МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт информационных технологий (ИТ)
Кафедра инструментального и прикладного обеспечения (ИиППО)

КУРСОВАЯ РАБОТА

по дисциплине: Разработка клиент-серверных приложений
по профилю: профессиональной подготовки: Разработка программных продуктов и проектирование информационных систем
направления Программная инженерия (09.03.04)

Тема: «Разработка клиент-серверного приложения для чата сотрудников организации»

Студент: Шумахер Марк Евгеньевич
Группа: ИКБО-20-22
Работа представлена к защите 20.05.2025/ _____/*Шумахер М.Е.*
(подпись и ф.и.о. студента)

Руководитель: ст.преп., Синицын Анатолий Васильевич

Работа допущена к защите ____.06.2025(дата) _____/*Синицын А.В.*
(подпись и ф.и.о.рук-ля)

Оценка по итогам защиты: _____
_____.06.2025 / *Синицын Анатолий Васильевич, ст.преп* /
_____.06.2025 / _____/
(подписи, дата, ф.и.о., должность, звание, уч. степень двух преподавателей, принявших защиту)

М. РТУ МИРЭА. 2025 г.

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА - Российский технологический университет»
РТУ МИРЭА

Институт информационных технологий (ИТ)
Кафедра инструментального и прикладного программного обеспечения (ИиППО)

ЗАДАНИЕ
на выполнение курсовой работы

по дисциплине: Разработка клиент-серверных приложений
по профилю: Разработка программных продуктов и проектирование информационных систем
направления профессиональной подготовки: Программная инженерия (09.03.04)
Студент: Шумахер Марк Евгеньевич
Группа: ИКБО-20-22
Срок представления к защите: 20.05.2025
Руководитель: старший преподаватель Синицын А.В.

Тема: «Разработка клиент-серверного приложения для чата сотрудников организации»

Исходные данные: Git, JavaScript, NodeJS, React, нормативный документ: инструкция по организации и проведению курсового проектирования СМКО МИРЭА 7.5.1/04.И.05-18.

Перечень вопросов, подлежащих разработке, и обязательного графического материала: 1. Провести анализ предметной области для выбранной темы. 2. Выбрать клиент-серверную архитектуру для разрабатываемого приложения и дать её детальное описание с помощью UML. 3. Выбрать программный стек для реализации фуллстек CRUD приложения. 4. Разработать клиентскую и серверную части приложения, реализовать авторизацию и аутентификацию пользователя, обеспечить работу с базой данных, заполнить тестовыми данными, валидировать ролевую модель на некорректные данные. 5. Провести фаззинг-тестирование. 6. Разместить исходный код клиент-серверного приложения в репозитории GitHub с наличием Dockerfile и описанием структуры проекта в readme файле. 7. Развернуть клиент-серверное приложение в облаке. 8. Разработать презентацию с графическими материалами.

Руководителем произведён инструктаж по технике безопасности, противопожарной технике и правилам внутреннего распорядка.

Зав. кафедрой ИиППО: _____ / *Болбаков Р. Г.* /, «21» февраля 2025 г.

Задание на КР выдал: _____ / *Синицын А. В.* /, «21» февраля 2025 г.

Задание на КР получил: _____ / *Шумахер М.Е.* /, «21» февраля 2025 г.

АННОТАЦИЯ

Отчет 36 с., 26 рис., 17 источн., 4 табл.

REACT, КЛИЕНТ-СЕРВЕРНОЕ ПРИЛОЖЕНИЕ, EXPRESS, NODEJS, ИНТЕРНЕТ-РЕСУРС, АРХИТЕКТУРА, ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Объект исследования – клиент-серверное приложение «TeamFlow».

Цель работы – анализ, проектирование и реализация клиент-серверного приложения для чата сотрудников организации.

В ходе работы был проведен анализ предметной области и обзор веб-приложений с аналогичной тематикой, а также проанализированы архитектурные стили и современные подходы к проектированию.

На основе оценки архитектурных стилей был выбран оптимальный подход, который лег в основу архитектуры клиент-серверного приложения. Результатом работы является веб-приложение «TeamFlow», которое исполняет функцию чата сотрудников.

ANNOTATION

Report 36 p., 26 fig., 17 sources, 4 tabl.

REACT, CLIENT-SERVER APPLICATION, EXPRESS, NODEJS, INTERNET RESOURCE, ARCHITECTURE, DESIGN PATTERNS

The object of the study is the client–server application «TeamFlow».

The purpose of the work is to analyze, design and implement a client–server application for employee chat.

In the course of the work, an analysis of the subject area and a review of web applications with a similar theme were carried out, as well as architectural styles and modern design approaches were analyzed.

Based on the assessment of architectural styles, the optimal approach was chosen, which formed the basis for the architecture of the client-server application. The result of the work is the «TeamFlow» web application, which performs the function of an employee chat.

СОДЕРЖАНИЕ

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

В настоящем отчете применяют следующие термины с соответствующими определениями:

Клиент – программное обеспечение или устройство, запрашивающее ресурсы или услуги у сервера в сети

Микросервис – независимый модуль системы, выполняющий определённую бизнес-логику

Монолит – архитектура, где вся система построена как единое приложение с общей кодовой базой

Фаззинг – процесс автоматического тестирования программного кода с генерацией разнообразных входных данных

Фреймворк – программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта

Хостинг – услуга по предоставлению ресурсов для размещения информации на сервере, постоянно имеющем доступ к сети

Dockerfile – файл для предварительной работы, набор инструкций, который нужен для записи образа

Render – бесплатный хостинг для размещения приложений

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящем отчете применяются следующие сокращения и обозначения:

СУБД	–	система управления базами данных
API	–	Application Programming Interface
CPU	–	Central Processing Unit
JS	–	JavaScript, язык программирования
SQL	–	Structured Query Language
UML	–	Unified Modeling Language

ВВЕДЕНИЕ

Цель работы – проанализировать существующие решения в сфере мессенджеров и приложений с чатами, изучить их функциональные возможности и особенности взаимодействия с пользователем, а также разработать собственный веб-сервис для чата сотрудников. В качестве технологической основы будут использованы язык программирования JavaScript (в среде выполнения Node.js) [1], фреймворк Express.js для создания серверной части (API), библиотека React [2] для разработки клиентской части и PostgreSQL [3] в качестве системы управления базами данных. Для достижения поставленной цели были сформулированы следующие задачи:

– провести анализ предметной области, связанной с разрабатываемым приложением,

– выбрать инструменты для реализации серверной и клиентской части,

– изучить популярные паттерны проектирования и обосновать выбор технологий для реализации выбранной архитектуры,

– разработать архитектуру и приложение с применением выбранных технологий,

– протестировать разработанное приложение, затем задеплоить.

Для выполнения этих задач применяются методы анализа и сравнительного исследования. Информационной базой для этой работы являются знания, полученные в ходе практических занятий курса «Разработка клиент-серверных приложений», а также данные из интернет-ресурсов.

В разделе с описанием предметной области расположена основная информация о проведенном анализе предметной области, функциональные и нефункциональные требования к системе, а также ограничения системы.

В разделе обоснования выбора технологий описано используемое программное обеспечение и технологии.

В разделе разработки приложения описана структура приложения, и реализация клиентской и серверной частей, тестирование и развертывание.

В разделе заключения подводятся итоги курсовой работы.

1 ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Анализ предметной области

Анализ предметной области проводился среди веб-приложений на тематику «Чат для сотрудников». Были выбраны три веб-приложения: Microsoft Teams [4], Telegram [5], Compass [6] представленные на рисунках 1.1-1.3.

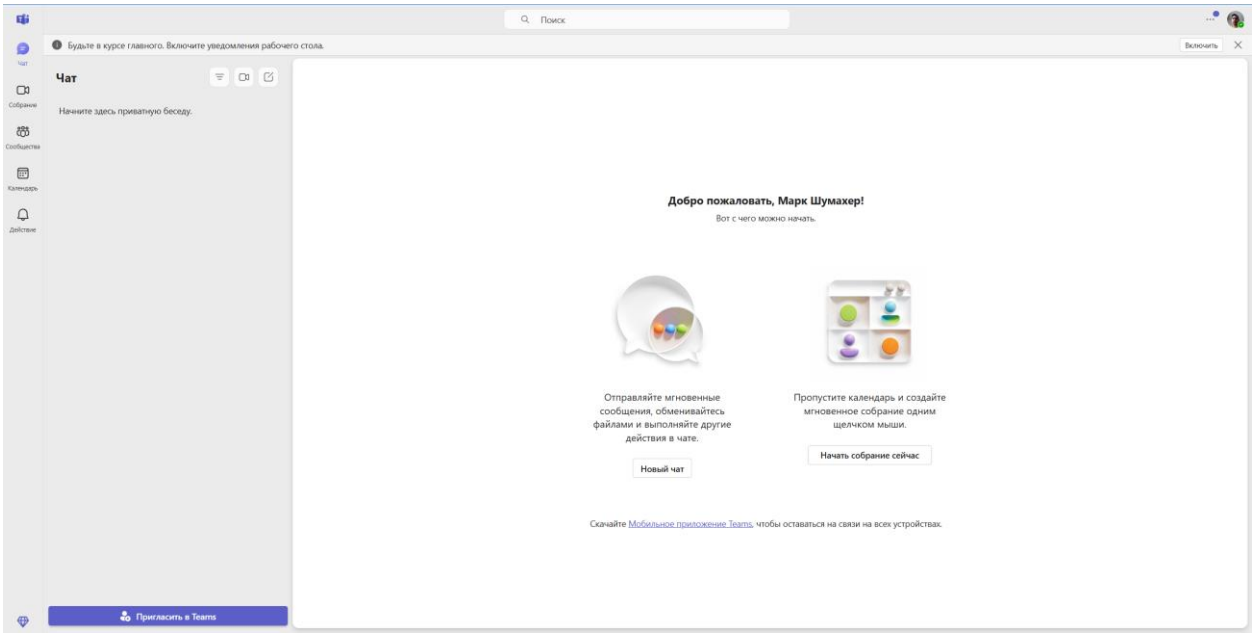

Рисунок 1.1 – Клиентская часть сайта Microsoft Teams

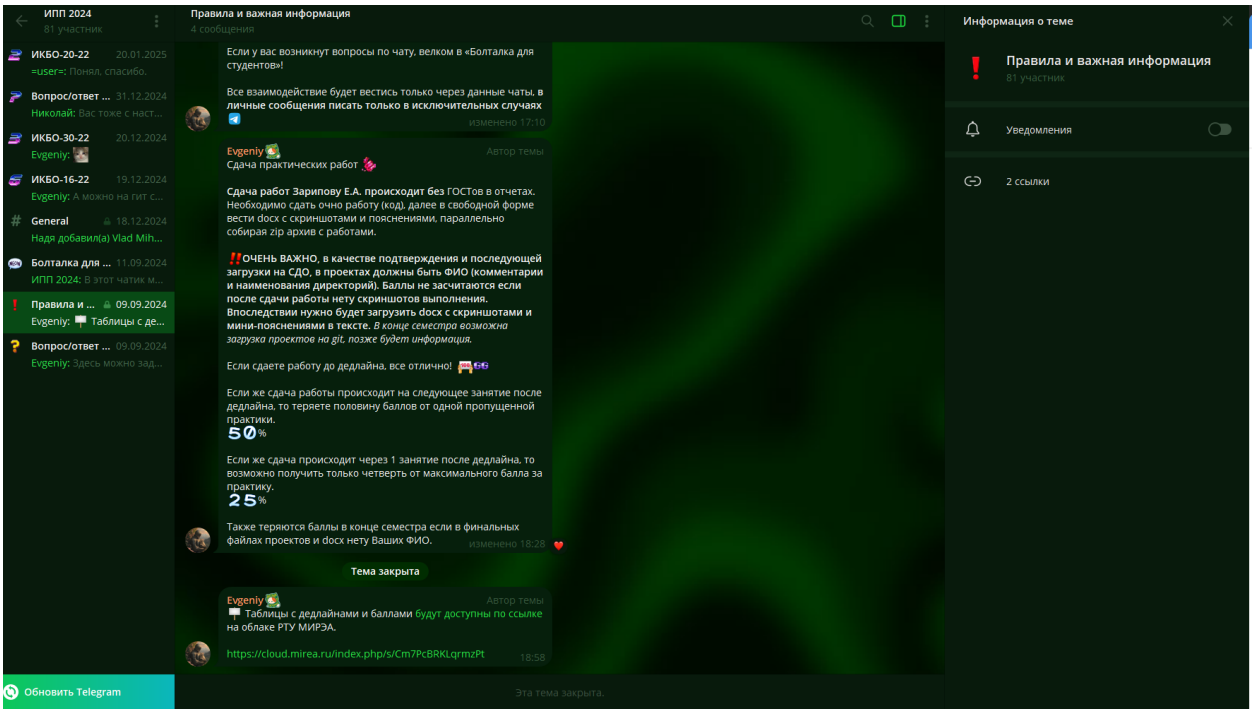

Рисунок 1.2 – Клиентская часть сайта Telegram

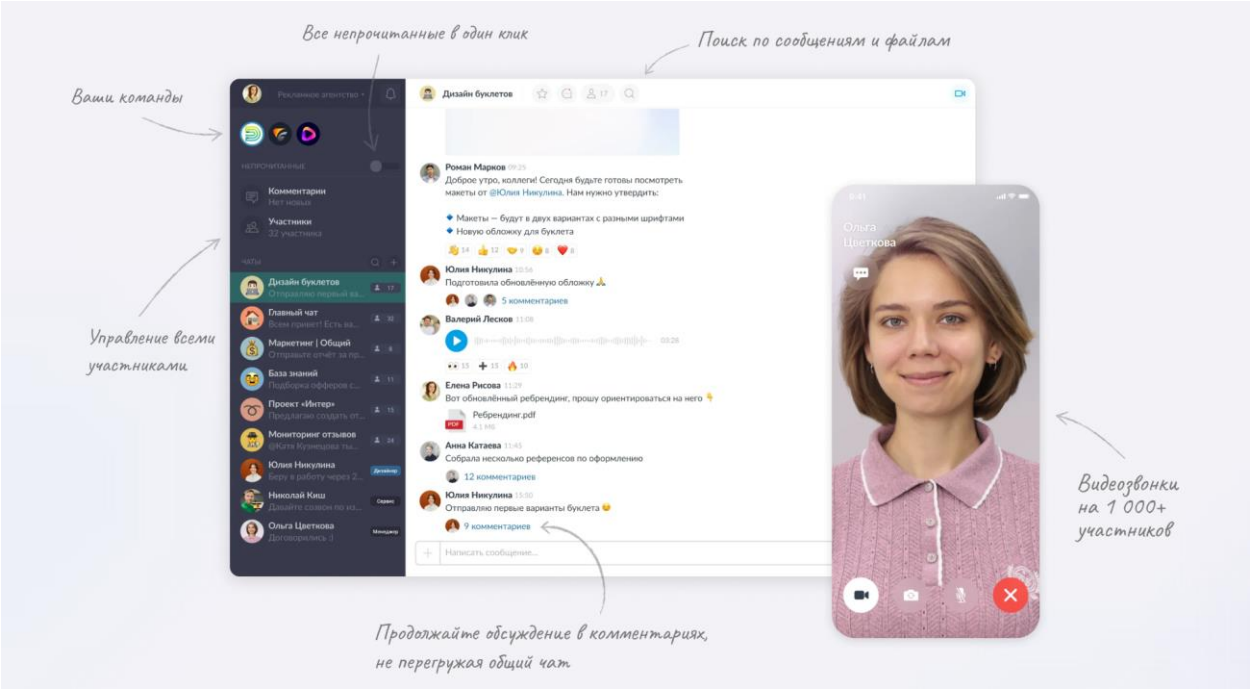

Рисунок 1.3 – Клиентская часть сайта Compass

Все рассмотренные веб-приложения предоставляют пользователям возможность регистрации и входа в аккаунт, создание чатов, возможность использовать чат и многие другие функции.

Из анализа данных веб-приложений следует выделить ключевые возможности, которыми должен обладать разрабатываемый веб-ресурс данной тематики:

– регистрация и авторизация на веб-сервис,

– возможность создания чатов,

– ввод ролевой системы внутри конкретного чата,

– добавления тегов для участников чата,

– возможность назначения администраторов чата,

– добавление пользователей в чат,

– использование эмодзи внутри чата,

– просмотр списка участников чата.

Приложение должно быть разработано с учетом требований современного рынка веб-приложений в данной тематике, обеспечивая конкурентоспособность. Для этого необходимо внедрить актуальные

функции, такие как использование тегов, что особенно актуально в рабочих чатах, а также возможность просмотра списка участников чата и назначения администраторов внутри чата. Кроме того, важно обеспечить удобство интерфейса и высокую производительность.

Уникальность приложения заключается в его простоте и удобстве для пользователей. Интуитивно понятный интерфейс позволяет легко разобраться в функционале. Также в чатах реализована функция назначения тегов (должностей) для каждого участника, что подчеркивает его ориентированность на различные организации, ведь при назначении тега на пользователя будет видно, в чате около его имени пользователя, в скобках назначенный тег. Например: «lord_of_cod (Иван Иванов – QA Manager)». А также чаты являются закрытыми, так как в чат не попасть без приглашения администратора.

1.2 Описание функционала приложения

В разрабатываемом приложении чата для сотрудников необходимо реализовать следующие функции: регистрация и авторизация на сайт; создание чатов; обработка отправки и принятия сообщений; использование эмодзи в чате; показ списка участников чата; возможность назначать и снимать администраторов из числа участников чата; приглашение и удаление пользователей из чата; назначение тегов на участников чата. Система должна обрабатывать ошибки, возникающие в процессе любых запросов или обновлений данных, и информировать пользователя о таких ситуациях.

Среди нефункциональных требований ключевое внимание уделяется удобству и производительности интерфейса, который должен быть интуитивно понятным и обеспечивать быстрый доступ к основным функциям. Отправка и принятие сообщений должны происходить без задержек. Взаимодействие клиента и сервера должно быть реализовано оптимальным образом для обеспечения стабильной работы приложения даже при большом количестве одновременных пользователей. Система должна быть легко масштабируемой, чтобы поддерживать рост нагрузки и объема данных. Это,

например, включает в себя увеличение количества одновременных активных пользователей с 15 до 500, а также расширение общей пользовательской базы с 100 до 10 000 зарегистрированных пользователей, которые когда-либо взаимодействовали с сервисом. Кроме того, архитектура должна эффективно обрабатывать рост объема данных в базе данных от 100 мегабайт до 100 гигабайт, обеспечивая стабильность, производительность и отказоустойчивость на всех этапах масштабирования. Особое внимание необходимо уделить безопасности данных пользователей, поэтому все пароли пользователей хранятся в БД в зашифрованном виде.

Проект имеет ряд ограничений, которые могут повлиять на его работу. Система будет функционировать в условиях ограниченных серверных ресурсов, а именно – оперативная память от 2 ГБ, а процессорное время от 300 секунд CPU в сутки. Это потребует эффективной оптимизации алгоритмов и процессов для снижения нагрузки на серверы, поддержания стабильности работы и обеспечения высокой производительности при увеличении нагрузки. Возможны задержки при обработке запросов, если в базе данных хранится большое количество данных, например, при значительном числе сохраненных сообщений. Увеличение нагрузки на сервер может привести к замедлению работы приложения, особенно при большом количестве пользователей, одновременно выполняющих действия, такие как обмен сообщениями. Клиентом для данного приложения является веб-браузер. Приложение не поддерживает использования его в виде мобильного, десктоп или иного варианта клиента.

Разработка и тестирование ключевых компонентов системы должны быть завершены в срок не более 2-3 месяцев, что обеспечит своевременный запуск проекта и готовность к увеличению нагрузки.

1.3 Описание функционала приложения в нотации UML

Для создания UML-диаграммы [7] использован сервис draw.io [8].

Согласно функциональным требованиям создана диаграмма Use-case, показанная на рисунке 1.4.

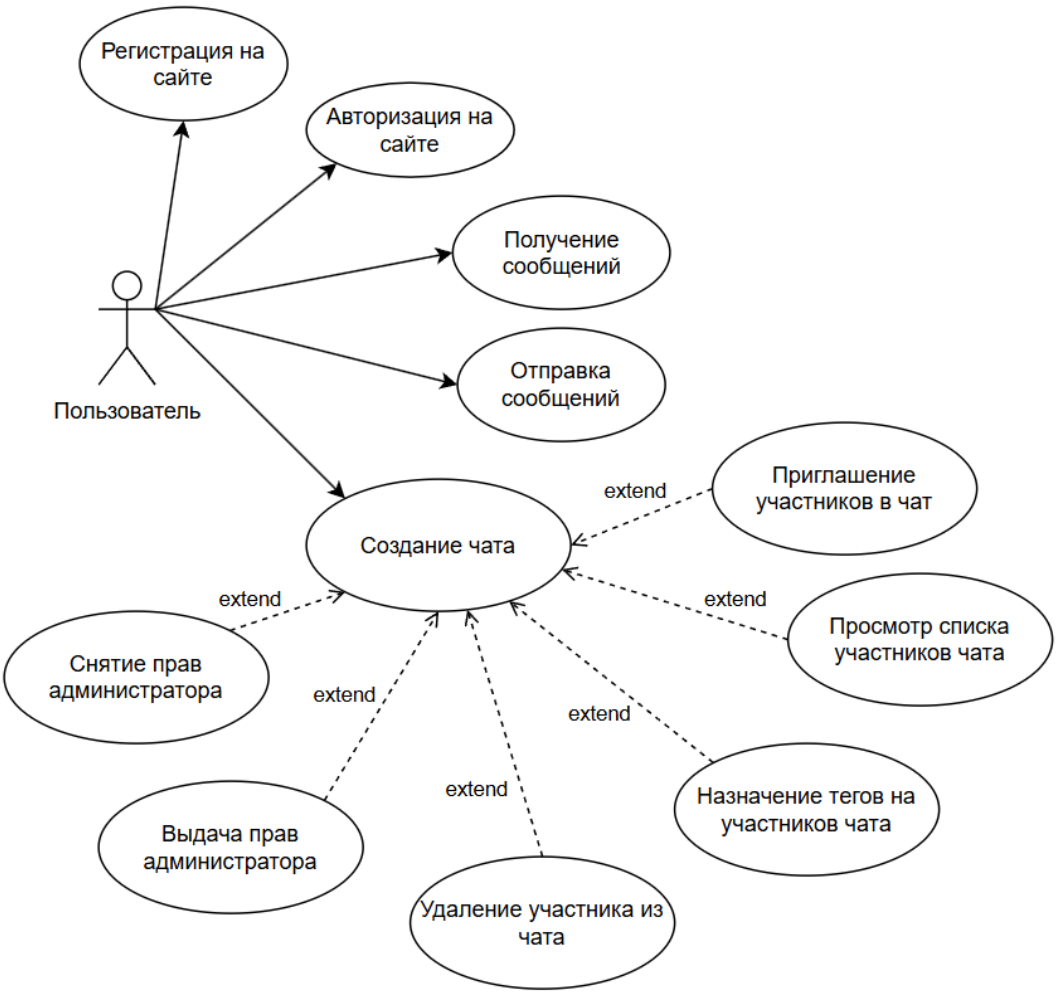

Рисунок 1.4 – Диаграмма Use-case

2 ОБОСНОВАНИЕ ВЫБОРА ТЕХНОЛОГИЙ

2.1 Используемое прикладное программное обеспечение

Для того, чтобы определить наиболее подходящую среду разработки проведем сравнительный анализ сред разработки VS Code [9] и WebStorm [10], как показано в таблице 2.1.

Таблица 2.1 – Характеристики сред разработки

Характеристика	WebStorm	VS Code
Поддержка Node.js	Глубокая интеграция, автодополнение, рефакторинг	Отличная поддержка через расширения (ESLint, Prettier и др.)
Поддержка Express	Встроенная помощь по роутингу, middleware	Требует расширений (REST Client, Thunder Client)
Поддержка React	Полная IDE-поддержка (JSX, хуки)	Лучшая поддержка через расширения (ES7+ React/Redux, React Refactor)
Производительность	Ресурсоемкий (требует больше ОЗУ)	Легковесный, быстрый запуск
Отладка	Продвинутые инструменты отладки	Хорошая отладка + расширения (Debugger for Chrome)
Интеграция с Git	Полноценная встроенная поддержка	Требует расширений (GitLens, GitHub Pull Requests)

Для разработки приложения выбран VS Code. Это окружение обладает широкими возможностями кастомизации, имеет низкие системные требования при сохранении высокой функциональности, поддерживает все необходимые технологии через систему расширений, обеспечивает кроссплатформенную совместимость. Данный выбор оптимально соответствует требованиям проекта по производительности, гибкости и удобству разработки.

Для проверки работы клиент-серверного приложения был выбран браузер Google Chrome, поскольку он широко используется, обладает высокой производительностью и поддерживает современные веб-стандарты.

2.2 Используемые технологии

2.2.1 Языки программирования

Для выбора языка программирования при разработке клиент-серверного приложения был проведён анализ ключевых аспектов: платформенная независимость, производительность, экосистема библиотек, поддержка асинхронности, скорость разработки и безопасность.

Рассмотрены JavaScript (Node.js + Express) [11], Python (FastAPI) [12], Java (Spring Boot) [13]. Основные характеристики этих языков приведены в таблице 2.2.

Таблица 2.2 – Характеристики языков программирования

Характеристики	JavaScript (Node.js + Express)	Python (FastAPI)	Java (Spring Boot)
Платформенная независимость	Высокая (запуск везде, где есть Node.js)	Высокая (интерпретируемый)	Высокая (JVM)
Производительность	Высокая (V8, асинхронный I/O)	Средняя (медленнее JS/Java)	Высокая (JIT-компиляция)
Библиотеки и фреймворки	Богатая экосистема (Express, NestJS, Socket.IO)	FastAPI, Flask, Django	Spring, Hibernate, Jakarta EE
Поддержка асинхронности	Лучшая (Event Loop, async/await)	Asyncio (но слабее, чем у Node.js)	Реактивные фреймворки (WebFlux)
Скорость разработки	Очень высокая (гибкость, NPM)	Высокая (меньше кода)	Средняя (много конфигурации)
Безопасность	Зависит от разработчика (нужны middleware)	Встроенная валидация (Pydantic)	Spring Security (Enterprise)

Для серверной разработки JavaScript (Node.js в сочетании с Express) обладает рядом ключевых преимуществ, которые делают его оптимальным выбором для создания клиент-серверных приложений. К этим преимуществам относятся высокая производительность за счет асинхронной модели выполнения, масштабируемость и универсальность использования. Эти характеристики особенно важны для современной backend-разработки.

2.2.2 Базы данных

Для выбора подходящей базы данных для разрабатываемого клиент-серверного приложения был проведен сравнительный анализ нескольких популярных баз данных. В рамках анализа были рассмотрены такие варианты, как PostgreSQL, MongoDB [14] и MySQL [15]. Основные характеристики этих баз данных, а также их сильные и слабые стороны, приведены в таблице 2.3.

Таблица 2.3 – Характеристики баз данных

Характеристики	PostgreSQL	MongoDB	MySQL
Тип базы данных	Реляционная	Документо-ориентированная	Реляционная
Масштабируемость	Хорошая вертикальная и горизонтальная масштабируемость	Отличная горизонтальная масштабируемость (sharding)	Хорошая вертикальная масштабируемость

Продолжение таблицы 2.3

Производительность	Высокая производительность для сложных запросов	Высокая производительность для больших объемов неструктурированных данных	Хорошая производительность для стандартных SQL-запросов
Интеграция с JavaScript/Node.js	Отличная (node-postgres, Sequelize, TypeORM)	Хорошая (Mongoose, официальный MongoDB driver)	Хорошая (mysql2, Sequelize, TypeORM)
Гибкость схемы	Строгая схема (нужна схема таблицы)	Очень гибкая (без схемы)	Строгая схема (нужна схема таблицы)

Выбор PostgreSQL в качестве базы данных был обусловлен рядом факторов, которые делают её идеальным выбором для разрабатываемого клиент-серверного приложения. PostgreSQL обладает мощными возможностями для работы с реляционными данными, включая полную поддержку SQL, сложные запросы, транзакции и интеграцию с различными типами данных. Это обеспечило высокую гибкость в хранении и управлении данными приложения, что важно для обеспечения надежности и эффективности работы.

2.2.3 Инструменты DevOps

Для хостинга клиент-серверного приложения был выбран облачный сервис Render [16], который предоставляет автоматизированные инструменты для развертывания, масштабирования и управления инфраструктурой. Это решение позволяет разработчикам сосредоточиться на разработке функционала приложения, избавляя от необходимости вручную настраивать серверы и управлять ресурсами. Render поддерживает широкий спектр технологий, таких как Docker [17], Node.js и Java, что делает его гибким и универсальным решением для хостинга различных типов приложений. Интеграция с GitHub позволяет автоматизировать обновление приложения при каждом изменении в репозитории. Также, благодаря функции автоматического масштабирования, Render эффективно управляет ресурсами в зависимости от текущей нагрузки, обеспечивая высокую доступность и

стабильную производительность системы. На рисунке 2.1 представлена диаграмма компонентов.

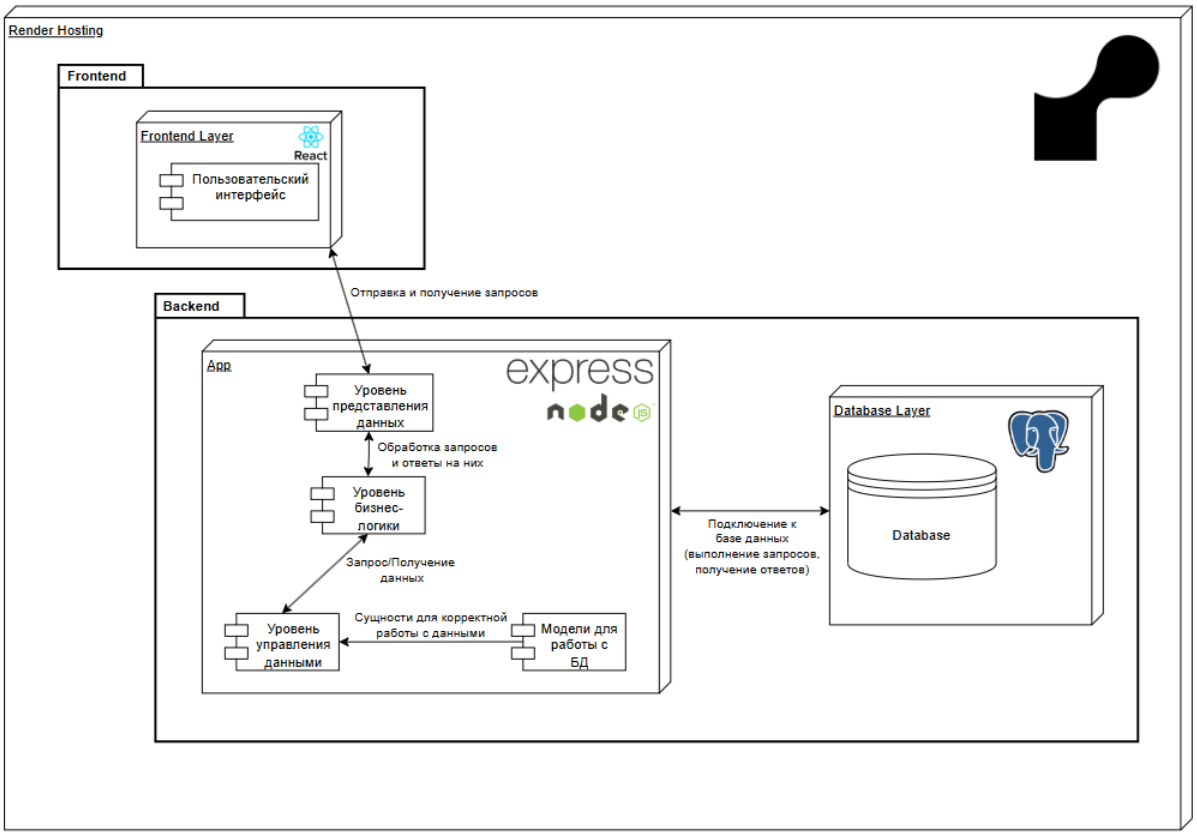

Рисунок 2.1 – Диаграмма компонентов системы

2.3 Анализ архитектурных стилей и паттернов проектирования

Для выбора архитектурного стиля разрабатываемого приложения необходимо провести анализ наиболее популярных архитектур.

Одним из наиболее распространенных является монолитная архитектура. Монолитная архитектура – это подход, при котором все компоненты и функции приложения объединены в единую систему. Она выделяется своей простотой в разработке и развертывании: все элементы находятся в одном месте, что уменьшает затраты времени и ресурсов. Кроме того, тестировать монолитное приложение проще, так как вся система проверяется как целое. Однако масштабируемость в монолите представляет собой вызов: для увеличения производительности необходимо масштабировать всю систему, а не отдельные её части. Это усложняет обновления, поскольку изменения одного компонента могут повлиять на

другие части, увеличивая вероятность ошибок и сбоев. Кроме того, использование разных технологий внутри одной системы ограничено.

Клиент-серверная архитектура, другой популярный стиль, основывается на разделении приложения на две части: клиент, запрашивающий услуги, и сервер, предоставляющий их. Основное преимущество такого подхода – чёткое разделение ролей, что упрощает поддержку и развитие системы. Однако этот стиль имеет и слабые стороны, например, зависимость от сети. Если клиент генерирует слишком много запросов, сервер может испытывать перегрузку, что негативно влияет на производительность системы.

Сервисно-ориентированная архитектура (SOA) строится на идее создания приложения в виде набора взаимосвязанных сервисов, каждый из которых выполняет конкретную задачу. Такой подход позволяет повторно использовать компоненты, снижая затраты на разработку, и упрощает масштабирование отдельных частей системы. Однако взаимодействие между сервисами часто требует сложного управления. Для обеспечения совместимости и стабильной работы приложения требуется значительное внимание и усилия.

Микросервисная архитектура – это современный подход, развивающий идеи сервисно-ориентированной архитектуры (SOA). Она предполагает деление приложения на множество небольших и автономных сервисов, каждый из которых отвечает за выполнение своей задачи. Такой стиль дает высокую гибкость: для каждого микросервиса можно использовать свои технологии и языки программирования, что облегчает адаптацию системы к разным требованиям. Кроме того, микросервисы упрощают процесс развертывания и обновления: отдельные сервисы можно обновлять независимо, без необходимости вмешательства в работу всего приложения.

Для разрабатываемого чат-приложения выбрана классическая монолитная архитектура. Во-первых, функционал приложения – аутентификация, чаты, сообщения и управление пользователями – тесно связан между собой. Все эти компоненты логично работают в рамках единой

системы, без необходимости разбивать их на отдельные сервисы. Во-вторых, монолитная архитектура ускоряет разработку. Весь бэкенд собран в одном Express-приложении, а фронтенд – в одном React-проекте. Это значит, что не нужно настраивать сложную инфраструктуру, API-шлюзы или отдельные базы данных для каждого модуля. Достаточно одной PostgreSQL, где хранятся все данные: пользователи, чаты и сообщения. Кроме того, такой подход упрощает тестирование и развертывание. Можно запускать и проверять приложение как единое целое, без необходимости отдельно разворачивать сервисы аутентификации, чатов и сообщений. Наконец, монолит удобен для небольших проектов, где важна скорость внесения изменений. Все правки вносятся в одну кодовую базу, и не нужно согласовывать обновления между разными сервисами, как в микросервисной архитектуре. Это особенно важно на ранних этапах, когда требования могут меняться, и нужно быстро адаптировать функционал.

Таким образом, выбор монолитной архитектуры для приложения полностью оправдан: он дает простоту разработки, удобство поддержки и достаточную гибкость для дальнейшего роста.

Для выбора архитектурного паттерна разрабатываемого приложения необходимо разобрать наиболее популярные виды. Сравнительный анализ характеристик архитектурных паттернов приведён в таблице 2.4.

Таблица 2.4 – Характеристики архитектурных паттернов

Характеристика	MVC (Model-View-Controller)	MVP (Model-View-Presenter)	MVVM (Model-View-ViewModel)
Сложность реализации	Низкая. Простая реализация на начальных этапах.	Средняя. Требует больше усилий для реализации из-за добавления презентера.	Высокая. Реализация сложнее всего из-за необходимости создания ViewModel.
Использование слоев	Да	Да	Да
Удобство тестирования	Хорошая, но тестирование контроллера сложно.	Очень высокая, презентер легко тестировать.	Отличная, ViewModel полностью изолируется.
Гибкость	Средняя, сложно добавлять новые функции.	Высокая, благодаря слабой зависимости слоев.	Очень высокая, легко адаптируется под изменения.

Продолжение таблицы 2.4

Тестируемость	Высокая	Очень высокая	Высокая
Поддержка изменений	Затруднена из-за сильных зависимостей.	Удобная благодаря изоляции логики в презентере.	Лучшая, изменения легко изолируются.
Масштабируемость	Умеренная, часто требует переписывания кода.	Высокая, легко добавлять новые функции.	Отличная, минимальное вмешательство в код.
Поддержка изменений	Затруднена из-за сильных зависимостей.	Удобная благодаря изоляции логики в презентере.	Лучшая, изменения легко изолируются.

Для разработки приложения наиболее подходящим является архитектурный паттерн MVC (Model-View-Controller). Он разделяет логику на модель, представление и контроллер, что делает процесс разработки и поддержки более удобным. Данный паттерн идеально подходит для приложений с простым интерфейсом, не требующих сложных взаимодействий, характерных для MVP или MVVM.

В итоге, выбранная архитектура проекта представляет собой модифицированный вариант классического паттерна MVC, адаптированный под специфику веб-приложений на базе Express.js. Основой системы остаются три ключевых компонента MVC. Модель включает сущности базы данных PostgreSQL и бизнес-логику работы с ними. Контроллеры обрабатывают входящие запросы и управляют потоком данных. Представление вынесено в отдельное клиентское приложение на React, взаимодействующее с бэкендом через JSON API. Архитектура дополнена элементами слоистой организации. Четко прослеживается разделение на транспортный уровень (Express-роутеры), уровень бизнес-логики (контроллеры) и уровень доступа к данным (SQL-запросы). Такая структура обеспечивает гибкость при сохранении понятной иерархии компонентов. Важным аспектом реализации стала интеграция Event-Driven подхода через Socket.io для обработки событий в реальном времени, без чего не представляется реализация чата. Этот механизм органично встроен в общую архитектуру, используя те же модели данных, что

и REST-часть приложения. При этом соблюдены принципы RESTful API для синхронных операций.

2.4 Модель базы данных

В системе реализована работа с базой данных с помощью СУБД PostgreSQL База данных приложения построена на PostgreSQL и включает четыре основные сущности: таблицу users для хранения данных пользователей (логины и хеши паролей), таблицу chats для создания чат-комнат с указанием создателя, таблицу chat_users (связующую) для управления участниками чатов и их ролями (администратор/участник), а также таблицу messages для хранения всей истории переписки с метаданными (отправитель, чат, время). Особенностью структуры являются связи по внешним ключам: чаты привязаны к создателям, сообщения - к чатам и отправителям, а таблица chat_users реализует многие-ко-многим между пользователями и чатами. На рисунке 2.2 изображена ERD-диаграмма базы данных.

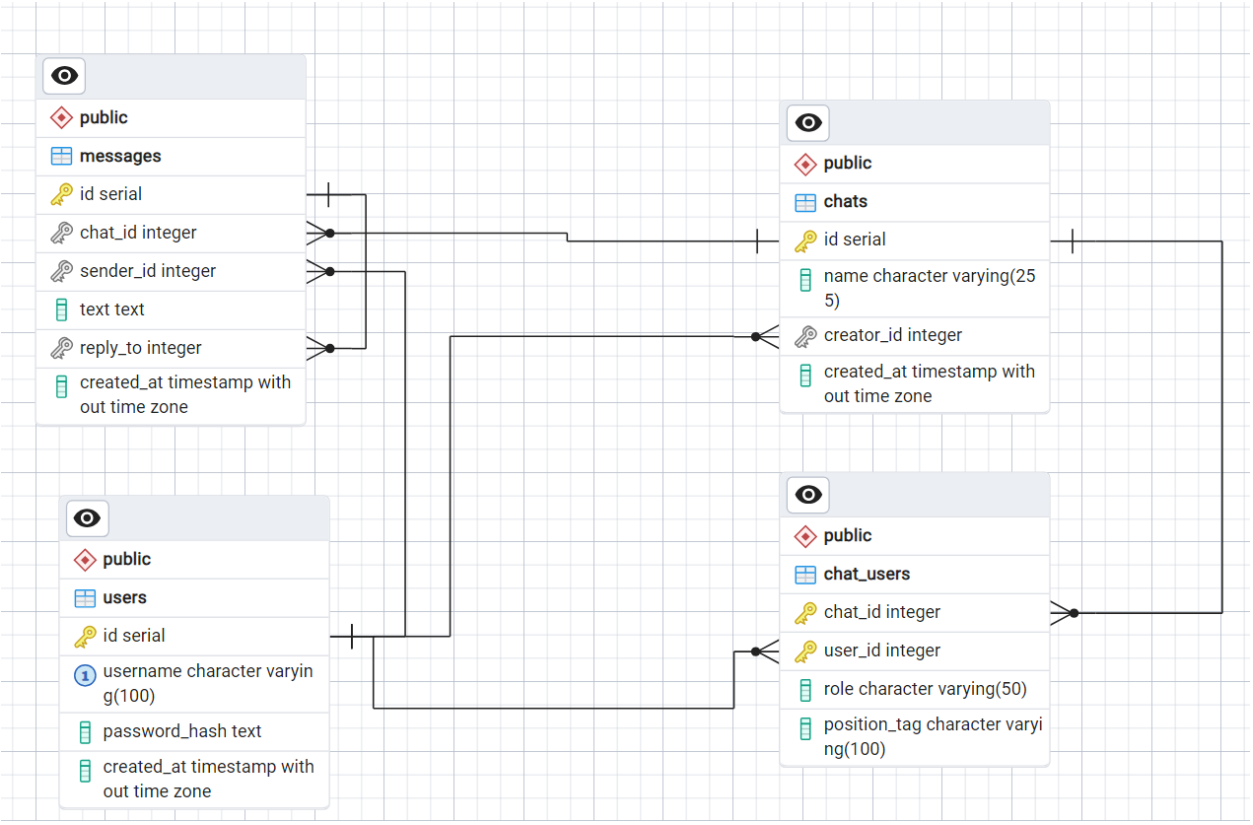

Рисунок 2.2 – ERD-диаграмма базы данных

3 РАЗРАБОТКА ПРИЛОЖЕНИЯ

3.1 Разработка серверной части приложения

После анализа предметной области, выбора архитектурного стиля, описания компонентов системы и сценариев их взаимодействия, а также определения необходимых технологий, начинается этап непосредственной реализации системы.

Разработка началась с серверной части, в рамках которой были реализованы ключевые точки взаимодействия между клиентом и сервером, а также работа с базой данных, валидация данных на проверку прав, добавление тестовых данных.

На рисунках 3.1-3.5 представлена основная логика работы серверной части приложения.

```
const register = async (req, res) => {
  const { username, password } = req.body;
  try {
    const hash = await bcrypt.hash(password, 10);
    const result = await pool.query(
      "INSERT INTO users (username, password_hash) VALUES ($1, $2) RETURNING id, username",
      [username, hash]
    );
    const user = result.rows[0];
    const token = jwt.sign({ userId: user.id }, process.env.JWT_SECRET, { expiresIn: "24h" });
    res.status(201).json({ token, userId: user.id, username: user.username });
  } catch (err) {
    console.error(err);
    if (err.code === '23505') { // уникальный ключ - username уже существует
      res.status(400).json({ error: "Данный логин занят" });
    } else {
      res.status(500).json({ error: "Ошибка на сервере" });
    }
  }
};

const login = async (req, res) => {
  const { username, password } = req.body;
  try {
    const userRes = await pool.query("SELECT * FROM users WHERE username = $1", [username]);
    const user = userRes.rows[0];

    if (!user) {
      return res.status(401).json({ error: "Такого пользователя не существует" });
    }

    const isMatch = await bcrypt.compare(password, user.password_hash);
    if (!isMatch) {
      return res.status(401).json({ error: "Неверные логин или пароль (на самом деле только пароль)" });
    }

    const token = jwt.sign({ userId: user.id }, process.env.JWT_SECRET, { expiresIn: "24h" });
    res.json({ token, username: user.username, userId: user.id });
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: "Ошибка на сервере" });
  }
};
```

Рисунок 3.1 – Контроллеры авторизации и регистрации

```
const Message = require("../models/Message");

exports.sendMessage = async (req, res) => {
  const { chatId, text, replyTo } = req.body;
  const senderId = req.user.userId;
  const username = req.user.username;

  try {
    const message = await Message.send(chatId, senderId, text, replyTo, username);
    res.status(201).json(message);
  } catch (err) {
    res.status(500).json({ error: "Ошибка отправки сообщения" });
  }
};

exports.getMessages = async (req, res) => {
  const chatId = req.params.chatId;
  try {
    const messages = await Message.getChatMessages(chatId);
    res.json(messages);
  } catch (err) {
    res.status(500).json({ error: "Ошибка получения сообщений" });
  }
};
```

Рисунок 3.2 – Контроллеры работы с сообщениями

```
const { pool } = require("../db");

const Message = {
  async send(chatId, senderId, text, replyTo = null) {
    const result = await pool.query(
      "INSERT INTO messages (chat_id, sender_id, text, reply_to) VALUES ($1, $2, $3, $4) RETURNING *",
      [chatId, senderId, text, replyTo]
    );
    return result.rows[0];
  },

  async getChatMessages(chatId) {
    const result = await pool.query(
      `SELECT
        m.id,
        m.chat_id,
        m.sender_id,
        m.text,
        m.reply_to,
        m.created_at AS timestamp,
        u.username,
        cu.position_tag
      FROM messages m
      JOIN users u ON m.sender_id = u.id
      JOIN chat_users cu ON m.sender_id = cu.user_id AND m.chat_id = cu.chat_id
      WHERE m.chat_id = $1
      ORDER BY m.created_at ASC`,
      [chatId]
    );
    return result.rows;
  },
};

module.exports = Message;
```

Рисунок 3.3 – Модель сообщений для работы сокета

```
// Создание чата
exports.createChat = async (req, res) => {
  const { name } = req.body;
  const userId = req.user.userId;

  try {
    const chatResult = await pool.query(
      'INSERT INTO chats (name, creator_id) VALUES ($1, $2) RETURNING *',
      [name, userId]
    );
    const chat = chatResult.rows[0];

    // добавим создателя в chat_users с ролью admin
    await pool.query(
      'INSERT INTO chat_users (chat_id, user_id, role) VALUES ($1, $2, $3)',
      [chat.id, userId, 'admin']
    );

    res.status(201).json(chat);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Ошибка при создании чата' });
  }
};

// проверка админки
const isUserAdmin = async (chatId, userId) => {···
};

// админка выдача
exports.assignRole = async (req, res) => {···
};

// лепим тег
exports.assignPositionTag = async (req, res) => {···
};

// Добавление пользователя в чат
exports.addUserToChat = async (req, res) => {···
};
//список юзеров в чате
exports.getChatUsers = async (req, res) => {···
```

Рисунок 3.4 – Часть кода контроллеров чата

```
const express = require('express');
const router = express.Router();
const auth = require('../middleware/authMiddleware');
const checkRole = require('../middleware/roleMiddleware');
const chatController = require('../controllers/chatController');

router.get("/:chatId/users", auth, chatController.getChatUsers);
router.get("/", auth, chatController.getUserChats);
router.post("/", auth, chatController.createChat);
router.post("/create", auth, chatController.createChat);
router.post("/add-user", auth, checkRole("admin"), chatController.addUserToChat);
router.post("/:chatId/assign-role", auth, chatController.assignRole);
router.post("/:chatId/assign-position-tag", auth, chatController.assignPositionTag);

router.delete("/:chatId", auth, chatController.deleteChat);
router.delete("/:chatId/users/:userId", auth, chatController.removeUserFromChat);
router.post("/:chatId/revoke-admin/:userId", auth, chatController.revokeAdmin);

module.exports = router;
```

Рисунок 3.5 – Роуты для чата

На рисунке 3.6 представлена валидация ролевой модели на проверку прав.

middleware > JS roleMiddleware.js > ...

```
const { pool } = require('../db');

module.exports = (requiredRole) => {
  return async (req, res, next) => {
    const userId = req.user.userId;
    const chatId = req.params.chatId || req.body.chatId;

    if (!chatId) {
      return res.status(400).json({ message: 'Не передан chatId' });
    }

    try {
      const result = await pool.query(
        'SELECT role FROM chat_users WHERE chat_id = $1 AND user_id = $2',
        [chatId, userId]
      );

      const userRole = result.rows[0]?.role;

      if (!userRole || (requiredRole === 'admin' && userRole !== 'admin')) {
        return res.status(403).json({ message: 'Недостаточно прав' });
      }

      next();
    } catch (err) {
      console.error(err);
      res.status(500).json({ message: 'Ошибка проверки прав' });
    }
  };
};
```

Рисунок 3.6 – Валидация данных

На рисунке 3.7 представлено добавление тестовых данных в БД при запуске сервера. Создается два пользователя, один чат, где один из пользователей имеет роль админа, а другой участника, каждому назначается тег (должность) и в чат от каждого отправляется по одному сообщению.

```
// Получить или создать пользователя
async function getOrCreateUser(username, password) {
  const check = await pool.query('SELECT id FROM users WHERE username = $1', [username]);

  if (check.rows.length > 0) {
    console.log(`✅ Пользователь ${username} уже существует (ID: ${check.rows[0].id})`);
    return check.rows[0].id;
  }

  const hashedPassword = await bcrypt.hash(password, 10);
  const result = await pool.query(
    'INSERT INTO users (username, password_hash) VALUES ($1, $2) RETURNING id',
    [username, hashedPassword]
  );
  console.log(`✅ Создан пользователь ${username} (ID: ${result.rows[0].id})`);
  return result.rows[0].id;
}

// Получить или создать чат
async function getOrCreateChat(name, creatorId) {
  const check = await pool.query('SELECT id FROM chats WHERE name = $1', [name]);

  if (check.rows.length > 0) {
    console.log(`✅ Чат "${name}" уже существует (ID: ${check.rows[0].id})`);
    return check.rows[0].id;
  }

  const result = await pool.query(
    'INSERT INTO chats (name, creator_id) VALUES ($1, $2) RETURNING id',
    [name, creatorId]
  );
  console.log(`✅ Создан чат "${name}" (ID: ${result.rows[0].id})`);
  return result.rows[0].id;
}
```

Рисунок 3.7 – Часть кода загрузки тестовых данных в БД

3.2 Клиентская часть приложения

Для разработки клиентской части использовался фреймворк React. Клиентская часть приложения при заходе на сайт состоит из страницы регистрации и авторизации, рисунок 3.8.

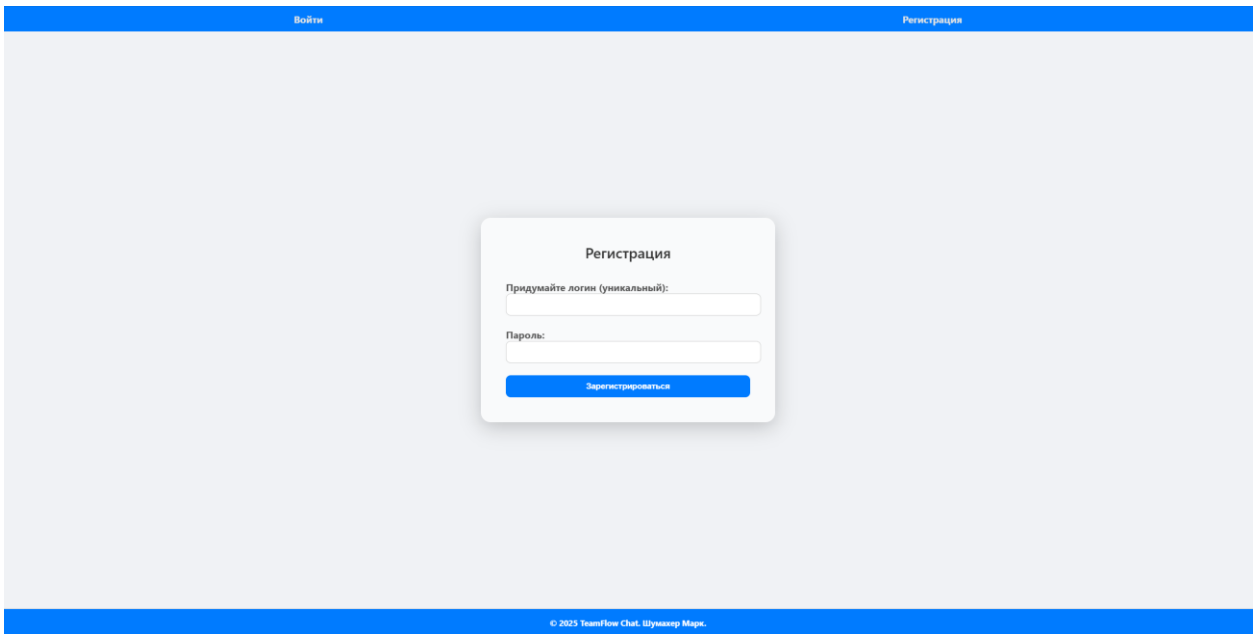

Рисунок 3.8 – Страница регистрации

Когда выполнен вход на сайт, пользователю отображается страница его чатов, куда он был приглашен и чаты, которые были созданы им самим, рисунок 3.9.

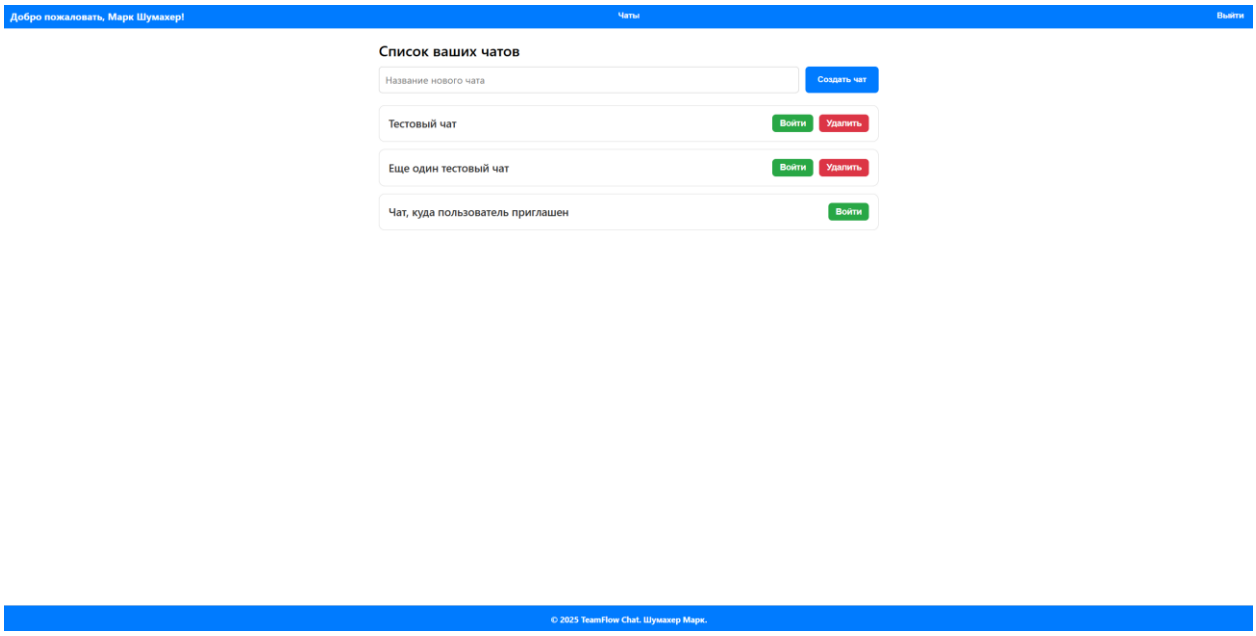

Рисунок 3.9 – Список чатов

Открыв чат, нам откроется само диалоговое окно, список пользователей и кнопки для добавления новых пользователей в чат, а также управление участниками чата (доступно только администратору чата) – назначение администратором (снятие), назначение должности (тега), рисунок 3.10.

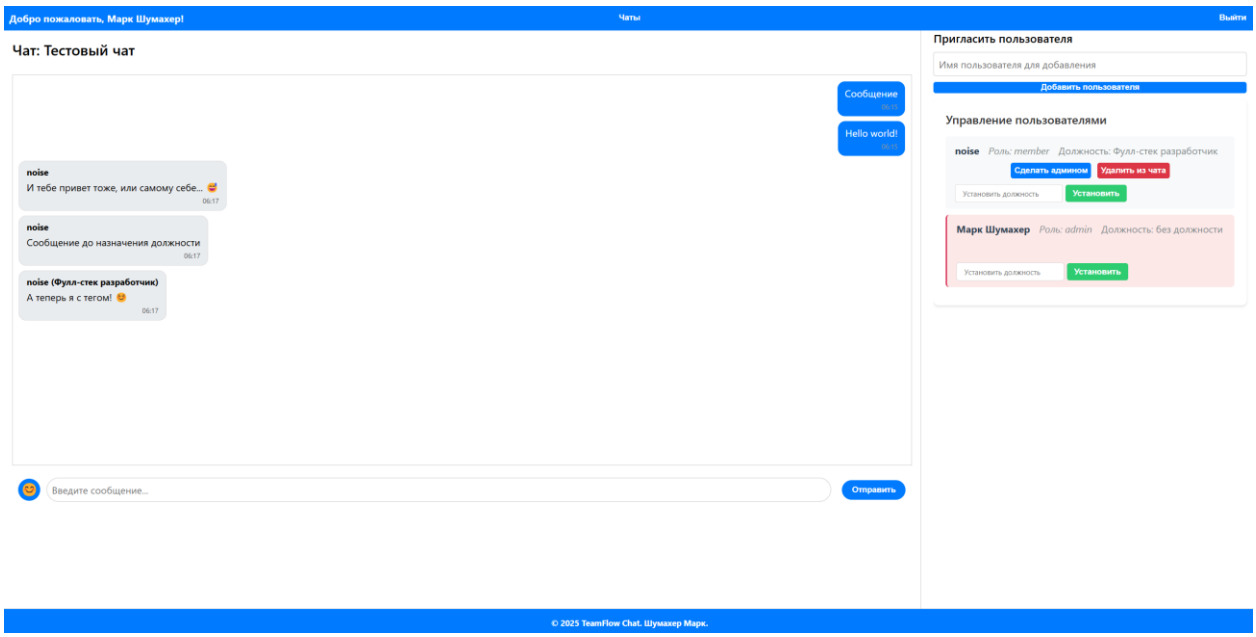

Рисунок 3.10 – Функционал страницы чата

3.3 Тестирование приложения

Используя программу Postman и браузер, а также фаззинг-тестирование, были протестированы несколько конечных точек приложения. На рисунках 3.11,3.12 представлены часть кода написанных тестов и результат их успешного прохождения.

```
describe('Фаззинг-тесты для аутентификации', () => {
  test('Регистрация с рандомными данными', async () => {
    await fc.assert(
      fc.asyncProperty(
        fc.string({ minLength: 3, maxLength: 20 }).filter(s => s.trim().length >= 3),
        fc.string({ minLength: 6, maxLength: 30 }).filter(s => s.trim().length >= 6),
        async (username, password) => {
          const res = await registerTestUser(username, password);

          expect([201, 400]).toContain(res.status);

          if (res.status === 201) {
            expect(res.body).toHaveProperty('token');
            expect(res.body).toHaveProperty('userId');
          } else {
            expect(res.body.error).toBe('Данный логин занят');
          }
        }
      ),
      { numRuns: 30 }
    );
  });

  test('Логин с рандомными данными', async () => {
    // Сначала создадим тестового пользователя
    await registerTestUser('testuser', 'testpass123');

    await fc.assert(
      fc.asyncProperty(
        fc.constant('testuser'),
        fc.string({ minLength: 1, maxLength: 50 }),
        async (username, password) => {
          const res = await loginTestUser(username, password);

          expect([200, 401]).toContain(res.status);

          if (res.status === 200) {
            expect(res.body).toHaveProperty('token');
          } else {
            expect(res.body.error).toMatch(/Неверные логин или пароль/);
          }
        }
      ),
      { numRuns: 20 }
```

Рисунок 3.11 – Часть кода написанных тестов

```
PASS ./fuzzTests.test.js (14.008 s)
  Фаззинг-тесты для аутентификации
    √ Регистрация с рандомными данными (2706 ms)
    √ Логин с рандомными данными (1730 ms)
  Фаззинг-тесты для чатов
    √ Создание чатов с рандомными именами (2239 ms)
    √ Добавление пользователей в чат с рандомными данными (1854 ms)
  Фаззинг-тесты для сообщений
    √ Отправка сообщений с рандомным текстом (1729 ms)
    √ Получение сообщений с рандомными параметрами (172 ms)
  Фаззинг-тесты для управления ролями
    √ Назначение ролей с рандомными данными (160 ms)
    √ Назначение тегов должностей с рандомными данными (161 ms)

Test Suites: 1 passed, 1 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        14.11 s, estimated 16 s
Ran all test suites matching /fuzzTests.test.js/i.
```

Рисунок 3.12 – Результаты тестов

Теперь протестируем через Postman, заранее возьмем токен авторизации и попробуем получить список чатов пользователя, рисунок 3.13.

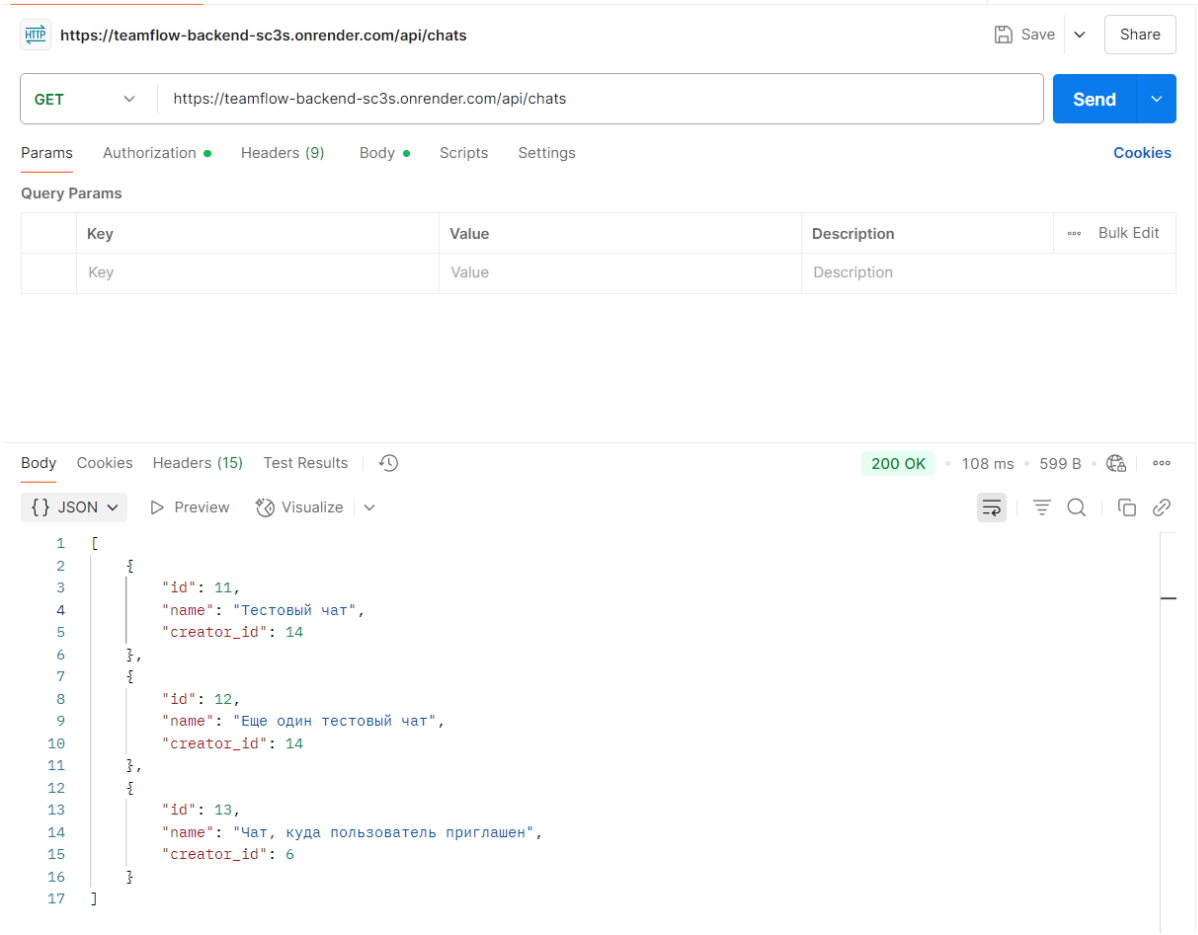

Рисунок 3.13 – GET запрос списка чатов пользователя

Попробуем получить список сообщений из чата с айди 11, рисунок 3.14.

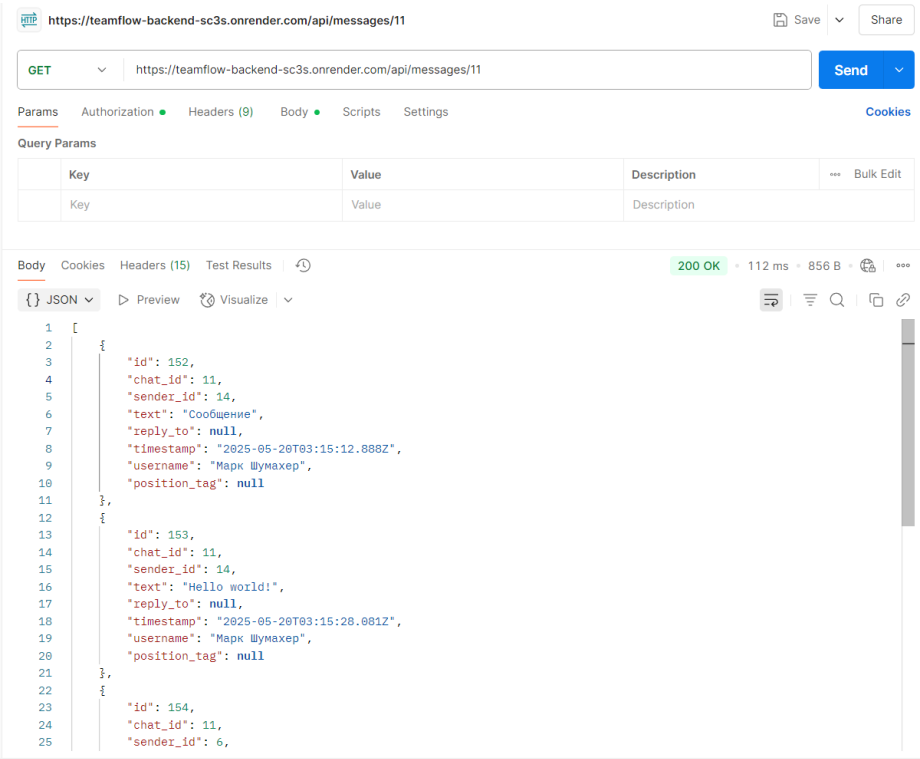

Рисунок 3.14– GET запрос получения сообщений из чата

Теперь попробуем назначить тег участнику noise, рисунок 3.15.

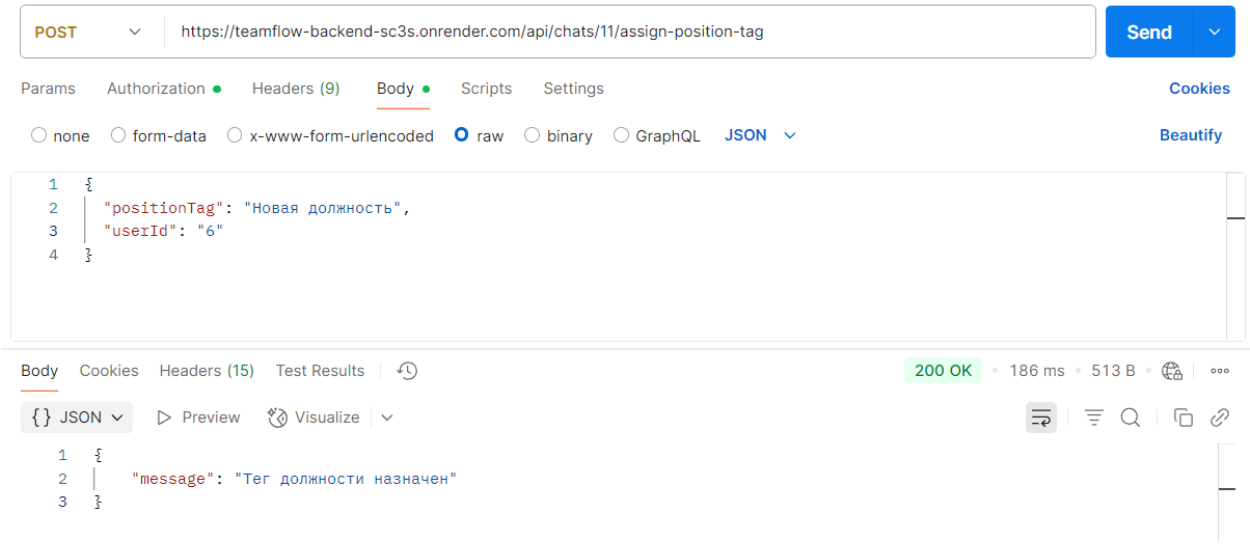

Рисунок 3.15– POST запрос на установку тега

Сделаем тоже самое, но в чате, где нет прав администратора, рисунок 3.16.

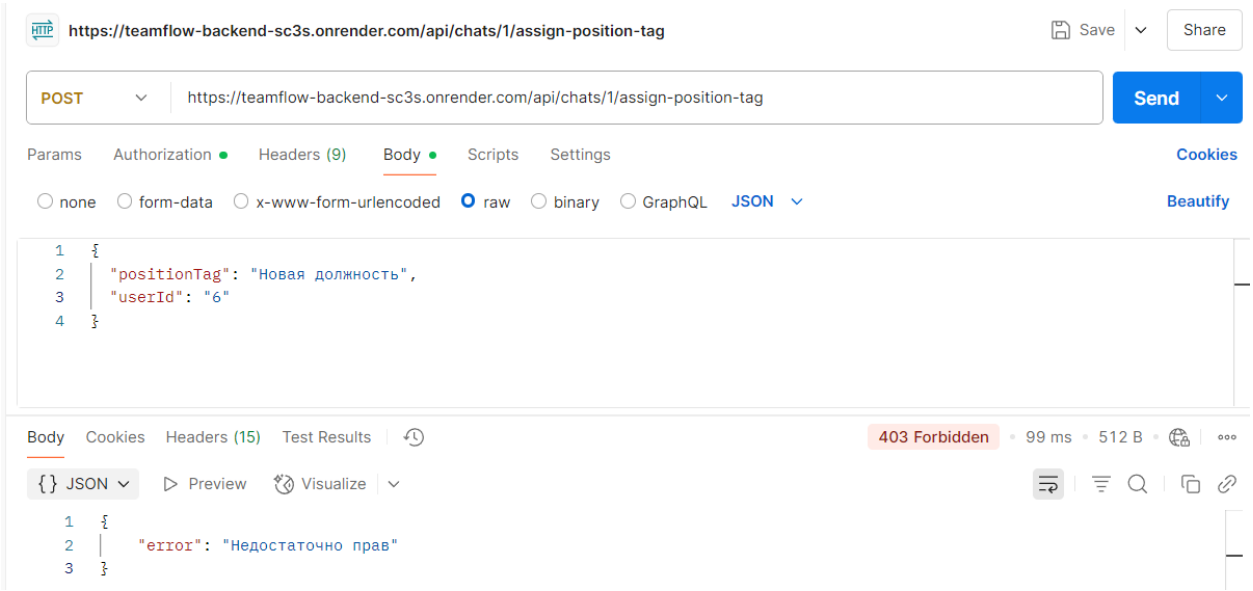

Рисунок 3.16– POST запрос на установку тега без прав администратора

Далее были проведены аналогичные тесты для всех оставшихся конечных точек, как с помощью Postman, так и с помощью браузера, все тесты были проведены успешно.

3.4 Контейнеризация

Чтобы развернуть разработанную систему необходимо прописать 2 файла Dockerfile, на одном будет находиться бекенд, на другом фронтенд, рисунки 3.17,3.18.

```
FROM node:18-alpine

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 5000

CMD ["npm", "start"]
```

Рисунок 3.17 – Файл Dockerfile для baceknd

```
FROM node:18-alpine as build

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .

RUN npm run build

FROM nginx:alpine

COPY --from=build /app/build /usr/share/nginx/html

COPY nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Рисунок 3.18 – Файл Dockerfile для frontend

Также был проведен процесс развертывания приложения на веб-хостинге Render, рисунок 3.19,3.20.

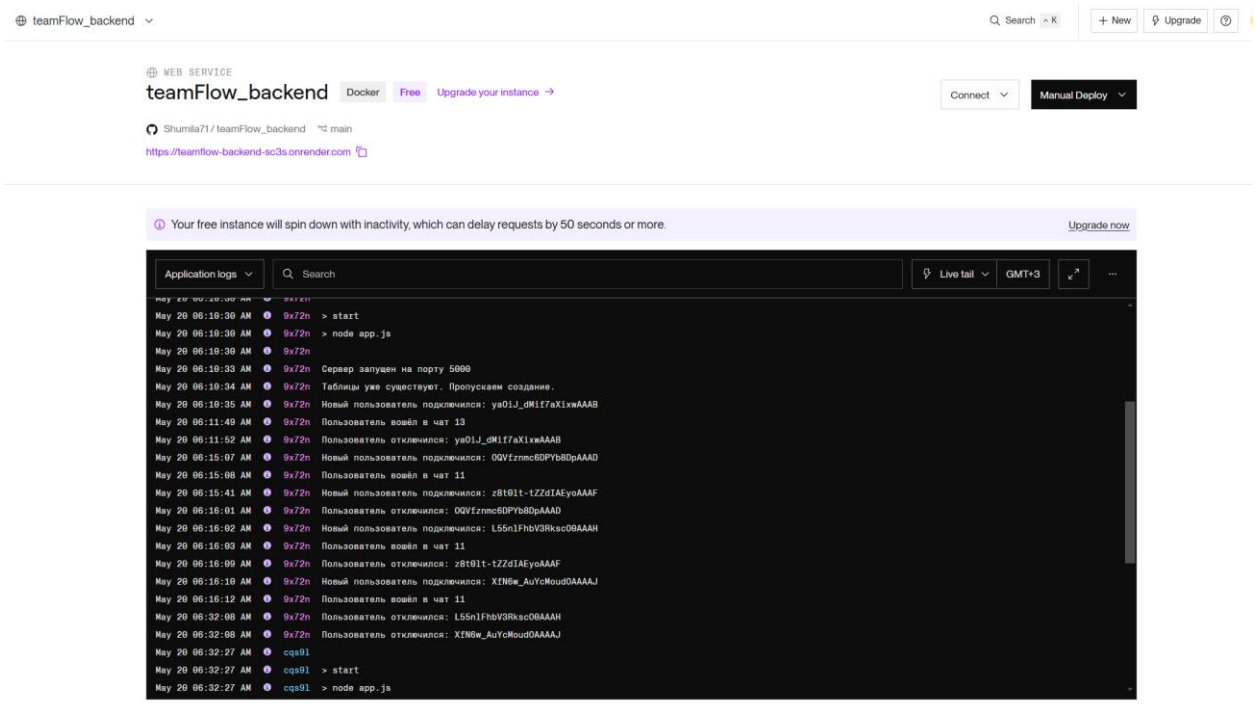

Рисунок 3.19 – Развертывание backend на Render

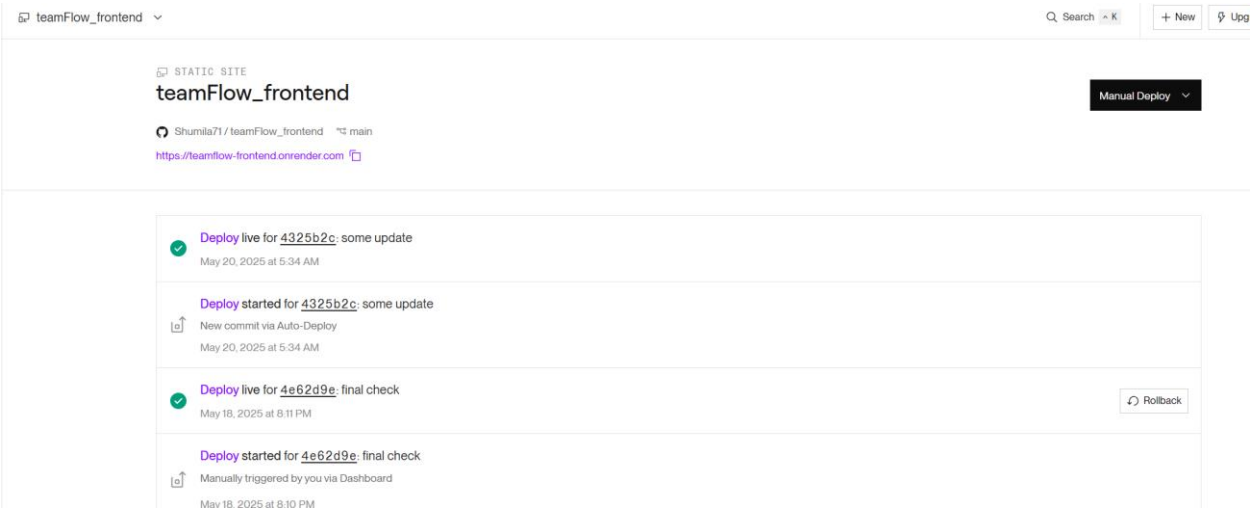

Рисунок 3.20 – Развертывание frontend на Render

ЗАКЛЮЧЕНИЕ

В ходе разработки клиент-серверного чат-приложения «TeamFlow» был проведен комплексный анализ современных архитектурных решений и технологических стеков. На основании исследования предметной области и оценки требований к проекту была выбрана монолитная архитектура с элементами MVC-паттерна, адаптированная под специфику веб-приложений на базе Express.js. Такой подход доказал свою эффективность для проекта данного масштаба, позволив в сжатые сроки реализовать весь необходимый функционал, включая систему аутентификации, чат-комнаты, обмен сообщениями в реальном времени через WebSocket, а также ролевое управление доступом.

В результате работы были достигнуты все поставленные цели и задачи: определены ключевые требования и ограничения системы, проанализированы инструменты и средства разработки. Итогом стало разработанное и протестированное приложение, которое демонстрирует стабильную работу, обладает интуитивно понятным интерфейсом и предоставляет пользователям необходимый функционал для обмена сообщениями и создания чатов.

Для ознакомления с разработанным приложением можно обратиться к исходному коду, который расположен на сайте GitHub: https://github.com/Shumila71/TeamFlow, а также само приложение, развернутое на Render – https://teamflow-frontend.onrender.com.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. JavaScript documentation [Электронный ресурс]. – URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (дата обращения 11.05.2025).

2. React documentation [Электронный ресурс]. – URL: https://react.dev/learn (дата обращения 11.05.2025).

3. PostgreSQL [Электронный ресурс]. – URL: https://www.postgresql.org/ (дата обращения 11.05.2025).

4. Microsoft Teams [Электронный ресурс]. – URL: https://teams.live.com (дата обращения 11.05.2025).

5. Telegram web [Электронный ресурс]. – URL: https://web.telegram.org/ (дата обращения 11.05.2025).

6. Compass [Электронный ресурс]. – URL: https://getcompass.ru/ (дата обращения 11.05.2025).

7. UML-диаграммы [Электронный ресурс]. – URL: https://www.uml-diagrams.org/ (дата обращения 14.05.2025).

8. Draw.io [Электронный ресурс]. – URL: https://app.diagrams.net/ (дата обращения 14.05.2025).

9. VS Code [Электронный ресурс]. – URL: https://code.visualstudio.com/ (дата обращения 11.05.2025).

10. Meet WebStorm [Электронный ресурс]. – URL: https://www.jetbrains.com/help/webstorm/meet-webstorm.html (дата обращения 11.05.2025).

11. Express 5.x [Электронный ресурс]. – URL: https://expressjs.com/en/api.html дата обращения 11.05.2025).

12. FastAPI [Электронный ресурс]. – URL: https://fastapi.tiangolo.com/ (дата обращения 11.05.2025).

13. Spring Boot [Электронный ресурс]. – URL: https://spring.io/projects/spring-boot (дата обращения 11.05.2025).

14. MongoDB [Электронный ресурс]. – URL: https://www.mongodbmanager.com/ (дата обращения 12.05.2025).

15. MySQL [Электронный ресурс]. – URL: https://www.mysql.com/ (дата обращения 13.05.2025).

16. Render Quickstarts [Электронный ресурс]. – URL: https://docs.render.com/ (дата обращения 15.05.2025).

17. Docker Docs [Электронный ресурс]. – URL: https://docs.docker.com/ (дата обращения 15.05.2025).