацию для каждого файла, метаданные которого хранятся в PostgreSQL.

1) `id`: уникальный идентификатор документа MongoDB.

2) `fileId`: внешний ключ (индексированный и уникальный), связывающий данный документ с соответствующей записью в таблице `files` PostgreSQL.

3) `name`: название документа, которое может синхронизироваться с `files.name` или быть специфичным.

4) `serviceType`: ключевой атрибут, определяющий тип сервиса и структуру контента (например, `note`, `scheme`, `presentation`, `mindmap`, `calendar`, `tracker`, `chat`).

5) `lastChangeDate`: время последнего изменения.

6) `stylesMapId`: (опционально) ссылка на документ из коллекции `styles_maps`, определяющий примененные к элементам контента стили.

7) Для блочно-ориентированных сервисов (`serviceType = note`, `presentation`, `chat`): `firstItem` - идентификатор первого блока контента в последовательности и `lastItem` - идентификатор последнего блока контента.

8) Для других сервисов поля `firstItem` и `lastItem` могут не использоваться, а контент будет структурирован в специфичных для сервиса коллекциях, связанных с `SuperObject` по его `id` или `fileId`.

`content_blocks:` коллекция, предназначенная для хранения отдельных блоков контента, используется для таких сервисов, как заметки, слайды презентаций, или сообщения в чате.

1) `id`: уникальный идентификатор блока.

2) `objectType`: строковый идентификатор типа блока (например, `paragraph`, `header`, `image`, `list`, `chatMessage`).

3) `data`: объект с парами ключ-значение со специфичными данными для данного `objectType`.

4) `nextItem`, `prevItem`: идентификаторы для связывания блоков в двунаправленный список, формирующий последовательный контент.

Специфичные коллекции для неблочных сервисов: для поддержки функциональности, выходящей за рамки простой блочной структуры, вводятся дополнительные специализированные коллекции:

1) Для схем и mindmap (`serviceType = scheme`, `mindmap`):

1) `nodes`: коллекция узлов (фигур) с их атрибутами (тип, координаты, размеры, содержимое). Каждый узел связан с родительским `SuperObject`.

2) `edges`: коллекция ребер (связей) между узлами, также с атрибутами (тип, узлы-источник и цель, метки) и связью с `SuperObject`.

2) Для календаря (`serviceType = calendar`): `calendar_events`: коллекция событий с атрибутами (название, время начала/окончания, описание, местоположение, участники, правила повторения). Каждое событие связано с `SuperObject`, представляющим календарь.

3) Для трекера задач (`serviceType = tracker`):

1) `task_columns`: коллекция колонок (статусов) на доске задач, каждая связана с `SuperObject` доски.

2) `task_items`: коллекция задач с их атрибутами (название, описание, исполнители, сроки, приоритет, вложенные файлы, подзадачи), каждая задача принадлежит определенной колонке и связана с `SuperObject` доски.

Система стилизации: для обеспечения гибкой настройки внешнего вида различных элементов контента предлагается следующая структура коллекций:

1) `styles:` коллекция с определениями конкретных стилей. Каждый документ стиля включает:

1) `id`: уникальный идентификатор стиля.

2) `targetType`: тип целевого объекта, к которому применим стиль (например, `text_block`, `scheme_node`, `calendar_event`), что позволяет группировать релевантные атрибуты.

3) `attributes`: объект с парами ключ-значение, описывающими визуальные свойства (например, `color`, `fontSize`, `backgroundColor`, `borderStyle`, специфичные для `targetType` атрибуты).

2) `styles_maps:` коллекция, обеспечивающая связь между элементами контента и определенными стилями. Каждый документ (один на `SuperObject`, связанный через `stylesMapId`) содержит:

1) `id`: уникальный идентификатор карты стилей.

2) `links`: массив объектов `style_link`.

3) `style_link`: описывает применение конкретного стиля к элементу:

1) `elementId`: идентификатор элемента (например, `ContentBlock.id`, `Node.id`), к которому применяется стиль.

2) `styleId`: ссылка на идентификатор документа в коллекции `styles`.

3) `scope`: (опционально) уточнение части элемента, к которой применяется стиль (например, `background`, `text`, `border`).

4) `state`: (опционально) Для описания стилей интерактивных состояний (например, `hover`, `active`).

Эта модель обеспечивает гибкость для хранения сложного и разнообразного контента.

Схема коллекций этой базы данных представлена на рисунке 2.5.

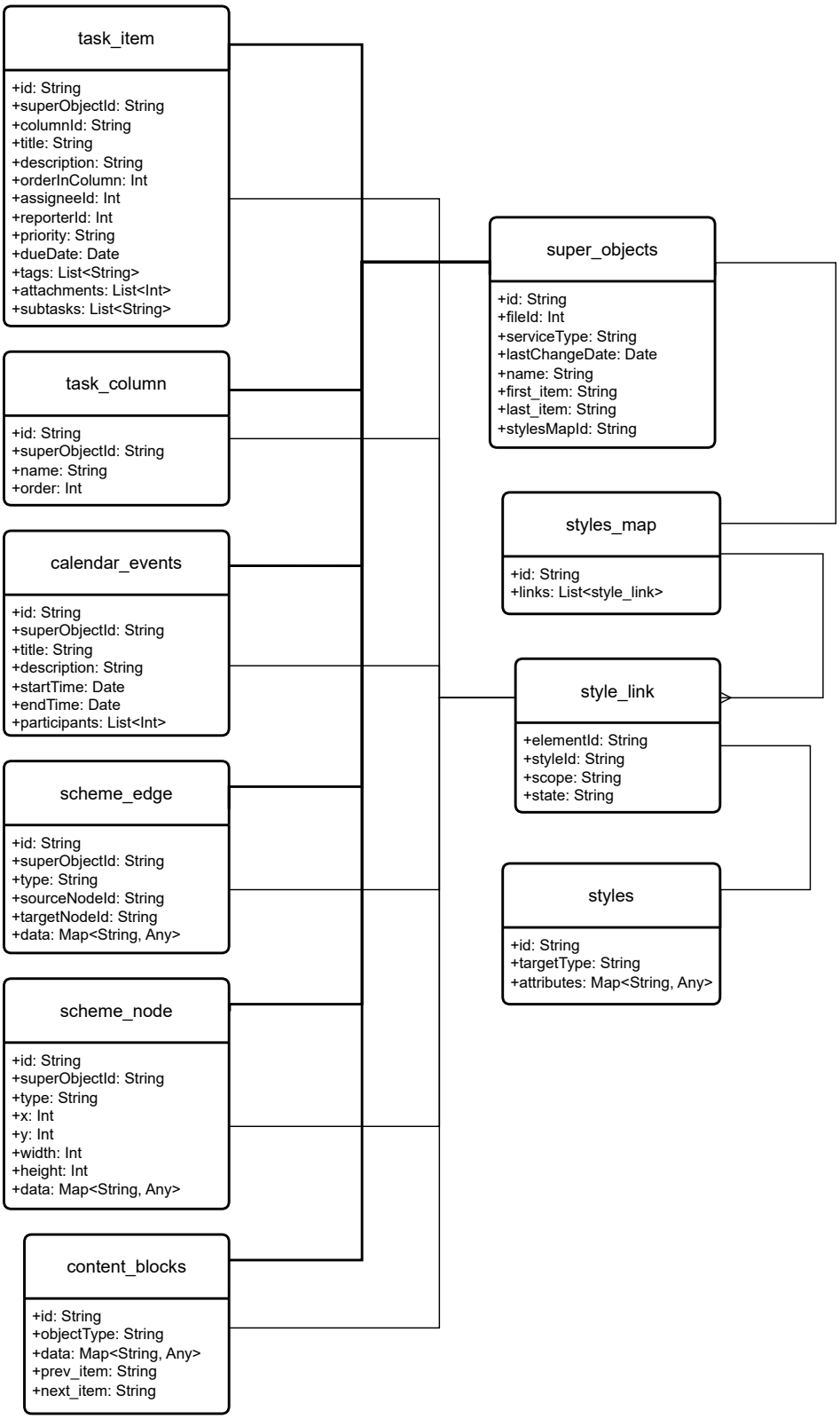

Рисунок 2.5 – Схема коллекций нереляционной базы данных

Вывод

В рамках конструкторского раздела были разработаны и описаны основные архитектурные решения для программно-алгоритмического комплекса совместной работы и управления проектами.

Определены ключевые положения, включая выбор клиент-серверной архитектуры, гибридной модели хранения данных с использованием PostgreSQL для структурированной информации и MongoDB для гибкого контента, а также комбинированного подхода к взаимодействию через REST API и WebSocket для поддержки реального времени. Представлена общая структура приложения, выделяющая клиентскую и серверную части, а также многоуровневую организацию серверного приложения (контроллеры, сервисы, доступ к данным), что способствует модульности и разделению ответственности.

Описаны основные компоненты серверного приложения (сервисы управления пользователями, проектами, файлами, контентом и др.) и принципы их взаимодействия, проиллюстрированные на примере сценария совместного редактирования с помощью диаграммы последовательности. Зафиксированы ключевые структуры данных для реляционной (PostgreSQL) и документной (MongoDB) моделей, обеспечивающие хранение всей необходимой информации – от метаданных до сложного содержимого файлов и стилей.

3 Технологический раздел

3.1 Выбор средств программной реализации

При выборе технологий для разработки программно-алгоритмического комплекса основной упор делался на их способность обеспечить реализацию поставленных задач, а также на распространенность инструментов и наличие документации.

Язык Kotlin был выбран в качестве основного для серверной разработки, так как он предоставляет все необходимые средства для решения поставленных задач. Ключевые аспекты выбора:

— Стандартная библиотека и возможности языка (например, корутины для асинхронных операций, data-классы) покрывают широкий спектр задач, возникающих при разработке.

— Возможность использования существующих Java-библиотек и фреймворков, включая Spring.

Фреймворк Spring Boot используется для построения REST API, интеграции с базами данных PostgreSQL и MongoDB, а также для реализации механизмов безопасности через Spring Security с JWT. Данный фреймворк предоставляет необходимую инфраструктуру для обработки HTTP-запросов, управления транзакциями и конфигурацией приложения.

TypeScript выбран для разработки клиентской части, так как статическая типизация способствует более предсказуемой разработке интерфейсов со сложной логикой. Библиотека React позволяет:

— Структурировать интерфейс в виде переиспользуемых компонентов.

— Оптимизированно обновлять DOM за счет использования концепции виртуального DOM.

Для управления общим состоянием, таким как информация об аутентифицированном пользователе, применяется React Context API. Маршрутизация в одностраничном приложении реализована с помощью библиотеки React Router. Для выполнения HTTP-запросов к серверному REST API применяется библиотека axios.

3.2 Тестирование программно-алгоритмического комплекса

Для проверки корректности работы реализованных модулей проводилось сквозное тестирование, а также были разработаны модульные тесты для некоторых компонентов серверной части.

В ходе разработки выполнялось тестирование ключевых пользовательских сценариев, таких как:

— Регистрация и аутентификация пользователя.

— Создание и выбор проекта.

— Создание заметки.

— Добавление и базовое редактирование контента в заметке.

— Загрузка изображения в заметку.

— Добавление и удаление участников проекта, изменение их роли.

В листингах 3.4 и 3.5 представлен E2E тест для сценария создания заметки, включая:

— регистрацию пользователя;

— авторизацию и получение JWT;

— создание проекта;

— создание типа файла;

— создание файла с привязкой к проекту.

Листинг 3.1 – E2E тест для сценария создания заметки (часть 1)

```
@SpringBootTest
@AutoConfigureMockMvc
class CreateNoteE2ETest {
    @Autowired
    lateinit var mockMvc: MockMvc
    @Autowired
    lateinit var objectMapper: ObjectMapper

    @Test
    fun `should create a note after auth and project setup`() {
        val registerRequest = RegisterRequest(
            name = "John",
            surname = "Doe",
            login = "johndoe",
            password = "password123"
        )

        mockMvc.perform(post("/users/register")
            .contentType(MediaType.APPLICATION_JSON)
            .content(objectMapper
            .writeValueAsString(registerRequest)))
            .andExpect(status().isCreated)

        val loginRequest = LoginRequest(
            login = "johndoe",
            password = "password123"
        )

        val loginResult = mockMvc.perform(post("/users/login")
            .contentType(MediaType.APPLICATION_JSON)
            .content(objectMapper
            .writeValueAsString(loginRequest)))
            .andExpect(status().isOk)
            .andReturn()

        val jwtResponse = objectMapper.readValue(
            loginResult.response.contentAsString,
            JwtResponse::class.java
        )
        val token = jwtResponse.token!!
```

Листинг 3.2 – E2E тест для сценария создания заметки (часть 2)

```
        val userId = jwtResponse.id!!

        val fileTypeResult = mockMvc.perform(post("/file-types")
            .header("Authorization", "Bearer $token")
            .contentType(MediaType.APPLICATION_JSON)
            .content("""{"name": "note"}"""))
            .andExpect(status().isOk)
            .andReturn()

        val fileTypeId =
           objectMapper.readTree(fileTypeResult.response
        .contentAsString).get("id").asLong()

        val project = Project(name = "Test Project", description
           = "Project for testing")
        val projectResult = mockMvc.perform(post("/projects")
            .header("Authorization", "Bearer $token")
            .contentType(MediaType.APPLICATION_JSON)
            .content(objectMapper.writeValueAsString(project)))
            .andExpect(status().isCreated)
            .andReturn()

        val projectId =
           objectMapper.readTree(projectResult.response
        .contentAsString).get("id").asLong()

        val projectUserDto = ProjectUserDto(
            projectId = projectId,
            userId = userId,
            role = ProjectRole.OWNER
        )

        mockMvc.perform(post("/projects-users")
            .header("Authorization", "Bearer $token")
            .contentType(MediaType.APPLICATION_JSON)
            .content(objectMapper
            .writeValueAsString(projectUserDto)))
            .andExpect(status().isCreated)
```

Листинг 3.3 – E2E тест для сценария создания заметки (часть 3)

```
        val fileDto = FileDto(
            name = "Test Note",
            typeId = fileTypeId,
            authorId = userId
        )

        val fileResult = mockMvc.perform(post("/files")
            .header("Authorization", "Bearer $token")
            .contentType(MediaType.APPLICATION_JSON)
            .content(objectMapper.writeValueAsString(fileDto)))
            .andExpect(status().isOk)
            .andReturn()

        val fileId = objectMapper.readTree(fileResult.response
        .contentAsString).get("id").asLong()

        mockMvc.perform(post("/projects/$projectId/files/link")
            .header("Authorization", "Bearer $token")
            .param("file_id", fileId.toString()))
            .andExpect(status().isOk)

        mockMvc.perform(get("/projects/$projectId/files")
            .header("Authorization", "Bearer $token"))
            .andExpect(status().isOk)
            .andExpect(jsonPath("$[0].name").value("Test Note"))
    }
}
```

Было проведено модульное тестированию серверной части. В листингах 3.4 и 3.5 представлены тесты регистрации пользователя и создания проекта.

Листинг 3.4 – Тест для AuthController (регистрация)

```
@ExtendWith(MockitoExtension::class)
class AuthControllerTest {
    @Mock
    lateinit var userRepository: UserRepository
    @Mock
    lateinit var passwordEncoder: PasswordEncoder
    @Mock
    lateinit var authenticationManager: AuthenticationManager
    @Mock
    lateinit var jwtTokenProvider: JwtTokenProvider

    @InjectMocks
    lateinit var authController: AuthController

    @Test
    fun `test register user success`() {
        val request = RegisterRequest("Test", "User",
           "testlogin", "password123", null)
        val encodedPassword = "encodedPassword"
        val savedUser = User(1L, "Test", "User", "testlogin",
           encodedPassword, null)

        `when`(userRepository.findByLogin(request.login))
        .thenReturn(Optional.empty())
        `when`(passwordEncoder.encode(request.password))
        .thenReturn(encodedPassword)
        `when`(userRepository.save(any(User::class.java)))
        .thenReturn(savedUser)

        val response = authController.register(request)

        assertEquals(HttpStatus.CREATED, response.statusCode)
        assertNotNull(response.body)
        verify(userRepository).findByLogin(request.login)
        verify(passwordEncoder).encode(request.password)
        verify(userRepository).save(any(User::class.java))
    }
}
```

Листинг 3.5 – Тест для ProjectService (создание проекта)

```
@ExtendWith(MockitoExtension::class)
class ProjectServiceTest {
    @Mock
    lateinit var projectRepository: ProjectRepository
    @Mock
    lateinit var userRepository: UserRepository
    @Mock
    lateinit var projectUserService: ProjectUserService
    @InjectMocks
    lateinit var projectService: ProjectService
    @Test
    fun `test create project success`() {
        val projectName = "My New Project"
        val ownerLogin = "ownerUser"
        val ownerId = 1L
        val ownerUser = User(ownerId, "Owner", "Test",
           ownerLogin, "pass", null)
        val projectToSave = Project(name = projectName, owner =
           ownerUser)
        val savedProject = Project(id=100L, name = projectName,
           owner = ownerUser)
        val authentication = mock(Authentication::class.java)
        val securityContext = mock(SecurityContext::class.java)
        `when`(securityContext.authentication)
        .thenReturn(authentication)
        SecurityContextHolder.setContext(securityContext)
        `when`(authentication.principal).thenReturn(ownerUser)
        `when`(userRepository.findById(ownerId))
        .thenReturn(Optional.of(ownerUser))
        `when`(projectRepository.save(any(Project::class.java)))
        .thenReturn(savedProject)
        val createdProjectDto =
           projectService.createProject(projectToSave)
        assertNotNull(createdProjectDto)
        assertEquals(projectName, createdProjectDto.name)
        assertEquals(ownerId, createdProjectDto.owner.id)
        verify(projectRepository).save(any(Project::class.java))
    }
}
```

Реализация серверной и клиентской части приложения представлены в Приложении Б и В соответственно.

3.3 Описание взаимодействия пользователя с программным обеспечением

Основные сценарии работы с реализованным функционалом (проекты и заметки).

1) **Вход и регистрация.** Пользователь начинает работу со страницы входа. Если у пользователя нет учетной записи, он может перейти на страницу регистрации. Интерфейсы этих страниц стандартны и включают поля для ввода логина/e-mail, пароля, имени и фамилии (при регистрации). После успешной аутентификации пользователь перенаправляется на главную страницу приложения — панель управления проектами.

2) **Работа с проектами.** На панели управления проектами интерфейс разделен на боковую панель для навигации по проектам и основную рабочую область для отображения содержимого выбранного проекта. Пользователь может:

— Создать новый проект, указав его название в модальном окне.

— Выбрать существующий проект из списка на боковой панели.

— В основной области просматривать файлы выбранного проекта.

— Создать новый файл (заметку) в текущем проекте.

— Редактировать названия проектов и файлов.

— Управлять доступом к проекту (для владельца): добавлять других пользователей по логину, назначать им роли («Редактор», «Читатель») и удалять их из проекта через модальное окно.

3) **Редактирование заметки.** Пользователь может:

— Вводить и форматировать текст.

— Вставлять, удалять, перемещать и изменять различные типы блоков: заголовки, списки, цитаты, изображения.

Изменения в документе сохраняются автоматически с определенным интервалом, отправляя данные на сервер для синхронизации. Пользователи, не имеющие прав на редактирование, видят документ в режиме «только чтение». Скриншоты основных экранов интерфейса представлены в Приложении Г.

3.4 Описание API программно-алгоритмического комплекса

Разработанный программно-алгоритмический комплекс предоставляет REST API для взаимодействия с клиентскими приложениями. API построено в соответствии с принципами REST и организовано по функциональным группам для обеспечения логического разделения интерфейсов доступа к различным сущностям системы.

Основные группы API

1) **Аутентификация** — интерфейсы для регистрации и аутентификации пользователей:

— `POST /users/login` — аутентификация пользователя и получение JWT-токена;

— `POST /users/register` — регистрация нового пользователя.

2) **Управление пользователями** — операции с данными пользователей:

— `GET /users/me` — получение информации о текущем аутентифицированном пользователе;

— `GET /users/search` — поиск пользователей по логину (для добавления в проекты);

— `GET|PUT|DELETE /users/{user_id}` — получение, обновление и удаление пользователя.

3) **Управление проектами** — создание и управление проектами:

— `POST /projects` — создание нового проекта;

— `GET /projects` — получение списка проектов текущего пользователя;

— `GET|PUT|DELETE /projects/{project_id}` — получение, обновление и удаление проекта.

4) **Управление доступом к проектам** — настройка прав доступа:

— `POST /projects-users` — добавление пользователя в проект с указанием роли;

— `GET /projects-users/project/{project_id}/users` — получение списка пользователей проекта;

— `PUT /projects-users/project/{project_id} /user/{user_id}` — изменение роли пользователя;

— `DELETE /projects-users/project/{project_id} /user/{user_id}` — удаление пользователя из проекта.

5) **Управление файлами в проектах** — операции с файлами в контексте проектов:

— `GET /projects/{project_id}/files` — получение всех файлов проекта;

— `POST /projects/{project_id}/files` — загрузка файла в проект;

— `GET|DELETE /projects/{project_id}/files/{file_id}` — получение и удаление файла из проекта;

— `PATCH /projects/{project_id}/files/{file_id}/name` — обновление имени файла;

— `PATCH /projects/{project_id}/files/ {file_id}/super-object` — связывание файла с документом;

— `GET /projects/{project_id}/files/{file_id}/download` — скачивание файла.

6) **Хранение документов (суперобъект)** — работа со структурой документов:

— `POST /super-objects` — создание нового документа;

— `GET /super-objects/by-file/{fileId}` — получение документа по идентификатору файла;

— `GET|PUT|DELETE /super-objects/{id}` — получение, обновление и удаление документа;

— `PUT /super-objects/{superObjectId}/sync-blocks` — синхронизация блоков документа.

7) **Блоки контента** — управление блоками содержимого документов:

— `POST /content-blocks` — создание нового блока;

— `GET|PUT|DELETE /content-blocks/{id}` — получение, обновление и удаление блока.

8) **Стили** и **карты стилей** — управление форматированием документов:

— `POST /styles`, `POST /styles-maps` — создание стиля и карты стилей;

— `GET|PUT|DELETE /styles/{id}` — операции со стилями;

— `GET|PUT|DELETE /styles-maps/{id}` — операции с картой стилей.

Особенности реализации API

API реализовано с использованием стандартных HTTP-методов:

— `GET` — для получения данных;

— `POST` — для создания новых ресурсов;

— `PUT` — для полного обновления ресурсов;

— `PATCH` — для частичного обновления ресурсов;

— `DELETE` — для удаления ресурсов.

Ответы сервера включают соответствующие HTTP-коды состояния:

— 200 (OK) — успешное выполнение операции;

— 201 (Created) — успешное создание ресурса;

— 204 (No Content) — успешное выполнение операции без возвращаемых данных;

— 400 (Bad Request) — ошибка в запросе;

— 401 (Unauthorized) — отсутствие аутентификации;

— 404 (Not Found) — запрашиваемый ресурс не найден.

Безопасность API обеспечивается через JWT-токены, передаваемые в заголовке `Authorization` каждого запроса, требующего аутентификации. Авторизация для доступа к ресурсам проверяется на серверной стороне на основе ролей пользователя в проекте (владелец, редактор, наблюдатель).

Данный набор API охватывает все необходимые операции для работы с программно-алгоритмическим комплексом и обеспечивает полноценное взаимодействие между клиентской и серверной частями приложения. Также для облегчения взаимодействия с API и его документирования был интегрирован Swagger UI. Спецификация OpenAPI содержит подробное описание всех эндпоинтов, включая требуемые параметры, форматы запросов и ответов, возможные коды ошибок и примеры использования.

Вывод

В данном разделе был обоснован выбор основных средств разработки: Kotlin и Spring Boot для серверной части, TypeScript и React для клиентской. Представлены подходы к тестированию, включая примеры модульных тестов для серверной части. Описаны основные сценарии взаимодействия пользователя с реализованным функционалом и предоставлено описание API. Разработанный программный комплекс является основой для дальнейшего расширения и добавления новых сервисов.

4 Исследовательский раздел

4.1 Исследование применимости разработанного программного обеспечения

В условиях исчерпания естественных входных данных для формирования датасетов, необходимых для обучения систем искусственного интеллекта, разработанный программно-алгоритмический комплекс представляет особую ценность. При его внедрении в образовательный процесс возможно накопление значительных объемов структурированных данных для научных исследований.

На основе учебного плана кафедры ИУ7 «Программное обеспечение ЭВМ и информационные технологии» и актуальных данных о контингенте студентов на 2025 год проведена оценка потенциала формирования датасетов при внедрении системы в образовательный процесс [17, 18].

Исходные данные для расчета

— **Фактический контингент студентов:**

  – 1 курс — 160 человек.

  – 2 курс — 117 человек.

  – 3 курс — 103 человек.

  – 4 курс — 93 человек.

  – **Всего:** 473 человек.

— **Коэффициент активных студентов:** 0,82 (учитывая отчисления и академические отпуски).

— **Фактический контингент активных студентов:** 389 человек.

— **Характер учебных активностей:**

  – По курсовым и НИР: отчеты со схемами.

  – На гуманитарных дисциплинах: минимум 1 презентация и 2 отчета на студента.

– Технические дисциплины: около 60% преподавателей используют презентации.

Приведенные в таблице 4.1 расчеты представляют приближенную оценку документов, которые могут быть созданы в системе.

Таблица 4.1 – Потенциальный объем документов при внедрении системы

Тип документа	**Формула расчёта**	**Итого**
Презентации студентов по гуманитарным дисциплинам	1 курс: 2 дисц. × 160 студ. × 0,82 × 1 = 262 2 курс: 2 дисц. × 117 студ. × 0,82 × 1 = 191 3 курс: 2 дисц. × 103 студ. × 0,82 × 1 = 168 4 курс: 1 дисц. × 93 студ. × 0,82 × 1 = 76	697
Отчёты по гуманитарным дисциплинам	1 курс: 2 дисц. × 160 студ. × 0,82 × 2 = 524 2 курс: 2 дисц. × 117 студ. × 0,82 × 2 = 383 3 курс: 2 дисц. × 103 студ. × 0,82 × 2 = 337 4 курс: 1 дисц. × 93 студ. × 0,82 × 2 = 152	1,396
Отчёты по курсовым работам и НИР	3 курс: 2 КР × 103 студ. × 0,82 = 168 4 курс: 2 КР × 93 студ. × 0,82 = 152 НИР: (103+93) × 0,82 = 160	480
Презентации преподавателей	25 дисциплин × 2 преп. × 17 недель × 0,6	510
Отчёты по лабораторным работам	1 курс: 5 дисц. × 160 студ. × 0,82 × 6 = 3,936 2 курс: 5 дисц. × 117 студ. × 0,82 × 6 = 2,878 3 курс: 5 дисц. × 103 студ. × 0,82 × 6 = 2,533 4 курс: 5 дисц. × 93 студ. × 0,82 × 6 = 2,287	11,634
Отчёты по практикам	1 курс: 2 × 160 × 0,82 = 262 2 курс: 1 × 117 × 0,82 = 95 3 курс: 2 × 103 × 0,82 = 168 4 курс: 1 × 93 × 0,82 = 76	601
Выпускные квалификационные работы	93 студентов × 0,82	76
ИТОГО		**15,394**

Корректировка оценки с учетом специфики учебного процесса

Необходимо отметить, что данный анализ основан на ряде допущений:

— Количество гуманитарных дисциплин по курсам и число отчетов на каждой дисциплине может варьироваться в зависимости от требований конкретных преподавателей.

— Процент преподавателей, использующих презентации на технических дисциплинах (60%), является приблизительной оценкой.

— Коэффициент активных студентов (0.82) представляет собой усредненное значение и может различаться по курсам и семестрам.

— Количество лабораторных работ на дисциплину является средним значением.

Для получения более реалистичной оценки необходимо также учесть следующие факторы:

— Не все преподаватели сразу перейдут на использование новой системы.

— Часть лабораторных работ может выполняться в виде кода без оформления формального отчета.

— Некоторые студенты могут использовать систему не для всех типов работ.

С учетом этих факторов, при коэффициенте внедрения системы 0.7 в первый год, можно ожидать накопление около 10,800 документов различных типов. При полноценном внедрении системы в образовательный процесс в течение 2-3 лет этот показатель может приблизиться к расчетному значению.

Качественная оценка потенциальных датасетов

При внедрении разработанного программно-алгоритмического комплекса возможно формирование следующих типов датасетов:

1) **Датасеты для анализа научно-технических текстов:**

— Корпус технической документации на русском языке.

— Специализированные терминологические словари по программной инженерии.

— Наборы данных для автоматизированной проверки текстов на плагиат.

2) **Датасеты для анализа схем и диаграмм:**

— Коллекции UML-диаграмм различных типов.

— Наборы блок-схем алгоритмов.

— Схемы баз данных.

3) **Датасеты для обучения системам автоматической оценки работ:**

— Пары «отчет-оценка» для предсказания качества работы.

— Коллекции типичных ошибок в студенческих работах.

4) **Образовательные датасеты:**

— Структурированные материалы для обучения программированию.

— Примеры кода с аннотациями и объяснениями.

Собранные в процессе функционирования системы данные могут быть использованы для:

— Развития методов автоматической генерации документации.

— Исследований в области распознавания диаграмм и схем.

— Разработки и обучения моделей автоматической проверки кода и технической документации.

— Создания интеллектуальных систем поддержки образовательного процесса.

— Анализа паттернов студенческих ошибок и оптимизации учебных программ.

Таким образом, внедрение разработанного программно-алгоритмического комплекса не только решит задачи организации совместной работы над проектами, но и создаст ценную инфраструктуру для формирования уникальных датасетов, необходимых для современных исследований в области искусственного интеллекта и обработки данных.

4.2 Исследование характеристик разработанного комплекса

Для оценки производительности разработанной системы было проведено исследование нагрузочных характеристик с использованием инструмента Artillery.

Технические характеристики машины, на которой производились исследования:

— операционная система: macOS Sequoia 15.4 [19];

— оперативная память: 36 Гб;

— процессор: Apple M3 Pro [20];

— количество ядер: 12.

В рамках исследования имитировались действия пользователей, включая:

1) регистрацию и вход в систему;

2) создание файла в проекте;

3) получение файла по ID;

4) создание блока контента.

Замеры проводились по метрике среднего времени отклика на каждую из операций. Количество одновременных пользователей варьировалось от 10 до 200. Были выбраны следующие значения нагрузки: 10, 15, 20, 30, 40, 50, 70, 90, 100, 140, 170 и 200 пользователей.

На рисунке 4.1 представлена более подробная диаграмма компонентов.

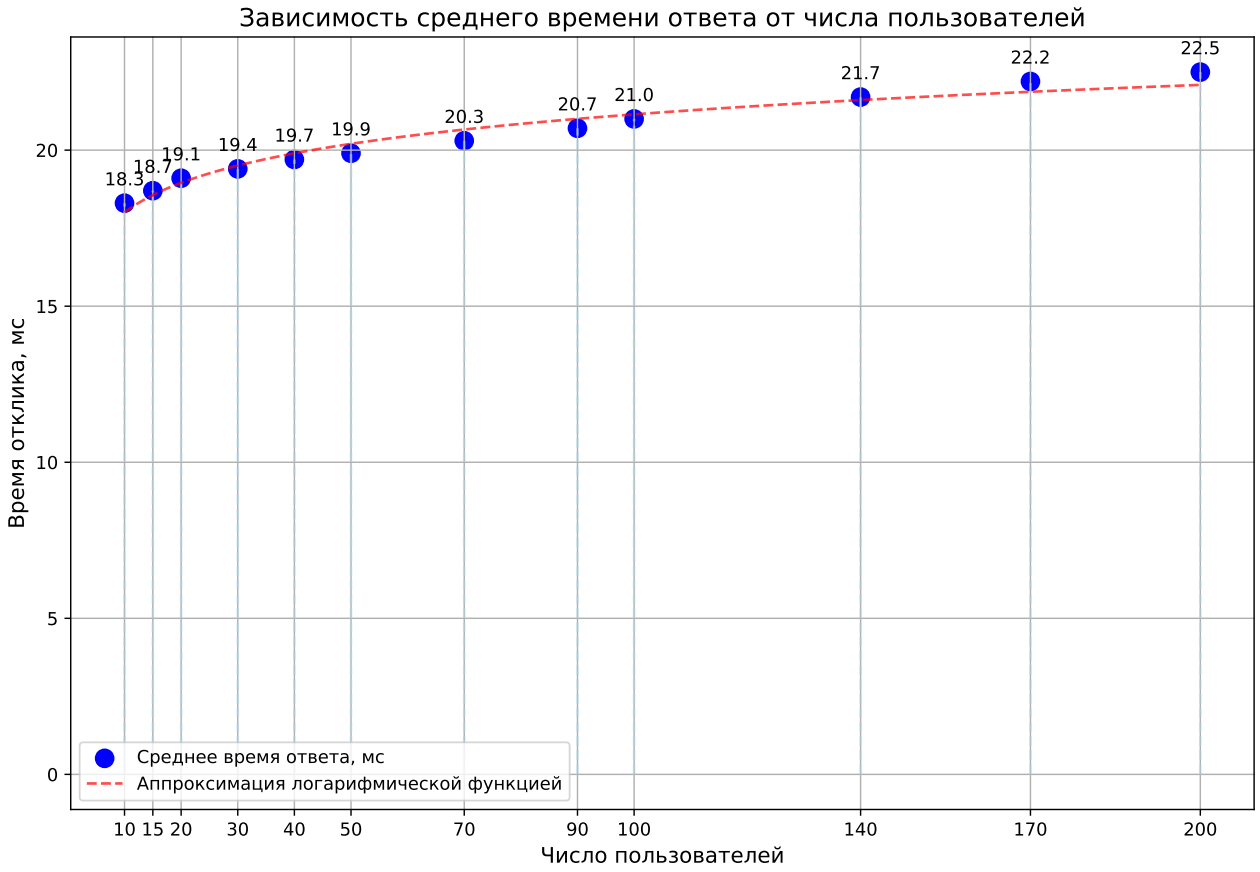

Рисунок 4.1 – Зависимость среднего времени ответа от числа пользователей

График показывает, что среднее время отклика системы увеличивается с ростом числа пользователей, однако этот рост происходит нелинейно и приближен к логарифмическому типу зависимости. При 20-кратном увеличении нагрузки (с 10 до 200 пользователей) среднее время отклика увеличилось только на 4.2 мс (с 18.3 до 22.5 мс), что составляет примерно 23%. Это свидетельствует о достаточной устойчивости и масштабируемости архитектуры при увеличении нагрузки.

Вывод

Система обладает значительным потенциалом для формирования структурированных датасетов — при внедрении в образовательный процесс кафедры может быть накоплено около 15 000 документов в год, что представляет ценность для исследований в области искусственного интеллекта. Исследование нагрузочных характеристик продемонстрировало масштабируемость системы: при 20-кратном увеличении нагрузки время отклика выросло лишь на 23%, что свидетельствует об оптимальности выбранных архитектурных решений. Логарифмический характер зависимости времени отклика от числа пользователей гарантирует стабильную работу даже при значительном росте пользовательской базы.

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы был разработан программно-алгоритмический комплекс с многоцелевой и масштабируемой архитектурой, предназначенный для совместной работы и управления проектами. Разработка велась с учетом потребности в универсальных платформах, способных обеспечить комплексную поддержку командной деятельности в условиях современных форматов работы. Особое внимание было уделено возможности хранения и управления неоднородными данными, генерируемыми пользователями, что открывает перспективы для их дальнейшего использования в исследовательских целях.

В соответствии с поставленной целью и задачами работы было выполнено следующее:

1) Были рассмотрены современные системы хранения данных, включая их классификацию и особенности работы с различными типами данных, проанализированы методы сетевого многопользовательского взаимодействия, сформулированы и формализованы исходные задачи и требования к разрабатываемому комплексу.

2) Предложена многоуровневая архитектура серверного приложения (контроллеры, сервисы, слой доступа к данным) и структура клиентского SPA-приложения, описаны основные компоненты системы, их взаимодействие, проработаны ключевые структуры данных с использованием гибридной модели хранения: PostgreSQL для реляционных метаданных и MongoDB для хранения контента.

3) Выбран технологический стек, включающий Kotlin и Spring Boot для серверной части, TypeScript и React для клиентской, разработаны основные функциональные модули системы, проведено функциональное тестирование реализованных модулей и представлены примеры модульных тестов для серверной части, описано взаимодействие пользователя с программным обеспечением.

4) Было проведено исследование в ходе которого было определено, что реализованный функционал соответствует основным поставленным задачам и обеспечивает базовые возможности для совместной работы,

заложенная архитектура и модель данных создают основу для дальнейшего расширения и добавления новых типов сервисов, система имеет значительный потенциал для формирования структурированных датасетов.

К достоинствам разработанного программно-алгоритмического комплекса можно отнести:

— Гибридность модели хранения и организацию моделей, позволяющих управлять разнообразными типами контента и расширять систему новыми сервисами.

— Модульность и расширяемость на стороне сервера и на стороне клиента.

— Формирование наборов данных, которые могут быть использованы для дальнейших исследований.

К недостаткам и областям для дальнейшего развития относятся:

— Ограниченный набор реализованных сервисов.

— Отсутствие полнофункционального механизма совместного редактирования в реальном времени для всех пользователей.

— Необходимость развертывания и интеграции комплекса на серверной инфраструктуре университета.

Поставленная цель работы — разработка программно-алгоритмического комплекса с многоцелевой и масштабируемой архитектурой для совместной работы и управления проектами — была достигнута. Разработанное программное обеспечение решает основные поставленные задачи, закладывает фундамент для дальнейшего развития и может служить основой для создания полнофункциональной платформы для командной работы, а также для формирования исследовательских наборов данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *С. А. Нестеров*. Базы данных: учеб. пособие. — СПб.: Издательство Политехнического универстета, 2013. — С. 150.
2. *Н. Р. Бухараев*. Введение в реляционные базы данных и программирование на языке SQL. — Казань: Казанский университет, 2018. — С. 134.
3. *Ю. Л. Назаренко*. Обзор технологии "большие данные"(Big Data) и программно-аппаратных средств, применяемых для их анализа и обработки. — Донской государственный технический университет, 2016. — С. 7.
4. *Р. Э. Мамедли*. Системы управления базами данных: учебное пособие. — Нижневаторск : Издательство Нижневартовского государственного университета, 2021. — С. 214.
5. MongoDB Documentation. — [Электронный ресурс]. — Режим доступа: `https://www.mongodb.com/docs/manual/` (дата обращения: 03.11.2024).
6. Cassandra Documentation. — [Электронный ресурс]. — Режим доступа: `https://cassandra.apache.org/doc/latest/` (дата обращения: 03.11.2024).
7. PostgreSQL Documentation. — [Электронный ресурс]. — Режим доступа: `https://www.postgresql.org/docs/` (дата обращения: 03.11.2024).
8. MySQL Documentation. — [Электронный ресурс]. — Режим доступа: `https://dev.mysql.com/doc/` (дата обращения: 03.11.2024).
9. *Wen Tao*. Data Aggregation: Encyclopedia of Big Data. — Springer International Publishing, Cham, 2020.
10. *С. Д. Кузнецов*, *А. В. Посконин*. Распределенные горизонтально масштабируемые решения для управления данными. — Труды Института системного программирования РАН, 2013.
11. *A. Silberschatz*, *F. Korth Henry*, *S. Sudarshan*. DATABASE SYSTEM CONCEPTS, SEVENTH EDITION. — McGraw-Hill Education, 2020. — С. 1337.

12. Документация PostgreSQL: Replication. — [Электронный ресурс]. — Режим доступа: https://www.postgresql.org/docs/current/runtime-config-replication.html (дата обращения: 20.11.2024).

13. Citus Data Documentation. — [Электронный ресурс]. — Режим доступа: https://docs.citusdata.com/en/stable/ (дата обращения: 20.11.2024).

14. Postgres-XL Documentation. — [Электронный ресурс]. — Режим доступа: https://omnidb.readthedocs.io/en/2.17.0/en/18_postgres-xl.html (дата обращения: 20.11.2024).

15. PgPool Documentation. — [Электронный ресурс]. — Режим доступа: https://www.pgpool.net/mediawiki/index.php/Main_Page (дата обращения: 20.11.2024).

16. PgBouncer Documentation. — [Электронный ресурс]. — Режим доступа: https://www.pgbouncer.org/config.html (дата обращения: 20.11.2024).

17. Учебный план ИУ7 09.03.04 на 2025 год. — [Электронный ресурс]. — Режим доступа: https://eu.bmstu.ru/ref/uchplan/ (дата обращения: 07.05.2025).

18. Летняя сессия 2025 года. — [Электронный ресурс]. — Режим доступа: https://eu.bmstu.ru/modules/session/ (дата обращения: 07.05.2025).

19. macOS Sequoia | Apple Developer Documentation. — [Электронный ресурс]. — Режим доступа: https://www.apple.com/macos/macos-sequoia/ (дата обращения: 20.04.2025).

20. M3 Pro | Apple Developer Documentation. — [Электронный ресурс]. — Режим доступа: https://developer.apple.com/videos/play/tech-talks/111375 (дата обращения: 20.04.2025).

ПРИЛОЖЕНИЕ А

В листингах А.1–А.14 представлена реализация моделей в серверной части приложения.

Листинг А.1 – Модель проекта

```
@Entity
@Table(name = "projects")
data class Project(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long = 0,
    var name: String,

    var date: LocalDateTime = LocalDateTime.now(),

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "user_id", nullable = false)
    var owner: User? = null,
}
```

Листинг А.2 – Модель связи проекта и файлов

```
@Entity
@Table(name = "projects_files")
data class ProjectFile(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long = 0,

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "project_id", nullable = false)
    val project_id: Project,

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "file_id", nullable = false)
    val file_id: File
)
```

Листинг А.3 – Модель файла

```
@Entity
@Table(name = "files")
data class File(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long = 0,

    var name: String,

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "type_id", nullable = false)
    var type_id: FileType?,

    @Column(name = "author")
    val authorId: Long,

    @Column(name = "upload_date")
    var date: LocalDateTime = LocalDateTime.now(),

    @Column(name = "super_object_id")
    var superObjectId: String? = null,

)

@Entity
@Table(name = "file_types")
data class FileType(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long = 0,
    val name: String
)
```

Листинг А.4 – Модель пользователя

```
@Entity
@Table(name = "users")
data class User(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long = 0,

    var name: String,
    var surname: String,

    @Column(unique = true, nullable = false)
    var login: String,

    @Column(nullable = false)
    var passwordInternal: String,

    var photo: String? = null,

    var enabledInternal: Boolean = true,
    var accountNonExpiredInternal: Boolean = true,
    var accountNonLockedInternal: Boolean = true,
    var credentialsNonExpiredInternal: Boolean = true
) : UserDetails {

    override fun getAuthorities(): Collection<GrantedAuthority>
      = emptyList()
    override fun getPassword(): String = this.passwordInternal
    override fun getUsername(): String = this.login
    override fun isAccountNonExpired(): Boolean =
      this.accountNonExpiredInternal
    override fun isAccountNonLocked(): Boolean =
      this.accountNonLockedInternal
    override fun isCredentialsNonExpired(): Boolean =
      this.credentialsNonExpiredInternal
    override fun isEnabled(): Boolean = this.enabledInternal
}
```

Листинг А.5 – Модель связи проекта и пользователя

```
@Entity
@Table(name = "projects_users")
data class ProjectUser(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long = 0,

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "project_id", nullable = false)
    val project_id: Project,

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "user_id", nullable = false)
    val user_id: User,

    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    var role: ProjectRole
)
```

Листинг А.6 – Модель блока данных

```
@Document(collection = "content_blocks")
data class ContentBlock(
    @Id
    var id: String? = null,
    var objectType: String? = null,

    var data: Map<String, Any>? = null,

    var nextItem: String? = null,
    var prevItem: String? = null,
)
```

Листинг А.7 – Модель суперобъекта

```
@Document(collection = "super_objects")
data class SuperObject(
    @Id
    var id: String? = null,
    @Indexed(unique = true)
    var fileId: Long? = null,
    var serviceType: String? = null,
    var lastChangeDate: String? = null,
    var name: String? = null,
    var firstItem: String? = null,
    var lastItem: String? = null,
    var stylesMapId: String? = null
)
```

Листинг А.8 – Модель стилей

```
@Document(collection = "styles")
data class Style(
    @Id
    var id: String? = null,
    var targetType: String?,
    var attributes: Map<String, Any>? = null,
)
```

Листинг А.9 – Модель связи стилей и блоков

```
@Document(collection = "styles_map")
data class StylesMap(
    @Id
    var id: String? = null,
    var links: List<StyleLink>? = null
)

data class StyleLink(
    var elementId: String? = null,
    var styleId: String? = null,
    var scope: String? = null,
    var state: String? = null
)
```

Листинг А.10 – Модель блока схемы

```
@Document(collection = "scheme_node")
data class SchemeNode(
    @Id
    var id: String? = null,
    var superObjectId: String? = null,
    var type: String? = null,
    var x: Int? = null,
    var y: Int? = null,
    var width: Int? = null,
    var height: Int? = null,
    var data: Map<String, Any>? = null
)
```

Листинг А.11 – Модель связей блоков схемы

```
@Document(collection = "scheme_edge")
data class SchemeEdge(
    @Id
    var id: String? = null,
    var superObjectId: String? = null,
    var type: String? = null,
    var sourceNodeId: String? = null,
    var targetNodeId: String? = null,
    var data: Map<String, Any>? = null
)
```

Листинг А.12 – Модель события календаря

```
@Document(collection = "calendar_events")
data class CalendarEvent(
    @Id
    var id: String? = null,
    var superObjectId: String? = null,
    var title: String? = null,
    var description: String? = null,
    var startTime: Date? = null,
    var endTime: Date? = null,
    var participants: List<Int>? = null
)
```

Листинг А.13 – Модель задачи

```
@Document(collection = "task_item")
data class TaskItem(
    @Id
    var id: String? = null,
    var superObjectId: String? = null,
    var columnId: String? = null,
    var title: String? = null,
    var description: String? = null,
    var orderInColumn: Int? = null,
    var assigneeId: Int? = null,
    var reporterId: Int? = null,
    var priority: String? = null,
    var dueDate: Date? = null,
    var tags: List<String>? = null,
    var attachments: List<Int>? = null,
    var subtasks: List<String>? = null
)
```

Листинг А.14 – Модель колонки задач

```
@Document(collection = "task_column")
data class TaskColumn(
    @Id
    var id: String? = null,
    var superObjectId: String? = null,
    var name: String? = null,
    var order: Int? = null
)
```

ПРИЛОЖЕНИЕ Б

В листингах Б.1–Б.5 представлена реализация редактора в клиентской части приложения.

Листинг Б.1 – Редактор заметки часть 1

```
import React, { useEffect, useRef, memo } from 'react';
import EditorJS, { OutputData, API as EditorJSAPI,
   BlockToolConstructable, ToolSettings } from
   '@editorjs/editorjs';
import Header from '@editorjs/header';
import List from '@editorjs/list';
import Paragraph from '@editorjs/paragraph';
import Underline from '@editorjs/underline';
import Marker from '@editorjs/marker';
import Strikethrough from 'editorjs-strikethrough';
import { StyleInlineTool } from 'editorjs-style';
import ImageTool from '@editorjs/image'
import Quote from '@editorjs/quote';
import CodeTool from '@editorjs/code';

interface EditorTools {
    [toolName: string]: BlockToolConstructable | ToolSettings;
}

interface UploadResponseFormat {
  success: 1 | 0;
  file: {
    url: string;
    name?: string;
    size?: number;
  };
  message?: string;
}

interface EditorJsWrapperProps {
    holderId: string;
    initialData?: OutputData;
    onChange?: (api: EditorJSAPI, newData: OutputData) => void;
    onReady?: (editor: EditorJS) => void;
    readOnly?: boolean;
```

Листинг Б.2 – Редактор заметки часть 2

```
    imageUploader?: {
      uploadByFile: (file: File) =>
        Promise<UploadResponseFormat>;
      uploadByUrl?: (url: string) =>
        Promise<UploadResponseFormat>;
  };
}

const EditorJsWrapper: React.FC<EditorJsWrapperProps> = ({
  holderId,
  initialData,
  onChange,
  onReady,
  readOnly = false,
  imageUploader,
}) => {
  const editorInstanceRef = useRef<EditorJS | null>(null);
  const holderRef = useRef<HTMLDivElement | null>(null);

  useEffect(() => {
    if (editorInstanceRef.current &&
       editorInstanceRef.current.readOnly &&
        typeof editorInstanceRef.current.readOnly.toggle ===
          'function') {
      editorInstanceRef.current.readOnly.toggle(readOnly);
    }
  }, [readOnly, holderId]);


  useEffect(() => {
    if (!holderRef.current) {
      return;
    }

    if (editorInstanceRef.current) {
        if (editorInstanceRef.current.readOnly && typeof
          editorInstanceRef.current.readOnly.toggle ===
          'function') {
           editorInstanceRef.current.readOnly.toggle(readOnly);
        }
```

Листинг Б.3 – Редактор заметки часть 3

```
    const toolsConfig: EditorTools = {
      paragraph: { class: Paragraph as any, inlineToolbar: true
         },
      header: { class: Header as any, inlineToolbar: true,
         config: { levels: [1, 2, 3, 4], defaultLevel: 2 } },
      list: { class: List as any, inlineToolbar: true },
      underline: { class: Underline as any, shortcut: 'CMD+U' },
      strikethrough: { class: Strikethrough as any, shortcut:
         'CMD+SHIFT+S' },
      marker: { class: Marker as any, shortcut: 'CMD+SHIFT+M' },
      style: {
        class: StyleInlineTool as any,
        config: { style: [ 'color', 'background-color',
           'font-size', 'font-family', 'border', 'text-align' ]
          },
      },
      image: {
        class: ImageTool as any,
        config: {
          uploader: imageUploader ? {
            uploadByFile: imageUploader.uploadByFile,
            uploadByUrl: imageUploader.uploadByUrl,
          } : {
            uploadByFile: (file: File) => {
              return new Promise<UploadResponseFormat>((resolve,
                 reject) => {
                setTimeout(() => {
                  const reader = new FileReader();
                  reader.onloadend = () => {
                    resolve({
                      success: 1,
                      file: { url: reader.result as string },});
                  };
                  reader.onerror = reject;
                  reader.readAsDataURL(file);
                }, 1000);
              });
            },
            uploadByUrl: (url: string) => {
```

Листинг Б.4 – Редактор заметки часть 4

```
                return Promise.resolve({
                    success: 1,
                    file: { url: url },
                });
            }
        },
        types: 'image/png, image/jpeg, image/gif, image/webp,
           image/svg+xml',
      },
    },
    quote: {
      class: Quote,
      inlineToolbar: true,
      shortcut: 'CMD+SHIFT+O',
      config: {
        quotePlaceholder: 'Введите цитату',
        captionPlaceholder: 'Автор цитаты',
      },
    },
    code: {
      class: CodeTool,
      shortcut: 'CMD+SHIFT+C',
      config: {
        placeholder: 'Введите код',
      },
    },
    table: {
      class: Table as any,
      inlineToolbar: true,
      config: {
        rows: 2,
        cols: 3,
      },
    },
    embed: {
      class: Embed,
      config: {
        services: {
            youtube: true,
            coub: true,
```

Листинг Б.5 – Редактор заметки часть 5

```
              github: true,
            } } }, };
    const editor = new EditorJS({
      holder: holderRef.current,
      placeholder: 'Начните вводить текст или выберите блок...',
      readOnly: readOnly,
      i18n: editorJsRussianLocale,
      data: initialData || { blocks: [] },
      onReady: () => {
        editorInstanceRef.current = editor;
        if (onReady) {
          onReady(editor);
        }
      },
      onChange: async (api, event) => {
        if (onChange && editorInstanceRef.current === editor) {
          const savedData = await
             editorInstanceRef.current.save();
          onChange(api, savedData);
        }
      },
      tools: toolsConfig,
    });
    return () => {
      const editorToDestroy = editor;
      const currentGlobalInstance = editorInstanceRef.current;
      if (typeof editorToDestroy.destroy === 'function') {
        editorToDestroy.destroy();
      }
      if (currentGlobalInstance === editorToDestroy) {
        editorInstanceRef.current = null;
      }
    };
  }, [holderId, onChange, onReady, imageUploader]);
  return <div ref={holderRef} id={holderId} style={{ border:
     '1px solid #ccc', minHeight: '200px' }} />;
};
export default memo(EditorJsWrapper);
```

ПРИЛОЖЕНИЕ В

На рисунках В.1–В.5 представлен интерфейс приложения.

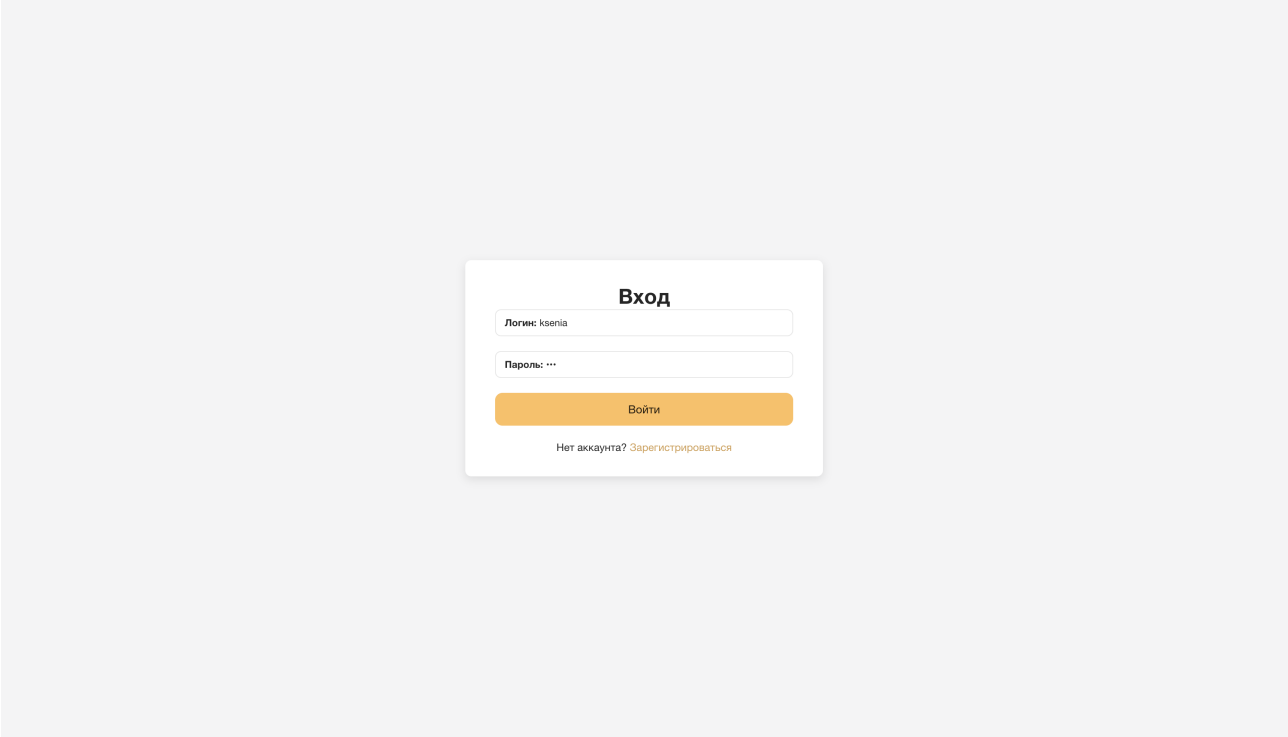

Рисунок В.1 – Экран входа в систему

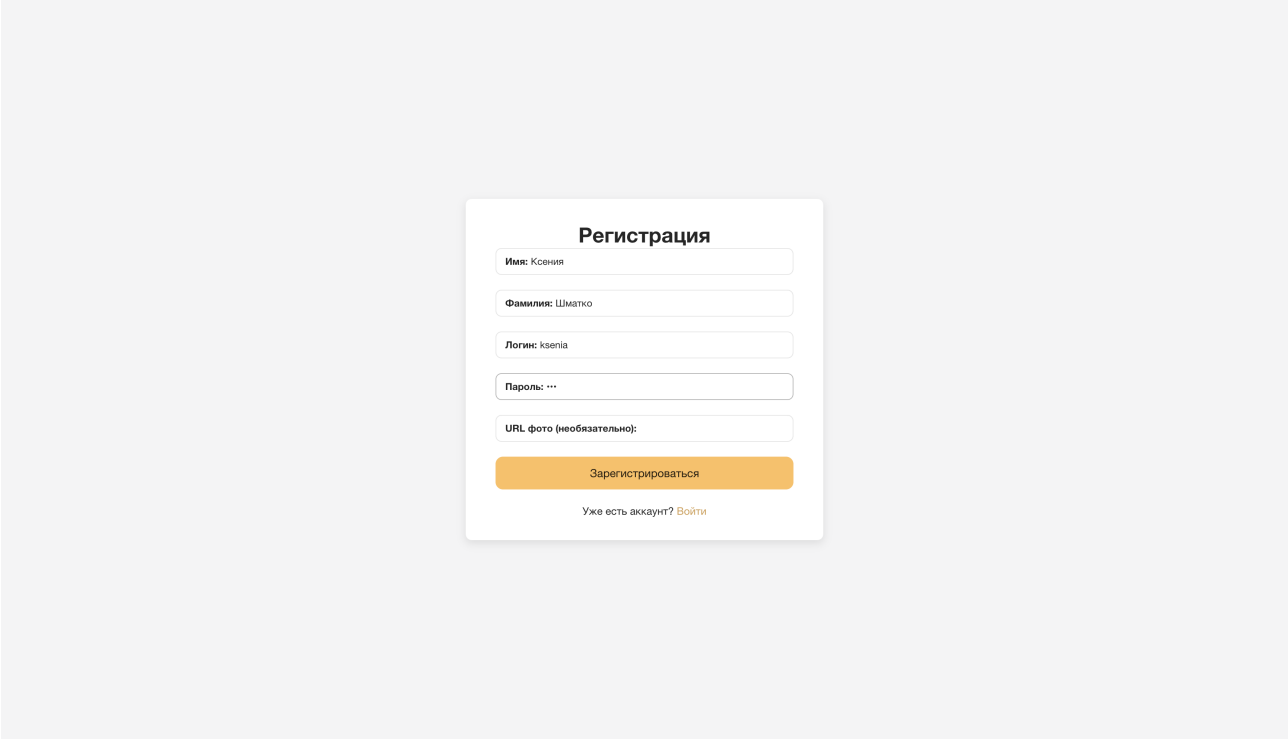

Рисунок В.2 – Экран регистрации в системе

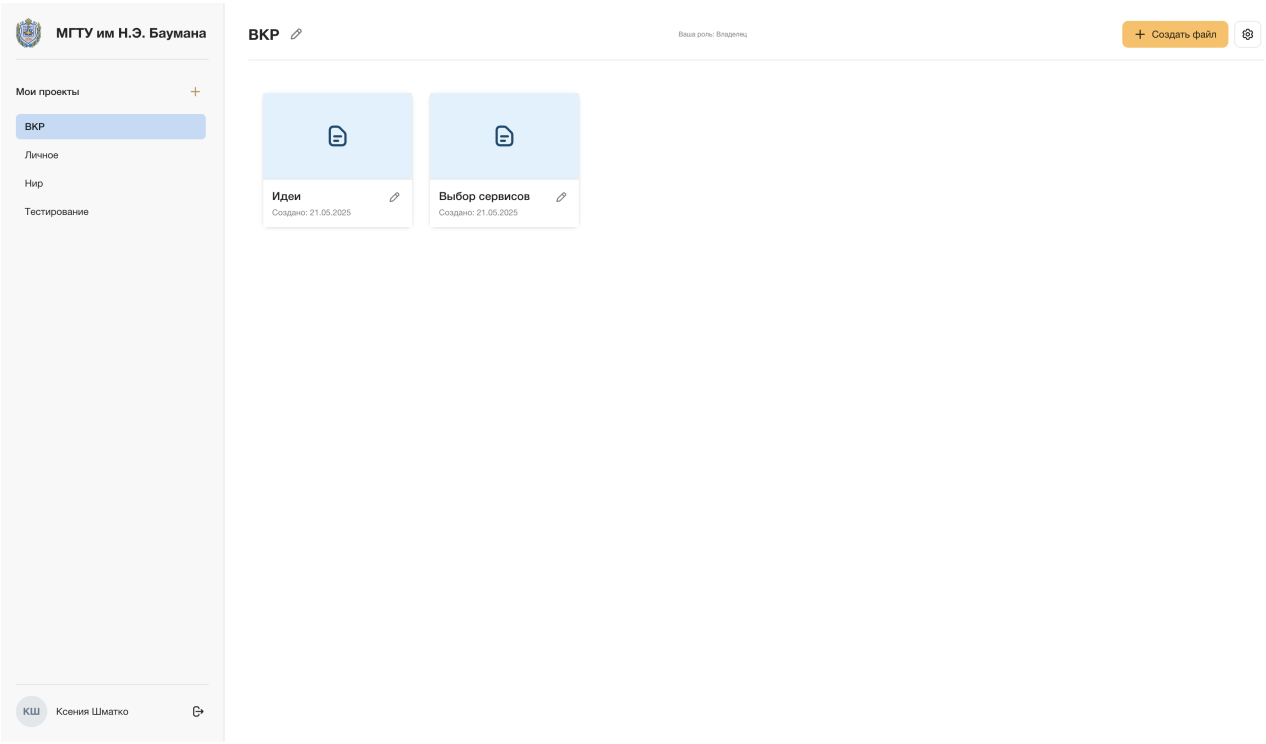

Рисунок В.3 – Экрана выбора проекта и просмотра файлов

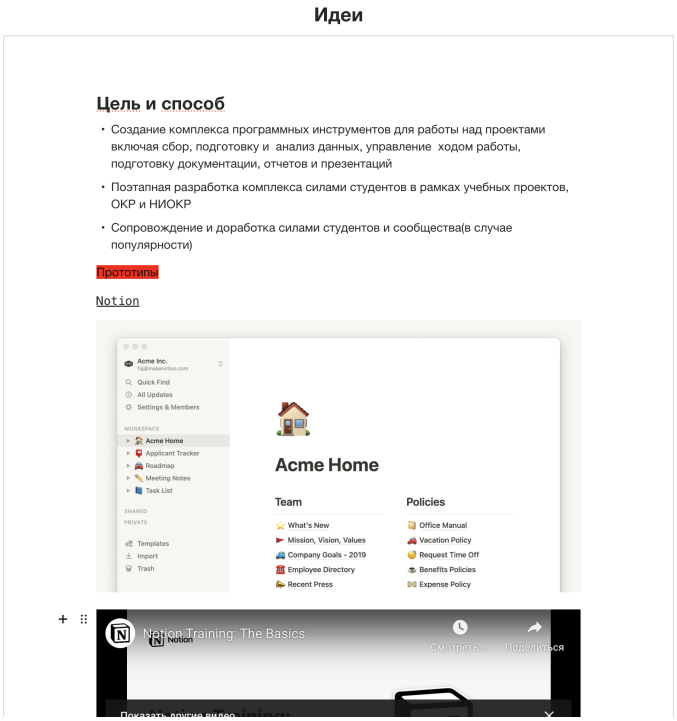

Рисунок В.4 – Экрана редактирования заметки

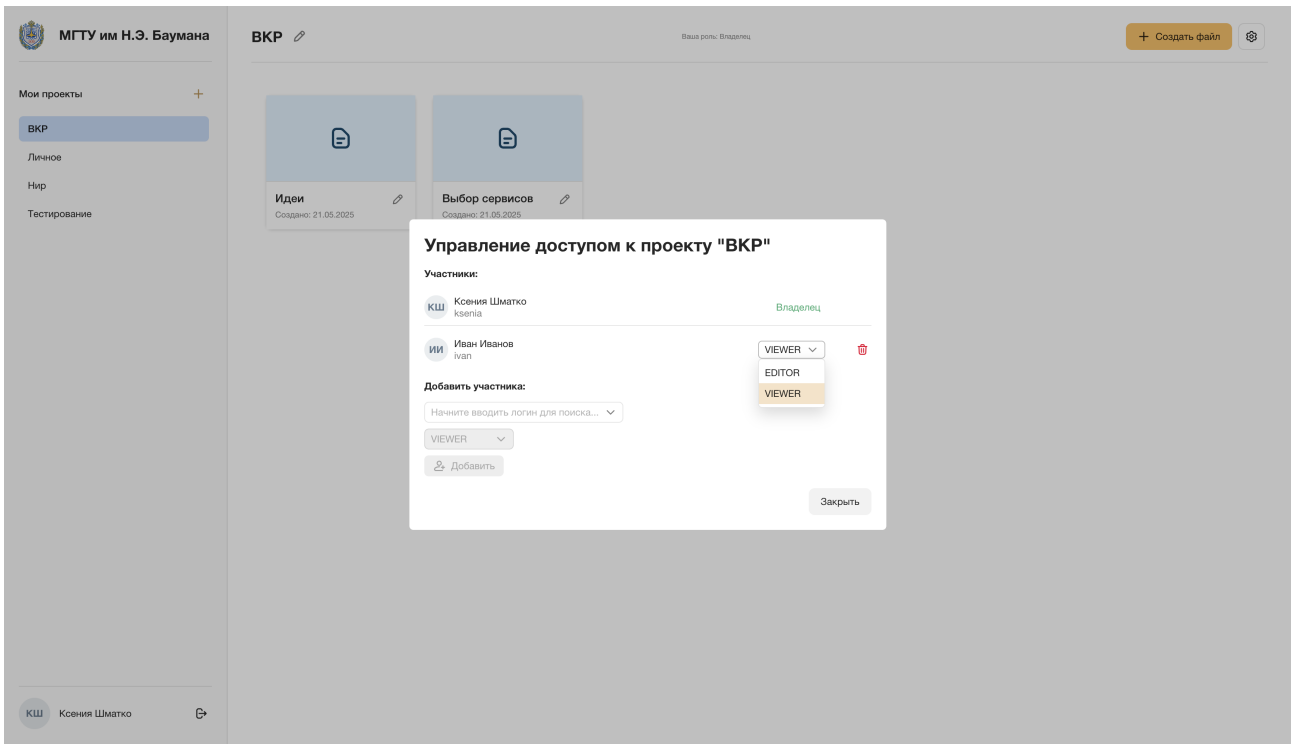

Рисунок В.5 – Экрана управления ролями в проекте

ПРИЛОЖЕНИЕ Г

Презентация к выпускной квалификационной работе состоит из 17 слайдов.