Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное 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 (Группа) _____ (Подпись, дата) Шматко К. М. (И. О. Фамилия)

Руководитель ВКР _____ (Подпись, дата) Никульшин А. М. (И. О. Фамилия)

2025 г.

РЕФЕРАТ

Расчетно-пояснительная записка 97 с., 18 рис., 1 табл., 16 источн., 3 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.

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. Это важно для обеспечения целостности данных, 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. Это упрощает реализацию функциональности, связанной с ролями доступа и управления проектами.

3) **Безопасность и транзакционность**. Реляционные базы данных поддерживают ACID-транзакции, что гарантирует надёжность и консистентность данных, особенно важных в операциях, таких как регистрация новых пользователей и управление их аутентификацией.

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),
    -- Дополнительные поля
);
```

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

2) **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].

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

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 [15][16].

Позволяют распределять запросы между основным сервером и репликами для оптимального использования ресурсов.

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

1.4.1 Архитектуры взаимодействия

Существуют две основные архитектуры для построения сетевого взаимодействия.

Централизованная (Клиент-Серверная)

Наиболее распространенная архитектура 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.

Децентрализованная (Peer-to-Peer, P2P)

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.

1.4.2 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) **База знаний** – формирование и поддержка базы знаний проекта.
5) **Графики и диаграммы** – визуализация данных в виде графиков и диаграмм.
6) **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.

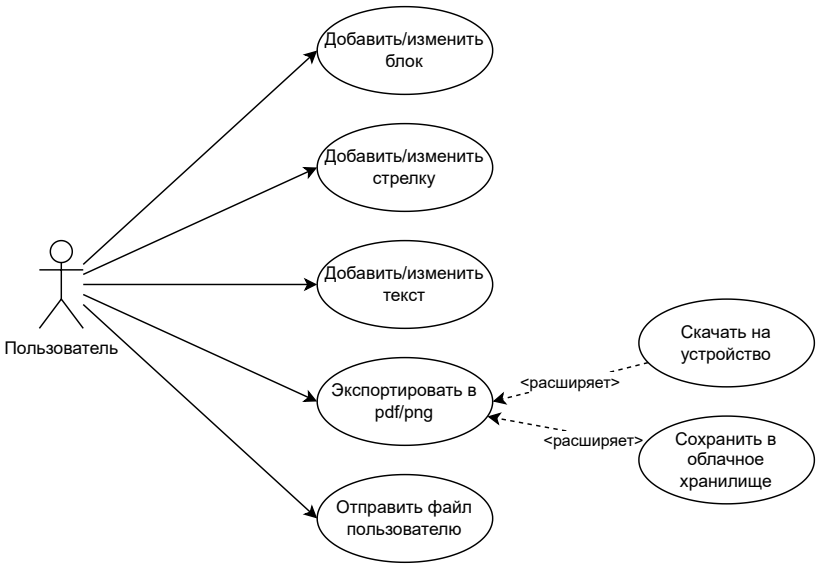

Рисунок 1.3 – Диаграмма вариантов использования схем и mind map

На рисунке 1.4 представлена диаграмма вариантов использования календаря.

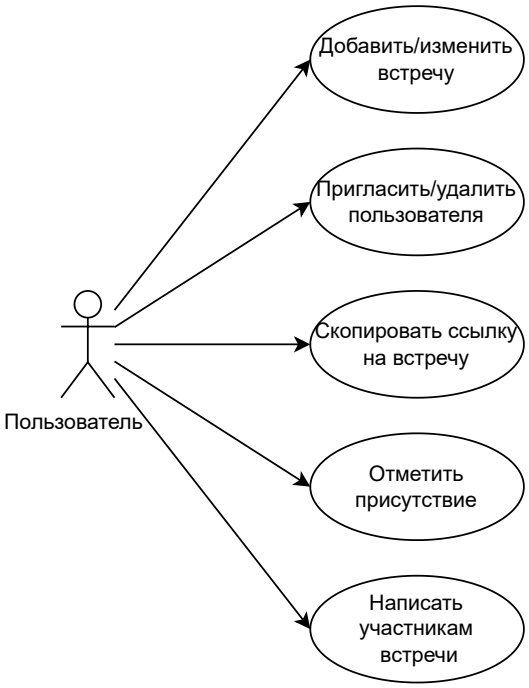

Рисунок 1.4 – Диаграмма вариантов использования календаря

На рисунке 1.5 представлена диаграмма вариантов использования трекера задач.

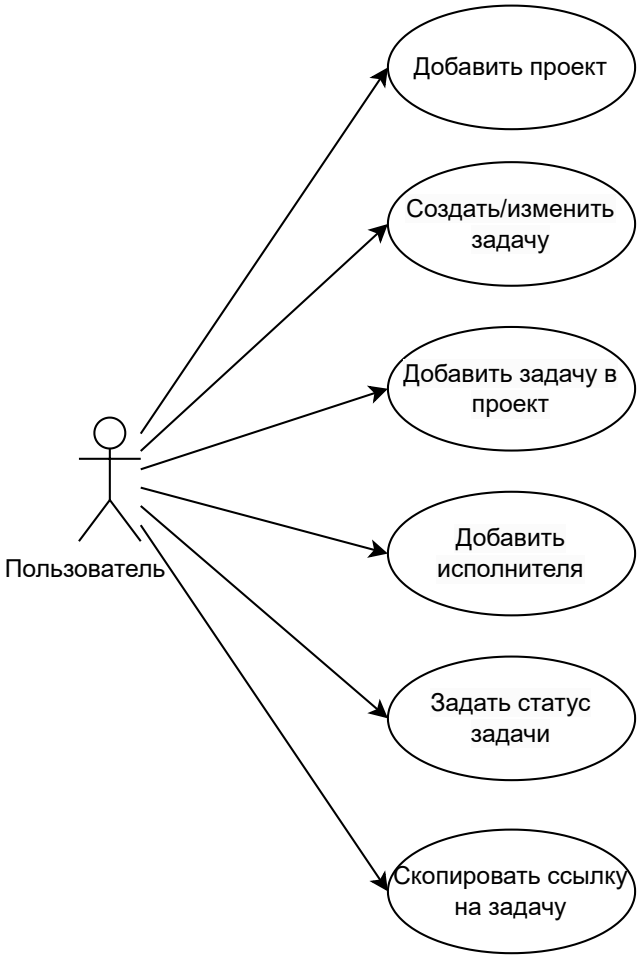

Рисунок 1.5 – Диаграмма вариантов использования трекера задач

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 программно-алгоритмического комплекса

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

1) **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.

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

- **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).
- **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.

3) **Комбинированное использование REST API и WebSockets:** Для взаимодействия между клиентом и сервером применяются два основных механизма:

- **REST API** используется 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.

— **WebSocket** используется для обеспечения взаимодействия в реальном времени важного для функций совместной работы: одновременное редактирование документов, обмен сообщениями в чате, доставка мгновенных уведомлений, отображение статуса присутствия пользователей.

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. Он выступает в роли контейнера метаинформации о файле и обеспечивает связь с набором этих блоков или со специфической структурой данных, соответствующей определенному типу сервиса (например, коллекция узлов и ребер для схемы, события для календаря). Такой подход позволяет добавлять новые типы контента и управлять документами с различной внутренней структурой в рамках единого механизма.

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.

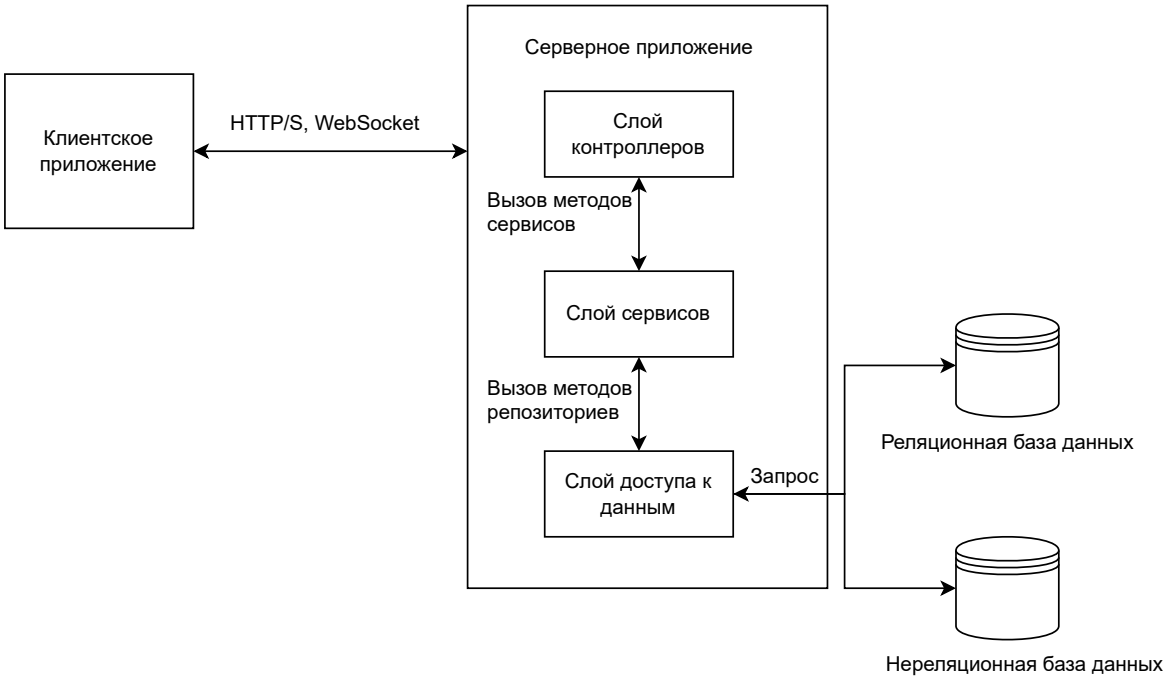

Рисунок 2.1 – Общая архитектура программно-алгоритмического комплекса

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.

На рисунке 2.2 представлена более подробная диаграмма компонентов серверного приложения.

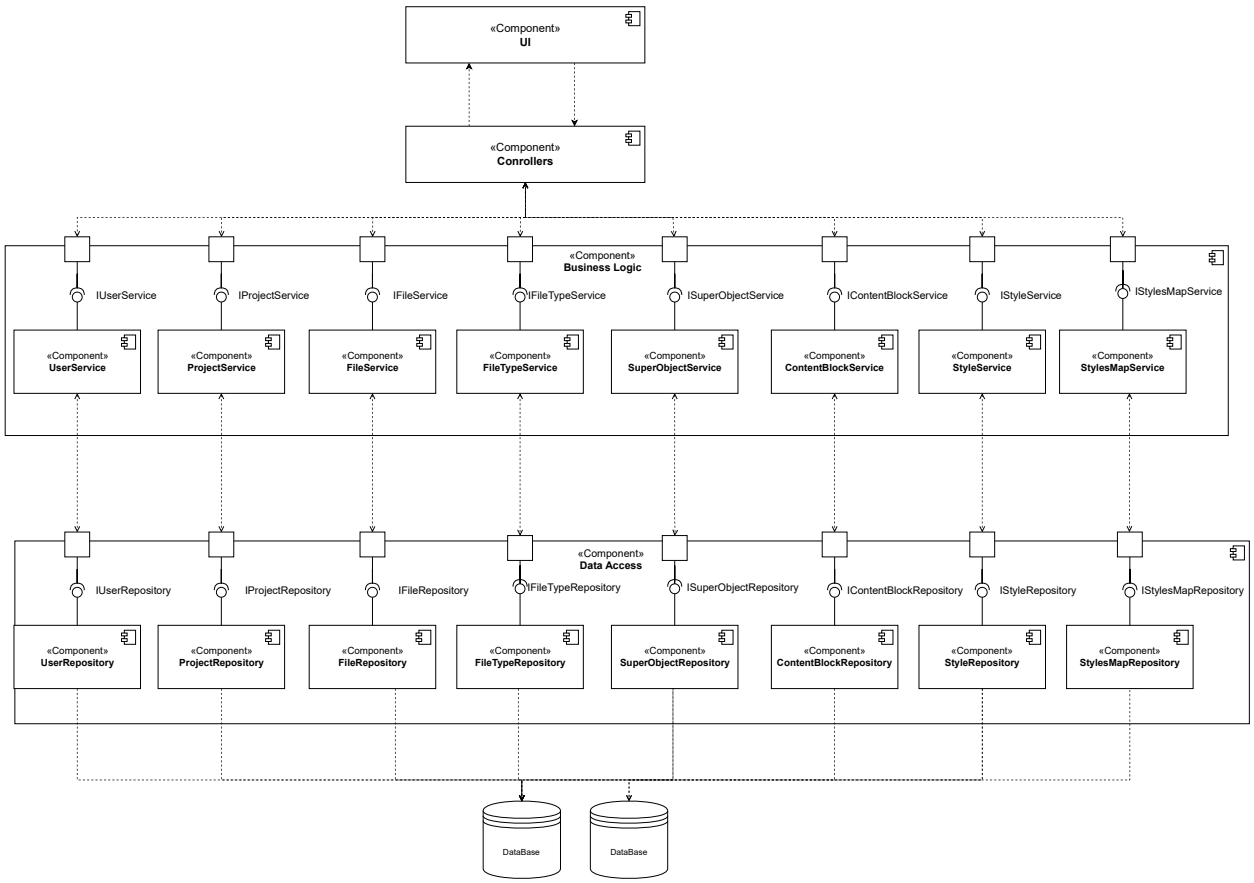

Рисунок 2.2 – Диаграмма компонентов серверного приложения

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).

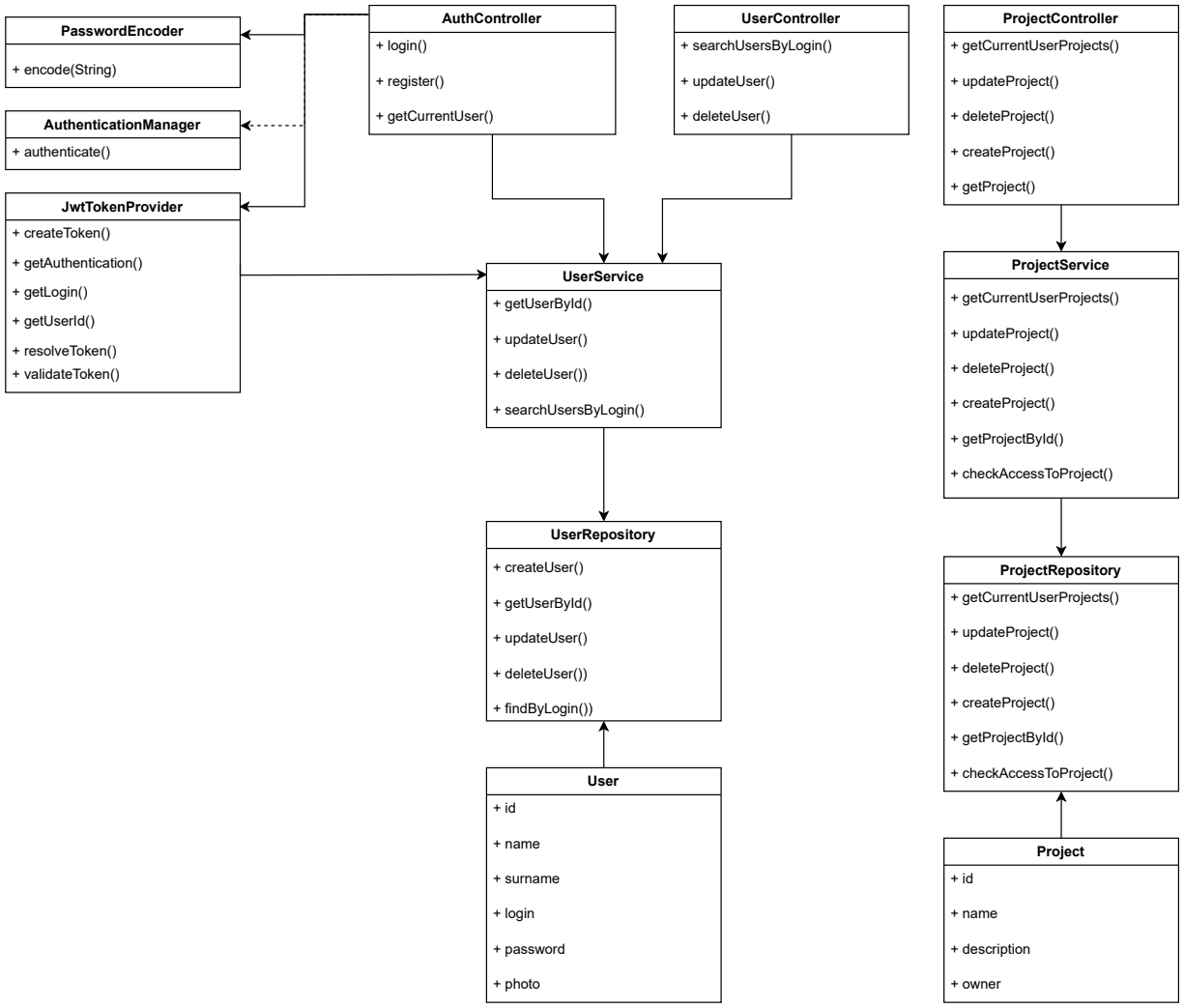

Рисунок 2.3 – Диаграмма классов для сущностей User, Project и связанных компонентов

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

2.3 Сценарий совместного редактирования

Для иллюстрации взаимодействия компонентов системы при выполнении операций в реальном времени рассмотрим сценарий совместного редактирования документа.

Процесс представлен диаграммой последовательности на рисунке 2.4.

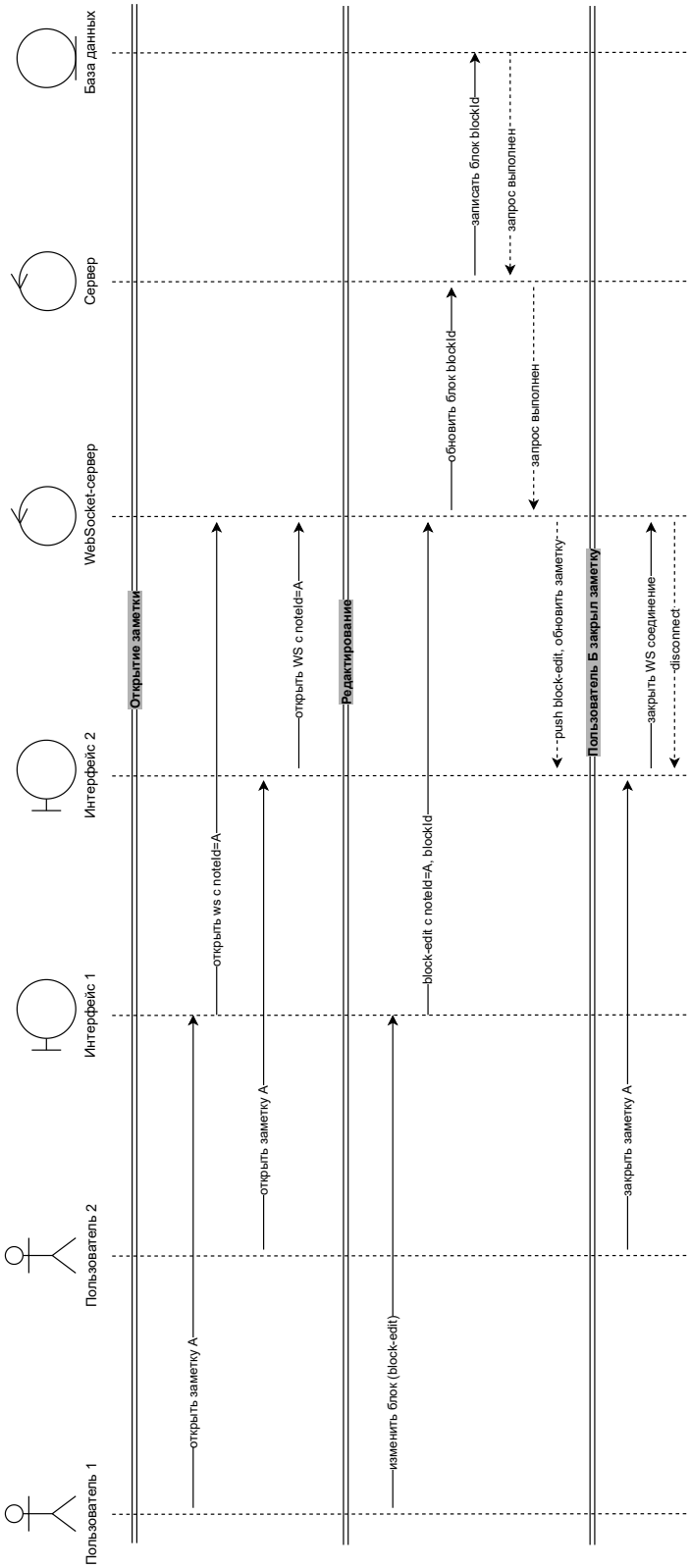

Рисунок 2.4 – Диаграмма последовательности для совместного редактирования документа

Основные шаги взаимодействия при внесении изменения пользователем *1* в документ, который также просматривает пользователь *2*:

1) Пользователь *1* выполняет действие редактирования (например, вводит символ) в клиентском приложении *1*.

2) Клиентское приложение *1* формирует сообщение об операции редактирования, указывая тип операции, позицию, измененные данные и идентификатор документа/файла, и отправляет его на сервер через установленное WebSocket-соединение.

3) Серверное приложение, в частности его компонент, отвечающий за обработку WebSocket-сообщений, принимает данное сообщение.

4) Обработчик WebSocket вызывает соответствующий метод сервисного слоя, ответственного за управление контентом, передавая ему детали операции редактирования.

5) Сервис контента проверяет полученную операцию и взаимодействует со слоем доступа к данным для сохранения изменений в соответствующей системе хранения.

6) После подтверждения успешного сохранения данных от слоя доступа к данным, компонент обработки WebSocket-сообщений получает уведомление об успешном применении операции.

7) Компонент обработки WebSocket-сообщений рассылает информацию о выполненной операции всем остальным пользователям, включая пользователя *2*, которые в данный момент работают с этим же документом и подключены к соответствующей WebSocket-сессии.

8) Клиентское приложение *2* получает сообщение об операции через WebSocket и применяет соответствующие изменения к своему локальному представлению документа, отображая актуальное состояние пользователю *2*.

Данный механизм обеспечивает синхронизацию состояния документа у всех активных пользователей в режиме реального времени.

2.4 Ключевые структуры данных

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

2.4.1 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 2.5.

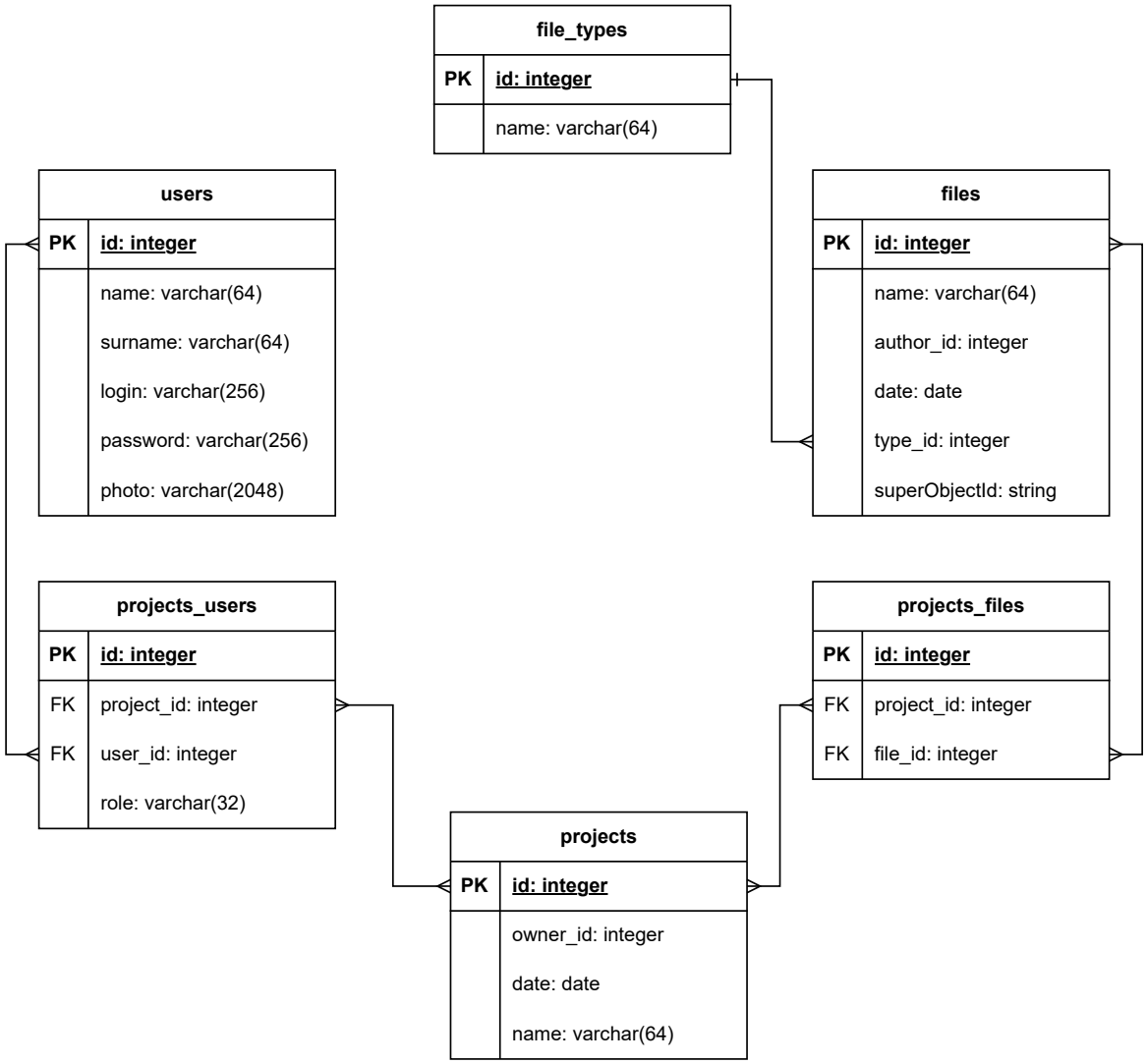

Рисунок 2.5 – Диаграмма реляционной базы данных

2.4.2 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.

Основные коллекции:

1) `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"`):
        1) `firstItem`: идентификатор первого блока контента (`ContentBlock`) в последовательности.
        2) `lastItem`: идентификатор последнего блока контента (`ContentBlock`).
    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`.
2) `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.

3) 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"`):

1) `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.

4) 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 2.6.

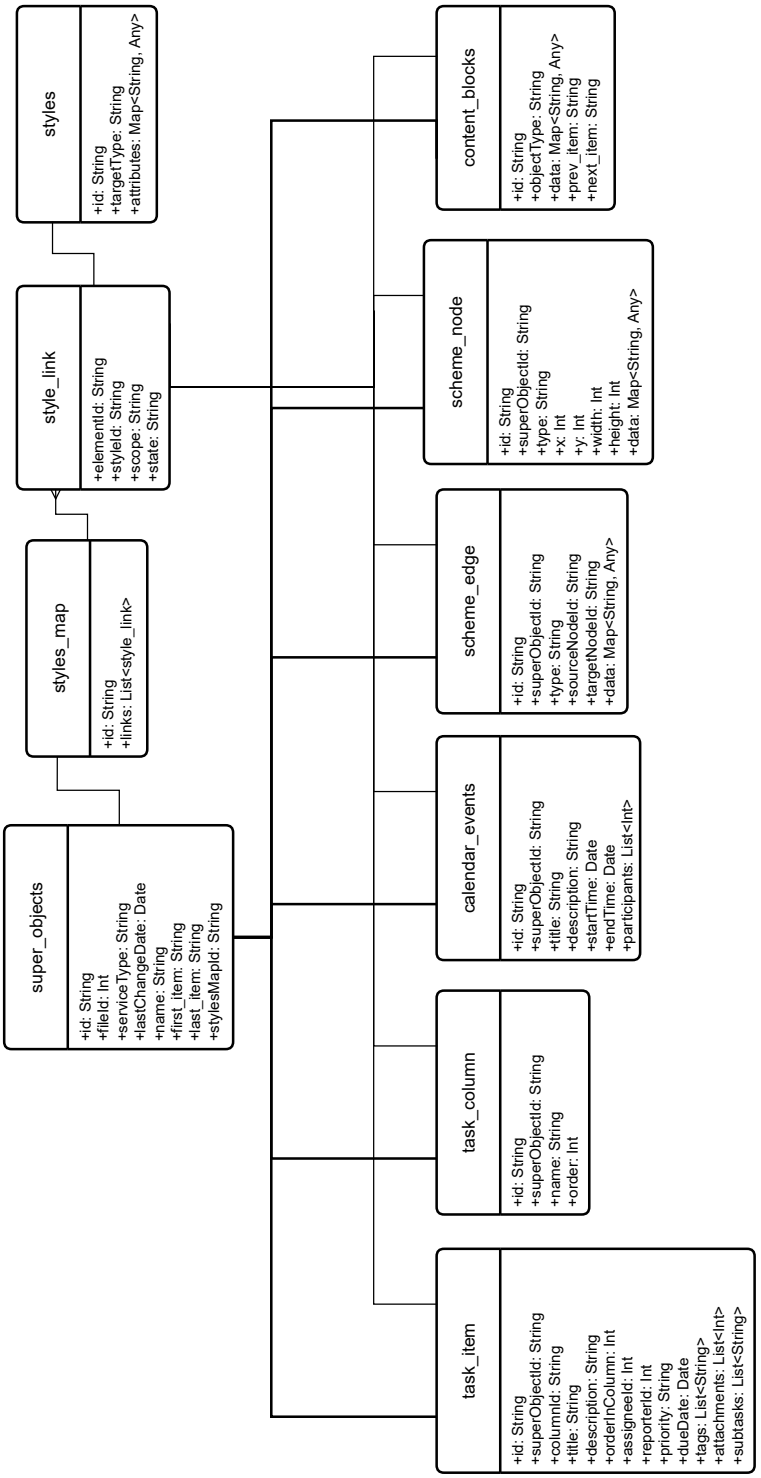

Рисунок 2.6 – Схема коллекций нереляционной базы данных

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 функциональное тестирование, а также были разработаны модульные тесты для некоторых компонентов серверной части.

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

— Регистрация и аутентификация пользователя.

— Создание и выбор проекта.

— Создание, открытие и переименование заметки.

— Добавление и базовое редактирование контента в заметке.

— Загрузка изображения в заметку.

— Добавление и удаление участников проекта, изменение их роли.

Тестирование API эндпоинтов проводилось с использованием инструмента Postman.

Было проведено модульное тестированию серверной части. В листингах 3.1 и 3.2 представлены тесты регистрации пользователя и создания проекта.

Листинг 3.1 – 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 3.2 – 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 А и Б соответственно.

3.3 Описание взаимодействия пользователя с программным обеспечением

Взаимодействие пользователя с комплексом осуществляется через веб-интерфейс, реализованный как одностраничное приложение (SPA). Далее описаны 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:** При открытии файла-заметки пользователь переходит на страницу редактора. Пользователь может:

   — Вводить и форматировать текст.

— Вставлять различные типы блоков: заголовки, списки, цитаты, изображения.

— Удалять, перемещать и изменять блоки данных.

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 В.

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. Разработанный программный комплекс является основой для дальнейшего расширения и добавления новых сервисов.

4 Исследовательский раздел

4.1 Исследование применимости разработанного программного обеспечения

Целью разработки программно-алгоритмического комплекса являлось создание системы для совместной работы над проектами с возможностью управления неоднородными данными. Исследование применимости основывается на сопоставлении реализованного функционала с исходными задачами, сформулированными ранее.

4.1.1 Соответствие основным задачам

На текущем этапе разработки комплекс успешно решает следующие ключевые задачи:

1) **Регистрация и аутентификация пользователей:** Система предоставляет функционал для создания учетных записей и безопасного входа пользователей с использованием JWT, что является базовым требованием для многопользовательских систем.

2) **Создание и управление проектами:** Пользователи могут создавать проекты, являющиеся контейнерами для дальнейшей работы. Реализовано присвоение роли владельца создателю проекта.

3) **Управление доступом к проектам:** Владелец проекта может добавлять других зарегистрированных пользователей в проект и назначать им роли («Редактор», «Читатель»), а также удалять участников. Это обеспечивает разграничение прав доступа к информации внутри проекта.

4) **Работа с файлами типа "заметка":** Реализована возможность создания, просмотра, переименования и редактирования текстовых документов (заметок) внутри проектов.

5) **Редактирование контента:** Редактор позволяет пользователям создавать структурированный контент, используя различные блоки (текст, заголовки, списки, изображения). Реализована загрузка изображений

на сервер и их отображение в редакторе. Данные редактора сохраняются в MongoDB, что соответствует требованию хранения неоднородного контента.

Данный набор реализованных функций подтверждает применимость разработанного ПО для организации базовой совместной работы с текстовыми документами в рамках проектной деятельности. Пользователи могут создавать изолированные рабочие пространства (проекты), управлять их содержимым (заметками) и контролировать доступ других участников.

4.1.2 Потенциал для реализации других сервисов

Заложенная архитектура и модель данных (в частности, концепция SuperObject и ContentBlock в MongoDB, а также таблица file_types в PostgreSQL) создают основу для последующей интеграции остальных сервисов (схем, презентаций, трекера задач и т.д.):

— **Поддержка различных типов файлов:** Система SuperObject с полем serviceType позволяет добавлять новые типы контента. Для каждого нового типа достаточно будет разработать соответствующий клиентский редактор и, при необходимости, специфическую логику обработки на сервере. Метаданные нового типа файла могут быть добавлены в таблицу file_types.

— **Гибкость хранения контента:** Использование MongoDB для SuperObject и связанных с ним блоков (или специфичных для сервиса коллекций) обеспечивает необходимую гибкость для хранения данных различной структуры, что важно для таких сервисов, как трекер задач или календарь.

Таким образом, применимость комплекса распространяется и на нереализованные на данный момент задачи, благодаря заложенной расширяемой архитектуре.

4.2 Исследование характеристик разработанного комплекса

Исследование характеристик комплекса проводится на основе анализа выбранных технологий, архитектурных решений и реализованного функционала.

4.2.1 Гибкость и расширяемость

— **Модель данных для неоднородного контента:** Принятый подход с использованием SuperObject в MongoDB как центральной сущности для представления файла (независимо от его внутреннего типа) и связанных с ним блоков ContentBlock (для блочных типов контента типа заметок или презентаций) или специфичных коллекций (для схем, календарей и т.д.) обеспечивает высокую гибкость. Система может быть расширена для поддержки новых типов документов и сервисов без значительного изменения основной архитектуры хранения. Добавление нового типа контента сведется к:

1) Определению нового значения для serviceType в SuperObject.
2) Добавлению записи в таблицу file_types (PostgreSQL).
3) Реализации соответствующего редактора на клиентской стороне.
4) При необходимости, разработке специфических коллекций в MongoDB и API-эндпоинтов для обработки данного типа контента.

— **Модульность серверного приложения:** Использование Spring Boot и разделение логики на контроллеры, сервисы и репозитории способствует модульности. Новые функциональные блоки могут быть добавлены как отдельные сервисы, минимизируя влияние на существующий код.

— **Компонентная архитектура на клиенте:** Применение React позволяет создавать переиспользуемые UI-компоненты, что упрощает добавление новых интерфейсов и модификацию существующих.

Эти факторы указывают на способность системы к расширению и адаптации под новые требования.

4.2.2 Масштабируемость

Полноценное нагрузочное тестирование не проводилось, однако архитектурные решения позволяют сделать качественную оценку потенциала масштабируемости:

— **Горизонтальная масштабируемость MongoDB:** MongoDB изначально спроектирована для горизонтального масштабирования (шардинг, репликация), что позволяет распределять нагрузку при хранении и обработке больших объемов контента документов.

— **Масштабируемость PostgreSQL:** Несмотря на то, что PostgreSQL традиционно масштабируется вертикально, существуют решения для горизонтального масштабирования, которые могут быть применены при необходимости. Метаданные обычно занимают меньше места и генерируют меньше нагрузки по сравнению с самим контентом, поэтому встроенных возможностей PostgreSQL может быть достаточно на длительный период.

— **Stateless Backend:** Серверное приложение на Spring Boot, использующее JWT для аутентификации, спроектировано как stateless. Это означает, что каждый запрос содержит всю необходимую информацию для его обработки, и состояние сессии пользователя не хранится на конкретном экземпляре сервера. Такой подход является ключевым для горизонтального масштабирования серверной части путем запуска нескольких экземпляров приложения за балансировщиком нагрузки.

Текущая реализация и выбранный стек технологий создают предпосылки для масштабирования системы при росте числа пользователей и объемов данных, однако для подтверждения потребуются специализированные нагрузочные тесты.

4.2.3 Производительность

Оценка производительности также носит качественный характер:

— **Отклик интерфейса:** Использование React с виртуальным DOM способствует созданию отзывчивого пользовательского интерфейса. Асинхронные запросы к API выполняются без блокировки основного потока. Загрузка данных для проектов и файлов происходит по мере необходимости.

— **Скорость работы с базами данных:** PostgreSQL обеспечивает быструю обработку запросов к структурированным метаданным, особенно при наличии корректных индексов. MongoDB оптимизирована для операций чтения и записи документов, что важно для скорости доступа к содержимому заметок.

— **Сохранение в редакторе:** Реализовано отложенное (debounced) сохранение изменений, что снижает частоту обращений к серверу и базам данных при активном редактировании, не перегружая систему частыми мелкими операциями.

— **Загрузка изображений:** Вынесена в отдельный эндпоинт, обработка файла происходит асинхронно.

В ходе ручного тестирования на тестовых данных заметных проблем с производительностью при выполнении основных операций (открытие проектов, файлов, редактирование текста) выявлено не было. Однако, для больших объемов данных в одной заметке или при очень большом количестве одновременных пользователей могут потребоваться дополнительные оптимизации (например, пагинация блоков контента, оптимизация запросов к БД).

Вывод

Исследование применимости показало, что разработанный программно-алгоритмический комплекс, на текущем этапе его развития, соответствует основным заявленным целям по организации совместной работы с проектами и документами типа "заметка". Заложенные архитектурные решения, в частности, модель данных для неоднородного контента и модульная структура,

создают прочную основу для дальнейшего расширения функциональности и добавления новых типов сервисов.

Качественный анализ характеристик комплекса выявил хороший потенциал в области гибкости, расширяемости и масштабируемости. Выбранный технологический стек (Kotlin/Spring Boot, TypeScript/React, PostgreSQL/MongoDB) позволяет эффективно решать поставленные задачи. Хотя количественные метрики производительности и масштабируемости требуют проведения специализированных тестов, текущая реализация демонстрирует приемлемую отзывчивость для основных пользовательских сценариев. Удобство использования обеспечивается применением современных UI-компонентов и знакомых паттернов взаимодействия.

Для дальнейшего развития рекомендуется провести нагрузочное тестирование, расширить покрытие автоматизированными тестами и собрать обратную связь от реальных пользователей для улучшения юзабилити и выявления приоритетных направлений доработки.

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.

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.

— Отсутствие количественной оценки производительности и масштабируемости.

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).

ПРИЛОЖЕНИЕ А
Приложение А

В листингах А.1–А.15 представлены реализации контроллеров.

Листинг А.1 – AuthController часть 1

```
@RestController
@RequestMapping("/users")
class AuthController(
    private val authenticationManager: AuthenticationManager,
    private val jwtTokenProvider: JwtTokenProvider,
    private val userRepository: UserRepository,
    private val passwordEncoder: PasswordEncoder
) {
    @PostMapping("/login")
    fun login(@RequestBody loginRequest: LoginRequest):
       ResponseEntity<*> {
        return try {
            val authentication =
               authenticationManager.authenticate(
                UsernamePasswordAuthenticationToken(
                    loginRequest.login, loginRequest.password)
            )
            SecurityContextHolder.getContext().authentication =
               authentication
            val user = authentication.principal as User
            val token = jwtTokenProvider.createToken(user.login,
               user.id)
            ResponseEntity.ok(JwtResponse(
                token = token,
                id = user.id,
                login = user.login,
                name = user.name,
                surname = user.surname,
                photo = user.photo
            ))
        } catch (e: AuthenticationException) {
            ResponseEntity.status(HttpStatus.UNAUTHORIZED)
            .body("Invalid login or password")
        }
    }
```

Листинг А.2 – AuthController часть 2

```
    @PostMapping("/register")
    fun register(@RequestBody registerRequest: RegisterRequest):
       ResponseEntity<*> {
        if (userRepository.findByLogin(registerRequest.login)
        .isPresent) {
            return ResponseEntity.badRequest().body("Login is
               already taken!")
        }
        val user = User(
            name = registerRequest.name,
            surname = registerRequest.surname,
            login = registerRequest.login,
            passwordInternal =
               passwordEncoder.encode(registerRequest.password),
            photo = registerRequest.photo
        )
        val savedUser = userRepository.save(user)
        return ResponseEntity.status(HttpStatus.CREATED).body(
            mapOf(
                "id" to savedUser.id,
                "name" to savedUser.name,
                "surname" to savedUser.surname,
                "login" to savedUser.login,
                "photo" to savedUser.photo
            )
        )
    }
    @GetMapping("/me")
    fun getCurrentUser(@AuthenticationPrincipal userDetails:
       User): ResponseEntity<Any> {
        return ResponseEntity.ok(mapOf(
            "id" to userDetails.id,
            "name" to userDetails.name,
            "surname" to userDetails.surname,
            "login" to userDetails.login,
            "photo" to userDetails.photo
        ))
    }
}
```

Листинг А.3 – FileController часть 1

```
@RestController
@RequestMapping("/files")
class FileController(private val fileService: FileService) {

    @PostMapping
    fun createFile(@RequestBody dto: FileDto): FileResponseDto =
        fileService.createFile(dto).toResponseDto()

    @GetMapping("/{file_id}")
    fun getFile(@PathVariable file_id: Long): FileResponseDto =
        fileService.getFileById(file_id).toResponseDto()

    @PutMapping("/{file_id}")
    fun updateFile(@PathVariable file_id: Long, @RequestBody
       updateDto: FileDto): FileResponseDto =
        fileService.updateFile(file_id,
           updateDto).toResponseDto()

    @PatchMapping("/{file_id}/super-object")
    fun updateSuperObjectId(
        @PathVariable file_id: Long,
        @RequestBody request: Map<String, String?>
    ): FileResponseDto {
        val superObjectId = request["superObjectId"]
        return fileService.updateSuperObjectId(file_id,
           superObjectId).toResponseDto()
    }

    @PatchMapping("/{file_id}/name")
    fun updateFileName(
        @PathVariable file_id: Long,
        @RequestBody dto: FileUpdateNameDto
    ): FileResponseDto {
        return fileService.updateFileName(file_id,
           dto.name).toResponseDto()
    }
```

Листинг А.4 – FileController часть 2

```
    @DeleteMapping("/{file_id}")
    fun deleteFile(@PathVariable file_id: Long) =
        fileService.deleteFile(file_id)
}


@RestController
@RequestMapping("/file-types")
class FileTypeController(private val service: FileTypeService) {

    @PostMapping
    fun createType(@RequestBody req: Map<String, String>) =
        service.createType(req["name"] ?: throw
           IllegalArgumentException("name required"))

    @GetMapping
    fun getAll() = service.getAllTypes()
}
```

Листинг А.5 – ContentBlockController

```
@RestController
@RequestMapping("/content-blocks")
class ContentBlockController(private val service:
   ContentBlockService) {
    @PostMapping fun create(@RequestBody s: ContentBlock) =
       service.create(s)

    @GetMapping("/{id}") fun getById(@PathVariable id: String) =
       service.getById(id)

    @PutMapping("/{id}") fun update(@PathVariable id: String,
       @RequestBody s: ContentBlock) = service.update(id, s)

    @DeleteMapping("/{id}") fun delete(@PathVariable id: String)
       = service.delete(id)
}
```

Листинг А.6 – FileDownloadController

```
@RestController
@RequestMapping("/projects/{project_id}/files/{file_id}/download")
class FileDownloadController(private val fileDownloadService:
   FileDownloadService) {

    @GetMapping
    fun downloadFile(
        @PathVariable project_id: Long,
        @PathVariable file_id: Long
    ): ResponseEntity<Resource> {
        val resource =
           fileDownloadService.getFileAsResource(project_id,
           file_id)
        val filename = resource.filename ?: "downloaded_file"

        return ResponseEntity.ok()
            .contentType(MediaType.APPLICATION_OCTET_STREAM)
            .header(HttpHeaders.CONTENT_DISPOSITION,
              "attachment; filename=\"$filename\"")
            .body(resource)
    }
}
```

Листинг А.7 – StyleController

```
@RestController
@RequestMapping("/styles")
class StyleController(private val service: StyleService) {
    @PostMapping fun create(@RequestBody s: Style) =
       service.create(s)

    @GetMapping("/{id}") fun getById(@PathVariable id: String) =
       service.getById(id)

    @PutMapping("/{id}") fun update(@PathVariable id: String,
       @RequestBody s: Style) = service.update(id, s)

    @DeleteMapping("/{id}") fun delete(@PathVariable id: String)
       = service.delete(id)
}
```

Листинг А.8 – FileStorageController

```
@RestController
@RequestMapping("/files-storage")
class FileStorageController(private val fileStorageService:
   FileStorageService) {
    @PostMapping("/upload/image")
    fun uploadImage(@RequestParam("image") file: MultipartFile):
       ResponseEntity<FileUploadResponse> {
        try {
            val response = fileStorageService.storeFile(file)
            return ResponseEntity.ok(response)
        } catch (e: Exception) {
            val errorResponse = FileUploadResponse(
                success = 0,
                file = com.example.projectmanagement.controllers
                .dto.FileDetails(
                    url = "",
                    name = file.originalFilename)
            )
            println("Error uploading file: ${e.message}")
            return ResponseEntity.status(HttpStatus
            .INTERNAL_SERVER_ERROR).body(errorResponse)
        }
    }
}
```

Листинг А.9 – StylesMapController

```
@RestController
@RequestMapping("/styles-maps")
class StylesMapController(private val service: StylesMapService)
   {
    @PostMapping fun create(@RequestBody s: StylesMap) =
       service.create(s)
    @GetMapping("/{id}") fun getById(@PathVariable id: String) =
       service.getById(id)
    @PutMapping("/{id}") fun update(@PathVariable id: String,
       @RequestBody s: StylesMap) = service.update(id, s)
    @DeleteMapping("/{id}") fun delete(@PathVariable id: String)
       = service.delete(id)
}
```

Листинг А.10 – ProjectController

```
@RestController
@RequestMapping("/projects")
class ProjectController(private val projectService:
   ProjectService) {
    @PostMapping
    fun createProject(@RequestBody project: Project):
       ResponseEntity<ProjectResponseDto> {
        return ResponseEntity.status(HttpStatus.CREATED)
        .body(projectService.createProject(project))
    }
    @GetMapping
    fun getCurrentUserProjects():
       ResponseEntity<List<ProjectResponseDto>> {
         return ResponseEntity.ok(projectService
         .getProjectsForCurrentUser())
    }
    @GetMapping("/{project_id}")
    fun getProject(@PathVariable project_id: Long):
       ResponseEntity<ProjectResponseDto> {
        return ResponseEntity.ok(projectService
        .getProjectById(project_id))
    }
    @PutMapping("/{project_id}")
    fun updateProject(@PathVariable project_id: Long,
       @RequestBody update: Project):
       ResponseEntity<ProjectResponseDto> {
        return ResponseEntity.ok(projectService
        .updateProject(project_id, update))
    }
    @DeleteMapping("/{project_id}")
    fun deleteProject(@PathVariable project_id: Long):
       ResponseEntity<Void> {
        projectService.deleteProject(project_id)
        return ResponseEntity.noContent().build()
    }
}
```

Листинг А.11 – ProjectFileController часть 1

```
@RestController
@RequestMapping("/projects/{project_id}/files")
class ProjectFileController(private val projectFileService:
   ProjectFileService) {
    @PostMapping("/upload")
    fun uploadAndLinkFileToProject(
        @PathVariable("project_id") projectId: Long,
        @RequestParam("file") multipartFile: MultipartFile,
        @RequestParam("type_id") typeId: Long
    ): ResponseEntity<ProjectFileResponseDto> {
        val createdFileDto =
           projectFileService.uploadAndLinkFile(projectId,
           multipartFile, typeId)
        return ResponseEntity.status(HttpStatus.CREATED)
        .body(createdFileDto)
    }

    @PostMapping("/link")
    fun linkExistingFileToProject(
        @PathVariable("project_id") projectId: Long,
        @RequestParam("file_id") fileId: Long
    ): ResponseEntity<Map<String, Long>> {
        val projectFile =
           projectFileService.linkExistingFile(projectId, fileId)
        return ResponseEntity.ok(mapOf("project_id" to
           projectFile.project.id, "file_id" to
           projectFile.file.id))
    }

    @GetMapping
    fun getFilesForProject(@PathVariable("project_id")
       projectId: Long):
       ResponseEntity<List<ProjectFileResponseDto>> {
        return ResponseEntity.ok(projectFileService
        .getFilesForProjectWithDetails(projectId))
    }
```

Листинг А.12 – ProjectFileController часть 2

```
@RestController
@RequestMapping("/projects/{project_id}/files")
class ProjectFileController(private val projectFileService:
   ProjectFileService) {
    @PostMapping("/upload")
    fun uploadAndLinkFileToProject(
        @PathVariable("project_id") projectId: Long,
        @RequestParam("file") multipartFile: MultipartFile,
        @RequestParam("type_id") typeId: Long
    ): ResponseEntity<ProjectFileResponseDto> {
        val createdFileDto =
           projectFileService.uploadAndLinkFile(projectId,
           multipartFile, typeId)
        return ResponseEntity.status(HttpStatus.CREATED)
        .body(createdFileDto)
    }

    @PostMapping("/link")
    fun linkExistingFileToProject(
        @PathVariable("project_id") projectId: Long,
        @RequestParam("file_id") fileId: Long
    ): ResponseEntity<Map<String, Long>> {
        val projectFile =
           projectFileService.linkExistingFile(projectId, fileId)
        return ResponseEntity.ok(mapOf("project_id" to
           projectFile.project.id, "file_id" to
           projectFile.file.id))
    }

    @GetMapping
    fun getFilesForProject(@PathVariable("project_id")
       projectId: Long):
       ResponseEntity<List<ProjectFileResponseDto>> {
        return ResponseEntity.ok(projectFileService
        .getFilesForProjectWithDetails(projectId))
    }
```

Листинг А.13 – ProjectAccessController

```
@RestController
@RequestMapping("/projects-users")
class ProjectAccessController(private val service:
   ProjectUserService) {
    @PostMapping
    fun linkUserToProject(@RequestBody dto: ProjectUserDto):
       ResponseEntity<ProjectUserView> {
        val pu = service.linkUserToProject(dto)
        return ResponseEntity.status(HttpStatus.CREATED)
        .body(pu.toView())
    }
    @GetMapping("/project/{project_id}/users")
    fun getUsersForProject(@PathVariable project_id: Long):
       ResponseEntity<List<ProjectUserView>> {
        return ResponseEntity.ok(service
        .getUsersForProject(project_id).map { it.toView() })
    }
    data class UpdateUserRoleDto(val role: ProjectRole)
    @PutMapping("/project/{project_id}/user/{user_id}")
    fun updateUserProjectRole(
        @PathVariable project_id: Long,
        @PathVariable user_id: Long,
        @RequestBody updateDto: UpdateUserRoleDto
    ): ResponseEntity<ProjectUserView> {
        val pu = service.updateUserProjectRole(project_id,
           user_id, updateDto.role)
        return ResponseEntity.ok(pu.toView())
    }
    @DeleteMapping("/project/{project_id}/user/{user_id}")
    fun removeUserFromProject(
        @PathVariable project_id: Long,
        @PathVariable user_id: Long
    ): ResponseEntity<Void> {
        service.removeUserFromProject(project_id, user_id)
        return ResponseEntity.noContent().build()
    }
}
```

Листинг А.14 – SuperObjectController

```
@RestController
@RequestMapping("/super-objects")
class SuperObjectController(private val service:
   SuperObjectService) {
    @PostMapping fun create(@RequestBody s: SuperObject) =
       service.create(s)

    @GetMapping("/by-file/{fileId}") fun
       getByFileId(@PathVariable fileId: Long) =
       service.getByFileId(fileId)

    @GetMapping("/{id}") fun getById(@PathVariable id: String) =
       service.getById(id)

    @PutMapping("/{id}") fun update(@PathVariable id: String,
       @RequestBody s: SuperObject) = service.update(id, s)

    @DeleteMapping("/{id}") fun delete(@PathVariable id: String)
       = service.delete(id)

    @PutMapping("/{superObjectId}/sync-blocks")
    fun syncDocumentBlocks(
        @PathVariable superObjectId: String,
        @RequestBody blocksPayload: List<EditorJsBlockDto>
    ): ResponseEntity<SuperObject> {
        val updatedSuperObject =
           service.syncDocumentBlocks(superObjectId,
           blocksPayload)
        return ResponseEntity.ok(updatedSuperObject)
    }
}
```

Листинг А.15 – UserController

```
@RestController
@RequestMapping("/users")
class UserController(private val userService: UserService) {

    @GetMapping("/search")
    fun searchUsersByLogin(@RequestParam("login") loginQuery:
       String): ResponseEntity<List<UserResponseDto>> {
        val users = userService.searchUsersByLogin(loginQuery)
        return ResponseEntity.ok(users)
    }

    @GetMapping("/{user_id}")
    fun getUser(@PathVariable user_id: Long): User {
        return userService.getUserById(user_id)
    }

    @PutMapping("/{user_id}")
    fun updateUser(@PathVariable user_id: Long, @RequestBody
       updateUser: User): User {
        return userService.updateUser(user_id, updateUser)
    }

    @DeleteMapping("/{user_id}")
    fun deleteUser(@PathVariable user_id: Long) {
        userService.deleteUser(user_id)
    }
}
```

ПРИЛОЖЕНИЕ Б
Приложение Б

В листингах Б.1–Б.12 представлены реализации клиентской части приложения.

Листинг Б.1 – Маршрутизация приложения часть 1

```
import React from 'react';
import { BrowserRouter as Router, Routes, Route, Link, Navigate
   } from 'react-router-dom';
import { ThemeProvider } from '@gravity-ui/uikit';
import '@gravity-ui/uikit/styles/styles.css';
import './index.scss';

import { NoteEditPage } from '../pages/note';
import ProjectDashboardPage from
   '../pages/project/ProjectDashboardPage';
import LoginPage from '../pages/auth/LoginPage';
import RegisterPage from '../pages/auth/RegisterPage';
import { useAuth } from '../shared/context/AuthContext';

const ProtectedRoute: React.FC<{ children: JSX.Element }> = ({
   children }) => {
  const { isAuthenticated, isLoading } = useAuth();

  if (isLoading) {
    return <div>Загрузка аутентификации...</div>;
  }

  if (!isAuthenticated) {
    return <Navigate to="/login" replace />;
  }
  return children;
};

function App() {
  const { isAuthenticated, isLoading } = useAuth();
```

Листинг Б.2 – Маршрутизация приложения часть 2

```
  if (isLoading) {
    return (
      <ThemeProvider theme="light">
        <div>Загрузка приложения...</div>
      </ThemeProvider>
    );
  }

  return (
    <ThemeProvider theme="light">
      <Router>
        <Routes>
          <Route path="/login" element={<LoginPage />} />
          <Route path="/register" element={<RegisterPage />} />
          <Route
            path="/projects"
            element={<ProtectedRoute><ProjectDashboardPage
               /></ProtectedRoute>}
          />
          <Route
            path="/notes/:noteId/edit"
            element={<ProtectedRoute><NoteEditPage
               /></ProtectedRoute>}
          />
          <Route
            path="/"
            element={isAuthenticated ? <Navigate to="/projects"
               replace /> : <Navigate to="/login" replace />}
          />
        </Routes>
      </Router>
    </ThemeProvider>
  );
}

export default App;
```

Листинг Б.3 – Компонент отображения содержимого проектов часть 1

```
import React, { useState, useRef, useEffect } from 'react';
import { Button, Text, Loader, TextInput, Icon } from
   '@gravity-ui/uikit';
import { Plus, Pencil, Gear } from '@gravity-ui/icons';

import { Project, ProjectFile, ProjectRole } from
   '../../shared/api/models';

import FileGrid from '../../entities/FileGrid/FileGrid';

import './MainArea.scss';

interface MainAreaProps {
    selectedProject?: Project | null;
    files: ProjectFile[];
    onOpenFile: (superObjectId: string) => void;
    onCreateNewFile: () => void;
    isLoading: boolean;
    onProjectNameUpdate: (projectId: number, newName: string) =>
       Promise<void>;
    onUpdateFileName: (fileId: number, superObjectId: string,
       newName: string) => Promise<void>;
    onOpenAccessModal: () => void;
}

const MainArea: React.FC<MainAreaProps> = ({
  selectedProject,
  files,
  onOpenFile,
  onCreateNewFile,
  isLoading,
  onProjectNameUpdate,
  onUpdateFileName,
  onOpenAccessModal,
}) => {
    const [isEditingProjectName, setIsEditingProjectName] =
       useState(false);
    const [newProjectName, setNewProjectName] = useState('');
    const [projectNameError, setProjectNameError] =
       useState<string | undefined>(undefined);
```

Листинг Б.4 – Компонент отображения содержимого проектов часть 2

```
    const projectNameInputRef = useRef<HTMLInputElement>(null);

    const canEditProject = selectedProject?.currentUserRole ===
       ProjectRole.OWNER || selectedProject?.currentUserRole ===
       ProjectRole.EDITOR;
    const isOwner = selectedProject?.currentUserRole ===
       ProjectRole.OWNER;

    useEffect(() => {
        if (selectedProject) {
            setNewProjectName(selectedProject.name);
            setIsEditingProjectName(false);
        }
    }, [selectedProject]);

    useEffect(() => {
        if (isEditingProjectName && projectNameInputRef.current)
           {
            projectNameInputRef.current.focus();
            projectNameInputRef.current.select();
        }
    }, [isEditingProjectName]);

    const handleProjectNameEditStart = () => {
        if (selectedProject) {
            setNewProjectName(selectedProject.name);
            setIsEditingProjectName(true);
        }
    };

    const handleProjectNameChange = (value: string) => {
        setNewProjectName(value);
        if (projectNameError && value.trim()) {
            setProjectNameError(undefined);
        }
    };

    const handleProjectNameSave = async () => {
        if (!selectedProject || !newProjectName.trim()) {
           return; }
```

Листинг Б.5 – Компонент отображения содержимого проектов часть 3

```
        if (newProjectName.trim() === selectedProject.name) {
           return; }

        try {
          await onProjectNameUpdate(selectedProject.id,
             newProjectName.trim());
          setIsEditingProjectName(false);
          setProjectNameError(undefined);
        } catch (error) {
          console.error("Ошибка обновления имени проекта (в
             MainArea):", error);
          setProjectNameError('Не удалось обновить имя
             проекта.');
        }
      };

    const handleProjectNameCancel = () => {
        setIsEditingProjectName(false);
        setProjectNameError(undefined);
        if (selectedProject) {
            setNewProjectName(selectedProject.name);
        }
    };

    if (!selectedProject) {
        return (
        <div className="main-area main-area--empty">
            <Text variant="header-1"
               className="main-area__placeholder-title">Выберите
               или создайте проект</Text>
            <Text variant="body-1" color="secondary">
            Чтобы начать работу, выберите проект из списка слева
               или создайте новый.
            </Text>
        </div>
        );
    }

    return (
        <div className="main-area">
```

Листинг Б.6 – Компонент отображения содержимого проектов часть 4

```
          <header className="main-area__header">
            {isEditingProjectName && canEditProject ? (
              <div className="main-area__project-name-edit">
                <TextInput
                  value={newProjectName}
                  onUpdate={handleProjectNameChange}
                  onKeyPress={(e) => { if (e.key === 'Enter')
                     handleProjectNameSave(); if (e.key ===
                     'Escape') handleProjectNameCancel(); }}
                  size="xl"
                  error={projectNameError}
                  controlRef={projectNameInputRef}
                />
                <Button view="action"
                   onClick={handleProjectNameSave}>
                  Сохранить</Button>
                <Button view="flat"
                   onClick={handleProjectNameCancel}>
                  Отмена</Button>
              </div>
            ) : (
              <div className="main-area__project-name-display"
                 onClick={handleProjectNameEditStart}
                 title="Редактировать имя проекта">
                <Text variant="header-1"
                   className="main-area__project-title">
                  {selectedProject.name}</Text>
                {canEditProject ? (
                  <Button view="flat-secondary"
                     className="main-area__edit-icon-button">
                    <Icon data={Pencil} size={18} />
                  </Button>
                ) : null}
              </div>
            )}
            {selectedProject.currentUserRole && (
                    <Text variant="caption-1" color="secondary"
                       className="main-area__current-role">
                        Ваша роль:
```

Листинг Б.7 – Компонент отображения содержимого проектов часть 5

```
                        {selectedProject.currentUserRole ===
                           ProjectRole.OWNER ? "Владелец" :
                           selectedProject.currentUserRole}
                    </Text>
            )}
            <div className="main-area__actions">
                {(canEditProject) && (
                    <Button view="action" size="l"
                       onClick={onCreateNewFile}>
                        <Button.Icon><Plus /></Button.Icon>
                        Создать файл
                    </Button>
                )}
                {isOwner && (
                    <Button view="outlined" size="l"
                       onClick={onOpenAccessModal}
                       title="Управление доступом">
                        <Icon data={Gear} />
                    </Button>
                )}
            </div>
          </header>

        {isLoading ? (
            <div className="main-area__loader">
            <Loader size="l" />
            </div>
        ) : (
            <FileGrid files={files} onOpenFile={onOpenFile}
               onUpdateFileName={onUpdateFileName}
               currentUserRole={selectedProject.currentUserRole}/>
        )}
        </div>
    );
};

export default MainArea;
```

Листинг Б.8 – 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';
import Table from '@editorjs/table';
import Embed from '@editorjs/embed';

import { editorJsRussianLocale } from
   '../../../shared/i18n/editorjs-ru';

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.9 – 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.10 – 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.11 – 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.12 – 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
Приложение В

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

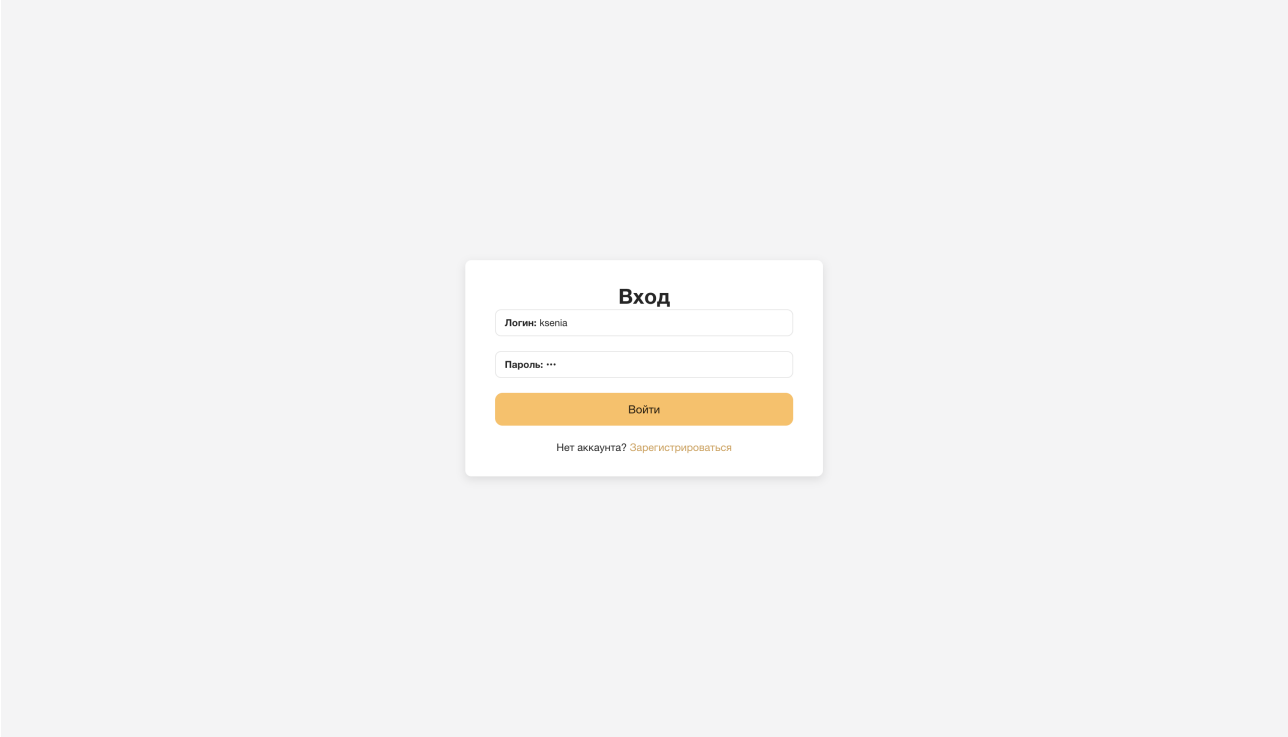

Рисунок В.1 – Экран входа в систему

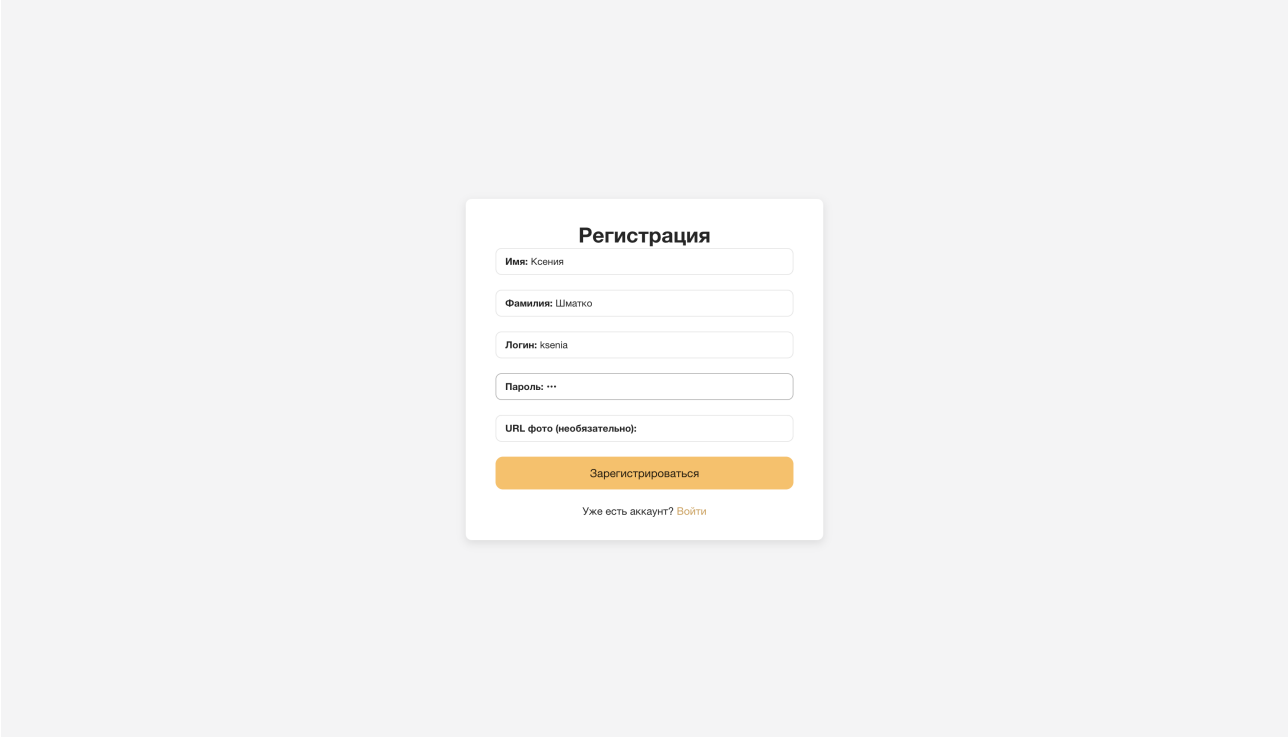

Рисунок В.2 – Экран регистрации в системе

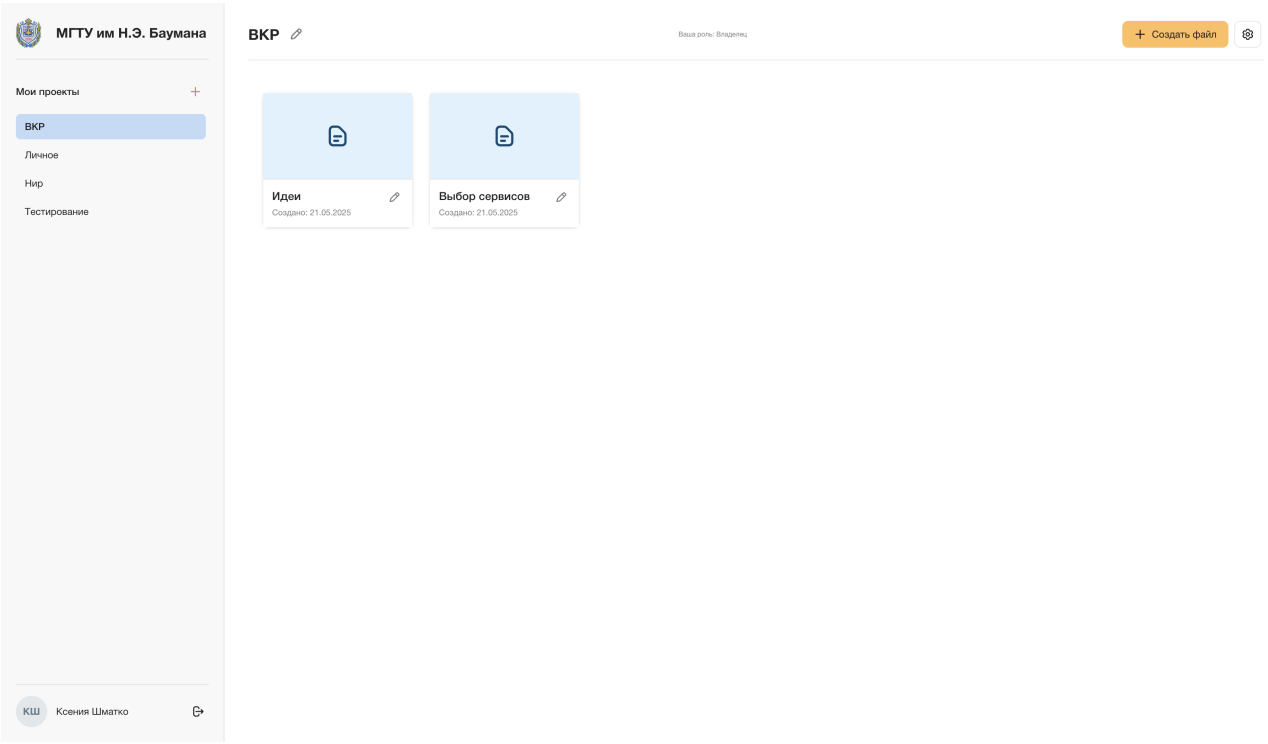

Рисунок В.3 – Экрана выбора проекта и просмотра файлов

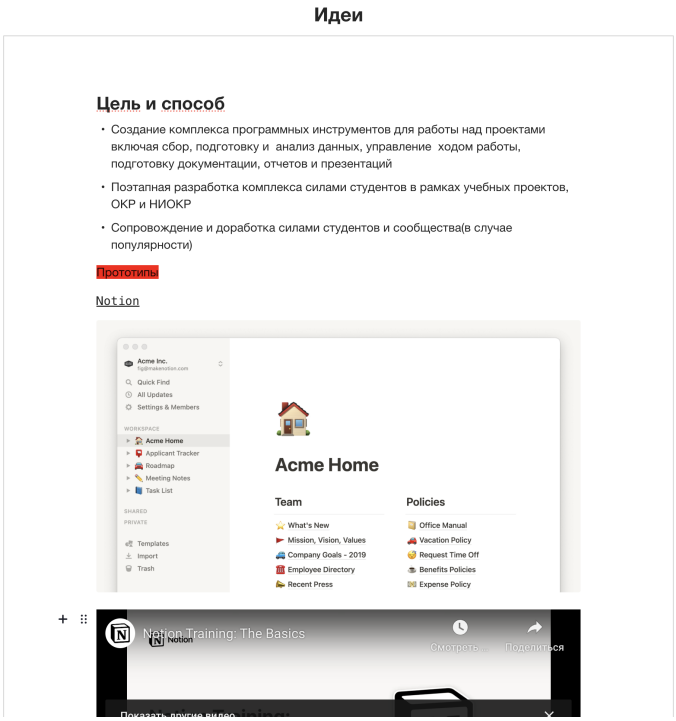

Рисунок В.4 – Экрана редактирования заметки

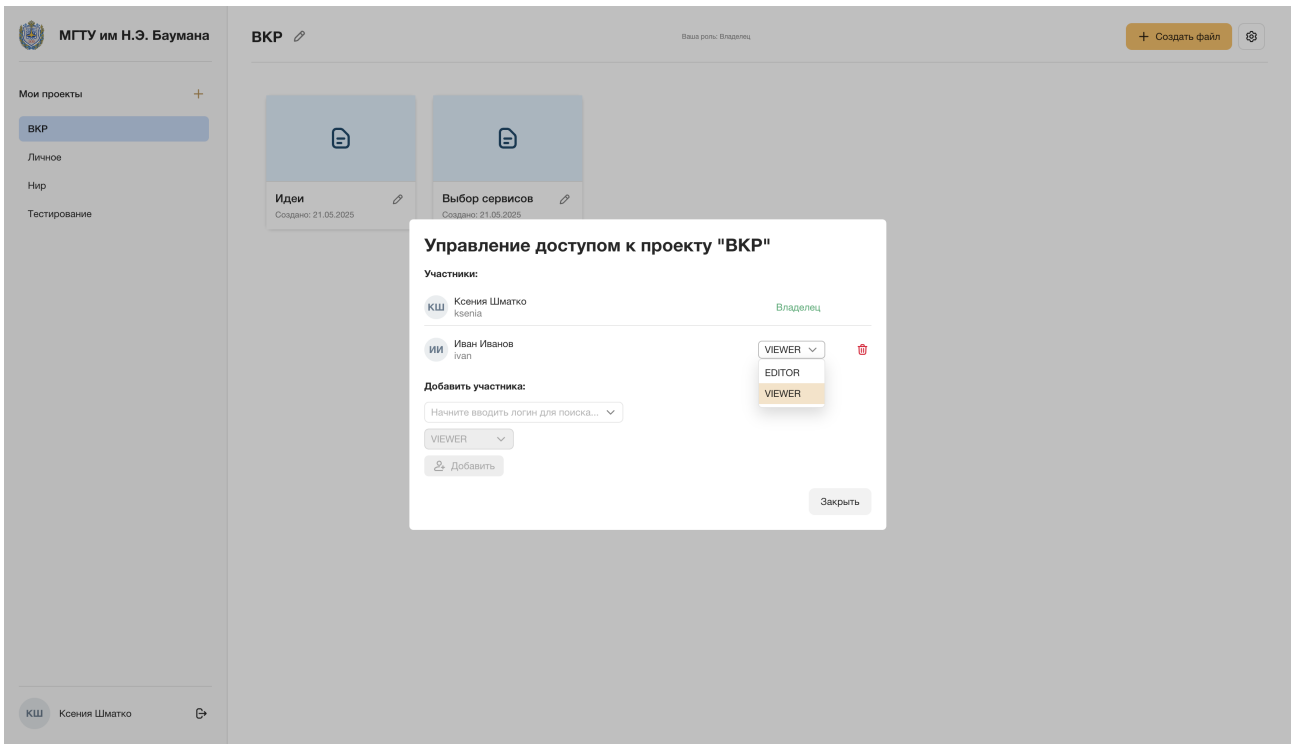

Рисунок В.5 – Экрана управления ролями в проекте