СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 4](#_Toc135597887)

[1. Анализ предметной области 6](#_Toc135597888)

[1.1. Анализ аналогов 6](#_Toc135597889)

[1.2. Выдвинутые требования 6](#_Toc135597890)

[2. Инструменты разработки 8](#_Toc135597891)

[3. Программная реализация 11](#_Toc135597892)

[3.1. Архитектура приложения 11](#_Toc135597893)

[3.2. Серверная часть 12](#_Toc135597894)

[3.2.1. Структура проекта 12](#_Toc135597895)

[3.2.2. Классы 13](#_Toc135597896)

[3.3. Клиентская часть 15](#_Toc135597897)

[3.3.1. Структура проекта 15](#_Toc135597898)

[3.4. База данных 16](#_Toc135597899)

[3.5. Авторизация 17](#_Toc135597900)

[3.6. Сервер 24](#_Toc135597901)

[3.6.1. Получение файлов 24](#_Toc135597902)

[3.6.2. Создание директории 25](#_Toc135597903)

[3.6.3. Загрузка файла и проверка файла 27](#_Toc135597904)

[3.6.4. Скачивание файла 28](#_Toc135597905)

[3.6.5. Удаление файла и рекурсивное удаление директорий 29](#_Toc135597906)

[3.6.6. Поиск файлов 30](#_Toc135597907)

[3.6.7. Класс Ошибок 32](#_Toc135597908)

[3.6.8. Middleware 34](#_Toc135597909)

[3.7. Клиент 35](#_Toc135597910)

[3.7.1. Перехватчики 36](#_Toc135597911)

[3.7.2. Получение файлов 38](#_Toc135597912)

[3.7.3. Поиск файлов 39](#_Toc135597913)

[3.7.4. Удаление файлов 39](#_Toc135597914)

[3.7.5. Создание директории 39](#_Toc135597915)

[3.7.6. Загрузка файлов 41](#_Toc135597916)

[4. Тестирование приложения 42](#_Toc135597917)

[ЗАКЛЮЧЕНИЕ 43](#_Toc135597918)

ВВЕДЕНИЕ

В современном информационном обществе, где доступ к информации и электронным ресурсам играет ключевую роль, важно иметь эффективные инструменты для хранения и обмена файлами. С развитием интернет-технологий и все большим использованием облачных сервисов, вопрос о создании удобных и безопасных платформ для хранения и предоставления доступа к файлам становится все более актуальным.

В настоящее время многие люди и организации используют облачные сервисы для хранения своих данных. Однако существующие решения не всегда удовлетворяют требованиям пользователей в отношении безопасности, надежности, скорости и функциональности.

Разработка собственного web-приложения для хранения файлов и предоставления доступа к ним имеет практическую значимость, так как позволяет создать индивидуальное решение, учитывающее особенности и потребности пользователей или организаций. Реализация указанных задач позволит создать удобное и надежное web-приложение, которое будет эффективным инструментом для хранения файлов и предоставления доступа к ним. Полученные результаты могут быть применены в различных сферах, включая личное использование пользователей, работу с данными в организациях или сотрудничество над проектами.

Целью данной курсовой работы является разработка web-приложения, аналогичного Google Диску, которое позволит пользователям хранить свои файлы в облачном хранилище и обеспечит им удобный доступ к этим файлам через интернет.

В рамках данной работы будут решены следующие задачи:

1. изучение существующих облачных хранилищ и анализ их функциональности, интерфейсов и особенностей;
2. сбор требований от потенциальных пользователей для определения основных функций и возможностей облачного хранилища;
3. проектирование архитектуры облачного хранилища, включая выбор подходящих технологий и протоколов;
4. разработка пользовательского интерфейса для работы с файлами и папками в облачном хранилище, включая синхронизацию и обмен файлами между устройствами, а также обеспечение безопасности данных и аутентификацию пользователей.
5. проведение тестирования и отладки разработанного облачного хранилища для проверки его функциональности и надежности;
6. оценка эффективности разработанного облачного хранилища на основе проведенных тестов и сравнение с существующими решениями.

# Анализ предметной области

В данном разделе произведен анализ предметной области, связанной с разработкой веб-приложения для хранения файлов и предоставления доступа к ним, схожего с Google Disk. Были рассмотрены следующие аналоги: Dropbox, Yandex.Disk и Google Disk. Анализ аналогов помог определить преимущества и недостатки существующих решений и выделить ключевые темы, которые необходимо рассмотреть при разработке предлагаемого веб-приложения.

## Анализ аналогов

Dropbox является одним из популярных облачных хранилищ, позволяющим пользователям сохранять и синхронизировать файлы на различных устройствах. Он обладает простым интерфейсом, хорошей скоростью загрузки и удобными возможностями совместной работы над файлами. Однако, основным недостатком Dropbox является ограничение на бесплатное хранилище и отсутствие возможности полностью контролировать свои данные.

Yandex.Disk предоставляет функционал схожий с Dropbox, но с акцентом на российскую аудиторию. Он обладает щедрым объемом бесплатного хранилища и интегрируется с другими сервисами Яндекса. Однако, с точки зрения функционала и возможностей совместной работы, Yandex.Disk несколько уступает Dropbox и Google Disk.

Google Disk (Google Drive) является одним из наиболее популярных облачных хранилищ. Он предлагает большой объем бесплатного хранилища, интеграцию с другими сервисами Google, а также широкие возможности совместной работы над файлами. Google Disk обладает удобным интерфейсом, автоматической синхронизацией файлов и доступом к ним с любого устройства. Однако, важно отметить, что Google Disk является коммерческим сервисом и может ограничивать контроль пользователя над его данными.

## Выдвинутые требования

На основе анализа данных выдвинуты следующие требования:

* Безопасность и конфиденциальность данных: одной из ключевых тем, вытекающих из анализа аналогов, является безопасность и конфиденциальность данных. Важно применять механизмы шифрования и защиты данных пользователей, а также предусмотреть меры по предотвращению несанкционированного доступа;
* Интерфейс и удобство использования:

Другой важной темой является разработка удобного и интуитивно понятного пользовательского интерфейса. Необходимо обеспечить простоту и удобство взаимодействия с веб-приложением, чтобы пользователи могли легко загружать, организовывать и совместно работать с файлами;

* Масштабируемость и производительность:

Также важно учесть масштабируемость и производительность веб-приложения. Предполагается, что количество пользователей и объем хранимых файлов могут значительно возрасти, поэтому необходимо разработать архитектуру, способную эффективно масштабироваться и обеспечивать высокую производительность;

# Инструменты разработки

В данном проекте использовались различные технологии, специально подобранные для обеспечения эффективной работы как на серверной, так и на клиентской стороне. Каждая из этих технологий играет важную роль в создании современных веб-приложений, обеспечивая базовую безопасность, производительность, удобство разработки и множество других возможностей.

Библиотеки, использованные в серверной части приложения:

* Archiver представляет собой библиотеку для упаковки и архивирования данных на сервере. Она позволяет создавать и распаковывать архивы различных форматов, таких как zip, tar и другие;
* Bcrypt — это криптографический алгоритм хеширования паролей. Он обеспечивает безопасное хранение и проверку паролей на сервере путем хеширования и сравнения хэшей паролей. Bcrypt широко применяется для защиты пользовательских паролей;
* Dotenv — это модуль, который позволяет загружать переменные окружения из файла .env на сервере. Файл .env содержит конфигурационные переменные, такие как секретные ключи, адреса баз данных и другие настройки, которые могут изменяться в разных средах разработки;
* Express — это минималистичный и гибкий фреймворк для создания веб-приложений на сервере с использованием языка JavaScript. Он предоставляет набор инструментов и маршрутизацию, чтобы упростить разработку серверной части приложения;
* Jsonwebtoken — это библиотека, которая позволяет создавать и проверять JSON Web Tokens (JWT) на сервере. JWT — это формат для представления утверждений между двумя сторонами в виде JSON-объекта. Они широко используются для аутентификации и обмена данными между клиентом и сервером;
* UUID (Универсальный уникальный идентификатор) — это стандарт используемый для генерации уникальных идентификаторов. UUID представляет собой строку символов, которая обладает крайне малой вероятностью коллизии (дублирования). Он может быть полезен, например, при генерации уникальных идентификаторов для объектов в базе данных;
* Winston — это модуль для регистрации событий и журналирования на сервере. Он предоставляет гибкий и настраиваемый механизм регистрации сообщений различных уровней, таких как отладка, информация, предупреждения и ошибки. Winston позволяет сохранять журналы в различных форматах и местах, таких как файлы или базы данных;
* Mongoose — это библиотека для работы с MongoDB, которая облегчает взаимодействие с базой данных MongoDB на сервере. Она предоставляет схемы и модели для определения структуры данных и выполнения запросов к базе данных. Mongoose упрощает работу с MongoDB и предлагает много полезных функций, таких как валидация данных и middleware;
* Nodemon — это инструмент разработки, который облегчает процесс разработки на сервере. Он автоматически перезапускает сервер при изменении файлов, что позволяет сразу видеть результаты внесенных изменений без ручного перезапуска сервера.

Технологии на клиенте:

* TypeScript — это язык программирования, который является надмножеством JavaScript. Он добавляет статическую типизацию к JavaScript, позволяя выявлять и предотвращать ошибки на этапе разработки. TypeScript повышает надежность и читаемость кода на клиентской стороне приложения;
* Redux Toolkit — это официальная библиотека для упрощения работы с управлением состоянием в приложениях, использующих Redux. Она предлагает удобные средства для определения и обновления состояния приложения, упрощает создание Redux-хранилища и управление побочными эффектами;
* React — это JavaScript-библиотека для разработки пользовательского интерфейса. Она позволяет создавать компоненты, которые являются независимыми и многократно используемыми блоками кода, отвечающими за отображение данных на веб-странице. React использует виртуальный DOM для эффективного обновления пользовательского интерфейса;
* Axios — это библиотека для выполнения HTTP-запросов на клиентской стороне. Она обеспечивает простой и удобный интерфейс для отправки запросов на сервер и обработки ответов. Axios поддерживает множество функций, таких как установка заголовков, обработка ошибок и прогресс загрузки.

# Программная реализация

## Архитектура приложения

Архитектура веб-приложений является критической составляющей успеха любого проекта. Она обеспечивает основу для организации, структурирования и управления компонентами приложения, позволяя разработчикам создавать масштабируемое, гибкое и легко поддерживаемое программное обеспечение.

Хорошо разработанная архитектура уменьшает затраты на разработку и сопровождение приложения, увеличивает его производительность, надежность и безопасность, а также облегчает добавление новых функций и модулей в будущем.

И перед тем, как продолжить, необходимо сделать небольшое теоретическое вступление и поговорить про так называемые слои абстракции. Основная суть в том, чтобы разделять логические части нашего приложения в какие-то отдельные модули.

Рассмотрим на примере. Есть слой, который называется DAL (data access layer) или слой доступа к данным. В нашем случае за этот слой можно принять Mongoose, поскольку все операции к базе данных мы делаем с помощью него. Однако, если бы мы сами описывали SQL запросы, то было бы разумно выделить эту логику по обращению в базу данных в отдельный слой, в отдельный модуль.

Следующий слой, это Controller. В нем идет работа с клиент-серверной составляющей, например params, body, headers и так далее и соответственно из контроллера, мы возвращаем ответ с сервера на клиент и указываем статус код.

И последний слой, это Service. В сервисе описана уже непосредственно какая-то логика, это получить список файлов из базы данных, как-то с ними там поработать и вернуть. Куда возвращаются данные, уже абсолютно не важно.

## Серверная часть

### Структура проекта

Разберем файловую структуру проекта на серверной части:

* Внутри папки "controllers" содержатся файлы, которые выполняют роль контроллеров в приложениях на основе Node.js. Контроллеры отвечают за обработку запросов, взаимодействие с базой данных и другую бизнес-логику.
* Папка "dtos" используется для хранения объектов передачи данных (DTO). DTO представляют собой структуры данных, которые служат для передачи информации между различными компонентами приложения.
* В "error" папке мы обрабатываем ошибки, выполняем валидацию запросов и прочие действия, связанные с обработкой и управлением ошибками.
* Папка "logger" содержит файлы, отвечающие за логирование. Здесь можно настроить запись различных событий, ошибок и информации для последующего анализа и отладки приложения.
* "Middleware" — это папка, в которой находятся файлы промежуточного программного обеспечения. Middleware выполняет функцию обработки запросов перед их достижением до контроллеров. Он может выполнять авторизацию, проверку безопасности, обработку сеансов и другие промежуточные операции.
* В папке "models" содержатся файлы моделей, которые представляют сущности и структуру данных, используемые в приложении. Модели могут быть связаны с базой данных и обеспечивать взаимодействие с ней.
* "Router" — это папка, где хранятся файлы маршрутизации. Маршрутизаторы определяют, какие запросы должны быть направлены к соответствующим контроллерам или обработчикам.
* Папка "schema" содержит файлы, описывающие схему данных, такие как схема базы данных или схема валидации запросов.
* "Service" — это папка, в которой находятся файлы сервисов. Сервисы предоставляют функциональность, необходимую контроллерам для обработки запросов. Они могут содержать бизнес-логику, взаимодействие с базой данных и другие операции.
* "Temp" — это временная папка, которая может использоваться для хранения временных файлов или данных во время выполнения приложения.
* "Static" — папка, где хранятся статические файлы, такие как HTML, CSS, JavaScript и изображения, которые могут быть предоставлены клиентам без обработки на сервере.

### Классы

Рассмотрим схему классов (рисунок 1).



1. Схема классов

В данной диаграмме классов представлена структура и взаимодействие различных классов, относящихся к системе управления пользователями и файлами.

Краткое описание классов:

* UserController: Класс, отвечающий за управление пользователями. Содержит методы для регистрации, входа в систему, выхода из системы, обновления токенов доступа и получения информации о пользователях;
* FileController: Класс, отвечающий за управление файлами. Включает методы для создания директории, получения списка файлов, загрузки файла, скачивания файла, удаления файла, поиска файла, загрузки аватара, удаления аватара, скачивания папки и загрузки папки;
* ApiError: Класс, представляющий различные типы ошибок, которые могут возникнуть при работе с API. Включает методы для обработки ошибок, таких как неверный запрос, неправильная валидация, неавторизованный доступ, истекший токен, ошибка JSON Web Token, запрет доступа и внутренняя ошибка сервера;
* UserDTO: Класс, содержащий данные о пользователе. Включает идентификатор, электронную почту, пароль, фамилию, имя, использованное пространство, пространство на диске, аватар и список файлов;
* UserService: Класс, предоставляющий функциональность, связанную с управлением пользователями. Включает методы для регистрации пользователя, входа в систему, выхода из системы, обновления токенов доступа и получения списка пользователей;
* TokenService: Класс, отвечающий за генерацию, сохранение и удаление токенов доступа. Включает методы для генерации токенов на основе информации о пользователе, сохранения токена в базе данных, удаления токена из базы данных и проверки валидности токена;
* FileService: Класс, предоставляющий функциональность, связанную с управлением файлами. Включает методы для создания директории, получения пути файла, удаления файла и рекурсивного удаления папки.

## Клиентская часть

### Структура проекта

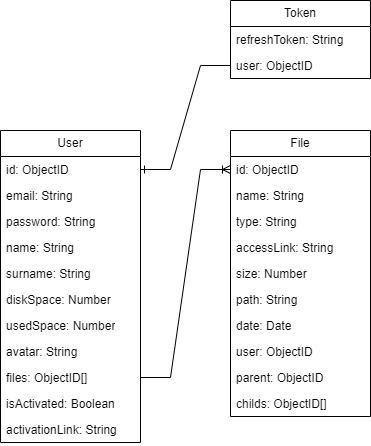
Разберем файловую структуру проекта на клиентской части (рисунок 3):

* Папка "assets" предназначена для хранения дополнительных материалов, таких как изображения, фотографии и другие связанные данные, которые могут быть использованы в проекте;
* В папке "components" располагаются компоненты React, которые являются составными элементами приложения и предназначены для повторного использования в различных частях проекта;
* Папка "config" содержит файлы с настройками и конфигурациями, которые определяют параметры и поведение проекта. Эти файлы позволяют настраивать и адаптировать приложение в соответствии с конкретными требованиями;
* В папке "hooks" находятся хуки React, написанные с использованием TypeScript. Хуки предоставляют возможность легко использовать и повторно применять логику в компонентах, а TypeScript обеспечивает типизацию для более надежной разработки;
* В папке "models" можно хранить модели данных, необходимые для работы приложения;
* Папка "redux" может использоваться для организации глобального состояния и управления состоянием приложения с помощью библиотеки Redux;
* В папке "service" можно разместить файлы, отвечающие за взаимодействие с сервером или сторонними API;
* Папка "utils" предназначена для хранения вспомогательных утилит и функций, которые могут быть использованы в различных частях проекта.

## База данных

При выборе базы данных принято решение использовать MongoDB и библиотеку Mongoose по нескольким причинам. MongoDB является документоориентированной NoSQL базой данных, которая предлагает гибкую и масштабируемую структуру хранения данных.

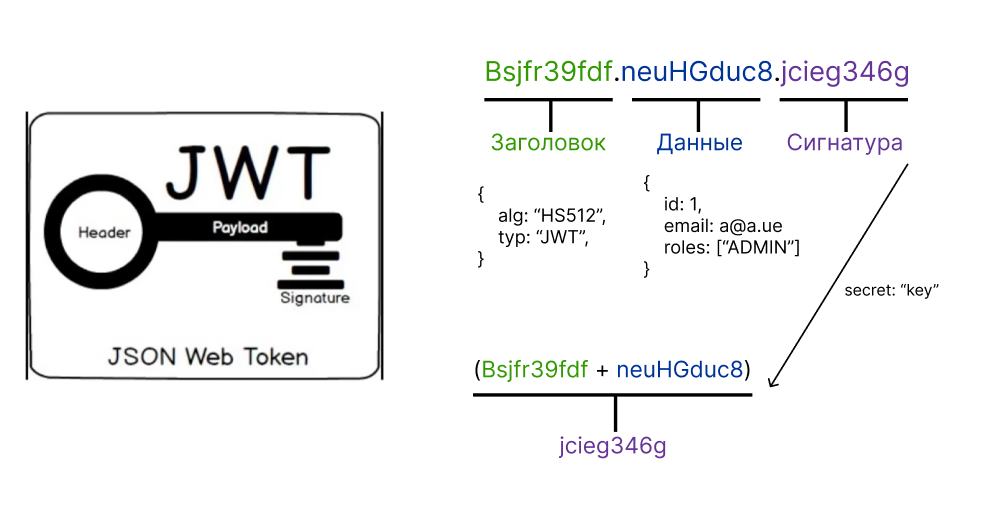
* Первая причина, по которой было принято решение использовать MongoDB, связана с ее гибкостью. MongoDB не требует жесткой схемы данных, что означает, что можно легко изменять структуру данных в процессе разработки, добавлять или удалять поля без необходимости миграции данных. Это особенно полезно при разработке итеративных проектов, когда требования могут меняться со временем;
* Вторая причина связана с масштабируемостью MongoDB. Она предлагает горизонтальное масштабирование, что означает, что можно легко расширять базу данных, добавляя новые серверы, чтобы обеспечить обработку больших объемов данных или высокую производительность. Это особенно важно, если проект предполагает большое количество пользователей или интенсивную нагрузку на базу данных;
* Третья причина связана с использованием библиотеки Mongoose вместе с MongoDB. Mongoose предоставляет простой и интуитивно понятный способ моделирования данных и взаимодействия с MongoDB. Она предлагает удобные средства для создания схем, валидации данных, управления связями между коллекциями и других полезных функций, которые упрощают разработку приложений на основе MongoDB. Ниже предоставлена схема базы данных (рисунок 2).



1. Схема базы данных

## Авторизация

JWT (JSON Web Token) — это открытый стандарт для создания компактных и безопасных токенов, которые используются для аутентификации и авторизации веб-приложений. JWT представляет собой стандартную закодированную строку, которая обычно выглядит так: "rjfJNDened.rfrj48hfkde.mejDnfnced". Она состоит из трех частей, разделенных точками (рисунок 3):



1. Структура JWT токена

* Первая часть – заголовок,
* Вторая часть - полезная нагрузка (данные),
* Третья часть - сигнатура.

Заголовок в основном не представляет большого интереса, но он должен содержать информацию о методе шифрования, используемом для токена, и может также содержать другие вспомогательные элементы или тип токена.

Вторая часть JWT токена содержит данные, которые мы вшиваем в него сами. Обычно это информация о пользователе, такая как его идентификатор, электронная почта, роли, имя пользователя и т. д. Эти данные позволяют определить, к каким страницам у пользователя есть доступ, а к каким запрещен.

Заголовок и данные хранятся в открытом виде и могут быть раскодированы любым. Поэтому важно не хранить в них конфиденциальную информацию. Для получения сигнатуры требуется секретный ключ, который известен только серверу.

Как это работает? Мы берем данные и заголовок, затем используем секретный ключ для их шифрования. В результате получаем сигнатуру. Сигнатура необходима для обеспечения подлинности токена и защиты от подделок или повреждений.

Рассмотрим далее токены, с которыми мы будем работать. У каждого JWT токена есть ограниченное время существования. Со временем он становится недействительным и, по сути, истекает. Это мера безопасности. Например, если злоумышленник украдет токен, а его срок жизни составляет 15 минут, он сможет воспользоваться сервисом только в течение этих 15 минут. Затем токен станет недействительным, и злоумышленник больше не сможет получить доступ.

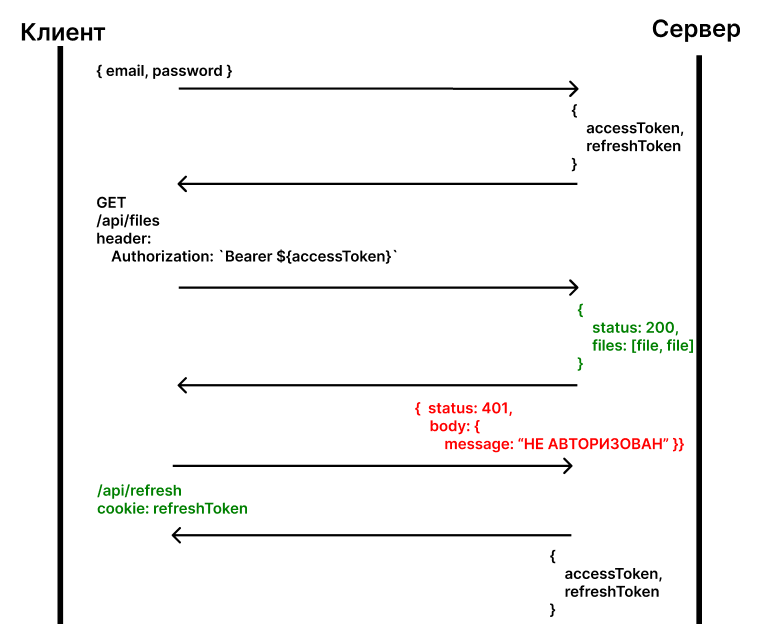
Теперь возникает вопрос: как мы можем получить новый токен? Мы же не хотим входить в систему каждые 15 минут. Для этого были разработаны два типа токенов: токен доступа (access token) и токен обновления (refresh token).

Давайте рассмотрим их различия (рисунок 6). Срок жизни access token составляет примерно 15-30 минут, но этот период может изменяться в зависимости от уровня конфиденциальности используемой в системе информации. Чем короче срок, тем безопаснее. Refresh token живет 15-60 дней. Если пользователь не посещал сервис в течение двух месяцев и его токен истек, пользователю придется повторно войти в систему.

Access token используется для доступа к сервису, а refresh token используется для обновления access токена.

На стороне клиента access token обычно хранится в cookie. А refresh token сохраняется в cookie с флагом httpOnly. Флаг httpOnly является важным условием, которое предотвращает изменение этих cookie через JavaScript. То есть только сервер может непосредственно устанавливать эти cookie. Refresh token также записывается в базу данных на сервере и представляет собой своего рода сессию. Там также можно сохранить IP-адрес, с которого было осуществлено подключение. Если обнаружен новый IP-адрес, можно выполнить определенные действия, например, отправить уведомление о новом входе в систему.

Рассмотрим взаимодействие между клиентом и сервером в представленной схеме (рисунок 4):



1. Клиент-серверное взаимодействие при авторизации пользователей с помощью JWT токенов

Пользователь открывает приложение и выполняет вход, вводя свой электронный адрес и пароль, а затем нажимает кнопку "Войти".

В запросе к серверу содержится указанный электронный адрес и пароль, которые отправляются для обработки. В ответ сервер генерирует пару токенов: токен доступа (access token) и токен обновления (refresh token), и передает их клиенту. Приложение сохраняет эти токены в cookie, а также сохраняет токен обновления в базе данных. Таким образом, мы успешно проходим аутентификацию в системе.

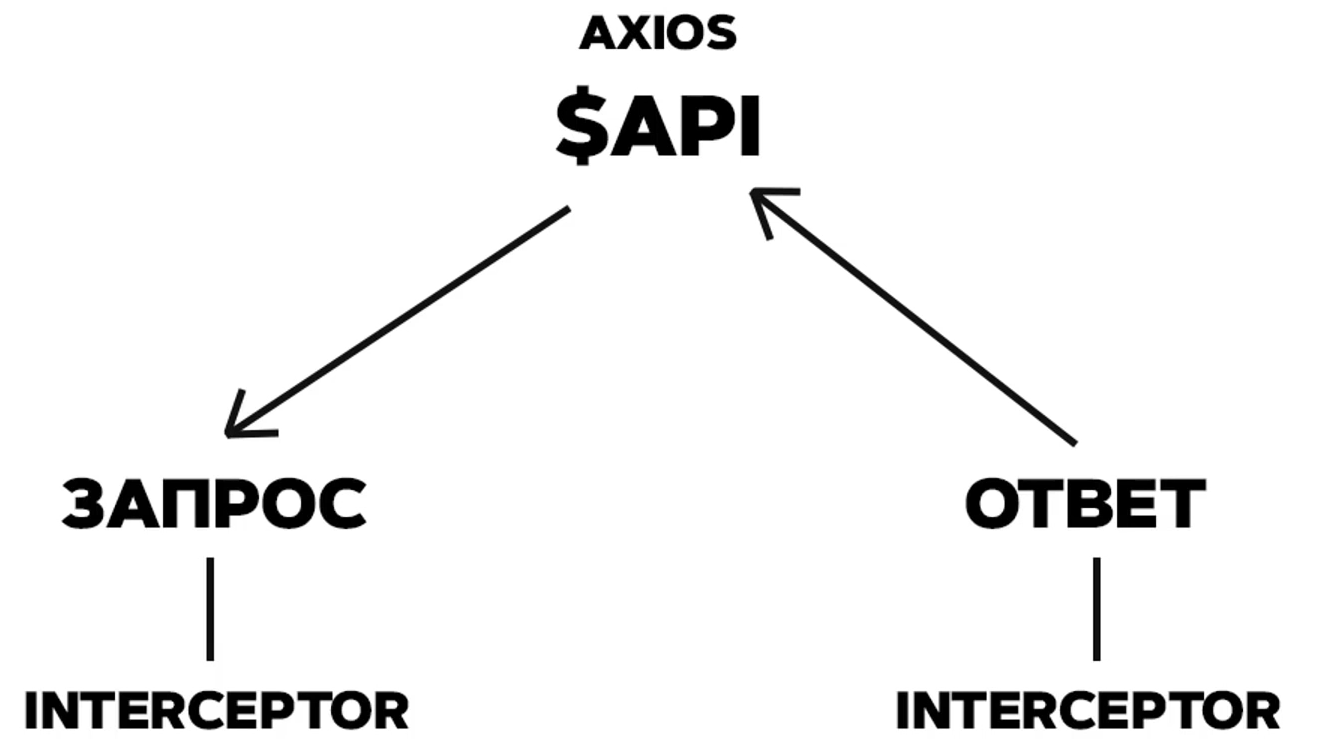
Далее рассмотрим пример, когда приложение отправляет GET-запрос для получения всех файлов пользователя. К запросу добавляется заголовок (header) Authorization, в котором указывается access token. Затем запрос отправляется на сервер. Первое действие сервера - проверка валидности access token: он проверяет, является ли токен подлинным, не истек ли его срок действия и, если все условия выполняются, сервер возвращает статус код 200 и полезную информацию, в данном случае массив с файлами пользователя. Это является стандартным сценарием работы системы, когда все функционирует корректно.

Однако, как упоминалось ранее, срок действия access token, например, составляет 15 минут. Если пользователь отправляет запрос после истечения этого времени и токен становится недействительным, сервер возвращает статус код 401, указывая на отсутствие авторизации пользователя.

Мы предусмотрели такое поведение на клиентской стороне и наличие перехватчика (interceptor) для статус кода 401. Когда мы получаем статус код 401, немедленно отправляется запрос на обновление access токена, причем refresh токен уже находится в cookie и не требует дополнительного указания. Запрос отправляется на сервер, который проверяет токен в соответствии с информацией в базе данных. Затем сервер возвращает новую пару access и refresh токенов, и мы снова можем пользоваться приложением в течение следующих 15 минут. По истечении этого времени весь процесс повторяется снова.

Разберем более подробно механизм перехватчиков и процесс обновления токенов в нашей работе.

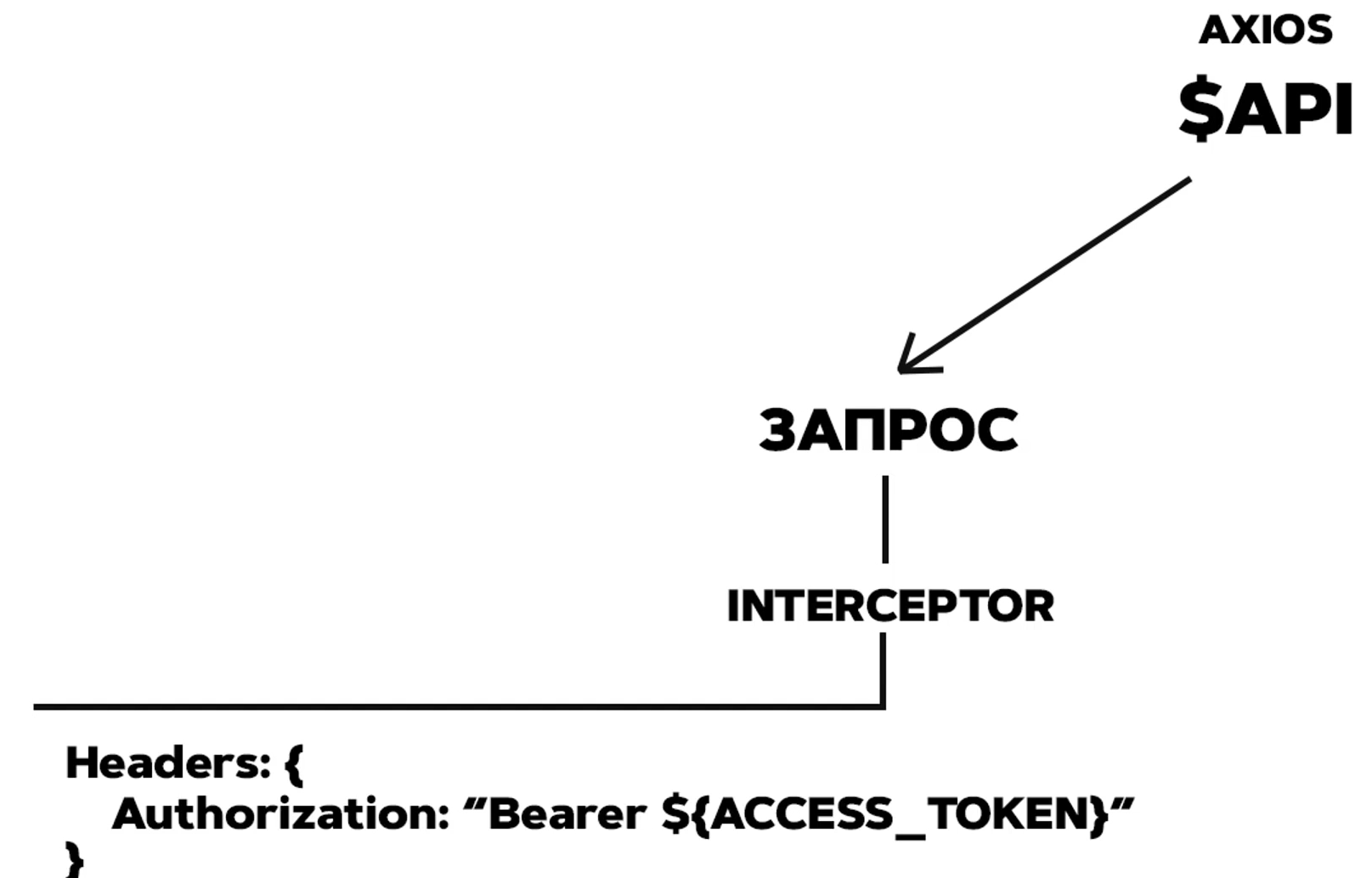
Во-первых, рассмотрим понятие перехватчика. С использованием библиотеки axios (рисунок 5) мы можем отправлять запросы и получать ответы от сервера.



1. Начало схемы перехватчиков с использованием библиотеки axios

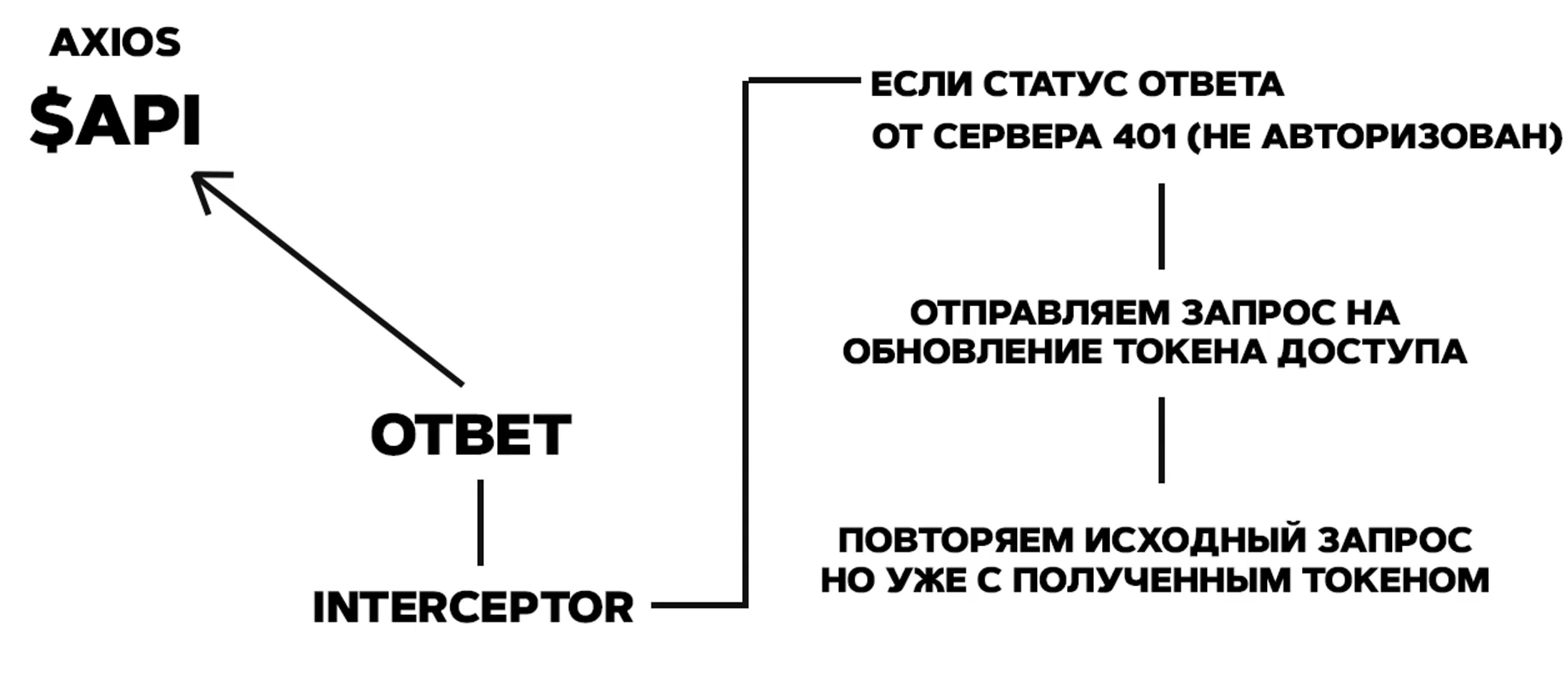
Для обработки этих запросов и ответов мы можем использовать перехватчики, которые являются функциями, выполняющимися для каждого запроса и ответа. Они позволяют нам контролировать и модифицировать данные в процессе обмена информацией с сервером.

Рассмотрим перехватчик, применяемого к запросам (рисунок 6). Здесь всё достаточно просто: перехватчик будет добавлять заголовок Authorization перед каждым запросом, вставляя в него наш токен доступа из cookie. Это позволяет нам избежать ручного добавления этого заголовка каждый раз.



1. Продолжение схемы перехватчика запросов

Теперь обратимся к перехватчику, используемому при получении ответа от сервера (рисунок 7).



1. Продолжение схемы перехватчик ответа

Здесь сложность немного возрастает. Если мы получаем ответ со статусом 200, это означает, что запрос прошел успешно, данные, которые нам нужны, были получены, и нам не требуется выполнять дополнительных действий.

Однако нас интересует ситуация, когда мы получаем ответ со статусом 401, который указывает на неавторизованного пользователя. Например, если срок действия токена доступа истек (токен умирает каждые 15 минут), у нас есть refresh-токен, который действует 30 дней. У сервера также есть конечная точка, которая обновляет токен доступа на основе действующего refresh-токена и возвращает новую пару токенов (доступа и обновления).

Если мы получаем ответ со статусом 401, мы отправляем refresh-токен на эту конечную точку. Если у нас есть действительный refresh-токен, срок его действия не истек, сервер вернет новую пару токенов (доступа и обновления). Затем мы снова записываем токен доступа в cookie на клиенте, и пользователь считается авторизованным. Главное, что нам необходимо сделать, это повторить исходный запрос с обновленным токеном.

Например, если мы отправили запрос для получения всех файлов, и получили ответ со статусом 401, мы отправляем запрос на обновление токенов, получаем их, обновляем и повторяем исходный запрос.

## Сервер

Для удобства разбора функционала, представим приложение в двух частях: сервер и клиент. Начнем с сервера, где основной акцент сделан на реализации бизнес-логики, расположенной в папке "services".

Также как известно из раздела Авторизация, каждый запрос на сервер должен содержать в себе заголовок Authorization, в дальнейшем этот момент будет опущен и не будет упомянут.

### Получение файлов

Когда пользователь заходит в свой Диск, необходимо сразу отправить запрос на получение всех файлов пользователя. Для этого на сервере предусмотрена соответствующая функция getFiles:

Данная асинхронная функция getFiles, которая получает список файлов с определенной сортировкой на основе параметров sort, parent и userId.

Функция принимает три параметра:

* sort - строковое значение, указывающее тип сортировки файлов (может быть 'name', 'type' или 'date').
* parent - значение, определяющее родительскую директорию, из которой нужно получить файлы.
* userId - идентификатор пользователя, для которого нужно получить файлы.

1. Далее, внутри функции, используется оператор switch для определения типа сортировки и выполнения соответствующего запроса к базе данных.
2. Если sort равно 'name', функция использует метод find модели File для поиска файлов, удовлетворяющих следующим условиям. Поле user равно userId и поле parent равно parent.
3. Результаты поиска сортируются по полю name в порядке возрастания (1).
4. Аналогичным образом, если sort равно 'type', файлы сортируются по полю type, а если sort равно 'date', файлы сортируются по полю date.
5. Если значение sort не соответствует ни одному из указанных случаев, функция выполняет обычный запрос find без сортировки.
6. В конце функция возвращает найденные файлы.

### Создание директории

Далее вероятнее всего пользователю нужно будет создать папку. Для этого на сервере создана функция createDir.

Перед тем как рассматривать бизнес-логику в папке service, необходимо подняться на уровень выше в контроллер, поскольку там также присутствует необходимая логика c путями, это сделано для того, чтобы избежать повторения кода:

const file = new File({ name, type, parent, user: userId })

const parentFile = await File.findOne({ \_id: parent })

if (!parentFile) {

file.path = name

fileService.createDir(file, user)

} else {

file.path = `${parentFile.path}\\${file.name}`

fileService.createDir(file, user)

parentFile.childs.push(file.\_id)

await parentFile.save()

}

Данный участок кода проверяет, существует ли объект parentFile, то есть есть ли у добавляемой директории родитель (папка, в которой она лежит). Для этого сначала необходимо получить объект parentFile из базы данных. Если объект parentFile не существует, то устанавливает значение свойства path объекта file равным значению свойства name (поскольку это корневая директория) и вызывает функцию fileService.createDir, передавая ей объекты file и user в качестве аргументов, иначе указываем путь до текущей новой директории с учетом всех родителей.

Рассмотрим основную логику этой функции в папке services. Код проверяет существование пути к файлу на файловой системе и создает папку по указанному пути, если путь не существует. Если путь уже существует, то генерируется исключение. Если происходят другие ошибки в процессе выполнения кода, они также обрабатываются и генерируется соответствующее исключение.

### Загрузка файла и проверка файла

Рассмотрим, как реализована функция загрузки файла на сервер. Данная функция является одной из самых главных и основных функций в приложении. Однако перед тем, как файл будет загружен на сервер и добавлен в базу данных, необходимо провести ряд проверок для данного файла. Это делается для того, чтобы защитить сервер от потенциально возможных действий злоумышленника, такие как: отправка бесконечного файла на сервер, файла слишком большого размера, уже существующего файла и так далее.

Рассмотрим функцию checkFile.

1. Извлекает значения полей name, size и parent из объекта req.body и присваивает их соответствующим переменным nameFile, sizeFile и parentFile.
2. Получает идентификатор пользователя (userId) из объекта req.user.
3. Ищет файл в базе данных, соответствующий указанному пользователю и идентификатору родительского файла.
4. Ищет пользователя в базе данных по идентификатору.
5. Если пользователь не найден, возвращает ошибку "Пользователь не найден".
6. Если размер файла превышает 10 ГБ, возвращает ошибку "Максимальный размер файла 10ГБ".
7. Если использованное пространство пользователя плюс размер файла превышают доступное дисковое пространство пользователя, возвращает ошибку "Недостаточно места на диске".
8. Формирует целевой путь для сохранения файла, учитывая наличие родительского файла.
9. Если целевой путь уже существует (файл с таким именем), возвращает ошибку "Файл уже существует".
10. Если все проверки прошли успешно, возвращает JSON-ответ с полем check, установленным в значение true.
11. Только после того, как клиент получил true, то есть все проверки прошли успешно, выполняется функция uploadFile.

Рассмотрим функцию uploadFile:

И так, для начала необходимо получить все необходимые данные из запроса. Здесь мы получаем сам файл, который был отправлен. Далее получаем временный путь из свойства path у file. Также уникальный идентификатор пользователя.

Далее мы находим директорию, в которую пользователь хочет загрузить новый файл, находим по параметрам. Также находим самого пользователя в базе данных.

Также необходимо изменить поле usedSpace у пользователя, поскольку мы добавляем новый файл, который имеет некий размер.

После этого создаем уже targetPath, это путь, по которому будет сохранен файл в зависимости от указанного parent. С помощью встроенной в nodejs библиотеки fs, перемещаем данный файл в указанную директорию.

В конце создаем объект в базе данных, обновляем некоторые поля, такие как user.files, type и так далее. Сохраняем объект файла в базу данных и возвращаем на клиент объект нового созданного файла и user, с обновленными полями.

### Скачивание файла

После того как пользователь загрузил файл, ему обязательно потребуется его скачать.

Рассмотрим функцию downloadFile, однако в ней нет ничего сложного.

Сначала мы ищем в базе данных файл, который пользователь хочет скачать. Далее если файл найден, то получаем до него верный путь в файловой системе. И в конце с помощью встроенной в nodejs библиотеки fs, отправляем на клиент загрузку физического файла.

### Удаление файла и рекурсивное удаление директорий

Как уже известно, в приложении папки могут храниться внутри других папок, а внутри этих папок могут находиться файлы. Что делает реализацию данных функций немного сложнее, чем если бы пришлось просто удалять файлы. Функция deleteFolderRecursive является рекурсивной функцией и также удаляет все файлы и папки у родителей.

Рассмотрим весь алгоритм целиком вместе с логикой контроллера:

Вначале необходимо получить из базы данных пользователя и файл, который пользователь хочет удалить. Если пользователя или файла нет, возвращаем ошибку.

Так как каждая модель файла в базе данных содержит поле type, это позволяет нам удобно определять, файл это или директория.

В зависимости от того файл это или директория, вызываем у fileService нужные нам функции deleteFile или deleteFolderRecursive соответственно.

Рассмотрим логику функции deleteFile:

Первым делом обновляем поля usedSpace и files у user. В files оставляем только те идентификаторы файлов, которые не равны идентификатору файла, который пользователь удаляет. Далее необходимо найти родителя файла в базе данных, который пользователь собирается удалить. После этого, если родитель найден, необходимо также обновить у него поле childs и убрать тот идентификатор файла, который пользователь собирается удалить. После всех шагов удаляем физический файл из файловой системы сервера.

Теперь необходимо рассмотреть папку deleteFolderRecursive, она посложнее:

В качестве параметров данная функция принимает folder и user. Folder это директория, которую пользователь собирается удалить.

Первым делом необходимо проверить длину поля childs у данной директории, если она равна 0, это означает, что папка пуста и ее можно удалить из базы данных и удалить физически с компьютера.

Далее интереснее, если папка не пуста, необходимо пробежаться по массиву уникальных идентификаторов этой папки, поля childs. Найти данный объект в базе данных и проверить его тип, если тип не dir, применить функцию по удалению файла deleteFile, однако если dir, рекурсивно вызвать deleteFolderRecursive.

После удаления всех файлов и подпапок, удаляем саму папку из базы данных и физически с сервера.

### Поиск файлов

Иногда пользователю необходимо в массе своих файлов и папок найти нужный объект. Для этого на сервере создана функция searchFile, рассмотрим ее:

Mongoose позволяет быстро легко извлекать нужные нам данные из базы данных с помощью регулярного выражения. Для этого сначала создадим его:

const searchRegex = new RegExp(

searchName

.split(/\s+/)

.map((word) => `(?=.\*${word})`)

.join('') + '.\*',

'i',

)

Данный код создает регулярное выражение (RegExp), которое используется для поиска текста по заданному шаблону. Регулярное выражение ищет текст, содержащий все слова из searchName, в любом порядке и в любом месте строки.

Вот пошаговое объяснение кода:

searchName — это переменная, которая содержит строку, по которой будет осуществляться поиск.

split(/\s+/) - вызывается метод split() для разделения строки searchName на отдельные слова. В данном случае, разделение происходит по одному или более пробельному символу.

.map((word) => (?=.\*${word})) - вызывается метод map() для преобразования каждого слова из разделенной строки. Здесь создается новое выражение (?=.\*<word>), где <word> заменяется на каждое отдельное слово из searchName. Выражение (?=.\*<word>) используется для проверки, что в строке содержится каждое слово, но в любом порядке и в любом месте.

.join('') - вызывается метод join() для объединения всех преобразованных выражений из предыдущего шага в одну строку без разделителей.

+'.\*' - добавляется символ .\* в конце строки. Этот символ соответствует любому количеству любых символов в тексте, что означает, что после найденных слов могут быть и другие символы.

'i' - это флаг регулярного выражения, указывающий на регистронезависимый поиск, то есть поиск будет игнорировать различия между прописными и строчными буквами.

Таким образом, финальное регулярное выражение будет искать текст, содержащий все слова из searchName в любом порядке и в любом месте строки, игнорируя регистр.

И далее просто выполняем поиск по данному регулярному выражению.

### Класс Ошибок

Класс ApiError позволяет создавать объекты ошибок, которые могут быть возвращены в ответе на запросы API. Например, если запрос клиента некорректен, серверное приложение может создать экземпляр ApiError с помощью статического метода BadRequest и вернуть его в качестве ответа с соответствующим статусом и сообщением об ошибке. Аналогично, при возникновении других ошибок, таких как ошибка авторизации, истечение срока действия токена или запрещенный доступ, можно использовать соответствующие статические методы класса ApiError, чтобы создать объекты ошибок с нужными значениями.

Разберем данный класс:

В конструкторе класса ApiError определены три свойства: status (статус ошибки), errors (массив ошибок) и name (имя ошибки). Конструктор принимает параметры status (статус ошибки), message (сообщение об ошибке) и errors (необязательный параметр, представляющий массив дополнительных ошибок).

Статические методы в классе ApiError представляют собой методы, которые могут быть вызваны непосредственно на самом классе, а не на его экземплярах. Это означает, что они не требуют создания объекта класса для их использования. Вместо этого они вызываются непосредственно на самом классе ApiError.

В данном случае класс ApiError определяет несколько статических методов, каждый из которых возвращает новый экземпляр класса ApiError с определенными значениями статуса и сообщения об ошибке.

Давайте рассмотрим каждый из этих статических методов более подробно:

* BadRequest(message): Этот метод создает новый экземпляр ApiError с статусом 400 (некорректный запрос), сообщением "Некорректный запрос" и дополнительными ошибками, если они указаны. Он принимает параметр message, который представляет собой дополнительное сообщение об ошибке.
* badValidation(errors): Этот метод создает новый экземпляр ApiError с статусом 422 (ошибка при валидации), сообщением "Ошибка при валидации" и массивом ошибок errors, который может содержать дополнительные информации об ошибках валидации.
* unauthorized(): Этот метод создает новый экземпляр ApiError с статусом 401 (не авторизован), сообщением "Не авторизован" и без дополнительных ошибок.
* tokenExpiredError(): Этот метод создает новый экземпляр ApiError с статусом 401 (срок действия токена истек), сообщением "Срок действия токена истек" и без дополнительных ошибок.
* jsonWebTokenError(): Этот метод создает новый экземпляр ApiError с статусом 401 (недействительный токен), сообщением "Недействительный токен" и без дополнительных ошибок.
* forbiddenError(): Этот метод создает новый экземпляр ApiError с статусом 403 (запрещено), сообщением "Запрещено" и без дополнительных ошибок.
* internalError(message, errors): Этот метод создает новый экземпляр ApiError с статусом 500 (внутренняя ошибка сервера), сообщением message и массивом ошибок errors, который может содержать дополнительные информации об ошибках.

Статические методы предоставляют удобный способ создания экземпляров класса ApiError с предопределенными значениями статуса и сообщения об ошибке, что может быть полезно при обработке ошибок в API.

### Middleware

Middleware в Node.js и Express представляет собой функции, которые выполняются последовательно в цепочке обработки запросов (request) и ответов (response) во время обработки HTTP-запросов в приложении Express. Middleware позволяет выполнять определенные операции, обрабатывать запросы, модифицировать объекты запроса и ответа, а также передавать управление следующей функции middleware в цепочке.

Разберем middleware, который проверяет аутентификацию и авторизацию пользователя:

Основные шаги, выполняемые в этом middleware, следующие:

1. Проверка типа запроса: если метод запроса равен 'OPTIONS', то управление передается следующей middleware функции с помощью вызова next(). Это позоляет обработать предварительные запросы CORS.
2. Получение заголовка Authorization: извлекается значение заголовка Authorization из запроса.
3. Проверка наличия заголовка Authorization: если заголовок Authorization отсутствует, то middleware вызывает ошибку "Не авторизован" с помощью next(ApiError.unauthorized()).
4. Проверка типа токена: Значение заголовка Authorization разделяется на две части: тип токена и сам токен. Если тип токена не равен 'Bearer', то middleware вызывает ошибку "Запрещено" с помощью next(ApiError.forbiddenError()).
5. Проверка наличия токена: Если токен отсутствует, то middleware вызывает ошибку "Недействительный токен" с помощью next(ApiError.jsonWebTokenError()).
6. Проверка данных токена: Токен проверяется с помощью функции tokenService.validateAccessToken(accessToken). Если данные токена не найдены, middleware вызывает ошибку "Запрещено" с помощью next(ApiError.forbiddenError()).
7. Проверка валидности токена: если данные токена являются экземпляром pkg.JsonWebTokenError, то middleware вызывает ошибку "Недействительный токен" с помощью next(ApiError.jsonWebTokenError()).
8. Создание объекта UserDto: Данные токена используются для создания объекта UserDto, который представляет информацию о пользователе. Объект UserDto присваивается свойству req.user.
9. Передача управления следующей middleware: после успешной проверки аутентификации, управление передается следующей middleware функции с помощью вызова next().
10. Обработка ошибок: если во время выполнения происходит ошибка, то middleware вызывает ошибку "Внутренняя ошибка сервера" с помощью next(ApiError.internalError(error)).

Таким образом, данная middleware функция обрабатывает проверку аутентификации пользователя, валидацию и обработку токена доступа, а также создание объекта UserDto для использования в дальнейшей обработке запросов API.

## Клиент

После тщательного рассмотрения реализации основных функций на серверной стороне, настало время перейти к рассмотрению аспектов, связанных с клиентской частью.

Для работы с сервером была использована надстройка на redux-toolkit – rtk query.

RTK Query (Redux Toolkit Query) - это библиотека, которая предоставляет набор инструментов и конвенций для упрощения работы с запросами к API в Redux-приложениях. Она разработана с учетом принципов Redux и цели упростить и стандартизировать управление состоянием запросов и данных, получаемых от удаленных серверов.

Для начала необходимо создать инстанс RTK QUERY с помощью метода createApi, это сделано для того, чтобы каждый запрос мы делали с одного и того же экземпляра класса, в котором уже будет необходимая настройка:

export const rtkAPI = createApi({

reducerPath: 'api',

baseQuery: baseQueryWithReauth,

tagTypes: ['Users', 'Files'],

endpoints: (build) => ({}),

})

В данном коде мы создаем экземпляр RTK API с помощью функции createApi. В параметрах функции мы указываем несколько конфигурационных параметров:

reducerPath: указывает путь, по которому будет храниться состояние API в Redux-хранилище.

baseQuery: предоставляет базовую функциональность для выполнения запросов к API. В данном случае, используется baseQueryWithReauth, что предполагает добавление автоматической пераутентификации при выполнении запросов.

tagTypes: определяет типы тегов, которые можно использовать для группировки запросов. В данном примере, указаны типы 'Users' и 'Files'.

endpoints: определяет список эндпоинтов, которые мы хотим использовать в приложении. В данном коде передается функция build, которая должна вернуть объект с определениями эндпоинтов.

В результате выполнения этого кода, у нас будет создан экземпляр RTK API, который будет доступен для использования в приложении для выполнения запросов к серверу с помощью определенных эндпоинтов.

### Перехватчики

Как было сказано в теории, необходимо реализовать перехватчики. В результате выполнения кода, у нас будет создан базовый запрос, который может быть использован в RTK API для выполнения запросов к серверу. Базовый запрос будет включать базовый URL, учетные данные и заголовки, которые содержат токен доступа (если он существует) для авторизации запросов к серверу.

Необходимо реализовать перехватчик для ответов от сервера, для этого используем функцию baseQueryWithReauth:

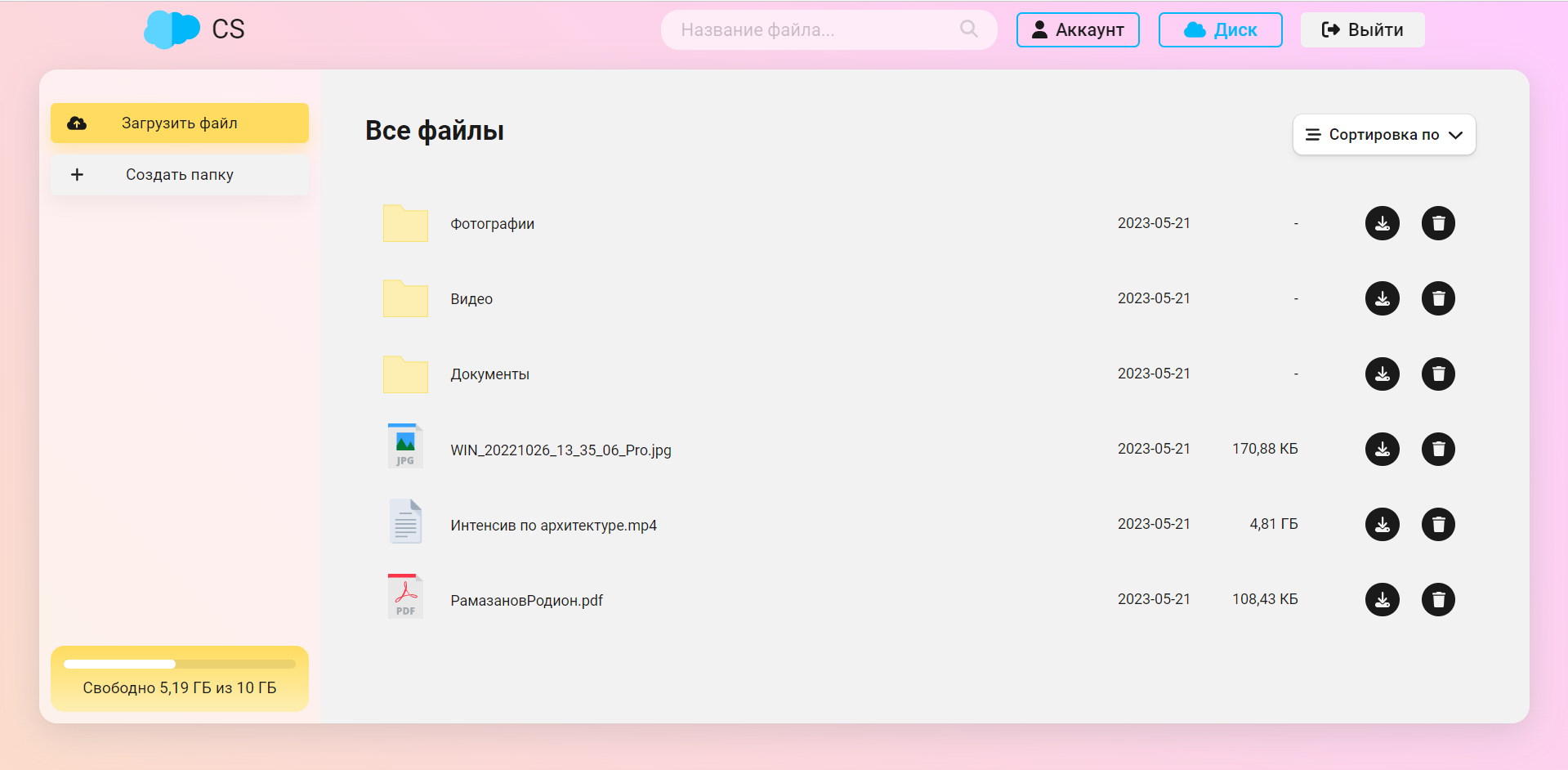
1. Сначала мы ожидаем, пока мьютекс (mutex) не станет доступным без блокировки его. Это гарантирует, что другие запросы, требующие обновления токена доступа, будут ожидать до его получения.
2. Затем мы выполняем базовый запрос baseQuery с переданными параметрами args, api и extraOptions и сохраняем результат в переменную result.
3. Мы проверяем, есть ли ошибка в результате запроса и если статус ошибки равен 401 (неавторизован), то выполняем следующие шаги:
4. Проверяем, не заблокирован ли мьютекс в данный момент. Если мьютекс не заблокирован, то мы захватываем его с помощью функции acquire.
5. В блоке try пытаемся получить новый токен доступа, выполнив запрос baseQuery с 'auth/refresh' в качестве аргумента. Если успешно получаем данные токена, то мы устанавливаем новый токен в куки с помощью библиотеки Cookies, сохраняем данные пользователя и повторяем первоначальный запрос, снова используя baseQuery.
6. Если не удалось получить новый токен доступа, то мы вызываем экшен logOut для выхода пользователя из системы.
7. В блоке finally вызываем функцию release для освобождения мьютекса и позволяет другим запросам получить доступ к обновленному токену.
8. Если мьютекс заблокирован в данный момент, то мы ожидаем, пока он не станет доступным без блокировки.
9. Затем мы выполняем базовый запрос baseQuery повторно с переданными параметрами args, api и extraOptions и сохраняем результат в переменную result.
10. Возвращаем результат запроса result из функции.

Эта модифицированная версия baseQuery позволяет автоматически обновлять токен доступа и повторять запросы после обновления токена, если полученная ошибка указывает на неавторизованный доступ. Это обеспечивает автоматическую пераутентификацию и более безопасную работу с авторизацией в приложении.

### Получение файлов

И так, для получения всех файлов (рисунок 8), необходимо использовать эндпоинт getAllFiles, который является запросом типа query и возвращает массив файлов (IFile[]).

Функция query принимает параметры { currentDir, sortValue }, которые представляют текущую директорию (currentDir) и значение сортировки (sortValue). Далее просто в зависимости от значения sortValue и currentDir, отправляем запрос на нужный эндпоинт.

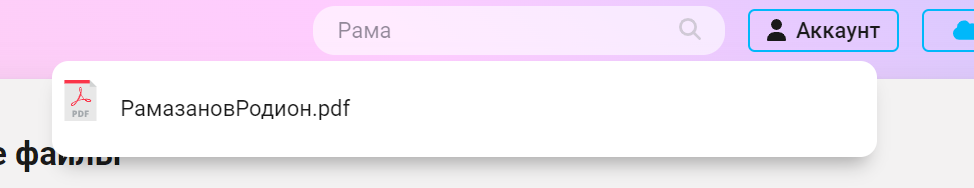


1. Страница Диска

### Поиск файлов

Для того, чтобы отправить запрос на сервер для поиска файлов (рисунок 9), необходимо использовать endpoint searchFile.

Функция query принимает параметр searchName, который представляет строку для поиска имени файла. URL-адрес формируется с использованием параметра searchName, добавляя его в параметр search в URL запроса.



1. Строка поиска файлов

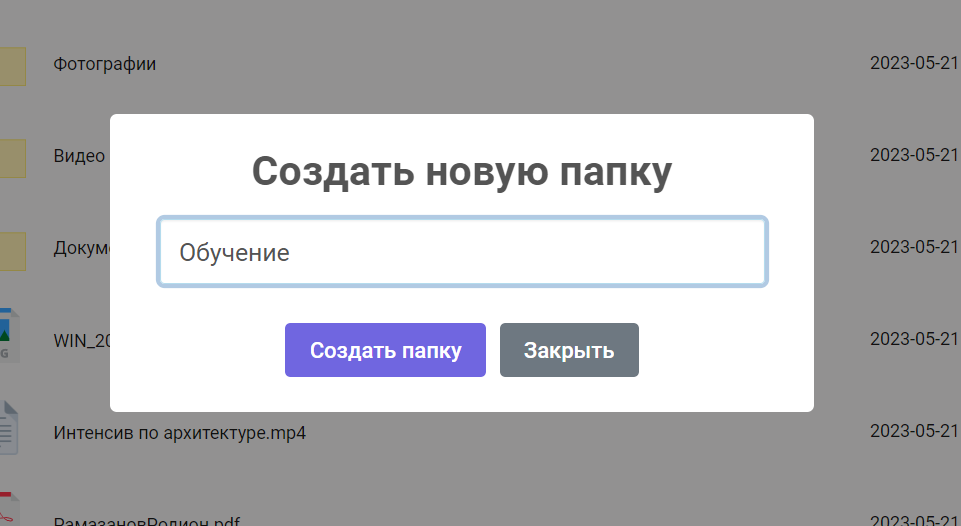
### Удаление файлов

Для удаления файла, необходимо использовать эндпоинт deleteFile. Рассмотрим его реализацию:

Эндпоинт является мутацией (mutation). Он выполняет запрос на удаление файла и возвращает объект с сообщением (message) и информацией о пользователе (user). Функция mutation принимает параметр file, представляющий информацию о файле, который нужно удалить (IFile). URL-адрес формируется с использованием параметра file.\_id для указания идентификатора файла, который нужно удалить. Метод запроса устанавливается в 'DELETE' для выполнения удаления.

### Создание директории

Для создания папок (рисунок 10) необходимо использовать эндпоинт createDir. createDir является мутацией (mutation).



1. Форма создания директории

Он выполняет запрос на создание новой директории и возвращает объект с информацией о созданной директории (file) и информацией о пользователе (user).

Функция mutation принимает параметры { name, currentDir }, представляющие имя новой директории (name) и текущую директорию (currentDir).

Внутри функции mutation мы возвращаем объект с настройками запроса, содержащий url, method и body. URL-адрес устанавливается в /files для указания места создания новой директории. Метод запроса устанавливается в 'POST' для выполнения создания. В поле body передаем объект с данными для создания новой директории, включая name (имя директории), parent (идентификатор родительской директории) и type (тип, в данном случае 'dir').

### Загрузка файлов

Теперь более подробно рассмотрим одну из основных функций, она позволяют пользователю загрузить физический файл на сервер.

Для загрузки файлов необходимо использовать эндпоинт uploadFile, который является мутацией (mutation). Он выполняет запрос на загрузку файла на сервер и возвращает объект с информацией о загруженном файле (file) и информацией о пользователе (user). В представленном коде определен эндпоинт uploadFile, который является мутацией (mutation). Он выполняет запрос на загрузку файла на сервер и возвращает объект с информацией о загруженном файле (file) и информацией о пользователе. Функция mutation принимает параметры { file, currentDir }, где file представляет загружаемый файл (File) и currentDir представляет текущую директорию (ICrumb).

Внутри функции mutation мы определяем асинхронную функцию queryFn, которая выполняет логику запроса на загрузку файла.

Далее мы отправляем POST-запрос на эендпоинт checkFile, который как было описано выше, проверяет файл на размер, проверяет существует ли данный файл и так далее. Если все успешно проходит, сервер возвращает check: true. Это означает, что все готово и теперь возможно отправить физический файл на сервер. Для того, чтобы отправить файл на сервер, необходимо преобразовать его в BLOB формат:

Формат BLOB (Binary Large Object) представляет собой тип данных, используемый веб-браузерами для хранения и обработки двоичных данных, таких как изображения, видео, аудио и другие файлы.

Далее мы отправляем запрос на сервер с отслеживанием прогресса загрузки. Если все прошло успешно, возвращаем данные в response.data.

# Тестирование приложения

При проведении тестирования приложения было уделено особое внимание проверке функциональности и совместимости клиентской части в различных браузерах. Были протестированы популярные браузеры, включая Google Chrome, Mozilla Firefox, Safari и Microsoft Edge. Целью тестирования было убедиться, что приложение корректно отображается и функционирует одинаково хорошо во всех этих браузерах. Были проверены различные аспекты, такие как правильность отображения интерфейса, работоспособность пользовательских функций и обработка событий. Все тесты клиентской части были проведены вручную.

Кроме того, для обеспечения надежности и корректной работы серверной части приложения были проведены ручные тесты с использованием программы Postman. Postman позволяет отправлять HTTP-запросы к серверу и проверять его ответы. Это позволило проверить различные API-методы и убедиться, что сервер правильно обрабатывает запросы и возвращает ожидаемые данные. Во время тестирования были проверены различные сценарии использования, включая обработку запросов на создание, чтение, обновление и удаление данных. Также было уделено внимание проверке обработки ошибок и безопасности сервера.

В результате проведенного тестирования было выявлено, что клиентская часть приложения успешно прошла все проверки и демонстрирует согласованную работу во всех поддерживаемых браузерах. Серверная часть также была успешно протестирована с помощью программы Postman, что подтвердило ее надежность и функциональность. Все выявленные проблемы были исправлены, и приложение готово к использованию.

ЗАКЛЮЧЕНИЕ

В рамках данной курсовой работы было разработано веб-приложение для хранения файлов и предоставления доступа к ним. Серверная часть приложения была реализована с использованием Node.js и фреймворка Express, в то время как клиентская часть была разработана с использованием React, Redux Toolkit и TypeScript.

В заключение можно отметить, что в рамках данной работы были выполнены следующие задачи:

* было проведено исследование существующих облачных хранилищ, а также проведен анализ их функциональности, интерфейсов и особенностей. Это позволило получить полное представление о текущем состоянии рынка и сравнить различные решения.
* путем сбора требований от потенциальных пользователей были определены основные функции и возможности, которые должны быть реализованы в облачном хранилище. Это позволило учесть потребности пользователей и создать продукт, отвечающий их ожиданиям.
* была разработана архитектура облачного хранилища, включая выбор подходящих технологий и протоколов. Это позволило оптимизировать работу хранилища, обеспечить его масштабируемость и эффективность.
* был создан пользовательский интерфейс для работы с файлами и папками в облачном хранилище, включая синхронизацию и обмен файлами между устройствами, а также обеспечение безопасности данных и аутентификацию пользователей. Это обеспечило удобство использования и защиту информации для конечных пользователей.
* проведение тестирования и отладки разработанного облачного хранилища позволило проверить его функциональность и надежность. Были выявлены и исправлены возможные ошибки и проблемы, что способствовало повышению качества продукта.
* была проведена оценка эффективности разработанного облачного хранилища на основе проведенных тестов и сравнение с существующими решениями. Результаты показали, что разработанное решение обладает высокой функциональностью и надежностью, а также конкурентоспособно в сравнении с аналогичными продуктами.

В ходе работы над данным проектом приобрелся ценный опыт в разработке веб-приложений, используя Node.js, Express, React, Redux Toolkit и TypeScript. Была осознана важность обеспечения безопасности при работе с файлами и применении соответствующих мер для защиты данных пользователей. Также были развиты навыки интеграции между серверной и клиентской частями приложения, обеспечения удобного пользовательского опыта и использования передовых технологий разработки.