МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное

образовательное учреждение высшего образования

**«Челябинский государственный университет»**

**(ФГБОУ ВО «ЧелГУ»)**

**ОТЧЕТ**

Производственная практика

(Технологическая (проектно-технологическая) практика)

(вид практик: учебная, производственная)

ООО «Технология Программирования»

(наименование организации и место прохождения практики)

Факультет (институт/филиал) Институт информационных технологий

Кафедра информационных технологий и экономической информатики

Ф.И.О. студента Хрусталёв Павел Алексеевич

Номер группы ИТЗ-501

|  |  |
| --- | --- |
| Руководитель практики от организации  Дубовицкий Игорь Евгеньевич  (фамилия, имя, отчество)  Директор  (занимаемая должность)  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись руководителя практики от  организации) М.П.  «\_\_\_\_» \_\_\_\_\_\_\_\_\_2024 г. | Руководитель практики от университета  Митянина Анастасия Владимировна  (фамилия, имя, отчество)  Доцент кафедры ИТиЭИ, канд. физ.-мат. наук  (ученая степень и /или звание, занимаемая должность) |
|  | Отметка о допуске к защите  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись руководителя практики  от университета)  «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2024 г. |

Челябинск, 2024 г.

**Оглавление**

1. Стек технологий разработки ……………………………………………4
2. Описание архитектурных решений …………………………………...12
3. Проектирование ………………………………………………………...18
4. Реализация ………………………………………………………………38

**Введение**

В современных корпоративных средах обеспечение эффективного и защищенного обмена информацией между сотрудниками является критически важной задачей. Обычные мессенджеры и почтовые клиенты часто не соответствуют требованиям безопасности, надежности и гибкости, необходимым в условиях корпоративной работы, особенно в сфере IT. В данном проекте разработан специализированный корпоративный мессенджер, который решает проблему безопасного обмена данными, улучшает коммуникацию внутри компании и повышает продуктивность сотрудников.

Традиционные инструменты коммуникации редко соответствуют высоким требованиям корпоративного сектора. Они либо не предлагают необходимые функции для взаимодействия сотрудников на уровне рабочих групп, либо обладают ограниченными возможностями для интеграции с внутренними процессами и безопасностью на уровне предприятия. В этом контексте актуальность разработки специализированного корпоративного мессенджера заключается в возможности обеспечить защищенную, функциональную и удобную платформу для внутренних коммуникаций.

Корпоративный мессенджер, созданный в рамках данного проекта, предоставляет широкий функционал, способствующий оптимизации взаимодействия сотрудников. Основные функции, такие как поддержка групповых и приватных чатов, видеосвязь и обмен файлами, реализованы с учетом потребностей сотрудников IT-компаний. В дополнение к этому, функционал для администратора, позволяющий контролировать верификацию пользователей, управление их статусами и поиск, способствует улучшению безопасности и контроля над рабочими процессами. Такой подход к внутренней коммуникации позволяет снизить риски, связанные с утечкой данных, а также оптимизировать рабочий процесс и повысить уровень доверия сотрудников к системе.

**1. Стек технологий разработки**

Мессенджер — это высоконагруженное приложение, которое должно обеспечивать работу с большими объемами данных в реальном времени, обеспечивая стабильность и безопасность взаимодействия между пользователями. Для этого был выбран стек технологий, который будет эффективно решать задачи по производительности, масштабируемости и удобству разработки. Ниже представлен сравнительный анализ выбора каждой технологии.

1.1. Клиентская часть: TypeScript + React.js + WebRTC + Redux

1.1.1. TypeScript

TypeScript был выбран для клиентской части благодаря строгой типизации, которая упрощает отладку и поддержку кода. Его преимущества наиболее заметны в крупных проектах с множеством взаимодействующих компонентов.

Среди альтернативных решений можно выделить:

1. **JavaScript**: Прост в использовании, но не имеет встроенной статической типизации. Это приводит к увеличению количества ошибок на этапе выполнения, что критично для сложных приложений.
2. **Flow**: Подобен TypeScript, но имеет меньшую поддержку в экосистеме JavaScript и сообществе, что замедляет разработку.
3. **Dart**: Хорошая альтернатива для фронтенда, но требует специального окружения, что повышает порог входа.

Вывод: TypeScript был выбран благодаря сочетанию строгой типизации, интеграции с существующей экосистемой JavaScript и широкому сообществу разработчиков.

* + 1. React.js

React был выбран для создания клиентской части из-за своей популярности, высокой производительности и удобства разработки UI в приложениях с динамическим обновлением интерфейса. Основные причины выбора:

* Компонентный подход: React позволяет создавать переиспользуемые и независимые компоненты, что идеально подходит для такого интерфейса, как мессенджер.
* Виртуальный DOM: это позволяет React эффективно обновлять только измененные части интерфейса без перерисовки всего документа, что важно для приложения с частыми обновлениями данных.
* Поддержка хуков и состояния: React отлично справляется с управлением состоянием компонентов, что критично для работы с сообщениями в реальном времени.
* Большое сообщество и множество готовых решений, библиотек, что ускоряет разработку.

Альтернативы React:

* **Angular**: Подходит для крупных корпоративных приложений благодаря встроенным инструментам, но сложен в обучении, имеет более высокий порог входа и больший размер финальной сборки, что критично для веб-приложений, ориентированных на производительность.
* **Vue.js**: Простота Vue делает его привлекательным, однако экосистема меньше, чем у React. Vue менее популярен среди крупных компаний, что может затруднить поиск готовых решений и интеграцию с некоторыми библиотеками.
* **Svelte**: Обеспечивает высокую производительность, но имеет меньшее сообщество и ограниченную поддержку в сравнении с React.

Вывод: React был выбран благодаря компонентному подходу, оптимизации через виртуальный DOM и наличию широкого сообщества, что позволяет ускорить разработку и обеспечить стабильность.

* + 1. WebRTC

WebRTC используется для организации видеочатов в мессенджере. Это единственная технология, предоставляющая нативную поддержку peer-to-peer соединений в браузере без необходимости сторонних плагинов или серверов для медиа-трафика. Альтернатив WebRTC на уровне браузера не существует. Преимущества технологии:

* Прямая связь (peer-to-peer): позволяет организовать видеосвязь без передачи видеопотока через сервер, что сокращает нагрузку.
* Широкая поддержка браузеров: WebRTC поддерживается большинством современных браузеров без необходимости устанавливать дополнительное ПО.
* Низкая задержка: WebRTC минимизирует задержку передачи данных, что критично для видеочатов.
  + 1. Redux

Redux — это инструмент для управления состоянием данных и пользовательским интерфейсом в приложениях JavaScript с большим количеством сущностей.

Для чего нужен Redux

* для управления состоянием приложения, работающего с большим количеством данных;
* удобной замены встроенных средств работы с состоянием в React;
* более легкого масштабирования приложения, его преобразования под разные задачи;
* избавления от ошибок, связанных с беспорядком в объекте состояния;
* более простой отладки и доработки;
* Отсутствие передачи данных по цепочке из компонента в компонент.

Среди альтернатив:

* **Context API** (React): Хорошо работает для небольших приложений, но сложен в масштабируемых проектах, где необходимо управлять большими данными.
* **MobX**: Обеспечивает более простую реализацию реактивного управления состоянием, но имеет слабую поддержку инструментов разработки (DevTools) и может быть сложен для новичков.
* **Recoil**: Перспективная альтернатива от команды React, но на момент выбора еще довольно «сырая».

Вывод: Redux был выбран благодаря зрелой экосистеме, удобству работы с глобальным состоянием и инструментам для отладки.

* 1. Серверная часть: Node.js + MongoDB + WebSocket
     1. Node.js

Node.js был выбран для серверной части мессенджера по нескольким причинам:

* Асинхронность и события: Node.js использует неблокирующую модель ввода-вывода, что идеально подходит для приложений с высокой нагрузкой на сеть и частыми запросами, такими как мессенджеры.
* Высокая производительность: Благодаря движку V8 и асинхронной обработке запросов, Node.js может обрабатывать большое количество соединений одновременно.
* Единый язык на сервере и клиенте: Использование JavaScript/TypeScript как на клиенте, так и на сервере упрощает разработку, позволяет использовать одни и те же библиотеки и модули, что ускоряет процесс разработки и тестирования.
* Поддержка WebSocket: Node.js имеет отличную поддержку WebSocket, что необходимо для обмена данными в реальном времени.

Альтернативы:

* **Python** (Django/Flask): Подходит для разработки, но имеет меньшую производительность для приложений в реальном времени из-за блокирующей модели обработки запросов.
* **Ruby on Rails**: Удобен для быстрой разработки, но его производительность и масштабируемость значительно ниже, чем у Node.js.
* **Go**: Очень производительный язык, но из-за своей строгой типизации требует больше времени на разработку, а экосистема меньше, чем у JavaScript.

Вывод: Node.js был выбран благодаря своей асинхронности, высокопроизводительному движку V8 и возможности унификации разработки (один язык для фронтенда и бэкенда).

* + 1. MongoDB

MongoDB была выбрана как база данных по нескольким причинам, связанным с особенностями работы мессенджера:

* Горизонтальное масштабирование: MongoDB поддерживает шардирование, что позволяет эффективно масштабировать систему при росте нагрузки. Для мессенджера, который может работать с большим количеством пользователей и сообщениями, горизонтальное масштабирование критично.
* Документо-ориентированная структура данных: MongoDB позволяет хранить данные в виде документов JSON, что дает гибкость при работе с различными типами данных, такими как текстовые сообщения, медиафайлы, метаданные о чатах и пользователях.
* Высокая скорость записи и чтения: MongoDB оптимизирована для быстрого чтения и записи, что важно для мессенджеров с высокой частотой обновлений данных (например, новые сообщения).
* Гибкость в работе с данными: В MongoDB можно легко добавлять новые поля в коллекции без миграции схемы.

Список альтернатив:

* **PostgreSQL**: Отличается высокой производительностью и поддержкой сложных запросов, но для хранения вложенных данных (например, вложенных сообщений и их метаданных) требует больше усилий.
* **MySQL**: Подходит для структурированных данных, но менее гибок при работе с изменяющимися схемами.
* **Cassandra**: Отличный выбор для больших объемов данных, но сложен в настройке и интеграции.

Вывод: MongoDB был выбран за счет гибкости работы с данными, масштабируемости и высокой скорости чтения/записи.

* + 1. WebSocket

WebSocket — это протокол, который используется для обмена сообщениями в реальном времени, и он критичен для мессенджера. В отличие от традиционного HTTP, который является односторонним (клиент отправляет запрос, сервер отвечает), WebSocket позволяет установить постоянное соединение между клиентом и сервером, через которое можно отправлять данные в обе стороны без дополнительных запросов. Это особенно важно для:

* Real-time: WebSocket позволяет моментально передавать новые сообщения и обновления состояния между клиентами, что является неотъемлемой частью мессенджера.
* Снижения нагрузки: WebSocket уменьшает необходимость в постоянных HTTP-запросах, что снижает нагрузку на сервер.

Альтернативы:

* **HTTP/REST + Polling:** Неэффективно, так как требует постоянного опроса сервера, создавая лишнюю нагрузку.
* **HTTP/REST + Long Polling:** Улучшенный вариант Polling, но увеличивает задержку в сравнении с WebSocket.
* **Server-Sent Events (SSE):** Подходит для однонаправленных потоков данных, но не поддерживает двустороннюю связь. Вывод: WebSocket был выбран как наиболее оптимальное решение для передачи данных в режиме реального времени с минимальной задержкой и нагрузкой на сервер.

1.2.4. Вспомогательные библиотеки

* **bcryptjs** – для хеширования паролей перед их сохранением, что повышает безопасность.
* **concurrently** – позволяет запускать несколько команд одновременно, что полезно при разработке, где требуется параллельный запуск клиентской и серверной частей.
* **cookie-parser** – для работы с cookie, что важно для управления аутентификацией.
* **cors** – позволяет клиенту взаимодействовать с сервером, расположенным на другом домене, защищая приложение от нежелательных междоменных запросов.
* **dotenv** – для управления конфиденциальными настройками и переменными окружения, такими как ключи API и параметры подключения.
* **express** – фреймворк для создания веб-приложений на Node.js. Упрощает создание API и маршрутизацию, поддерживает интеграцию с другими библиотеками.
* **express-async-handler** – помогает с обработкой ошибок в асинхронных функциях, упрощая код.
* **express-validator** – для проверки и валидации данных, вводимых пользователем, снижая вероятность ошибок и защищая систему от некорректных данных.
* **http-proxy-middleware** – промежуточный слой для перенаправления запросов, полезен при разработке приложений с разделением на фронтенд и бэкенд.
* **jsonwebtoken** – используется для создания и проверки токенов JWT, обеспечивая безопасное управление доступом.
* **multer**  – для обработки загружаемых файлов, например, изображений и других вложений в чаты.
* **nodemon** – автоматический перезапуск сервера при изменении кода, что ускоряет разработку.
* **peer** – для упрощения установки peer-to-peer соединений с использованием WebRTC.
* **sharp** – для обработки изображений (обрезка и оптимизация), что экономит трафик и повышает производительность.
* **ts-node** – позволяет запускать TypeScript-код в Node.js, ускоряя разработку и отладку без предварительной компиляции в JavaScript.
* **validator** – библиотека для валидации данных, например, проверки корректности email и других полей, вводимых пользователем.

**2. Описание архитектурных решений**

В разработке мессенджера будет применена клиент - серверная архитектура, которая позволяет обеспечивать гибкую работу приложения. Взаимодействие происходит через REST API и WebSocket для обмена данными в реальном времени.

2.1. Клиентская часть (Frontend)

Структура клиентской̆ части разделена на следующие части:

* Компоненты интерфейса (UI): эти компоненты отвечают за отображение данных и взаимодействие с пользователем, такие как поля ввода, кнопки, страницы и т.д. Для их реализации используется HTML, CSS и TypeScript.
* Управление состоянием приложения: с помощью библиотеки Redux Toolkit, происходит централизованное хранение состояния данных, которое доступно в любой части приложения посредством импорта необходимых функций.
* Маршрутизация: осуществление навигации по приложению, благодаря библиотеке – react-router-dom, без перезагрузки страницы, что оптимизирует загрузку контента.
* Вспомогательные файлы: так как проект написан на TypeScript, приложение содержит отдельные файлы с описанием типов сообщений, ролей, вида чатов, а также их интерфейсов и данных, которые приходят с сервера и отправляются на сервер.

2.2. Серверная часть (Backend)

На сервере будет использоваться архитектура с разделением логики на модели и контроллеры, поэтому здесь применена архитектура типа MVC (Model-View-Controller), но с адаптацией под работу с клиентом на React. Подробнее о структуре:

1. **Model** (Модель):

* Описание структуры данных: Модели представляют структуру данных, хранящуюся в базе данных MongoDB, и описываются с помощью вспомогательной библиотеки Mongoose. Приложение содержит такие сущности, как: User, Room, и Message, каждая из которых определяет поля и методы, связанные с объектами в базе данных.
  + User (Пользователь)

Эта сущность представляет пользователя приложения и описывает его основные атрибуты, включая персональные данные, роль и состояние.

Поля:

* + - \_id: string — уникальный идентификатор пользователя.
    - name: string — имя пользователя.
    - surname: string — фамилия пользователя.
    - patronymic: string — отчество.
    - email: string — адрес электронной почты.
    - position: string — должность пользователя.
    - status: string — текущий статус пользователя.
    - notifications: string[] — список уведомлений пользователя.
    - isVerified: boolean — подтвержден ли пользователь.
    - avatar: string — путь к аватару пользователя, по умолчанию «default.jpg».
    - phone: string — телефонный номер пользователя.
    - gender: string — пол пользователя.
    - role: string — роль пользователя (администратор или пользователь).
    - dateOfBirthday: Date — дата рождения пользователя.
    - password: string — хэшированный пароль.
    - last\_seen: number — время последней активности пользователя в виде метки времени.
  + Room (Комната):

Сущность представляет собой чат-комнату, в которой пользователи могут обмениваться сообщениями.

Поля:

* + - \_id: string — уникальный идентификатор комнаты.
    - avatar: string — изображение комнаты.
    - imageGroup: string— изображение группового чата
    - title: string— название комнаты.
    - creator: string — создатель комнаты.
    - type: «private» | «group» | «video» — тип комнаты (приватная, групповая, видеочат).
    - participants: objectId[] — участники комнаты.
    - lastMessage: string — последнее сообщение в комнате.
    - createdAt: Date — дата создания комнаты.
    - updatedAt: Date — дата последнего обновления комнаты. archivedUsers: objectId [] — список идентификаторов пользователей, которые архивировали эту комнату.
    - messages: objectId[]— массив сообщений, связанных с комнатой.
  + Message (Сообщение)

Сущность определяет структуру сообщения, отправляемого пользователем.

Поля:

* + - \_id: string — уникальный идентификатор сообщения.
    - roomId: ObjectId— комната, в которой отправлено сообщение.
    - senderId: ObjectId — отправитель сообщения.
    - messageType: «text» | «file» | «image» | «voice» — тип сообщения (текст, файл, изображение, голосовое сообщение).
    - isRead: boolean — статус прочтения сообщения.
    - createdAt: Date — дата и время создания сообщения.
    - repliedMessage: ObjectId— информация о сообщении, на которое был дан ответ.
    - updatedAt: Date (опционально) — дата и время обновления сообщения.
    - text: string (опционально) — текст сообщения (если тип — текст).
    - content: contentType (опционально) — метаданные вложенного файла, содержащие:
    - filename: string — имя файла.
    - size: string — размер файла.
* Бизнес-логика и валидация: Некоторые модели могут включать методы для выполнения бизнес-логики, такие как обновление статуса пользователя, архивация чатов, а также валидацию полей данных, чтобы обеспечить их корректность перед сохранением.

1. **Controller** (Контроллер):

* Обработка запросов: Контроллеры служат для обработки запросов от клиента и взаимодействия с моделями. Они принимают запросы и выполняют операции, такие как: добавление, обновление, получение или удаление данных, и возвращают ответы обратно клиенту (frontend).
* **REST API**: Контроллеры реализуют взаимодействие с клиентом посредством REST API, который позволяет клиенту на React выполнять запросы на создание, обновление или получение данных в виде JSON-объектов. Список групп контроллеров проекта включает:

1. **AuthControllers** для обработки запросов на регистрацию и авторизацию.
2. **RoomControllers** для управления чатами.
3. **MessageControllers** для получения списка сообщений, их отправки и удаления, работа с вложенными файлами.
4. **UserControllers** для получения списка пользователей, их поиска, редактирования.
5. **AdminControllers** для контроля доступа пользователей к чату, обновлению должности.
6. **Middleware**. Это функции промежуточной обработки, которые выполняются до основного кода приложения, например HTTP-запроса, внося какие-либо дополнительные изменения перед выполнением:

* Проверка валидности JWT – токена при аутентификации пользователя. Выполнение middleware происходит перед контроллерами, проверяя, что запросы исходят от авторизованных пользователей.
* Обработка ошибок: Middleware в данном приложении включают обработчики ошибок для возврата стандартных ответов при возникновении исключений.

1. **Express Router**. Позволяет структурировать маршруты API и привязывает их к соответствующим контроллерам, чтобы запросы клиента направлялись в нужные контроллеры и обрабатывались там.

Каждая группа маршрутов представляет отдельный функциональный модуль, связанный с различными задачами приложения. Описание маршрутов:

* **/api/auth** – эта группа маршрутов отвечает за авторизацию и аутентификацию пользователей.
* **/api/user** – эта группа маршрутов отвечает за взаимодействие с пользователями.
* **/api/admin** – данная группа маршрутов необходима для управления приложением со стороны администратора
* **/api/room**– эта группа маршрутов отвечает за управление чатами (комнатами)
* **/api/messages** – этот маршрут управляет сообщениями в приложении.

1. **WebSocket**. Для лучшей организации кода при работе с сокетами используется библиотека socket.io. Так как для мессенджера необходима работа в реальном времени, обязательно всегда поддерживать соединение с клиентом, чтобы обеспечить мгновенные уведомления, обновления статусов, синхронизацию чатов и участников в них. Socket.IO работает параллельно с REST API, позволяя клиенту подписываться на события (например, «user-online», «message-received»), которые контроллеры могут инициировать через Socket.io. При отправке сообщений разумно использовать сокеты уже после записи данных в бд, исключая риск потери сообщений после перезагрузки страницы.

Архитектура корпоративного мессенджера представлена на рисунке 1.

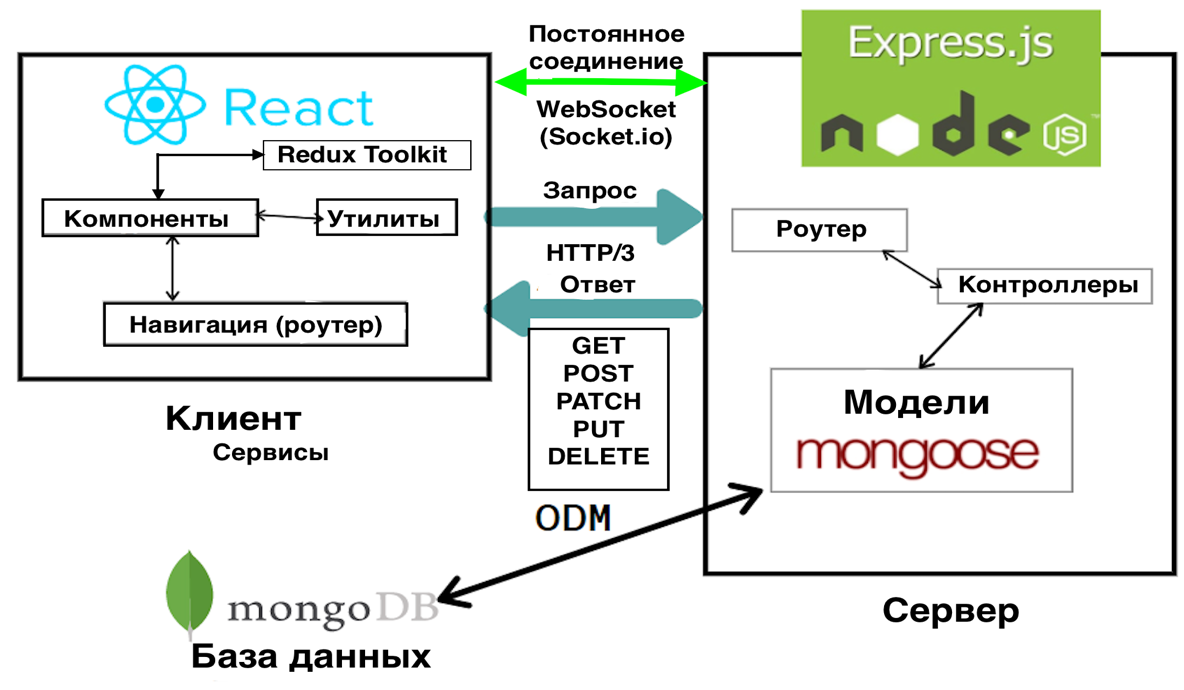


Рисунок 1 – Архитектура корпоративного мессенджера

Данный гибридный подход MVC c REST API и WebSocket, объединяющий традиционную MVC-архитектуру с API и событиями реального времени, позволяет отделить логику взаимодействия с данными от интерфейса, улучшая читаемость и поддержку кода, а также повышая масштабируемость и интерактивность приложения.

**3. Проектирование**

3.1. Структура интерфейса пользователя

В данном разделе представлены результаты проектирования интерфейса корпоративного мессенджера. Прототипы экранов (страниц) разработаны с целью наглядного представления внешнего вида приложения и понимания его функциональности конечными пользователями.

3.1.1. Страница регистрации пользователя

На данной странице пользователь заполняет необходимую о себе информацию: ФИО, дату рождения, должность, почту, номер телефона, пол и пароль. После заполнения всех полей пользователь нажимает кнопку «Регистрация». Прототип представлен на рисунке 2.

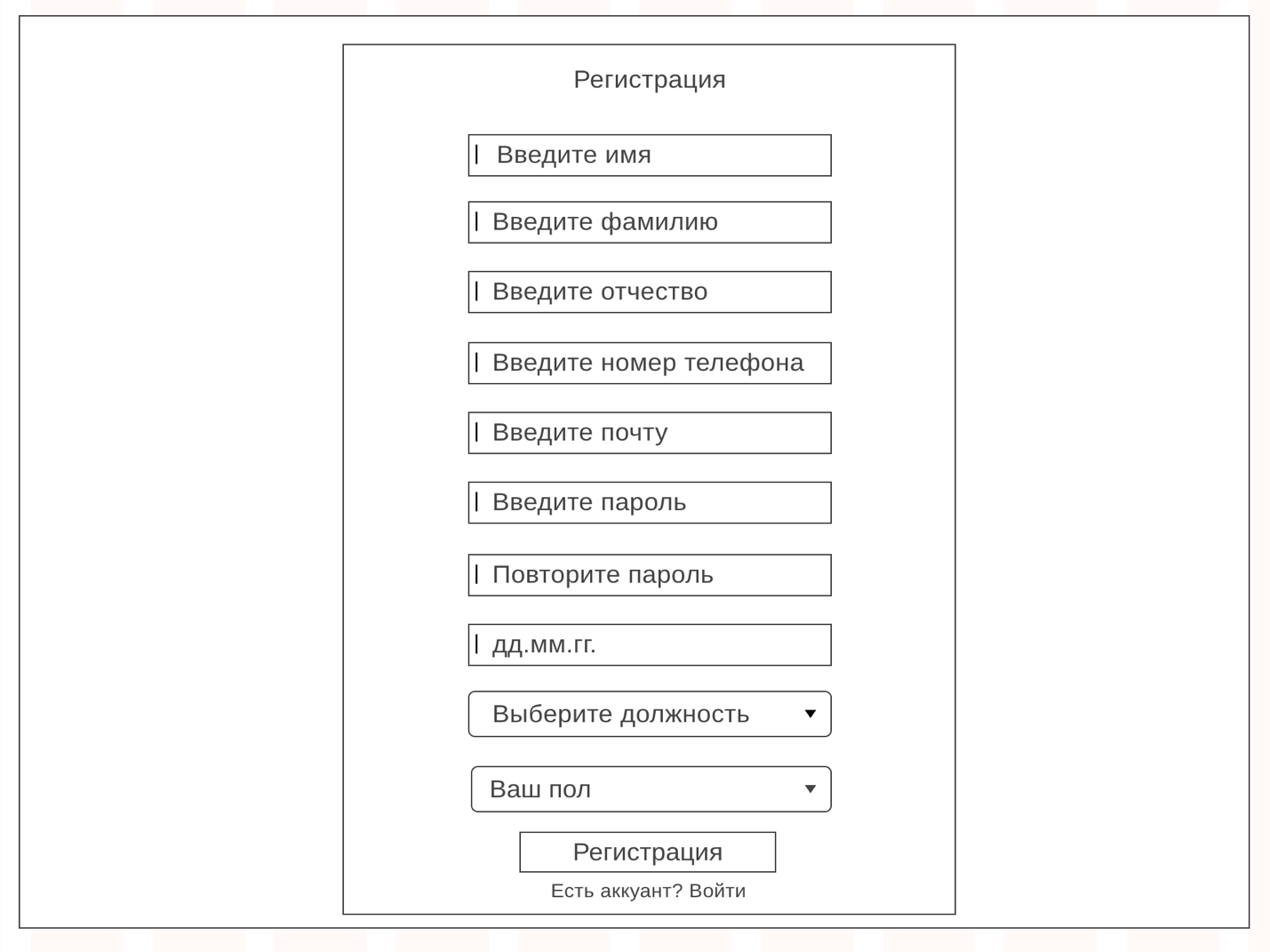


Рисунок 2 – прототип страницы регистрации мессенджера

3.1.2. Страница авторизации пользователя

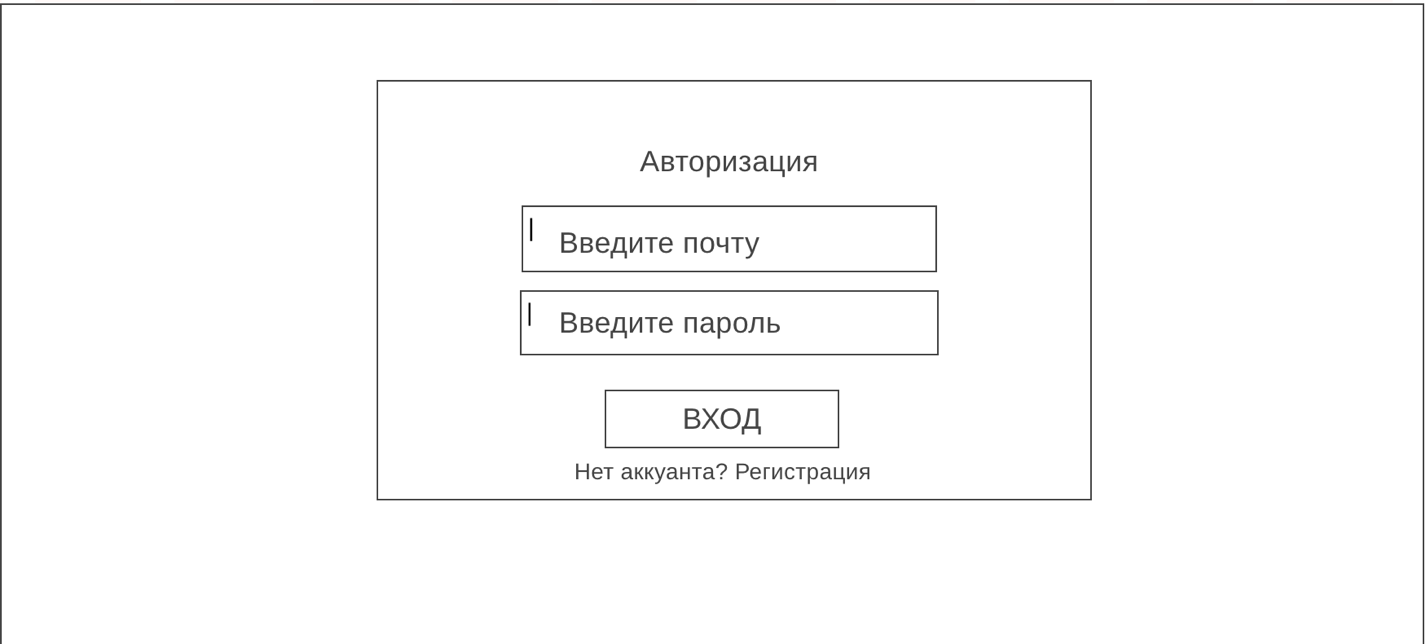
На странице авторизации пользователь вводит свой логин и пароль, указанные при регистрации, затем нажимает кнопку «Вход», чтобы войти систему. Пример показан на рисунке 3.

Рисунок 3 – прототип страницы авторизации мессенджера

3.1.3. Главная страница

На главной странице находится сам мессенджер. Сбоку имеется панель навигации, где пользователь, с помощью всплывающих модальных окон может выбрать, какой тип чата для создания ему нужен, а также его участников, сменить аватар, посмотреть свой профиль или пользовательскую тему. Также имеется боковая панель со списком диалогов и возможностью фильтрации по их типам: групповые чаты, приватные и архив. На главной панели расположен сам чат с сообщениями, полем ввода и дополнительной информацией о собеседнике и вложенных файлах. Примеры модального окна для создания группового чата и главной страницы на рисунках 4 и 5 ниже.



Рисунок 4 – прототип главной страницы мессенджера

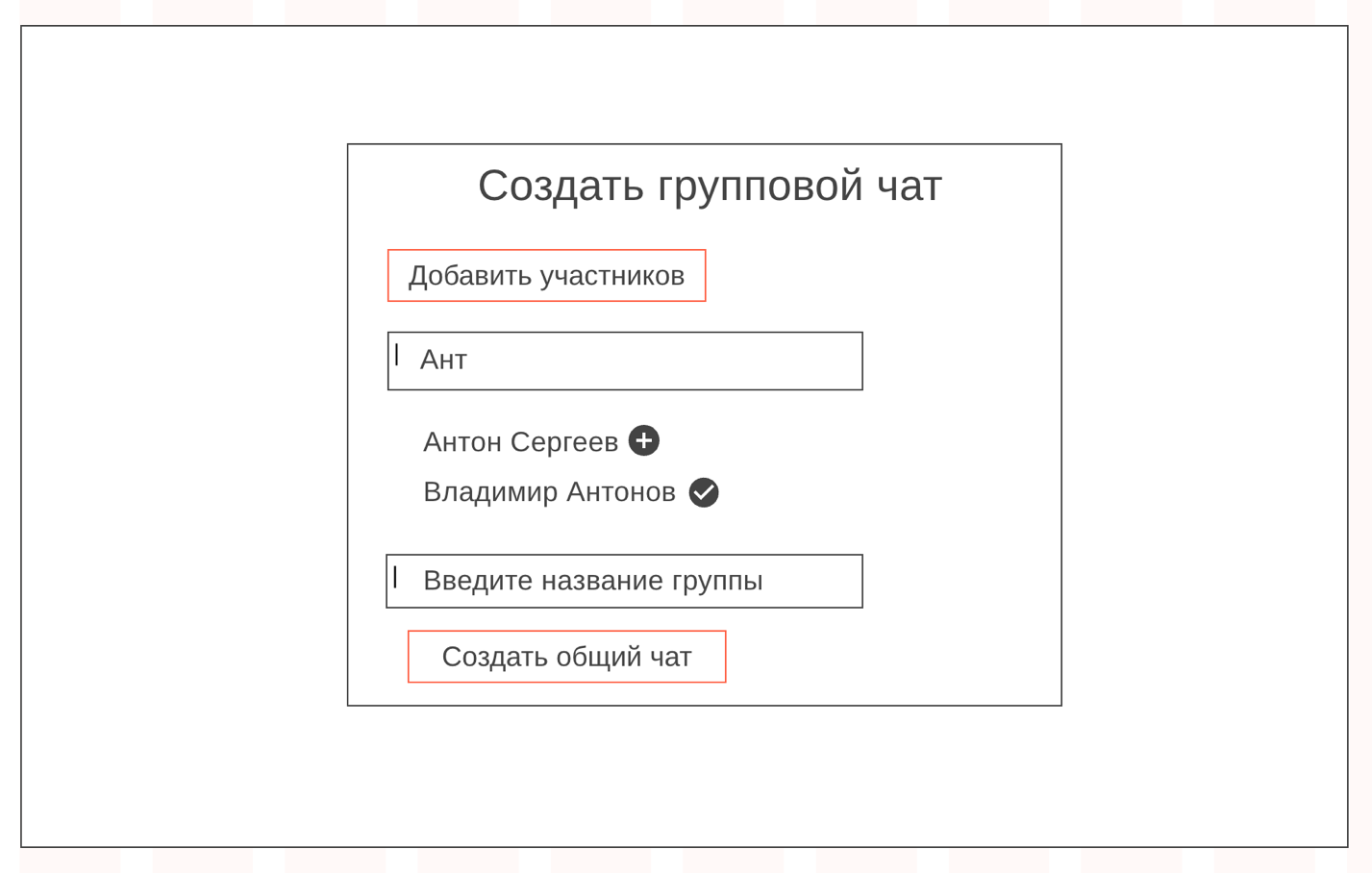


Рисунок 5 – прототип модального окна для создания группового чата

3.1.4. Страница администратора

Интерфейс на странице представляет собой административную панель, предназначенную для управления сотрудниками. Сверху находится поле ввода, для более быстрого поиска необходимого пользователя. Ниже – таблица пользователей с их данными и возможностью для администратора управлять ими: осуществить смену должности и верифицировать / снять верификацию в системе. Пример показан на рисунке 6.



Рисунок 6 – прототип страницы администратора

3.2. Структура базы данных

В рамках разрабатываемого проекта была создана база данных с использованием MongoDB и ODM Mongoose для Node.js. Структура базы данных включает три основные сущности: User (Пользователь), Room (Комната) и Message (Сообщение).

Модель пользователя (**User**) содержит следующие ключевые поля:

* \_id (ObjectId): Первичный ключ, уникальный идентификатор пользователя.
* name (String): Имя пользователя.
* surname (String): Фамилия пользователя.
* patronymic (String): Отчество пользователя.
* dateOfBirthday (Date): Дата рождения пользователя.
* position (String): Должность пользователя (enum допустимых значений/должностей).
* email (String): Email пользователя.
* password (String): Зашифрованный пароль пользователя.
* gender (String): Пол пользователя (enum: мужской/женский).
* status (String): Статус пользователя (онлайн/офлайн).
* isVerified (Boolean): Флаг подтверждения email пользователя.
* avatar (String): Имя файла аватара пользователя.
* phone (String): Номер телефона пользователя.
* role (String): Роль пользователя (user/admin).
* last\_seen (Date): Дата последней активности пользователя.
* createdAt (Date): Дата создания записи пользователя.
* updatedAt (Date): Дата последнего обновления записи пользователя.

Модель включает методы аутентификации и хеширования пароля с использованием пакета **bcrypt**.

Модель комнаты/чата (**Room**) описывает чат-комнату со следующими характеристиками:

* \_id (ObjectId): Первичный ключ, уникальный идентификатор комнаты.
* creator (ObjectId): Внешний ключ, ссылается на \_id пользователя-создателя комнаты. Создатель комнаты является единственным администратором группы.
* type (String): Тип комнаты (enum: private/group/video).
* title (String): Название комнаты (для групповых комнат).
* participants (ObjectId[]): Массив внешних ключей, ссылается на \_id участников комнаты.
* lastMessage (ObjectId): Внешний ключ, ссылается на \_id последнего сообщения в комнате.
* imageGroup (String): Имя файла изображения групповой комнаты.
* messages (ObjectId[]): Массив внешних ключей, ссылается на \_id сообщений в комнате.
* archivedUsers (ObjectId[]): Массив внешних ключей, ссылается на \_id пользователей, которые заархивировали комнату.
* createdAt (Date): Дата создания записи комнаты.
* updatedAt (Date): Дата последнего обновления записи комнаты.

Модель сообщения (**Message**) представляет собой структуру для хранения сообщений:

* \_id (ObjectId): Первичный ключ, уникальный идентификатор сообщения.
* roomId (ObjectId): Внешний ключ, ссылается на \_id комнаты, в которой находится сообщение.
* senderId (ObjectId): Внешний ключ, ссылается на \_id пользователя, отправившего сообщение.
* messageType (String): Тип сообщения (enum: text/file/image/voice).
* isRead (Boolean): Флаг прочтения сообщения.
* content (Object): Объект с информацией о файле (имя файла, размер), если сообщение содержит файл.
* repliedMessage (Object): Ссылка на сообщение, на которое дан ответ (если есть). Структура:
  + senderId (ObjectId): Идентификатор отправителя отвеченного сообщения.
  + messageId (ObjectId): Идентификатор отвеченного сообщения.
* text (String): Текстовое содержимое сообщения.
* createdAt (Date): Дата создания записи сообщения.
* updatedAt (Date): Дата последнего обновления записи сообщения.

Общая схема БД представлена на рисунке 7.

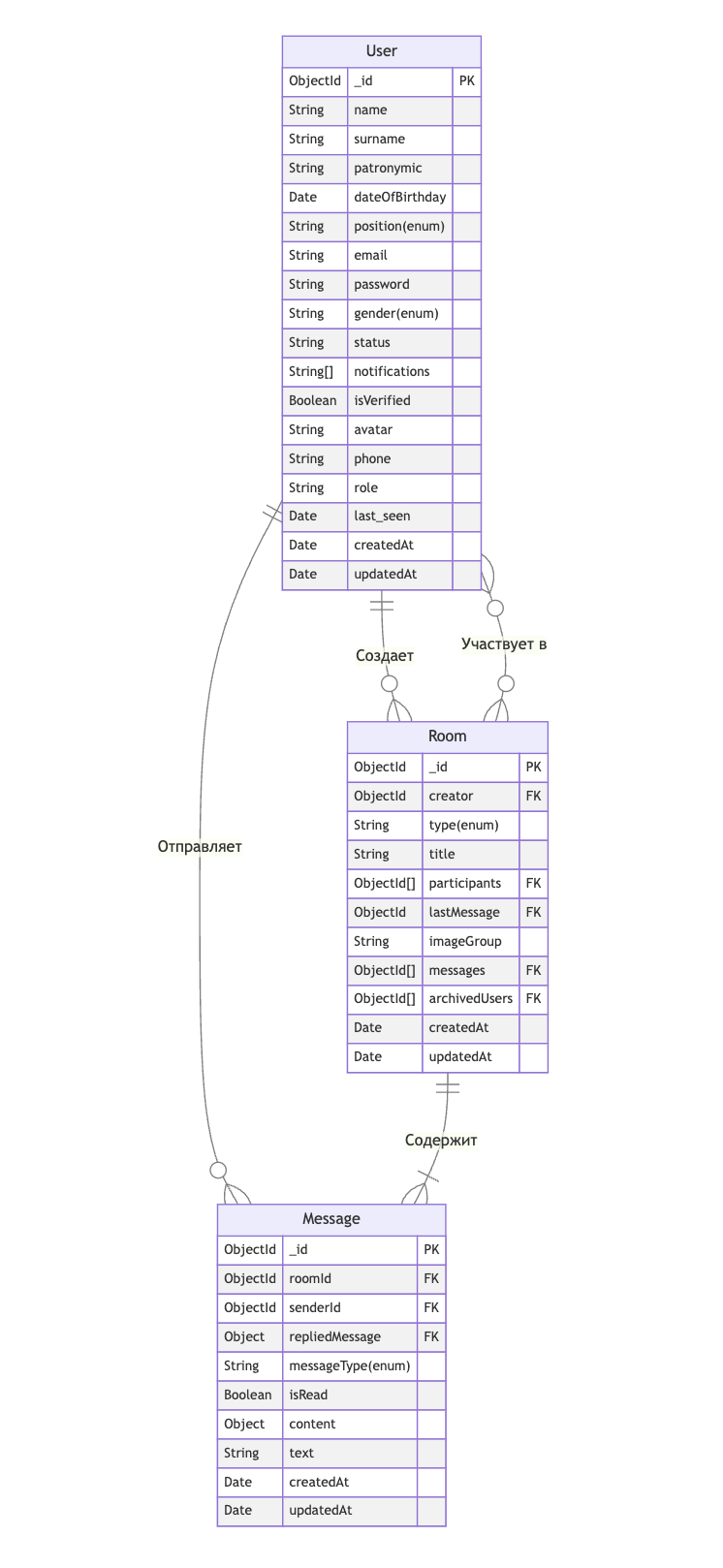


Рисунок 7 – разработанная база данных корпоративного мессенджера

3.2.1. Анализ связей в базе данных

* **Связь Room - User (participants)**

Тип связи: многие-ко-многим (Many-to-Many).

Описание: Чат/комната может иметь несколько участников и каждый пользователь может быть участником нескольких комнат.

Реализация: В схеме Room есть поле «participants», которое является массивом ссылок на документы User. Это позволяет хранить информацию о том, какие пользователи являются участниками конкретной комнаты.

* **Связь Room - User (creator)**

Тип связи: один-ко-многим (One-to-Many)

Описание: Комната может быть создана одним пользователем, но пользователь может создать несколько комнат.

Реализация: В схеме Room есть поле creator, которое является ссылкой на документ User. Это позволяет хранить информацию о том, какой пользователь является создателем конкретной комнаты.

* **Связь Room - Message (messages):**

Тип связи: один-ко-многим (One-to-Many)

Описание: Комната может содержать несколько сообщений, но каждое сообщение принадлежит только одной комнате.

Реализация: В схеме Room есть поле messages, которое является массивом ссылок на документы Message. Это позволяет хранить информацию о том, какие сообщения принадлежат конкретной комнате.

* **Связь Message - User (senderId):**

Тип связи: один-ко-многим (One-to-Many)

Описание: Сообщение отправляется одним пользователем, но пользователь может отправить несколько сообщений.

Реализация: В схеме Message есть поле senderId, которое является ссылкой на документ User. Это позволяет хранить информацию о том, какой пользователь является отправителем конкретного сообщения.

* **Связь Room - User (archivedUsers):**

Тип связи: многие-ко-многим (Many-to-Many)

Описание: Комната может быть заархивирована несколькими пользователями, и каждый пользователь может заархивировать несколько комнат.

Реализация: В схеме Room есть поле archivedUsers, которое является массивом ссылок на документы User. Это позволяет хранить информацию о том, какие пользователи заархивировали конкретную комнату.

Схема связей БД представлена на рисунке 8.

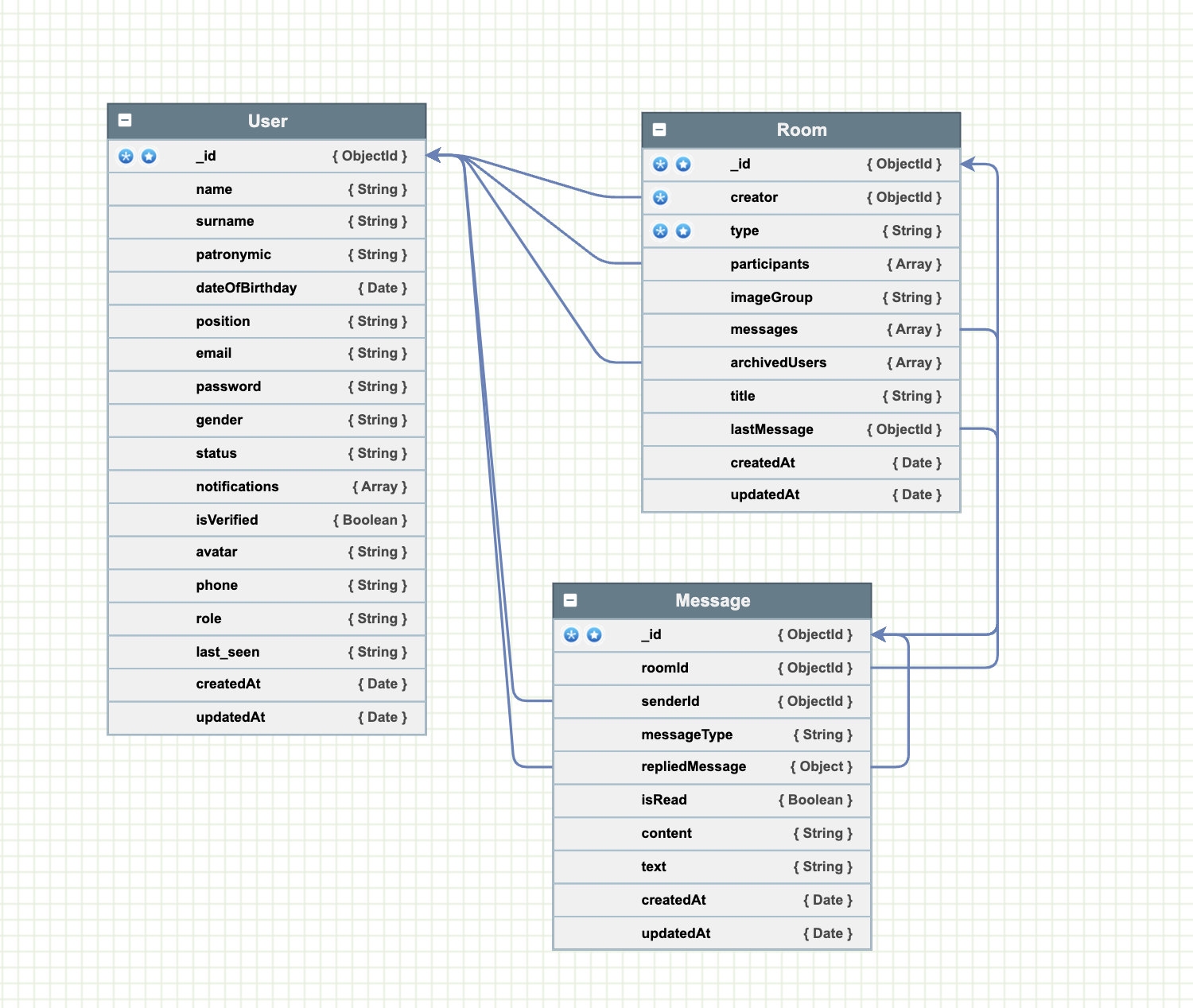


Рисунок 8 – схема связей БД корпоративного мессенджера

3.3. Проектирование API

API – это программный интерфейс приложений, набор инструкций, который позволяет разным приложениям общаться между собой.

Данное API будет построено по принципу разделения ответственности между отдельными модулями и слоями приложения. Выделяются следующие основные компоненты:

1. Контроллеры (controllers) - содержат логику обработки запросов от клиента. В зависимости от типа запроса выполняют операции с базой данных, возвращают ответы, обрабатывают ошибки.
2. Маршруты (routes) - определяют конечные точки API (endpoints) и связывают их с соответствующими методами контроллеров.
3. Также здесь происходит начальная валидация входных данных.
4. Модели (models) - описывают структуру и правила валидации объектов базы данных с помощью Mongoose-схем (Модели подробно описаны в разделе «Проектирование БД»)
5. Middlewares - промежуточные обработчики запросов, которые могут выполнять такие функции, как аутентификация пользователя, проверка прав доступа, логирование и т.д.

Архитектура API сервера корпоративного мессенджера представлена на рисунке 9.

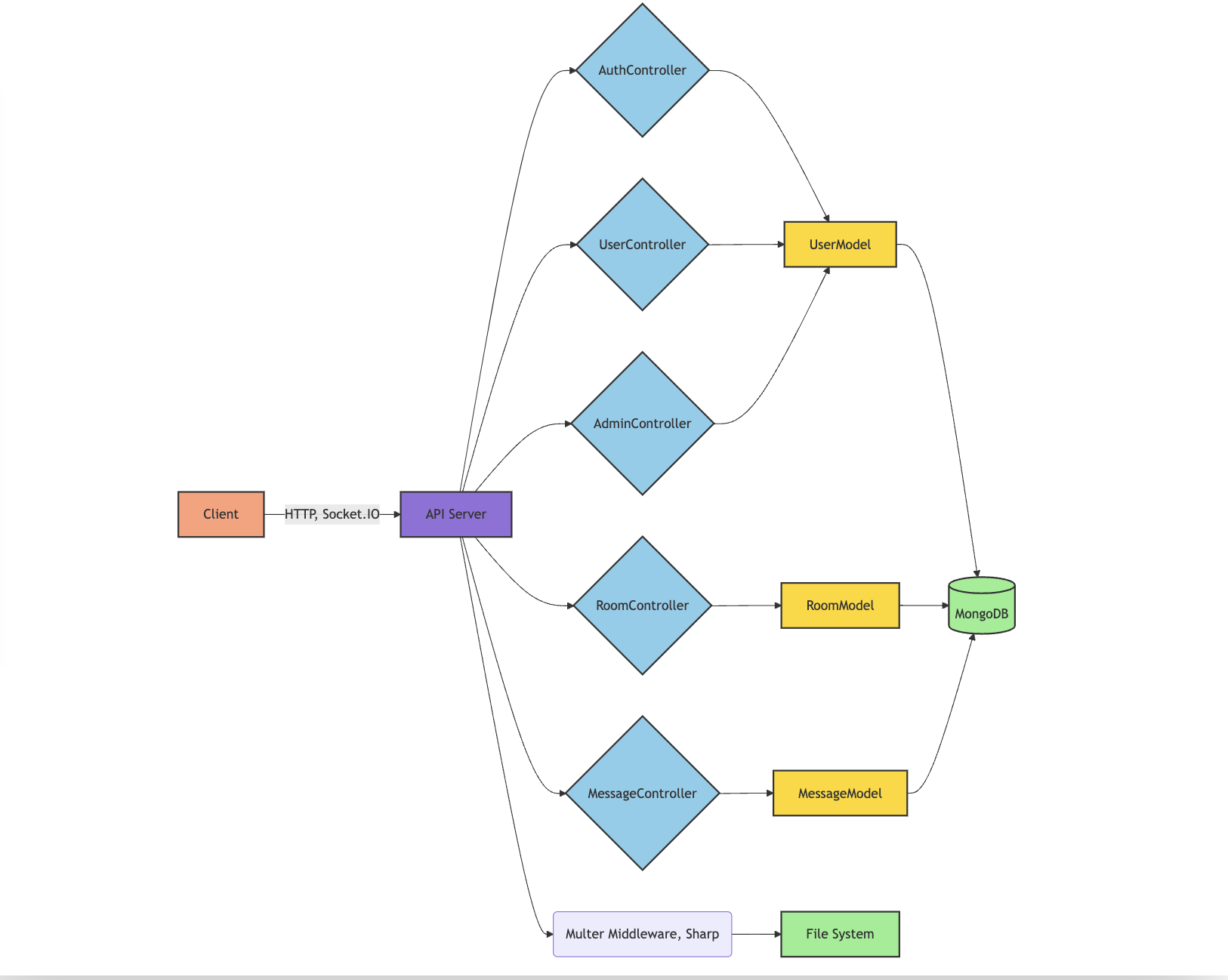


Рисунок 9 – Архитектура API сервера корпоративного мессенджера

Взаимодействие между клиентской и серверной частью приложения происходит по протоколу HTTP. Клиент отправляет запросы на определенные URL-адреса (endpoints), указывая при этом тип запроса (GET, POST, PUT, DELETE и др.), необходимые параметры и заголовки (в т.ч. JWT-токен для аутентификации). Сервер получает запрос, выполняет необходимые операции и отправляет ответ в формате JSON. В случае успеха возвращается код ответа 2xx и данные (если есть), при ошибке - код 4xx или 5xx и сообщение об ошибке. Например, при регистрации нового пользователя клиент отправляет POST-запрос на **/api/auth/register** с параметрами в теле запроса. Сервер проверяет данные через дополнительную библиотеку-валидатор «express-validator», создает нового пользователя в БД с помощью Mongoose и возвращает код 201 и сообщение об успешной регистрации.

3.3.1. Маршрутизация

API будет предоставлять следующие основные маршруты:

1. Пользователи:
   * **POST /api/auth/register** - регистрация нового пользователя.
   * **POST /api/auth/login** - авторизация пользователя
   * **GET /api/auth/me** - получение информации о текущем пользователе (требует авторизации)
   * **PATCH /api/users/update** - обновление данных пользователя (требует авторизации)
   * **PATCH /api/users/upload** - загрузка аватара пользователя (требует авторизации)
   * **GET /api/users/search?query=** - поиск пользователей по имени или фамилии
   * **GET /api/users/:userId** - получение информации о пользователе по ID (требует авторизации)
2. Комнаты чата:

* **GET /api/rooms/rooms** - получение списка комнат, в которых состоит текущий пользователь (требует авторизации)
* **GET /api/rooms/:roomId** - получение информации о комнате по ID (требует авторизации)
* **POST /api/rooms/create-room** - создание новой комнаты.
* **PATCH /api/rooms/invite** - приглашение пользователей в групповой чат (требует авторизации и права создателя комнаты).
* **PATCH /api/rooms/kickOut** - удаление пользователя из группового чата (требует авторизации и права создателя комнаты).
* **DELETE /api/rooms/leave/:roomId** - выход из комнаты (требует авторизации)
* **POST /api/rooms/archive/:roomId** - архивация чата для текущего пользователя (требует авторизации)
* **POST /api/rooms/unarchive/:roomId** - разархивация чата (требует авторизации)
* **PATCH /api/:roomId/upload** - загрузка аватара для комнаты (требует авторизации и права создателя комнаты)

1. Сообщения:

* **GET /api/messages/:roomId** - получение списка сообщений в указанной комнате.
* **POST /api/messages/create** - создание нового сообщения. Можно прикреплять файлы, изображения, голосовые сообщения.
* **DELETE /api/messages/delete/:messageId/:senderId** - удаление сообщения по ID (доступно только отправителю)
* **PATCH /api/messages/read-messages/:roomId** - пометить сообщения в комнате как прочитанные (требует авторизации)

3.3.2. Контроллеры

Контроллеры определяют логику обработки запросов от клиента. Они взаимодействуют с моделями для выполнения операций с базой данных, обрабатывают ошибки и возвращают ответы.

Спроектировано 5 основных контроллеров:

1. UserController

Методы:

* getUserbyId - получение пользователя по ID
* getUsers - получение всех пользователей
* searchUsers - поиск пользователей по имени или фамилии
* updateUser - обновление данных пользователя
* updateAvatar - загрузка и обновление аватара пользователя

1. MessageController

Методы:

* getMessages - получение списка сообщений в комнате
* createMessage - создание нового сообщения, в т.ч. с файлами.
* deleteMessage - удаление сообщения
* readMessages - пометка сообщений в комнате как прочитанные.

1. RoomController

Методы:

* getRooms - получение списка комнат пользователя
* getRoomById - получение информации о комнате по ID
* createRoom - создание новой комнаты (приватной или групповой). leaveRoom - выход пользователя из комнаты
* inviteToGroupRoom - приглашение пользователей в групповую комнату (доступно только создателю).
* kickOutOfGroup - удаление пользователя из группы (доступно только создателю).
* archive, unarchive - архивация/разархивация комнаты для текущего пользователя
* updateGroupRoomImage - обновление аватара групповой комнаты (доступно только создателю).

1. AdminController

Методы:

* unverifyUser - отмена верификации пользователя по ID.
* verifyUser - подтверждение пользователя по ID.
* updatePosition - изменение должности (position) пользователя.

1. AuthController

Методы:

* logOut - выход из системы.
* register - регистрация нового пользователя.
* login - авторизация пользователя по email и паролю.
* getMe - получение информации о текущем пользователе.

3.3.3. Middlewares

Основными промежуточными обработчиками запросов будут являться:

* **Multer** — это middleware для обработки данных, отправляемых через multipart/form-data, обычно используется для загрузки файлов на сервер. Он позволяет принимать файлы в запросах, обрабатывать их (например, сохранять на диск) и передавать информацию о файле в объект req.

Пример использования:

Принимает файлы, загружаемые пользователем, и сохраняет их на сервере. Может быть настроен для ограничения типа файлов, их размера и других параметров.

* **Sharp** — это библиотека для обработки изображений в Node.js. В сочетании с Multer она может быть использована для изменения размеров, обрезки или других преобразований изображений, сразу после их загрузки.

Пример использования:

Используется для обработки изображений, например, для их изменения (сжатие, изменение размера), чтобы они занимали меньше места на сервере или соответствовали заданным размерам. Может быть встроена для обработки файлов после загрузки с помощью Multer.

* **JWT Authentication Middleware** — этот middleware использует JSON Web Token (JWT) для аутентификации пользователей, которые обращаются к защищенным маршрутам.
* **Error Handling Middleware** — этот middleware помогает централизованно управлять ошибками и исключениями в данном приложении.

API предоставляет обширные возможности для создания современного веб-приложения чата, включая приватные и групповые чаты, обмен текстовыми сообщениями и файлами, уведомления в реальном времени. Архитектура API позволяет легко добавлять новую функциональность и масштабировать приложение при необходимости.

Дальнейшее развитие разработанного API может включать такие направления, как:

1. Добавление функции поиска по истории сообщений
2. Реализация end-to-end шифрования для повышения безопасности и приватности общения
3. Интеграция с внешними сервисами (например, возможность авторизации через OAuth-провайдеров)
4. Улучшение алгоритмов поиска и фильтрации
5. Оптимизация производительности при работе с большим количеством одновременных подключений
6. Реализация функций аналитики и сбора статистики использования приложения

Подводя итог, можно сказать, что спроектированное API является надежной основой для создания функционального и удобного в использовании веб-приложения чата. Представленные в работе подходы и решения могут быть адаптированы для многих других типов веб-приложений, требующих организации общения пользователей в реальном времени.

3.4. Структура программных единиц

Клиентская часть приложения использует фреймворк React и TypeScript. Разработанная структура директорий и файлов представлена на рисунке 10:

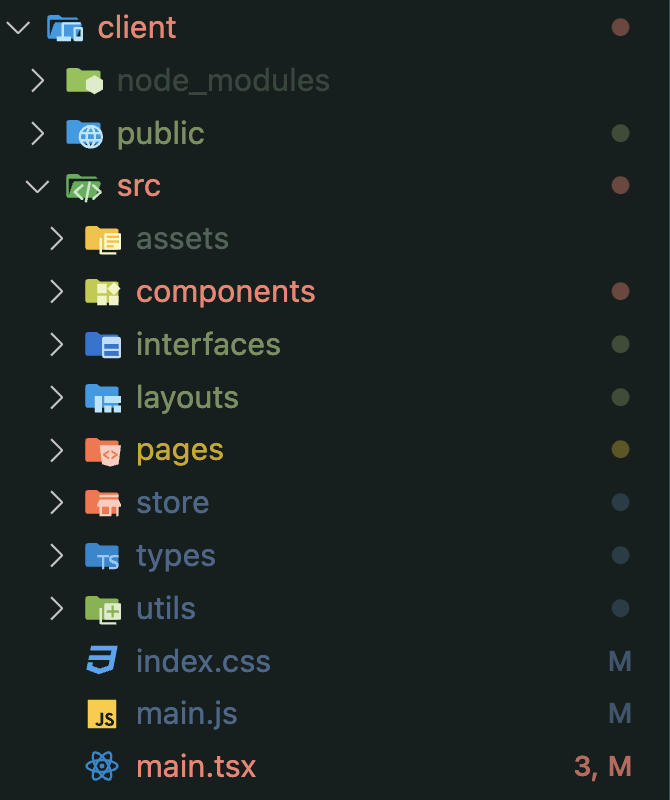


Рисунок 10 – структура директорий и файлов на клиенте

Основные папки содержат следующие файлы:

1. components – хранит в себе переиспользуемые компоненты, которые в целом составляют интерфейс приложения.

* **Auth**/ - компоненты для авторизации и регистрации
* **Avatar**/ - компонент для отображения и загрузки аватара пользователя
* **CreateMessage**/ - компонент для создания нового сообщения
* **CreateRoom**/ - компонент для создания новой комнаты
* **Modal**/ - компонент модального окна
* **MessageItem**/ - компонент одного сообщения в списке
* **MessageList**/ - компонент списка сообщений
* **Navigation**/ - компонент навигации по приложению
* **RoomItem**/ - компонент одной комнаты в списке
* **RoomList**/ - компонент списка комнат
* **RoomData**/ - компонент для отображения данных о комнате
* **Sidebar**/ - компонент боковой панели с списком комнат и навигацией
* **ChangeAvatar**/ - компонент для смены аватара
* **VideoConference**/ - компонент для проведения видеоконференций

1. interfaces – содержит интерфейсы TypeScript для основных сущностей приложения:

* **Message.ts** – описание интерфейса сообщения
* **IRoom.ts** - описание интерфейса комнаты
* **IUser.ts** - описание интерфейса пользователя

1. layouts – хранит шаблоны страниц приложения:

* **AdminLayout.tsx** - шаблон для страниц административной панели
* **AuthLayout.tsx** - шаблон для страниц авторизации и регистрации
* **ChatLayout.tsx** - шаблон для страниц чата

1. pages – содержит компоненты страниц приложения:

* **AdminPage.tsx** - страница административной панели
* **AuthPage.tsx** - страница авторизации и регистрации
* **RoomPage.tsx** - страница конкретной комнаты чата

1. store – хранит начальное состояние приложения и отдельные файлы для управления состоянием и работой с сервером:

* **authSlice.ts** – управление состоянием авторизации
* **modalSlice.ts** – управление состоянием модальных окон
* **roomSlice.ts** – управление состоянием комнат с помощью Redux
* **messageSlice.ts** – управление состоянием для сообщений
* **userSlice.ts** – управление состоянием для пользователей
* **store.ts** – конфигурация и инициализация Redux store

1. types – содержит глобальные типы данных, используемые в приложении для строгой типизации отдельных переменных или функций.
2. Utils – хранит вспомогательные функции для работы с сокетами, датами и темой приложения.

* **socketUtils.ts** - функции для инициализации и работы с Socket.IO
* **convertDate.ts** - функции для работы с датами
* **themesUtils.ts** - функции для работы со светлой/темной темой приложения

1. **main.tsx** - главный файл приложения с настройкой роутинга.

Серверная часть приложения использует Node.js и фреймворк Express. Структура директорий и файлов представлена на рисунке 11:

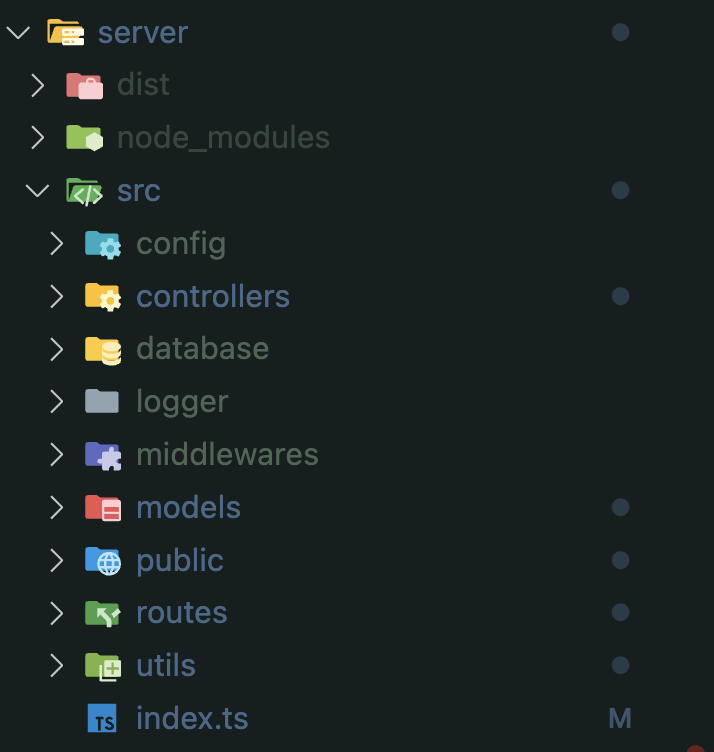


Рисунок 11 – структура директорий и файлов на сервере

1. config: содержит файл .env с переменными окружения.
2. controllers: хранит контроллеры Express, которые обрабатывают запросы, взаимодействуют с базой данных и отправляют ответы.
3. **AdminControllers/**

* **verifyUser.ts** –верифицирует пользователя в приложении
* **unverifyUser.ts** – блокирует пользователя в приложении
* **updatePosition.ts** – обновляет должность пользователя

1. **AuthControllers/**

* **logout.ts** –выход из приложения
* **register.ts** – регистрация в приложении
* **login.ts** –вход в приложение
* **getMe.ts** – запрос на получение своего профиля

1. **MessageControllers/**

* **getMessages.ts** - получение списка сообщений конкретного чата / комнаты
* **createMessage.ts** – запрос на создание сообщения
* **deleteMessage.ts** – запрос на удаление сообщения
* **readMessages.ts** – обработчик прочитанных сообщений

1. **RoomControllers/**

* **getRooms.ts** – получение списка всех комнат текущего пользователя
* **getRoomById.ts** – получение конкретной комнаты по идентификатору
* **createRoom.ts** – создание новой комнаты / чата
* **leaveRoom.ts** –выход из чата
* **inviteToGroupRoom.ts** – обработчик приглашения в групповой чат
* **kickOutOfGroup.ts** – обработчик исключения из группового чата
* **archive.ts –** перемещение чата в архив
* **unarchive.ts –** изъятие чата из архива
* **updateGroupRoomImage.ts** –обновление аватара группового чата

1. **UserControllers/**

* **getUserbyId.ts**
* **getUsers.ts** –получение конкретного пользователя по идентификатору
* **searchUsers.ts** – обработчик поиска пользователей по ФИО.
* **updateUser.ts** – обновление данных пользователей
* **updateAvatar.ts** – обновление фото профиля пользователя
* **updatePassword.ts** – обновление пароля пользователя

1. models: содержит модели данных MongoDB, описывают структуру и схемы сущностей БД.

* **User.ts**
* **Message.ts**
* **Room.ts**

1. routes: хранит маршруты Express, которые определяют конечные точки API и связывают их с соответствующими контроллерами.

* **AdminRoutes.ts**
* **AuthRoutes.ts**
* **MessageRoutes.ts**
* **RoomRoutes.ts**
* **UserRoutes.ts**

1. middlewares: содержит промежуточные обработчики Express, такие как middleware для проверки JWT токена и обработки ошибок на сервере.

* **Auth.ts**
* **errorMiddleware.ts**

1. utils: хранит вспомогательные функции и настройки для работы с Multer (загрузка файлов), sharp (обработка изображений) и Socket.IO (взаимодействие пользователей в реальном времени).

* **multer.ts**
* **sharp.ts**
* **socketUtils.ts**

1. public: содержит директории для хранения загруженных файлов - аватаров пользователей, файлов, изображений и голосовых сообщений.

* **/avatars**
* **/files**
* **/group\_avatars**
* **/message\_images**
* **/voices**

Такая структура организует код приложения по принципу разделения ответственности, что облегчает его поддержку и масштабирование. Клиентская часть разбита на компоненты, страницы, шаблоны и управляется централизованным хранилищем данных. Серверная часть включает контроллеры, модели, маршруты, middleware и утилиты для работы с файлами. Использование языка TypeScript на клиенте и сервере повышает контроль типов и снижает вероятность ошибок на этапе компиляции.

**4. Реализация**

В данном разделе будет подробно изложен процесс разработки корпоративного мессенджера, описаны его ключевые функции, приведены фрагменты кода и скриншоты.

Так как корпоративный мессенджер – это высоконагруженное приложение, особое внимание уделяется последовательности загрузки данных, взаимодействию с сервером через Socket.IO, управлению состоянием с помощью Redux Toolkit, а также оптимизации производительности интерфейса с использованием таких технологий, как предзагрузка и «мемоизация» компонентов.

4.1.1. Процесс авторизации и переход на главную страницу чата

После успешной авторизации пользователя происходит переход на главную страницу чата.

Этот процесс включает несколько ключевых шагов:

* Инициализация соединения с сервером через Socket.IO: В корневом компоненте **ChatLayout** осуществляется подключение к серверу через библиотеку **socket.io-client**.
* Это соединение обеспечивает возможность обмена данными в реальном времени между клиентом и сервером.

Загрузка списка комнат/чатов:

* Клиент отправляет запрос на сервер для получения списка доступных комнат/чатов пользователя.
* Сервер обрабатывает этот запрос, извлекает информацию из базы данных и возвращает клиенту список комнат, включая идентификаторы, названия и информацию о последних сообщениях в каждом из чатов.
* Полученные данные сохраняются в глобальном состоянии Redux Toolkit, что позволяет отображать список комнат в интерфейсе пользователя.

Отображение плейсхолдера для чата:

* На главной странице отображается список комнат/чатов, а также плейсхолдер с надписью «Выберите чат».
* Плейсхолдер является статическим элементом, который не требует загрузки дополнительных данных до выбора конкретного чата.

4.1.2. Загрузка и отображение сообщений при выборе чата

При выборе конкретного чата пользователем происходит загрузка последних сообщений и подготовка интерфейса для отображения диалога:

1. Запрос на получение сообщений:

* При клике на определенный чат клиент отправляет запрос на сервер для получения последних 10 сообщений из этого чата.
* Сервер извлекает соответствующие сообщения из базы данных и отправляет их клиенту.
* Полученные сообщения сохраняются в глобальном состоянии Redux Toolkit.

1. Отображение сообщений с предзагрузкой:

* При получении сообщений проверяется тип каждого сообщения (текстовое, файл/изображение или голосовое).
* Для сообщений, содержащих изображения или голосовые сообщения, отображается предзагрузка:
* **Изображения:** Применяется эффект блюра (размытия) до полной загрузки изображения. Это достигается с помощью CSS-стилей и состояния загрузки изображения. После полной загрузки изображение заменяет блюр, обеспечивая плавный переход.
* **Голосовые сообщения/файлы:** Отображаются индикаторы загрузки, указывающие на процесс загрузки аудиофайла.

1. Использование мемоизации для оптимизации производительности: Мемоизация — это техника программирования, которая используется для оптимизации производительности за счёт запоминания результатов ранее выполненных вычислений. В контексте React она позволяет избежать повторного выполнения ресурсоёмких операций, таких как рендеринг компонентов, пересчёт сложных значений или переопределение функций, если входные данные не изменились.

Преимущества мемоизации:

* Снижение нагрузки на процессор: повторные вычисления заменяются на использование ранее сохранённого результата.
* Улучшение производительности: уменьшение количества рендеров в React-приложениях.
* Повышение отзывчивости интерфейса: ускоряются сложные вычисления или взаимодействия.

Когда применять мемоизацию:

* При работе с компонентами, которые получают неизменные данные и часто перерисовываются.
* При выполнении сложных вычислений, зависящих от входных параметров (пропсов) или состояния.
* Для предотвращения лишних перерисовок в глубоких структурах компонентов.

На рисунке представлен пример использования мемоизации в компоненте **MessageItem. React.memo** — это специальная функция в библиотеке React, которая используется для оптимизации производительности функциональных компонентов. Она предотвращает повторный рендеринг компонента, если его свойства не изменились.

Необходимость использования **React.memo** в **MessageItem**:

* Предотвращение лишних рендеров: Компонент **MessageItem** перерисовывается для каждого сообщения в списке. При изменении состояния или свойств в родительском компоненте весь список сообщений может быть пересоздан, что приводит к повторному рендерингу даже тех сообщений, которые не изменились. **memo** позволяет избежать этого, сравнивая текущие и предыдущие входные данные. Если входные данные не изменились, компонент не перерисовывается.
* Стабильность отображения: В мессенджере каждое сообщение содержит динамический контент, такой как текст, изображения, голосовые сообщения и статус доставки. Эти данные меняются редко, поэтому повторный рендер из-за изменений в других частях приложения не нужен.
* Оптимизация производительности: Когда на экране одновременно отображаются десятки или сотни сообщений, повторные рендеры без видимой причины могут замедлять работу интерфейса. Использование memo минимизирует нагрузку на процессор и ускоряет работу приложения.

1. Обработка подгрузки дополнительных сообщений:

* В интерфейсе диалога присутствует кнопка «Загрузить еще», которая позволяет пользователю подгружать предыдущие сообщения
* При нажатии на кнопку отправляется запрос на сервер для получения следующей партии сообщений (предыдущие 10 сообщений). Сервер возвращает дополнительные сообщения, которые добавляются в глобальное состояние Redux и отображаются в интерфейсе пользователя.

4.1.3. Обработка изображений с разными разрешениями

В приложении предусмотрена поддержка отправки изображений в двух разрешениях:

* Низкое разрешение для отображения в чате: При отправке изображения сервер генерирует уменьшенную версию изображения, оптимизированную для быстрого отображения в интерфейсе чата. Это позволяет ускорить загрузку сообщений и снизить объем передаваемых данных.
* Высокое разрешение для отображения в модальном окне: При клике на изображение в чате открывается модальное окно, где отображается изображение в полном размере. Для этого загружается изображение высокого разрешения, обеспечивая пользователю возможность рассмотреть изображение детальнее.

Реализация работы приложения при старте включает последовательную и оптимизированную загрузку данных, использование Socket.IO для обеспечения взаимодействия в реальном времени, эффективное управление состоянием через Redux Toolkit и применение методов оптимизации производительности. Вместе эти компоненты обеспечивают быстрый отклик интерфейса, актуальность отображаемых данных и удобство использования мессенджера для конечных пользователей. Применение индикаторов загрузки, мемоизации и частичной подгрузки данных позволяет улучшить пользовательский опыт и ускорить работу корпоративного мессенджера.

4.2. Реализация взаимодействия клиента и сервера с помощью REST API.

В данном случае необходимо рассмотреть все шаги взаимодействия клиента и сервера посредством REST API на конкретном примере. В целом, их работа будет похожа на всех этапах реализации мессенджера.

На клиенте, с помощью библиотеки React создается функциональный компонент RoomPage, который будет отображать окно чата.

В первую очередь необходимо получить данные текущего чата, если он существует. При каждом открытии любого из чатов, с помощью хука жизненного цикла **useEffect**, который позволяет нам получать какую-либо информацию с сервера и выполнять другие побочные операции, вызывается функция **getRoomById** из централизованного хранилища Redux, которая обращается к серверу посредством функции «createAsyncThunk». Это функция из библиотеки Redux Toolkit, которая позволяет создавать асинхронные действия для Redux.

Асинхронные действия – это действия, которые могут выполнять операции, например, такие как запросы к серверу, минуя основной поток данных приложения, не мешая ему выполняться, что и необходимо. Посредством библиотеки **Axios** создается запрос на сервер, методом **get** по конкретному адресу, который прописан в API и дополнительным параметром в виде идентификатора комнаты/чата. В результате выполнения либо возвращаются данные, которые будут обработаны дальше или произойдет выброс ошибки, которая будет сразу отображена пользователю.

Код клиента и хранилища представлены на рисунках 12 и 13.



Рисунок 12 – этап получения данных комнаты на клиенте

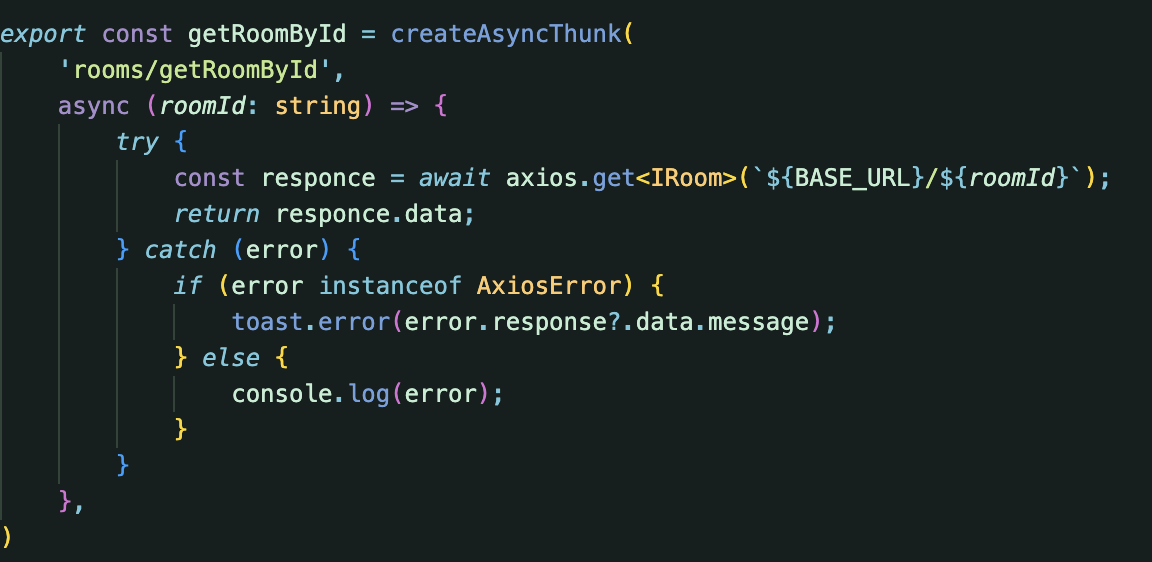


Рисунок 13 – код обращения к API сервера посредством createAsyncThunk

Далее, запрос попадает на сервер, по конкретному адресу, который был задан в маршрутизаторе Express по методу **get**. Пример на рисунке.  
Первый параметр метода: адрес по которому производится запрос, второй (необязательный) – самописная функция промежуточного ПО «authMiddleware», которая выполняется перед основным запросом и проверяет, авторизован ли пользователь, который его совершает. Если нет – выбрасывает ошибку авторизации. Третий параметр – это контроллер, который выполняет основную бизнес - логику: извлекает идентификатор чата/комнаты, проверяет существует ли она, обращается к модели Room и делает запрос к базе данных, где c помощью метода «populate», который используется для заполнения ссылочных полей в документе данными из других коллекций, получает конечный результат и отправляет его обратно на клиент в формате JSON.

Пример роута и контроллера на рисунках 14 и 15.



Рисунок 14 – код маршрутизатора Express



Рисунок 15 – код контроллера getRoomById

Следущим шагом необходимо отрисовать данные комнаты на клиенте. Получив от сервера результат, он снова попадает в хранилище Redux Toolkit.

Для того, чтобы данные отображались, необходимо задать начальное состояние (state) Room, которое по умолчанию равно null. С помощью встроенного метода «extraReducers» можно контролировать загрузку данных и визуально ее отображать, записывать результат в состояние, обрабатывать ошибки и показывать их конечному пользователю. Код представлен на рисунке 16.

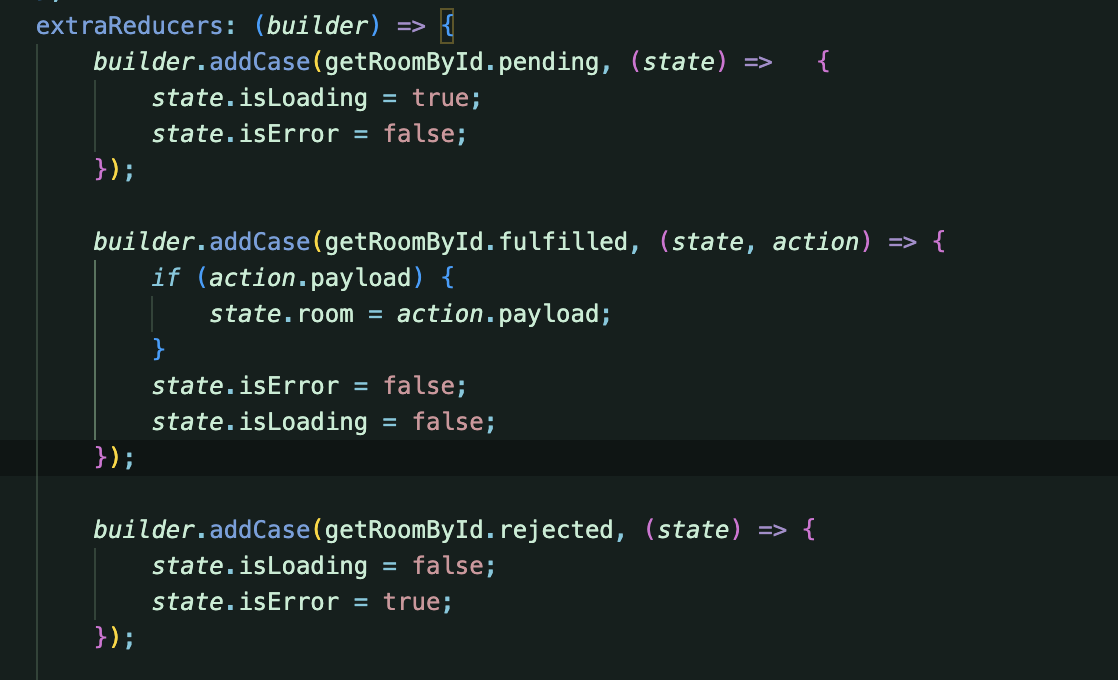


Рисунок 16 – код записи данных c сервера в состояние Redux Toolkit

После того, как данные успешно получены и записаны в state, их нужно получить в компоненте RoomPage, для дальнейшего отображения. В первую очередь мы обращаемся к хранилищу и извлекаем оттуда текущее состояние комнаты с помощью встроенной функции **useSelector**:

const { room } = useSelector((*state*: RootState) => *state*.rooms);

Далее, имея конечные данные, их можно отобразить. В этом поможет React, с помощью JSX – расширения языка Javascript, который необходим для отображения UI. Полученные данные можно передавать в любые компоненты (в данном случае в дочерние), отображать их по определенному условию (например, типу комнаты), а также извлекать только какую-то часть из них, так что полученные данные могут существовать в абсолютно любой части интерфейса приложения. Код для отображения представлен на рисунке 17.



Рисунок 17 – Код отображения интерфейса в компоненте RoomPage.

4.3. Реализация работы приложения с помощью socketIO

В данном разделе рассматривается использование библиотеки Socket.IO для взаимодействия между клиентом и сервером в режиме реального времени. В корпоративном мессенджере Socket.IO позволяет реализовать различные функциональные возможности, такие как обновление статусов пользователей, отправка сообщений, звуковые уведомления, отметки о прочтении и приглашения в чаты. Работа структурирована вокруг событий, которые обрабатываются как на клиентской, так и на серверной стороне. Для управления состоянием интерфейса используется библиотека Redux Toolkit, что обеспечивает эффективное обновление UI в ответ на события Socket.IO.

4.2.1 Архитектура взаимодействия

Socket.IO функционирует на основе событийной модели, где клиент и сервер обмениваются сообщениями через определенные события. Каждое событие имеет уникальное название и может передавать данные в виде параметров. В приложении мессенджера Socket.IO используется для следующих ключевых событий:

* Статусы пользователей: «user-online», «user-offline»
* Вход и выход из чата: «join-room», «leave-room»
* Отправка, получение/прочтение сообщений: «create-message», «is-received»
* Звуковые уведомления: «sound-notification»
* Приглашение, удаление и выход из чата: «invite-to-room», «kick-from-group», «leave-from-group»
* Событие «Пользователь печатает»: «set-typing»
* Верификация пользователей в админ-панели: «verify-user»

Общая схема работы Socket.IO в приложении выглядит следующим образом:

* С клиента приходит запрос в БД на обновление каких-либо данных (удаление пользователя из группового чата, новое сообщение и пр.),
* После успешного обновления БД, с сервера на клиент отправляется Socket.IO событие, с помощью метода **emit** (рисунок 18), где первым параметром передается название события, а вторым - данные, которые необходимы для дальнейшего использования (идентификаторы пользователей, данные чатов, сообщений), например, для их отрисовки.
* Далее происходит инициализация текущего события на клиенте с полученными данными с помощью метода **on** (Рисунок 19)
* Следующим шагом, с помощью метода **dispatch** из библиотекиRedux Toolkit, происходит обращение к централизованному хранилищу приложения, где находятся различные функции управления состоянием, которые необходимы для дальнейшей перерисовки UI у определенных пользователей, если это необходимо (рисунок 19 и 20).
* Финальной операцией является отрисовка обновленного состояния у пользователей (рисунок 21).

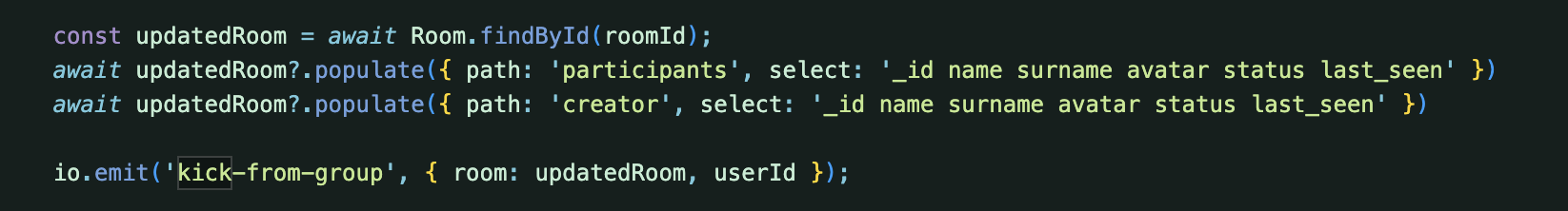


Рисунок 18 – отправка Socket.IO события на клиент с помощью метода **emit**



Рисунок 19 – обработка Socket.IO события на клиенте с помощью метода **on**



Рисунок 20 – функции управления состоянием в Redux Toolkit



Рисунок 21 – отрисовка нового состояния в компоненте React

* 1. Реализация видеочата с помощью WebRTC

Видеочат является одним из трех типов чатов (приватные, групповые и видеочаты), доступных в приложении. Главным инструментом его реализации является использование WebRTC для передачи аудио- и видеоданных между участниками, а также Socket.IO для обмена сигналами между клиентами для установки соединения. В данном разделе будут рассмотрены ключевые этапы работы компонента **VideoConference**.

Основные этапы работы видеочата:

1. Инициализация:

* Создание локального видео и подключение камеры и микрофона. Установка соединения между участниками чата через WebRTC.

2. Обмен сигналами через Socket.IO:

* Передача информации о предложении (offer), ответе (answer) и ICE-кандидатах между клиентами через сервер Socket.IO.

3. Отображение видео:

* Локальное видео отображается пользователю через тег <video>.
* Видеопотоки других участников отображаются в динамически создаваемых тегах <video>.

4. Управление ресурсами:

* Остановка видеопотока и очистка соединений при завершении сессии.

Разделение ролей: сервер и клиент

Серверная часть

Код сервера (Socket.IO) отвечает за маршрутизацию сообщений, передавая сигналы WebRTC от одного клиента к другому (рисунок 22):

* «offer»: Сервер передает предложение участника остальным участникам комнаты roomId.
* «answer»: Сервер пересылает ответ участника отправителю предложения.
* «signal»: Сервер пересылает ICE-кандидаты между участниками

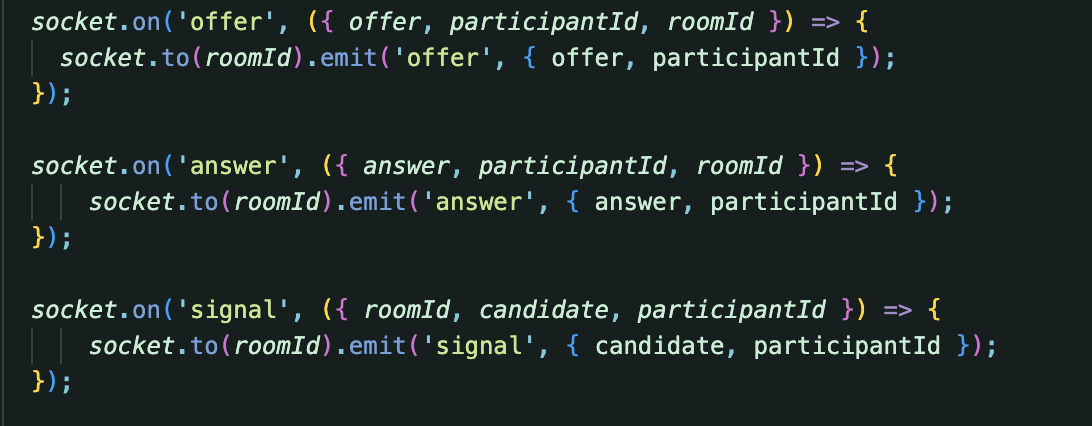


Рисунок 22 – обмен сигналами WebRTC от одного клиента к другому с помощью Socket.IO

Клиентская часть

На стороне клиента реализована основная логика видеочата:

1. Инициализация локального видео (рисунок 23):

* С помощью встроенного в Javascript - объекта **navigator.mediaDevices.getUserMedia** получаем поток с камеры и микрофона. В случае разрешения пользователем на использование камеры возвращается объект **MediaStream**.
* Устанавливаем его в **localVideoRef** для отображения в чате.

2. Создание и управление **RTCPeerConnection** (Рисунок 24):

* Для каждого участника создается экземпляр **RTCPeerConnection**.
* Поток текущего пользователя добавляется в **PeerConnection** с помощью **addTrack**.
* Обрабатываются события:
  + **onicecandidate**: Отправка ICE-кандидатов через Socket.IO.
  + **ontrack**: Отображение видеопотока другого участника.



Рисунок 23 – инициализация локального видео

****

Рисунок 24 – создание и управление RTCPeerConnection

3. Обмен сигналами

Обмен сигналами — это ключевая часть работы WebRTC, необходимая для установления соединения между участниками видеочата

Этапы обмена сигналами:

1. Обработка предложений (Offer). Когда один из участников инициирует видеочат, он создает SDP-описание (Session Description Protocol), содержащее информацию о своем медиапотоке, и отправляет его другим участникам через Socket.IO. Код представлен на рисунке 25.  
Методы:

* **setRemoteDescription**: Сохраняем предложение (SDP) другого участника.
* **createAnswer**: Создаем ответ, чтобы согласовать параметры соединения.
* **setLocalDescription**: Устанавливаем собственное SDP-описание для отправки. socket.emit(«answer»): Пересылаем ответ через сервер Socket.IO.

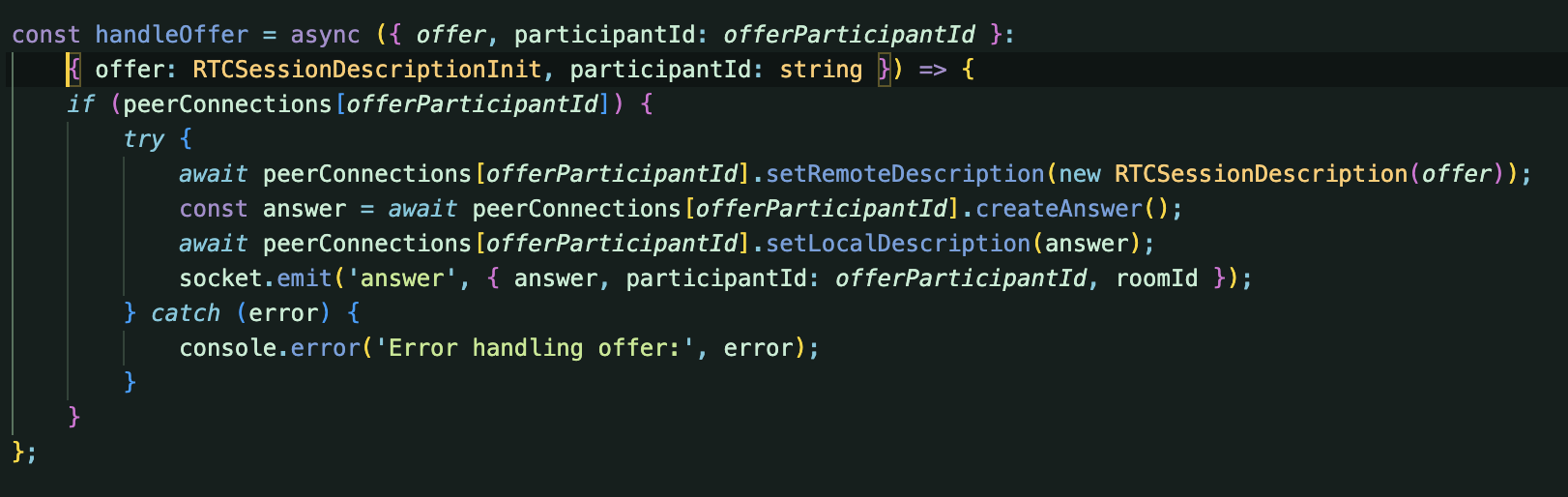


Рисунок 25 – код обработка предложений (Offer)

2. Обработка ответов (Answer)

Когда инициатор видеочата получает ответ, он использует его для завершения настройки соединения. Код на рисунке 26.

Методы:

* **setRemoteDescription**: Завершаем установку соединения, принимая ответ от другого участника.

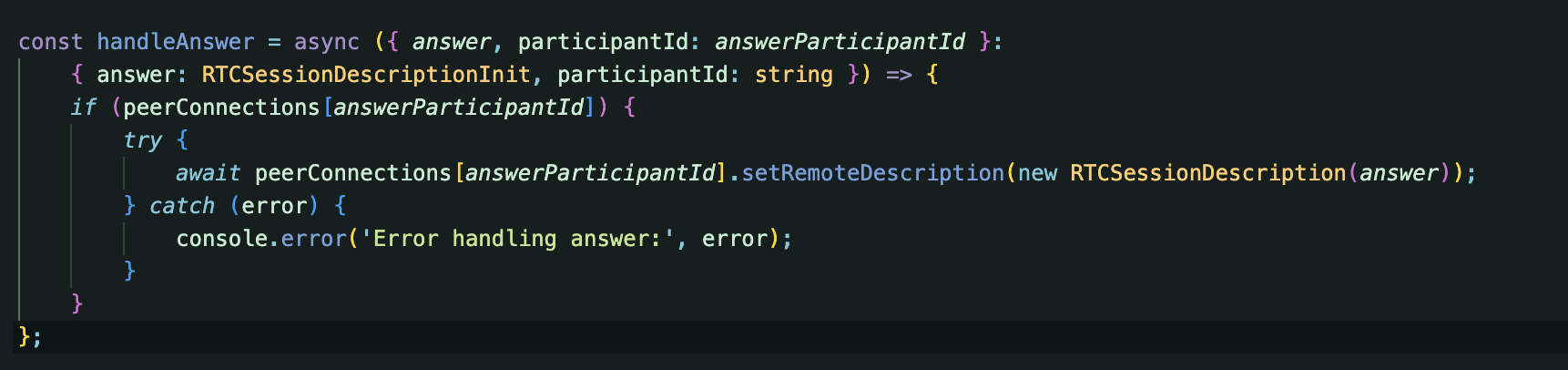
****

Рисунок 26 – код обработки ответов (Answer)

3. Обработка ICE-кандидатов

Для эффективной передачи данных WebRTC использует ICE-кандидаты (Interactive Connectivity Establishment), которые содержат информацию о сетевых маршрутах. Эти данные отправляются через сервер Socket.IO.

На отправляющей стороне: Каждый раз, когда генерируется новый ICE-кандидат, он отправляется другим участникам (рисунок 27):

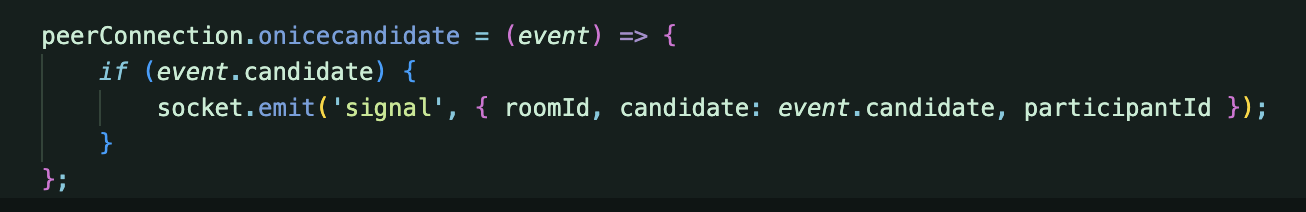
****

Рисунок 27 – отправка ICE-кандидата другим участникам через Socket.IO

На принимающей стороне: Когда клиент получает ICE-кандидат, он добавляет его в **RTCPeerConnection** (рисунок 28):

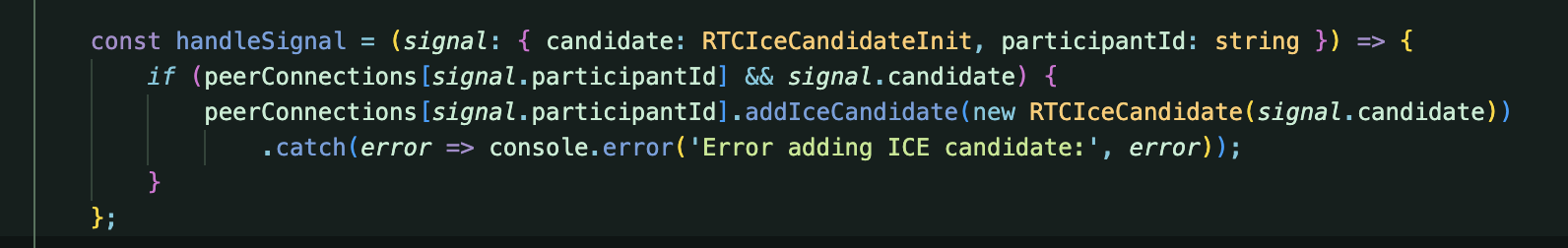
****

Рисунок 28 – добавление ICE-кандидата в RTCPeerConnection

Методы:

* **onicecandidate**: Событие генерируется, когда WebRTC обнаруживает новый сетевой маршрут.
* **addIceCandidate**: Добавляем полученного ICE-кандидата, чтобы обеспечить корректное соединение между участниками.

4. Рендеринг интерфейса

Рендеринг интерфейса видеочата отвечает за отображение локального видео (с камеры текущего пользователя) и видеопотоков других участников. Код представлен на рисунке 29.

Рендеринг реализован таким образом, чтобы:

* Локальное видео отображалось сразу после получения потока с камеры и микрофона.
* Видеопотоки участников автоматически добавлялись в интерфейс при их появлении.
* Удобно стилизовать и масштабировать интерфейс для любого количества участников.

Список атрибутов и функций:

* **autoPlay**: Автоматически запускает воспроизведение видео.
* **muted**: Отключает звук, чтобы пользователь не слышал собственное эхо.
* Каждый участник видеочата отображается в отдельном <video> элементе. Эти элементы создаются динамически для каждого участника с использованием функции **map**.
* **participants**: Список участников видеочата.
* **filter(id => id !== user?.\_id):** Убираем текущего пользователя из списка участников, чтобы его видео не отображалось дважды.
* **ref**: Сохраняем ссылки на <video> элементы в объекте **remoteVideosRef**, чтобы в будущем можно было обновлять их поток.
* **ref.srcObject**: Привязываем поток **MediaStream** удаленного участника, если он уже доступен.

****

Рисунок 29 – код отображения видеочата

Текущая реализация видеочата с использованием WebRTC, React и Socket.IO представляет собой современное, функциональное и производительное решение для организации видеоконференций, обеспечивая взаимодействие пользователей в реальном времени.

4.5. Работа с изображениями и файлами с помощью multer и sharp

Полноценная работа корпоративного мессенджера невозможна без обмена файлами, изображениями и прочим контентом. В этой главе будет рассмотрен подход к обработке изображений и файлов в веб-приложении с использованием библиотек **Multer** и **Sharp**. Данные инструменты обеспечивают эффективную загрузку, фильтрацию и обработку медиаконтента.

4.5.1. Подготовка к обработке файлов

Для начала была настроена библиотека **Multer**, которая используется для обработки входящих файлов. Она предоставляет гибкость в выборе места хранения файлов и позволяет выполнить фильтрацию входящих форматов изображений/файлов перед их обработкой. В данном случае применен метод memoryStorage(), позволяющий сохранять файлы в оперативной памяти, что ускоряет последующую обработку. Код инициализации на рисунке 30.

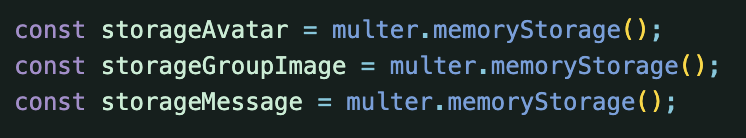


Рисунок 30 – инициализация хранилища библиотеки Multer

4.5.2. Фильтрация форматов файлов

Для предотвращения загрузки неподдерживаемых форматов созданы функции-фильтры, проверяющие тип файла. Например, для сообщений поддерживаются изображения, текстовые файлы, аудио (для голосовых сообщений) и документы (рисунок 31)

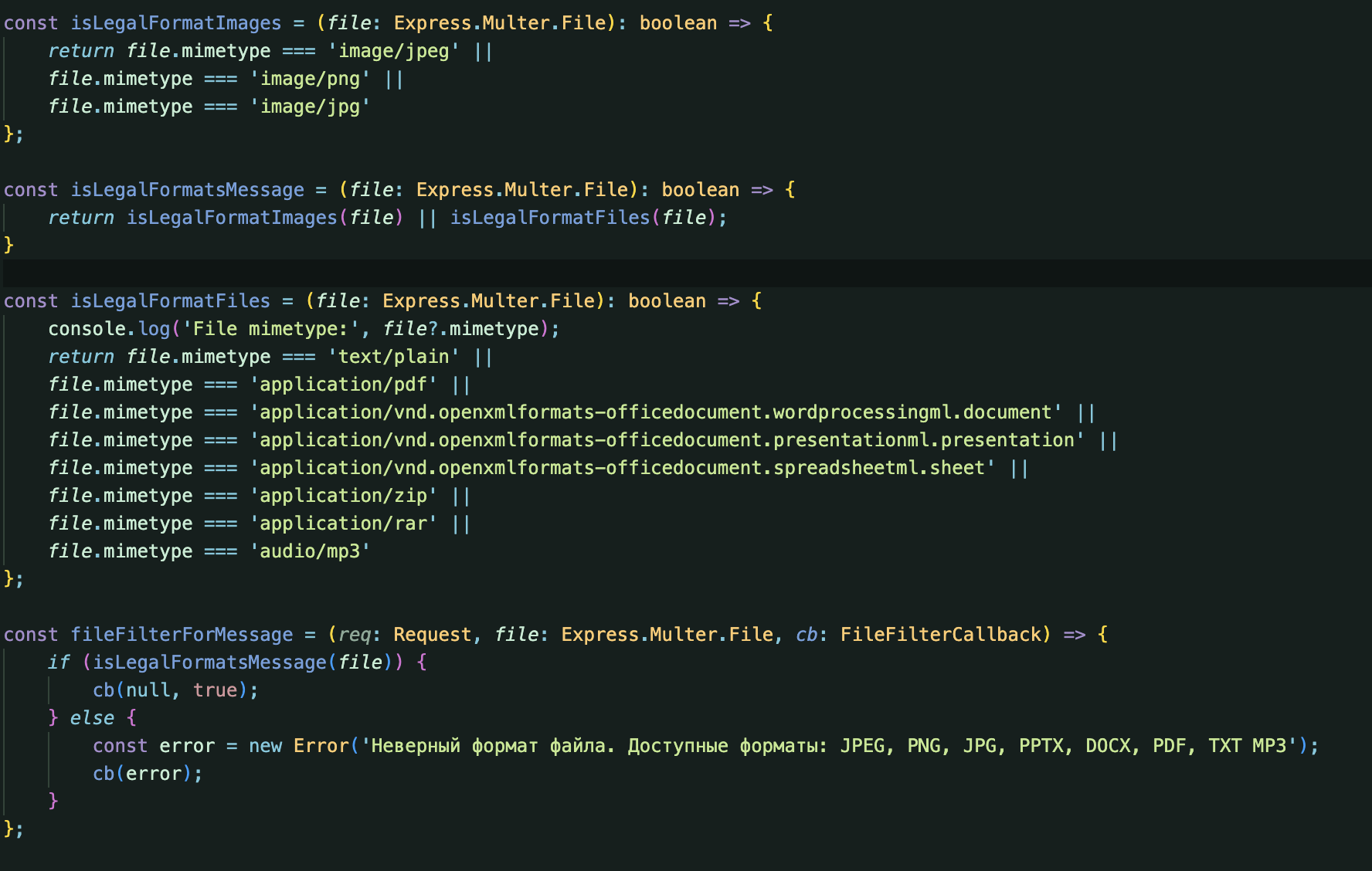


Рисунок 31 – фильтрация форматов файлов

4.5.3. Обработка изображений с Sharp

После успешной загрузки и проверки формата начинается обработка изображений с использованием библиотеки **Sharp**. Эта библиотека позволяет изменять размеры, формат и качество изображений. Например, при загрузке аватара, файл преобразуется в изображение формата JPEG размером 150x150 пикселей, стопроцентным качеством и сохраняется по определенному пути с помощью соответствующих методов у объекта sharp (рисунок 32).



Рисунок 32 – Обработка изображений с Sharp

4.5.4. Работа с другими типами файлов

Не все файлы являются изображениями. Для текстовых документов, архивов и аудиофайлов в примере используется сохранение файла в папке с помощью встроенного в nodeJS модуля: **fs.writeFile** (рисунок 33).

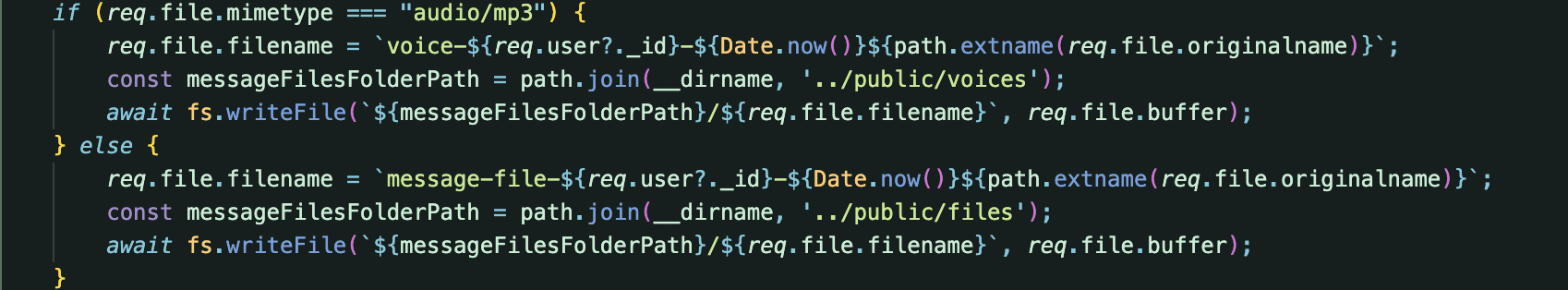


Рисунок 33 – сохранение прочих файлов мессенджера с помощью модуля fs

4.5.5. Интеграция с Multer

Для каждого типа файлов был создан отдельный загрузчик, объединяющий хранилище и фильтр. Выглядит это следующим образом на рисунке 34.



Рисунок 34 – загрузчик объекта Multer

4.5.6. Интеграция загрузчиков с роутами

Данные загрузчики и функции, использующие **Multer** и **Sharp**, интегрируются в виде middleware в маршруты. Они работают перед выполнением контроллеров, обеспечивая передачу уже обработанных данных. Пример кода на рисунке 35.

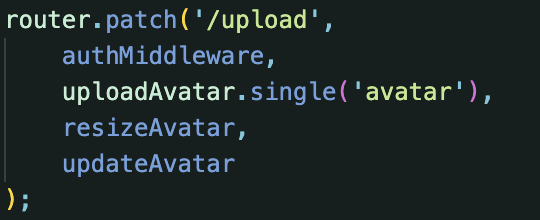


Рисунок 35 – пример интеграции middlware для загрузки файлов в роуты Express

Использование библиотек **Multer** и **Sharp** позволяет гибко обрабатывать и хранить изображения и файлы в веб-приложении. Такой подход обеспечивает безопасность и удобство работы с контентом. Разделение обработки изображений, документов и аудиофайлов повышает структурированность кода и улучшает масштабируемость корпоративного мессенджера.

Итог:

В данном разделе была подробно рассмотрена реализация корпоративного мессенджера, описаны его ключевые функции и технологии. Особое внимание было уделено оптимизации загрузки контента, взаимодействию клиента и сервера через REST API и Socket.IO, реализации видеочата с помощью WebRTC, а также работе с файлами и изображениями с использованием сторонних библиотек.

**Заключение**

В рамках данной практики был разработан корпоративный мессенджер, который обеспечивает эффективное и безопасное общение между сотрудниками в IT-компаниях.

Реализованное решение отвечает современным требованиям к корпоративным коммуникационным системам и готово к внедрению в реальную производственную среду.

Стек технологий, выбранный для реализации мессенджера, включает в себя передовые инструменты и фреймворки, которые обеспечивают высокую производительность, масштабируемость и удобство разработки. На клиентской стороне использовались TypeScript, React.js, Redux Toolkit и WebRTC, что позволило создать интерактивный и отзывчивый пользовательский интерфейс с поддержкой видеоконференций. На серверной стороне применялись Node.js, Express, MongoDB и WebSocket, обеспечивающие надежную и масштабируемую backend-инфраструктуру для обработки запросов и взаимодействия с базой данных в реальном времени.

Архитектура мессенджера построена на принципах разделения ответственности и модульности. Клиентская часть следует компонентному подходу React, где интерфейс разбит на переиспользуемые компоненты, управляемые централизованным хранилищем состояния Redux. Серверная часть реализована по принципу MVC (Model-View-Controller) с разделением на модели, контроллеры и маршруты. Взаимодействие между клиентом и сервером осуществляется посредством REST API и WebSocket, что обеспечивает гибкость и возможность расширения функционала в будущем.

Разработанный мессенджер обладает широким набором функций, включая приватные и групповые чаты, обмен текстовыми сообщениями, файлами и медиаконтентом, видеоконференции, уведомления в реальном времени, управление пользователями и их статусами. Решение учитывает вопросы безопасности, предоставляя функции аутентификации и авторизации пользователей, проверки целостности данных и защиты от несанкционированного доступа.

В процессе реализации были применены лучшие практики разработки, такие как использование TypeScript для строгой типизации, модульное тестирование, обработка ошибок, логирование и документирование кода. Структура проекта организована таким образом, чтобы облегчить поддержку, масштабирование и дальнейшее развитие приложения.

Готовый корпоративный мессенджер был успешно протестирован и продемонстрировал стабильную работу и соответствие заявленным требованиям. Решение готово к внедрению в IT-компаниях и может значительно повысить эффективность внутренних коммуникаций, улучшить взаимодействие между сотрудниками и ускорить рабочие процессы.

В дальнейшем развитии мессенджера можно рассмотреть такие направления, как интеграция с другими корпоративными системами, расширение функционала администрирования, внедрение end-to-end шифрования для повышения конфиденциальности, а также оптимизация производительности и масштабируемости для поддержки большого количества одновременных пользователей.

В целом, разработанный корпоративный мессенджер представляет собой современное, функциональное и надежное решение для обеспечения эффективных коммуникаций в IT-компаниях. Выбранный стек технологий, архитектурные решения и реализованный функционал делают его готовым к успешному внедрению и дальнейшему развитию в соответствии с потребностями бизнеса.

**Список литературы:**

1. Бэнкс А., Порселло Е. React и Redux: функциональная веб-разработка. — СПб.: Питер, 2018. — 336 с.
2. Дакетт Дж. HTML и CSS. Разработка и дизайн веб-сайтов. — М.: Эксмо, 2019. — 480 с.
3. Филдинг Р. Т. Рест. Архитектурный стиль передачи состояния // Архитектурные стили и проектирование сетевых архитектур на основе программного обеспечения. — 2000. — Гл. 5.
4. Флэнаган Д. JavaScript. Подробное руководство. — СПб.: Символ-Плюс, 2012. — 1080 с.
5. Фримен А. ReactJS. Основы веб-разработки. — М.: ДМК Пресс, 2022. — 364 с.
6. Хэррон Д. Node.js. Разработка серверных веб-приложений на JavaScript. — М.: ДМК Пресс, 2012. — 144 с.
7. Чамберс Дж., Пэкетт Д., Тиммс С. ASP.NET Core. Разработка приложений. — СПб.: Питер, 2018. — 464 с.
8. Шварц Б., Зайцев П., Ткаченко В. MySQL по максимуму. Оптимизация, репликация, резервное копирование. — СПб.: Питер, 2018. — 864 с.
9. Эспозито Д. Разработка современных веб-приложений: анализ предметных областей и технологий. — М.: Вильямс, 2017. — 464 с.
10. Chodorow K. MongoDB: The Definitive Guide: Powerful and Scalable Data Storage. — O'Reilly Media, Inc., 2013. — 432 p.
11. Fain Y., Moiseev A. Angular Development with TypeScript. — Manning Publications, 2019. — 584 p.
12. Kozlowski P., Darwin P. B. Mastering Web Application Development with Express. — Packt Publishing Ltd, 2014. — 308 p.
13. Negrino T., Smith D. JavaScript and Ajax for the Web: Visual QuickStart Guide (7th Edition). — Peachpit Press, 2008. — 456 p.
14. Newman S. Building Microservices: Designing Fine-Grained Systems. — O'Reilly Media, Inc., 2015. — 280 p.
15. Stefanov S. React: Up & Running: Building Web Applications. — O'Reilly Media, Inc., 2016. — 222 p.