

СОДЕРЖАНИЕ

Введение.....	6
1 ОБЗОР ЛИТЕРАТУРЫ.....	7
1.1 Обзор аналогов и существующих решений	7
1.2 Файловые хранилища	8
1.3 Серверная разработка	10
2 Системное проектирование.....	14
2.1 Ключевые сценарии	14
2.2 Структура приложения.....	16
2.3 Блок представления	17
2.4 Серверная часть.....	19
3 Функциональное проектирование	22
3.1 Модель данных.....	22
3.2 Настройка серверной части.....	28
3.3 Блок управления учетными записями.....	29
3.4 Блок управления рабочим пространством.....	32
3.5 Блок управления файловым хранилищем	42
3.6 Блок сервисов	43
3.7 Клиентская часть.....	49
4 Разработка программных модулей.....	53
4.1 Алгоритмы серверной части	53
4.2 Алгоритмы клиентской части.....	58
5 Программа и методика испытаний.....	62
5.1 Тестирование	62
6 Руководство пользователя.....	69
6.1 Первый запуск приложения	69
6.2 Основное рабочее пространство.....	70
7 Техничко-экономическое обоснование разработки и реализации на рынке программной платформы единого рабочего пространства	76
7.1 Характеристика программного средства, разрабатываемого для реализации на рынке.....	76
7.2 Расчет инвестиций в разработку программного средства для его реализации на рынке.....	77
7.3 Расчет экономического эффекта от реализации программного средства на рынке	79
7.4 Расчет показателей экономической эффективности разработки и реализации программного средства на рынке	80
7.5 Вывод об экономической целесообразности разработки программного средства.....	80
Заключение	82
Список использованных источников	83
Приложение А	85
Приложение Б	119
Приложение В.....	120

ВВЕДЕНИЕ

В мире объем информации, с которой сталкиваются люди и организации, стремительно растет. Это касается как личных заметок и списков дел, так и сложных рабочих процессов, включающих управление знаниями, проектами и задачами. Эти данные представлены в разных форматах, хранятся в различных местах и зачастую неудобны для структурирования и запоминания. Для эффективной работы существуют различные решения, но каждое из них имеет свои ограничения.

На протяжении долгого времени люди использовали физические носители для ведения записей и организации информации. Например: бумажные блокноты, ежедневники, папки с документами. Однако такие методы имеют значительные недостатки.

С развитием технологий стали популярными цифровые средства организации информации, включая текстовые редакторы, облачные хранилища и специализированные приложения. Наиболее распространенные категории таких решений: приложения для заметок, системы управления проектами, облачные сервисы.

Несмотря на разнообразие решений, пользователи часто сталкиваются с необходимостью комбинировать несколько инструментов, что усложняет процесс работы с информацией и снижает продуктивность. Существующие инструменты либо узко специализированы, либо требуют сложной интеграции для выполнения различных задач.

Целью данной дипломной работы станет разработка платформы, которая решит многие проблемы неудобной работы с данными, поможет пользователям организовывать и структурировать информацию, улучшать продуктивность и оптимизировать рабочие процессы.

Для достижения цели данного дипломного проекта можно выделить следующие задачи:

- проанализировать аналоги и существующие решения;
- выбор технологий и инструментов для реализации проекта;
- определение основных функциональных требований и пользовательских сценариев;
- спроектировать архитектуру приложения и взаимосвязей модулей;
- спроектировать базы данных и соответствующие сервисы;
- написать программное обеспечение;
- разработать методику испытаний и протестировать программное обеспечение;
- написать подробное руководство пользователя.

Данный дипломный проект выполнен мной лично, проверен на заимствования, процент оригинальности составляет XX% (отчет о проверке на заимствования прилагается).

1 ОБЗОР ЛИТЕРАТУРЫ

В данном разделе будут описаны существующие аналоги. Также будет изложен основной теоретический материал, который необходим для понимания данного дипломного проекта.

1.1 Обзор аналогов и существующих решений

В начале проектирования приложения хорошим решением будет начать с обзора и анализа существующих аналогов, чтобы выявить их преимущества и недостатки. На основе выводов можно выявить ключевой функционал и сделать свое приложение конкурентоспособным и актуальным.

1.1.1 Приложение Obsidian. Obsidian – приложение для персональной базы знаний, которое служит инструментом для создания и управления заметками [1]. Оно приобрело популярность благодаря своей гибкости, возможностям персонализации и поддержке взаимосвязанных заметок.

Ключевыми особенностями Obsidian являются:

- полный контроль над файлами;
- возможность работы офлайн;
- безопасность информации;
- быстрый поиск и навигация;
- поддержка плагинов.

Приложение Obsidian сохраняет всю информацию локально на устройстве и использует упрощенный язык разметки Markdown в качестве основного инструмента для работы с файлами. Такой подход обеспечивает полный контроль над данными и гарантирует, что информация не попадет к третьим лицам. Однако это накладывает определенные требования на пользователя – необходимо умение работать с Markdown-файлами, что может сделать приложение менее удобным для новичков.

Кроме того, Obsidian не предоставляет собственного облачного хранилища. Для синхронизации заметок между несколькими устройствами пользователю придется использовать сторонние сервисы.

Можно выделить основные недостатки данного приложения:

- для синхронизации данных между устройствами требуется ручная настройка;
- отсутствие продвинутых инструментов для работы с данными;
- новичкам может потребоваться время для освоения всех возможностей;
- отсутствие встроенной поддержки совместной работы.

1.1.2 Приложение Notion. На данный момент главным конкурентом является приложение Notion. Эта платформа для ведения заметок, управления знаниями проектами и задачами [2]. Она популярна среди свободных работников, малых бизнесов, крупных компаний и студентов.

Главной особенностью Notion является ее богатый набор инструментов и возможность интеграции с такими сервисами, как Google Calendar, Slack, Trello и другими.

Платформа Notion позволяет создавать текстовые документы, списки задач, таблицы, календари, галереи изображений и другие элементы, которые можно комбинировать для создания гибких рабочих пространств. Также доступна возможность создания баз данных с фильтрацией, сортировкой и различными вариантами отображения. Для упрощения работы предоставляется множество готовых шаблонов, подходящих как для личных, так и для рабочих задач, таких как планирование, бюджетирование и управление проектами.

Весь функционал возможен с совместным редактированием документов в реальном времени.

У Notion мало недостатков, которые смогли бы сделать платформу неконкурентоспособной в каком-то направлении. Но можно выделить несколько минусов у приложения:

- ограничения оффлайн-доступа;
- сложность освоения;
- производительность при работе с большими базами данных;
- ограничения использования функционала.

1.1.3 Приложение Miro. Miro – это цифровая платформа для рабочего пространства, в котором команды управляют проектами, разрабатывают продукты и создают карты мыслей [3].

Это интересное решение, непохожее на остальные. Приложение представляет собой виртуальную интерактивную доску. Оно позволяет пользователям создавать визуальные схемы, диаграммы, текстовые блоки и многое другое. Самое главное отличие в том, что пользователи не привязаны к строгой структуре страниц и могут использовать все неограниченное пространство доски.

Платформа является прекрасным решением для совещаний, презентаций, визуализации сложных схем, а также для совместной работы над проектами.

Несмотря на оригинальность, Miro имеет минусы, которые могут ограничить его использование:

- на больших досках с множеством объектов приложение может работать медленно;
- бесплатная версия ограничивает количество досок и функций;
- miro имеет достаточно большой функционал, и новому пользователю может понадобиться время.

1.2 Файловые хранилища

Обязательным элементом для приложений, которые работают с медиа, являются файловые хранилища.

Системы хранения данных делят на три основных типа:

- файловые системы;
- блочные системы;
- объектные системы.

Для проектирования своего приложения необходимо понимать преимущества и недостатки каждого решения, а также основные принципы его функционирования.

Файловые системы управляют данными, организуя их в виде файлов и директорий, предоставляя пользователю привычный интерфейс работы с информацией. Доступ к файлам осуществляется по пути, который включает имя сервера, путь к каталогу и имя файла. На низком уровне используется блочный метод представления информации.

Блочное хранилище предоставляет низкоуровневый доступ к данным, разбивая данные на блоки. Блок данных – это минимальная единица хранения, содержащая часть информации, которая может быть записана, прочитана или изменена независимо от других блоков. Блоки имеют фиксированный размер и хранятся в произвольных местах на диске. Блочное хранилище предоставляет приложениям доступ к этим блокам без информации о файловой системе, позволяя операционной системе или базе данных управлять их структурированием и организацией. Файлы состоят из конечного числа блоков. Серверная операционная система назначает каждому блоку данных уникальный идентификатор расположения, позволяющий быстро находить его. Благодаря этому блочные системы хранения обеспечивают высокую скорость доступа к данным.

Объектное хранилище организует данные в виде объектов. Объекты представляют собой самостоятельные единицы данных, которые сохраняются без строгой структуры или иерархии. Каждый объект содержит сами данные, метаданные с описательной информацией и уникальный идентификатор. Системное программное обеспечение использует эти характеристики для поиска и доступа к объектам. Благодаря такому подходу объектное хранилище обеспечивает высокую масштабируемость, так как данные могут распределяться по множеству серверов или узлов.

На рисунке 1.1 изображены основные типы хранилищ.



Рисунок 1.1 – Основные типы хранилищ

Для лучшего понимания следует проанализировать и сравнить хранилища. Также определить основные сценарии использования. Результат анализа представлен в таблице 1.1.

Таблица 1.1 – Сравнения типов хранилищ

Характеристики	Файловые	Блочные	Объектные
Относительная стоимость	Средняя	Высокая	Низкая
Быстродействие	Высокое	Очень высокое	Среднее
Масштабируемость	Ограниченная	Средняя	Высокая
Совместимость с облачными технологиями	Средняя	Низкая	Высокая
Гибкость управления данными	Высокая	Низкая	Высокая

Относительная стоимость определяет, насколько дорого обходится развертывание и эксплуатация данного типа хранилища. Масштабируемость – способность системы увеличивать объем хранилища без значительных изменений в архитектуре. Совместимость с облачными технологиями показывает насколько хорошо хранилище интегрируется с облачными платформами. Гибкость управления данными отражает, насколько удобно управлять данными.

По таблице 1.1 можно сказать, к каким сценариям подходят данные типы хранилищ.

Файловые хранилища удобны для традиционных систем хранения данных, но обладают ограниченной масштабируемостью. Они подходят для хранения общего пользовательского контента и веб-контента.

Блочные хранилища оптимальны для высокопроизводительных задач, таких как базы данных и виртуализация. Однако это дорогостоящее решение, которое также слабо совместимо с облачными технологиями.

Объектные хранилища являются лучшим выбором для облачных и распределенных систем, но они менее производительны при частом доступе к данным.

1.3 Серверная разработка

Для разработки серверного приложения необходимо понимать современные подходы к проектированию и соответствующие методы. Также понимать сценарии, при которых необходимо использовать технологии.

1.3.1 Асинхронные и синхронные операции. Популярным подходом к построению систем является использование асинхронной модели ввода-вывода.

Преимущества WebSocket над HTTP:

1. Нет необходимости устанавливать новое соединение для каждого запроса, что значительно снижает задержку.
2. В отличие от традиционного HTTP, WebSocket не требует заголовков в каждом сообщении.
3. Сервер не обрабатывает повторные запросы от клиентов, что снижает нагрузку.

1.3.3 Современная архитектура серверных приложений. Одной из доминирующих архитектур является многоуровневая архитектура, которая позволяет разработчикам структурировать код в виде модулей, контроллеров и провайдеров. Это способствует поддержанию чистоты кода, его повторному использованию и легкости тестирования. На рисунке 1.3 изображена упрощенная многоуровневая архитектура.

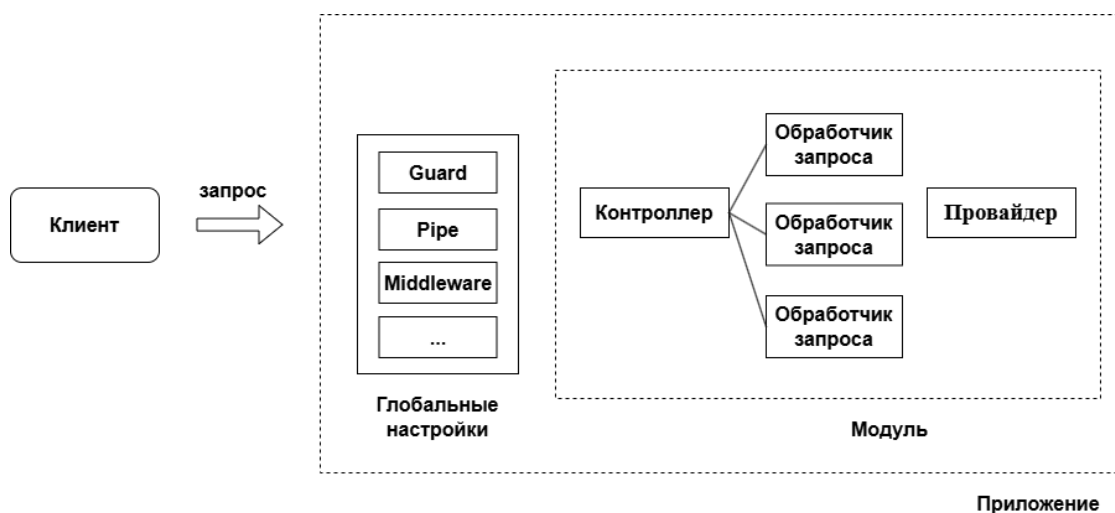


Рисунок 1.3 – Многоуровневая архитектура

1.3.4 Серверный рендеринг. Это решение необходимо для создания производительных приложений.

Серверный рендеринг – это подход, при котором страницы и статические файлы генерируются на сервере на этапе сборки и отправляются клиенту в уже готовом виде. Это отличается от клиентского рендеринга, где большая часть работы выполняется в браузере с помощью JavaScript. Серверный рендеринг имеет несколько ключевых преимуществ, которые делают его популярным выбором для создания быстрых и SEO-оптимизированных приложений.

Еще одна мощная функция – инкрементальная статическая регенерация. Она сочетает преимущества статической генерации и динамических обновлений: страницы генерируются на этапе сборки, но при необходимости могут обновляться без полной пересборки проекта.

Схематичное отображение работы рендеринга на стороне сервера изображено на рисунке 1.4.

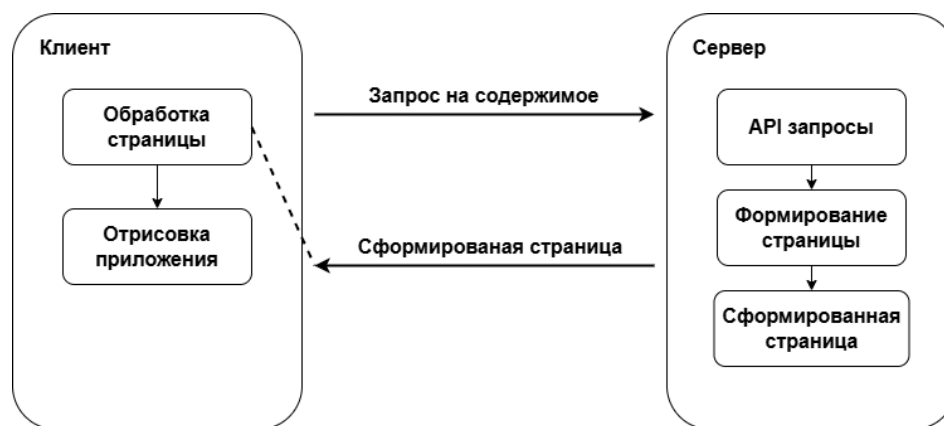


Рисунок 1.4 – Рендеринг на стороне сервера

1.3.5 Серверная безопасность. Безопасность платформы во многом определяется защитой серверной части приложений, включая контроль доступа, шифрование данных и защиту от сетевых атак. Для определения актуальных угроз можно использовать открытый проект OWASP (Open Web Application Security Project) [4]. Эксперты этой организации ежегодно обновляют список критических уязвимостей приложений.

При анализе потенциальных атак можно выделить минимальный набор технологий обеспечения безопасности:

1. JSON Web Token (JWT) – стандарт, для создания токенов аутентификации и передачи информации между участниками системы. Безопасность зависит от правильного применения алгоритмов подписи и шифрования.

2. Современные криптографические методы – применение актуальных алгоритмов шифрования и хеширования.

3. CAPTCHA – технология защиты от автоматизированных запросов и атак методом перебора.

4. ORM-библиотеки – инструменты, обеспечивающие безопасное взаимодействие с базой данных за счет автоматического экранирования пользовательского ввода и защиты от SQL-инъекций.

5. Механизмы валидации – технология автоматической проверки и фильтрации пользовательского ввода для предотвращения атак, связанных с передачей некорректных или вредоносных данных.

2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе изложены основные сценарии взаимодействия системы с одним или несколькими действующими лицами, а также представлено высокоуровневое описание приложения.

2.1 Ключевые сценарии

Основной задачей при проектировании приложения является определение ключевых сценариев, которые помогут принять оптимальное решение относительно архитектуры. Важно найти баланс между потребностями пользователя, бизнеса и техническими требованиями системы. Для этого нужно определить ключевые варианты использования со стороны пользователя и провести их анализ, чтобы выявить соответствующие задачи для разработки.

Основные сценарии взаимодействия с системой:

- авторизация пользователей;
- аутентификация пользователей;
- управление рабочим пространством;
- управление досками;
- совместная работа;
- экспорт и импорт данных.

Далее необходимо разбить основные сценарии на элементарные составляющие с точки зрения реализации.

2.1.1 Авторизации и аутентификация пользователей. Когда пользователь открывает страницу регистрации, система отображает форму с полями почта, пароль и подтверждение пароля. Форма проводит проверку ввода на корректность в реальном времени.

После успешного заполнения формы данные отправляются на сервер с помощью запроса. Сервер проверяет: не зарегистрирован ли уже пользователь. Если данные уникальны, выполняет хеширование пароля. Затем данные записываются в базу данных.

После успешной регистрации система генерирует уникальный ключ подтверждения и отправляет пользователю письмо со ссылкой. При переходе по этой ссылке сервер проверяет ключ, активирует учетную запись и уведомляет пользователя об успешной верификации.

Когда пользователь вводит почту и пароль на странице входа, форма отправляет данные на сервер. Сервер выполняет поиск пользователя по почте в базе данных. Если учетная запись найдена, происходит сравнение введенного пароля с сохраненным хешем. Если пароль совпадает, система генерирует ключ доступа, который передается пользователю. Этот ключ хранится в локальном хранилище браузера (LocalStorage).

При последующих запросах на защищенные ресурсы клиент передает этот ключ, а сервер проверяет его достоверность, срок действия и соответствие

пользователю. Если ключ истек или недействителен, пользователь перенаправляется на страницу входа.

Этот подход обеспечивает защиту данных, безопасное хранение паролей и удобный механизм аутентификации.

2.1.2 Управление рабочим пространством и досками. После успешной авторизации пользователь запрашивает загрузку рабочего пространства, и система отправляет API-запрос. Сервер через API передает информацию о рабочем пространстве пользователя, включая его структуру, настройки и связанный контент.

После получения данных система применяет базовые параметры интерфейса, загружает пользовательские настройки и адаптирует отображение элементов в соответствии с предпочтениями пользователя.

Затем контент связывается с элементами интерфейса, и данные интегрируются в рабочее пространство. Текстовые записи, файлы, таблицы и другие компоненты привязываются к своим местам, обеспечивая целостность структуры. Как только все элементы загружены и отображены, система визуализирует рабочее пространство, предоставляя пользователю готовую среду для работы.

На каждой странице пользователь имеет специальную ячейку, в которую можно вставлять шаблонизированный компонент. Этот компонент поддерживает ввод данных, позволяя пользователю настраивать его содержимое в зависимости от текущих задач и потребностей.

Все изменения, внесенные в компонент, сохраняются автоматически и синхронизируются с сервером в реальном времени. Данные компонентов сохраняются в специальном формате, обеспечивающем их корректное восстановление и дальнейшую обработку при последующих загрузках рабочего пространства.

Также сессия локально хранит историю изменений, чтобы предотвратить ошибки случайного пользовательского ввода.

Страницы могут храниться внутри других страниц. Это позволяет формировать иерархическую структуру, где одна страница может выступать в роли контейнера для нескольких вложенных страниц. Такой подход создает гибкую систему досок.

2.1.3 Совместная работа. Для реализации совместной работы владелец страницы должен иметь возможность делиться ею с другими, предоставляя доступ с разными уровнями прав. Для этого он может либо сформировать специальную ссылку, либо отправить приглашение через платформу, указав конкретных пользователей.

При создании ссылки владелец выбирает уровень доступа: просмотр, комментирование или полный доступ к редактированию. Аналогично, при приглашении пользователей он может задать индивидуальные права.

Когда владелец изменяет настройки доступа, система обновляет права на сервере, синхронизируя их в реальном времени. Формируется журнал

активности, который фиксирует изменения и действия пользователей на странице, а также систему уведомлений, информирующую участников о новых приглашениях, изменениях прав или обновлениях в содержимом.

2.1.4 Экспорт и импорт данных. Пользователь может запросить у сервера доступные ему данные в специальном формате, предназначенном для сохранения и последующего восстановления структуры информации. Сервер обрабатывает запрос, собирая все необходимые данные в единый пакет, включая содержимое страниц, вложенные элементы, файлы и настройки. После подготовки данные передаются пользователю в виде загружаемого файла.

Необходимо реализовать выборочные параметры экспорта, позволяя пользователю загружать только определенные данные, а также поддержку нескольких форматов.

Также пользователь может загрузить подготовленный файл с данными на сервер. При получении пакета система анализирует его содержимое и проверяет совместимость формата с платформой. Если файл поддерживается, сервер корректно связывает данные с учетной записью пользователя, воссоздавая структуру страниц, элементов и вложенных данных.

Чтобы минимизировать ошибки, необходимо добавить предварительный просмотр импортируемых данных перед их окончательной загрузкой.

2.2 Структура приложения

На основе ключевых сценариев для приложения можно спроектировать архитектуру платформы, которая будет соответствовать всем требованиям.

Были выделены следующие блоки:

- блок представления;
- серверная часть;
- база данных;
- файловое хранилище.

Блок представления можно разбить на несколько ключевых частей, которые обеспечивают различные аспекты взаимодействия с пользователем. Можно выделить следующие модули:

- модуль интерфейса пользователя;
- модуль хранилища состояний;
- модуль загрузки;
- модуль компонент интерфейса;
- модуль компонент пользователя;
- модуль контроля данных;
- модуль контроля сессии;
- модуль локального хранилища.

Серверную часть для упрощения разработки можно разделить на следующие модули:

- модуль управления учетными записями;
- модуль управления рабочим пространством;
- модуль управления файловым хранилищем;
- модуль сервисов.

База данных и файловое хранилище являются самостоятельными блоками. База данных предназначена для хранения структурированных данных, тогда как файловое хранилище используется для работы с неструктурированными данными.

В качестве системы управления базами данных (СУБД) была выбрана PostgreSQL [5]. Это надежная, высокопроизводительная реляционная СУБД, полностью соответствующая требованиям платформы.

В качестве файлового хранилища использовано MinIO [6]. Это облачное объектное хранилище, обеспечивающее высокую масштабируемость и безопасность. Оно подходит для работы с данными благодаря распределенной архитектуре, репликации и удобному API.

Структурная схема, иллюстрирующая перечисленные блоки и модули, связи между ними, приведена на чертеже ГУИР.400201.063 С1.

2.3 Блок представления

Блок представления отвечает за визуализацию информации. Это включает в себя организацию данных в понятном и доступном формате, а также обеспечение интуитивно понятных инструментов для их просмотра, редактирования и анализа.

Основной технологией разработки данного блока будет фреймворк Next.js [7], так он включает все необходимые инструменты и соответствует современным методам разработки.

2.3.1 Модуль интерфейса пользователя. Модуль интерфейса пользователя является главной частью блока представления.

Этот модуль отвечает за обеспечение взаимодействия пользователя с системой через визуальные элементы и обработку действий. Он выступает связующим звеном между пользователем и функционалом приложения, предоставляя интуитивно понятный способ управления данными и процессами.

Эта часть приложения будет управлять процессами других модулей блока. Также она будет содержать конфигурационные настройки для всех компонентов.

2.3.2 Модуль хранилища состояний. Модуль хранилища состояний отвечает за централизованное управление данными, которые определяют текущее состояние приложения. Для приложений с большим количеством связей необходимо хранилище, которое поможет контролировать поток информации. Данный блок предоставляет инструментарий модулю интерфейса пользователя для управления промежуточными данными.

Главная задача модуля – сохранять целостность данных и синхронизировать их с интерфейсом и бизнес-логикой.

2.3.3 Модуль загрузки. Этот модуль отвечает за управление процессами загрузки данных, файлов или ресурсов. Он обеспечивает запросы к серверу для получения данных, которые необходимы компонентам интерфейса и пользователя.

Этот модуль связан с модулем хранилища состояний, обеспечивая другим модулям контроль над потоком данных через инструментарий.

2.3.4 Модуль компонентов интерфейса. Этот модуль отвечает за создание, управление и повторное использование стандартизированных элементов пользовательского интерфейса. Он содержит компоненты единообразного дизайна.

Данный блок связан напрямую с модулем пользовательского интерфейса.

2.3.5 Модуль компонентов пользователя. Этот модуль отвечает за реализацию элементов интерфейса, которые позволяют пользователю взаимодействовать с данными – просматривать, редактировать, структурировать и анализировать информацию. Он предоставляет стандартизированные шаблоны для отображения контента и включает как базовые, так и продвинутое компоненты, обеспечивая гибкость и удобство работы.

Базовые элементы представления данных будут включать: текст, списки, таблицы, медиа, файлы, код.

Также у модуля есть подмодуль, который будет отвечать за продвинутое компоненты. Подмодуль продвинутого компонентов будет включать: графики и визуализации, базы данных, сложные структуры.

Данный модуль связан с модулем пользовательского интерфейса.

2.3.6 Модуль представления данных. Этот модуль отвечает за преобразование, форматирование и структурирование данных для их корректного и удобного отображения в интерфейсе.

Он выступает промежуточным звеном между «сырыми» данными и UI-компонентами, обеспечивая их совместимость и готовность к визуализации. Когда данные пользователя поступают в модуль хранения состояний, тогда модуль представления готовит их к отображению.

2.3.7 Модуль контроля сессии. Этот модуль отвечает за отслеживание и сохранение действий пользователя в рамках одной сессии работы с приложением. Он позволяет реализовать функционал работы операций со страницей, анализировать поведение пользователя и восстанавливать контекст работы после сбоев или перезагрузки. Модуль хранит информацию с помощью модулей локального хранилища и состояний.

Также блок необходим для инициализации загрузки изменений на сервер, поэтому он связан с хранилищем состояний.

2.3.8 Модуль локального хранилища. Этот модуль отвечает за сохранение данных на стороне клиента, обеспечивая ускорение работы приложения и снижение нагрузки на сервер. Он предоставляет инструменты для безопасного и эффективного управления локальными данными, синхронизируя их с облаком при необходимости. Этот блок необходим для сохранения информации сессии пользователя. Данный блок связан с модулем хранения состояний для обеспечения централизованного управления.

2.4 Серверная часть

Серверная часть необходима для каждого продвинутого приложения. Данный блок содержит основную бизнес логику платформы. Это центральная часть программы.

Основной технологией разработки данного блока будет фреймворк Nest.js [8], так как он предоставляет удобную модульную архитектуру и объединяет элементы объектно-ориентированного и функционального программирования. Также платформа поддерживает TypeScript. Это язык программирования на основе JavaScript, который позволяет использовать строгую типизацию. Такой выбор позволит сделать разработку платформы экономически более выгодной.

2.4.1 Модуль управления учетными записями. Модуль отвечает за редактирование учетных записей пользователей в системе.

Этот модуль является центральным для администрирования пользовательских данных и обеспечивает функциональность, необходимую для поддержки жизненного цикла учетных записей.

Одной из основных функций модуля является назначение ролей и прав пользователей, назначая им соответствующие уровни доступа в системе.

Модуль также будет вести журнал действий пользователей, что полезно для отслеживания изменений и статистики.

Также задачей модуля является регистрация новых пользователей, вход в систему, восстановление пароля и удаление аккаунтов. Модуль ответственен за генерацию ключей доступа, которые используются для аутентификации в запросах.

Для повышения безопасности будет реализована многофакторная аутентификация, которая требует дополнительного подтверждения личности через электронную почту.

2.4.2 Модуль управления рабочим пространством. Данный модуль является одним из ключевых в работе платформы.

Модуль обеспечивает централизованную логику для работы со страницами, участниками и взаимодействиями пользователей. Он отвечает за

создание и настройку страниц, организацию доступа, управление содержимым страниц и взаимодействия между пользователями в рамках рабочего пространства.

Благодаря модулю пользователи могут создавать новые страницы, задавать их структуру, изменять параметры и визуальные элементы, а также удалять или восстанавливать их при необходимости. Система реализует строгую валидацию и аутентификацию через, что обеспечивает безопасный доступ к данным.

Модуль поддерживает гибкую систему управления участниками: пользователи могут приглашать других по email, подтверждать участие через токен-приглашение, удалять участников и изменять их роли. Для реализации приглашений предусмотрена интеграция с почтовым сервисом, позволяющая отправлять уведомления и ссылки для подтверждения участия.

Также реализованы функции подписки на страницы, их оценки и управления комментариями. Комментарии можно создавать, получать и удалять, а сами действия отслеживаются с учетом прав доступа и текущего пользователя. Это позволяет пользователям взаимодействовать с контентом и друг с другом более гибко и прозрачно.

У модуля есть подмодуль часть, отвечающая за синхронизацию и взаимодействие пользователей в реальном времени. После подключения пользователя он инициирует передачу структуры страницы, блоков и связанных элементов, а также организует поток операций с помощью системы транзакций.

2.4.3 Модуль управления файловым хранилищем. Данный модуль является связующим звеном между приложением и системой хранения файлов.

Этот модуль отвечает за организацию и управление файлами, а также за поддержание их структуры. Он отслеживает состояние «озера данных», обеспечивая корректное размещение и доступ к файлам, а также контролирует их целостность, чтобы предотвратить потерю или повреждение данных. Кроме того, модуль создает и управляет метаданными файлов, такими как название, размер, тип, дата создания, автор и другие атрибуты, которые помогают быстро находить и идентифицировать нужные файлы.

Также модуль занимается экспортом и импортом данных. Он анализирует корректность и формат информации.

2.4.4 Модуль сервисов. Модуль сервисов представляет собой ядро бизнес-логики приложения. В нем сосредоточены все основные операции, связанные с обработкой данных, взаимодействием с базой данных, правами доступа и координацией между различными сущностями. Каждый сервис инкапсулирует логику своей предметной области и предоставляет методы, которые используются как контроллерами, так и другими модулями системы.

Модуль будет включать, например, сервис авторизации, который обеспечивает контроль доступа к данным и функциональным возможностям.

Этот сервис подтверждает личность пользователя и определяет, что именно он может делать в системе. Сервис проверяет, имеет ли пользователь достаточные права для выполнения запрашиваемых действий, таких как просмотр, редактирование или удаление данных.

Кроме того, сервис отвечает за управление сессиями пользователей, включая отслеживание активных сессий, их завершение по истечении срока действия ключей доступа.

Сервис авторизации тесно интегрирован с другими модулями, так как для проверки прав доступа необходимо знать, кто именно выполняет запрос. Например, после успешной аутентификации пользователь получает ключ доступа. При каждом запросе к защищенному ресурсу модуль проверяет этот ключ и определяет, разрешено ли действие.

Также в модуле реализован сервис почты, отвечающий за отправку email-сообщений пользователям. Он используется для подтверждения регистрации, восстановления пароля, уведомлений о событиях в рабочем пространстве и других операций, требующих коммуникации с пользователем вне приложения.

3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

Данный раздел дает исчерпывающие знания о реализации приложения. Здесь находится описание о структуре платформы, разработанной в предыдущем разделе (см. раздел 2 Системное проектирование), с точки зрения описания данных и обрабатывающих подпрограмм, функций, процедур.

3.1 Модель данных

В приложении используется реляционная база данных, поэтому необходимо определить структуру таблиц и их взаимосвязи. Типы данных будут описаны с использованием модели TypeScript.

Модель данных, иллюстрирующая перечисленные таблицы, связи между ними, приведена на чертеже ГУИР.400201.063 РР.1.

3.1.1 Таблица `User` является начальной и связующей таблицей приложения, которая описывает учетную запись пользователя. Структура `User` представлена в таблице 3.1.

Таблица 3.1 – Структура таблицы `User`

Атрибут	Тип	Описание
<code>User_id</code>	<code>number</code>	Уникальный идентификатор пользователя
<code>email</code>	<code>string</code>	Электронная почта пользователя
<code>password_hash</code>	<code>string</code>	Хешированный пароль пользователя
<code>name</code>	<code>string</code>	Псевдоним пользователя в системе
<code>storage_limit</code>	<code>number</code>	Максимальный объем данных, который можно хранить в системе
<code>storage</code>	<code>number</code>	Текущий объем данных пользователя
<code>created_at</code>	<code>Date</code>	Дата создания аккаунта пользователя
<code>updated_at</code>	<code>Date</code>	Дата обновления данных пользователя

Первичным ключом является атрибут `User_id`.

У пользователя может быть доступ к многим страницам, а страница может относиться к многим пользователям. Таблица `User` связана с `Pages` отношением «многие ко многим» через отношения `WorkspaceMembers`, `Subscriptions`, `Likes`.

Также пользователь имеет свои настройки и рабочее пространство, поэтому `User` связан с `Workspace` и `UserSettings` «один к одному».

В системе клиент может отправлять приглашения и загружать много файлов, следовательно `User` имеет отношение «один ко многим» к соответствующим таблицам.

3.1.2 Таблица `Workspace` описывает рабочее пространство пользователя. Структура `Workspace` представлена в таблице 3.2.

Таблица 3.2 – Структура таблицы Workspace

Атрибут	Тип	Описание
Workspace_id	number	Уникальный идентификатор рабочего пространства пользователя
name	string	Имя рабочего пространства
owner_id	number	Уникальный идентификатор пользователя ассоциированного с рабочим пространством
user_limit	number	Максимальное число пользователей, которые могут присоединиться к страницам данного рабочего пространства
created_at	Date	Дата создания рабочего пространства
updated_at	Date	Дата обновления рабочего пространства

Первичным ключом является атрибут Workspace_id.

Пользователь может иметь только одно рабочее пространство, поэтому таблица принадлежит таблице User связью «один к одному».

Также рабочее пространство включает в себя страницы. Workspace имеет отношение к Page, как «один ко многим».

3.1.3 Таблица Pages описывает страницы, которые относятся к рабочему пространству пользователя и участникам. Структура Pages представлена в таблице 3.3.

Таблица 3.3 – Структура таблицы Pages

Атрибут	Тип	Описание
1	2	3
Page_id	number	Уникальный идентификатор страницы
title	string	Имя страницы
description	string	Описание страницы
parent_page_id	number	Уникальный идентификатор-ссылка для вложенных страниц
picture_url	string	Ссылка на фоновое изображение
avatar_url	string	Ссылка на тематическое изображение
type	string	Тип станицы. Например, шаблон или персональная страница
is_public	boolean	Флаг, указывающий, является ли страница публичной
category	string	Тематическая категория страницы. Например, «life» или «work»
deleted_at	Date	Дата удаления страницы
created_at	Date	Дата создания страницы
updated_at	Date	Дата обновления страницы

Продолжение таблицы 3.3

1	2	3
workspace_id	number	Уникальный идентификатор родительского рабочего пространства

Первичным ключом является атрибут Page_id.

Рабочее пространство может иметь много страниц, поэтому таблица Workspace имеет связь «один ко многим» с Pages.

Таблица Pages имеет связь «многие ко многим» с User.

Также страница имеет много Invitations, Blocks, ChangeHistory.

3.1.4 Таблица Blocks описывает элементарные единицы и представляет собой основу для организации контента страниц. Этот подход значительно повышает эффективность работы приложения.

Поскольку страница состоит из отдельных блоков, нет необходимости загружать или обновлять ее целиком. Например, если пользователь вносит изменения в один блок, обновляются только данные этого блока, а не всей страницы. Это уменьшает объем передаваемых данных и ускоряет выполнение операций.

Структура Blocks представлена в таблице 3.4.

Таблица 3.4 – Структура таблицы Blocks

Атрибут	Тип	Описание
Block_id	number	Уникальный идентификатор блока
type	string	Тип элементарной единицы. Например, «text» или «image»
content	string	Структурированные данные о данном блоке
pointer_to	number	Указатель на родительский блок
is_element	boolean	Флаг элемента блока «база данных»
sub_pointer_to	number	Указатель на родительский блок, типа «база данных»
database_y	number	Координата по оси y элемента
database_x	number	Координата по оси x элемента
page_id	number	Уникальный идентификатор страницы, в которой находится данный блок

Первичным ключом является атрибут Block_id.

Страница состоит из блоков, поэтому таблица Blocks принадлежит Pages. Также Blocks связаны с Files отношением «один ко многим».

3.1.5 Таблица WorkspaceMembers описывает уровень доступа

пользователей к страницам. Структура `WorkspaceMembers` представлена в таблице 3.5.

Таблица 3.5 – Структура таблицы `WorkspaceMembers`

Атрибут	Тип	Описание
<code>role</code>	<code>string</code>	Роль пользователя.
<code>created_at</code>	<code>Date</code>	Дата присоединения пользователя
<code>updated_at</code>	<code>Date</code>	Дата обновления уровня доступа пользователя
<code>user_id</code>	<code>number</code>	Уникальный идентификатор пользователя, который связан со страницей
<code>page_id</code>	<code>number</code>	Уникальный идентификатор страницы

Первичным ключом являются атрибуты `user_id` и `page_id`.
Данную таблицу можно считать связующей между `Pages` и `User`.

3.1.6 Таблица `Files` описывает медиа, которое принадлежит определенному блоку. Структура `Files` представлена в таблице 3.6.

Таблица 3.6 – Структура таблицы `Files`

Атрибут	Тип	Описание
<code>File_id</code>	<code>number</code>	Уникальный идентификатор файла
<code>file_url</code>	<code>string</code>	Ссылка на файл
<code>size</code>	<code>number</code>	Размер файла
<code>created_at</code>	<code>Date</code>	Дата создания файла
<code>uploaded_by</code>	<code>number</code>	Уникальный идентификатор пользователя, который загрузил файл
<code>block_id</code>	<code>number</code>	Уникальный идентификатор блока, с которым связан файл

Первичным ключом является атрибут `File_id`.

Таблица медиа имеет связь «один ко многим» с `Blocks` и `User`, так как клиент может загружать много файлов, а блок может содержать несколько медиа.

3.1.7 Таблица `Comments` описывает комментарии пользователей, оставленные к различным страницам или блокам. Структура `Comments` представлена в таблице 3.7.

Таблица 3.7 – Структура таблицы `Comments`

Атрибут	Тип	Описание
1	2	3
<code>Comment_id</code>	<code>number</code>	Уникальный идентификатор комментария

Продолжение таблицы 3.7

1	2	3
content	string	Содержимое комментария
created_at	Date	Дата создания комментария
updated_at	Date	Дата обновления комментария
block_id	number	Уникальный идентификатор блока, к которому относится комментарий
user_id	number	Уникальный идентификатор пользователя, который оставил комментарий

Первичным ключом является атрибут `Comment_id`.

У блока или пользователя может быть много комментариев, поэтому с данными таблицами `Comments` имеет связь «один ко многим».

3.1.8 Таблица `UserSettings` хранит служебную информацию о пользователе. Структура `UserSettings` представлена в таблице 3.8.

Таблица 3.8 – Структура таблицы `UserSettings`

Атрибут	Тип	Описание
Settings_id	number	Уникальный идентификатор настроек пользователя
theme	string	Пользовательская тема пользователя
picture_url	string	Ссылка на изображения пользователя
font	string	Шрифт текста в системе
font_size	number	Размер шрифта
user_id	number	Уникальный идентификатор пользователя

Первичным ключом является атрибут `Settings_id`.

У одной учетной записи одни настройки, поэтому связь «один к одному» между `User` и `UserSettings`.

3.1.9 Таблица `ChangeHistory` будет хранить информацию о том, кто, когда и какие изменения внес в содержимое страниц. Структура `ChangeHistory` представлена в таблице 3.9.

Таблица 3.9 – Структура таблицы `ChangeHistory`

Атрибут	Тип	Описание
1	2	3
Change_id	number	Уникальный идентификатор действия
action	string	Тип действия. Например, «create», «update», «delete»
changes	string	Данные об изменениях

Продолжение таблицы 3.9

1	2	3
created_at	Date	Дата и время внесения изменения
block_id	number	Уникальный идентификатор блока, к которому относится изменение
page_id	number	Уникальный идентификатор страницы

Первичным ключом является атрибут `Change_id`.

У сущности может быть множество изменений и в странице множество действий, поэтому таблица истории изменений связана с соответствующими таблицами отношением «многие к одному».

3.1.10 Таблица `Invitations` описывает приглашения, отправленных пользователям для доступа к рабочим пространствам. Структура `Invitations` представлена в таблице 3.10.

Таблица 3.10 – Структура таблицы `Invitations`

Атрибут	Тип	Описание
<code>Invate_id</code>	number	Уникальный идентификатор приглашения
<code>invite_email</code>	string	Электронная почта приглашенного пользователя
<code>role</code>	string	Роль, которую получит приглашенный пользователь.
<code>status</code>	string	Статус приглашения.
<code>token</code>	string	Уникальный ключ для подтверждения
<code>expires_at</code>	Date	Дата и время истечения срока действия приглашения
<code>created_at</code>	Date	Дата и время создания приглашения
<code>page_id</code>	number	Уникальный идентификатор страницы, к которой приглашают
<code>invited_by</code>	number	Уникальный идентификатор пользователя, отправившего приглашение

Первичным ключом является атрибут `Invate_id`.

Таблица связана с `User` и `Pages` отношением «один ко многим».

3.1.11 Таблица `Subscriptions` описывает подписки пользователей. Структура `Subscriptions` представлена в таблице 3.11.

Таблица 3.11 – Структура таблицы `Subscriptions`

Атрибут	Тип	Описание
1	2	3
<code>Subscription_id</code>	number	Уникальный идентификатор подписки

Продолжение таблицы 3.11

1	2	3
access_end_date	Date	Дата конца подписки
status	string	Статус подписки
created_at	Date	Дата и время создания шаблона
user_id	number	Уникальный идентификатор подписчика
page_id	number	Уникальный идентификатор отслеживаемой страницы

Первичным ключом является атрибут `Subscription_id`.

Клиенты имеют возможность отслеживать страницы с интересующим их контентом. У клиента может быть много подписок, а у страницы много подписчиков. Таблицы пользователей и страниц связаны с таблицей подписок отношением «один ко многим».

3.1.12 Таблица `Likes` описывает пользовательские отзывы. Клиенты могут оценивать страницы открытого доступа. Главное поле таблицы – `stars`. Это оценка от одного до пяти баллов.

Связана таблица отношением «один ко многим» с `User` и `Pages`.

3.2 Настройка серверной части

Запуск приложения начинается с выполнения функции `bootstrap`, которая инициализирует основной модуль `AppModule` с помощью `NestFactory` из библиотеки `@nestjs/core`. В процессе инициализации подключается глобальная валидация данных через `ValidationPipe` из `@nestjs/common`, которая обеспечивает автоматическую трансформацию и проверку входящих запросов на соответствие определенным схемам DTO (Data Transfer Object). Порт, на котором запускается сервер, задается через переменные окружения и считывается из `ConfigService`, предоставляемый модулем `@nestjs/config`.

Основной модуль приложения `AppModule` содержит конфигурацию всех необходимых зависимостей и модулей. Здесь производится:

1. Загрузка конфигурации из файла в `configuration` с помощью `ConfigModule`.
2. Настройка подключения к базе данных PostgreSQL.
3. Импорт модулей, отвечающих за различные аспекты работы приложения:

Работа с PostgreSQL реализована с помощью `Sequelize`. `Sequelize` – это современный TypeScript и Node.js ORM [9].

Настройка подключения к базе данных произведена с использованием метода `SequelizeModule.forRootAsync`, что позволяет загружать конфигурацию асинхронно. Параметры подключения берутся из переменных

окружения. Также производится автоматическая загрузка моделей, используемых в проекте. Каждая модель описывает структуру соответствующей таблицы в базе данных, а также связи между таблицами.

Модели в проекте создаются с использованием декораторов `@Table()` и `@Column()` из библиотеки `sequelize-typescript`, которая обеспечивает удобную и типобезопасную интеграцию Sequelize с NestJS.

Каждая модель представляет собой класс, аннотированный декораторами, где `@Table()` указывает, что данный класс является моделью таблицы, а `@Column()` описывает поля таблицы.

Диаграмма классов системы приведена на чертеже ГУИР.400201.063 РР.2.

3.3 Блок управления учетными записями

3.3.1 Класс `AccountModule` представляет собой функциональный модуль, предназначенный для обработки бизнес-логики, связанной с аккаунтами пользователей. Он инкапсулирует соответствующие зависимости, контроллеры и провайдеры.

Модуль описан с использованием декоратора `@Module`, принимающего объект конфигурации, соответствующий интерфейсу `ModuleMetadata`.

Основные атрибуты у всех модулей:

1. `imports` – массив импортируемых модулей, предоставляющих зависимости, необходимые для работы.

2. `controllers` – массив контроллеров, обрабатывающих входящие HTTP-запросы.

3. `providers` – массив сервисов, которые используются для реализации бизнес-логики:

Класс `AccountModule` импортирует: `UsersModule`, `WorkspaceModule`, `MailModule`, `FileModule`.

Контроллер модуля – `AccountController`.

Провайдеры – `AccountService` и `JwtStrategy`.

3.3.2 Класс `AccountController` представляет собой контроллер, отвечающий за обработку HTTP-запросов, связанных с пользовательскими аккаунтами. Он реализует маршруты, обеспечивающие регистрацию, вход в систему, подтверждение электронной почты, сброс пароля, удаление аккаунта, а также обновление пользовательских настроек.

Контроллер аннотирован декоратором `@Controller('api/users')`, который задает префикс маршрутов, обрабатываемых данным классом.

Через конструктор в класс внедряются два сервиса:

1. `AccountService` – содержит бизнес-логику, связанную с аккаунтами.

2. `MailService` – отвечает за отправку электронных писем.

Методы класса:

1. `initSignup(dto: EmailDto)` – метод инициализирует процессы регистрации, генерирует токен подтверждения и отправляет письмо с ссылкой для подтверждения регистрации. Обработывает HTTP-запрос `POST /api/users/init-signup`.

2. `completeSignup({ token: string; password: string })` – метод, который завершает процессы регистрации. По переданному токenu и паролю создается новый пользователь. Обработывает `POST /api/users/complete-signup`.

3. `confirmEmail(token: string)` – метод, который проверяет токены и возвращает связанный с ними email. Обработывает `GET /api/users/confirm-email`.

4. `loginUsers(dto: LoginUserDto)` – производит авторизацию пользователей. Обработывает `GET /api/users/login`.

5. `destroyUsers(user_id: number)` – удаляет аккаунты пользователей. Защищен декоратором `@UseGuards`, требует авторизации. Обработывает `DELETE /api/users/destroy`.

6. `requestPasswordResets(dto: EmailDto)` – обработывает HTTP-запрос `POST /api/users/request-password-reset` на сброс паролей. Генерирует токен, отправляет письмо с ссылкой на форму сброса.

7. `resetPasswords(user_id: number, dto: PasswordDto)` – устанавливает новые пароли для пользователей. Требуется JWT-авторизация. Обработывает `PATCH /api/users/reset-password`.

8. `updateSettings(user_id: number, dto: UserSettingsDto, file: Express.Multer.File)` – обновляет настройки пользователей, включая загрузку аватара. Использует `FileInterceptor` для обработки файла и `ParseFilePipe` для валидации изображения. Требуется JWT-авторизация. Обработывает `PATCH /api/users/update-settings`.

9. `getPages(user_id: number)` – получает страницы пользователя. Обработывает `GET /api/users/pages`.

10. `getSettings(user_id: number)` – получает настройки пользователя. Обработывает `GET /api/users/settings`.

11. `getWorkspaces(user_id: number)` – получает рабочее пространство пользователя. Обработывает `GET /api/users/workspace`.

12. `getUsers(user_id: number)` – получает пользователя. Обработывает `GET /api/users/user`.

Для обработки загрузки файлов Nest предоставляет встроенный модуль на основе пакета промежуточного программного обеспечения `multer` [10].

3.3.3 Класс `AccountService` представляет собой сервисный класс. Реализует бизнес-логику, связанную с `AccountModule`.

Атрибуты класса:

1. `userService` – экземпляр сервиса `UserService` для работы с пользователями.

2. `workspaceService` – экземпляр сервиса `WorkspaceService` для работы с рабочими пространствами.

3. `fileService` – экземпляр сервиса `FileService` для управления файлами.

4. `cache` – экземпляр сервиса `Cache`, предоставляемый `@nestjs/cache-manager`.

Методы класса:

1. `registerUser (dto: RegisterUserDto)` – метод, который создает аккаунт, настройки, рабочее пространство и первую страницу пользователя. Возвращает объект с JWT-токеном.

2. `generateEmailToken(email: string)` – метод для генерации токена подтверждения email. Принимает email.

3. `getPages(user_id: number)` – метод получения страниц пользователя.

4. `getSetting(user_id: number)` – метод получения настроек пользователя.

5. `getWorkspace(user_id: number)` – метод получения рабочего пространства пользователя.

6. `getUser(user_id: number)` – метод получения пользователя.

7. `verifyEmailToken(token: string)` – метод проверки на наличие токена в кэше.

8. `deleteToken(dto: RegisterUserDto)` – метод удаления токена из кэша.

9. `loginUser(dto: LoginUserDto)` – метод, который выполняет вход пользователя. Возвращает объект с JWT-токеном.

10. `destroyUser(user_id: number)` – метод удаления пользователя по его идентификатору.

11. `sendPasswordResetLink(dto: EmailDto)` – метод, который возвращает JWT-токен для сброса пароля. Принимает email.

12. `resetPassword(dto: PasswordDto, user_id: number)` – метод, который хэширует и обновляет новый пароль пользователя.

13. `updateUserSettings(dto: UserSettingsDto, picture: Express.Multer.File)` – метод обновления настроек профиля пользователя. Загружает новое изображение, если оно передано.

3.3.4 Класс `RegisterUserDto` представляет собой DTO, который используется для передачи данных при регистрации нового пользователя.

Поля класса:

1. `email` – адрес электронной почты пользователя.

2. `name` – строка, представляющая имя пользователя.

3. `password` – строка, представляющая пароль.

Методы отсутствуют, так как класс представляет собой DTO и содержит только свойства с декораторами валидации.

3.3.5 Класс `UserSettingsDto` представляет собой DTO. Предназначен для передачи данных, связанных с пользовательскими настройками интерфейса.

Поля класса:

1. `theme` – тема оформления интерфейса.
2. `picture_url` – ссылка на изображение профиля пользователя.
3. `font` – используемый шрифт интерфейса.
4. `font_size` – размер шрифта.
5. `user_id` – идентификатор пользователя.

3.3.6 Класс `LoginUserDto` представляет собой DTO. Предназначен для передачи данных при авторизации пользователя. Объект имеет два поля: `email` и `password`.

3.3.7 Классы `EmailDto` и `PasswordDto` представляют собой DTO. Предназначены для передачи адреса электронной почты и передачи пароля пользователя соответственно.

3.4 Блок управления рабочим пространством

Блок управления рабочим пространством можно представить из двух модулей: `ManagerModule` и `EventsModule`.

Диаграмма последовательности, иллюстрирующая сценарий приглашения для совместного доступа, приведена на чертеже ГУИР.400201.063 РР.3.

3.4.1 Класс `ManagerModule` представляет собой функциональный модуль, предназначенный для обработки бизнес-логики, связанной с управлением рабочим пространством.

Класс `ManagerModule` импортирует: `UsersModule`, `WorkspaceModule`, `MailModule`, `FileModule`.

Контроллер модуля – `ManagerController`.

Провайдеры – `ManagerService` и `JwtStrategy`.

3.4.2 Класс `ManagerController` представляет собой контроллер, отвечающий за обработку HTTP-запросов, связанных с управлением рабочими пространствами, страницами, комментариями, участниками, подписками и ролями пользователей.

Контроллер аннотирован декоратором `@Controller('api/manager')`.

Через конструктор в класс внедряются два сервиса:

1. `ManagerService` – содержит бизнес-логику, связанную с управлением рабочим пространством.

2. `MailService` – отвечает за отправку электронных писем, включая приглашения.

Методы класса:

1. `createPages(user_id: number, dto: CreatePageDto)` – создает новые страницы. Обработывает `POST /api/manager/create-page`.

2. `updatePages(user_id: number, dto: UpdatePageDto)` – обновляет существующие страницы. Обработывает `PATCH /api/manager/update-page`.

3. `updatePageImgs(user_id: number, query: PageImgQueryDto, file: Express.Multer.File)` – загружает изображения аватара и фонов страниц. Обработывает `PATCH /api/manager/page-img`.

4. `deletePages(user_id: number, dto: DeletePageDto)` – удаляет страницы. Обработывает `DELETE /api/manager/destroy-page`.

5. `restorePages(user_id: number, dto: RestorePagesDto)` – восстанавливает ранее удаленные страницы. Обработывает `PATCH /api/manager/restore-page`.

6. `inviteUsers(user_id: number, dto: InviteUsersDto)` – отправляет приглашения для совместной работы по электронной почте. Обработывает `POST /api/manager/invite-user`.

7. `confirmInvitations(user_id: number, query: ConfirmInvitationsDto)` – подтверждает участие приглашенных пользователей. Обработывает `POST /api/manager/confirm-invitation`.

8. `destroyMembers(user_id: number, dto: DeleteMemberDto)` – удаляет участников из рабочего пространства. Обработывает `DELETE /api/manager/destroy-member`.

9. `changeRoles(user_id: number, dto: ChangeRolesDto)` – изменяет роль пользователей в рабочем пространстве. Обработывает `PATCH /api/manager/change-role`.

10. `createComments(user_id: number, dto: CreateCommentsDto)` – добавляет комментарии к странице. Обработывает `POST /api/manager/create-comment`.

11. `destroyComments(user_id: number, dto: DeleteCommentsDto)` – удаляет комментарии. Обработывает `DELETE /api/manager/destroy-comment`.

12. `subscribeToPages(user_id: number, dto: SubscriptionDto)` – подписывает пользователей на обновления страницы. Обработывает `POST /api/manager/subscribe`.

13. `unsubscribeToPages(user_id: number, dto: SubscriptionDto)` – отменяет подписки на страницу. Обработывает `DELETE /api/manager/unsubscribe`.

14. `likePages(user_id: number, dto: LikeDto)` – ставит оценки на страницу. Обрабатывает POST `/api/manager/like`.

15. `deleteLikePages(user_id: number, dto: LikeDto)` – удаляет оценки со страницы. Обрабатывает DELETE `/api/manager/destroy-like`.

16. `getMembers(user_id: number, page_id: number)` – получает участников страницы. Обрабатывает GET `/api/manager/members`.

17. `getComments(user_id: number, block_id: number)` – получает комментарии блока. Обрабатывает GET `/api/manager/get-comments`.

3.4.3 Класс `ManagerService` представляет собой сервисный класс. Реализует бизнес-логику, связанную с `ManagerModule`.

Атрибуты класса:

1. `workspaceService` – экземпляр сервиса `WorkspaceService`.

2. `userService` – экземпляр сервиса `UsersService`.

3. `fileService` – экземпляр сервиса `FileService`.

Методы класса:

1. `createNewPage(user_id: number, dto: CreatePageDto)` – создает новую страницу в рабочем пространстве пользователя. Возвращает объект новой страницы.

2. `parentPageIdCheck(parent_page_id: number | null, workspaceId: number)` – проверяет корректность родительской страницы. Возвращает идентификатор родительской страницы или `null`.

3. `updatePages(user_id: number, dto: UpdatePageDto)` – обновляет данные страницы, в том числе указатель на родительскую страницу. Возвращает обновленную страницу.

4. `updatePageImg(user_id: number, type: string, file: Express.Multer.File)` – обновляет аватар или картинку страницы. Возвращает обновленную страницу.

5. `deletePages(user_id: number, dto: DeletePageDto)` – удаляет страницу и ее дочерние страницы рекурсивно. Возвращает объект с флагом успешного удаления.

6. `restorePage(user_id: number, dto: RestorePagesDto)` – восстанавливает удаленную страницу и ее дочерние страницы рекурсивно. Возвращает объект с флагом успешного восстановления.

7. `inviteUser(user_id: number, dto: InviteUsersDto)` – приглашает пользователя присоединиться к странице. Возвращает объект с данными приглашения и email владельца страницы.

8. `confirmInvitation(token: string, user_id: number)` – подтверждает приглашение на страницу. Возвращает объект сообщение об успешном подтверждении.

9. `getComments(block_id: number, user_id: number)` – возвращает комментарии блока.

10. `getMembers(page_id: number, user_id: number)` –

возвращает участников страницы.

11. `destroyMember (user_id: number, dto: DeleteCommentsDto)` – удаляет участника со страницы. Возвращает результат удаления участника.

12. `changeRole (user_id: number, dto: ChangeRolesDto)` – изменяет роль участника страницы. Возвращает обновленные данные участника.

13. `createComment (user_id: number, dto: CreateCommentsDto)` – создает комментарий к блоку страницы. Возвращает созданный комментарий.

14. `destroyComment (user_id: number, dto: DeleteCommentsDto)` – удаляет комментарий, если он принадлежит пользователю. Возвращает результат удаления комментария.

15. `subscribeToPage (user_id: number, dto: SubscriptionDto)` – оформляет подписку на публичную подписную страницу. Возвращает результат создания подписки.

16. `unsubscribeToPage (user_id: number, dto: SubscriptionDto)` – отменяет подписку на страницу, если подписка существует. Возвращает результат удаления подписки.

17. `likePage (user_id: number, dto: LikeDto)` – ставит оценку публичной странице. Возвращает результат создания или пересоздания оценки.

18. `deleteLikePage (user_id: number, dto: LikeDto)` – удаляет оценку с публичной страницы. Возвращает результат удаления лайка, если он существовал.

3.4.4 Класс `CreatePageDto` представляет собой DTO. Используется для создания новой страницы в рабочем пространстве. У объекта одно поле – `parent_page_id`. Это идентификатор родительской страницы, положительное число или `null`.

3.4.5 Класс `PageImgQueryDto` представляет собой DTO. Используется для передачи параметров обновления изображения страницы. Содержит поля `type` и `page_id`.

3.4.6 Класс `UpdatePageDto` представляет собой DTO. Используется для обновления данных страницы в рабочем пространстве пользователя.

Поля класса:

1. `page_id` – идентификатор страницы.

2. `title` – заголовок страницы.

3. `description` – описание страницы.

4. `type` – тип страницы. Например, «Doc» или «Subscription».

5. `category` – категория страницы.

6. `parent_page_id` – идентификатор родительской страницы.

3.4.7 Класс `DeletePageDto` представляет собой DTO. Используется для удаления страницы пользователя и всех ее дочерних страниц.

Поля класса:

1. `page_id` – идентификатор удаляемой страницы.
2. `force` – флаг принудительного удаления.

3.4.8 Класс `RestorePagesDto` представляет собой DTO. Используется для восстановления ранее удаленной страницы и всех ее дочерних страниц. Необходим для передачи идентификатора восстанавливаемой страницы.

3.4.9 Класс `CreateWorkspaceMemberDto` представляет собой DTO. Используется для добавления участника к странице рабочего пространства с определенной ролью.

Поля класса:

1. `user_id` – идентификатор пользователя.
2. `page_id` – идентификатор страницы, к которой добавляется участник.
3. `role` – роль пользователя на странице. Допустимые значения: «owner», «editor», «view», «comment».

3.4.10 Класс `InviteUsersDto` представляет собой DTO. Используется для приглашения пользователя на страницу с указанием роли.

Поля класса:

1. `invite_email` – электронная почта пользователя.
2. `role` – роль, предоставляемая приглашенному пользователю.
3. `page_id` – идентификатор страницы.

3.4.11 Класс `ConfirmInvitationsDto` – DTO, используемый для подтверждения приглашения пользователя на страницу. Содержит токен приглашения.

3.4.12 Класс `DeleteMemberDto` – DTO, используемый для удаления участника со страницы. Содержит идентификатор участника и идентификатор страницы.

3.4.13 Класс `ChangeRolesDto` – DTO, используемый для изменения роли участника страницы. Содержит идентификатор участника, идентификатор страницы и новую роль.

3.4.14 Класс `CreateCommentsDto` – DTO, используемый для создания комментария к блоку страницы. Содержит идентификатор блока и

текст комментария.

3.4.15 Класс `DeleteCommentsDto` – DTO, используемый для удаления собственного комментария. Содержит идентификатор комментария.

3.4.16 Класс `SubscriptionDto` – DTO, используемый для подписки или отписки от страницы. Содержит идентификатор страницы.

3.4.17 Класс `LikeDto` – DTO, используемый для лайка страницы. Содержит идентификатор страницы и, опционально, количество звезд.

3.4.18 Класс `EventsModule` представляет собой функциональный модуль, предназначенный для обработки бизнес-логики, связанной с редактированием страниц на уровне блоков.

Класс `EventsModule` импортирует: `ScheduleModule`, `TokenModule`, `WorkspaceModule`.

`ScheduleModule` позволяет выполнение произвольного кода через повторяющиеся интервалы или один раз после указанного интервала. Предоставляется модуль из `@nestjs/schedule`.

Провайдеры – `EventsGateway` и `EventsService`.

3.4.19 Класс `EventsGateway` представляет собой `WebSocket`-шлюз, отвечающий за обработку событий в реальном времени, связанных со следующими процессами:

- управлением подключениями и отключениями клиентов;
- синхронизацией изменений на страницах между пользователями;
- обработкой операций с блоками и элементами страниц;
- поддержанием согласованного состояния данных между клиентами.

Контроллер аннотирован декоратором `@WebSocketGateway(3002)` с поддержкой CORS.

Дополнительно используется `ValidationPipe` для валидации входящих сообщений.

Через конструктор в класс внедряется сервис `EventsService`, который содержит бизнес-логику для работы с `WebSocket`-соединениями и обработки операций.

Основные свойства класса:

1. `server` – экземпляр `Server` из `socket.io` [11], используемый для отправки событий клиентам.
2. `clientsStorage` – хранилище активных соединений клиентов.
3. `transactionsStorage` – хранилище операций.
4. `listStorage` – хранилище структуры страниц.
5. `listStorageElements` – хранилище структуры элементов страниц.

6. `lockedRooms` – заблокированные комнаты.

Методы класса:

1. `afterInit(server: Server)` – выполняется при инициализации сервера. Обрабатывает подключение клиента, выполняет валидацию токена, добавляет пользователя в комнату, сохраняет клиента в `clientsStorage`.

2. `handleConnection(client: Socket)` – выполняется при подключении клиента. Загружает блоки страницы и ее структуру с помощью `EventsService`, инициализирует список и элементы, рассылает клиентам события «page-init», «page-init-list», «page-init-elements» и «page-init-list-elements».

3. `handleDisconnect(client: Socket)` – выполняется при отключении клиента. Удаляет данные о клиенте.

4. `handleTransaction(client: Socket, operation: Operation)` – обрабатывает событие «add-operation». Сохраняет переданную операцию в `transactionsStorage` для дальнейшей пакетной обработки.

5. `handleScheduledProcessing()` – выполняется по расписанию каждые 5 секунд, аннотирован декоратором `@Cron()`, предоставляемым модулем `@nestjs/schedule`. Обрабатывает накопленные операции в `transactionsStorage` через `EventsService`. Отправляет клиентам обновления через события: «change-list, change-create», «change-update», «change-create-elements», «change-list-elements», «change-update-elements».

3.4.20 Класс `EventsService` представляет собой сервисный класс, отвечающий за обработку бизнес-логики, связанной с операциями в реальном времени в рамках WebSocket-соединений.

Основная задача класса – обеспечение корректной синхронизации состояния блоков и элементов страниц между пользователями, обработка транзакций и проверка прав доступа.

Атрибуты класса:

1. `workspaceService` – экземпляр сервиса `WorkspaceService`.

2. `tokenService` – экземпляр сервиса `TokenService`.

Методы класса:

1. `verifyUserForWS(token: string | undefined, query: ParsedUrlQuery)` – выполняет проверку пользователя при подключении. При успешной проверке возвращает `payload` токена и идентификатор комнаты.

2. `processStorage(storage: TransactionStorage, userRoom: number, list: DoublyLinkedList<number>, listElements: Set<number>)` – выполняет пакетную обработку накопленных операций с блоками и элементами страницы. Возвращает обновленный список изменений для отправки клиентам.

3. `filterUpdateOps` – фильтрует операции обновления. Принимает

`updateOps: Operation<UpdateOperationArgs>[]` и
соответствующие параметры.

4. `filterDeleteOps` – фильтрует операции удаления.
5. `filterCreateOps` – фильтрует операции создания.
6. `filterChangePosOps` – фильтрует операции изменения порядка блоков.
7. `filterCreateInDbOps` – фильтрует операции создания элемента в специализированных блоках.
8. `filterDeleteInDbOps` – фильтрует операции удаления элемента в специализированных блоках.
9. `getBlocksAndStructure(roomId: number)` – получает список блоков и элементов страницы по `roomId`. Возвращает отсортированные блоки и их идентификаторы для формирования структуры страницы на клиентской стороне.
10. `getUserRoom(client: Socket)` – определяет комнату пользователя на основании WebSocket-соединения.

3.4.21 Класс `TransactionsStorage` отвечает за хранение и управление операциями, выполняемыми в процессе работы с блоками и элементами страниц. Используется для накопления и фильтрации операций до их пакетной обработки и отправки клиентам.

Свойства класса:

1. `operations` – массив объектов `Operation[]`, в котором хранятся все поступившие операции. Аннотирован декораторами `@IsArray()` и `@ValidateNested` для валидации, а также `@Type(() => Operation)` для преобразования входных данных.

Методы класса:

1. `clear()` – очищает список накопленных операций.
2. `addOperation(operation: Operation)` – добавляет новую операцию в хранилище.
3. `getCount()` – возвращает общее количество операций, находящихся в хранилище.
4. `getUpdateOperations()` – возвращает массив операций с командой «update».
5. `getCreateOperations()` – возвращает массив операций с командой «create», предназначенных для создания новых блоков.
6. `getChangePosOperations()` – возвращает массив операций с командой «change-position», которые фиксируют перемещения блоков.
7. `getDeleteOperations()` – возвращает массив операций с командой «delete», связанных с удалением блоков.
8. `getCreateInDbOperations()` – возвращает массив операций с командой «create-in-db», которые предназначены для создания элементов.
9. `getDeleteInDbOperations()` – возвращает массив операций с

командой «delete-in-db», отвечающих за удаление.

3.4.22 Класс `Operation` представляет собой универсальную модель операции, которая описывает действие, выполняемое пользователем или системой при работе с блоками и элементами страницы.

Свойства класса:

1. `userAction` – опциональное текстовое поле, в котором может храниться описание пользовательского действия.

2. `command` – поле, определяющее тип выполняемой операции. Обязательно к заполнению.

3. `args` – параметры операции. Тип аргументов зависит от значения `command`.

3.4.23 Класс `BaseBlockArgs` представляет собой класс, который используется для представления аргументов блоков, где каждый блок может иметь различные типы содержимого в зависимости от его типа.

Свойства класса:

1. `type` – поле, которое определяет, какого типа контент будет храниться в блоке.

2. `content` – поле, которое хранит содержание блока. Содержание зависит от типа блока.

3.4.24 Классы `CreateOperationArgs`, `UpdateOperationArgs`, `DeleteOperationArgs`, `ChangePosOperationArgs`, `CreateInDbOperationArgs`, `DeleteInDbOperationArgs` представляют аргументы различных операций, которые могут быть выполнены с блоками на страницах. Они служат для передачи данных о изменениях в структуре страниц, таких как создание, обновление, удаление и изменение позиции блоков.

`CreateOperationArgs` имеет поле `pointer_to`, которое указывает на родительский блок, за которым будет создана сущность. Также класс наследует `BaseBlockArgs`.

`UpdateOperationArgs` позволяет обновлять блок, имеет поля `block_id`, флаг `is_element`, а также обновленное содержимое блока.

`DeleteOperationArgs` используется для удаления блока, где указывается только `block_id`.

`ChangePosOperationArgs` описывает операцию изменения позиции блока, где указывается `block_id` и `pointer_to`, новая позиция, на которую блок должен быть перемещен.

`CreateInDbOperationArgs` описывает операцию создания элемента в блоке базы данных. В классе указываются идентификатор блока в базе, а также его тип.

`DeleteInDbOperationArgs` описывает операцию удаления элемента из блока базы данных, с возможностью указания строки и колонки

для более точного удаления.

3.4.25 Класс `TextSpanDto` представляет собой DTO. Используется для описания текстового фрагмента с возможностью указания его форматирования. `TextSpanDto` используется как элемент массива внутри класса `FormattedTextContentDto`. Такой подход позволяет точно воспроизводить сложное форматирование в пользовательском интерфейсе

Поля класса:

1. `text` – текстовая строка, содержащая фрагмент текста.
2. `is_bold` – флаг, указывающий, выделен ли текст жирным шрифтом.
3. `is_ital` – флаг, указывающий, применен ли к тексту курсив.
4. `is_under` – флаг, указывающий, подчеркнут ли текст.
5. `is_strike` – флаг, указывающий, перечеркнут ли текст.
6. `is_highlight` – флаг, указывающий, подсвечен ли текст фоном.

3.4.26 Класс `FormattedTextContentDto` представляет собой DTO. Используется для хранения набора фрагментов форматированного текста, каждый из которых описывается объектом `TextSpanDto`.

3.4.27 Класс `ListContentDTO` – DTO, используемый для описания списка, содержащего тип списка и его текстовое содержимое.

3.4.28 Класс `CodeContentDTO` – DTO, используемый для описания текстового блока с кодом, включая его тип и содержимое.

3.4.29 Класс `TableContentDTO` – DTO, используемый для описания таблицы с указанием количества строк, столбцов и содержимого.

3.4.30 Класс `TaskContentDTO` – DTO, используемый для описания задачи с возможностью хранения текстового содержимого.

3.4.31 Класс `DataBaseComponentDto` представляет собой DTO. Используется для описания компонента базы данных с определенным макетом отображения.

Поля класса:

1. `layout_type` – строка, определяющая тип макета компонента базы данных. Может принимать значения: «Graph», «Donut», «Board», «Default».
2. `args` – поле, объект, который зависит от выбранного `layout_type`.

3.4.32 Классы `LayoutGraphDto`, `LayoutDonutDto`, `LayoutBoardDto`, `LayoutDefaultDto` описывают конфигурации для различных макетов отображения данных в специальном компоненте «базы данных».

3.5 Блок управления файловым хранилищем

3.5.1 Класс `ExplorerModule` представляет собой функциональный модуль, предназначенный для обработки бизнес-логики, связанной с работой с файлами в рамках рабочего пространства.

Класс `ExplorerModule` импортирует: `UsersModule`, `WorkspaceModule`, `FileModule`.

Контроллер модуля – `ExplorerController`.

Провайдеры – `ExplorerService` и `JwtStrategy`.

3.5.2 Класс `ExplorerController` представляет собой контроллер, отвечающий за обработку HTTP-запросов, связанных с загрузкой, получением, удалением и экспортом файлов, привязанных к блокам и страницам.

Контроллер аннотирован декоратором `@Controller('api/explorer')`.

Через конструктор в класс внедряется сервис `ExplorerService` – содержит бизнес-логику, связанную с работой с файлами.

Методы класса:

1. `addFiles(user_id: number, file: Express.Multer.File, block_id: number)` – загружает файлы, связанный с определенным блоком. Обработывает `POST /api/explorer/add-file`.

2. `getFiles(user_id: number, file_id: number, block_id: number, res: Response)` – возвращает файлы пользователей в виде потока для скачивания. Обработывает `GET /api/explorer/get-file`.

3. `deleteFiles(user_id: number, block_id: number, file_id: number)` – удаляет файлы, связанные с блоком. Обработывает `DELETE /api/explorer/delete-file`.

4. `exportPages(user_id: number, page_id: number, res: Response)` – экспортирует содержимое страницы в ZIP-архив. Обработывает `GET /api/explorer/export-page`.

3.5.3 Класс `ExplorerService` представляет собой сервисный класс. Реализует бизнес-логику, связанную с `ExplorerModule`.

Атрибуты класса:

1. `workspaceService` – экземпляр сервиса `WorkspaceService`.

2. `fileService` – экземпляр сервиса `FileService`.

Методы класса:

1. `addFile(file: Express.Multer.File, user_id: number, block_id: number)` – загружает файл в указанный блок после проверки прав доступа. Возвращает информацию о созданной записи файла.

2. `getSafeFilename(s3Response:`

GetObjectCommandOutput) – возвращает безопасное имя файла на основе метаданных или типа содержимого.

3. getFile(user_id: number, block_id: number, file_id: number) – проверяет доступ и возвращает объект файла из хранилища.

4. deleteFile(file_id: number, user_id: number, block_id: number) – проверяет права доступа и удаляет файл как из базы данных, так и из хранилища.

5. validateFileAccess(user_id: number, block_id: number, file_id: number, permission: string) – вспомогательный метод для валидации доступа пользователя к файлу. Возвращает блок, список файлов и нужный файл.

6. exportPage(user_id: number, page_id: number) – экспортирует все блоки страницы, включая файлы, в ZIP-архив. Возвращает поток архива.

3.6 Блок сервисов

3.6.1 Класс TokenService представляет собой сервисный класс. Реализует бизнес-логику, связанную с генерацией и верификацией JWT-токенов.

Атрибуты класса:

1. jwtService – экземпляр сервиса JwtService, используется для создания и проверки JWT-токенов. Предоставлен библиотекой @nestjs/jwt.

2. configService – экземпляр сервиса ConfigService.

Методы класса:

1. generateJwtToken(user_id: number) – создает JWT-токен на основе идентификатора пользователя. Возвращает строку с токеном.

2. verifyJwtToken(token: string) – проверяет подлинность JWT-токена. Возвращает полезную нагрузку в виде JwtPayloadDto.

3.6.2 Класс JwtStrategy представляет собой стратегию авторизации. Реализует механизм верификации JWT-токенов через модуль Passport, который предоставлен библиотекой @nestjs/passport. Используется для защиты маршрутов в @UseGuards и проверки подлинности пользователей.

Атрибуты класса:

1. configService – экземпляр сервиса ConfigService.

2. userService – экземпляр сервиса UsersService.

Методы класса:

1. validate(payload: JwtPayloadDto) – проверяет наличие пользователя, идентификатор которого указан в полезной нагрузке токена. В случае отсутствия пользователя выбрасывает исключение

UnauthorizedException. Возвращает объект с user_id.

3.6.3 Класс MailService представляет собой сервисный класс. Реализует бизнес-логику, связанную с отправкой электронной почты. Для реализации используется nodemailer. Nodemailer – это модуль для приложений Node.js, который позволяет легко отправлять электронные письма, используя Simple Mail Transfer Protocol (SMTP) [12]. В данном модуле будет использоваться сервис Gmail [13].

Атрибуты класса:

1. configService – экземпляр сервиса ConfigService.
2. transporter – экземпляр nodemailer.Transporter, используется для отправки писем через SMTP-сервер.

Методы класса:

1. sendMail(sendMailDto: SendMailDto) – отправляет электронное письмо на указанный адрес. Использует данные из SendMailDto, включая получателя, тему, текстовое и HTML-содержимое.

3.6.4 Класс FileService представляет собой сервисный класс. Реализует бизнес-логику, связанную с загрузкой, хранением и удалением файлов в хранилище MinIO и управлением записями о файлах в базе данных.

Для работы с хранилищем используется AWS SDK [14]. Пакет средств разработки ПО (SDK) – это набор инструментов для разработчиков, ориентированных на платформу. Хранилище MinIO совместимо с Amazon S3 [15].

Атрибуты класса:

1. s3Client – экземпляр класса S3Client для взаимодействия с MinIO.
2. publicBucket – название публичного бакета.
3. privateBucket – название приватного бакета.
4. filesRepository – экземпляр модели Files, используется для взаимодействия с таблицей файлов в базе данных. Внедряется с помощью декоратора @InjectModel() из @nestjs/sequelize.
5. configService – экземпляр сервиса ConfigService.

Методы класса:

1. uploadFile(fileName: string, fileBuffer: Buffer, contentType: string, isPublic: boolean) – загружает файл в указанный публичный или приватный бакет MinIO.
2. getFile(fileName: string, isPublic: boolean) – получает файл из соответствующего бакета.
3. getPresignedUrl(fileName: string, expiresIn: number) – генерирует подписанную ссылку на приватный файл.
4. listFiles(isPublic: boolean) – возвращает список всех файлов в соответствующем бакете.

5. `deleteFile(fileName: string, isPublic: boolean)` – удаляет файл из соответствующего бакета.

6. `generateFileName(file: Express.Multer.File, user_id: number, isPublic: boolean)` – генерирует уникальное имя для загружаемого файла.

7. `createFileRecord(name: string, size: number, block_id: number, user_id: number)` – создает запись о файле в базе данных.

8. `findFilesRecByBlockId(block_id: number)` – возвращает все записи о файлах, привязанных к указанному блоку.

9. `destroyFileRec(file_id: number)` – удаляет запись о файле по идентификатору.

3.6.5 Класс `UserService` представляет собой сервисный класс. Реализует бизнес-логику, связанную с пользователями и их настройками.

Атрибуты класса:

1. `userRepository` – экземпляр модели `User`, используется для взаимодействия с таблицей пользователей.

2. `userSettingsRepository` – экземпляр модели `UserSettings`, используется для управления настройками пользователей.

3. `tokenService` – экземпляр сервиса `TokenService`.

Методы класса:

1. `passwordHash(password: string)` – хэширует пароль с помощью `Argon2`. Возвращает хэш строки.

2. `passwordVerify(hash: string, password: string)` – проверяет соответствие пароля и хэша. Возвращает булев результат.

3. `findUserByEmail(email: string)` – возвращает пользователя по email. Возвращает объект `User` или `null`.

4. `findUserById(User_id: number)` – возвращает пользователя по идентификатору. Возвращает объект `User` или `null`.

5. `findUserSettingsByUserId(User_id: number)` – возвращает настройки пользователя по его идентификатору. Возвращает `UserSettings` или `null`.

6. `createUser(dto: RegisterUserDto)` – создает нового пользователя. Возвращает объект `User`.

7. `generateToken(user: User)` – генерирует JWT-токен для пользователя. Возвращает строку токена.

8. `deleteUser(user_id: number)` – удаляет пользователя по идентификатору. Возвращает количество удаленных записей.

9. `updateUserPassword(newPassword: string, user_id: number)` – обновляет пароль пользователя.

10. `createUserSettings(dto: UserSettingsDto)` – создает настройки пользователя. Возвращает объект `UserSettings`.

Argon2 – это современный криптографический алгоритм хеширования паролей, признанный одним из самых безопасных на сегодняшний день.

3.6.6 Класс `WorkspaceService` представляет собой сервисный класс. Реализует бизнес-логику, связанную с рабочими пространствами, страницами, блоками, комментариями, подписками, лайками, участниками, приглашениями и историей изменений. Используется для работы с различными сущностями в рамках модели данных приложения.

Атрибуты класса:

1. `workspaceRepository` – экземпляр модели `Workspace`, используется для работы с рабочими пространствами.
2. `pageRepository` – экземпляр модели `Pages`, используется для работы со страницами.
3. `workspaceMembersRepository` – экземпляр модели `WorkspaceMembers`, используется для управления участниками страниц.
4. `blocksRepository` – экземпляр модели `Blocks`, используется для работы с блоками.
5. `invitationsRepository` – экземпляр модели `Invitations`, используется для управления приглашениями пользователей.
6. `changeHistoryRepository` – экземпляр модели `ChangeHistory`, используется для хранения истории изменений.
7. `commentsRepository` – экземпляр модели `Comments`, используется для работы с комментариями.
8. `subscriptionsRepository` – экземпляр модели `Subscriptions`, используется для управления подписками на страницы.
9. `likesRepository` – экземпляр модели `Likes`, используется для управления лайками и рейтингами.
10. `sequelize` – экземпляр `Sequelize`, используется для управления транзакциями и взаимодействия с базой данных на более низком уровне.

Методы класса:

1. `findWorkspaceByUserId(user_id: number)` – возвращает рабочее пространство, принадлежащее пользователю.
2. `findPageById(page_id: number, includeDeleted: boolean)` – возвращает страницу по ID, с опцией учета удаленных страниц.
3. `findPagesByParentId(parent_page_id: number, includeDeleted: boolean)` – возвращает дочерние страницы по ID родительской страницы.
4. `restorePage(pageId: number)` – восстанавливает ранее удаленную страницу.
5. `findInvitationByToken(token: string)` – возвращает активное приглашение по токenu.
6. `findCommentById(comment_id: number)` – возвращает комментарий по ID.

7. `findSubscriptionById(user_id: number, page_id: number)` – возвращает подписку пользователя на страницу.
8. `findLikeById(user_id: number, page_id: number)` – возвращает лайк, оставленный пользователем на странице.
9. `checkRightConnectToPage(user_id: number, page_id: number, mode: string)` – проверяет наличие прав доступа пользователя к странице в заданном режиме.
10. `getUserPagesByUserId(user_id: number)` – возвращает все страницы, с которыми связан пользователь.
11. `createWorkspace(user_id: number)` – создает новое рабочее пространство с базовой настройкой.
12. `createPage(parent_page_id: number | null, workspace_id: number)` – создает новую страницу в заданном рабочем пространстве.
13. `updatePage(dto: UpdatePageDto, avatar_url: string, picture_url: string, page: Pages)` – обновляет параметры страницы.
14. `deletePage(page_id: number, force: boolean)` – удаляет страницу, с возможностью полного удаления.
15. `createComment(content: string, block_id: number, user_id: number)` – создает новый комментарий к блоку.
16. `deleteComment(comment_id: number)` – удаляет комментарий по ID.
17. `createWorkspaceMember(dto: CreateWorkspaceMemberDto)` – добавляет пользователя в качестве участника страницы.
18. `deleteWorkspaceMember(user_id: number, page_id: number)` – удаляет участника со страницы.
19. `updateWorkspaceMember(user_id: number, page_id: number, newRole: string)` – обновляет роль участника страницы.
20. `createSubscription(dto: SubscriptionDto, user_id: number)` – создает подписку пользователя на страницу.
21. `deleteSubscription(dto: SubscriptionDto, user_id)` – удаляет подписку пользователя на страницу.
22. `createLike(dto: LikeDto, user_id: number)` – создает оценку для страницы.
23. `deleteLike(dto: LikeDto, user_id: number)` – удаляет оценку с заданной страницы.
24. `createBlock(dto: CreateBlockDto, transaction: Transaction)` – создает новый блок в рамках страницы или базы данных.
25. `createInvitation(dto: InviteUsersDto, token: string, user_id: number)` – создает приглашение пользователя к странице с ограничением по сроку действия.

26. `createHistory(action: string, changes: Record<string, any>, page_id: number, block_id: number, transaction: Transaction)` – сохраняет запись об изменениях в блоках или страницах.

27. `getBlockByPageId(page_id: number)` – возвращает отсортированные блоки и элементы страницы.

28. `updateBlocks(updateOps: Operation<UpdateOperationArgs>[])` – обновляет содержимое блоков с учетом их типа и состояния, включая особую обработку баз данных.

29. `isDatabaseContent(content: unknown)` – проверяет, является ли переданное содержимое содержимым базы данных.

30. `hasProperty(dbBlock: Blocks, property: string)` – проверяет наличие свойства (столбца) у указанной базы данных.

31. `createBlocks(createOps: Operation<CreateOperationArgs>[], page_id: number)` – создает набор блоков на странице, при необходимости добавляет таблицу и первый столбец.

32. `createColumnOfDb(dbBlock: Blocks, typeProperty: string)` – создает столбец в таблице (базе данных), включая координаты и тип данных.

33. `createRowOfDb(dbBlock: Blocks)` – создает строку в таблице, основываясь на координатах X и уже существующих свойствах.

34. `findDatabaseX(subPointerTo: number)` – возвращает уникальные координаты X в таблице.

35. `findDatabaseY(subPointerTo: number)` – возвращает уникальные координаты Y в таблице.

36. `decodeTypeProperty(typeProperty: string)` – преобразует имя свойства в тип блока.

37. `changePointers(changes)` – изменяет указатели `pointer_to` для набора блоков.

38. `deleteBlocks(deleteBlockIds: Set<number>, page_id: number)` – удаляет указанные блоки и связанные с ними элементы таблиц (строки, столбцы).

39. `findBlockById(block_id: number)` – находит блок по его ID.

40. `createInDb(rowOps, columnOps, listElements: Set<number>)` – создает строки и столбцы в таблицах, обновляет список измененных блоков.

41. `deleteColumnInDb(dbBlock: Blocks, column: number)` – удаляет указанный столбец таблицы и возвращает удаленные ID.

42. `deleteRowInDb(dbBlock: Blocks, row: number)` – удаляет указанную строку таблицы и возвращает удаленные ID.

43. `deleteInDb(delRowOps, delColumnOps)` – удаляет строки и столбцы в таблицах на основе переданных операций.

44. `getUsersByPageId(page_id: number)` – получает всех участников страницы.

45. `findCommentsByBlockId(block_id: number)` – получает все комментарии, которые относятся к соответствующему блоку.

3.7 Клиентская часть

Структура приложения на клиентской стороне строится вокруг корневого `layout` и компонентов, которые определяют глобальные стили, контексты и базовую HTML-разметку.

Файл `layout.tsx` – это корневой слой интерфейса. Он отображается один раз и сохраняется при навигации между страницами. Это отличает его от `page.tsx`, который перерисовывается при каждой навигации.

Next.js предлагает гибкий способ описания маршрутов и логики отображения. Каждый сегмент пути в URL соответствует папке `app/`, а специальный файл `page.tsx` в этой папке отвечает за содержимое и оформление.

Также для конфигурации приложения на уровне сборки и сервера используется специальный файл `next.config.ts`. Здесь задается поведение среды, политика загрузки изображений, маршруты и другое.

Корневой слой интерфейса обернут в `Providers`. Этот компонент, который используется для подключения глобального состояния через `Redux`. Он оборачивает все приложение в `Provider` из библиотеки `react-redux`, чтобы сделать `Redux`-хранилище доступным на всех уровнях `React`-дерева.

`Redux` – библиотека для JavaScript, предназначенная для предсказуемого и поддерживаемого глобального управления состояниями приложения [16].

3.7.1 Для создания хранилища состояний в клиентской части используется библиотека `Redux Toolkit` [17]. Она обеспечивает предсказуемость, масштабируемость и удобство при работе с состояниями, особенно в сложных пользовательских интерфейсах.

Основные элементы хранилища:

1. `store` – центральное хранилище `Redux`, созданное с помощью `configureStore`.

2. `rootReducer` – объединенный редьюсер, включающий все подмодули состояния.

Редьюсер – это функция, которая определяет, как изменяется состояние хранилища `Redux` в ответ на определенные действия. Она получает текущее состояние и действие, и возвращает новое состояние.

Модули состояний:

1. `authSlice` – отвечает за авторизацию пользователя.

2. `userSlice` – хранит информацию о текущем пользователе.

3. `pageSlice` – управляет состоянием текущей страницы. Ее содержимым, структурой, активными блоками и участниками.

Типы и утилиты:

1. `RootState` – тип, описывающий полную структуру состояния.
2. `AppDispatch` – тип для правильной типизации `dispatch`.
3. `AppStore` – тип хранилища.

Вместо стандартных решений создаются собственные обертки:

1. `useAppDispatch` – типизированная версия `useDispatch`, чтобы TypeScript распознователь знал, какие действия можно отправлять на основе `AppDispatch`.

2. `useAppSelector` – типизированная версия `useSelector`, позволяющая безопасно получать данные из хранилища `RootState`, с проверкой типов.

Эти хуки повышают удобство и безопасность работы с Redux в приложении.

Хуки – это специальные функции в React, которые позволяют подключаться к возможностям React-компонентов, таким как состояние, жизненный цикл, контексты и другие внутренние механизмы.

3.7.2 Модуль состояний `pageSlice`. Этот модуль отвечает за хранение и управление данными, связанными с текущей страницей и ее содержимым.

Он реализован с помощью `createSlice` из `@reduxjs/toolkit` и включает как синхронные редьюсеры, так и асинхронные действия `thunks`.

Структура состояния `PageState`:

1. `blocks` – массив всех блоков на странице.
2. `structure` – структура расположения блоков.
3. `elements` – массив всех элементов на странице.
4. `structure_elements` – структура для элементов.
5. `page` – объект текущей страницы с основной метаданной.
6. `members` – список участников страницы с их ролями.

Модуль содержит функции для синхронного обновления состояния:

1. `setBlocks` – установка списка блоков.
2. `setStructure` – установка структуры блоков.
3. `setElements` – установка активных элементов.
4. `setStructureElements` – установка структуры элементов.
5. `setPage` – установка данных текущей страницы.

Асинхронные операции `thunks` обрабатываются через `extraReducers`:

1. `getMembers` – получает участников страницы и сохраняет их в `members`.
2. `inviteUser` – отправляет приглашение для совместной работы.
3. `destroyMember` – исключает участника из рабочего пространства.
4. `changeRole` – изменяет роль участника

3.7.3 Модуль состояний `userSlice`. Этот модуль отвечает за

управление пользовательскими данными, настройками и страницами, к которым у пользователя есть доступ.

Структура состояния User:

1. email – адрес электронной почты пользователя.
2. name – имя пользователя.
3. pages – список всех страниц пользователя.
4. workspace – объект, содержащий данные текущего рабочего пространства.
5. settings – настройки пользователя: тема, шрифт, размер текста и аватар.

Асинхронные операции:

1. getUser – получает имя и email пользователя.
2. getPages – загружает список страниц пользователя.
3. getSettings – загружает настройки пользователя.
4. getWorkspace – получает данные рабочего пространства.
5. updateUserSettings – сохраняет обновленные пользовательские настройки.
6. createPage – добавляет новую страницу в список.
7. destroyPage – удаляет страницу.
8. renamePage – переименовывает страницу.
9. movePage – обновляет позицию страницы в структуре.
10. updatePageImgs – обновляет изображения страницы.
11. descriptionPage – обновляет описание страницы.
12. typePage – изменяет тип страницы.

3.7.4 Сервис SocketService. Сервис управляет созданием и отключением соединения, а также хранит текущее состояние и экземпляр сокета.

Для организации двустороннего взаимодействия между клиентом и сервером используется WebSocket через библиотеку socket.io-client [18].

Основные функции:

1. connectSocket(token: string, page_id: number) – устанавливает новое WebSocket-соединение. Перед подключением отключает предыдущее, если оно было активно. При инициализации указываются: токен авторизации в auth, page_id в параметрах запроса query. Также устанавливаются слушатели на события «connect» и «disconnect».
2. disconnectSocket() – завершает текущее соединение и очищает ссылку на сокет.
3. isSocketReady() – возвращает булево значение. Установлено ли в данный момент соединение.
4. getSocket() – возвращает текущий экземпляр сокета Socket, если он существует.

3.7.5 Хук `useSocketListeners`. Этот хук подключает клиент к потоку событий от WebSocket-сервера, связанных с инициализацией и изменением содержимого страницы. Он автоматически подписывается на нужные события и обновляет состояния при получении данных. При размонтировании компонента все обработчики событий отключаются для предотвращения утечек памяти.

События WebSocket:

1. `page-init` – начальная загрузка блоков страницы.
2. `page-init-list` – начальная загрузка структуры страницы, на основе которой будут отображены блоки.
3. `page-init-elements` – начальная загрузка элементов.
4. `page-init-list-elements` – начальная загрузка списка элементов, которые находятся на странице.
5. `change-list` – измененная структура страницы.
6. `change-create` – список новых блоков.
7. `change-update` – список обновленных блоков.
8. `change-create-elements` – список новых элементов.
9. `change-update-elements` – список изменений в отображаемых элементах.
10. `change-list-elements` – обновление структуры элементов.

4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

В данном разделе представлено детальное описание ключевых алгоритмов, используемых в серверной и клиентской частях приложения, с пояснением их логики и особенностью реализации.

4.1 Алгоритмы серверной части

4.1.1 Алгоритм обработки операций. Данный алгоритм реализует метод `processStorage`.

Метод `processStorage` является центральным элементом логики обработки изменений внутри страницы в рамках многопользовательской среды. Он реализует сложный, но согласованный алгоритм, обеспечивающий последовательную и атомарную обработку операций редактирования, таких как создание, обновление, удаление и изменение порядка блоков, с учетом текущего состояния страницы и потенциальных конфликтов при совместной работе пользователей.

Метод отвечает за применение всех накопленных операций в `TransactionsStorage` к текущему состоянию страницы, представленной в виде двусвязного списка `DoublyLinkedList` [19] блоков. Он обеспечивает:

1. Корректную обработку операций разных типов с приоритетом и фильтрацией.
2. Согласованность данных между отображаемым списком блоков и их хранилищем в базе данных.
3. Разрешение конфликтов, возникающих при работе нескольких пользователей.
4. Отслеживание изменений порядка блоков и управление указателями `pointer_to`, отражающими позиционирование элементов.

Иллюстрация работы алгоритма изображена на рисунке 4.1.

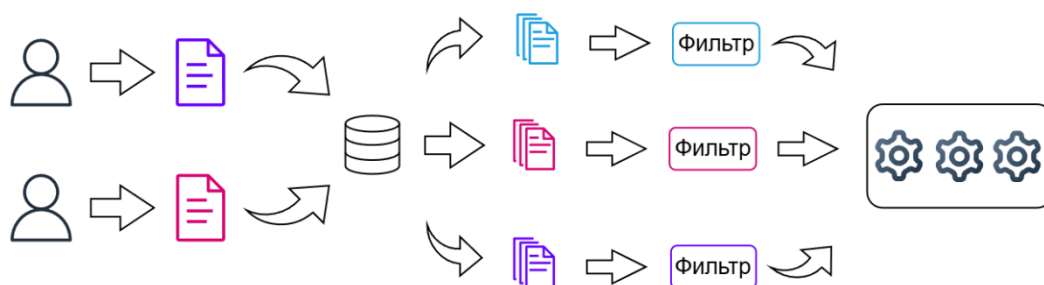


Рисунок 4.1 – Обработки операций

Шаги алгоритма:

Шаг 1. Проверка валидности входных данных. Проверяем, что переданы список блоков `list` и множество элементов `listElements`. Если что-то отсутствует – выбрасываем исключение `WsException`.

Шаг 2. Клонировать `listElements` в `updatedListElements`, чтобы изменения не затрагивали исходный объект.

Шаг 3. Клонировать `list` в `originalList` для сохранения исходного состояния списка блоков.

Шаг 4. Формируем множество ID всех блоков `originalListSet`, чтобы быстро проверять их наличие.

Шаг 5. Извлечение и группировка операций из транзакционного хранилища. Получаем из `storage` списки операций по типам: `create`, `update`, `delete`, `changePosition`, `createInDb`, `deleteInDb`.

Шаг 6. Фильтруем операции удаления, вызвав метод `filterDeleteOps`, чтобы удалить только те блоки, которые действительно есть в оригинальном списке.

Шаг 7. Фильтруем операции обновления, вызвав метод `filterUpdateOps`, чтобы не обновлять удаленные блоки и избежать конфликты между операциями, которые относятся к одному блоку.

Шаг 8. Фильтруем операции создания, вызвав метод `filterCreateOps`, чтобы не создать блоки, которые будут идти за несуществующими блоками.

Шаг 9. Фильтруем операции изменения позиции, вызвав метод `filterChangePosOps`, чтобы убрать некорректные перемещения.

Шаг 10. Фильтруем операции создания столбцов и строк, вызвав метод `filterCreateInDbOps`, чтобы не создать элементы в несуществующем блоке.

Шаг 11. Фильтруем операции удаления элементов, вызвав метод `filterDeleteInDbOps`, чтобы удалить только существующие элементы.

Шаг 12. Создаются переменные для сохранения результатов алгоритма: массив `createdBlocksInfo` с информацией об идентификаторах созданных блоков и их связях `pointer_to`, массивы созданных и обновлённых блоков.

Шаг 13. Если есть операции обновления `filteredUpdateOps`, вызываем метод `updateBlocks`, который обновляет блоки в базе данных.

Шаг 14. Если есть операции удаления строк и столбцов, `delRowOps` и `delColumnOps` соответственно, вызываем метод `deleteInDbs`. Удаляем соответствующие ID из `updatedListElements`.

Шаг 15. Если есть операции создания строк и столбцов, `rowOps` и `columnOps` соответственно. Вызываем метод `createInDbs`. Обновляем `updatedListElements` новыми ID. Добавляем новые созданные элементы в общий список `createdElements`.

Шаг 16. Если есть операции создания новых блоков `filteredCreateOps`, вызываем метод `createBlocks`. Обновляем список блоков `list`: вставляем новые блоки либо в начало списка, либо после заданного блока `pointer_to`. Добавляем новые блоки в `createdBlocks` и их элементы в `createdElements`. Обновляем `updatedListElements` новыми ID блоков.

Шаг 17. Если есть операции изменения позиции `filteredChangePosOps`, то перемещаем блоки внутри `list` согласно новым `pointer_to`.

Шаг 18. Если в списке есть блоки, которые нужно удалить `deleteBlockIds`, то удаляем соответствующие узлы из `list`.

Шаг 19. Сравниваем старую версию списка `originalList` и новую `list`. Находим блоки, у которых изменился предыдущий элемент `pointer_to`.

Шаг 20. Формируем список изменений порядка `changes`.

Шаг 21. Если есть изменения порядка `changes`, то вызываем метод `changePointers`, чтобы обновить `pointer_to` в базе данных.

Шаг 22. Если были удалены блоки `deleteBlockIds`, то вызываем метод `deleteBlocks`, чтобы физически удалить их из базы данных. Убираем их ID из `updatedListElements`.

Шаг 23. Возвращаем итоговый объект: обновленный список `list`, созданные и обновленные блоки `createdBlocks` и `updatedBlocks`, созданные и обновленные элементы таблиц `createdElements` и `updatedElements`, итоговое множество элементов `updatedListElements`.

Пример фильтра `filterDeleteOps`:

```
filterDeleteOps (
  deleteOps: Operation<DeleteOperationArgs>[],
  originalListSet: Set<number>,
): Set<number> {
  if (deleteOps.length === 0) {
    return new Set<number>();
  }
  const deleteBlockIds = new Set(deleteOps.map((op) =>
op.args.block_id));
  const filteredDeleteBlockIds = new Set<number>();
  for (const blockId of deleteBlockIds) {
    if (originalListSet.has(blockId)) {
      filteredDeleteBlockIds.add(blockId);
    }
  }
  return filteredDeleteBlockIds;
}
```

Основными операциями являются – создание и удаление. На основе операции создания можно понять общий принцип их обработчиков. Ниже приведен фрагмент обработчика `createBlocks`:

```
await this.sequelize.transaction(async (transaction) => {
  const createdBlocks: Blocks[] = [];
  const newColumns: Blocks[] = [];
  for (const op of createOps) {
```

```

const createDto: CreateBlockDto = {
  type: op.args.type,
  pointer_to: op.args.pointer_to,
  is_element: false,
  sub_pointer_to: null,
  database_y: null,
  database_x: null,
  page_id: page_id,
  content:
    op.args.type === 'database'
      ? { layout_type: 'Default', args: { name:
'NewTable' } }
      : op.args.content || {},};
const newBlock = await this.createBlock(createDto,
transaction);
if (newBlock.type === 'database') {
  const newColumn = await this.createColumnOfDb(
    newBlock,
    'text-property',
  );
  newColumns.push(...newColumn);
}
createdBlocks.push(newBlock);
}
return { createdBlocks, newColumns };
});

```

Этот фрагмент кода демонстрирует общий подход к обработке операций – применение в рамках транзакции, итерация по входным данным, вызов соответствующего метода сервиса, а затем возможная дополнительная логика в зависимости от операции.

4.1.2 Алгоритм подключения клиента к странице. Данный алгоритм состоит из методов: `afterInit`, `verifyUserForWS`, `verifyJwtToken`, `checkRightConnectToPage`.

Диаграмма последовательности, иллюстрирующая сценарий работы со страницей, приведена на чертеже ГУИР.400201.063 РР.4.

Метод `afterInit` выполняет настройку логики подключения клиентов к WebSocket-серверу `server` после его инициализации.

По сути, он задает посредник `middleware` для всех входящих соединений, который проверяет пользователя перед допуском к серверу.

Описание `afterInit` по шагам:

Шаг 1. Установка промежуточного обработчика через `server.use`. Для каждого нового клиента сервер вызывает переданное `middleware`.

Шаг 2. Извлекаются: `token` из `client.handshake.auth`, параметры запроса `client.handshake.query`. Эти данные передаются в `eventsService.verifyUserForWS`, который проверяет пользователя.

Шаг 3. Если проверка успешна, то возвращаются данные пользователя

payload и ID комнаты, куда должен подключиться клиент roomId.

Шаг 4. Проверяется, нет ли уже активного подключения с таким user_id в clientsStorage. Если есть – выбрасывается ошибка WsException с соответствующим сообщением, чтобы не допустить множественные подключения одного пользователя.

Шаг 5. Клиент добавляется в clientsStorage, связав user_id с объектом сокета.

Шаг 6. В объект данных клиента client.data записывается user_id.

Шаг 7. Клиент присоединяется к своей комнате через client.join(roomId).

Шаг 8. После успешной проверки и регистрации вызывается next(), чтобы разрешить дальнейшую обработку соединения.

Шаг 9. Если в процессе возникла ошибка, то в next() передается WsException с причиной ошибки или исходная ошибка WsException.

Метод verifyUserForWS выполняет проверку пользователя, который пытается подключиться к WebSocket-серверу, на подлинность и право подключения к определенной странице.

Этот метод предназначен для защиты соединения на уровне авторизации и доступа.

Описание verifyUserForWS по шагам:

Шаг 1. Берется page_id – уникальный идентификатор страницы из параметров URL запроса query.

Шаг 2. Если token или page_id отсутствует или передан в неправильном формате, то выбрасывается ошибка WsException о несправедном формате параметров.

Шаг 3. page_id преобразуется в число. Если результат преобразования – NaN, это также считается ошибкой параметров, выбрасывается WsException.

Шаг 4. С помощью tokenService.verifyJwtToken() token проверяется на подлинность.

Шаг 5. В случае успешной проверки извлекаются данные о пользователе payload.

Шаг 6. Вызывается workspaceService.checkRightConnectToPage, чтобы удостовериться, что пользователь имеет право подключаться к странице roomId.

Шаг 7. Если прав нет, выбрасывается ошибка WsException с сообщением об этом.

Шаг 8. При успешной проверке метод возвращает объект из payload и roomId.

Метод verifyJwtToken занимается проверкой подлинности JWT-токена.

Его задача – расшифровать и проверить переданный token с помощью

заданного секретного ключа.

Описание `verifyJwtToken` по шагам:

Шаг 1. Метод принимает строку `token`.

Шаг 2. С помощью `jwtService.verify` происходит проверка токена: Проверяется его подпись на валидность с использованием секрета `secret_jwt`, который извлекается из конфигурации приложения `this.configService.get<string>('secret_jwt')`.

Шаг 3. Если токен валидный, то расшифрованные данные `payload` возвращаются в виде объекта типа `JwtPayloadDto`.

Шаг 4. Если токен невалидный, то `jwtService.verify` автоматически выбрасывает ошибку.

Метод `checkRightConnectToPage` проверяет, имеет ли пользователь право на доступ к определенной странице `page_id` в зависимости от указанного режима проверки `mode`.

Описание `checkRightConnectToPage` по шагам:

Шаг 1. Сначала метод обращается к базе данных через `workspaceMembersRepository`. Пытается найти запись о членстве пользователя `user_id` на странице `page_id`.

Шаг 2. Из записи берется только поле `role` роль пользователя.

Шаг 3. Если членство не найдено, то пользователь не связан со страницей – возвращается `false`.

Шаг 4. Если членство найдено, то приводит роль пользователя `membership.role` к нижнему регистру `toLowerCase()`, чтобы избежать ошибок из-за разных регистров.

Шаг 5. Выполняется проверка в зависимости от переданного режима.

Шаг 6. Если роль пользователя входит в допустимый список для данного режима – возвращается `true`, иначе – `false`.

4.2 Алгоритмы клиентской части

4.2.1 Алгоритм построения дерева страниц. Целью данного алгоритма является организация плоского массива страниц в структуру, представляющую иерархию «родитель – дочерние страницы». Это необходимо, например, для построения дерева страниц в пользовательском интерфейсе, где вложенные страницы отображаются в виде вложенных списков.

Входные данные – массив `pages`, содержащий объекты, представляющие страницы. Каждая страница имеет уникальный идентификатор `Page_id` и поле `parent_page_id`, указывающее на родительскую страницу. Если `parent_page_id` отсутствует `null`, страница считается корневой.

Объект `pagesByParent`, представляющий «карту» страниц по родителю, где ключом является `parent_page_id` или строка «root» для

корневых страниц. Значением – массив всех страниц, для которых данный ключ является родителем.

Фрагмент кода:

```
const pagesByParent: Record<number | "root", Page[]> =
pages.reduce(
  (acc, page) => {
    const parentKey = page.parent_page_id ?? "root";
    if (!acc[parentKey]) acc[parentKey] = [];
    acc[parentKey].push(page);
    return acc;
  },
  {} as Record<number | "root", Page[]>
);
```

4.2.2 Алгоритм рекурсивного рендеринга дерева страниц. Цель данного алгоритма – построение визуального дерева страниц с учетом их иерархии. Каждая страница может иметь дочерние элементы, которые рендерятся рекурсивно внутри текущего элемента. Это позволяет отображать вложенные структуры произвольной глубины.

Входные данные – `parentId`, идентификатор родительской страницы. Может быть числом или строкой «root» – для страниц верхнего уровня. Объект `pagesByParent`, сформированный ранее.

Выходные данные – JSX-дерево, представляющее список страниц и их дочерних элементов в виде вложенных списков.

Описание `renderPages` по шагам:

Шаг 1. Получение дочерних страниц. Извлекается массив страниц `currentPages` из `pagesByParent[parentId]`.

Шаг 2. Проверка на наличие потомков. Если `currentPages` не найден, функция возвращает `null`.

Шаг 3. Рендеринг списка. Возвращается JSX-элемент, содержащий список `PageItem` для каждой дочерней страницы. В `PageItem` передается: сама страница `page`, функция `renderChildren`, которая вызывает `renderPages(page.Page_id)` рекурсивно для вложенных страниц, обработчики создания страницы и управления меню.

4.2.3 Алгоритм обработки текстового ввода `handleBeforeInput`. Данный алгоритм реализует пользовательскую обработку событий ввода `InputEvent` в редактируемом текстовом блоке. Он позволяет вручную управлять вставкой и удалением текста с учетом текущего положения курсора и структуры данных в виде массива `TextSpan[]`, где каждый `TextSpan` содержит форматированный фрагмент текста.

Входные данные

1. `e: InputEvent` – событие ввода.

2. `spans: TextSpan[]` – массив отформатированных текстовых

фрагментов.

3. `cursorPos` – текущая глобальная позиция курсора, получаемая из `getCursorPosition()`.

4. `updateContent` – функция для обновления контента.

5. `cursorPosRef` – ссылка, используемая для восстановления позиции курсора после обновления.

Описание `handleBeforeInput` по шагам:

Шаг 1. Получение типа действия и данных. Извлекаются `inputType` и `data` из события `e`.

Шаг 2. Определяется текущая позиция курсора `cursorPos`; если позиция не определена, обработка прекращается.

Шаг 3. Создается новый массив `newSpans`, в который будут собираться обновленные фрагменты.

Шаг 4. Переменная `offset` отслеживает глобальную позицию начала текущего `span`.

Шаг 5. Обработка вставки или удаления текста. Проход по `spans` до тех пор, пока не найден `span`, в который попадает `cursorPos`.

Шаг 6. Вычисляется `localOffset` – локальная позиция курсора в пределах текущего `span`.

Шаг 7. Если тип события `insertText`. Вставляется символ `data` внутри текущего `span` в нужной позиции.

Шаг 8. Если тип события `deleteContentBackward`. Если курсор в начале `span`, удаляется последний символ из предыдущего `span`, если он существует. Иначе, удаляется символ в текущем `span` перед курсором.

Шаг 9. Обновленный `span` добавляется в `newSpans`.

Шаг 10. Остальные `spans` добавляются без изменений.

Шаг 11. Вызывается `updateContent` с результатом `mergeSimilarSpans(newSpans)` – объединение смежных одинаковых фрагментов.

Шаг 12. Обновляется позиция курсора `cursorPosRef` в соответствии с событием.

Позиция курсора определяется с помощью функции `getCursorPosition`. Код функции:

```
const getCursorPosition = () => {
  const sel = window.getSelection();
  if (!sel || !sel.anchorNode || sel.anchorOffset == null)
    return null;
  const offset = sel.anchorOffset;
  let node = sel.anchorNode;
  while (node && node.nodeType !== Node.ELEMENT_NODE &&
    node.parentNode) {
    node = node.parentNode;
  }
  const container = divRef.current;
```

```

if (!node || !container) return null;
let total = 0;
for (const child of container.childNodes) {
  if (child === node) break;
  total += child.textContent?.length || 0;
}
return total + offset;
};

```

Функция `mergeSimilarSpans` объединяет смежные одинаковые фрагменты. Это необходимо, чтобы сократить количество `span` и упростить их обработку, рендер, редактирование, отправку на сервер. Блок-схема функции изображена на рисунке 4.2.

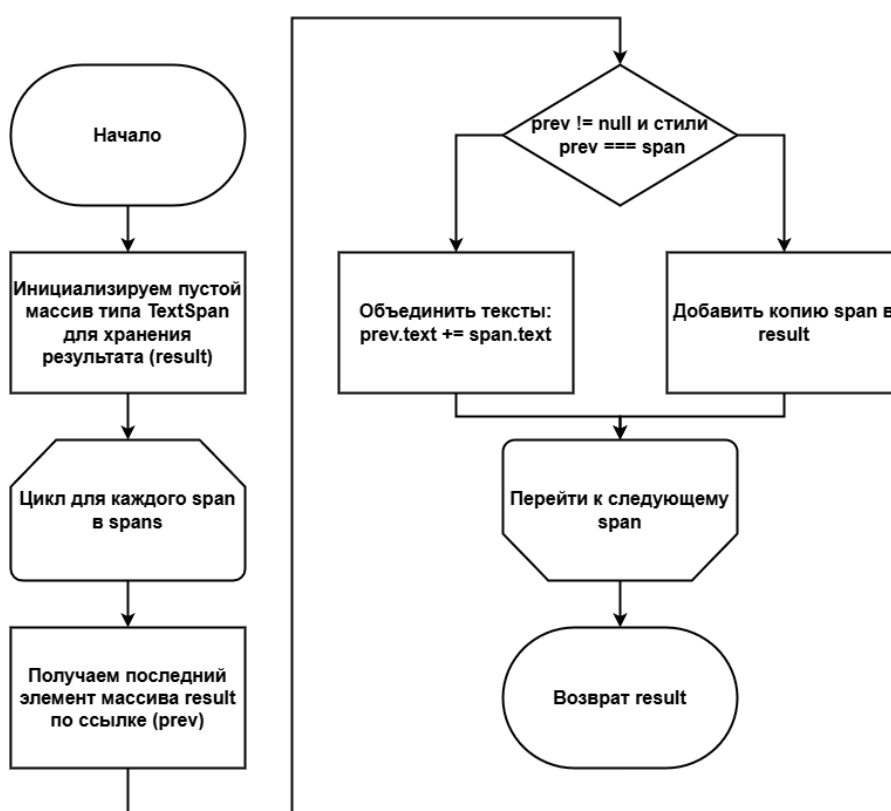


Рисунок 4.2 – Блок-схема функции `mergeSimilarSpans`

5 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

Раздел содержит описание порядка проведения испытаний разрабатываемой системы с целью проверки соответствия системы установленным требованиям.

В данном разделе приводятся цели испытаний, перечень проверяемых параметров, условия проведения, и средства измерений, а также последовательность и методика выполнения испытаний.

Указанные сведения обеспечивают объективную оценку работоспособности и качества системы, а также служат основанием для принятия решений о его дальнейшем применении или доработке.

5.1 Тестирование

Так как в качестве серверной платформы используется Node.js приложение, разработанное с использованием фреймворка NestJS версии 11.x, языком программирования TypeScript (версия 5.7.3). Проведение тестов осуществляется с помощью инструментов jest [20] и supertest [21].

Jest – это мощный фреймворк для тестирования, разработанный Facebook. Он предназначен для простого написания и запуска тестов, поддерживает TypeScript. В проектах на NestJS Jest используется как основной инструмент для тестирования: он позволяет проверять работу отдельных компонентов приложения или всего API в целом, обеспечивая высокую надежность логики на всех уровнях.

Supertest – это вспомогательная библиотека для работы с HTTP-запросами, особенно полезная в e2e-тестах. Она позволяет эмулировать обращения к серверу напрямую в коде, не запуская приложение на реальном порту. С ее помощью можно отправлять POST, GET и другие запросы к маршрутам приложения NestJS и проверять ответы.

В качестве методики испытаний будут проведены интеграционные тесты. Описание тестов будет представлено в виде таблиц.

5.1.1 Тестирование блока управления учетными записями представлено в таблице 5.1. В таблице описана проверка ключевых конечных точек.

Таблица 5.1 – Тестирование блока управления учетными записями

Эндпоинты теста	Тест	Ожидаемый результат	Статус
1	2	3	4
/api/users/init-signup	Отправка валидного email	Статус 200	Успех
/api/users/init-signup	Отправка обычную строку	Статус 400	Успех
/api/users/complete-signup	Отправка валидного email и token	Статус 200 и JWT-токен	Успех

Продолжение таблицы 5.1

1	2	3	4
/api/users/complete-signup	Отправка email существующего пользователя	Статус 409	Успех
/api/users/login	Ввод корректных данных авторизации	Статус 200 и JWT-токен	Успех
/api/users/login	Ввод некорректных данных авторизации	Статус 401	Успех
/api/users/pages	Получение страниц с валидным JWT	Статус 200 и список страниц	Успех
/api/users/pages	Получение страниц с подделанным JWT	Статус 401	Успех
/api/users/destroy	Удаление пользователя с валидным JWT	Статус 204	Успех
/api/users/request-password-reset	Запрос смены пароля с валидным email	Статус 200 и письмо на email	Успех

По таблице 5.1 можно сделать вывод, что основная логика блока и механизмы работают корректно.

5.1.2 Тестирование части блока управления рабочим пространством представлено в таблице 5.2. В таблице описана проверка ключевых конечных точек.

Таблица 5.2 – Тестирование блока управления рабочим пространством

Эндпоинты теста	Тест	Ожидаемый результат	Статус
1	2	3	4
/api/manager/create-page	Создание страницы с корректными данными	Статус 201. Возвращается объект страницы с id и данными	Успех
/api/manager/update-page	Обновление страницы с корректными данными	Статус 401	Успех
/api/manager/update-page	Попытка доступа к чужой странице	Статус 403	Успех
/api/manager/page-img	Обновление изображения страницы с правильным файлом	Статус 200. Изображение страницы обновлено	Успех

Продолжение таблицы 5.2

1	2	3	4
/api/manager/page-img	Отправка файла, который не является изображением	Статус 400, ошибка валидации	Успех
/api/users/login	Ввод некорректных данных авторизации	Статус 401	Успех
/api/manager/delete-page	Удаление страницы по корректному id страницы	Статус 200. Страница успешно удалена	Успех
/api/manager/restore-page	Восстановление страницы с корректными данными	Статус 200. Страница успешно восстановлена	Успех
/api/manager/confirm-invitation	Подтверждение приглашения с подделанным токеном приглашения	Статус 403	Успех
/api/manager/delete-member	Удаление члена страницы без соответствующих прав	Статус 403	Успех
/api/manager/change-role	Изменение роли участника с невалидными данными	Статус 400	Успех

Тестирование блока управления рабочим пространством подтвердило корректность обработки основных операций, включая создание, обновление, удаление и восстановление страниц, а также управление участниками и их ролями. Система корректно реагирует на попытки несанкционированного доступа, обеспечивая защиту данных, валидирует входные данные и успешно выполняет операции при корректных запросах.

Таким образом, тесты показали, что механизмы аутентификации, авторизации и валидации данных работают стабильно и обеспечивают безопасное управление рабочим пространством.

5.1.3 Тестирование второй части блока управления рабочим пространством. Так как вторая часть блока управления рабочим пространством использует Websocket, то может использоваться API-инструментарий для проверки поведения приложения. Например, Postman.

Postman – это инструмент для тестирования API, который позволяет разработчикам и тестировщикам легко отправлять запросы к серверу и получать ответы [22].

В первую очередь необходимо проверить корректность логики подключения к соединению. Не авторизованные пользователи не могут

подключиться к странице. Попытка подключения не авторизованного пользователя проиллюстрирована на рисунке 5.1.



Рисунок 5.1 – Подключение не авторизованного пользователя

Также пользователь должен иметь право подключиться к определенной странице. Попытка подключения к странице, к которой нет доступа, проиллюстрирована на рисунке 5.2.

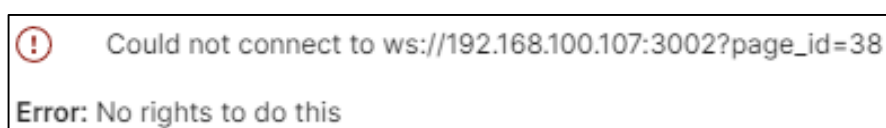


Рисунок 5.2 – Подключение без прав

Система не допускает одновременное подключение одного пользователя к одной странице через несколько WebSocket-соединений. Результат одновременного подключения изображен на рисунке 5.3.

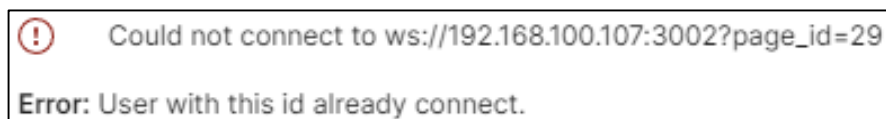


Рисунок 5.3 – Одновременное подключение

При удачном подключении к странице клиент должен получить актуальные данные о странице. Пример событий, которые клиент получает при подключении представлен на рисунке 5.4.

↓	page-init-list-ele...	{"structure_elements":[]}
↓	page-init-elemen...	{"elements":[]}
↓	page-init-list	{"list":[9]}
↓	page-init	{"bloks":[{"Block_id":9,"type":"text"}

Рисунок 5.4 – События при подключении

После тестирования логики подключения необходимо проверить корректность обработки операций. При тестировании важно обратить внимание на такие ситуации:

- обращение к несуществующим блокам в странице;
- корректное изменение структуры блоков в странице;
- конкурентные операции;
- взаимоисключающие операции.

В таблице 5.3 описана проверка ключевых сценариев работы с операциями.

Таблица 5.3 – Тестирование операций

Сценарий	Результат	Вывод
1	2	3
Создание блока, за блоком которого нет в странице	Операция отсеивается фильтром	Логика фильтров операции корректна
Попытка обновления блока, который уже был удален	Операция отфильтровывается	Исключаются конфликтующие операции
Перемещение блока, отсутствующего в списке	Операция пропускается	Защита от некорректных ссылок
Удаление блока, отсутствующего в списке	Операция отфильтровывается, блок не удаляется	Удаление корректно игнорирует несуществующие блоки.
Создание блоков без указателя на родителя	Блок вставляется в начало списка	Обеспечивается корректная вставка даже без указателя на родителя
Много операций конфликтующих за изменение блока	Побеждает операция, пришедшая позже остальных	Защита от коллизий
Много операций меняющих структуру страницы	Все операции обрабатываются в корректной последовательности для правильного отслеживания изменения указателей	Гарантируется целостность порядка блоков
Повторное удаление одного и того же блока	Повторное удаление невозможно. Операции фильтруются	Защита от дублирующих операций удаления.
Операции с таблицами при удаленных родителях	Соответствующие операции фильтруются	Защита от некорректных операций с блоком «базой данных»

Продолжение таблицы 5.3

1	2	3
Конфликт между удалением и обновлением или созданием элемента по одному блоку	Операции фильтруются, удаление имеет приоритет.	Обеспечивается приоритет удаления в случае конфликта.

Тестирование операций показало устойчивость системы к различным конфликтным и некорректным сценариям, таким как обновление или удаление уже несуществующих блоков, создание без родительского указателя, множественные одновременные изменения и структурные конфликты. Фильтрация операций и приоритеты позволяют эффективно устранять ошибки и сохранять целостность структуры страницы. В результате обеспечивается надежная обработка операций и защита от коллизий, что критически важно при работе с многопользовательским взаимодействием и асинхронной синхронизацией данных.

5.1.3 Тестирование блока управления файловым хранилищем представлено в таблице 5.4.

Таблица 5.4 – Тестирование блока управления файловым хранилищем

Эндпоинты теста	Тест	Ожидаемый результат	Статус
/api/explorer/add-file	Загрузка файла в блок	Файл успешно загружается, возвращается информация о файле	Успех
/api/explorer/get-file	Получение файла по file_id и block_id	Загруженный файл возвращается в виде потока с правильными заголовками	Успех
/api/explorer/delete-file	Удаление файла по file_id и block_id	Файл удаляется, возвращается подтверждение	Успех
/api/explorer/export-page	Экспорт страницы в ZIP по page_id	ZIP-файл со страницей формируется и возвращается в виде потока	Успех

Для того чтобы проверить в корректность работы с файлами можно использовать специальную административную панель MinIO, изображенную на рисунке 5.5.

Результаты тестирования блока управления файловым хранилищем

подтверждают его корректную работу по всем ключевым сценариям: загрузка, получение, удаление файлов и экспорт страниц. Все тесты показали ожидаемое поведение — операции выполняются успешно, файлы обрабатываются корректно, возвращаются в нужном формате, и система выдает соответствующие ответы.

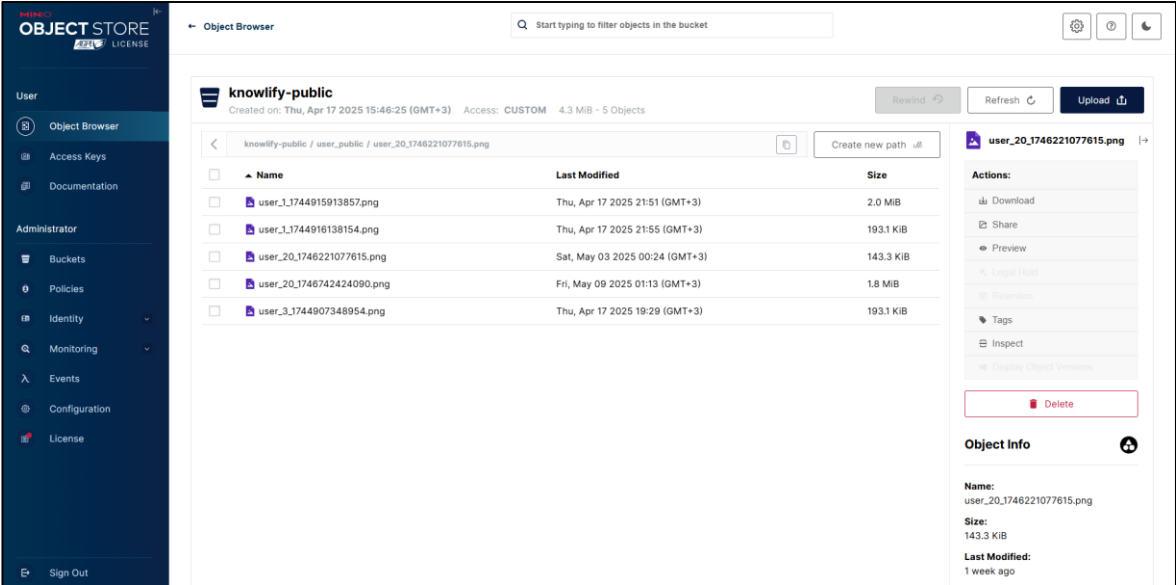


Рисунок 5.5 – Административная панель MinIO

6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Данный раздел содержит инструкции по использованию приложения, включая описание основных функций интерфейса и пошаговое руководство по выполнению ключевых операций.

Поскольку интерфейс реализован в виде веб-приложения, доступ к нему возможен через любой современный браузер.

6.1 Первый запуск приложения

При первом запуске пользователю необходимо выполнить вход в систему. На экране отображается форма авторизации, включающая поля для ввода адреса электронной почты и пароля. Также доступны ссылки для восстановления пароля «Forgot password?» и регистрации новой учетной записи «Create». Форма «Sign in» представлена на рисунке 6.1.

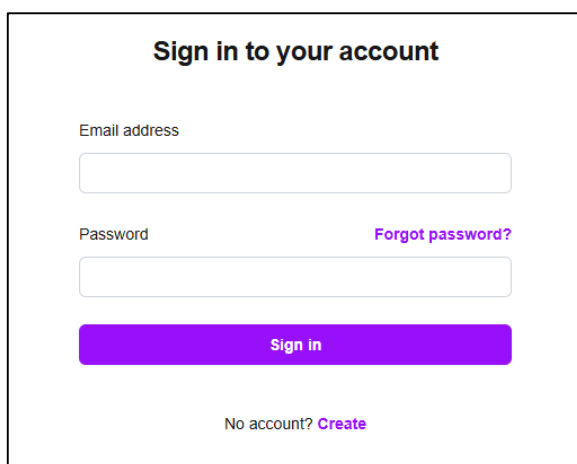
The image shows a login form titled "Sign in to your account". It contains two input fields: "Email address" and "Password". To the right of the password field is a link "Forgot password?". Below the fields is a blue "Sign in" button. At the bottom, there is a link "No account? Create".

Рисунок 6.1 – Форма «Sign in»

После ввода данных в поля «Email address» и «Password» необходимо нажать кнопку «Sign in» для входа в аккаунт, если он уже зарегистрирован.

Если у вас еще нет аккаунта, нажмите на ссылку «Create» на экране входа. Откроется форма начала регистрации (см. рисунок 6.2).

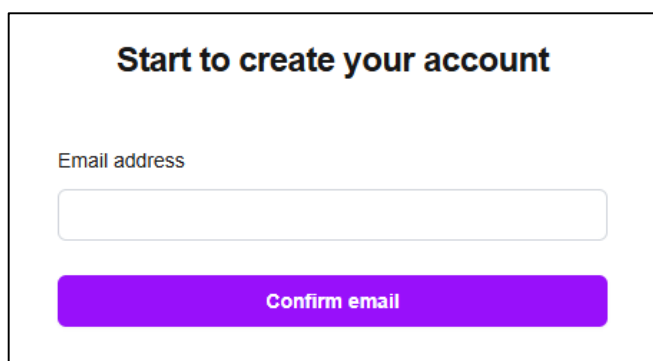
The image shows a registration form titled "Start to create your account". It contains one input field: "Email address". Below the field is a blue "Confirm email" button.

Рисунок 6.2 – Форма начала регистрации

Далее необходимо ввести адрес электронной почты и нажать кнопку «Confirm email» для подтверждения. На указанный email будет отправлено письмо с дальнейшими инструкциями для завершения создания учетной записи. Пример письма изображен на рисунке 6.3.

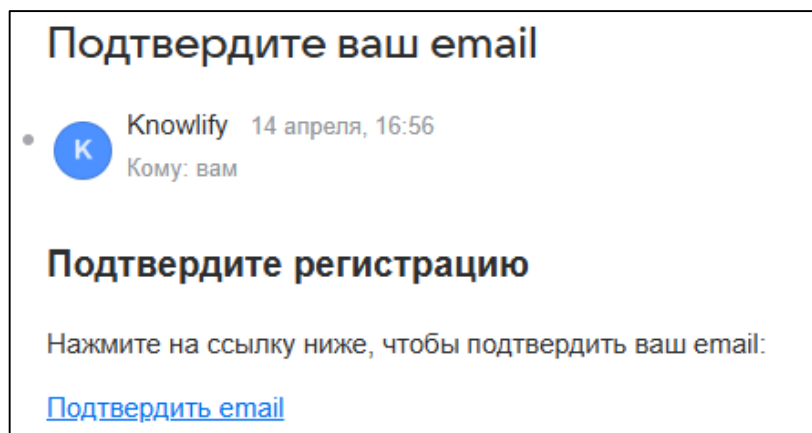


Рисунок 6.3 – Пример письма на регистрацию

После нажатия на «Подтвердить email» откроется форма завершения регистрации (см. рисунок 6.4). Здесь отображается введенный ранее адрес электронной почты, и необходимо придумать и ввести пароль. Для завершения регистрации нажмите кнопку «Sign Up». После этого аккаунт будет создан, и вы сможете войти в систему.

The image shows a registration completion form. At the top, it says "Sign up ExampleEmail@mail.ru". Below this, there is a label "Password" next to a text input field. At the bottom of the form, there is a large blue button with the text "Sign Up" in white.

Рисунок 6.4 – Форма завершения регистрации

6.2 Основное рабочее пространство

После удачной регистрации или входа в аккаунт отобразится страница основного рабочего пространства (см рисунок 6.5). Здесь пользователь получает доступ к основным функциям приложения: созданию и редактированию страниц, управлению блоками, навигации по структуре проекта и взаимодействию с другими участниками. Интерфейс интуитивно понятен и предназначен для удобной и продуктивной работы.

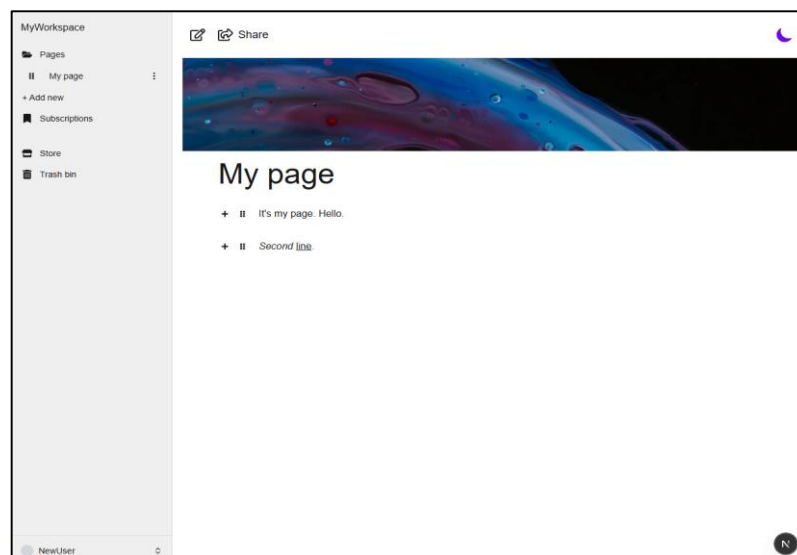


Рисунок 6.5 – Страница основного рабочего пространства

6.2.1 Блок управления страницами. Блок находится в боковой панели интерфейса и предназначен для управления структурой страниц рабочего пространства. Блок изображен на рисунке 6.6.

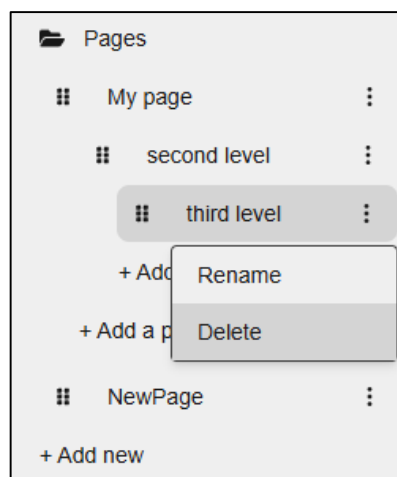


Рисунок 6.6 – Блок управления страницами

Здесь расположен список личных и совместных страниц с иерархической структурой. Иерархия страниц визуализирована через отступы, что упрощает навигацию по сложной структуре. Чтобы увидеть вложенные страницы, необходимо нажать дважды на интересующую страницу, а для обычного открытия надо кликнуть один раз.

Присутствуют следующие кнопки управления:

1. «Rename» – переименование выбранной страницы.
2. «Add a page inside» – добавление страницы вложенного уровня.
3. «Delete» – удаление выбранной страницы.
4. «Add new» – создание новой страницы для корневого раздела.
5. «Grip Icon» – кнопка для удобного перетаскивания страницы.

6.2.2 Блок управления пользовательским данными. Блок находится внизу боковой панели и отвечает за управление учетной записью пользователя. Блок изображен на рисунке 6.7.

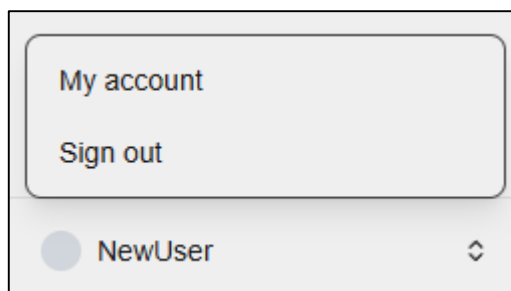


Рисунок 6.7 – Блок управления пользовательским данными

Данный блок включает кнопку для доступа к настройкам профиля и кнопку для безопасного выхода из системы. Также блок иллюстрирует изображение пользователя и его имя в системе.

При нажатии на кнопку «My account» откроется форма с настройками пользователя (см. рисунок 6.8).

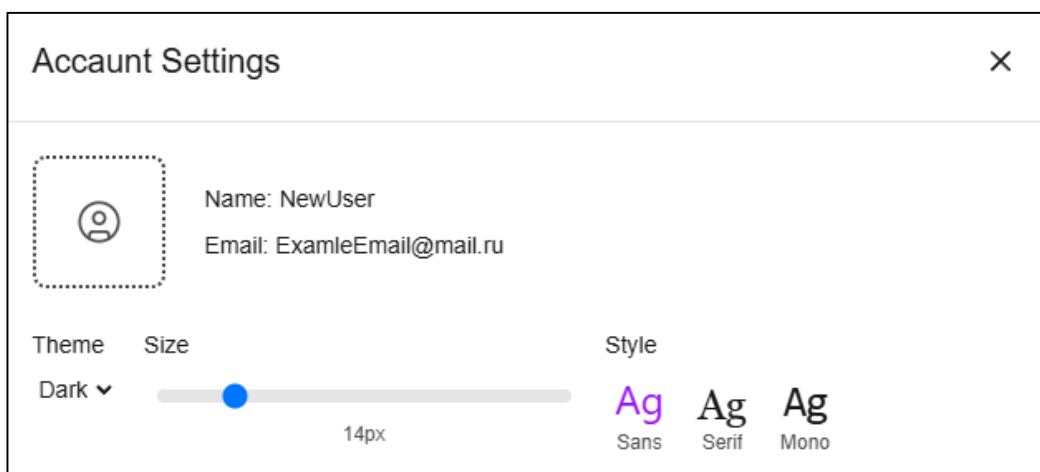


Рисунок 6.8 – Форма с настройками пользователя.

Форма включает такие персональные данные, как имя и электронная почта. Меню позволяет настроить цветовую схему приложения, нажав на ячейку «Theme». Также есть возможность настроить шрифт. Варианты шрифтов: «Sans», «Serif», «Mono». Шрифты могут быть настроены переключателем «Style». Размер шрифта регулируется специальным слайдером «Size».

Также квадратная иконка пользователя используется для загрузки собственного изображения.

6.2.3 Блок управления страницей. Блок находится в верхней панели и отвечает за управление настройками страницы (см рисунок 6.9).



Рисунок 6.9 – Блок управления страницей

Блок содержит две кнопки необходимых для открытия форм с соответствующими им функциями.

Если нажать на первую кнопку, то откроется форма настройки страницы, изображенная на рисунке 6.10.

Рисунок 6.10 – Форма настройки страницы

Форма позволяет пользователю загрузить собственное изображение для фона страницы. Для этого необходимо открыть системный диалог выбора файла с помощью кнопки «Choose a file...». Также можно добавить описание к странице. Для этого необходимо заполнить поле для ввода «Description» и нажать клавишу «Submit Description». Переключатель «Make Public» определяет, будет ли страница доступна другим пользователям.

Также пользователь может получить данные страницы в ZIP-файле, нажав на кнопку «Export».

Если нажать на вторую кнопку с подписью «Share», то откроется форма настройки совместной работы страницы, изображенная на рисунке 6.11.

Рисунок 6.11 – Форма настройки совместной работы страницы

Форма содержит поля для приглашения других пользователей для совместной работы. Для этого необходимо в поле под надписью «Invite user» вписать почту пользователя, которого хотите пригласить, и выбрать его роль, далее нажать на кнопку «Send Invite». Приглашенному пользователю отправится письмо с ссылкой на подтверждение.

После того, как пользователь, которого пригласили, перейдет по ссылке, он увидит окно с результатом обработки приглашения (см. рисунок 6.12).

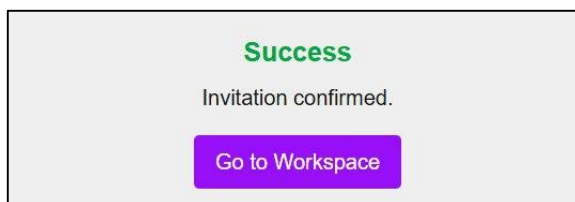


Рисунок 6.12 – Окно с результатом обработки приглашения

Когда кто-либо присоединится по приглашению, он отобразится в «Members». Пример от лица владельца страницы изображен на рисунке 6.13.



Рисунок 6.13 – Пример отображения «Members»

Владелец страницы имеет право менять роль участников и удалять их. Эти формы редактирования недоступны участникам.

6.2.4 Основное рабочее пространство. Это центральная область интерфейса, где пользователь взаимодействует с содержимым страницы. Пространство предназначено для просмотра, создания и редактирования контента. Пример рабочего пространства изображен на рисунке 6.14.

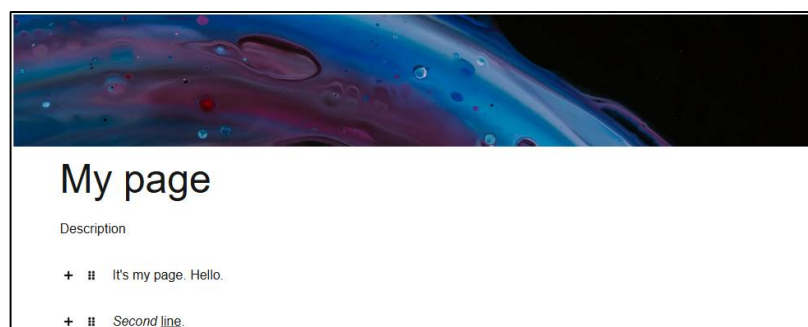


Рисунок 6.14 – Пример рабочего пространства

Рабочее пространство включает фоновое изображение, заголовок, описание, блоки и элементы.

Каждый блок содержит по две кнопки управления:

- кнопку создания нового блока;
- кнопку для открытия меню блока и перетаскивания.

Если нажать на кнопку создания, то откроется меню для выбора типа блока. Меню изображено на рисунке 6.15.

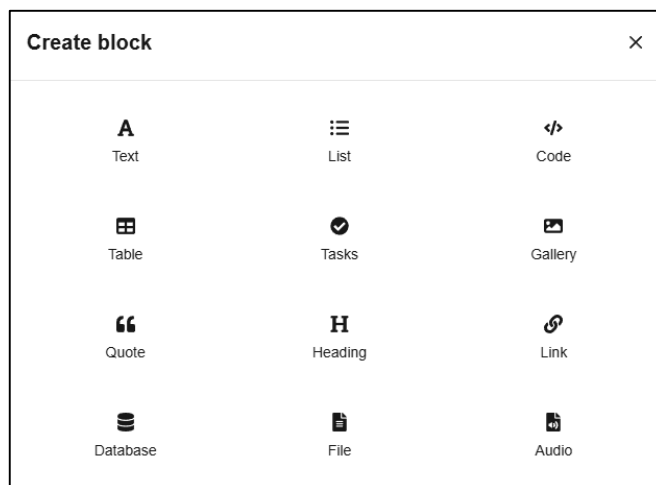


Рисунок 6.15 – Меню создания нового блока

После выбора типа нового блока он создастся за блоком с кнопкой, на которую было совершено нажатие.

Если кликнуть на вторую кнопку правым щелчком, то откроется специальное меню блока (см. рисунок 6.16), в котором можно оставлять комментарии к блоку и удалить его.

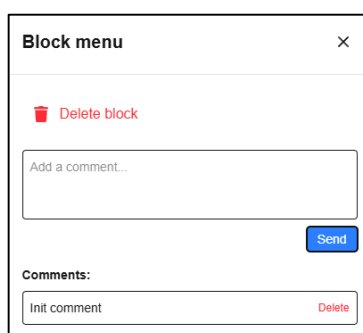


Рисунок 6.16 – Меню блока

Таким образом, интерфейс основного рабочего пространства обеспечивает интуитивное и гибкое взаимодействие пользователя с содержимым страницы. Возможности создания, редактирования и управления блоками реализованы в удобной визуальной форме, что способствует эффективной организации информации и комфортной работе с ней.

7 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ И РЕАЛИЗАЦИИ НА РЫНКЕ ПРОГРАММНОЙ ПЛАТФОРМЫ ЕДИНОГО РАБОЧЕГО ПРОСТРАНСТВА

7.1 Характеристика программного средства, разрабатываемого для реализации на рынке

Программное средство в данном дипломном проекте разрабатывается с целью повышения продуктивности и оптимизации рабочих процессов. Платформа создается для замены множества разрозненных инструментов, чтобы пользователи могли проектировать собственные системы управления данными.

Приложение реализует принцип централизации информации, объединяя инструменты, документы и базы данных в едином пространстве. Это устраняет необходимость переключаться между разрозненными сервисами. Платформа предоставляет возможность общего доступа в реальном времени, что упрощает командную работу. При этом пользователи могут персонализировать рабочие пространства, комбинируя инструменты и создавая интерфейсы, адаптированные под конкретные задачи, будь то управление проектом, ведение личного дневника или организация корпоративной базы знаний. Такой подход превращает платформу в универсальный инструмент, где структура, автоматизация и сотрудничество дополняются свободой настройки под индивидуальные потребности. Доступ через веб-интерфейс сделает приложение более гибким и экономически выгодным.

Целевая аудитория платформы – профессиональные коллективы и организации, нуждающиеся в эффективных инструментах для централизованного управления корпоративными знаниями, документацией и проектами.

Предполагаемая модель монетизации программного продукта: бесплатная базовая и платная расширенная версии. Пользователю будет предоставляться базовый функционал бесплатно, а за дополнительную плату открывается расширенный функционал. Например, увеличение объема хранилища и снятие ограничений.

Для привлечения новых пользователей будут использоваться контекстная реклама, SEO-оптимизация и распространение в социальных сетях. Бесплатная базовая версия станет ключевым инструментом для вовлечения пользователей.

Главными конкурентами разрабатываемой платформы являются такие приложения, как Notion и Obsidian. Стратегия разработки включает выявление преимуществ и недостатков аналогов и формирование уникального рыночного предложения. Кроме того, политики конфиденциальности и подходы к управлению данными, принятые конкурентами, могут создавать риски для рабочих процессов. Поэтому целесообразно создать платформу, ориентированную на безопасность данных пользователей.

7.2 Расчет инвестиций в разработку программного средства для его реализации на рынке

Инвестициями для организации-разработчика программного средства являются затраты на его разработку. Общая сумма затрат на разработку и реализацию рассчитывается по следующим параметрам:

- основная заработная плата разработчиков;
- дополнительная заработная плата разработчиков;
- отчисления на социальные нужды;
- прочие расходы;
- расходы на реализацию.

Расчет затрат на основную заработную плату команды разработчиков осуществляется исходя из состава и численности команды, размера месячной заработной платы каждого участника команды, а также трудоемкости работ, выполняемых при разработке программного средства отдельными исполнителями, по формуле

$$З_0 = K_{\text{пр}} \cdot \sum_{i=1}^n З_{\text{чи}} \cdot t_i, \quad (7.1)$$

где $K_{\text{пр}}$ – коэффициент премий и иных стимулирующих выплат;

n – категории исполнителей, занятых разработкой;

$З_{\text{чи}}$ – часовой оклад плата исполнителя i -й категории, р.;

t_i – трудоемкость работ, выполняемых исполнителем i -й категории, ч.

Рекомендуется принять коэффициент премий и иных стимулирующих выплат равным единице, так как в статистике среднемесячной заработной платы для сотрудников различных категорий ИТ-отрасли, как правило, уже учитываются выплаты подобного рода.

Часовой оклад каждого исполнителя определяется путем деления его месячного оклада на количество рабочих часов в месяце. Расчетная норма рабочего времени на 2025 год для пяти дневной недели составляет 167 часов по данным Министерства труда и социальной защиты населения.

Размер месячного оклада можно определить с помощью сервиса по поиску работы и персонала rabota.by [23].

Расчет затрат на основную заработную плату команды разработчиков представлен в таблице 7.1.

Таблица 7.1 – Расчет затрат на основную заработную плату команды разработчиков

Категория исполнителя	Месячный оклад, р.	Часовой оклад, р.	Трудоемкость работ, ч	Итого, р.
1	2	3	4	5
Программист	2822	16,9	240	4056

Продолжение таблицы 7.1

1	2	3	4	5
Тестировщик	1902	11,4	40	456
Дизайнер	1807	10,8	24	259,2
Итого				4771,2
Премия и иные стимулирующие выплаты				0
Всего затрат на основную заработную плату разработчиков				4771,2

Дополнительная заработная плата определяется по формуле

$$З_д = \frac{З_о \cdot Н_д}{100}, \quad (7.2)$$

где $З_о$ – затрат на основную заработную плату, р.;

$Н_д$ – норматив дополнительной зарплаты, 15 %.

Отчисления на социальные нужды определяется в соответствии с действующим законодательством по формуле

$$Р_{соц} = \frac{(З_о + З_д) \cdot Н_{соц}}{100} \quad (7.3)$$

где $Н_{соц}$ – норматив отчислений в ФСЗН и Белгосстрах (в соответствии с действующим законодательством по состоянию на апрель 2025 г. – 34,6 %).

Прочие расходы определяется по формуле

$$Р_{пр} = \frac{З_о \cdot Н_{пр}}{100}, \quad (7.4)$$

где $Н_{пр}$ – норматив прочих расходов, 30 %.

Расходы на реализацию определяется по формуле

$$Р_p = \frac{З_о \cdot Н_p}{100}, \quad (7.5)$$

где $Н_p$ – норматив расходов на реализацию, 4 %.

Расчет общей суммы затрат на разработку и реализацию программного средства представлен в таблице 7.2.

Таблица 7.2 — Расчет затрат на разработку программного средства

Наименование статьи затрат	Формула/таблица для расчета	Сумма, р.
1	2	3
Основная заработная плата разработчиков	Табл. 7.1	4771,2

Продолжение таблицы 7.2

1	2	3
Дополнительная заработная плата разработчиков	Формула (7.2)	715,68
Отчисления на социальные нужды	Формула (7.3)	1898,46
Прочие расходы	Формула (7.4)	1431,36
Расходы на реализацию	Формула (7.5)	190,85
Общая сумма затрат на разработку и реализацию		9007,55

7.3 Расчет экономического эффекта от реализации программного средства на рынке

Экономический эффект организации-разработчика программного средства представляет собой прирост чистой прибыли от его продажи на рынке потребителям, величина которого зависит от объема продаж, цены реализации и затрат на разработку программного средства.

Для определения экономического эффекта необходимо определить объем продаж. Если предположить, что в месяц приложением интересуются 15000 человек. Из 15000 человек зарегистрируются 1500 пользователей. А из оставшихся 15 процентов купят расширенные возможности платформы, то это 2700 продаж в год.

На основе Notion можно определить цену расширенной версии. Цена конкурента составляет от 10 до 20 долларов США [24], поэтому цена единицы программного средства будет составлять 30 белорусских рублей.

Прирост чистой прибыли, полученной разработчиком от реализации программного средства на рынке, можно рассчитать по формуле

$$\Delta\Pi_{\text{ч}}^{\text{р}} = (\Pi_{\text{отп}} \cdot N - \text{НДС}) \cdot P_{\text{пр}} \cdot \left(1 - \frac{H_{\text{п}}}{100}\right), \quad (7.6)$$

где $\Pi_{\text{отп}}$ – отпускная цена копии программного средства, р.;

N – количество копий программного средства, реализуемое за год, шт.;

НДС – сумма налога на добавленную стоимость, р.;

$P_{\text{пр}}$ – рентабельность продаж копий, %;

$H_{\text{п}}$ – ставка налога на прибыль, %.

Ставка налога на прибыль, согласно действующему законодательству, по состоянию на 2025 равна 20 %. Рентабельность продаж копий взята в размере 30 %.

Налог на добавленную стоимость определяется по формуле

$$\text{НДС} = \frac{\Pi_{\text{отп}} \cdot N \cdot H_{\text{д.с.}}}{100\% + H_{\text{д.с.}}}, \quad (7.7)$$

где $H_{д.с.}$ – ставка налога на добавленную стоимость в соответствии с действующим законодательством, % (по состоянию на июль 2025 г. – 20 %).

Используя полученные значения, посчитаем НДС по формуле 7.7:

$$НДС = \frac{30 \cdot 2700 \cdot 20\%}{100\% + 20\%} = 13500 \text{ р.}$$

Посчитав налог на добавленную стоимость, можно рассчитать прирост чистой прибыли по формуле 7.6:

$$\Delta\Pi_{\text{ч}}^p = (30 \cdot 2700 - 13500) \cdot 30\% \cdot \left(1 - \frac{20}{100}\right) = 16200 \text{ р.}$$

7.4 Расчет показателей экономической эффективности разработки и реализации программного средства на рынке

Оценка экономической эффективности разработки и реализации программного средства на рынке зависит от результата сравнения инвестиций (затрат) в его разработку (модернизацию, совершенствование) и полученного годового прироста чистой прибыли.

Если сумма инвестиций (затрат) на разработку меньше суммы годового экономического эффекта, то есть инвестиции окупятся менее чем за год, оценка экономической эффективности инвестиций в разработку программного средства осуществляется с помощью расчета рентабельности инвестиций по формуле (7.8).

$$ROI = \frac{\Delta\Pi_{\text{ч}}^p - Z_p}{Z_p} \cdot 100\% \quad (7.8)$$

где $\Delta\Pi_{\text{ч}}^p$ – прирост чистой прибыли, полученной от реализации программного средства на рынке информационных технологий, р.;

$$ROI = \frac{16200 - 9007,55}{9007,55} \cdot 100\% = 79,85\%$$

7.5 Вывод об экономической целесообразности разработки программного средства

По результатам технико-экономического анализа можно сделать вывод об экономической целесообразности разработки программного продукта. Общие затраты на разработку и внедрение платформы составили 9007,55 белорусских рублей. Прогнозируемый прирост чистой прибыли за первый год эксплуатации оценивается в 16200 белорусских рублей, а рентабельность инвестиций (ROI) составит 79,85 %. Такие показатели свидетельствуют о

высокой эффективности проекта и подтверждают его экономическую обоснованность.

Срок окупаемости проекта составляет менее семи месяцев, что минимизирует финансовые риски и позволяет быстро вернуть вложенные средства. Это особенно важно в условиях динамичного рынка, где скорость реализации идеи напрямую влияет на успех. Даже при снижении прогнозируемой прибыли на 20% ROI останется на уровне 44%, а срок окупаемости не превысит 11 месяцев. Это подтверждает устойчивость проекта к рыночным колебаниям.

Также необходимо понимать, что продажа программного продукта широкой аудитории неминуемо содержит возможность коммерческой неудачи. Основные риски проекта включают высокую конкуренцию на рынке, непредсказуемое поведение рынка. А также возможный провал плановых показателей, включая отставание по срокам или невыполнение финансовых прогнозов.

ЗАКЛЮЧЕНИЕ

В ходе дипломного проектирования было разработано программное средство, предоставляющее пользователям возможность создавать собственные рабочие пространства, адаптированные под индивидуальные потребности. Реализована поддержка различных типов контента, а также функциональность совместной работы в реальном времени.

Приложение обладает модульной и масштабируемой архитектурой, что обеспечивает гибкость при расширении функциональности и адаптации под различные сценарии использования. В процессе реализации были использованы современные технологические стеки, что способствовало достижению высокой производительности, читаемости кода и удобства сопровождения проекта.

Разработка успешно прошла тестирование и соответствует всем заявленным требованиям. Результаты экономического анализа подтверждают практическую значимость и целесообразность внедрения системы.

Проект может служить основой для дальнейших разработок в области цифровых платформ, управления знаниями и организации совместной работы. В частности, архитектура и реализованные технологии позволяют интегрировать дополнительные модули. Также возможна адаптация платформы под различные профессиональные сферы: образование, корпоративное управление, проектную деятельность, исследовательскую и научную работу. Гибкость решения и заложенные принципы масштабируемости открывают возможности для развития коммерческих и открытых версий приложения, а также для внедрения в организациях различного масштаба.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] Obsidian [Электронный ресурс]. – Режим доступа: <https://obsidian.md/> – Дата доступа: 17.02.2025.
- [2] Notion [Электронный ресурс]. – Режим доступа: <https://www.notion.com/> – Дата доступа: 17.02.2025.
- [3] Miro [Электронный ресурс]. – Режим доступа: <https://miro.com/>. – Дата доступа: 17.02.2025.
- [4] OWASP [Электронный ресурс]. – Режим доступа: <https://owasp.org/www-project-top-ten/> – Дата доступа: 22.02.2025.
- [5] PostgreSQL [Электронный ресурс]. – Режим доступа: <https://www.postgresql.org/> – Дата доступа: 23.03.2025.
- [6] MinIO [Электронный ресурс]. – Режим доступа: <https://min.io/> – Дата доступа: 23.03.2025.
- [7] Next.js [Электронный ресурс]. – Режим доступа: <https://nextjs.org/> – Дата доступа: 28.02.2025.
- [8] Nest.js [Электронный ресурс]. – Режим доступа: <https://nestjs.com/> – Дата доступа: 28.02.2025.
- [9] Sequelize [Электронный ресурс]. – Режим доступа: <https://sequelize.org/> – Дата доступа: 15.04.2025.
- [10] Multer [Электронный ресурс]. – Режим доступа: <https://www.npmjs.com/package/multer> – Дата доступа: 15.04.2025.
- [11] Socket.io [Электронный ресурс]. – Режим доступа: <https://socket.io/> – Дата доступа: 16.04.2025.
- [12] Nodemailer [Электронный ресурс]. – Режим доступа: <https://nodemailer.com> – Дата доступа: 16.04.2025.
- [13] Gmail Service [Электронный ресурс]. – Режим доступа: <https://developers.google.com/apps-script/reference/gmail?hl=ru> – Дата доступа: 16.04.2025.
- [14] AWS SDK [Электронный ресурс]. – Режим доступа: <https://aws.amazon.com/ru/sdk-for-javascript/> – Дата доступа: 20.04.2025.
- [15] Amazon S3 [Электронный ресурс]. – Режим доступа: <https://aws.amazon.com/fr/s3/> – Дата доступа: 20.04.2025.
- [16] Redux [Электронный ресурс]. – Режим доступа: <https://redux.js.org/> – Дата доступа: 25.04.2025.
- [17] Redux Toolkit [Электронный ресурс]. – Режим доступа: <https://redux-toolkit.js.org/> – Дата доступа: 25.04.2025.
- [18] Socket.io-client [Электронный ресурс]. – Режим доступа: <https://www.npmjs.com/package/socket.io-client> – Дата доступа: 28.04.2025.
- [19] DoublyLinkedList [Электронный ресурс]. – Режим доступа: <https://www.npmjs.com/package/doubly-linked-list-typed> – Дата доступа: 28.04.2025.
- [20] Jest [Электронный ресурс]. – Режим доступа: <https://jestjs.io/> – Дата доступа: 28.04.2025.
- [21] Supertest [Электронный ресурс]. – Режим доступа:

<https://www.npmjs.com/package/supertest> – Дата доступа: 28.04.2025.

[22] Postman [Электронный ресурс]. – Режим доступа: <https://www.postman.com/> – Дата доступа: 1.05.2025.

[23] Rabota.by [Электронный ресурс]. – Режим доступа: <https://rabota.by/> – Дата доступа: 11.03.2025.

[24] Notion Pricing Plans [Электронный ресурс]. – Режим доступа: <https://www.notion.com/pricing> – Дата доступа: 11.03.2025.

Стандарт предприятия [Электронный ресурс] : Минск БГУИР 2024. – Электронные данные. – Режим доступа: https://www.bsuir.by/m/12_100229_1_185586.pdf – Дата доступа: 28.03.2025.

Вычислительные машины, системы и сети: дипломное проектирование (методическое пособие) [Электронный ресурс] : Минск БГУИР 2019. – Электронные данные. – Режим доступа: https://drive.google.com/file/d/1hBy9kO-EKbHbdSy87tE0FcTUp_EYYsrr/view – Дата доступа: 28.03.2025.

ПРИЛОЖЕНИЕ А

(обязательное)

Код программы

main.ts

```
0001. import { NestFactory } from '@nestjs/core';
0002. import { AppModule } from '../modules/app/app.module';
0003. import { ConfigService } from '@nestjs/config';
0004. import { ValidationPipe } from '@nestjs/common';
0005.
0006. async function bootstrap() {
0007.   const app = await NestFactory.create(AppModule);
0008.   app.enableCors({
0009.     origin: 'http://192.168.100.107:3000',
0010.     credentials: true,
0011.   });
0012.   const configService = app.get(ConfigService);
0013.   const port = configService.get<number>('port')!;
0014.   app.useGlobalPipes(
0015.     new ValidationPipe({
0016.       transform: true,
0017.       transformOptions: {
0018.         enableImplicitConversion: true,
0019.       },
0020.     }),
0021.   );
0022.   await app.listen(port, '0.0.0.0');
0023. }
0024. bootstrap();
```

account.controller.ts

```
0025. import {
0026.   Body,
0027.   Controller,
0028.   Delete,
0029.   FileTypeValidator,
0030.   Get,
0031.   HttpStatusCode,
0032.   MaxFileSizeValidator,
0033.   ParseFilePipe,
0034.   ParseIntPipe,
0035.   Patch,
0036.   Post,
0037.   Query,
0038.   UploadedFile,
0039.   UseGuards,
0040.   UseInterceptors,
0041. } from '@nestjs/common';
0042. import {
0043.   EmailDto,
0044.   LoginUserDto,
0045.   PasswordDto,
0046.   RegisterUserDto,
0047.   UserSettingsDto,
0048. } from '../dto';
0049. import { JwtAuthGuard } from 'src/guards/jwt-guard';
0050. import { AccountService } from '../account.service';
0051. import { MailService } from '../mail/mail.service';
0052. import { FileInterceptor } from '@nestjs/platform-express';
0053. import { UserID } from 'src/utils/decorators/user-id.decorator';
0054.
0055. @Controller('api/users')
0056. export class AccountController {
0057.   constructor(
0058.     private readonly accountService: AccountService,
0059.     private readonly mailService: MailService,
0060.   ) {}
0061.
0062.   @Post('init-signup')
0063.   @HttpCode(200)
0064.   async initSignup(@Body() dto: EmailDto) {
0065.     const token = await this.accountService.generateEmailToken(dto.email);
0066.     await this.mailService.sendMail({
```

```

0067.         to: dto.email,
0068.         subject: 'Подтвердите ваш email',
0069.         html: `
0070.         <h2>Подтвердите регистрацию</h2>
0071.         <p>Нажмите на ссылку ниже, чтобы подтвердить ваш email:</p>
0072.         <a href="http://192.168.100.107:3000/confirm-email?token=${token}">
0073.             Подтвердить email
0074.         </a>
0075.         `;
0076.     });
0077.     return { success: true };
0078. }
0079.
0080. @Get('confirm-email')
0081. @HttpCode(200)
0082. async confirmEmail(@Query('token') token: string) {
0083.     const email = await this.accountService.verifyEmailToken(token);
0084.     return { email };
0085. }
0086.
0087. @Post('complete-signup')
0088. @HttpCode(201)
0089. async completeSignup(
0090.     @Body() { token, password }: { token: string; password: string },
0091. ) {
0092.     const email = await this.accountService.verifyEmailToken(token);
0093.     const registerDto = new RegisterUserDto();
0094.     registerDto.email = email;
0095.     registerDto.name = 'NewUser';
0096.     registerDto.password = password;
0097.     return this.accountService.registerUser(registerDto);
0098. }
0099.
0100. @UseGuards(JwtAuthGuard)
0101. @Get('pages')
0102. @HttpCode(200)
0103. async getPages(@UserID('user_id', ParseIntPipe) user_id: number) {
0104.     return this.accountService.getPages(user_id);
0105. }
0106.
0107. @UseGuards(JwtAuthGuard)
0108. @Get('settings')
0109. @HttpCode(200)
0110. async getSettings(@UserID('user_id', ParseIntPipe) user_id: number) {
0111.     const Settings = await this.accountService.getSetting(user_id);
0112.     return {
0113.         theme: Settings.theme,
0114.         picture_url: Settings.picture_url,
0115.         font: Settings.font,
0116.         font_size: Settings.font_size,
0117.     };
0118. }
0119.
0120. @UseGuards(JwtAuthGuard)
0121. @Get('workspace')
0122. @HttpCode(200)
0123. async getWorkspaces(@UserID('user_id', ParseIntPipe) user_id: number) {
0124.     const workspace = await this.accountService.getWorkspace(user_id);
0125.     return { name: workspace.name };
0126. }
0127.
0128. @UseGuards(JwtAuthGuard)
0129. @Get('user')
0130. @HttpCode(200)
0131. async getUsers(@UserID('user_id', ParseIntPipe) user_id: number) {
0132.     const user = await this.accountService.getUser(user_id);
0133.     return { email: user.email, name: user.name };
0134. }
0135.
0136. @Post('login')
0137. @HttpCode(200)
0138. async loginUsers(@Body() dto: LoginUserDto) {
0139.     return this.accountService.loginUser(dto);
0140. }
0141.
0142. @UseGuards(JwtAuthGuard)
0143. @Delete('destroy')
0144. @HttpCode(204)
0145. async destroyUsers(
0146.     @UserID('user_id', ParseIntPipe) user_id: number,

```

```

0147. ): Promise<number> {
0148.     return this.accountService.destroyUser(user_id);
0149. }
0150.
0151. @Post('request-password-reset')
0152. @HttpCode(200)
0153. async requestPasswordResets(@Body() dto: EmailDto) {
0154.     const res = await this.accountService.sendPasswordResetLink(dto);
0155.     await this.mailService.sendMail({
0156.         to: res.email,
0157.         subject: 'Смена пароля',
0158.         html: `
0159.             <h2>Подтвердите смену пароля</h2>
0160.             <p>Нажмите на ссылку ниже, чтобы сменить ваш пароль:</p>
0161.             <a href="http://192.168.100.107:3000/reset-password?token=${res.resetToken}">
0162.                 Сменить пароль
0163.             </a>
0164.         `,
0165.     });
0166.     return { success: true };
0167. }
0168.
0169. @UseGuards(JwtAuthGuard)
0170. @Patch('reset-password')
0171. @HttpCode(200)
0172. async resetPasswords(
0173.     @UserID('user_id', ParseIntPipe) user_id: number,
0174.     @Body() dto: PasswordDto,
0175. ) {
0176.     return this.accountService.resetPassword(dto, user_id);
0177. }
0178.
0179. @UseGuards(JwtAuthGuard)
0180. @Patch('update-settings')
0181. @HttpCode(200)
0182. @UseInterceptors(FileInterceptor('avatar'))
0183. async updateSettings(
0184.     @UserID('user_id', ParseIntPipe) user_id: number,
0185.     @Body() dto: UserSettingsDto,
0186.     @UploadedFile(
0187.         new ParseFilePipe({
0188.             validators: [
0189.                 new MaxFileSizeValidator({ maxSize: 5242880 }),
0190.                 new FileTypeValidator({ fileType: 'image/' }),
0191.             ],
0192.             fileIsRequired: false,
0193.         }),
0194.     )
0195.     file: Express.Multer.File,
0196. ) {
0197.     dto.user_id = user_id;
0198.
0199.     return this.accountService.updateUserSettings(dto, file);
0200. }
0201. }

```

account.service.ts

```

0202. import {
0203.     BadRequestException,
0204.     ConflictException,
0205.     Inject,
0206.     Injectable,
0207.     UnauthorizedException,
0208. } from '@nestjs/common';
0209. import {
0210.     EmailDto,
0211.     LoginUserDto,
0212.     PasswordDto,
0213.     RegisterUserDto,
0214.     UserSettingsDto,
0215. } from './dto';
0216. import { UsersService } from '../users/users.service';
0217. import { ErrorLog } from 'src/errors';
0218. import { WorkspaceService } from '../workspace/workspace.service';
0219. import { CACHE_MANAGER } from '@nestjs/cache-manager';
0220. import { Cache } from 'cache-manager';
0221. import { FileService } from '../file/file.service';
0222.
0223. @Injectable()

```

```

0224. export class AccountService {
0225.     constructor(
0226.         private readonly userService: UsersService,
0227.         private readonly workspaceService: WorkspaceService,
0228.         private readonly fileService: FileService,
0229.         @Inject(CACHE_MANAGER) private cache: Cache,
0230.     ) {}
0231.
0232.     async getPages(user_id: number) {
0233.         return this.workspaceService.getUserPagesByUserId(user_id);
0234.     }
0235.
0236.     async getSetting(user_id: number) {
0237.         const userSettings =
0238.             await this.userService.findUserSettingsByUserId(user_id);
0239.         if (!userSettings) {
0240.             throw new BadRequestException(ErrorLog.SETTINGS_NOT_EXIST);
0241.         }
0242.         return userSettings;
0243.     }
0244.
0245.     async getWorkspace(user_id: number) {
0246.         const workspace =
0247.             await this.workspaceService.findWorkspaceByUserId(user_id);
0248.         if (!workspace) {
0249.             throw new BadRequestException(ErrorLog.WORKSPACE_NOT_EXIST);
0250.         }
0251.         return workspace;
0252.     }
0253.
0254.     async getUser(user_id: number) {
0255.         const user = await this.userService.findUserById(user_id);
0256.         if (!user) {
0257.             throw new BadRequestException(ErrorLog.USER_NOT_EXIST);
0258.         }
0259.         return user;
0260.     }
0261.
0262.     async registerUser(dto: RegisterUserDto) {
0263.         const userExistCheck = await this.userService.findUserByEmail(dto.email);
0264.         if (userExistCheck) {
0265.             throw new ConflictException(ErrorLog.USER_EXIST);
0266.         }
0267.         dto.password = await this.userService.passwordHash(dto.password);
0268.         const newUser = await this.userService.createUser(dto);
0269.         await this.userService.createUserSettings({
0270.             theme: 'dark',
0271.             picture_url: '',
0272.             font: 'Sans',
0273.             font_size: 14,
0274.             user_id: newUser.User_id,
0275.         });
0276.         const newWorkspace = await this.workspaceService.createWorkspace(
0277.             newUser.User_id,
0278.         );
0279.         const newPage = await this.workspaceService.createPage(
0280.             null,
0281.             newWorkspace.Workspace_id,
0282.         );
0283.         await this.workspaceService.createWorkspaceMember({
0284.             user_id: newUser.User_id,
0285.             page_id: newPage.Page_id,
0286.             role: 'owner',
0287.         });
0288.         const tokenJWT: string = await this.userService.generateToken(newUser);
0289.         return { tokenJWT };
0290.     }
0291.
0292.     async generateEmailToken(email: string): Promise<string> {
0293.         const token = Math.random().toString(36).slice(2, 10);
0294.         await this.cache.set(`email:confirm:${token}`, email, 600000);
0295.         return token;
0296.     }
0297.
0298.     async verifyEmailToken(token: string): Promise<string> {
0299.         const email = await this.cache.get<string>(`email:confirm:${token}`);
0300.         if (!email) {
0301.             throw new BadRequestException(ErrorLog.EML_TOKEN_FAILURE);
0302.         }
0303.         return email;

```

```

0304.   }
0305.
0306.   async deleteToken(token: string): Promise<void> {
0307.       await this.cache.del(`email:confirm:${token}`);
0308.   }
0309.
0310.   async loginUser(dto: LoginUserDto) {
0311.       const userExistCheck = await this.userService.findUserByEmail(dto.email);
0312.       if (!userExistCheck) {
0313.           throw new UnauthorizedException(ErrorLog.LOGIN_FAILURE);
0314.       }
0315.       const passwordCheck: boolean = await this.userService.passwordVerify(
0316.           userExistCheck.password_hash,
0317.           dto.password,
0318.       );
0319.       if (!passwordCheck) {
0320.           throw new UnauthorizedException(ErrorLog.LOGIN_FAILURE);
0321.       }
0322.       const tokenJWT: string =
0323.           await this.userService.generateToken(userExistCheck);
0324.       return { tokenJWT };
0325.   }
0326.
0327.   async destroyUser(user_id: number): Promise<number> {
0328.       return this.userService.deleteUser(user_id);
0329.   }
0330.
0331.   async sendPasswordResetLink(dto: EmailDto) {
0332.       const email = dto.email;
0333.       const userExistCheck = await this.userService.findUserByEmail(email);
0334.       if (!userExistCheck) {
0335.           throw new BadRequestException(ErrorLog.USER_NOT_EXIST);
0336.       }
0337.       const resetToken: string =
0338.           await this.userService.generateToken(userExistCheck);
0339.       return { resetToken, email };
0340.   }
0341.
0342.   async resetPassword(dto: PasswordDto, user_id: number) {
0343.       dto.newPassword = await this.userService.passwordHash(dto.newPassword);
0344.       return this.userService.updateUserPassword(dto.newPassword, user_id);
0345.   }
0346.
0347.   async updateUserSettings(dto: UserSettingsDto, picture: Express.Multer.File) {
0348.       const { theme, font, font_size } = dto;
0349.       if (!dto.user_id) {
0350.           throw new BadRequestException(ErrorLog.SETTINGS_NOT_EXIST);
0351.       }
0352.       const settings = await this.userService.findUserSettingsByUserId(
0353.           dto.user_id,
0354.       );
0355.       if (!settings) {
0356.           throw new BadRequestException(ErrorLog.SETTINGS_NOT_EXIST);
0357.       }
0358.       let picture_url: string | undefined = undefined;
0359.       if (picture) {
0360.           picture_url = this.fileService.generateFileName(
0361.               picture,
0362.               dto.user_id,
0363.               true,
0364.           );
0365.           if (settings.picture_url !== '') {
0366.               await this.fileService.deleteFile(settings.picture_url, true);
0367.           }
0368.           await this.fileService.uploadFile(
0369.               picture_url,
0370.               picture.buffer,
0371.               picture.mimetype,
0372.               true,
0373.           );
0374.       }
0375.       return settings.update({ theme, picture_url, font, font_size });
0376.   }
0377. }

```

app.module.ts

```

0378. import { Module } from '@nestjs/common';
0379. import { AppController } from './app.controller';
0380. import { AppService } from './app.service';

```

```

0381. import { UsersModule } from 'src/modules/users/users.module';
0382. import { ConfigModule, ConfigService } from '@nestjs/config';
0383. import { SequelizeModule } from '@nestjs/sequelize';
0384. import configuration from 'src/configuration';
0385. import { User } from 'src/models/user.model';
0386. import { TokenModule } from 'src/modules/token/token.module';
0387. import { Workspace } from 'src/models/workspace.model';
0388. import { WorkspaceModule } from 'src/modules/workspace/workspace.module';
0389. import { Pages } from 'src/models/pages.model';
0390. import { AccountModule } from '../account/account.module';
0391. import { ManagerModule } from '../manager/manager.module';
0392. import { WorkspaceMembers } from 'src/models/workspace-members.model';
0393. import { Invitations } from 'src/models/invitations.model';
0394. import { Blocks } from 'src/models/blocks.model';
0395. import { EventsModule } from '../events/events.module';
0396. import { MailModule } from '../mail/mail.module';
0397. import { CacheModule } from '@nestjs/cache-manager';
0398. import { ChangeHistory } from 'src/models/change-history.model';
0399. import { Comments } from 'src/models/comments.model';
0400. import { Subscriptions } from 'src/models/subscriptions.model';
0401. import { Likes } from 'src/models/likes.model';
0402. import { UserSettings } from 'src/models/user-settings.model';
0403. import { FileModule } from '../file/file.module';
0404. import { Files } from 'src/models/files.model';
0405. import { ExplorerModule } from '../explorer/explorer.module';
0406.
0407. @Module({
0408.   imports: [
0409.     ConfigModule.forRoot({
0410.       isGlobal: true,
0411.       load: [configuration],
0412.     }),
0413.     SequelizeModule.forRootAsync({
0414.       imports: [ConfigModule],
0415.       inject: [ConfigService],
0416.       useFactory: (configService: ConfigService) => ({
0417.         dialect: 'postgres',
0418.         host: configService.get('db_host'),
0419.         port: configService.get('db_port'),
0420.         username: configService.get('db_user'),
0421.         password: configService.get('db_password'),
0422.         database: configService.get('db_name'),
0423.         synchronize: true,
0424.         autoLoadModels: true,
0425.         models: [
0426.           User,
0427.           Workspace,
0428.           Pages,
0429.           WorkspaceMembers,
0430.           Invitations,
0431.           Blocks,
0432.           ChangeHistory,
0433.           Comments,
0434.           Subscriptions,
0435.           Likes,
0436.           UserSettings,
0437.           Files,
0438.         ],
0439.         logging: false,
0440.       }),
0441.     ],
0442.     UsersModule,
0443.     WorkspaceModule,
0444.     TokenModule,
0445.     AccountModule,
0446.     ManagerModule,
0447.     EventsModule,
0448.     MailModule,
0449.     CacheModule.register({
0450.       isGlobal: true,
0451.     }),
0452.     FileModule,
0453.     ExplorerModule,
0454.   ],
0455.   controllers: [AppController],
0456.   providers: [AppService],
0457. })
0458. export class AppModule {}

```

events.gateway.ts

```

0459. import {
0460.   OnGatewayConnection,
0461.   OnGatewayDisconnect,
0462.   OnGatewayInit,
0463.   SubscribeMessage,
0464.   WebSocketGateway,
0465.   WebSocketServer,
0466.   WsException,
0467. } from '@nestjs/websockets';
0468. import { Server, Socket } from 'socket.io';
0469. import { ErrorLog } from 'src/errors';
0470. import { Operation, TransactionsStorage } from './dto';
0471. import { EventsService } from './events.service';
0472. import { Cron, CronExpression } from '@nestjs/schedule';
0473. import { DoublyLinkedList } from 'doubly-linked-list-typed';
0474. import { UsePipes, ValidationPipe } from '@nestjs/common';
0475.
0476. @UsePipes(
0477.   new ValidationPipe({
0478.     exceptionFactory: (errors) => {
0479.       console.log(errors);
0480.       new WsException(errors);
0481.     },
0482.     transform: true,
0483.     whitelist: true,
0484.     forbidNonWhitelisted: true,
0485.   }),
0486. )
0487. @WebSocketGateway(3002, {
0488.   cors: {
0489.     origin: '*',
0490.   },
0491. })
0492. export class EventsGateway
0493.   implements OnGatewayInit, OnGatewayDisconnect, OnGatewayConnection
0494. {
0495.   @WebSocketServer()
0496.   server: Server;
0497.
0498.   constructor(private readonly eventsService: EventsService) {}
0499.
0500.   private clientsStorage = new Map<number, Socket>();
0501.
0502.   private transactionsStorage = new Map<number, TransactionsStorage>();
0503.
0504.   private listStorage = new Map<number, DoublyLinkedList<number>>();
0505.
0506.   private listStorageElements = new Map<number, Set<number>>();
0507.
0508.   private lockedRooms = new Set<number>();
0509.
0510.   afterInit(server: Server) {
0511.     server.use((client: Socket, next) => {
0512.       void (async () => {
0513.         const token = client.handshake.auth?.token as string | undefined;
0514.         try {
0515.           const answer = await this.eventsService.verifyUserForWS(
0516.             token,
0517.             client.handshake.query,
0518.           );
0519.           const payload = answer.payload;
0520.           const roomId = answer.roomId;
0521.
0522.           if (this.clientsStorage.has(payload.user_id)) {
0523.             throw new WsException(ErrorLog.WS_CONNECTION);
0524.           }
0525.
0526.           this.clientsStorage.set(payload.user_id, client);
0527.           client.data = { user_id: payload.user_id };
0528.
0529.           await client.join(roomId.toString());
0530.
0531.           next();
0532.         } catch (error) {
0533.           next(
0534.             error instanceof WsException
0535.               ? error
0536.               : new WsException(ErrorLog.WS_INIT),
0537.           );

```



```

0538.         }
0539.     })();
0540. });
0541. }
0542.
0543. async handleConnection(client: Socket) {
0544.     const usersRoom = this.eventsService.getUserRoom(client);
0545.
0546.     const res = await this.eventsService.getBlocksAndStructure(usersRoom);
0547.     const sortedBlocks = res.sortedBlocks;
0548.     const structure = res.structure;
0549.     const elements = res.elements;
0550.     const structureElements = res.structureElements;
0551.
0552.     if (!this.listStorage.has(usersRoom)) {
0553.         this.listStorage.set(usersRoom, new DoublyLinkedList(structure));
0554.     }
0555.
0556.     if (!this.listStorageElements.has(usersRoom)) {
0557.         this.listStorageElements.set(usersRoom, new Set(structureElements));
0558.     }
0559.
0560.     this.server.to(usersRoom.toString()).emit('page-init', {
0561.         bloks: sortedBlocks,
0562.     });
0563.
0564.     this.server.to(usersRoom.toString()).emit('page-init-list', {
0565.         list: structure,
0566.     });
0567.
0568.     this.server.to(usersRoom.toString()).emit('page-init-elements', {
0569.         elements: elements,
0570.     });
0571.
0572.     this.server.to(usersRoom.toString()).emit('page-init-list-elements', {
0573.         structure_elements: structureElements,
0574.     });
0575. }
0576.
0577. handleDisconnect(client: Socket) {
0578.     if ((client.data as { user_id?: number })?.user_id) {
0579.         const user_id = (client.data as { user_id: number }).user_id;
0580.         if (this.clientsStorage.get(user_id)?.id === client.id) {
0581.             this.clientsStorage.delete(user_id);
0582.             console.log(
0583.                 `Client disconnected - User ID: ${user_id}, Socket ID: ${client.id}`,
0584.             );
0585.         }
0586.     } else {
0587.         console.log(`Unknown client disconnected - Socket ID: ${client.id}`);
0588.     }
0589. }
0590.
0591. @SubscribeMessage('add-operation')
0592. async handleTransaction(client: Socket, operation: Operation) {
0593.     // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment, @typescript-
0594.     // eslint/no-unsafe-member-access
0595.     const userId: number = client.data.user_id;
0596.
0597.     const usersRoom = this.eventsService.getUserRoom(client);
0598.
0599.     const right = await this.eventsService.checkUser(userId, usersRoom);
0600.     if (!right) {
0601.         return { status: 'failure' };
0602.     }
0603.
0604.     if (!this.transactionsStorage.has(usersRoom)) {
0605.         this.transactionsStorage.set(usersRoom, new TransactionsStorage());
0606.     }
0607.
0608.     const pageTransactions = this.transactionsStorage.get(usersRoom);
0609.     pageTransactions?.addOperation(operation);
0610.
0611.     return { status: 'success' };
0612. }
0613. @Cron(CronExpression.EVERY_5_SECONDS)
0614. async handleScheduledProcessing() {
0615.     for (const [usersRoom, storage] of this.transactionsStorage.entries()) {
0616.         if (this.lockedRooms.has(usersRoom)) {

```

```

0617.         continue;
0618.     }
0619.
0620.     if (storage.getCount() === 0) {
0621.         continue;
0622.     }
0623.
0624.     this.lockedRooms.add(usersRoom);
0625.     console.log(`Locked room ${usersRoom}`);
0626.
0627.     try {
0628.         const list = this.listStorage.get(usersRoom);
0629.         const listElements = this.listStorageElements.get(usersRoom);
0630.
0631.         const resultOfProcess = await this.eventsService.processStorage(
0632.             storage,
0633.             usersRoom,
0634.             list,
0635.             listElements,
0636.         );
0637.         const newList = resultOfProcess.list;
0638.         const createdBlocks = resultOfProcess.createdBlocks;
0639.         const updatedBlocks = resultOfProcess.updatedBlocks;
0640.         const createdElements = resultOfProcess.createdElements;
0641.         const updatedListElements = resultOfProcess.updatedListElements;
0642.         const updatedElements = resultOfProcess.updatedElements;
0643.         this.listStorage.set(usersRoom, newList);
0644.         this.listStorageElements.set(usersRoom, updatedListElements);
0645.         this.server.to(usersRoom.toString()).emit('change-list', {
0646.             list: newList.toArray(),
0647.         });
0648.         this.server.to(usersRoom.toString()).emit('change-create', {
0649.             created_blocks: createdBlocks,
0650.         });
0651.         this.server.to(usersRoom.toString()).emit('change-update', {
0652.             updated_blocks: updatedBlocks,
0653.         });
0654.         this.server.to(usersRoom.toString()).emit('change-create-elements', {
0655.             created_elements: createdElements,
0656.         });
0657.         this.server.to(usersRoom.toString()).emit('change-list-elements', {
0658.             list_elements: Array.from(updatedListElements),
0659.         });
0660.         this.server.to(usersRoom.toString()).emit('change-update-elements', {
0661.             updated_elements: updatedElements,
0662.         });
0663.     } catch (error) {
0664.         console.log(error);
0665.         if (error instanceof WsException) {
0666.             throw error;
0667.         }
0668.         throw new WsException(ErrorLog.CRON_FAILURE);
0669.     } finally {
0670.         if (this.lockedRooms.has(usersRoom)) {
0671.             this.lockedRooms.delete(usersRoom);
0672.             storage.clear();
0673.             console.log(`Unlocked room ${usersRoom}`);
0674.         }
0675.     }
0676. }
0677. }
0678. }

```

events.service.ts

```

0679. import { Injectable } from '@nestjs/common';
0680. import { TokenService } from '../token/token.service';
0681. import { WorkspaceService } from '../workspace/workspace.service';
0682. import { ParsedUrlQuery } from 'querystring';
0683. import { WsException } from '@nestjs/websockets';
0684. import { ErrorLog } from 'src/errors';
0685. import {
0686.     ChangePosOperationArgs,
0687.     CreateInDbOperationArgs,
0688.     CreateOperationArgs,
0689.     DeleteInDbOperationArgs,
0690.     DeleteOperationArgs,
0691.     Operation,
0692.     TransactionsStorage,
0693.     UpdateOperationArgs,

```

```

0694. } from './dto';
0695. import { Socket } from 'socket.io';
0696. import { DoublyLinkedList } from 'doubly-linked-list-typed';
0697. import { Blocks } from 'src/models/blocks.model';
0698.
0699. @Injectable()
0700. export class EventsService {
0701.   constructor(
0702.     private readonly tokenService: TokenService,
0703.     private readonly workspaceService: WorkspaceService,
0704.   ) {}
0705.
0706.   async verifyUserForWS(token: string | undefined, query: ParsedUrlQuery) {
0707.     const { page_id } = query;
0708.
0709.     if (!token || !page_id || Array.isArray(page_id)) {
0710.       throw new WsException(ErrorLog.WS_PARAMS);
0711.     }
0712.
0713.     const roomId = Number(page_id);
0714.     if (isNaN(roomId)) {
0715.       throw new WsException(ErrorLog.WS_PARAMS);
0716.     }
0717.
0718.     const payload = this.tokenService.verifyJwtToken(token);
0719.     const right = await this.workspaceService.checkRightConnectToPage(
0720.       payload.user_id,
0721.       roomId,
0722.       'all',
0723.     );
0724.
0725.     if (!right) {
0726.       throw new WsException(ErrorLog.RIGHTS_FAILURE);
0727.     }
0728.
0729.     return { payload, roomId };
0730.   }
0731.
0732.   async checkUser(user_id: number, roomId: number) {
0733.     return this.workspaceService.checkRightConnectToPage(
0734.       user_id,
0735.       roomId,
0736.       'all',
0737.     );
0738.   }
0739.
0740.   async processStorage(
0741.     storage: TransactionsStorage,
0742.     usersRoom: number,
0743.     list: DoublyLinkedList<number> | undefined,
0744.     listElements: Set<number> | undefined,
0745.   ) {
0746.     if (list === undefined) {
0747.       throw new WsException(ErrorLog.WS_LIST);
0748.     }
0749.     if (listElements === undefined) {
0750.       throw new WsException(ErrorLog.WS_LIST_EL);
0751.     }
0752.
0753.     const updatedListElements = new Set(listElements);
0754.
0755.     const originalList = new DoublyLinkedList<number>();
0756.     list.forEach((blockId) => originalList.push(blockId));
0757.
0758.     const originalListSet = new Set<number>();
0759.     let current = originalList.head;
0760.     while (current) {
0761.       originalListSet.add(current.value);
0762.       current = current.next;
0763.     }
0764.
0765.     const operationsByType = {
0766.       update: storage.getUpdateOperations(),
0767.       create: storage.getCreateOperations(),
0768.       delete: storage.getDeleteOperations(),
0769.       changePosition: storage.getChangePosOperations(),
0770.       createInDb: storage.getCreateInDbOperations(),
0771.       deleteInDb: storage.getDeleteInDbOperations(),
0772.     };
0773.

```

```

0774.     const deleteBlockIds = this.filterDeleteOps(
0775.         operationsByType.delete,
0776.         originalListSet,
0777.     );
0778.
0779.     const filteredUpdateOps = this.filterUpdateOps(
0780.         operationsByType.update,
0781.         deleteBlockIds,
0782.         originalListSet,
0783.         updatedListElements,
0784.     );
0785.
0786.     const filteredCreateOps = this.filterCreateOps(
0787.         operationsByType.create,
0788.         originalListSet,
0789.     );
0790.
0791.     const filteredChangePosOps = this.filterChangePosOps(
0792.         operationsByType.changePosition,
0793.         originalListSet,
0794.     );
0795.
0796.     const { rowOps, columnOps } = this.filterCreateInDbOps(
0797.         operationsByType.createInDb,
0798.         originalListSet,
0799.         deleteBlockIds,
0800.     );
0801.
0802.     const { delRowOps, delColumnOps } = this.filterDeleteInDbOps(
0803.         operationsByType.deleteInDb,
0804.         originalListSet,
0805.         deleteBlockIds,
0806.     );
0807.
0808.     const createdBlocksInfo: { id: number; pointer_to: number | null }[] = [];
0809.     let createdBlocks: Blocks[] = [];
0810.     const createdElements: Blocks[] = [];
0811.     let updatedBlocks: Blocks[] = [];
0812.     let updatedElements: Blocks[] = [];
0813.
0814.     if (filteredUpdateOps.length > 0) {
0815.         const res = await this.workspaceService.updateBlocks(filteredUpdateOps);
0816.         updatedBlocks = res.updatedBlocks;
0817.         createdElements.push(...res.newColumns);
0818.         updatedElements = res.updatedElements;
0819.     }
0820.
0821.     if (delRowOps.length > 0 || delColumnOps.length > 0) {
0822.         const delIds = await this.workspaceService.deleteInDbs(
0823.             delRowOps,
0824.             delColumnOps,
0825.         );
0826.         delIds.forEach((id) => updatedListElements.delete(id));
0827.     }
0828.
0829.     if (rowOps.length > 0 || columnOps.length > 0) {
0830.         const res = await this.workspaceService.createInDbs(
0831.             rowOps,
0832.             columnOps,
0833.             updatedListElements,
0834.         );
0835.         const createdRowsColumns = res.createdElements;
0836.         res.updatedListElements.forEach((id) => updatedListElements.add(id));
0837.         createdElements.push(...createdRowsColumns);
0838.     }
0839.
0840.     if (filteredCreateOps.length > 0) {
0841.         const res = await this.workspaceService.createBlocks(
0842.             filteredCreateOps,
0843.             usersRoom,
0844.         );
0845.         createdBlocks = res.createdBlocks;
0846.         createdElements.push(...res.newColumns);
0847.         createdElements.forEach((element) =>
0848.             updatedListElements.add(element.Block_id),
0849.         );
0850.
0851.         createdBlocks.forEach((block) => {
0852.             createdBlocksInfo.push({
0853.                 id: block.Block_id,

```

```

0854.         pointer_to: block.pointer_to,
0855.     });
0856. });
0857. for (const info of createdBlocksInfo) {
0858.     if (info.pointer_to !== null) {
0859.         const targetNode = list.getNode(info.pointer_to);
0860.         if (targetNode) {
0861.             list.addAfter(targetNode, info.id);
0862.         }
0863.     } else {
0864.         list.unshift(info.id);
0865.     }
0866. }
0867. }
0868.
0869. if (filteredChangePosOps.length > 0) {
0870.     for (const op of filteredChangePosOps) {
0871.         const { block_id, pointer_to } = op.args;
0872.
0873.         if (block_id === pointer_to) continue;
0874.
0875.         const movingNode = list.getNode(block_id);
0876.         if (!movingNode) continue;
0877.
0878.         if (pointer_to !== null) {
0879.             const targetNode = list.getNode(pointer_to);
0880.             if (targetNode) {
0881.                 list.delete(block_id);
0882.                 list.addAfter(targetNode, block_id);
0883.             }
0884.         } else {
0885.             list.delete(block_id);
0886.             list.unshift(block_id);
0887.         }
0888.
0889.         await this.workspaceService.createHistory(
0890.             'change-position',
0891.             {},
0892.             usersRoom,
0893.             block_id,
0894.         );
0895.     }
0896. }
0897.
0898. if (deleteBlockIds.size > 0) {
0899.     for (const id of deleteBlockIds) {
0900.         const block_id = id;
0901.         const nodeToDelete = list.getNode(block_id);
0902.
0903.         if (nodeToDelete) {
0904.             list.delete(block_id);
0905.         }
0906.     }
0907. }
0908.
0909. const originalArray = originalList.toArray();
0910. const currentArray = list.toArray();
0911. const changes: { block_id: number; pointer_to: number | null }[] = [];
0912.
0913. const originalMap = new Map<number, number>();
0914. originalArray.forEach((id, index) => originalMap.set(id, index));
0915.
0916. for (let i = 0; i < currentArray.length; i++) {
0917.     const blockId = currentArray[i];
0918.     const originalIndex = originalMap.get(blockId);
0919.
0920.     const prevInCurrent = i > 0 ? currentArray[i - 1] : null;
0921.
0922.     if (originalIndex === undefined) {
0923.         changes.push({ block_id: blockId, pointer_to: prevInCurrent });
0924.         continue;
0925.     }
0926.
0927.     const prevInOriginal =
0928.         originalIndex > 0 ? originalArray[originalIndex - 1] : null;
0929.
0930.     if (prevInOriginal !== prevInCurrent) {
0931.         changes.push({ block_id: blockId, pointer_to: prevInCurrent });
0932.     }
0933. }

```

```

0934.
0935.     if (changes.length > 0) {
0936.         await this.workspaceService.changePointers(changes);
0937.     }
0938.
0939.     if (deleteBlockIds.size > 0) {
0940.         const elementIdsDel = await this.workspaceService.deleteBlocks(
0941.             deleteBlockIds,
0942.             usersRoom,
0943.         );
0944.         elementIdsDel.forEach((id) => updatedListElements.delete(id));
0945.     }
0946.
0947.     return {
0948.         list,
0949.         createdBlocks,
0950.         updatedBlocks,
0951.         createdElements,
0952.         updatedListElements,
0953.         updatedElements,
0954.     };
0955. }
0956.
0957. filterUpdateOps(
0958.     updateOps: Operation<UpdateOperationArgs>[],
0959.     deleteBlockIds: Set<number>,
0960.     originalListSet: Set<number>,
0961.     updatedListElements: Set<number>,
0962. ): Operation<UpdateOperationArgs>[] {
0963.     if (updateOps.length === 0) {
0964.         return updateOps;
0965.     }
0966.     const latestUpdates = new Map<number, Operation<UpdateOperationArgs>>();
0967.
0968.     for (let i = updateOps.length - 1; i >= 0; i--) {
0969.         const op = updateOps[i];
0970.         const blockId = op.args.block_id;
0971.
0972.         if (!op.args.is_element) {
0973.             if (!originalListSet.has(blockId)) continue;
0974.         } else {
0975.             if (!updatedListElements.has(blockId)) continue;
0976.         }
0977.
0978.         if (!latestUpdates.has(blockId)) {
0979.             latestUpdates.set(blockId, op);
0980.         }
0981.     }
0982.
0983.     const result = Array.from(latestUpdates.values());
0984.     return deleteBlockIds.size === 0
0985.         ? result
0986.         : result.filter((op) => !deleteBlockIds.has(op.args.block_id));
0987. }
0988.
0989. filterDeleteOps(
0990.     deleteOps: Operation<DeleteOperationArgs>[],
0991.     originalListSet: Set<number>,
0992. ): Set<number> {
0993.     if (deleteOps.length === 0) {
0994.         return new Set<number>();
0995.     }
0996.     const deleteBlockIds = new Set(deleteOps.map((op) => op.args.block_id));
0997.     const filteredDeleteBlockIds = new Set<number>();
0998.     for (const blockId of deleteBlockIds) {
0999.         if (originalListSet.has(blockId)) {
1000.             filteredDeleteBlockIds.add(blockId);
1001.         }
1002.     }
1003.     return filteredDeleteBlockIds;
1004. }
1005.
1006. filterCreateOps(
1007.     createOps: Operation<CreateOperationArgs>[],
1008.     originalListSet: Set<number>,
1009. ) {
1010.     if (createOps.length === 0) {
1011.         return createOps;
1012.     }
1013.

```

```

1014.     const validCreateOps = createOps.filter((op) => {
1015.         if (op.args.pointer_to === null) return true;
1016.         return originalListSet.has(op.args.pointer_to);
1017.     });
1018.
1019.     return validCreateOps;
1020. }
1021.
1022. filterChangePosOps(
1023.     changePosOps: Operation<ChangePosOperationArgs>[],
1024.     originalListSet: Set<number>,
1025. ) {
1026.     if (changePosOps.length === 0) {
1027.         return changePosOps;
1028.     }
1029.
1030.     return changePosOps.filter((op) => {
1031.         if (!originalListSet.has(op.args.block_id)) {
1032.             return false;
1033.         }
1034.         return (
1035.             op.args.pointer_to === null || originalListSet.has(op.args.pointer_to)
1036.         );
1037.     });
1038. }
1039.
1040. filterCreateInDbOps(
1041.     createOps: Operation<CreateInDbOperationArgs>[],
1042.     originalListSet: Set<number>,
1043.     deleteBlockIds: Set<number>,
1044. ): {
1045.     rowOps: Operation<CreateInDbOperationArgs>[];
1046.     columnOps: Operation<CreateInDbOperationArgs>[];
1047. } {
1048.     if (createOps.length === 0) {
1049.         return { rowOps: [], columnOps: [] };
1050.     }
1051.     const rowOps: Operation<CreateInDbOperationArgs>[] = [];
1052.     const columnOps: Operation<CreateInDbOperationArgs>[] = [];
1053.     createOps.forEach((op) => {
1054.         if (!originalListSet.has(op.args.block_db_id)) {
1055.             return;
1056.         }
1057.         if (deleteBlockIds.has(op.args.block_db_id)) {
1058.             return;
1059.         }
1060.         const { is_row, is_column, property_type } = op.args;
1061.         if (is_row === is_column) {
1062.             return;
1063.         }
1064.         if (is_column && property_type === null) {
1065.             return;
1066.         }
1067.         if (is_row) {
1068.             rowOps.push(op);
1069.         } else if (is_column) {
1070.             columnOps.push(op);
1071.         }
1072.     });
1073.     return { rowOps, columnOps };
1074. }
1075.
1076. filterDeleteInDbOps(
1077.     deleteOps: Operation<DeleteInDbOperationArgs>[],
1078.     originalListSet: Set<number>,
1079.     deleteBlockIds: Set<number>,
1080. ): {
1081.     delRowOps: Operation<DeleteInDbOperationArgs>[];
1082.     delColumnOps: Operation<DeleteInDbOperationArgs>[];
1083. } {
1084.     const delRowOps: Operation<DeleteInDbOperationArgs>[] = [];
1085.     const delColumnOps: Operation<DeleteInDbOperationArgs>[] = [];
1086.
1087.     if (deleteOps.length === 0) {
1088.         return { delRowOps, delColumnOps };
1089.     }
1090.
1091.     for (const op of deleteOps) {
1092.         const hasRow = op.args.row !== null && op.args.row !== undefined;
1093.         const hasColumn = op.args.column !== null && op.args.column !== undefined;

```

```

1094.
1095.     if (!(hasRow || hasColumn) || (hasRow && hasColumn)) {
1096.         continue;
1097.     }
1098.
1099.     const dbId = op.args.block_db_id;
1100.     if (!originalListSet.has(dbId)) {
1101.         continue;
1102.     }
1103.
1104.     if (deleteBlockIds.has(dbId)) {
1105.         continue;
1106.     }
1107.
1108.     if (hasRow) {
1109.         delRowOps.push(op);
1110.     } else {
1111.         delColumnOps.push(op);
1112.     }
1113. }
1114.
1115. return { delRowOps, delColumnOps };
1116. }
1117.
1118. async getBlocksAndStructure(roomId: number) {
1119.     const res = await this.workspaceService.getBlockByPageId(roomId);
1120.     const sortedBlocks = res.sortedBlocks;
1121.     const elements = res.elements;
1122.     const structure = sortedBlocks.map((block) => block.Block_id);
1123.     const structureElements = elements.map((block) => block.Block_id);
1124.     return { sortedBlocks, structure, elements, structureElements };
1125. }
1126.
1127. getUserRoom(client: Socket) {
1128.     const userRooms = Array.from(client.rooms).filter(
1129.         (room) => room !== client.id,
1130.     );
1131.
1132.     if (userRooms.length !== 1) {
1133.         throw new WsException(ErrorLog.WS_ROOMS);
1134.     }
1135.
1136.     const [userRoom] = userRooms;
1137.     return Number(userRoom);
1138. }
1139. }

```

file.service.ts

```

1140. import { Injectable } from '@nestjs/common';
1141. import {
1142.     DeleteObjectCommand,
1143.     GetObjectCommand,
1144.     ListObjectsCommand,
1145.     PutObjectCommand,
1146.     S3Client,
1147.     S3ClientConfig,
1148. } from '@aws-sdk/client-s3';
1149. import { ConfigService } from '@nestjs/config';
1150. import { getSignedUrl } from '@aws-sdk/s3-request-presigner';
1151. import { InjectModel } from '@nestjs/sequelize';
1152. import { Files } from 'src/models/files.model';
1153.
1154. @Injectable()
1155. export class FileService {
1156.     private readonly s3Client: S3Client;
1157.     private readonly publicBucket: string;
1158.     private readonly privateBucket: string;
1159.
1160.     constructor(
1161.         private configService: ConfigService,
1162.         @InjectModel(Files)
1163.         private readonly filesRepository: typeof Files,
1164.     ) {
1165.         const endpoint = this.configService.get<string>('minio_endpoint');
1166.         const accessKeyId = this.configService.get<string>('minio_access_key');
1167.         const secretAccessKey = this.configService.get<string>('minio_secret_key');
1168.
1169.         if (!endpoint || !accessKeyId || !secretAccessKey) {
1170.             throw new Error('Missing required MinIO configuration');

```



```

1171.     }
1172.
1173.     const config: S3ClientConfig = {
1174.         endpoint,
1175.         region: 'us-east-1',
1176.         credentials: {
1177.             accessKeyId,
1178.             secretAccessKey,
1179.         },
1180.         forcePathStyle: true,
1181.     };
1182.
1183.     this.s3Client = new S3Client(config);
1184.     this.publicBucket = 'knowlify-public';
1185.     this.privateBucket = 'knowlify-private';
1186. }
1187.
1188. async uploadFile(
1189.     fileName: string,
1190.     fileBuffer: Buffer,
1191.     contentType: string,
1192.     isPublic: boolean = false,
1193. ) {
1194.     const bucket = isPublic ? this.publicBucket : this.privateBucket;
1195.
1196.     const command = new PutObjectCommand({
1197.         Bucket: bucket,
1198.         Key: fileName,
1199.         Body: fileBuffer,
1200.         ContentType: contentType,
1201.     });
1202.
1203.     await this.s3Client.send(command);
1204.     return {
1205.         fileName,
1206.         bucket,
1207.         isPublic,
1208.     };
1209. }
1210.
1211. async getFile(fileName: string, isPublic: boolean = false) {
1212.     const bucket = isPublic ? this.publicBucket : this.privateBucket;
1213.
1214.     const command = new GetObjectCommand({
1215.         Bucket: bucket,
1216.         Key: fileName,
1217.     });
1218.
1219.     return await this.s3Client.send(command);
1220. }
1221.
1222. async getPresignedUrl(fileName: string, expiresIn: number = 3600) {
1223.     const command = new GetObjectCommand({
1224.         Bucket: this.privateBucket,
1225.         Key: fileName,
1226.     });
1227.
1228.     return getSignedUrl(this.s3Client, command, { expiresIn });
1229. }
1230.
1231. async listFiles(isPublic: boolean = false) {
1232.     const bucket = isPublic ? this.publicBucket : this.privateBucket;
1233.
1234.     const command = new ListObjectsCommand({
1235.         Bucket: bucket,
1236.     });
1237.
1238.     const response = await this.s3Client.send(command);
1239.     return response.Contents || [];
1240. }
1241.
1242. async deleteFile(fileName: string, isPublic: boolean = false) {
1243.     const bucket = isPublic ? this.publicBucket : this.privateBucket;
1244.
1245.     const command = new DeleteObjectCommand({
1246.         Bucket: bucket,
1247.         Key: fileName,
1248.     });
1249.
1250.     await this.s3Client.send(command);

```

```

1251.     return {
1252.         deleted: fileName,
1253.         bucket,
1254.     };
1255. }
1256.
1257. generateFileName(
1258.     file: Express.Multer.File,
1259.     user_id: number,
1260.     isPublic: boolean = false,
1261. ) {
1262.     const fileExtension = file.mimetype.split('/')[1];
1263.     const uniqueName = `user_${user_id}_${Date.now()}${fileExtension}`;
1264.     return isPublic
1265.         ? `user_public/${uniqueName}`
1266.         : `user_private/${uniqueName}`;
1267. }
1268.
1269. async createFileRecord(
1270.     name: string,
1271.     size: number,
1272.     block_id: number,
1273.     user_id: number,
1274. ) {
1275.     return this.filesRepository.create({
1276.         file_url: name,
1277.         size: size,
1278.         block_id: block_id,
1279.         upload_by: user_id,
1280.     });
1281. }
1282.
1283. async findFilesRecByBlockId(block_id: number) {
1284.     return this.filesRepository.findAll({
1285.         where: {
1286.             block_id: block_id,
1287.         },
1288.     });
1289. }
1290.
1291. async destroyFileRec(file_id: number) {
1292.     return this.filesRepository.destroy({
1293.         where: {
1294.             File_id: file_id,
1295.         },
1296.     });
1297. }
1298. }

```

manager.controller.ts

```

1299. import {
1300.     Body,
1301.     Controller,
1302.     Delete,
1303.     FileTypeValidator,
1304.     Get,
1305.     HttpStatusCode,
1306.     MaxFileSizeValidator,
1307.     ParseFilePipe,
1308.     ParseIntPipe,
1309.     Patch,
1310.     Post,
1311.     Query,
1312.     UploadedFile,
1313.     UseGuards,
1314.     UseInterceptors,
1315. } from '@nestjs/common';
1316. import { JwtAuthGuard } from 'src/guards/jwt-guard';
1317. import {
1318.     ChangeRolesDto,
1319.     ConfirmInvitationsDto,
1320.     CreateCommentsDto,
1321.     CreatePageDto,
1322.     DeleteCommentsDto,
1323.     DeleteMemberDto,
1324.     DeletePageDto,
1325.     InviteUsersDto,
1326.     LikeDto,
1327.     PageImgQueryDto,

```

```

1328.   RestorePagesDto,
1329.   SubscriptionDto,
1330.   UpdatePageDto,
1331. } from './dto';
1332. import { ManagerService } from './manager.service';
1333. import { MailService } from '../mail/mail.service';
1334. import { FileInterceptor } from '@nestjs/platform-express';
1335. import { UserID } from 'src/utlils/decorators/user-id.decorator';
1336.
1337. @Controller('api/manager')
1338. export class ManagerController {
1339.   constructor(
1340.     private readonly managerService: ManagerService,
1341.     private readonly mailService: MailService,
1342.   ) {}
1343.
1344.   @UseGuards(JwtAuthGuard)
1345.   @Post('create-page')
1346.   @HttpCode(201)
1347.   async createPages(
1348.     @UserID('user_id', ParseIntPipe) user_id: number,
1349.     @Body() dto: CreatePageDto,
1350.   ) {
1351.     return this.managerService.CreateNewPage(dto, user_id);
1352.   }
1353.
1354.   @UseGuards(JwtAuthGuard)
1355.   @Patch('update-page')
1356.   @HttpCode(200)
1357.   async updatePages(
1358.     @UserID('user_id', ParseIntPipe) user_id: number,
1359.     @Body() dto: UpdatePageDto,
1360.   ) {
1361.     return this.managerService.updatePages(dto, user_id);
1362.   }
1363.
1364.   @UseGuards(JwtAuthGuard)
1365.   @Patch('page-img')
1366.   @HttpCode(200)
1367.   @UseInterceptors(FileInterceptor('file'))
1368.   async updatePageImgs(
1369.     @UserID('user_id', ParseIntPipe) user_id: number,
1370.     @Query() query: PageImgQueryDto,
1371.     @UploadedFile(
1372.       new ParseFilePipe({
1373.         validators: [
1374.           new MaxFileSizeValidator({ maxSize: 5242880 }),
1375.           new FileTypeValidator({ fileType: 'image/' }),
1376.         ],
1377.       )),
1378.   )
1379.   file: Express.Multer.File,
1380. ) {
1381.   const { type, page_id } = query;
1382.   return this.managerService.updatePageImg(file, user_id, page_id, type);
1383. }
1384.
1385. @UseGuards(JwtAuthGuard)
1386. @Delete('destroy-page')
1387. @HttpCode(200)
1388. async deletePages(
1389.   @UserID('user_id', ParseIntPipe) user_id: number,
1390.   @Body() dto: DeletePageDto,
1391. ) {
1392.   return this.managerService.deletePages(dto, user_id);
1393. }
1394.
1395. @UseGuards(JwtAuthGuard)
1396. @Patch('restore-page')
1397. @HttpCode(200)
1398. async restorePages(
1399.   @UserID('user_id', ParseIntPipe) user_id: number,
1400.   @Body() dto: RestorePagesDto,
1401. ) {
1402.   return this.managerService.restorePage(dto, user_id);
1403. }
1404.
1405. @UseGuards(JwtAuthGuard)
1406. @Post('invite-user')
1407. @HttpCode(200)

```

```

1408. async inviteUsers(
1409.     @UserID('user_id', ParseIntPipe) user_id: number,
1410.     @Body() dto: InviteUsersDto,
1411. ) {
1412.     const res = await this.managerService.inviteUser(dto, user_id);
1413.     const invitation = res.invitation;
1414.     const owner = res.owner;
1415.     await this.mailService.sendMail({
1416.         to: invitation.invite_email,
1417.         subject: `Приглашение от ${owner}`,
1418.         html: `
1419.             <h2>Подтвердите приглашение</h2>
1420.             <p>Нажмите на ссылку ниже, чтобы подтвердить приглашение:</p>
1421.             <a href="http://frontend.example.com/confirm-invitation?token=${invitation.token}">
1422.                 Подтвердить приглашение
1423.             </a>
1424.         `,
1425.     });
1426.     return { success: true };
1427. }
1428.
1429. @UseGuards(JwtAuthGuard)
1430. @Post('confirm-invitation')
1431. @HttpCode(200)
1432. async confirmInvitations(
1433.     @UserID('user_id', ParseIntPipe) user_id: number,
1434.     @Query() query: ConfirmInvitationsDto,
1435. ) {
1436.     const { token } = query;
1437.     return this.managerService.confirmInvitation(token, user_id);
1438. }
1439.
1440. @UseGuards(JwtAuthGuard)
1441. @Get('members')
1442. @HttpCode(200)
1443. async getMembers(
1444.     @UserID('user_id', ParseIntPipe) user_id: number,
1445.     @Query('page_id', ParseIntPipe) page_id: number,
1446. ) {
1447.     return this.managerService.getMembers(page_id, user_id);
1448. }
1449.
1450. @UseGuards(JwtAuthGuard)
1451. @Delete('destroy-member')
1452. @HttpCode(204)
1453. async destroyMembers(
1454.     @UserID('user_id', ParseIntPipe) user_id: number,
1455.     @Body() dto: DeleteMemberDto,
1456. ) {
1457.     return this.managerService.destroyMember(dto, user_id);
1458. }
1459.
1460. @UseGuards(JwtAuthGuard)
1461. @Patch('change-role')
1462. @HttpCode(200)
1463. async changeRoles(
1464.     @UserID('user_id', ParseIntPipe) user_id: number,
1465.     @Body() dto: ChangeRolesDto,
1466. ) {
1467.     return this.managerService.changeRole(dto, user_id);
1468. }
1469.
1470. @UseGuards(JwtAuthGuard)
1471. @Post('create-comment')
1472. @HttpCode(201)
1473. async createComments(
1474.     @UserID('user_id', ParseIntPipe) user_id: number,
1475.     @Body() dto: CreateCommentsDto,
1476. ) {
1477.     return this.managerService.createComment(dto, user_id);
1478. }
1479.
1480. @UseGuards(JwtAuthGuard)
1481. @Get('get-comments')
1482. @HttpCode(200)
1483. async getComments(
1484.     @UserID('user_id', ParseIntPipe) user_id: number,
1485.     @Query('block_id', ParseIntPipe) block_id: number,
1486. ) {
1487.     return this.managerService.getComments(block_id, user_id);

```

```

1488.     }
1489.
1490.     @UseGuards(JwtAuthGuard)
1491.     @Delete('destroy-comment')
1492.     @HttpCode(204)
1493.     async destroyComments(
1494.         @UserID('user_id', ParseIntPipe) user_id: number,
1495.         @Body() dto: DeleteCommentsDto,
1496.     ) {
1497.         return this.managerService.destroyComment(dto, user_id);
1498.     }
1499.
1500.     @UseGuards(JwtAuthGuard)
1501.     @Post('subscribe')
1502.     @HttpCode(201)
1503.     async subscribeToPages(
1504.         @UserID('user_id', ParseIntPipe) user_id: number,
1505.         @Body() dto: SubscriptionDto,
1506.     ) {
1507.         return this.managerService.subscribeToPage(dto, user_id);
1508.     }
1509.
1510.     @UseGuards(JwtAuthGuard)
1511.     @Delete('unsubscribe')
1512.     @HttpCode(204)
1513.     async unsubscribeToPages(
1514.         @UserID('user_id', ParseIntPipe) user_id: number,
1515.         @Body() dto: SubscriptionDto,
1516.     ) {
1517.         return this.managerService.unsubscribeToPage(dto, user_id);
1518.     }
1519.
1520.     @UseGuards(JwtAuthGuard)
1521.     @Post('like')
1522.     @HttpCode(201)
1523.     async likePages(
1524.         @UserID('user_id', ParseIntPipe) user_id: number,
1525.         @Body() dto: LikeDto,
1526.     ) {
1527.         return this.managerService.likePage(dto, user_id);
1528.     }
1529.
1530.     @UseGuards(JwtAuthGuard)
1531.     @Delete('destroy-like')
1532.     @HttpCode(204)
1533.     async deleteLikePages(
1534.         @UserID('user_id', ParseIntPipe) user_id: number,
1535.         @Body() dto: LikeDto,
1536.     ) {
1537.         return this.managerService.deleteLikePage(dto, user_id);
1538.     }
1539. }

users.service.ts

1540. import { Injectable } from '@nestjs/common';
1541. import { InjectModel } from '@nestjs/sequelize';
1542. import { User } from 'src/models/user.model';
1543. import * as argon2 from 'argon2';
1544. import { RegisterUserDto, UserSettingsDto } from '../account/dto';
1545. import { TokenService } from 'src/modules/token/token.service';
1546. import { UserSettings } from 'src/models/user-settings.model';
1547.
1548. @Injectable()
1549. export class UsersService {
1550.     constructor(
1551.         @InjectModel(User) private readonly userRepository: typeof User,
1552.         @InjectModel(UserSettings)
1553.         private readonly userSettingsRepository: typeof UserSettings,
1554.         private readonly tokenService: TokenService,
1555.     ) {}
1556.
1557.     async passwordHash(password: string): Promise<string> {
1558.         return argon2.hash(password);
1559.     }
1560.
1561.     async passwordVerify(hash: string, password: string): Promise<boolean> {
1562.         return argon2.verify(hash, password);
1563.     }
1564. }

```

```

1565.   async findUserByEmail(email: string): Promise<User | null> {
1566.       return this.userRepository.findOne({ where: { email: email } });
1567.   }
1568.
1569.   async findUserById(User_id: number): Promise<User | null> {
1570.       return this.userRepository.findOne({ where: { User_id: User_id } });
1571.   }
1572.
1573.   async findUserSettingsByUserId(
1574.       User_id: number,
1575.   ): Promise<UserSettings | null> {
1576.       return this.userSettingsRepository.findOne({
1577.           where: { user_id: User_id },
1578.       });
1579.   }
1580.
1581.   async createUser(dto: RegisterUserDto): Promise<User> {
1582.       return this.userRepository.create({
1583.           email: dto.email,
1584.           name: dto.name,
1585.           password_hash: dto.password,
1586.           storage_limit: 2048,
1587.           storage: 0,
1588.       });
1589.   }
1590.
1591.   async generateToken(user: User): Promise<string> {
1592.       return this.tokenService.generateJwtToken(user.User_id);
1593.   }
1594.
1595.   async deleteUser(user_id: number): Promise<number> {
1596.       return this.userRepository.destroy({
1597.           where: { User_id: user_id },
1598.       });
1599.   }
1600.
1601.   async updateUserPassword(newPassword: string, user_id: number) {
1602.       return this.userRepository.update(
1603.           { password_hash: newPassword },
1604.           {
1605.               where: { User_id: user_id },
1606.           },
1607.       );
1608.   }
1609.
1610.   async createUserSettings(dto: UserSettingsDto): Promise<UserSettings> {
1611.       return this.userSettingsRepository.create({
1612.           theme: dto.theme,
1613.           picture_url: dto.picture_url,
1614.           font: dto.font,
1615.           font_size: dto.font_size,
1616.           user_id: dto.user_id,
1617.       });
1618.   }
1619. }

```

workspace.module.ts

```

1620. import { Module } from '@nestjs/common';
1621. import { WorkspaceController } from './workspace.controller';
1622. import { WorkspaceService } from './workspace.service';
1623. import { SequelizeModule } from '@nestjs/sequelize';
1624. import { Workspace } from 'src/models/workspace.model';
1625. import { Pages } from 'src/models/pages.model';
1626. import { WorkspaceMembers } from 'src/models/workspace-members.model';
1627. import { Blocks } from 'src/models/blocks.model';
1628. import { Invitations } from 'src/models/invitations.model';
1629. import { ChangeHistory } from 'src/models/change-history.model';
1630. import { Comments } from 'src/models/comments.model';
1631. import { Subscriptions } from 'src/models/subscriptions.model';
1632. import { Likes } from 'src/models/likes.model';
1633.
1634. @Module({
1635.   imports: [
1636.     SequelizeModule.forFeature([
1637.       Workspace,
1638.       Pages,
1639.       WorkspaceMembers,
1640.       Blocks,
1641.       Invitations,

```

```

1642.         ChangeHistory,
1643.         Comments,
1644.         Subscriptions,
1645.         Likes,
1646.     ]),
1647. ],
1648. controllers: [WorkspaceController],
1649. providers: [WorkspaceService],
1650. exports: [WorkspaceService],
1651. })
1652. export class WorkspaceModule {}

workspace.service.ts

1653. import { Injectable } from '@nestjs/common';
1654. import { InjectModel } from '@nestjs/sequelize';
1655. import { Workspace } from 'src/models/workspace.model';
1656. import { Pages } from 'src/models/pages.model';
1657. import {
1658.     CreateBlockDto,
1659.     CreateWorkspaceMemberDto,
1660.     InviteUsersDto,
1661.     LikeDto,
1662.     SubscriptionDto,
1663.     UpdatePageDto,
1664. } from '../manager/dto';
1665. import { WorkspaceMembers } from 'src/models/workspace-members.model';
1666. import { User } from 'src/models/user.model';
1667. import { Blocks } from 'src/models/blocks.model';
1668. import {
1669.     CreateInDbOperationArgs,
1670.     CreateOperationArgs,
1671.     DeleteInDbOperationArgs,
1672.     Operation,
1673.     UpdateOperationArgs,
1674. } from '../events/dto';
1675. import { Sequelize } from 'sequelize-typescript';
1676. import { ErrorLog } from 'src/errors';
1677. import { Transaction } from 'sequelize';
1678. import { DataBaseComponentDto } from '../events/dto/database.component';
1679. import { Invitations } from 'src/models/invitations.model';
1680. import { ChangeHistory } from 'src/models/change-history.model';
1681. import { Comments } from 'src/models/comments.model';
1682. import { Subscriptions } from 'src/models/subscriptions.model';
1683. import { Likes } from 'src/models/likes.model';
1684.
1685. @Injectable()
1686. export class WorkspaceService {
1687.     constructor(
1688.         @InjectModel(Workspace)
1689.         private readonly workspaceRepository: typeof Workspace,
1690.         @InjectModel(Pages)
1691.         private readonly pageRepository: typeof Pages,
1692.         @InjectModel(WorkspaceMembers)
1693.         private readonly workspaceMembersRepository: typeof WorkspaceMembers,
1694.         @InjectModel(Blocks)
1695.         private readonly blocksRepository: typeof Blocks,
1696.         @InjectModel(Invitations)
1697.         private readonly invitationsRepository: typeof Invitations,
1698.         @InjectModel(ChangeHistory)
1699.         private readonly changeHistoryRepository: typeof ChangeHistory,
1700.         @InjectModel(Comments)
1701.         private readonly commentsRepository: typeof Comments,
1702.         @InjectModel(Subscriptions)
1703.         private readonly subscriptionsRepository: typeof Subscriptions,
1704.         @InjectModel(Likes)
1705.         private readonly likesRepository: typeof Likes,
1706.         private readonly sequelize: Sequelize,
1707.     ) {}
1708.
1709.     async findWorkspaceByUserId(user_id: number): Promise<Workspace | null> {
1710.         return this.workspaceRepository.findOne({ where: { user_id: user_id } });
1711.     }
1712.
1713.     async findPageById(
1714.         page_id: number,
1715.         includeDeleted: boolean = false,
1716.     ): Promise<Pages | null> {
1717.         return this.pageRepository.findOne({
1718.             where: { Page_id: page_id },

```

```

1719.         paranoid: !includeDeleted,
1720.     });
1721. }
1722.
1723. async findPagesByParentId(
1724.     parent_page_id: number,
1725.     includeDeleted: boolean = false,
1726. ): Promise<Pages[]> {
1727.     return this.pageRepository.findAll({
1728.         where: { parent_page_id },
1729.         paranoid: !includeDeleted,
1730.     });
1731. }
1732.
1733. async restorePage(pageId: number) {
1734.     return this.pageRepository.restore({
1735.         where: { Page_id: pageId },
1736.     });
1737. }
1738.
1739. async findInvitationByToken(token: string) {
1740.     const invitation = await this.invitationsRepository.findOne({
1741.         where: { token },
1742.     });
1743.     if (!invitation) {
1744.         return null;
1745.     }
1746.     const now = new Date();
1747.     if (invitation.expires_at < now) {
1748.         return null;
1749.     }
1750.     return invitation;
1751. }
1752.
1753. async findCommentById(comment_id: number): Promise<Comments | null> {
1754.     return this.commentsRepository.findOne({
1755.         where: { Comment_id: comment_id },
1756.     });
1757. }
1758.
1759. async findCommentsByBlockId(block_id: number): Promise<Comments[]> {
1760.     return this.commentsRepository.findAll({
1761.         where: { block_id: block_id },
1762.     });
1763. }
1764.
1765. async findSubscriptionById(
1766.     user_id: number,
1767.     page_id: number,
1768. ): Promise<Subscriptions | null> {
1769.     return this.subscriptionsRepository.findOne({
1770.         where: {
1771.             user_id,
1772.             page_id,
1773.         },
1774.     });
1775. }
1776.
1777. async findLikeById(user_id: number, page_id: number): Promise<Likes | null> {
1778.     return this.likesRepository.findOne({
1779.         where: {
1780.             user_id,
1781.             page_id,
1782.         },
1783.     });
1784. }
1785.
1786. async checkRightConnectToPage(
1787.     user_id: number,
1788.     page_id: number,
1789.     mode: string,
1790. ): Promise<boolean> {
1791.     const membership = await this.workspaceMembersRepository.findOne({
1792.         where: {
1793.             user_id,
1794.             page_id,
1795.         },
1796.         attributes: ['role'],
1797.     });
1798.     if (!membership) {

```



```

1799.         return false;
1800.     }
1801.     switch (mode) {
1802.         case 'ws-connect':
1803.             return ['owner', 'editor'].includes(membership.role.toLowerCase());
1804.         case 'owner':
1805.             return ['owner'].includes(membership.role.toLowerCase());
1806.         case 'comment':
1807.             return ['owner', 'editor', 'comment'].includes(
1808.                 membership.role.toLowerCase(),
1809.             );
1810.         case 'all':
1811.             return ['owner', 'editor', 'comment', 'view'].includes(
1812.                 membership.role.toLowerCase(),
1813.             );
1814.         default:
1815.             throw new Error('Unknown mode');
1816.     }
1817. }
1818.
1819. async getUserPagesByUserId(user_id: number): Promise<Pages[]> {
1820.     return this.pageRepository.findAll({
1821.         include: [
1822.             {
1823.                 model: User,
1824.                 as: 'users',
1825.                 where: { User_id: user_id },
1826.                 through: { attributes: [] },
1827.                 required: true,
1828.             },
1829.         ],
1830.     });
1831. }
1832.
1833. async createWorkspace(user_id: number): Promise<Workspace> {
1834.     return this.workspaceRepository.create({
1835.         name: 'MyWorkspace',
1836.         user_limit: 5,
1837.         user_id: user_id,
1838.     });
1839. }
1840.
1841. async createPage(
1842.     parent_page_id: number | null,
1843.     workspace_id: number,
1844. ): Promise<Pages> {
1845.     return this.pageRepository.create({
1846.         title: 'NewPage',
1847.         parent_page_id: parent_page_id,
1848.         type: 'Work',
1849.         description: '',
1850.         picture_url: '',
1851.         avatar_url: '',
1852.         is_public: false,
1853.         workspace_id: workspace_id,
1854.     });
1855. }
1856.
1857. async updatePage(
1858.     dto: UpdatePageDto,
1859.     avatar_url: string | undefined,
1860.     picture_url: string | undefined,
1861.     page: Pages,
1862. ) {
1863.     const { title, description, type, category, parent_page_id } = dto;
1864.     if (page) {
1865.         return page.update({
1866.             title,
1867.             description,
1868.             type,
1869.             category,
1870.             avatar_url,
1871.             parent_page_id,
1872.             picture_url,
1873.             is_public: type === 'Subscription',
1874.         });
1875.     } else {
1876.         throw new Error(ErrorLog.PAGE_FAILURE);
1877.     }
1878. }

```

```

1879.
1880.   async deletePage(page_id: number, force: boolean) {
1881.       return this.pageRepository.destroy({
1882.           where: {
1883.               Page_id: page_id,
1884.           },
1885.           force: force,
1886.       });
1887.   }
1888.
1889.   async createComment(content: string, block_id: number, user_id: number) {
1890.       return this.commentsRepository.create({
1891.           content: content,
1892.           block_id: block_id,
1893.           user_id: user_id,
1894.       });
1895.   }
1896.
1897.   async deleteComment(comment_id: number) {
1898.       return this.commentsRepository.destroy({
1899.           where: {
1900.               Comment_id: comment_id,
1901.           },
1902.       });
1903.   }
1904.
1905.   async createWorkspaceMember(
1906.       dto: CreateWorkspaceMemberDto,
1907.   ): Promise<WorkspaceMembers> {
1908.       return this.workspaceMembersRepository.create({
1909.           user_id: dto.user_id,
1910.           page_id: dto.page_id,
1911.           role: dto.role,
1912.       });
1913.   }
1914.
1915.   async getUsersByPageId(
1916.       page_id: number,
1917.   ): Promise<{ User_id: number; email: string; role: string }[]> {
1918.       const members = await this.workspaceMembersRepository.findAll({
1919.           where: { page_id },
1920.           include: [
1921.               {
1922.                   model: User,
1923.                   attributes: ['User_id', 'email'],
1924.               },
1925.           ],
1926.       });
1927.
1928.       return members.map((member) => ({
1929.           User_id: member.user.User_id,
1930.           email: member.user.email,
1931.           role: member.role,
1932.       }));
1933.   }
1934.
1935.   async deleteWorkspaceMember(user_id: number, page_id: number) {
1936.       return this.workspaceMembersRepository.destroy({
1937.           where: {
1938.               user_id: user_id,
1939.               page_id: page_id,
1940.           },
1941.       });
1942.   }
1943.
1944.   async updateWorkspaceMember(
1945.       user_id: number,
1946.       page_id: number,
1947.       newRole: string,
1948.   ) {
1949.       return this.workspaceMembersRepository.update(
1950.           { role: newRole },
1951.           {
1952.               where: {
1953.                   user_id,
1954.                   page_id,
1955.               },
1956.           },
1957.       );
1958.   }

```

```

1959.
1960. async createSubscription(dto: SubscriptionDto, user_id: number) {
1961.     const access_end_date = new Date();
1962.     access_end_date.setDate(access_end_date.getDate() + 365);
1963.     return this.subscriptionsRepository.create({
1964.         user_id: user_id,
1965.         page_id: dto.page_id,
1966.         status: true,
1967.         access_end_date: access_end_date,
1968.     });
1969. }
1970.
1971. async deleteSubscription(dto: SubscriptionDto, user_id: number) {
1972.     return this.subscriptionsRepository.destroy({
1973.         where: {
1974.             user_id: user_id,
1975.             page_id: dto.page_id,
1976.         },
1977.     });
1978. }
1979.
1980. async createLike(dto: LikeDto, user_id: number) {
1981.     if (dto.stars === undefined || dto.stars === null) {
1982.         dto.stars = 0;
1983.     }
1984.     return this.likesRepository.create({
1985.         user_id: user_id,
1986.         page_id: dto.page_id,
1987.         stars: dto.stars,
1988.     });
1989. }
1990.
1991. async deleteLike(dto: LikeDto, user_id: number) {
1992.     return this.likesRepository.destroy({
1993.         where: {
1994.             user_id: user_id,
1995.             page_id: dto.page_id,
1996.         },
1997.     });
1998. }
1999.
2000. async createBlock(
2001.     dto: CreateBlockDto,
2002.     transaction?: Transaction,
2003. ): Promise<Blocks> {
2004.     return this.blocksRepository.create(
2005.         {
2006.             type: dto.type,
2007.             pointer_to: dto.pointer_to,
2008.             is_element: dto.is_element,
2009.             sub_pointer_to: dto.sub_pointer_to,
2010.             database_y: dto.database_y,
2011.             database_x: dto.database_x,
2012.             page_id: dto.page_id,
2013.             content: dto.content || {},
2014.         },
2015.         { transaction },
2016.     );
2017. }
2018.
2019. async createInvitation(dto: InviteUsersDto, token: string, user_id: number) {
2020.     const tomorrow = new Date();
2021.     tomorrow.setDate(tomorrow.getDate() + 1);
2022.     return this.invitationsRepository.create({
2023.         invite_email: dto.invite_email,
2024.         role: dto.role,
2025.         status: false,
2026.         token: token,
2027.         expires_at: tomorrow,
2028.         page_id: dto.page_id,
2029.         invited_by: user_id,
2030.     });
2031. }
2032.
2033. async createHistory(
2034.     action: string,
2035.     changes: Record<string, any> | undefined,
2036.     page_id: number,
2037.     block_id: number,
2038.     transaction?: Transaction,

```

```

2039.   ) {
2040.       return this.changeHistoryRepository.create(
2041.           {
2042.               action: action,
2043.               changes: changes || {},
2044.               block_id: block_id,
2045.               page_id: page_id,
2046.           },
2047.           { transaction },
2048.       );
2049.   }
2050.
2051.   async getBlockByPageId(page_id: number): Promise<{
2052.       sortedBlocks: Blocks[];
2053.       elements: Blocks[];
2054.   }> {
2055.       const allBlocks = await this.blocksRepository.findAll({
2056.           where: { page_id },
2057.       });
2058.       const rootBlock = allBlocks.find(
2059.           (block) => block.pointer_to === null && block.is_element === false,
2060.       );
2061.
2062.       if (!rootBlock) return { sortedBlocks: [], elements: [] };
2063.
2064.       const sortedBlocks: Blocks[] = [rootBlock];
2065.       let currentBlock = rootBlock;
2066.
2067.       while (true) {
2068.           const nextBlock = allBlocks.find(
2069.               (block) => block.pointer_to === currentBlock.Block_id,
2070.           );
2071.           if (!nextBlock) break;
2072.
2073.           sortedBlocks.push(nextBlock);
2074.           currentBlock = nextBlock;
2075.       }
2076.
2077.       const elements = allBlocks.filter((block) => block.is_element === true);
2078.
2079.       return { sortedBlocks, elements };
2080.   }
2081.
2082.   async updateBlocks(updateOps: Operation<UpdateOperationArgs>[]): Promise<{
2083.       updatedBlocks: Blocks[];
2084.       newColumns: Blocks[];
2085.       updatedElements: Blocks[];
2086.   }> {
2087.       try {
2088.           return await this.sequelize.transaction(async (transaction) => {
2089.               const transactionHost = { transaction };
2090.
2091.               const updatedBlocks: Blocks[] = [];
2092.               const updatedElements: Blocks[] = [];
2093.               const newColumns: Blocks[] = [];
2094.
2095.               for (const op of updateOps) {
2096.                   const { block_id, content, type } = op.args;
2097.
2098.                   const block = await this.blocksRepository.findByPk(
2099.                       block_id,
2100.                       transactionHost,
2101.                   );
2102.                   if (!block) {
2103.                       throw new Error(ErrorLog.BLOCKS_FAILURE);
2104.                   }
2105.                   if (block.type !== type) {
2106.                       throw new Error(ErrorLog.BLOCKS_TYPE);
2107.                   }
2108.                   if (block.type === 'database') {
2109.                       if (!this.isDatabaseContent(content)) {
2110.                           throw new Error('Invalid database content format');
2111.                       }
2112.                       const { layout_type } = content;
2113.                       console.log(layout_type);
2114.                       console.log(block.content.layout_type);
2115.                       if (block.content.layout_type !== layout_type) {
2116.                           switch (layout_type) {
2117.                               case 'Graph': {
2118.                                   const xCoords = await this.findDatabaseX(block.Block_id);

```

```

2119.         if (!(xCoords.size >= 2)) {
2120.             const newColumn = await this.createColumnOfDb(
2121.                 block,
2122.                 'text-property',
2123.             );
2124.             newColumns.push(...newColumn);
2125.         }
2126.         break;
2127.     }
2128.     case 'Donut': {
2129.         const xCoords = await this.findDatabaseX(block.Block_id);
2130.         if (!(xCoords.size >= 1)) {
2131.             const newColumn = await this.createColumnOfDb(
2132.                 block,
2133.                 'text-property',
2134.             );
2135.             newColumns.push(...newColumn);
2136.         }
2137.         break;
2138.     }
2139.     case 'Board': {
2140.         console.log('point');
2141.         const hasStatusCol = await this.hasProperty(
2142.             block,
2143.             'status-property',
2144.         );
2145.         if (!hasStatusCol) {
2146.             const newColumn = await this.createColumnOfDb(
2147.                 block,
2148.                 'status-property',
2149.             );
2150.             newColumns.push(...newColumn);
2151.         }
2152.         break;
2153.     }
2154.     case 'Default':
2155.         break;
2156.     default:
2157.         break;
2158. }
2159. }
2160. }
2161.
2162.     const updatedBlock = await block.update({ content }, transactionHost);
2163.     await this.createHistory(
2164.         'update',
2165.         content,
2166.         updatedBlock.page_id,
2167.         updatedBlock.Block_id,
2168.         transactionHost.transaction,
2169.     );
2170.     if (!updatedBlock.is_element) {
2171.         updatedBlocks.push(updatedBlock);
2172.     } else {
2173.         updatedElements.push(updatedBlock);
2174.     }
2175. }
2176.
2177.     return { updatedBlocks, newColumns, updatedElements };
2178. });
2179. } catch (error) {
2180.     if (error instanceof Error) {
2181.         throw new Error(`UpdateBlocks Error: ${error.message}`);
2182.     }
2183.     throw new Error(ErrorLog.BLOCKS_UPD);
2184. }
2185. }
2186.
2187. isDatabaseContent(content: unknown): content is DataBaseComponentDto {
2188.     return (
2189.         typeof content === 'object' &&
2190.         content !== null &&
2191.         'layout_type' in content &&
2192.         typeof (content as Record<string, unknown>).layout_type === 'string'
2193.     );
2194. }
2195.
2196. async hasProperty(dbBlock: Blocks, property: string) {
2197.     const relatedBlocks = await this.blocksRepository.findAll({
2198.         where: {

```

```

2199.         sub_pointer_to: dbBlock.Block_id,
2200.         database_x: 0,
2201.     },
2202.     attributes: ['type'],
2203.     raw: true,
2204. });
2205. for (const block of relatedBlocks) {
2206.     if (block.type === property) {
2207.         return true;
2208.     }
2209. }
2210. return false;
2211. }
2212.
2213. async createBlocks(
2214.     createOps: Operation<CreateOperationArgs>[],
2215.     page_id: number,
2216. ): Promise<{ createdBlocks: Blocks[]; newColumns: Blocks[] }> {
2217.     try {
2218.         return await this.sequelize.transaction(async (transaction) => {
2219.             const createdBlocks: Blocks[] = [];
2220.             const newColumns: Blocks[] = [];
2221.             for (const op of createOps) {
2222.                 const createDto: CreateBlockDto = {
2223.                     type: op.args.type,
2224.                     pointer_to: op.args.pointer_to,
2225.                     is_element: false,
2226.                     sub_pointer_to: null,
2227.                     database_y: null,
2228.                     database_x: null,
2229.                     page_id: page_id,
2230.                     content:
2231.                         op.args.type === 'database'
2232.                         ? { layout_type: 'Default', args: { name: 'NewTable' } }
2233.                         : op.args.content || {},
2234.                 };
2235.                 const newBlock = await this.createBlock(createDto, transaction);
2236.                 await this.createHistory(
2237.                     'create',
2238.                     createDto.content,
2239.                     page_id,
2240.                     newBlock.Block_id,
2241.                     transaction,
2242.                 );
2243.                 if (newBlock.type === 'database') {
2244.                     const newColumn = await this.createColumnOfDb(
2245.                         newBlock,
2246.                         'text-property',
2247.                     );
2248.                     newColumns.push(...newColumn);
2249.                 }
2250.                 createdBlocks.push(newBlock);
2251.             }
2252.
2253.             return { createdBlocks, newColumns };
2254.         });
2255.     } catch (error) {
2256.         if (error instanceof Error) {
2257.             throw new Error(`createBlocks Error: ${error.message}`);
2258.         }
2259.         throw new Error(ErrorLog.BLOCKS_UPD);
2260.     }
2261. }
2262.
2263. async createColumnOfDb(dbBlock: Blocks, typeProperty: string) {
2264.     try {
2265.         return await this.sequelize.transaction(async (transaction) => {
2266.             const newColumn: Blocks[] = [];
2267.             const xCoords = await this.findDatabaseX(dbBlock.Block_id);
2268.
2269.             let X: number;
2270.             if (xCoords.size === 0) {
2271.                 X = 0;
2272.             } else {
2273.                 X = Math.max(...xCoords) + 1;
2274.             }
2275.
2276.             const yCoords = await this.findDatabaseY(dbBlock.Block_id);
2277.             if (yCoords.size === 0) {
2278.                 yCoords.add(0);

```

```

2279.         yCoords.add(1);
2280.     }
2281.     const type = this.decodeTypeProperty(typeProperty);
2282.     for (const y of yCoords) {
2283.         const createColumnDto: CreateBlockDto = {
2284.             type: y === 0 ? typeProperty : type,
2285.             pointer_to: null,
2286.             is_element: true,
2287.             sub_pointer_to: dbBlock.Block_id,
2288.             database_x: X,
2289.             database_y: y,
2290.             page_id: dbBlock.page_id,
2291.             content: {},
2292.         };
2293.         const newBlock = await this.createBlock(createColumnDto, transaction);
2294.         await this.createHistory(
2295.             'create',
2296.             createColumnDto.content,
2297.             createColumnDto.page_id,
2298.             newBlock.Block_id,
2299.             transaction,
2300.         );
2301.         newColumn.push(newBlock);
2302.     }
2303.     return newColumn;
2304. });
2305. } catch (error) {
2306.     if (error instanceof Error) {
2307.         throw new Error(`createColumnOfDb Error: ${error.message}`);
2308.     }
2309.     throw new Error(ErrorLog.DB_COLUMN);
2310. }
2311. }
2312.
2313. async createRowOfDb(dbBlock: Blocks) {
2314.     try {
2315.         return await this.sequelize.transaction(async (transaction) => {
2316.             const newRow: Blocks[] = [];
2317.             const xCoords = await this.findDatabaseX(dbBlock.Block_id);
2318.             if (xCoords.size === 0) {
2319.                 throw new Error(ErrorLog.DB_ROW);
2320.             }
2321.             const yCoords = await this.findDatabaseY(dbBlock.Block_id);
2322.             let Y: number;
2323.             if (yCoords.size === 0) {
2324.                 Y = 1;
2325.             } else {
2326.                 Y = Math.max(...yCoords) + 1;
2327.             }
2328.             for (const x of xCoords) {
2329.                 const propertyBlock = await this.blocksRepository.findOne({
2330.                     where: { database_x: x, database_y: 0 },
2331.                     transaction,
2332.                 });
2333.                 if (!propertyBlock) {
2334.                     throw new Error(ErrorLog.DB_ROW);
2335.                 }
2336.                 const type = this.decodeTypeProperty(propertyBlock.type);
2337.                 const createRowDto: CreateBlockDto = {
2338.                     type: type,
2339.                     pointer_to: null,
2340.                     is_element: true,
2341.                     sub_pointer_to: dbBlock.Block_id,
2342.                     database_x: x,
2343.                     database_y: Y,
2344.                     page_id: dbBlock.page_id,
2345.                     content: {},
2346.                 };
2347.                 const newBlock = await this.createBlock(createRowDto, transaction);
2348.                 await this.createHistory(
2349.                     'create',
2350.                     createRowDto.content,
2351.                     createRowDto.page_id,
2352.                     newBlock.Block_id,
2353.                     transaction,
2354.                 );
2355.                 newRow.push(newBlock);
2356.             }
2357.             return newRow;
2358.         });

```

```

2359.     } catch (error) {
2360.         if (error instanceof Error) {
2361.             throw new Error(`createRowOfDb Error: ${error.message}`);
2362.         }
2363.         throw new Error(ErrorLog.DB_ROW);
2364.     }
2365. }
2366.
2367. async findDatabaseX(subPointerTo: number): Promise<Set<number>> {
2368.     const blocks = await this.blocksRepository.findAll({
2369.         where: {
2370.             is_element: true,
2371.             sub_pointer_to: subPointerTo,
2372.         },
2373.         attributes: ['database_x'],
2374.     });
2375.
2376.     return new Set(
2377.         blocks
2378.             .map(({ database_x }) => database_x)
2379.             .filter((x): x is number => x !== null),
2380.     );
2381. }
2382.
2383. async findDatabaseY(subPointerTo: number): Promise<Set<number>> {
2384.     const blocks = await this.blocksRepository.findAll({
2385.         where: {
2386.             is_element: true,
2387.             sub_pointer_to: subPointerTo,
2388.         },
2389.         attributes: ['database_y'],
2390.     });
2391.
2392.     return new Set(
2393.         blocks
2394.             .map(({ database_y }) => database_y)
2395.             .filter((y): y is number => y !== null),
2396.     );
2397. }
2398.
2399. decodeTypeProperty(typeProperty: string) {
2400.     switch (typeProperty) {
2401.         case 'text-property':
2402.             return 'text';
2403.         case 'number-property':
2404.             return 'number';
2405.         case 'status-property':
2406.             return 'status';
2407.         case 'formula-property':
2408.             return 'formula';
2409.         case 'file-property':
2410.             return 'file';
2411.         default:
2412.             throw new Error(ErrorLog.TYPE_PROP);
2413.     }
2414. }
2415.
2416. async changePointers(
2417.     changes: { block_id: number; pointer_to: number | null }[],
2418. ): Promise<Blocks[]> {
2419.     try {
2420.         return await this.sequelize.transaction(async (transaction) => {
2421.             const transactionHost = { transaction };
2422.             const changedPosBlocks: Blocks[] = [];
2423.             for (const change of changes) {
2424.                 const { block_id, pointer_to } = change;
2425.
2426.                 const block = await this.blocksRepository.findByPk(
2427.                     block_id,
2428.                     transactionHost,
2429.                 );
2430.                 if (!block) {
2431.                     throw new Error(ErrorLog.BLOCKS_FAILURE);
2432.                 }
2433.
2434.                 const changedPosBlock = await block.update(
2435.                     { pointer_to },
2436.                     transactionHost,
2437.                 );
2438.

```



```

2439.         changedPosBlocks.push(changedPosBlock);
2440.     }
2441.
2442.     return changedPosBlocks;
2443. });
2444. } catch (error) {
2445.     if (error instanceof Error) {
2446.         throw new Error(`createBlocks Error: ${error.message}`);
2447.     }
2448.     throw new Error(ErrorLog.BLOCKS_CNG);
2449. }
2450. }
2451.
2452. async deleteBlocks(deleteBlockIds: Set<number>, page_id: number) {
2453.     try {
2454.         return await this.sequelize.transaction(async (transaction) => {
2455.             const idsToDelete = Array.from(deleteBlockIds);
2456.             const elementIdsDel: Set<number> = new Set();
2457.
2458.             const databaseBlocks = await this.blocksRepository.findAll({
2459.                 where: {
2460.                     Block_id: idsToDelete,
2461.                     type: 'database',
2462.                 },
2463.                 attributes: ['Block_id'],
2464.                 transaction,
2465.             });
2466.
2467.             if (databaseBlocks.length > 0) {
2468.                 for (const databaseBlock of databaseBlocks) {
2469.                     const databaseElements = await this.blocksRepository.findAll({
2470.                         where: {
2471.                             sub_pointer_to: databaseBlock.Block_id,
2472.                             is_element: true,
2473.                         },
2474.                         attributes: ['Block_id'],
2475.                         transaction,
2476.                     });
2477.                     databaseElements.forEach((el) => elementIdsDel.add(el.Block_id));
2478.                 }
2479.
2480.                 const databaseBlockIds = databaseBlocks.map(
2481.                     (block) => block.Block_id,
2482.                 );
2483.
2484.                 await this.blocksRepository.destroy({
2485.                     where: {
2486.                         sub_pointer_to: databaseBlockIds,
2487.                     },
2488.                     transaction,
2489.                 });
2490.
2491.                 for (const elementIdDel of elementIdsDel) {
2492.                     await this.createHistory(
2493.                         'delete',
2494.                         {},
2495.                         page_id,
2496.                         elementIdDel,
2497.                         transaction,
2498.                     );
2499.                 }
2500.             }
2501.
2502.             const deletedCount = await this.blocksRepository.destroy({
2503.                 where: {
2504.                     Block_id: idsToDelete,
2505.                     is_element: false,
2506.                 },
2507.                 transaction,
2508.             });
2509.
2510.             for (const idToDelete of idsToDelete) {
2511.                 await this.createHistory(
2512.                     'delete',
2513.                     {},
2514.                     page_id,
2515.                     idToDelete,
2516.                     transaction,
2517.                 );
2518.             }

```

```

2519.
2520.         if (deletedCount !== deleteBlockIds.size) {
2521.             throw new Error(ErrorLog.BLOCKS_DEL_COUNT);
2522.         }
2523.         return elementIdsDel;
2524.     });
2525. } catch (error) {
2526.     if (error instanceof Error) {
2527.         throw new Error(`createBlocks Error: ${error.message}`);
2528.     }
2529.     throw new Error(ErrorLog.BLOCKS_DEL);
2530. }
2531. }
2532.
2533. async findBlockById(block_id: number): Promise<Blocks | null> {
2534.     return this.blocksRepository.findOne({
2535.         where: { Block_id: block_id },
2536.     });
2537. }
2538.
2539. async createInDb(
2540.     rowOps: Operation<CreateInDbOperationArgs>[],
2541.     columnOps: Operation<CreateInDbOperationArgs>[],
2542.     listElements: Set<number>,
2543. ): Promise<{
2544.     createdElements: Blocks[];
2545.     updatedListElements: Set<number>;
2546. }> {
2547.     const createdElements: Blocks[] = [];
2548.     const updatedListElements = new Set(listElements);
2549.     if (columnOps.length > 0) {
2550.         for (const op of columnOps) {
2551.             const db_block = await this.findBlockById(op.args.block_db_id);
2552.             if (db_block && op.args.property_type && db_block.type === 'database') {
2553.                 const createdColumns = await this.createColumnOfDb(
2554.                     db_block,
2555.                     op.args.property_type,
2556.                 );
2557.                 createdElements.push(...createdColumns);
2558.                 createdColumns.forEach((block) =>
2559.                     updatedListElements.add(block.Block_id),
2560.                 );
2561.             }
2562.         }
2563.     }
2564.
2565.     if (rowOps.length > 0) {
2566.         for (const op of rowOps) {
2567.             const db_block = await this.findBlockById(op.args.block_db_id);
2568.             if (db_block && db_block.type === 'database') {
2569.                 const createdRows = await this.createRowOfDb(db_block);
2570.                 createdElements.push(...createdRows);
2571.                 createdRows.forEach((block) =>
2572.                     updatedListElements.add(block.Block_id),
2573.                 );
2574.             }
2575.         }
2576.     }
2577.     return {
2578.         createdElements,
2579.         updatedListElements,
2580.     };
2581. }
2582.
2583. async deleteColumnInDb(db_block: Blocks, column: number): Promise<number[]> {
2584.     const xCoords = await this.findDatabaseX(db_block.Block_id);
2585.     if (xCoords.size === 0 || !xCoords.has(column)) {
2586.         throw new Error(ErrorLog.COLUMN_DEL);
2587.     }
2588.     const blocksToDelete = await this.blocksRepository.findAll({
2589.         where: {
2590.             sub_pointer_to: db_block.Block_id,
2591.             database_x: column,
2592.         },
2593.         attributes: ['Block_id'],
2594.     });
2595.     const deletedIds = blocksToDelete.map((block) => block.Block_id);
2596.
2597.     await this.blocksRepository.destroy({
2598.         where: {

```

```

2599.         sub_pointer_to: db_block.Block_id,
2600.         database_x: column,
2601.     },
2602. });
2603. for (const deletedId of deletedIds) {
2604.     await this.createHistory('delete', {}, db_block.page_id, deletedId);
2605. }
2606. return deletedIds;
2607. }
2608.
2609. async deleteRowInDb(db_block: Blocks, row: number): Promise<number[]> {
2610.     const yCoords = await this.findDatabaseY(db_block.Block_id);
2611.     if (yCoords.size === 1 || !yCoords.has(row)) {
2612.         throw new Error(ErrorLog.ROW_DEL);
2613.     }
2614.     const blocksToDelete = await this.blocksRepository.findAll({
2615.         where: {
2616.             sub_pointer_to: db_block.Block_id,
2617.             database_y: row,
2618.         },
2619.         attributes: ['Block_id'],
2620.     });
2621.     const deletedIds = blocksToDelete.map((block) => block.Block_id);
2622.
2623.     await this.blocksRepository.destroy({
2624.         where: {
2625.             sub_pointer_to: db_block.Block_id,
2626.             database_y: row,
2627.         },
2628.     });
2629.     for (const deletedId of deletedIds) {
2630.         await this.createHistory('delete', {}, db_block.page_id, deletedId);
2631.     }
2632.     return deletedIds;
2633. }
2634.
2635. async deleteInDb(
2636.     delRowOps: Operation<DeleteInDbOperationArgs>[],
2637.     delColumnOps: Operation<DeleteInDbOperationArgs>[],
2638. ) {
2639.     const delIds = new Set<number>();
2640.     if (delColumnOps.length > 0) {
2641.         for (const op of delColumnOps) {
2642.             const db_block = await this.findBlockById(op.args.block_db_id);
2643.             if (db_block && op.args.column && db_block.type === 'database') {
2644.                 const deletedIds = await this.deleteColumnInDb(
2645.                     db_block,
2646.                     op.args.column,
2647.                 );
2648.                 deletedIds.forEach((id) => delIds.add(id));
2649.             }
2650.         }
2651.     }
2652.     if (delRowOps.length > 0) {
2653.         for (const op of delRowOps) {
2654.             const db_block = await this.findBlockById(op.args.block_db_id);
2655.             if (db_block && op.args.row && db_block.type === 'database') {
2656.                 const deletedIds = await this.deleteRowInDb(db_block, op.args.row);
2657.                 deletedIds.forEach((id) => delIds.add(id));
2658.             }
2659.         }
2660.     }
2661.
2662.     return delIds;
2663. }
2664. }

```

ПРИЛОЖЕНИЕ Б
(обязательное)

Спецификация

ПРИЛОЖЕНИЕ В
(обязательное)

Ведомость документов