Міністерство освіти і науки України
Криворізький національний університет
Кафедра моделювання та програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти бакалавра

зі спеціальності 121 – Інженерія програмного забезпечення

На тему: Розробка веб-застосунку для реального часового обміну повідомленнями з використанням WebSocket та сучасних методів автентифікації

Засвідчую, що в цій кваліфікаційній роботі немає запозичень із праць інших авторів без відповідних посилань.
Студент гр. ІПЗ–22–1
_____/Р. А. Бінюков /

Керівник кваліфікаційної роботи _____ / А. В. Козиков /

Завідувач кафедри _____ / А. М. Стрюк /

Кривий Ріг
2026

Криворізький національний університет
Факультет: Інформаційних технологій
Кафедра: Моделювання та програмного забезпечення
Ступінь вищої освіти: бакалавр
Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедри
_____ А. М. Стрюк
«_» _____ 2026р.

ЗАВДАННЯ
на кваліфікаційну роботу

студенту групи ІПЗ–22–1 Бінюков Роман Андрійович

1. Тема: Розробка веб-застосунку для реального часового обміну повідомленнями з використанням WebSocket та сучасних методів автентифікації затверджено наказом по КНУ № 284 від «28» травня 2026 р.
2. Термін подання студентом закінченої роботи: «05» червня 2026 р.
3. Вихідні дані по роботі: розроблюваний веб-застосунок реального часу повинен забезпечувати обмін повідомленнями між користувачами, керування чатами та профілями, а також збереження й обробку даних користувачів у захищеному середовищі з автентифікацією та авторизацією.
4. Зміст пояснювальної записки (перелік питань, що їх треба розробити): виконати аналіз існуючих веб-застосунків для обміну повідомленнями та технологій реалізації систем реального часу, визначити вимоги до функціональності та інтерфейсу веб-застосунку, спроєктувати архітектуру системи онлайн-чатів, здійснити програмну реалізацію на базі ASP.NET Core WebAPI та React, реалізувати взаємодію з базою даних для збереження користувачів, чатів і повідомлень, забезпечити інтеграцію механізмів автентифікації та авторизації, а також провести тестування розробленого веб-застосунку.
5. Перелік ілюстративного матеріалу: діаграма архітектури веб-застосунку, схема взаємодії клієнта і сервера, діаграма модулів системи, діаграма

потоків обміну повідомленнями в режимі реального часу, знімки екранних форм інтерфейсу.

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Дослідження наукових джерел та огляд існуючих вебсистем обміну повідомленнями в режимі реального часу	19.02.2026 – 28.02.2026
2	Аналіз підходів до реалізації взаємодії в режимі реального часу у веб-застосунках	28.03.2026 – 02.04.2026
3	Аналіз методів забезпечення автентифікації та авторизації користувачів у веб-застосунках	11.03.2026 – 20.03.2026
4	Формування та систематизація матеріалів першого розділу роботи	21.03.2026 – 31.03.2026
5	Проєктування архітектурної структури системи та побудова моделі даних чат-застосунку	01.04.2026 – 15.04.2026
6	Оформлення матеріалів другого розділу роботи	16.04.2026 – 24.04.2026
7	Розробка серверної та клієнтської частин веб-застосунку обміну повідомленнями	25.04.2026 – 18.05.2026
8	Тестування програмного забезпечення та підготовка матеріалів третього розділу роботи	19.05.2026 – 25.05.2026
9	Аналіз результатів та підготовка матеріалів четвертого розділу роботи	26.05.2026 – 31.05.2026
10	Оформлення пояснювальної записки	01.06.2026 – 05.06.2026

Дата видачі завдання: «15» лютого 2026 р.
Студент _____/ Р. А. Бінюков /
Керівник роботи _____/ А. В. Козиков /

РЕФЕРАТ

ВЕБ-ЗАСТОСУНОК, МЕСЕНДЖЕР, ASP.NET CORE, REACT, SIGNALR, JWT, POSTGRESQL, РЕАЛЬНОЧАСОВА КОМУНІКАЦІЯ, АВТЕНТИФІКАЦІЯ, БАЗА ДАНИХ.

Пояснювальна записка: 78 с., 3 табл., 28 рис., 1 дод., 9 джерел.

Метою кваліфікаційної роботи є розробка веб-застосунку для організації обміну повідомленнями з мінімальною затримкою доставки даних, що забезпечує гнучке створення чатів та інтерактивне керування користувацькими профілями.

У межах предметної області досліджено сучасні підходи до проєктування комунікаційних вебсервісів. Основний акцент зроблено на механізмах синхронної передачі повідомлень між клієнтами та стратегіях масштабування архітектури за умов зростання навантаження.

Програмну архітектуру декомпоновано на незалежні функціональні модулі: керування користувачами, модерація чатів та обробка повідомлень. Такий підхід гарантував сувору ізоляцію бізнес-логіки та суттєво спростив подальше розширення функціоналу.

Як основне реляційне сховище використано СУБД PostgreSQL. Для оптимізації швидкодії та мінімізації звернень до бази даних у сценаріях частого читання імплементовано проміжний шар кешування на базі Redis.

Безперебійна маршрутизація повідомлень реалізована за допомогою технології SignalR, що забезпечує підтримку постійного з’єднання між клієнтом і сервером та гарантує доставку даних без ініціації повторних запитів.

Захист інформаційного контуру побудовано на базі стандарту JWT, де комбінація короткотривалого токена доступу (access token) та

механізму його оновлення (refresh token) суттєво знижує ризики компрометації сесій. На додаток інтегровано можливість авторизації через зовнішнього провайдера Google.

Серверну частину імплементовано за допомогою ASP.NET Core Web API, тоді як клієнтський інтерфейс розроблено на базі бібліотеки React (TypeScript) із застосуванням складальника Vite. Це дозволило чітко розмежувати зони відповідальності між фронтендом і бекендом та забезпечити їхній незалежний розвиток.

Отриманий продукт являє собою веб-застосунок для обміну повідомленнями, який підтримує як приватні, так і групові чати та може бути використаний як базова платформа для подальшого розвитку систем реального часу.

ABSTRACT

WEB APPLICATION, CHAT SYSTEM, ASP.NET CORE, REACT, SIGNALR, JWT, POSTGRESQL, REAL-TIME MESSAGING, AUTHENTICATION, DATABASE.

Thesis in 78 p., 3 tab., 28 fig., 1 app., 9 references.

The objective of this qualification thesis is to develop a web application for messaging with minimal data delivery latency, facilitating chat creation and interactive user profile management.

Within the domain area, modern approaches to designing communication web services were investigated. Emphasis was placed on mechanisms for synchronous message transmission among concurrent clients and architectural scaling strategies under increasing loads.

The software architecture is decomposed into independent functional modules: user management, chat moderation, and message processing. This approach ensured strict isolation of business logic and significantly simplified future functional extensions.

PostgreSQL is utilized as the primary relational database. To optimize performance and minimize database queries in frequent read scenarios, an intermediate caching layer based on Redis was implemented. Seamless message routing is achieved using SignalR technology, which maintains persistent client-server connections and guarantees data delivery without polling.

The security perimeter is built upon the JWT standard, where the combination of a short-lived access token and a refresh mechanism significantly mitigates the risks of session compromise. Additionally, external authentication via Google provider is integrated.

The backend is implemented using ASP.NET Core Web API, while the client interface is developed with the React library (TypeScript) utilizing the Vite bundler. This enabled a clear separation of concerns between the frontend and backend, ensuring their independent development.

System validation was performed by testing core user workflows, including chat creation, message exchange, authentication, and profile operations. The results confirmed stable system behavior and correct interaction between components.

The final outcome is a web-based messaging application supporting private and group communication, serving as a foundation for further development of real-time systems.

ЗМІСТ

ВСТУП

Експоненційне зростання обсягів цифрової інформації та глобалізація комунікаційних процесів зумовлюють критичну потребу в надійних інструментах для миттєвого обміну даними. Сьогодні інтерактивні вебплатформи є невіддільним елементом як корпоративного, так і приватного спілкування, проте їхнє проєктування вимагає розв'язання складних архітектурних завдань: від мінімізації затримок передачі повідомлень до надійного управління сесіями користувачів. Оскільки традиційні моделі синхронної клієнт-серверної взаємодії не завжди здатні забезпечити належну продуктивність та інтерактивність, розробка спеціалізованого вебдодатка для безперебійної онлайн-комунікації в реальному часі набуває безперечної актуальності.

Використання базових підходів до маршрутизації даних часто призводить до розсинхронізації стану чатів та вразливостей у безпеці. Зважаючи на це, виникає об’єктивна необхідність у створенні комплексної системи на базі ASP.NET Core та React. Її головним завданням є оптимізація процесу двосторонньої передачі повідомлень (за допомогою технології SignalR), гарантування безпечного зберігання профільної інформації та медіафайлів (PostgreSQL, Cloudinary), а також забезпечення стійкості до спам-атак (reCAPTCHA) і гнучкості авторизації (JWT, Google Auth). Запропонований програмний продукт повністю відповідає сучасним інженерним стандартам веброзробки, формуючи масштабоване та відмовостійке середовище для безпечної взаємодії користувачів.

1 АКТУАЛЬНІСТЬ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1.1 Критичний аналіз літературних джерел за темою

Проблематика проєктування вебдодатків для миттєвого обміну повідомленнями становить значний інтерес як у теоретичній, так і в прикладній інформатиці. Нині дослідники зосереджують увагу на питаннях оптимізації двостороннього зв'язку, кешуванні даних та безпечній інтеграції інтерактивних компонентів у сучасні цифрові екосистеми.

У фаховій літературі, присвяченій розробці на базі платформи ASP.NET Core, ґрунтовно досліджено ключові аспекти побудови програмних архітектур [1]. Дослідники акцентують на важливості строгого розмежування бізнес-логіки, клієнтського інтерфейсу (зокрема за допомогою бібліотеки React) та шару доступу до даних. Такий підхід гарантує високу підтримуваність коду та полегшує подальше масштабування програмного продукту.

Окрема увага у наукових працях приділяється механізмам реляційного зберігання та швидкого доступу до інформації [2]. Використання СУБД PostgreSQL у зв'язці з in-memory сховищем Redis забезпечує суттєві переваги: мінімізацію часу відгуку сервера, надійне кешування запитів та ефективну обробку структурованих даних за умов високого навантаження.

Важливим вектором досліджень є оптимізація алгоритмів передачі даних у реальному часі. У низці робіт аналізуються протоколи WebSockets та розглядаються методи усунення колізій під час багатокористувацької взаємодії [3]. Проте більшість існуючих архітектурних шаблонів орієнтована на базові сценарії трансляції повідомлень і не враховує специфічних потреб комплексної модерації чатів із розгалуженою системою ролей.

Критичний огляд наявних рішень для організації комунікації виявляє їхню структурну неповноту. Значна частина платформ позбавлена інтегрованих механізмів гнучкого управління правами доступу,

персоналізованих профілів або надійного захисту від автоматизованих бот-атак [4]. Це свідчить про наявність незадоволеного попиту на ринку програмного забезпечення та підкреслює доцільність розробки більш захищеного й функціонального вебмесенджера.

Цільовою аудиторією подібних платформ є користувачі, які потребують створення приватних або публічних кімнат для швидкого спілкування: від невеликих робочих груп до масштабних спільнот. Ці особи формують власні інформаційні простори, самостійно керують складом учасників та вимагають наявності безперебійного, інтуїтивно зрозумілого каналу зв'язку.

З огляду на це, сучасна комунікаційна платформа має задовольняти такі ключові потреби:

а) Безперебійна інтерактивна взаємодія: забезпечення миттєвого управління повідомленнями та гнучкої системи адміністрування чатів без необхідності перезавантаження клієнтського застосунку;

б) Комплексний захист ідентифікації: реалізація багаторівневої автентифікації з інтеграцією сторонніх провайдерів (Google Auth), обов'язкової верифікації електронних адрес та надійного екранування від автоматизованих атак;

в) Оптимізація ресурсів: підвищення загальної швидкодії системи шляхом застосування механізмів кешування та делегування обробки медіафайлів спеціалізованим хмарним сервісам.

Відповідно, запропонований програмний продукт володіє низкою суттєвих переваг:

а) Архітектурна цілісність: синергія чуйного інтерфейсу на базі React та системи керування з'єднаннями Microsoft SignalR у форматі єдиного вебдодатка (SPA);

б) Посилені стандарти безпеки: імплементація технології JWT із механізмом оновлення токенів, що суттєво мінімізує ризики компрометації сесій користувачів;

в) Портативність середовища: застосування технологій контейнеризації (Docker) та автоматизації (Make) для швидкого й прогнозованого розгортання проєкту.

У сучасному науково-практичному дискурсі часто аналізуються традиційні засоби комунікації, проте їхня архітектура не адаптована до вимог обміну даними в реальному часі. Наприклад, класичні вебплатформи форумного типу, зокрема популярна система управління контентом phpBB (рис. 1.1), ефективно забезпечують асинхронне обговорення тем, але не підтримують миттєвої синхронізації станів без ресурсомістких запитів до сервера та позбавлені засобів динамічного керування локальними сесіями.

З огляду на ці обмеження, виникає об'єктивна необхідність у розробці спеціалізованого програмного забезпечення, здатного забезпечити високу продуктивність бекенду та безперебійну інтерактивність фронтенду.

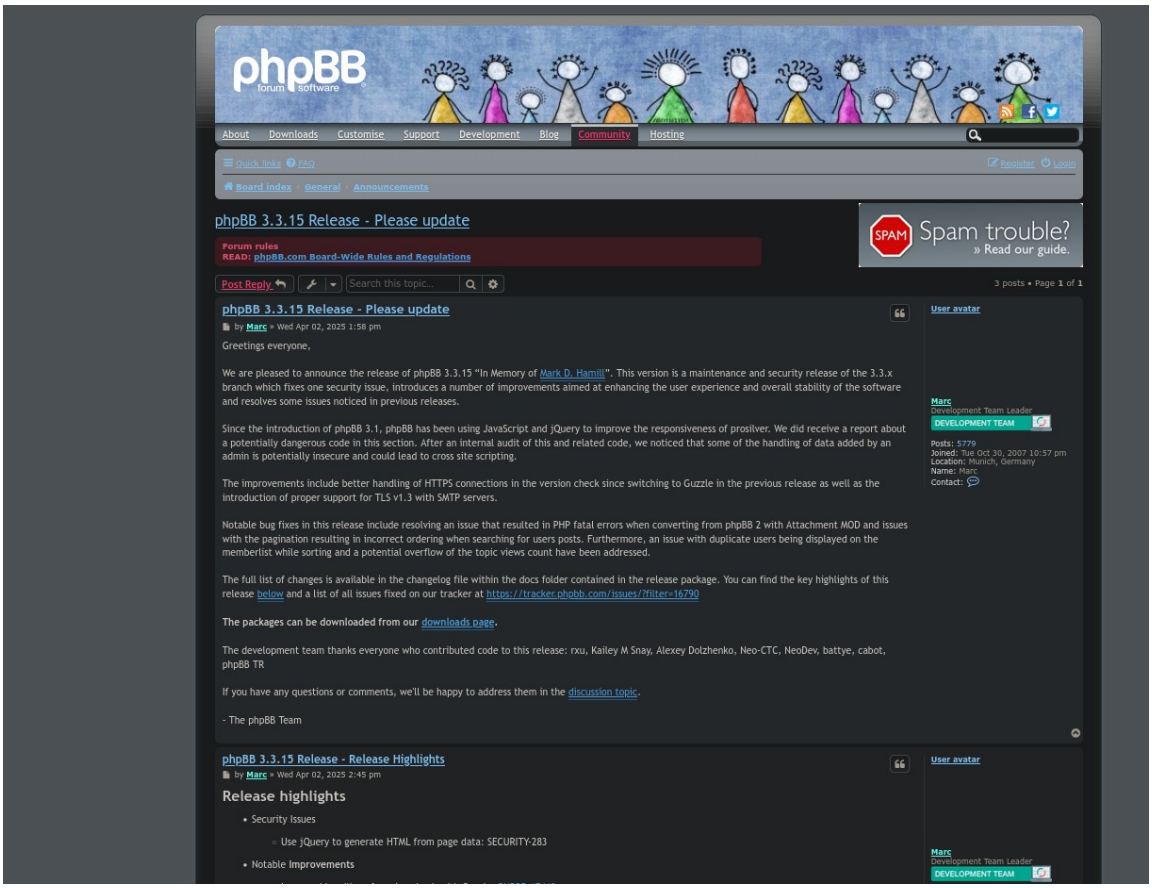

Рисунок 1.1 – phpBB

Широким інструментарієм володіють комплексні корпоративні платформи, зокрема Slack, що позиціонуються як універсальні екосистеми для організації командної взаємодії [5]. Типове представлення робочого простору в таких системах, ілюстроване на рисунку 1.2, демонструє інтеграцію багатьох супутніх модулів. Проте надмірна функціональність цих платформ зумовлює високий поріг входження для пересічного користувача, а архітектурна орієнтація на великий бізнес робить їх занадто громіздкими для швидкого приватного спілкування. Окрім цього, інтегровані в них механізми захисту сесій не завжди забезпечують необхідний баланс між безпекою та швидкістю доступу без застосування корпоративних протоколів.

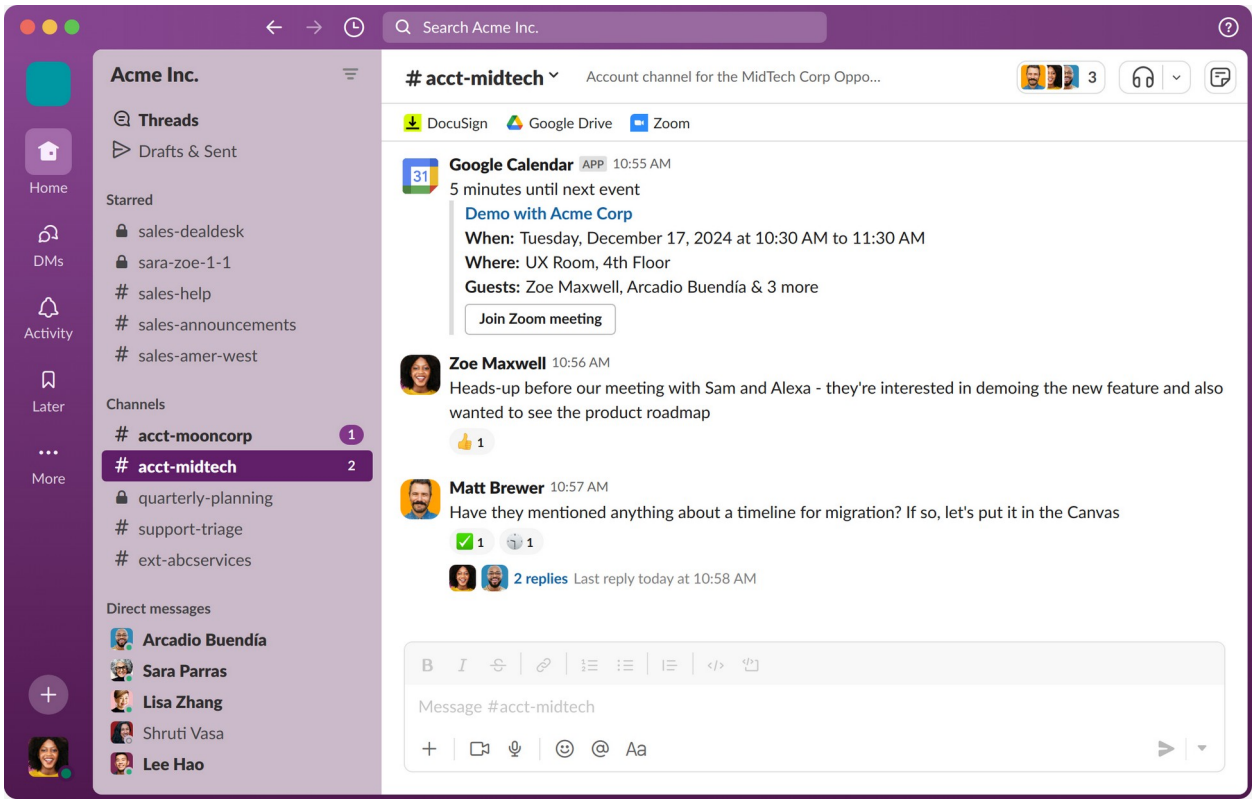

Рисунок 1.2 – Інтерфейс корпоративної платформи Slack

Для ведення групових текстових та голосових дискусій користувачі часто застосовують платформу Discord. Її перевагами є розвинена канальна структура та можливість підключення сторонніх програмних розширень [6]. Водночас подібні рішення перевантажені медіаелементами та ігровими

сервісами (рис. 1.3), що ускладнює їхнє мобільне використання та створює ризики випадкової зміни налаштувань кімнат недосвідченими учасниками.

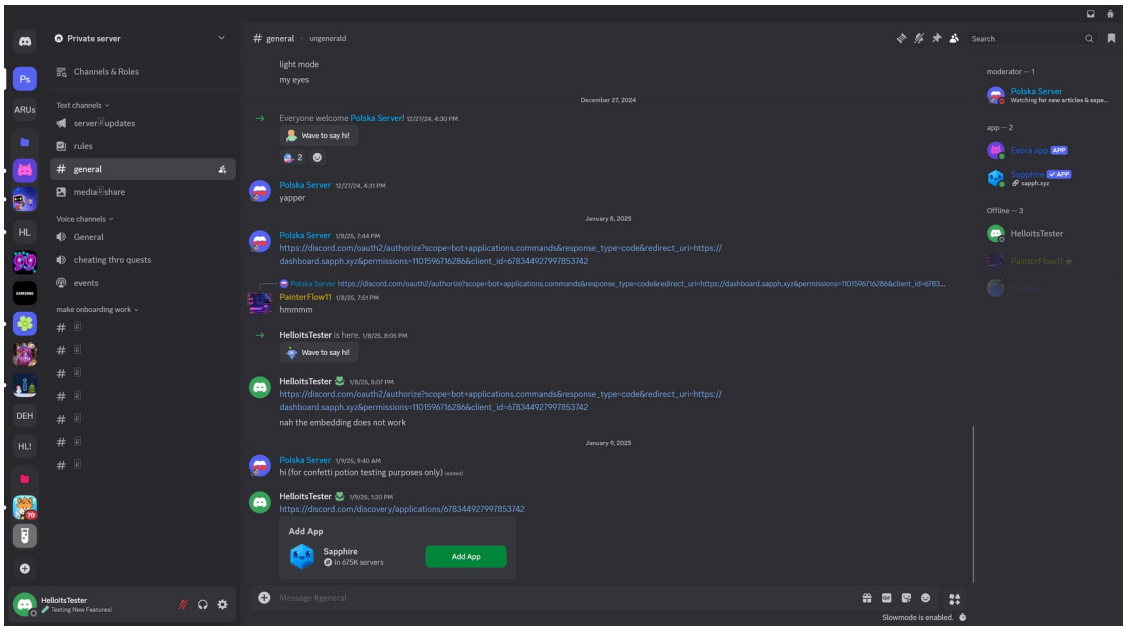

Рисунок 1.3 – Інтерфейс комунікаційної платформи Discord

На противагу цим незграбним архітектурним моделям, розроблений вебдодаток на базі React та ASP.NET Core пропонує лаконічне комунікаційне середовище, що функціонує в режимі реального часу. Ключовою особливістю системи є імплементація технології SignalR: після відправки або редагування повідомлення стан інтерфейсу у всіх учасників чату оновлюється за частки секунди, а очищення чи модерація кімнати власником виконується миттєво без потреби ручного оновлення сторінки.

У межах дослідження проведено порівняльний аналіз наявних програмних аналогів, які найчастіше використовуються для командної взаємодії та обміну інформацією: вебплатформи форумного типу (phpBB) як представники асинхронного підходу та універсальні комунікаційні системи (Slack, Discord) як приклади багатофункціональних екосистем синхронного зв'язку.

Таблиця 1.1 – Аналіз існуючих аналогів

Критерій порівняння	phpBB (Вебфоруми)	Slack / Discord (Екосистеми)	Розроблений вебдодаток
Модель обміну даними	Асинхронна (базові HTTP-запити)	Синхронна (WebSockets)	Синхронна (імплементація Microsoft SignalR)
Архітектура клієнтської частини	Багатосторінкова (MPA), потребує перезавантажень	Ресурсомісткі SPA або настільні клієнти	Легковаге Single Page Application (React)
Управління сесіями та авторизація	Стандартна (cookies, логін/пароль)	Багаторівнева (OAuth, SSO)	Гібридна (JWT + Refresh token, Google Auth)
Захист від автоматизовани х атак	Базові модулі капчі (часто вразливі)	Внутрішні закриті алгоритми	Інтегрована перевірка Google reCAPTCHA
Зберігання медіафайлів (зображень)	Локальне (навантаження на сервер)	Власні розподілені хмарні кластери	Делеговане спеціалізованому сервісу Cloudinary
Динамічне керування привілеями	Статичне (через панель адміністратора)	Громіздке (багатошарова система ролей)	Інтуїтивне (зміна власника/вилучен ня в 1 клік)
Розгортання та інфраструктура	Ручне налаштування вебсервера	Закритий вихідний код (SaaS)	Автоматизоване (Docker Compose, Make)
Головні архітектурні недоліки	Відсутність миттєвої синхронізації станів	Надмірна перевантаженість побічними функціями	Вузька спеціалізація виключно на чатах (за метою)

Підсумовуючи результати проведеного аналізу, можна констатувати брак збалансованих програмних рішень, які б органічно інтегрували механізми синхронної трансляції повідомлень, гнучкого адміністрування локальних сесій та багаторівневого захисту даних у межах єдиного легковагового вебдодатка (SPA), не обтяженого надмірним корпоративним функціоналом.

1.2 Актуальність теми кваліфікаційної роботи

Стрімкий розвиток інформаційних технологій диктує нові вимоги до швидкості та надійності цифрової комунікації. Зростання потреби користувачів у миттєвому обміні даними зумовлює необхідність переходу від класичних вебсторінок до високоінтерактивних застосунків. Традиційні моделі клієнт-серверної взаємодії, побудовані на синхронних HTTP-запитах, втрачають свою ефективність. Вони перевантажують мережеву інфраструктуру та не гарантують миттєвого доставлення контенту, що робить їх непридатними для сучасних платформ спілкування у режимі реального часу.

Стандартні підходи до створення систем обміну повідомленнями часто ігнорують потребу в безперервному двосторонньому зв'язку. Відсутність оптимізованої архітектури ускладнює адміністрування багатокористувацьких чатів, перешкоджає швидкому доступу до історії листування та не дозволяє реалізувати розширене управління ролями (наприклад, надання привілеїв власника чи блокування учасників).

Використання застарілих або неповних архітектурних рішень під час розробки чатів провокує низку критичних проблем:

а) Виникнення відчутних затримок під час синхронізації даних, що руйнує логіку взаємодії між користувачами;

б) Обмеженість функціоналу для керування контентом, зокрема неможливість миттєвого редагування чи видалення надісланих повідомлень без перезавантаження клієнта;

в) Вразливість до автоматизованих атак та несанкціонованого доступу через слабкі механізми автентифікації.

Подібні технічні недоліки критично знижують продуктивність застосунку та погіршують якість користувацького досвіду. Окрім того, фундаментальним аспектом залишається безпека персональних даних та оптимізація серверного навантаження. Розгортання надійних систем потребує інтеграції багаторівневого захисту (зокрема систем антибот-перевірки та JWT-авторизації), а також механізмів кешування для зменшення кількості звернень до бази даних. Отже, розробка комплексного веб-застосунку, який поєднує технології безперервного зв'язку із сучасними стандартами безпеки та управління даними, є об'єктивною технічною потребою, що підтверджує своєчасність та актуальність обраного напряму дослідження.

Рисунок 1.4 – Інтерфейс обміну повідомленнями в режимі реального часу

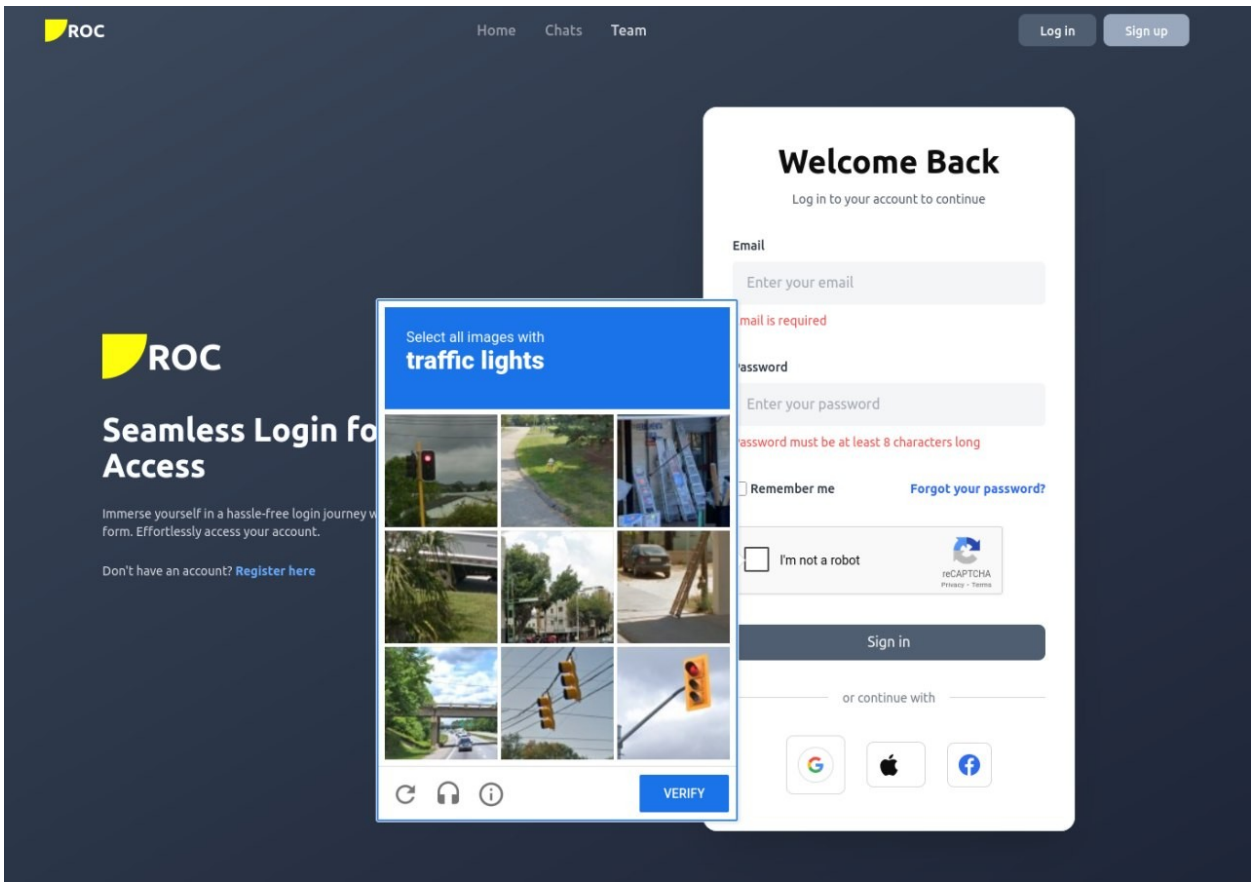

Рисунок 1.5 – Механізм захисту застосунку від автоматизованих атак

Застосування кросплатформних вебтехнологій, зокрема екосистеми React, забезпечує універсальний та безперервний доступ до комунікаційного середовища з будь–якого пристрою.

Така архітектура надає можливість:

а) Здійснювати миттєву маршрутизацію та відображення текстових повідомлень;

б) Динамічно керувати параметрами чатів та ієрархією прав доступу (власник, учасник);

в) Інтегрувати надійні протоколи автентифікації без необхідності створення локальних облікових записів;

г) Оптимізувати обробку та зберігання медіафайлів за допомогою спеціалізованих хмарних платформ.

Інтеграція протоколів постійного з'єднання (наприклад, бібліотеки Microsoft SignalR) у поєднанні з реляційними базами даних та системами

кешування суттєво підвищує продуктивність веб-застосунків. Ці інструменти гарантують надійну персистентність даних, мінімізацію затримок під час мережевої синхронізації та швидке відновлення стану клієнта після розриву з'єднання, що формує беззаперечну перевагу над традиційними підходами до маршрутизації HTTP–запитів [8].

Використання застарілих методів періодичного опитування сервера у системах обміну повідомленнями спричиняє не лише надмірне споживання апаратних ресурсів, а й накопичення критичних помилок синхронізації даних. Брак оптимізованого кеш–шару (на кшталт Redis) призводить до розбіжностей між фактичним станом бази даних та клієнтським інтерфейсом користувача. У результаті сервер марнує обчислювальні потужності на обробку надлишкових запитів, замість забезпечення стабільного потоку двосторонньої комунікації.

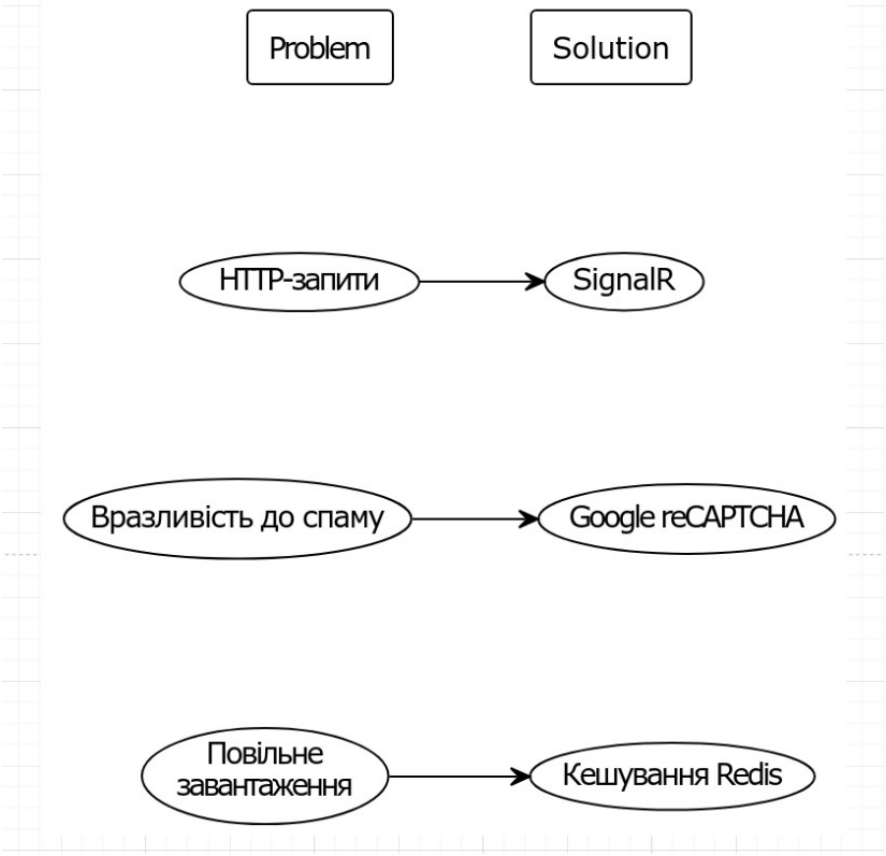

Рисунок 1.6 – Архітектурні недоліки класичних вебчатів та шляхи їх вирішення

Вразливість до автоматизованого спаму та несанкціонованого доступу залишається однією з найгостріших проблем незахищених систем. Некоректна обробка сесій (відсутність JWT з механізмом Refresh–токенів) або ігнорування систем протидії ботам генерують нерелевантне навантаження на сервери та погіршують якість користувацького досвіду [7]. Такі інциденти негативно позначаються на репутації платформи та знижують рівень довіри до неї.

Окремим технологічним викликом є втрата або затримка пакетів даних під час інтенсивного листування. Відсутність надійного механізму керування подіями може спричинити десинхронізацію, коли повідомлення не доставляються адресату вчасно, або порушується логіка надання привілеїв. Подібні збої критично знижують цінність комунікаційного програмного забезпечення.

Необхідність проєктування спеціалізованих систем реального часу зумовлена стрімким переходом суспільства до цифрових форматів взаємодії. Сучасні користувачі вимагають високої швидкості та безпеки обміну інформацією, що потребує впровадження надійних архітектурних патернів. За таких умов веб-застосунок, побудований на базі ASP.NET Core та оптимізований для миттєвої комунікації, перетворюється з простого інструменту на комплексне рішення для безперебійної цифрової взаємодії.

1.3 Цілі та завдання кваліфікаційної роботи

Стрімкий розвиток вебтехнологій та підвищення вимог до швидкості цифрової комунікації зумовлюють необхідність створення оптимізованих платформ для миттєвого обміну даними. Сучасний користувач потребує інтегрованого середовища, яке поєднує надійні механізми авторизації, безперебійне доставлення повідомлень та зручне адміністрування профілю.

Використання традиційних підходів до маршрутизації на базі періодичних HTTP-запитів часто супроводжується критичними архітектурними недоліками:

а) Відчутною затримкою під час синхронізації стану чатів між клієнтами;
б) Неефективним споживанням апаратних ресурсів сервера через відсутність багаторівневого кешування;
в) Високою вразливістю до автоматизованих атак та деградацією продуктивності під час пікових навантажень.

Відсутність оптимізованої інфраструктури ускладнює масштабування системи та призводить до погіршення користувацького досвіду. Відтак, виникає об’єктивна технічна потреба у проєктуванні програмного рішення, яке централізовано розв'язує проблеми двосторонньої взаємодії. Інтеграція протоколів постійного з'єднання дозволяє мінімізувати мережеві втрати та забезпечити безперервність комунікаційного процесу.

1.3.1 Мета кваліфікаційної роботи

Головною метою дослідження є проєктування та програмна реалізація клієнт-серверного веб-застосунку, орієнтованого на безпечний обмін повідомленнями в режимі реального часу.

Для розв'язання поставленої проблеми архітектура системи має забезпечувати:

а) Встановлення стабільного двостороннього зв'язку між клієнтськими застосунками за допомогою технології SignalR;
б) Впровадження комплексної системи ідентифікації, що включає авторизацію на базі JWT-токенів, інтеграцію OAuth-провайдерів та превентивний захист від ботів;
в) Оптимізацію роботи з даними через делегування збереження медіафайлів хмарним сервісам та використання in-memory кешування.

Висока відмовостійкість та продуктивність створеної системи гарантуватиметься завдяки поєднанню надійного реляційного сховища із сучасними патернами проєктування веб-API.

1.3.2 Завдання кваліфікаційної роботи

Для досягнення поставленої мети необхідно розв'язати таку послідовність науково-практичних завдань:

а) Провести системний аналіз предметної області, дослідивши архітектурні патерни побудови систем обміну повідомленнями у реальному часі та виявивши вузькі місця традиційної клієнт-серверної взаємодії;

б) Сформулювати специфікацію вимог до програмного продукту, визначивши ключові функціональні модулі (маршрутизація повідомлень, управління ролями у чатах) та нефункціональні критерії (відмовостійкість, безпека);

в) Спроєктувати серверну інфраструктуру та логіку обробки даних, що включає розробку реляційної схеми у PostgreSQL з використанням інструментарію Entity Framework Core, налаштування in-memory кешування через Redis для прискорення доступу до ендпойнтів та імплементацію хабів SignalR для забезпечення постійного двостороннього зв'язку;

г) Реалізувати інтегровану систему захисту інформації, налаштувавши генерацію JWT-токенів, механізм оновлення сесій, OAuth-авторизацію через сервіси Google та превентивну перевірку запитів за допомогою reCAPTCHA;

д) Створити інтерактивний клієнтський застосунок на базі екосистеми React (Vite, TailwindCSS), забезпечивши динамічне керування станом, миттєве відображення історії листування та управління профілем користувача;

е) Виконати верифікацію та тестування розробленої системи, перевіривши коректність роботи API, цілісність делегованого збереження медіафайлів у хмарному сховищі Cloudinary та стабільність WebSocket-з'єднань під навантаженням;

Послідовна реалізація зазначених етапів забезпечує комплексний інженерний підхід до повного життєвого циклу розробки програмного забезпечення. Ґрунтовний аналіз наявних архітектурних рішень на початковій стадії дозволяє запобігти виникненню типових помилок під час проєктування високонавантажених систем. Водночас детальна розробка структури бази даних та кеш-шару закладає міцний фундамент для подальшого масштабування застосунку. Підсумкове тестування інтеграційних процесів виступає критично важливим етапом, який підтверджує стійкість серверної частини до збоїв та гарантує цілісність передачі даних між клієнтами.

1.3.3 Очікувані результати

У підсумку виконання кваліфікаційної роботи передбачається створення повнофункціонального веб-застосунку для комунікації в режимі реального часу. Очікується, що розроблений програмний продукт характеризуватиметься такими технічними показниками:

а) Забезпеченням безперервної двосторонньої взаємодії між клієнтами завдяки інтеграції технології SignalR;
б) Наявністю багаторівневої системи безпеки, яка включає JWT-автентифікацію, OAuth-провайдери та превентивний захист від автоматизованого трафіку;
в) Високою продуктивністю маршрутизації даних, що досягається шляхом використання in-memory кешування та делегування обробки медіафайлів хмарним сервісам;
г) Адаптивним користувацьким інтерфейсом із можливістю динамічного керування правами доступу, профілями та контентом чатів.

Впровадження спроєктованого рішення дозволить суттєво оптимізувати клієнт-серверну взаємодію. Відмова від традиційних синхронних запитів на користь постійного WebSocket-з'єднання мінімізує

мережеві затримки, усуваючи проблему десинхронізації стану застосунку. Водночас використання строгої реляційної схеми гарантує абсолютну цілісність історії листування та надійність збереження облікових даних.

Архітектура створеної системи закладає міцний фундамент для подальшого масштабування. Модульна структура бекенду дозволяє безболісно розширювати функціонал, наприклад, додаючи системи глибокої аналітики, механізми відеозв'язку або інтеграцію зі сторонніми месенджерами. Отримані результати практично підтвердять ефективність обраного технологічного стека для побудови сучасних, безпечних та стійких до навантажень комунікаційних вебплатформ.

1.4 Завдання кваліфікаційної роботи

Повний цикл створення веб-застосунку для обміну повідомленнями охоплює етапи від проєктування архітектури до фінального тестування інфраструктури. Оскільки сучасні комунікаційні платформи вимагають високої швидкості та стійкості до навантажень, робота зосереджена на побудові оптимізованого клієнт-серверного рішення.

Виконання дослідження передбачає реалізацію таких технологічних завдань:

а) проєктування масштабованої серверної частини (API) на базі ASP.NET Core із застосуванням реляційної бази даних PostgreSQL;

б) інтеграцію протоколів реального часу (SignalR) та механізмів in-memory кешування (Redis) для мінімізації мережевих затримок;

в) розробку інтерактивного клієнтського інтерфейсу з використанням бібліотеки React, а також впровадження систем багаторівневої автентифікації та захисту від ботів.

Комплексний підхід до розробки гарантує не лише програмне втілення, а й створення надійного інструменту, стійкого до вебнавантаження. Автоматизація маршрутизації даних та управління сесіями дозволяє уникнути критичних вразливостей і забезпечити безперебійну роботу чатів.

1.4.1 Етапи виконання завдань та розробки додатка

Процес розробки розпочався з дослідження архітектурних патернів сучасних чат-систем. Аналіз виявив, що класична модель синхронних HTTP-запитів не здатна забезпечити миттєве доставлення контенту, що зумовило потребу в переході до протоколів постійного з'єднання. На основі цих висновків було сформовано сучасну клієнт-серверну архітектуру з чітким розділенням відповідальності (Separation of Concerns) між фронтендом, бекендом та хмарним сховищем медіафайлів.

Наступним кроком став вибір інструментарію. Серверну логіку реалізовано мовою C# (платформа .NET Core), що забезпечує сувору типізацію та високу продуктивність обробки запитів. Для організації двостороннього зв'язку інтегровано технологію SignalR. Клієнтську частину побудовано за допомогою екосистеми React (з використанням збирача Vite), що гарантує швидкий рендеринг компонентів інтерфейсу [9].

Завершальна стадія передбачала комплексне тестування розроблених модулів. Перевірці підлягали коректність генерації JWT-токенів, стабільність роботи WebSocket-каналів, цілісність збереження зображень через API Cloudinary та стійкість системи до автоматизованих спам-запитів за допомогою Google reCAPTCHA.

1.4.2 Практична цінність реалізованих завдань

Практична значущість розробленого веб-застосунку полягає у створенні відмовостійкої комунікаційної платформи, готової до розгортання в реальному середовищі (зокрема через контейнеризацію Docker). Завдяки впровадженню технологій реального часу усунено проблему затримок синхронізації – користувачі можуть миттєво відправляти, редагувати та видаляти повідомлення без перезавантаження сторінки.

Спроєктована система вирізняється високим рівнем безпеки. Інтеграція сервісу reCAPTCHA та OAuth-провайдерів унеможливлює несанкціонований

доступ, а кешування запитів у Redis суттєво знижує навантаження на основну базу даних PostgreSQL. Універсальність запропонованих архітектурних рішень дозволяє легко масштабувати створений застосунок або інтегрувати його комунікаційні модулі у більші корпоративні портали.

2 РОЗРОБКА ФУНКЦІОНАЛЬНОЇ СХЕМИ І АЛГОРИТМУ

2.1 Розробка та докладний опис функціональної схеми програми

Проектований веб-застосунок функціонує як комплексна платформа для обміну повідомленнями в режимі реального часу. Головна концепція системи полягає у забезпеченні безперервної комунікації, де користувачі можуть миттєво створювати чати, керувати контентом та взаємодіяти з іншими учасниками.

З технічної точки зору архітектура платформи базується на чіткому розподілі обов’язків між клієнтською та серверною частинами. Інтерфейс користувача (Frontend), побудований за допомогою React, функціонує незалежно від бізнес-логіки (Backend на базі ASP.NET Core WebAPI) та механізмів збереження даних.

Відповідно до цього, функціональну структуру застосунку поділено на низку взаємопов’язаних модулів:

а) Підсистема ідентифікації та управління користувачами: охоплює процеси безпечної реєстрації, генерації JWT-токенів, верифікації за допомогою Google reCAPTCHA, а також кастомізацію профілів з інтеграцією сервісу Cloudinary для обробки медіаконтенту;

б) Підсистема інтерактивної взаємодії та адміністрування: реалізована на базі бібліотеки SignalR для миттєвого обміну повідомленнями та надає розширені інструменти для створення чат-кімнат, пошуку, а також управління правами учасників (передача статусу власника, вилучення користувачів);

в) Підсистема збереження та кешування даних: гарантує цілісність інформації завдяки використанню СУБД PostgreSQL спільно з Entity Framework Core, паралельно підвищуючи загальну швидкодію платформи через механізми кешування запитів у Redis.

Упровадження такого підходу суттєво оптимізує процес онлайн-спілкування: замість розрізнених і повільних інструментів користувач отримує єдине, високопродуктивне середовище. Функціональна схема застосунку ілюструє базову послідовність обміну даними між клієнтським інтерфейсом та серверною частиною під час доступу до системи. Зокрема, первинні дії користувача, пов'язані з процесами реєстрації та авторизації, генерують стандартні HTTP-запити, які проходять обробку на рівні API з подальшою валідацією облікових даних та оновленням безпекових токенів у базі.

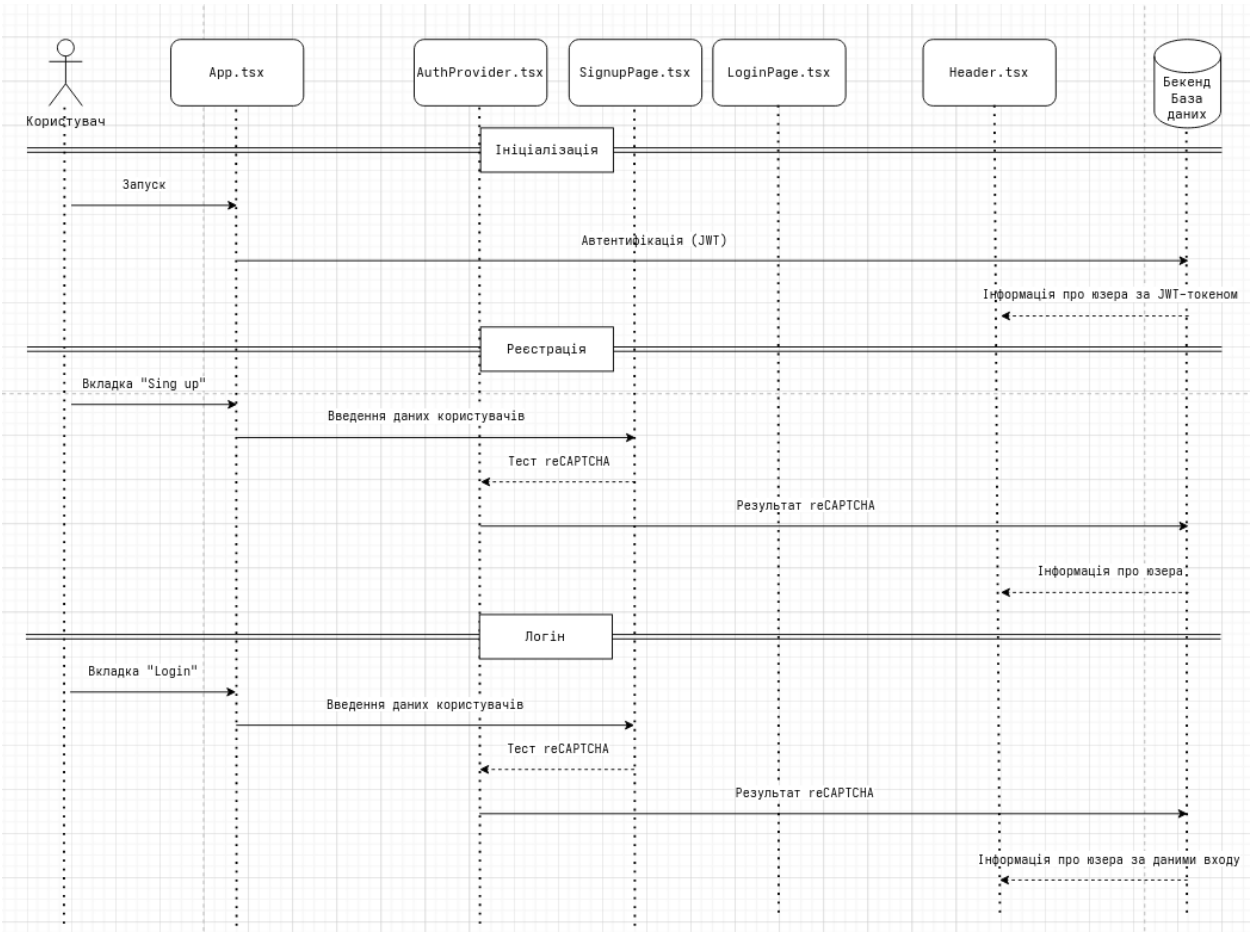

Рисунок 2.1 – Сценарій взаємодії компонентів системи автентифікації

Водночас центральним елементом комунікаційної архітектури виступає підсистема WebSocket-з’єднань, що забезпечує безпосереднє функціонування чату. Після успішної авторизації надсилання текстових повідомлень генерує вже не HTTP-запити, а події реального часу. Завдяки інтеграції бібліотеки

SignalR клієнтські застосунки підтримують постійний двосторонній зв’язок із сервером, що уможливлює миттєву синхронізацію стану чатів та статусів учасників в активній сесії.

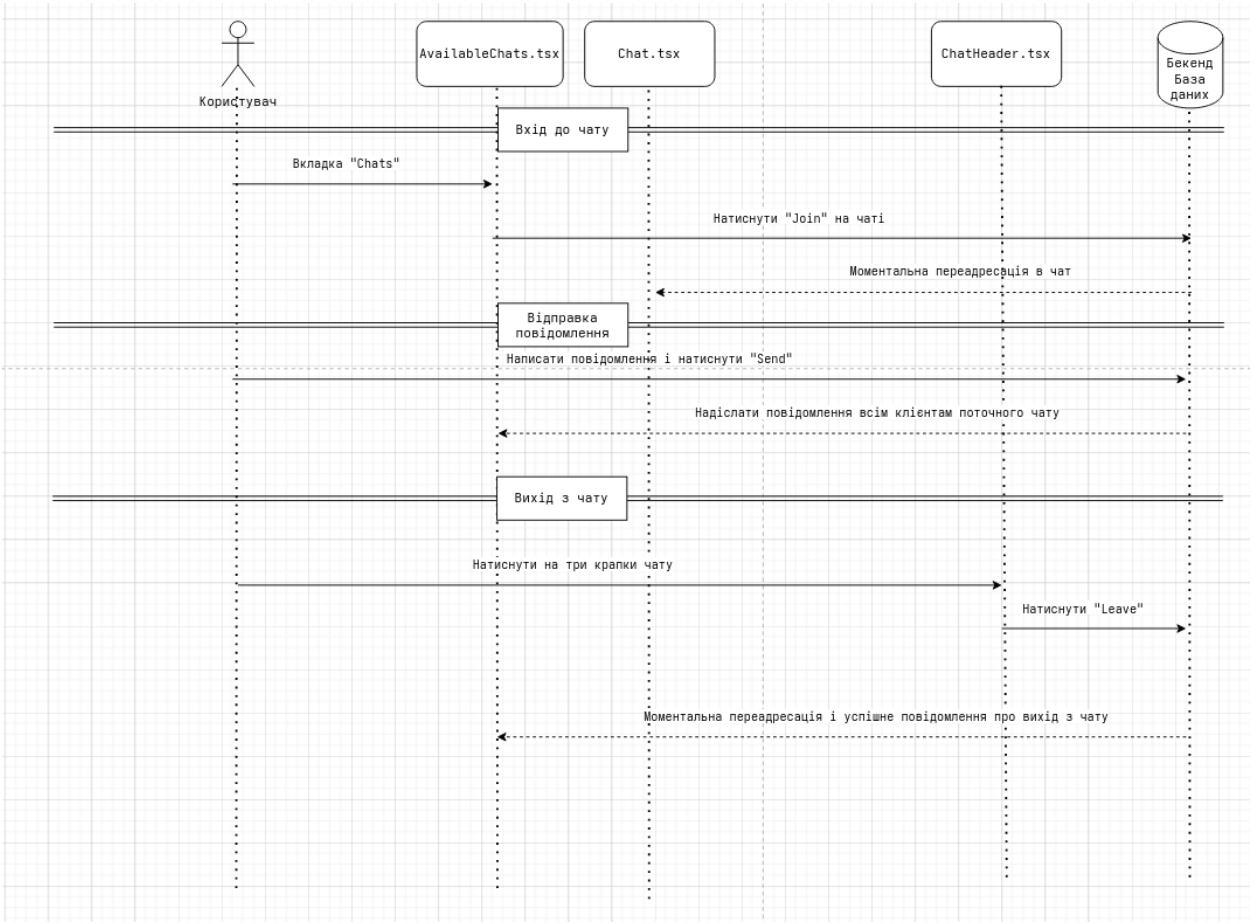

Рисунок 2.2 – Сценарій взаємодії компонентів системи чату

2.2 Розробка та докладний опис алгоритму роботи програми

Основним рушієм платформи виступає механізм реального часу, який забезпечує безперервність комунікаційного процесу. Під час ініціалізації клієнтський застосунок виконує запит до сервера для отримання списку доступних чатів та завантаження профілю користувача. Після вибору конкретної кімнати алгоритм автоматично підвантажує історію листування з бази даних та встановлює активне WebSocket-з’єднання через бібліотеку SignalR.

Процес генерації нового повідомлення ініціює складний внутрішній

цикл обробки. Текстова інформація миттєво проходить попередню валідацію на стороні React-клієнта, після чого транслюється до WebAPI. На цьому етапі система перевіряє права доступу користувача та криптографічну валідність JWT-токена, щоб запобігти несанкціонованому втручанню.

Функціонування застосунку базується на перманентній взаємодії між користувацьким інтерфейсом та бекендом. Будь-яка активність – редагування надісланого тексту, видалення чату чи делегування прав власника іншому учаснику – негайно опрацьовується логічним рівнем системи та фіксується у СУБД PostgreSQL. Це дає змогу гарантувати абсолютну консистентність даних та унеможливлює розсинхронізацію стану додатка в авторизованих учасників сесії.

Для мінімізації навантаження на основну базу даних алгоритм інтегрує використання Redis-кешування. Наприклад, під час пошуку бесід за назвою система спершу звертається до швидкої пам’яті, що значно пришвидшує відмальовування інтерфейсу та знижує затримки.

Важливою складовою алгоритму є суворий контроль безпеки. Перед етапами реєстрації або авторизації програма обов'язково вимагає проходження Google reCAPTCHA. Це виключає ймовірність автоматизованого створення фейкових акаунтів (ботів) та забезпечує цілісність екосистеми чатів.

2.3 Алгоритм процесу програми з автоматизацією ручної праці

Головна мета розроблених алгоритмічних рішень полягає у забезпеченні безперервної та безпечної взаємодії користувачів із системою чатів. Завдяки подієво-орієнтованій архітектурі процеси отримання контенту та верифікації сесій повністю автоматизовано, що мінімізує затримки та гарантує захист даних.

Ключовим елементом цієї підсистеми є алгоритм опрацювання вхідних повідомлень, який глибоко інтегровано з механізмами контролю доступу. Процес розпочинається в момент надходження нових даних, що

візуалізується для користувача через оновлення лічильника непрочитаних сповіщень. Коли учасник бесіди ініціює дію прочитання, клієнтський застосунок запускає автоматичну фонову перевірку криптографічної валідності поточного JWT-токена.

Для підтримання стабільного користувацького досвіду алгоритм включає механізм прозорої регенерації ключів. Якщо термін дії основного токена вичерпано, система непомітно для користувача виконує спробу його оновлення через «Refresh Token». Успішне генерування нової пари токенів дозволяє зняти блокування та відобразити вміст повідомлення. У разі ж невдалої спроби оновлення генерується системна помилка авторизації, що примусово завершує активний сеанс.

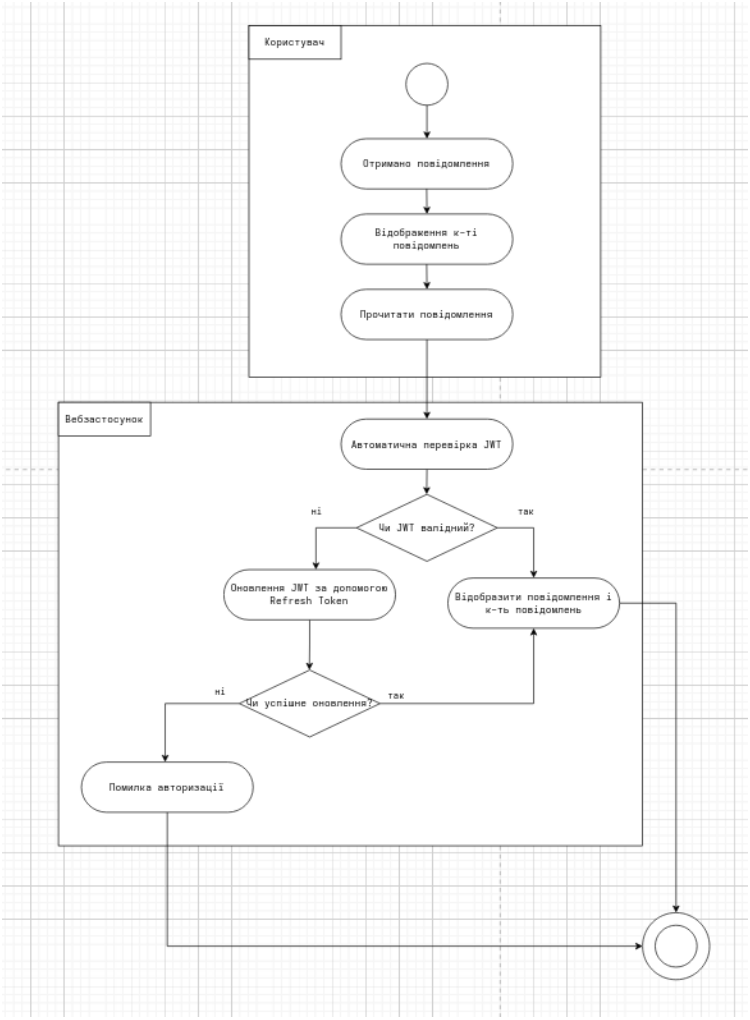

Рисунок 2.3 – Алгоритм опрацювання вхідних повідомлень та валідації токенів доступу

2.4 Алгоритм ініціалізації та завантаження даних

Процес старту клієнтського веб-застосунку побудований за принципом поступового монтування інтерфейсних елементів, що гарантує швидке відображення базової структури сторінки паралельно з виконанням асинхронних фонових запитів. Як проілюстровано на розробленій блок-схемі, життєвий цикл завантаження розпочинається з ініціалізації кореневого модуля App.tsx, який відповідає за первинну маршрутизацію та обробку глобального стану програми.

Після успішного відмальовування базового контейнера система здійснює перехід до основного робочого простору – компонента Chat.tsx. Саме тут запускається алгоритм ініціалізації активної бесіди, що включає встановлення постійного двостороннього зв'язку із сервером через протокол WebSocket. Фінальним етапом процесу є асинхронне звернення до бази даних для отримання актуального переліку активних користувачів кімнати та підвантаження історії повідомлень, після чого клієнтський застосунок стає повністю готовим до обміну інформацією в реальному часі.

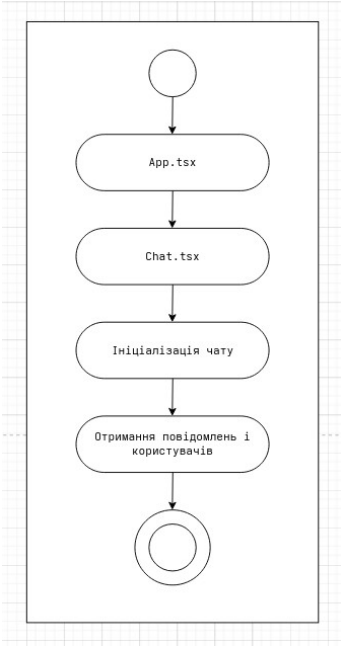

Рисунок 2.4 – Схема алгоритму ініціалізації чату та завантаження даних

2.4.1 Алгоритм надсилання та видалення повідомлень

Оптимізація взаємодії користувача з чатом досягається шляхом інтеграції базових операцій управління контентом у єдиний безперервний цикл обробки. Як проілюстровано на розробленій блок-схемі, після завантаження кореневого модуля App.tsx та переходу до основного інтерфейсу Chat.tsx, користувач отримує доступ до двох паралельних сценаріїв маніпуляції даними.

Перший сценарій (створення допису): після відправлення тексту сервер зберігає дані в PostgreSQL, а SignalR-хаб миттєво транслює клієнтам сигнал для оновлення інтерфейсу.

Альтернативний сценарій активується під час видалення обраного повідомлення. Користувач ініціює дію в графічному інтерфейсі, після чого сервер виконує очищення запису безпосередньо в таблицях бази даних PostgreSQL. Далі SignalR-хаб миттєво транслює всім підключеним учасникам команду на вилучення елемента з клієнтських екранів, що гарантує абсолютну узгодженість стану чату в реальному часі.

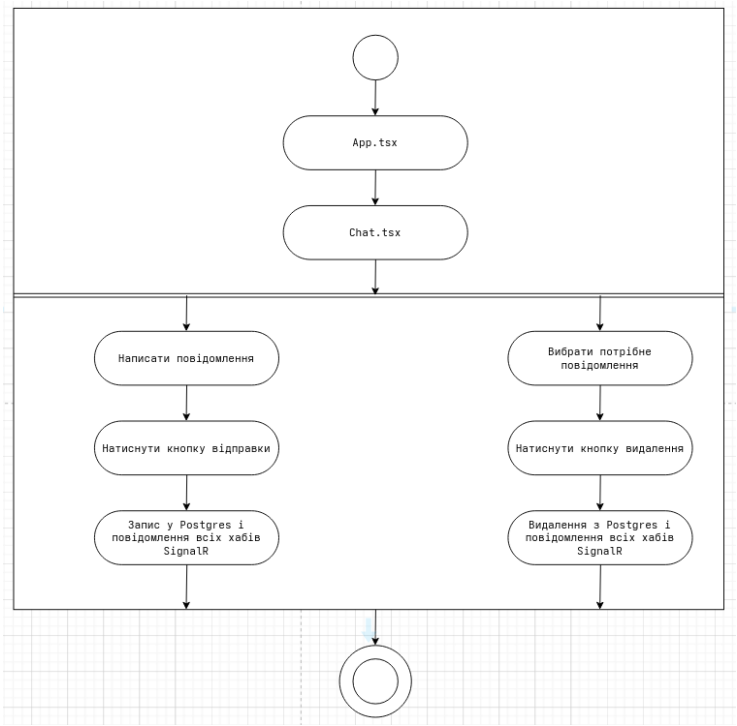

Рисунок 2.5 – Схема алгоритму надсилання та видалення повідомлень

2.4.2 Алгоритм створення нової бесіди

Основою розширення комунікаційного простору застосунку є механізм ініціалізації нових бесід. Цей процес забезпечує формування ізольованого середовища для обміну повідомленнями, яке прив'язується до поточного користувача та автоматично інтегрується в загальну архітектуру системи.

Як відображено на наведеній блок-схемі, навігаційний потік алгоритму розпочинається з кореневого модуля App.tsx, звідки виконується перехід до компонента AvailableChats.tsx, котрий відповідає за рендеринг переліку доступних кімнат. Процес безпосереднього створення активується взаємодією з відповідним елементом управління (кнопкою «Create chat»), що викликає появу форми для введення початкових параметрів. Після зазначення цільової назви бесіди та підтвердження дії ініціюється процес генерації нової сутності. Завершальним кроком алгоритму є миттєва автоматична переадресація користувача до робочого простору новоствореного чату (Chat.tsx), що дозволяє уникнути зайвих кроків навігації та без затримок розпочати спілкування.

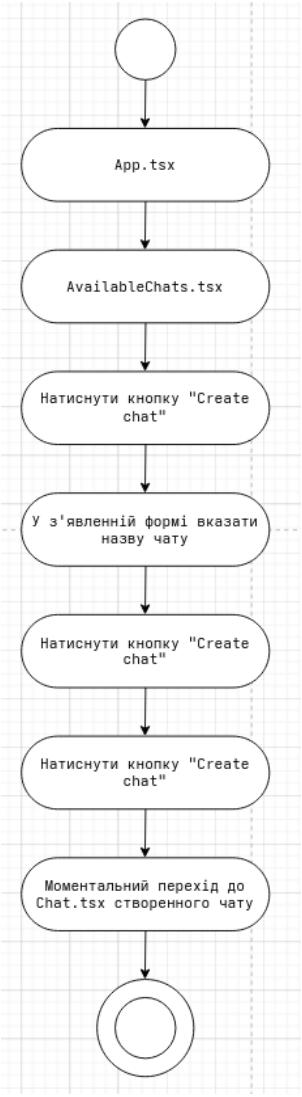

Рисунок 2.6 – Схема алгоритму створення чат-кімнати

2.4.3 Алгоритм адміністрування чат-кімнати та управління учасниками

Важливою складовою функціоналу застосунку є підсистема адміністрування, яка дозволяє авторизованим особам керувати правами та складом учасників бесіди. Відповідно до розробленої блок-схеми, алгоритм управління ініціюється з базового модуля App.tsx із подальшим переходом до активного робочого простору Chat.tsx. Безпосередній доступ до адміністративних функцій відкривається через взаємодію з контекстним меню, що викликається натисканням іконки у верхній частині інтерфейсу чату.

Після активації меню системна логіка розгалужується на два незалежні

адміністративні сценарії. Перший сценарій описує процес вилучення небажаного учасника з бесіди. Вибір цієї опції ініціює транзакцію видалення зв'язку користувача з чатом у реляційній базі даних PostgreSQL. Одразу після цього через хаби SignalR генерується широкомовне сповіщення, яке примусово переадресовує вигнаного користувача зі сторінки чату до загального переліку доступних кімнат (AvailableChats.tsx).

Другий сценарій призначений для делегування повноважень і дозволяє надати обраному учаснику привілеї адміністратора. У цьому разі система оновлює відповідний статус користувача у базі даних PostgreSQL і так само використовує SignalR для миттєвого сповіщення всіх учасників хабу. Завершальним кроком є локальне оновлення клієнтського інтерфейсу, завдяки чому новий статус адміністратора автоматично та без затримок відображається у загальному списку користувачів чату.

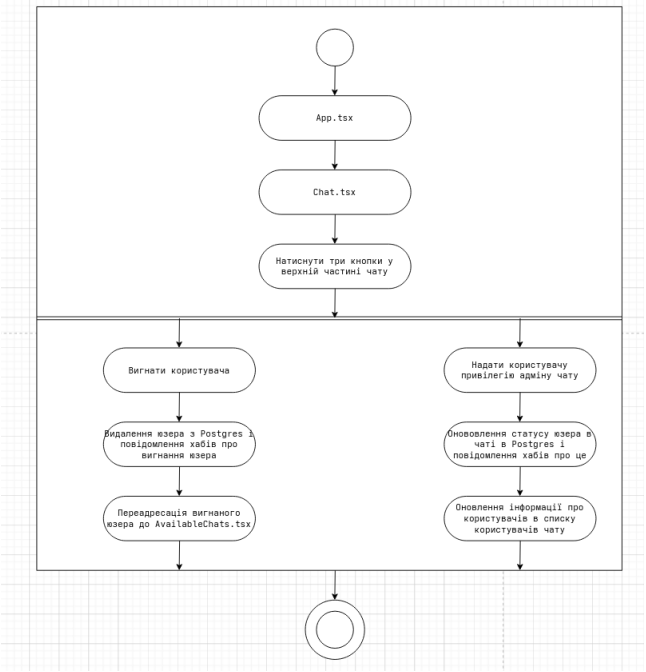

Рисунок 2.7 – Схема алгоритму адміністрування учасників чату

2.4.4 Алгоритм автентифікації користувача

Процес входу до системи ініціюється через базовий модуль App.tsx із подальшим переходом до екрана авторизації LoginPage.tsx. На цьому етапі

алгоритм розгалужується, пропонуючи користувачеві два альтернативні сценарії отримання доступу.

Перший варіант передбачає класичну мануальну авторизацію: користувач вводить електронну пошту та пароль, проходить обов'язкову верифікацію через Google reCAPTCHA для захисту від автоматизованих скриптів та спроб несанкціонованого доступу, після чого підтверджує дію кнопкою входу.

Другий сценарій реалізує механізм швидкої автентифікації (наприклад, через Google), де користувач обирає свій обліковий запис у зовнішньому вікні провайдера. Незалежно від обраного шляху, успішна перевірка даних завершується єдиним результатом: сервер генерує та повертає клієнтському застосунку JWT, Refresh-токен, а також базову інформацію про профіль авторизованого користувача для ініціалізації сесії.

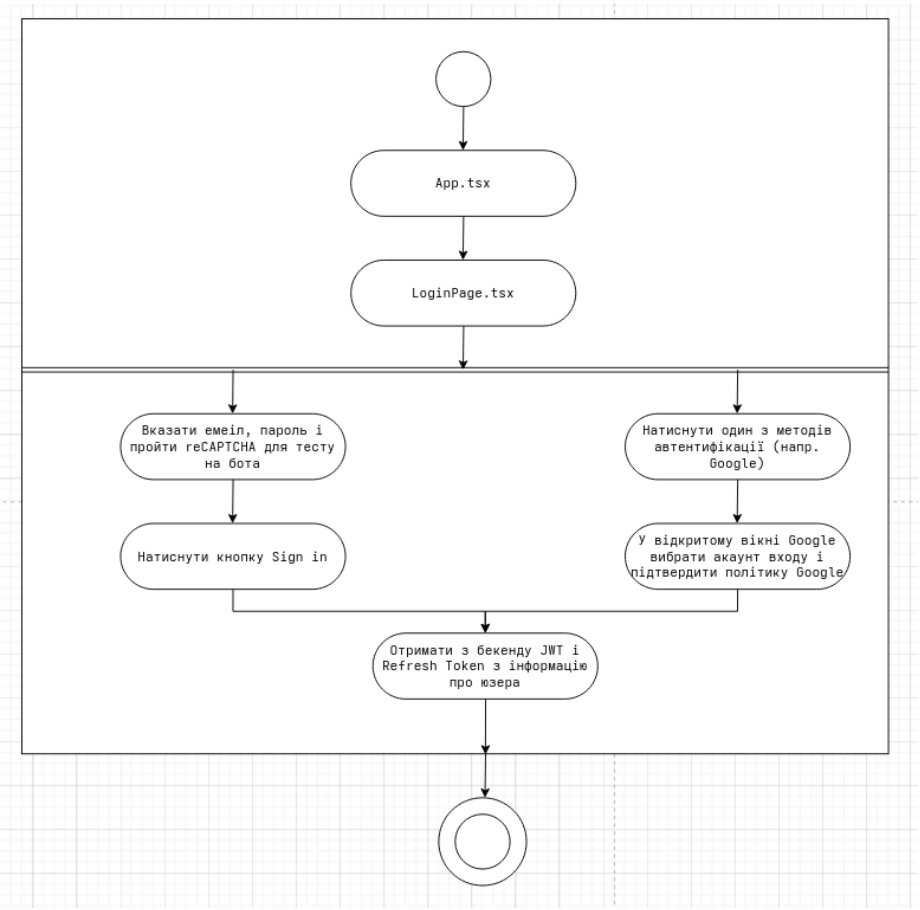

Рисунок 2.8 – Схема алгоритму автентифікації користувача

2.4.5 Алгоритм реєстрації нового користувача

Процес створення облікового запису розпочинається з базового модуля App.tsx та переходу до сторінки реєстрації SignupPage.tsx. Алгоритм пропонує користувачеві два паралельні шляхи ініціалізації профілю.

Перший сценарій – класична реєстрація. Користувач заповнює розширену форму (ім'я, електронна пошта, телефон, пароль), проходить перевірку reCAPTCHA для захисту від ботів та ініціює створення акаунта. Важливою умовою цього шляху є обов'язкове підтвердження електронної пошти за посиланням, що надсилається у системному листі. Альтернативний сценарій (швидка реєстрація) дозволяє скористатися провайдером ідентифікації (наприклад, Google), де користувачеві достатньо обрати свій акаунт у зовнішньому вікні авторизації.

Незалежно від обраного методу, обидва потоки завершуються єдиним системним кроком: відбувається транзакція збереження нового користувача у базі даних PostgreSQL, після чого бекенд генерує та повертає клієнтському застосунку стартові JWT та Refresh-токен.

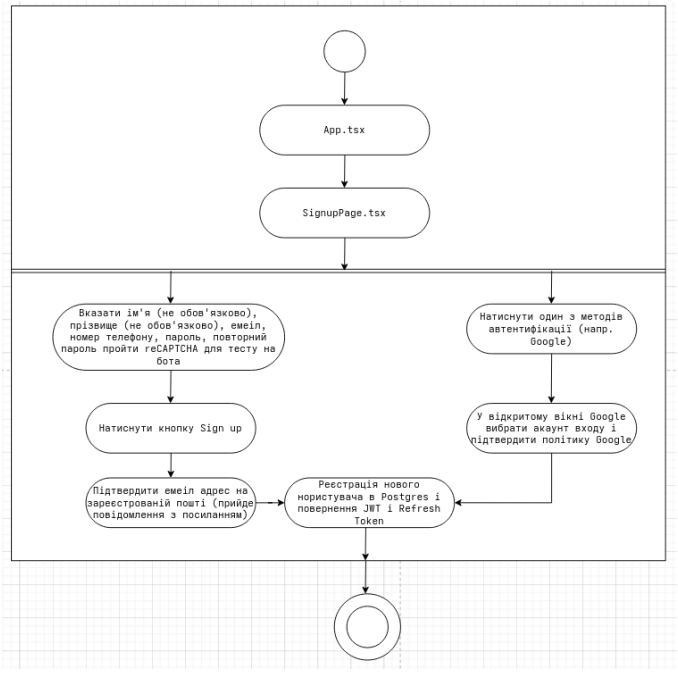

Рисунок 2.9 – Алгоритм реєстрації нового користувача

2.4.6 Реєстр функціональних вимог

Таблиця 2.1 – Вимоги до застосунку

ID Вимоги	Опис вимоги	Критерії приймання
US.01	Як неавторизований користувач, я хочу мати змогу зареєструватися та увійти в систему, щоб отримати доступ до чатів.	1. Форма входу з reCAPTCHA та підтвердженням email. 2. Автентифікація користувачів через акаунт Google. 3. Отримання та збереження токенів JWT/Refresh.
US.02	Як користувач, я хочу створювати нові чат-кімнати, щоб мати ізольований простір для спілкування.	1. Кнопка створення чату та поле назви. 2. Переадресація до робочого простору Chat.tsx.
US.03	Як користувач, я хочу надсилати, читати та видаляти повідомлення, щоб підтримувати комунікацію в реальному часі.	1. Збереження повідомлень у базі даних PostgreSQL. 2. Миттєве оновлення інтерфейсу в усіх учасників через хаби SignalR. 3. Фонова перевірка JWT під час читання.
US.04	Як адміністратор чату, я хочу керувати учасниками бесіди, щоб модерувати спілкування.	1. Контекстне меню для керування учасниками. 2. Вилучення користувача з редиректом через SignalR. 3. Делегування прав із синхронним оновленням статусів.

2.5 Нефункціональні вимоги

Специфікація нефункціональних вимог, що визначають загальні характеристики якості розробленого веб-застосунку, базується на таких критеріях:

а) Надійність функціонування: гарантування суворої цілісності збережених повідомлень та облікових записів у реляційній базі даних PostgreSQL під час виконання транзакцій, а також забезпечення безперебійної клієнт-серверної взаємодії, за якої показник успішної доставки даних наближається до абсолютного максимуму за умови стабільного мережевого з'єднання;

г) Адаптивність та кросплатформність: динамічне пропорційне підлаштування клієнтського React-інтерфейсу під різні роздільні здатності екранів (від мобільних пристроїв до десктопних моніторів) без втрати інформативності та уніфікація середовища виконання завдяки технології Docker, що нівелює залежність працездатності застосунку від конкретної операційної системи;

д) Ергономічність користувацького інтерфейсу: логічно структурована архітектура вебклієнта, що мінімізує когнітивне навантаження та дозволяє новим користувачам інтуїтивно орієнтуватися у системі без попереднього інструктажу, а також мінімізація користувацького шляху для ключових операцій (процес ініціалізації нового чату або редагування профілю потребує здійснення не більше трьох послідовних кліків);

е) Інформаційна безпека та захищеність: блокування автоматизованого спаму та превентивний захист від реєстрації шкідливих ботів шляхом обов'язкової верифікації через алгоритми Google reCAPTCHA, а також захист сесій та конфіденційних даних за допомогою надійного механізму авторизації на основі JSON Web Tokens (JWT) із застосуванням системи оновлення токенів (Refresh tokens);

ж) Експлуатаційна продуктивність та швидкодія: оптимізація маршрутизації повідомлень у режимі реального часу із мінімальними затримками завдяки використанню протоколу WebSockets та бібліотеки SignalR, і пришвидшення генерації відповідей сервера шляхом інтеграції in-memory сховища Redis, що забезпечує відображення оновлених даних в інтерфейсі за частки секунди;

2.6 Проектування інтерфейсу програми

При розробці візуальної частини веб-застосунку ключовий акцент робився на досягненні високого рівня ергономіки та оперативності, залишаючи суто декоративні рішення на другому плані. Після успішної автентифікації користувач одразу потрапляє у робочий простір, де структура екрана дозволяє миттєво оцінити список активних діалогів та статус комунікації.

Архітектура панелі керування побудована за принципом просторової централізації, що інтегрує такі критично важливі компоненти:

а) Модуль пошуку та фільтрації: інструмент для швидкого знаходження потрібної розмови за назвою;

б) Динамічний перелік чатів: область для моніторингу активності та зручної навігації між кімнатами;

в) Блок швидких дій: елементи керування для миттєвої ініціалізації нового діалогу, налаштування профілю або перегляду деталей поточної бесіди.

Завдяки такій компоновці вдалося радикально зменшити глибину вкладеності навігаційного меню. У контексті систем обміну повідомленнями в реальному часі мінімізація дій для доступу до цільової функції (виконання базового сценарію за 2–3 кліки) є фундаментальною вимогою юзабіліті.

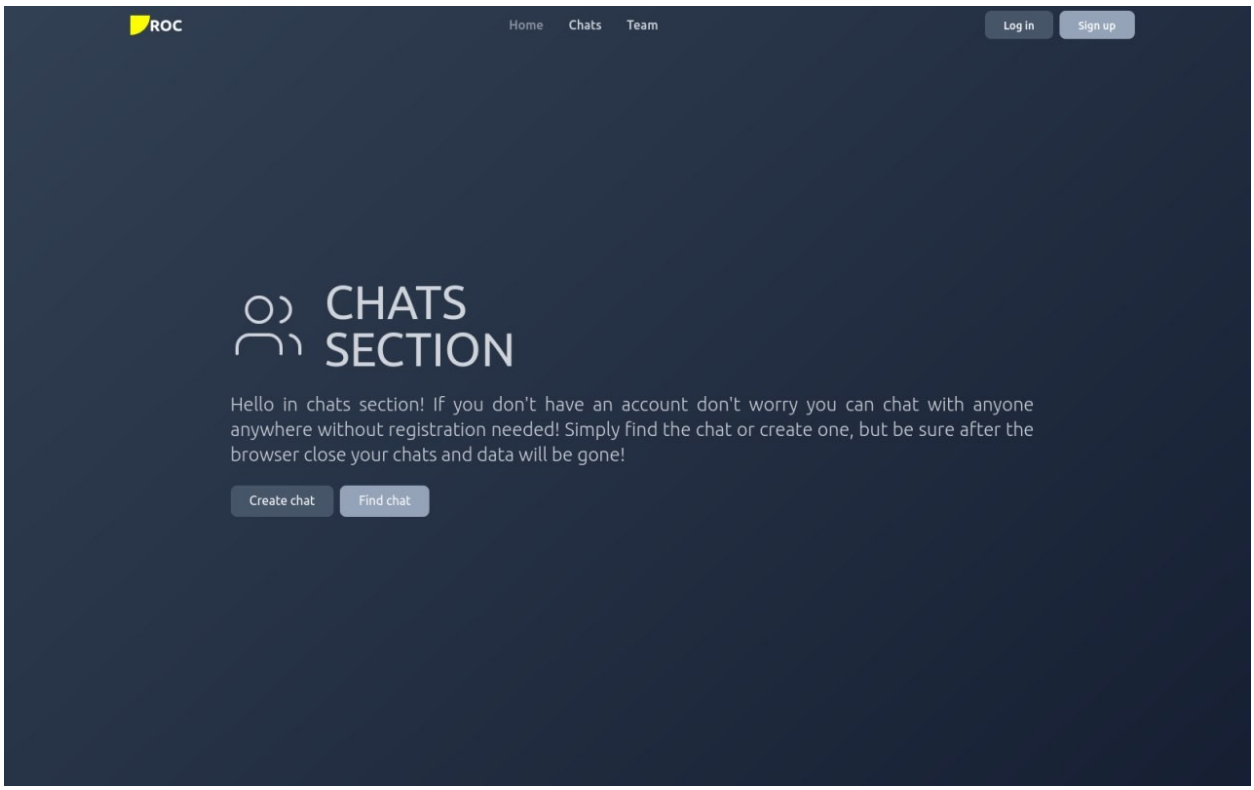

Рисунок 2.11 – Робоча область розділу чатів

На рисунку 2.5 продемонстровано основний екран взаємодії, де консолідовано перелік доступних діалогів, поточний статус з'єднання та базову панель навігації.

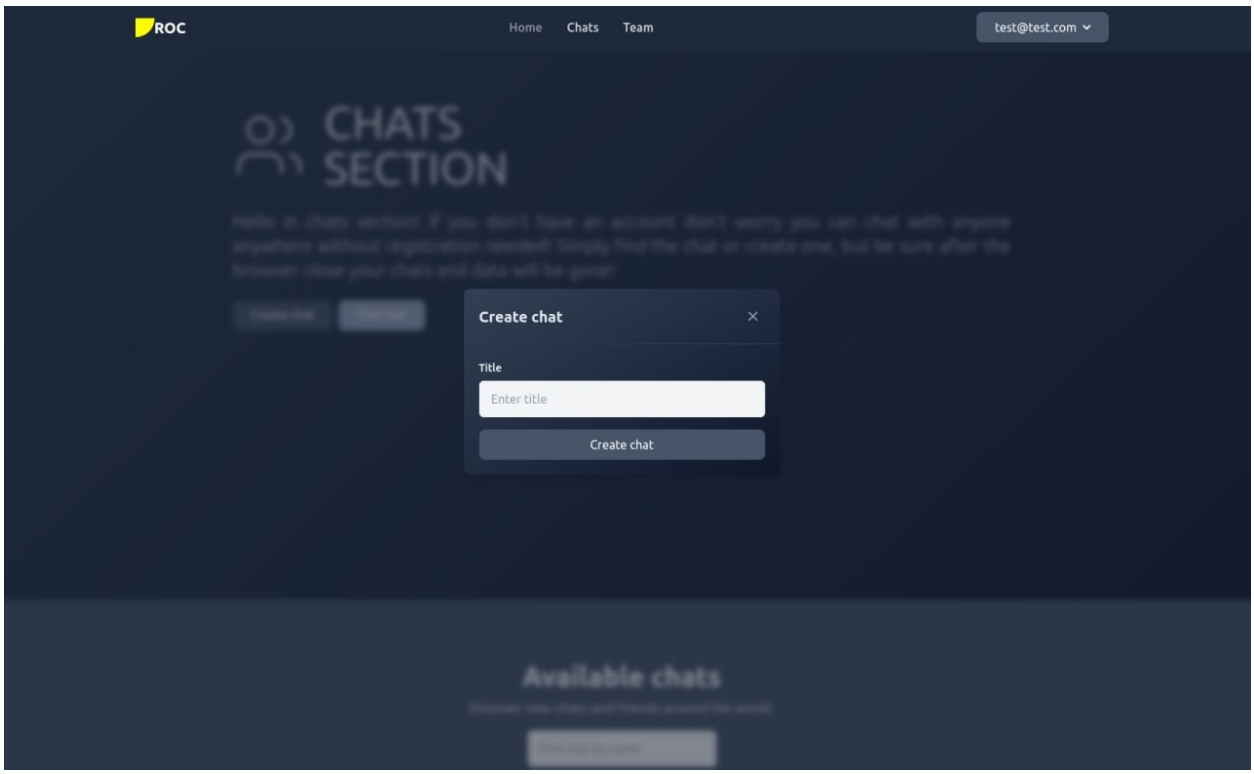

Рисунок 2.12 – Модальне вікно ініціалізації чату

При активації функції додавання нової кімнати система генерує спливаюче модальне вікно (рисунок 2.6), яке дозволяє задати необхідні параметри розмови, не розриваючи поточний контекст сторінки.

Рисунок 2.13 – Інтерфейс комунікації та обміну повідомленнями

До базових принципів функціонування клієнтської частини належать:

а) Сувора ізоляція логіки: візуальні React-компоненти (UI) повністю відокремлені від механізмів управління даними бази PostgreSQL, а взаємодія з сервером відбувається виключно через абстракції WebAPI та підключення SignalR;

з) Реактивне оновлення інтерфейсу: управління даними на стороні клієнта побудовано на базі React Context. Зміна інформації у глобальному стані автоматично ініціює процес рендерингу залежних DOM-елементів (наприклад, області повідомлень), що забезпечує миттєву візуальну синхронізацію;

и) Оптимізація ресурсів: підвищення загальної швидкодії системи шляхом застосування механізмів кешування та делегування обробки медіафайлів спеціалізованим хмарним сервісам.

3 РОЗРОБКА БАЗИ ДАНИХ І ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Аналіз обраного середовища розробки та технологічного стека

Проєктування архітектури системи обміну повідомленнями в реальному часі базується на сучасній розподіленій клієнт-серверній моделі, що історично прийшла на зміну традиційним монолітним рішенням задля підвищення загальної відмовостійкості, ізоляції модулів та забезпечення їхнього незалежного горизонтального масштабування.

Для реалізації високонавантаженої серверної частини (Backend) обґрунтовано обрано кросплатформну платформу C# ASP.NET Core. Вона забезпечує максимальну продуктивність та стабільність обробки HTTP-запитів через архітектурний стиль WebAPI завдяки асинхронній моделі програмування (async/await), оптимізованому конвеєру проміжного програмного забезпечення (Middleware) та вбудованому IoC-контейнеру впровадження залежностей (Dependency Injection). Це дозволяє ефективно керувати пулом серверних потоків, мінімізувати затримки під час взаємодії з базою даних, а також безперешкодно інтегрувати технологію SignalR для підтримки постійних WebSocket-з'єднань із клієнтом.

Клієнтський застосунок (Frontend) побудовано на базі бібліотеки React та інструмента збірки Vite. Це забезпечує миттєве гаряче перезавантаження модулів (HMR), глибоку оптимізацію статичних ресурсів методом tree-shaking, а також швидкий рендеринг інтерфейсу через Virtual DOM без перезавантаження сторінки чату.

Як основне середовище розробки (IDE) обрано кросплатформний редактор Visual Studio Code. Завдяки екосистемі плагінів він надає потужні інструменти для автоматизованого рефакторингу вихідного коду, інтелектуального підсвічування й автодоповнення (IntelliSense), наскрізного зневадження та контролю версій Git.

Таблиця 3.1 – Аналіз базових компонентів технологічного стека

Серверна платформа (Backend)	C# ASP.NET Core – забезпечує створення масштабованого WebAPI з підтримкою асинхронної обробки запитів та надійної системи авторизації (JWT).
Клієнтський інтерфейс (Frontend)	React (з TypeScript та Vite) – дозволяє будувати динамічні користувацькі інтерфейси за допомогою компонентного підходу, гарантуючи високу швидкість рендерингу.
Система управління базами даних	PostgreSQL + Entity Framework Core – об'єктно-реляційне відображення у поєднанні з надійною СУБД для гарантування цілісності транзакцій (ACID).
Комунікація в реальному часі	Microsoft SignalR – бібліотека для двосторонньої взаємодії між сервером та вебклієнтами через протокол WebSockets без потреби оновлення сторінки.
Кешування даних	Redis – in-memory сховище, що мінімізує навантаження на основну базу даних та радикально пришвидшує доступ до часто запитуваної інформації.

Впровадження зазначених технологій дозволяє створити масштабований веб-застосунок, який повністю відповідає актуальним стандартам безпеки та швидкодії. Важливим етапом проєктування стало дотримання архітектурного принципу розділення відповідальності (Separation of Concerns). Логіка обробки даних, взаємодії з БД та перевірки бізнес-правил жорстко інкапсульована на серверному боці, тоді як клієнтська частина відповідає виключно за візуалізацію даних та обробку подій користувача.

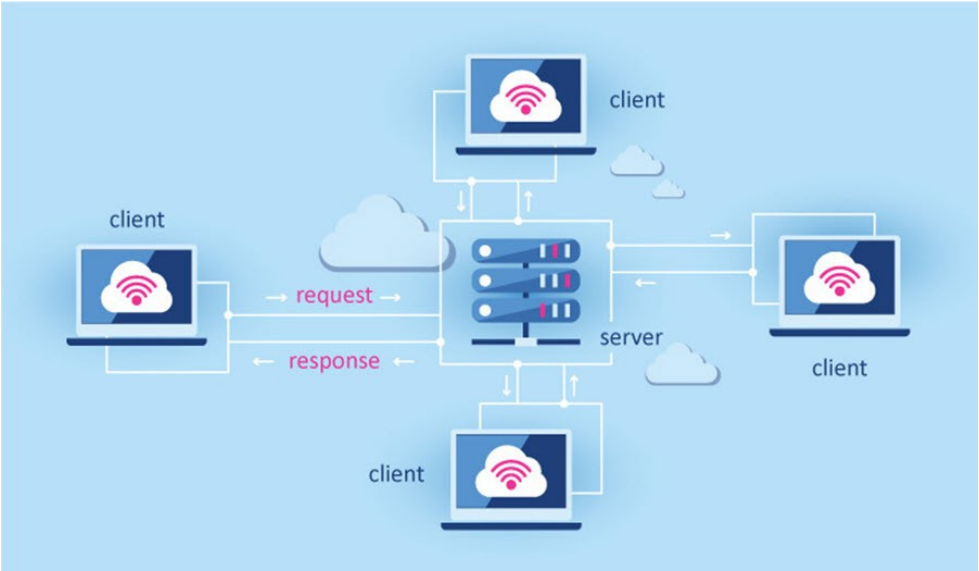

Рисунок 3.1 – Структура взаємодії компонентів клієнт-серверної архітектури

Особливу увагу під час розробки приділено кросплатформності візуальної складової. Інтеграція React із фреймворком TailwindCSS дозволила реалізувати гнучку систему стилізації, що автоматично адаптує розмітку під екрани мобільних пристроїв та десктопних моніторів. На противагу динамічній типізації стандартного JavaScript, застосування TypeScript на клієнті суттєво підвищило якість коду. Сувора статична типізація виявляє потенційні дефекти ще на етапі компіляції, роблячи архітектуру застосунку більш прогнозованою та знижуючи ризик виникнення помилок під час виконання.

Розробка власного API у зв'язці з PostgreSQL надала повний контроль над структурою даних. Миттєву доставку повідомлень реалізовано через SignalR, який встановлює постійне з'єднання, усуваючи потребу в polling-запитах.

Отже, обраний набір програмних інструментів формує надійний фундамент для стабільного функціонування застосунку. Комплексне використання ASP.NET Core, React, PostgreSQL та інструментів контейнеризації Docker не лише гарантує високу доступність системи, але й створює сприятливі умови для її подальшого масштабування без радикальної перебудови архітектури.

3.2 Проєктування структури бази даних

На відміну від нереляційних сховищ даних, інформаційна інфраструктура розробленого веб-застосунку побудована на базі об'єктно-реляційної моделі. Як основну систему управління базами даних (СУБД) задіяно PostgreSQL. Процес генерування та подальшої модифікації схеми реалізовано за допомогою підходу Code-First інструментарію Entity Framework Core. Це дає змогу автоматично трансформувати доменні моделі (C#-класи) у відповідні таблиці та забезпечує жорстку типізацію зв'язків.

Фізична архітектура бази даних є нормалізованою, що мінімізує надмірність інформації та гарантує її транзакційну цілісність через механізми зовнішніх ключів. Усю сукупність таблиць логічно структуровано за трьома ключовими доменами:

а) Підсистема ідентифікації та управління профілями (таблиці AspNetUsers, RefreshTokens та сутності ролей): призначена для криптографічно захищеного збереження облікових записів, токенів авторизації, а також персональних метаданих користувачів (статуси активності, посилання на аватар у Cloudinary, біографічні відомості);

к) Комунікаційні простори (таблиці Chats та ChatEntityUserEntity): забезпечує збереження інформації про створені кімнати обміну повідомленнями, фіксує ідентифікатори власників діалогів та реалізує зв'язок типу «багато до багатьох» для динамічного обліку учасників кожної бесіди;

л) Журнал взаємодії (таблиця Messages): виступає центральним сховищем усього згенерованого текстового контенту, де кожен об'єкт містить точну часову мітку та має строгу реляційну прив'язку як до автора-відправника, так і до конкретної гілки чату.

Рисунок 3.2 – Відображення талиці AspNetUsers (користувачів) в Postgresql

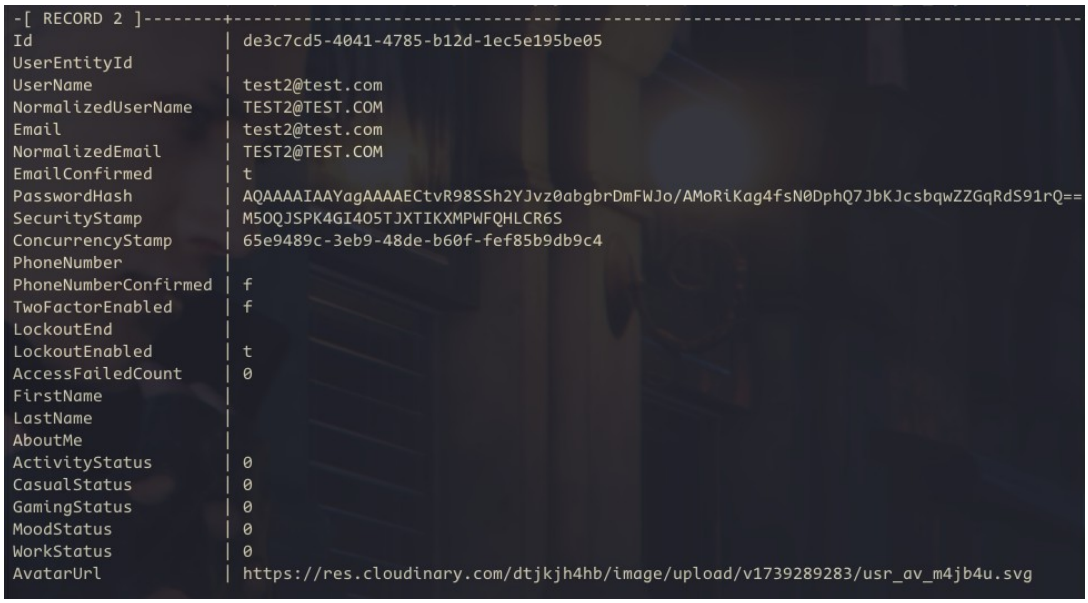

Рисунок 3.3 – Відображення талиці AspNetUsers (користувачів) в Postgresql

3.3 Зв'язки моделей бази даних PostgreSQL

3.3.1 Зв'язки доменної області (Бізнес-логіка застосунку)

а) Користувач – Створені чати (Власництво): зв'язок типу «один до багатьох» (1:N), де один користувач може бути творцем (власником) багатьох чатів, але кожен чат має лише одного

власника, що реалізується через PK AspNetUsers.Id → FK Chats.OwnerId;

м) Користувач – Участь у чатах (Членство): зв'язок типу «багато до багатьох» (M:N), де один користувач може бути учасником багатьох чатів, і водночас один чат може містити багато учасників-користувачів, який реалізується через проміжну таблицю ChatEntityUserEntity (PK AspNetUsers.Id → FK ChatEntityUserEntity.MembersId та PK Chats.Id → FK ChatEntityUserEntity.MemberChatsId);

н) Користувач – Повідомлення (Авторство): зв'язок типу «один до багатьох» (1:N), за якого один користувач може написати безліч повідомлень, але кожне конкретне повідомлення має лише одного автора, що реалізується через PK AspNetUsers.Id → FK Messages.UserId;

о) Чат – Повідомлення (Приналежність до кімнати): зв'язок типу «один до багатьох» (1:N), де в одному чаті може міститися безліч повідомлень, але кожне повідомлення належить суворо одному чату, що реалізується через PK Chats.Id → FK Messages.ChatId.

3.3.2 Зв'язки підсистеми ідентифікації (ASP.NET Core Identity)

а) Користувач – Ролі: зв'язок типу «багато до багатьох» (M:N), де користувач може мати кілька ролей, а одна роль може належати багатьом користувачам, який реалізується через проміжну таблицю AspNetUserRoles (PK AspNetUsers.Id → FK AspNetUserRoles.UserId та PK AspNetRoles.Id → FK AspNetUserRoles.RoleId);

п) Користувач – Токени оновлення (Refresh Tokens): зв'язок типу «один до багатьох» (1:N), де один користувач може мати кілька активних токенів оновлення (наприклад, для авторизації з різних пристроїв), що реалізується через PK AspNetUsers.Id → FK RefreshTokens.UserId;

р) Користувач – Додаткові дані (Claims, Logins, Tokens): зв'язок типу «один до багатьох» (1:N), де користувач може мати багато додаткових атрибутів (Claims), способів зовнішнього входу (Logins) та специфічних токенів, а реалізація відбувається завдяки тому, що PK AspNetUsers.Id є зовнішнім ключем для таблиць AspNetUserClaims, AspNetUserLogins та AspNetUserTokens;

с) Ролі – Додаткові права (Role Claims): зв'язок типу «один до багатьох» (1:N), за якого одна роль може містити багато додаткових правил або політик, що реалізується через PK AspNetRoles.Id → FK AspNetRoleClaims.RoleId.

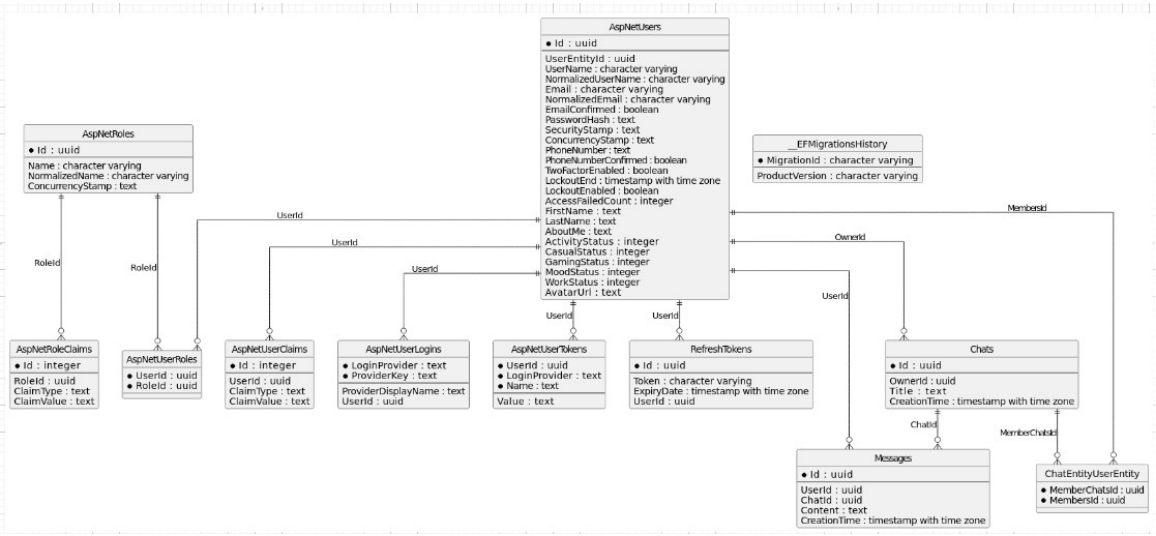

Рисунок 3.4 – Діаграма зв'язків моделей в Postgresql

3.4 Методика роботи користувача

Для мінімізації затримок у комунікації вебзастосунок використовує технологію SignalR, що забезпечує миттєвий обмін даними. Відтак учасникам бесіди не потрібно вручну оновлювати сторінку, оскільки синхронізація повідомлень відбувається автоматично у фоновому режимі.

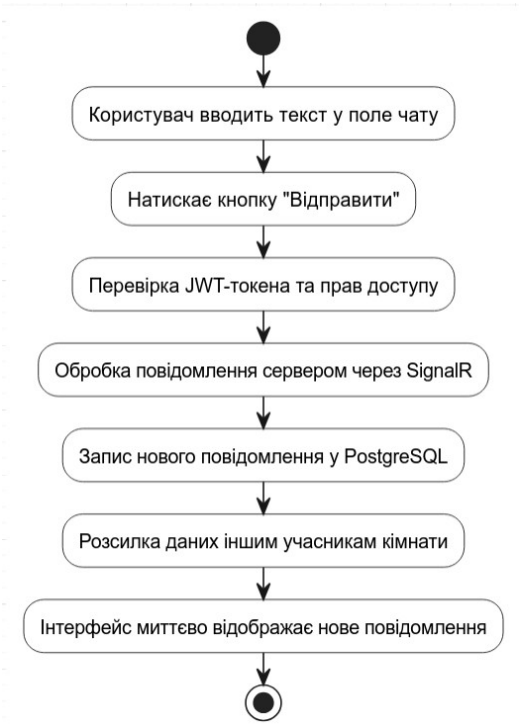

Рисунок 3.5 – Діаграма процесу обробки та надсилання повідомлення в реальному часі

Авторизувавшись у системі, особа спрямовується на центральну панель керування, яка містить перелік доступних чатів та інтегровану систему пошуку. Користувацький інтерфейс, побудований на базі бібліотеки React, спроєктовано з урахуванням принципів ергономіки, що суттєво пришвидшує адаптацію до функціонала платформи та робить навігацію інтуїтивною.

Подальша взаємодія із програмним продуктом базується на виконанні стандартних комунікативних сценаріїв. Архітектура клієнтської частини реалізована таким чином, щоб ключові дії (редагування повідомлень, приєднання до кімнат або управління профілем) здійснювалися без зайвих переходів між маршрутами. Це знижує когнітивне навантаження та підвищує загальну продуктивність роботи із застосунком.

Функціонал платформи дозволяє гнучко керувати власним профілем та адмініструвати створені чати. Усі дані, включно з історією листування та налаштуваннями доступу, надійно зберігаються у реляційній базі PostgreSQL, що, завдяки використанню Entity Framework Core, гарантує цілісність інформації та запобігає виникненню конфліктів у записах. Медіафайли своєю

чергою делегуються до хмарного сховища Cloudinary, що оптимізує навантаження на основну базу даних.

Управління комунікацією реалізовано через багаторівневу систему розмежування прав, де власник чату наділений повноваженнями видаляти повідомлення, вилучати учасників або передавати адміністративні привілеї. Будь-які зміни у статусі користувачів чи структурі бесіди миттєво транслюються всім підключеним клієнтам, підтримуючи абсолютну актуальність стану застосунку на всіх пристроях.

Безпека та ідентифікація сесій глибоко інтегровані у процес обміну інформацією. Під час кожної транзакції чи спроби авторизації система валідує JWT-токени та перевіряє користувача через Google reCAPTCHA. Такий підхід зводить до мінімуму ризики несанкціонованого втручання ботів і гарантує захищений канал зв'язку.

Отже, алгоритм взаємодії користувача з веб-застосунком зосереджений на забезпеченні безперервного та захищеного діалогу в реальному часі. Вибір сучасного стека технологій, застосування ефективних механізмів кешування даних (Redis) та оптимізована архітектура бази даних покликані гарантувати високу відмовостійкість системи та закласти міцний фундамент для її подальшого масштабування.

3.5 Інструкція користувача веб-застосунку

У цьому підрозділі наведено вичерпний посібник із практичної експлуатації платформи кінцевим споживачем. Матеріал охоплює алгоритми використання ключового інструментарію та принципи взаємодії клієнта з графічним інтерфейсом. Опис побудовано у вигляді послідовного аналізу базових сценаріїв: від етапу авторизації та налаштування профілю до безпосереднього обміну повідомленнями й адміністрування кімнат.

Архітектура клієнтської частини, розроблена на базі бібліотеки React, спроєктована з акцентом на максимальну ергономіку. Завдяки логічному групуванню елементів управління та мінімізації навігаційних кроків,

застосунок не вимагає від нових користувачів попереднього інструктажу. Графічна оболонка забезпечує швидкий доступ до активних діалогів і параметрів облікового запису, формуючи ясну візуальну ієрархію.

Окрім цього, розкрито механізми оптимізації комунікативного процесу. Система бере на себе автоматизацію таких завдань, як миттєве оновлення стрічки повідомлень (за допомогою SignalR), фільтрація бесід через вбудований пошук та управління ролями учасників чату. Наведений далі покроковий розбір ключових модулів формує комплексне розуміння того, як інтегровані технології забезпечують комфортну та безперебійну роботу користувача.

3.5.1 Ініціалізація платформи та стартова сторінка

Запуск клієнтської частини веб-застосунку розпочинається із завантаження базового компонента App.tsx, який відповідає за маршрутизацію та перевірку стану авторизації. Після успішної ідентифікації – класичної або через інтегрований сервіс Google Auth – користувач спрямовується на головну панель. Цей простір містить модуль пошуку та динамічний перелік доступних чатів.

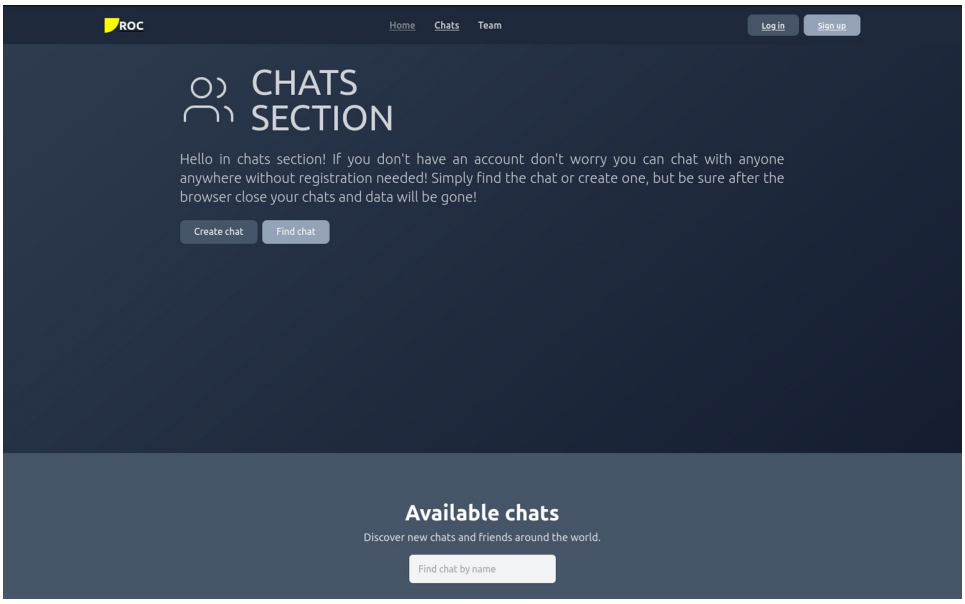

Рисунок 3.6 – Інтерфейс головного екрана з переліком активних чатів

Базова структура ініціалізації головного екрана та системи маршрутизації у середовищі React має такий вигляд:

```
function App() {
  return (
    <AuthProvider>
      <ToastContainer />
      <Outlet />
    </AuthProvider>
  );
}
```

3.5.2 Робота з учасниками групи чату

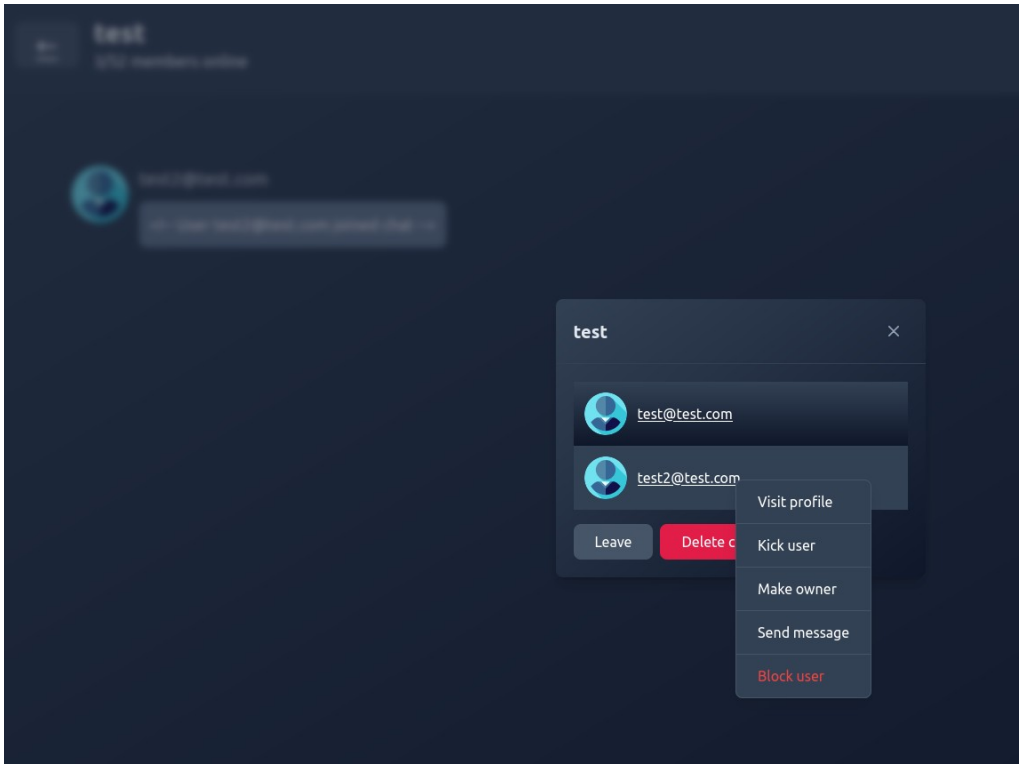

Рисунок 3.7 – Відображення учасників групи

Представлений програмний модуль відповідає за клієнтську логіку адміністрування групових бесід та управління складом їхніх учасників. Архітектурно рішення реалізовано у вигляді композиції React-компонентів, які взаємодіють із глобальним станом застосунку (через Context API) для отримання актуальних ідентифікаційних даних та інформації про кімнату.

Інтерфейс управління інкапсульовано в модальному вікні, де реалізовано строгу диференціацію прав доступу на рівні відображення елементів (UI). Система автоматично ідентифікує статус поточного користувача: власнику чату надаються розширені привілеї, що включають можливість повної ліквідації бесіди та виклик спеціалізованого контекстного меню для управління іншими учасниками. Натомість рядовий клієнт наділений базовим рівнем доступу, який обмежується опцією виходу з групи. Усі тригери подій, включно з позиціюванням контекстного меню та ініціацією HTTP-запитів до серверної частини застосунку, обробляються асинхронно, що гарантує плавність роботи графічної оболонки та відсутність блокувань основного потоку виконання

Відображення учасників групи:

```
type ChatUsersMenuProps = {
  isModalOpen: boolean;
  setIsModalOpen: React.Dispatch<React.SetStateAction<boolean>>;
}

const ChatUsersMenu = ({ isModalOpen, setIsModalOpen }:
ChatUsersMenuProps) => {
  const { user } = useAuth();

  const { chatInfo } = useChatInfo();

  const navigate = useNavigate();

  const handleChatLeave = () => {
    if (chatInfo.id) {
      ChatUsersService.deleteMemberMe(chatInfo.id)
        .then(() => navigate("/chats"))
        .catch((e) => console.error("Error leaving chat:",
e.message));
    }
  };

  const handleChatDelete = () => {
    if (chatInfo.id) {
      ChatService.deleteChat(chatInfo.id)
        .then(() => navigate('/chats'))
        .catch((e) => console.error("Error deleting chat:",
e.message));
    }
  };
```

```
  return <Modal
    isHeaderEllipsis={true}
    className="flex flex-col gap-4 p-6 shadow-lg"
    title={chatInfo?.title}
    isModalOpen={isModalOpen}
    setIsModalOpen={setIsModalOpen}
  >
    <UsersListMenu />
    <div className="flex gap-2">
      <Button variant="primary" onClick={handleChatLeave}>
        Leave
      </Button>
      {user?.id === chatInfo?.ownerId && (
        <Button variant="danger" onClick={handleChatDelete}>
          Delete chat
        </Button>
      )}
    </div>
  </Modal>
}

export default memo(ChatUsersMenu);

const UsersListMenu = memo(function UsersListMenu() {
  const { user } = useAuth();

  const { chatInfo } = useChatInfo();
  const { chatUsers } = useChatUsers();

  const [menuPosition, setMenuPosition] = useState<{ x: number;
y: number }>({ x: 0, y: 0 });
  const [selectedUserId, setSelectedUserId] = useState<string |
null>(null);

  const handleUserContextMenu = (
    e: React.MouseEvent<HTMLLIElement, MouseEvent>,
    userId: string
  ) => {
    e.preventDefault();

    if (menuPosition.x !== 0 && menuPosition.y !== 0) {
      setSelectedUserId(null);
      setMenuPosition({ x: 0, y: 0 });
      return;
    }

    setMenuPosition({ x: e.clientX, y: e.clientY });
    setSelectedUserId(userId);
  };
```

```
  return <ul className="text-white">
    {chatUsers?.map((userChat, index) =>
      <li onContextMenu={(e) => handleUserContextMenu(e,
userChat.id)} key={index}>
        <Link
          to={`/profile/${userChat.id}`}
          className={`flex items-center gap-3 p-3 transition-
colors h-full w-full duration-300 ${chatInfo?.ownerId ===
userChat.id
            ? "bg-gradient-to-b from-slate-700 hover:from-
slate-600 to-slate-900"
            : "bg-slate-700 hover:bg-slate-800"
            }`}
        >
          <UserAvatar avatarUrl={userChat.avatarUrl}
width="48px" height="48px" />
          <div className="overflow-x-auto text-wrap break-
words">{userChat.email}</div>
        </Link>
        {userChat.id === user?.id || user?.id !==
chatInfo.ownerId ?
          <UserContextMenu
            isVisible={selectedUserId === userChat.id}
            userId={userChat.id}
            position={menuPosition}
          />
          :
          <OwnerContextMenu
            isVisible={selectedUserId === userChat.id}
            position={menuPosition}
            userId={userChat.id}
          />
        }
      </li>
    )}
  </ul>
});
```

3.5.3 Додавання чату

Програмний модуль ініціалізації нових чатів реалізовано у вигляді інкапсульованого React-компонента, інтегрованого у систему модальних вікон застосунку. Захоплення та попередня обробка користувацького вводу базується на патерні «контрольованого компонента» (controlled component), що забезпечує двосторонню синхронізацію між локальним станом форми та графічним інтерфейсом у режимі реального часу.

Архітектурно рішення суворо дотримується принципу єдиної відповідальності: компонент виступає виключно як рівень представлення (Presentation Layer). Він відповідає за валідацію вводу на рівні інтерфейсу, блокування стандартної поведінки браузера під час відправки форми та агрегацію введених метаданих (назви чату). Подальша обробка та запис даних делегуються батьківському вузлу через механізм функцій зворотного виклику (callback). Такий підхід гарантує слабку зв'язність (low coupling) компонентів, відокремлюючи візуальну логіку відображення модального вікна від бізнес-логіки створення сутностей у базі даних.

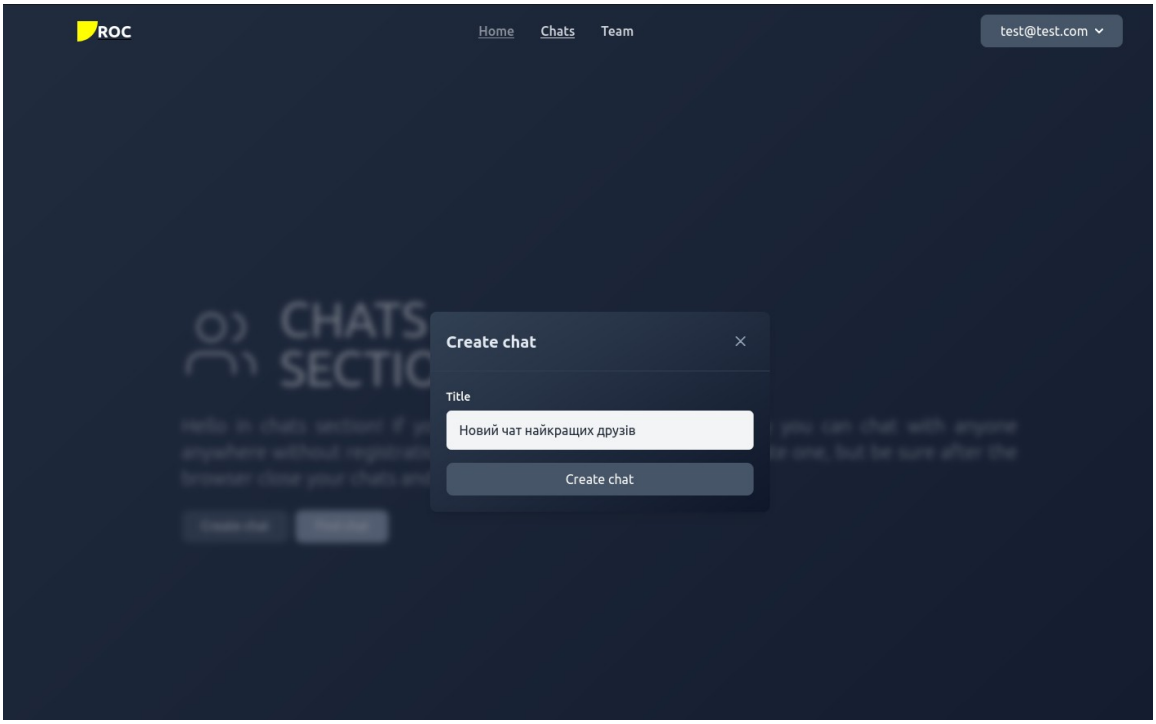

Рисунок 3.8 – Створення чату

Створення чату:

```
import Button from "@src/components/ui/Button";
import Input from "@src/components/ui/Input";
import Modal from "@src/components/ui/Modal";
import React, { useState } from "react";

export interface CreateChatFormData {
  title: string;
}
```

```
type Props = {
  onSubmit: (e: React.FormEvent<HTMLFormElement>, data:
CreateChatFormData) => void;
  isChatFormOpen: boolean;
  setIsChatFormOpen:
React.Dispatch<React.SetStateAction<boolean>>;
};

const CreateChatForm = ({ onSubmit, isChatFormOpen,
setIsChatFormOpen }: Props) => {
  const [formData, setFormData] = useState<CreateChatFormData>({
    title: "",
  });

  const handleChange = (e: React.ChangeEvent<HTMLInputElement>)
=> {
    const { name, value } = e.target;

    setFormData((prevData) => ({
      ...prevData,
      [name]: value,
    }));
  };

  const handleCreateChatFormSubmit = (e:
React.FormEvent<HTMLFormElement>) => {
    e.preventDefault();
    onSubmit(e, formData);
  };

  return (
    <Modal isModalOpen={isChatFormOpen}
setIsModalOpen={setIsChatFormOpen} title={"Create chat"}>
      <form className="space-y-4"
onSubmit={handleCreateChatFormSubmit}>
        <div>
          <label className="block mb-2 text-sm font-medium text-
white">Title</label>
          <Input
            onChange={handleChange}
            value={formData.title}
            type="text"
            name="title"
            id="title"
            className="w-full"
            placeholder="Enter title"
          />
        </div>
        <Button type="submit" className="w-full">
          Create chat
        </Button>
```

```
      </form>
    </Modal>
  );
};

export default CreateChatForm;
```

3.5.4 Профіль і редагування профілю

Модуль редагування профілю реалізовано на базі React-компонентів, що мінімізують надлишкові мережеві запити завдяки алгоритмам мемоїзації стану. Інтерфейс сегментовано на блоки конфігурації багатовекторних статусів, біографічних даних та реєстру контактів. Усі модифікації асинхронно передаються до PostgreSQL, після чого клієнтська частина миттєво ревалідує оновлений контекст застосунку.

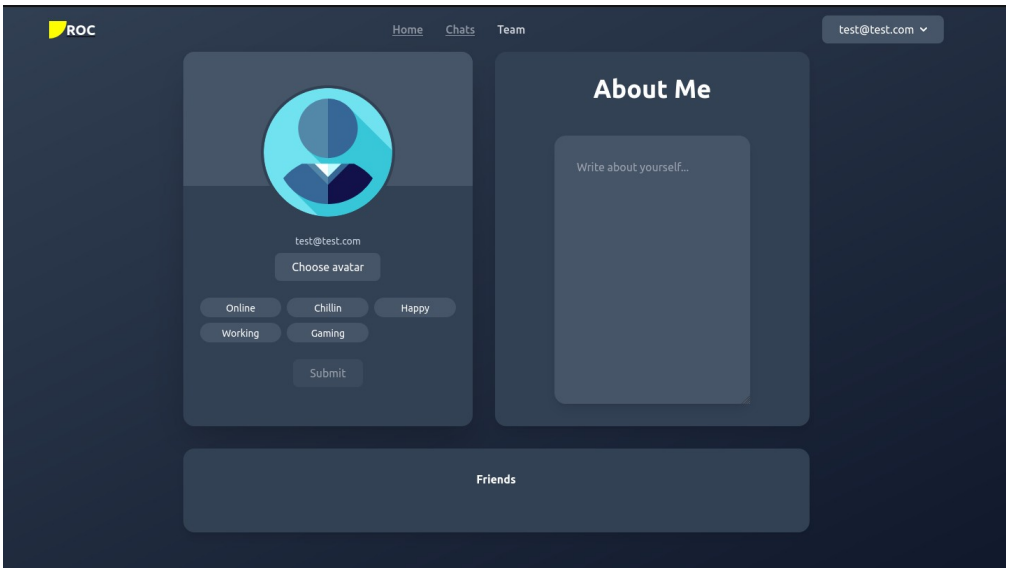

Рисунок 3.9 – До редагування профілю

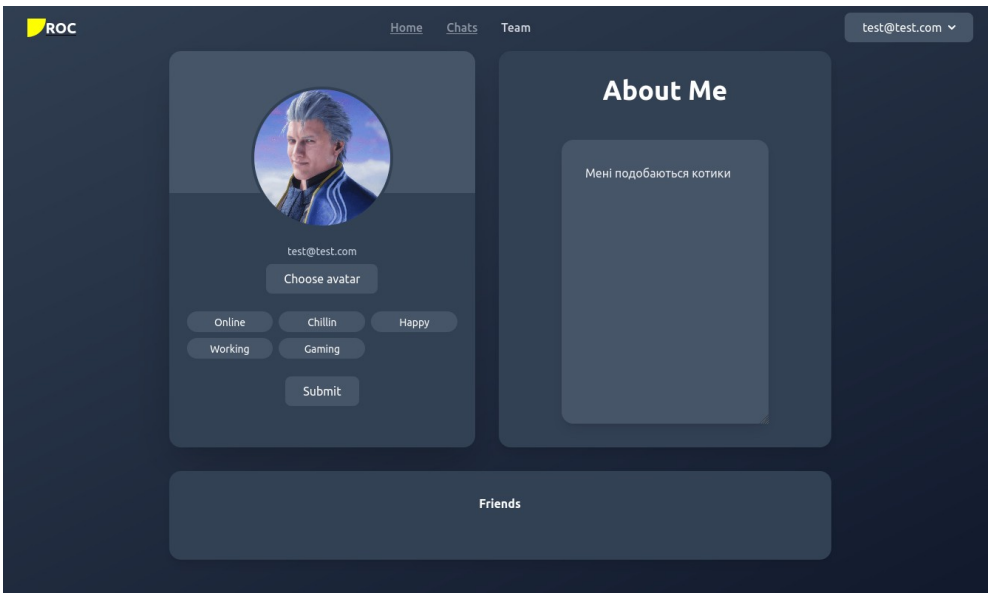

Рисунок 3.10 – Після редагування профілю

EditProfile.tsx:

```
import { useMemo, useState } from "react";
import { useNavigate, useParams } from "react-router-dom";
import FriendPreview from "./FriendPreview";
import { UserService } from "@src/services/api/UserService";
import { EditUserProfileType } from "./profile.types";
import EditProfileUserCard from "./EditProfileUserCard";
import { useUserProfile } from "./UserProfileContext";
import ErrorScreen from "@src/components/ui/ErrorScreen";

const EditUserProfile = () => {
  const { refreshUser, friends, ...userProfile } =
useUserProfile();
  const { userId } = useParams<{ userId: string }>();
  const navigate = useNavigate();

  const [editUserProfile, setEditUserProfile] =
useState<EditUserProfileType>(userProfile);

  const isChanged = useMemo(() => {
    return JSON.stringify(userProfile) !==
JSON.stringify(editUserProfile);
  }, [userProfile, editUserProfile]);

  if (!userId) return <ErrorScreen />;

  const handleEditFormSubmit = async (e:
React.FormEvent<HTMLFormElement>) => {
    e.preventDefault();

    try {
```

```
      await UserService.updateUserProfile(userId,
editUserProfile);
      await refreshUser();
      console.log("Updated profile successfully");
    } catch (e: any) {
      console.error("Error updating profile:", e.message);
    }

    navigate(-1);
  };

  const handleInputChange = (field: keyof EditUserProfileType,
value: any) => {
    setEditUserProfile((prev) => ({ ...prev, [field]: value }));
  };

  return (
    <>
      <section>
        <form
          onSubmit={handleEditFormSubmit}
          className="relative flex flex-col gap-8 m-16 max-w-4xl
mx-auto"
        >
          <div className="flex gap-8 lg:flex-row flex-col">
            <EditProfileUserCard
              isSubmitButtonEnabled={isChanged}
              onAvatarChange={(value) =>
handleInputChange("avatar", value)}
              className="w-full flex-grow"
              onActivityStatusChange={(value) =>
handleInputChange("activityStatus", value)}
              onCasualStatusChange={(value) =>
handleInputChange("casualStatus", value)}
              onMoodStatusChange={(value) =>
handleInputChange("moodStatus", value)}
              onWorkStatusChange={(value) =>
handleInputChange("workStatus", value)}
              onGamingStatusChange={(value) =>
handleInputChange("gamingStatus", value)}
            />

            <div className="flex w-full flex-grow flex-col gap-6
p-8 bg-slate-700 rounded-2xl shadow-lg">
              <section className="relative flex flex-col gap-12
m-auto">
                <h2 className="text-4xl font-bold text-white
text-center">About Me</h2>
                <div className="flex gap-8 lg:flex-row flex-
col">
                  <div className="w-full flex flex-col gap-6 ">
```

```
                    <textarea
                      value={editUserProfile.aboutMe ?? ""}
                      name="aboutMe"
                      onChange={(e) =>
handleInputChange("aboutMe", e.target.value)}
                      placeholder="Write about yourself..."
                      className="rounded-2xl shadow-lg
focus:outline-none min-h-96 p-8 text-white bg-slate-600 leading-
relaxed text-opacity-90"
                    />
                  </div>
                </div>
              </section>
            </div>
          </div>

          <div className="bg-slate-700 rounded-2xl shadow-lg
p-8">
            <p className="font-semibold text-2~xl text-white
text-center mb-8">Friends</p>
            <ul className="flex flex-col gap-6">
              {friends?.map((value, index) => (
                <li key={index}>
                  <FriendPreview
                    avatarUrl={value.avatarUrl}
                    firstName={value.firstName}
                    lastName={value.lastName}
                    email={value.email}
                    id={value.id}
                  />
                </li>
              ))}
            </ul>
          </div>
        </form>
      </section>
    </>
  );
};

export default EditUserProfile;
```

ВИСНОВОК

У результаті виконання кваліфікаційної роботи було спроєктовано та успішно розгорнуто багатокористувацький веб-застосунок для обміну повідомленнями в режимі реального часу. Актуальність розробленого рішення зумовлена стабільним попитом на високопродуктивні, безпечні та масштабовані комунікаційні платформи, здатні забезпечити миттєву синхронізацію даних між десятками підключених клієнтів.

Програмний продукт побудовано за принципами сучасної клієнт-серверної архітектури. Серверну частину реалізовано на базі ASP.NET Core WebAPI, а клієнтську – з використанням екосистеми React. Ключовим технологічним рушієм проєкту стала інтеграція протоколу SignalR, який забезпечив безперервний дуплексний зв'язок. Управління персистентними даними покладено на реляційну СУБД PostgreSQL та Entity Framework Core. Водночас для оптимізації мережевого навантаження застосовано принципи розподіленої обробки даних: in-memory кешування через Redis та винесення медіафайлів у хмарне середовище Cloudinary.

Система пропонує комплексний інструментарій для взаємодії. Реалізовано надійні механізми автентифікації (JWT та Google Auth), систему захисту від несанкціонованого бот-трафіку (Google reCAPTCHA), а також розгалужену ієрархію прав доступу, що дозволяє власникам чатів ефективно модерувати учасників та контент. Графічна оболонка спроєктована з огляду на стандарти UI/UX-дизайну, формуючи інтуїтивне та ергономічне середовище.

Комплексне тестування підтвердило високу відмовостійкість архітектури. Застосунок демонструє стабільність під час виконання всіх запланованих алгоритмів – від безпечної авторизації до безперебійної трансляції повідомлень, – що свідчить про цілковите досягнення поставлених цілей та готовність платформи до практичної експлуатації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Microsoft Learn. Офіційна документація по розробці програмних архітектур на платформі ASP.NET Core [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/

2. PostgreSQL. Офіційна документація з механізмів реляційного зберігання та інтеграції кешування [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/

3. The WebSocket Protocol. Специфікація IETF RFC 6455 для передачі даних у режимі реального часу [Електронний ресурс]. – Режим доступу: https://datatracker.ietf.org/doc/html/rfc6455

4. Radware. Bot Security: Risks, Defensive Technologies & Best Practices [Електронний ресурс]. – Режим доступу: https://www.radware.com/cyberpedia/bot-management/bot-security/

5. Bettermode. Should You Use Slack for Community? Pros and Cons (аналіз архітектурної перевантаженості та недоліків платформи) [Електронний ресурс]. – Режим доступу: https://bettermode.com/hub/articles/post/should-you-use-slack-for-community-pros-and-cons-rSr1zOghLfesbNE

6. Eleken. 12 Bad UX Examples: Learn from Criticized Apps (дослідження перевантажених інтерфейсів у месенджерах) [Електронний ресурс]. – Режим доступу: https://www.eleken.co/blog-posts/bad-ux-examples

7. JSON Web Token (JWT). Специфікація IETF RFC 7519 для безпечної обробки сесій та авторизації [Електронний ресурс]. – Режим доступу: https://datatracker.ietf.org/doc/html/rfc7519

8. Microsoft Learn. Огляд бібліотеки ASP.NET Core SignalR для безперервного з'єднання та синхронізації [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/signalr/

9. Vite. Документація інструменту для збирання проекту React [Електронний ресурс]. – Режим доступу: https://vitejs.dev/guide/

ДОДАТОК А – Лістинг коду

Модуль useAuth.tsx реалізує архітектурний патерн React Context для управління станом сесії. Він інкапсулює логіку базової та OAuth-авторизації (Google), а також конфігурує HTTP-інтерсептори Axios.

```
import { createContext, useEffect, useLayoutEffect, useMemo,
useState } from "react";
import { useNavigate } from "react-router-dom";
import api from "@services/axios/instance";
import { LoginRequest, SignupRequest } from
"@src/models/dtos/Auth";
import { AuthRoutes } from "@src/services/api/ApiRoutes";
import { AuthService } from "@src/services/api/AuthService";
import { GoogleService } from
"@src/services/google/GoogleService";
import { AxiosRequestConfig } from "axios";
import { handleError, useCustomHook } from "@src/utils/helpers";

export type UserProfile = {
  id: string;
  firstName: string | null;
  lastName: string | null;
  email: string;
};

type AuthContextType = {
  token: string | null | undefined;
  user: UserProfile | null | undefined;
  loginUser: (request: LoginRequest, config?:
AxiosRequestConfig<any>) => Promise<void>;
  signupUser: (request: SignupRequest, config?:
AxiosRequestConfig<any>) => Promise<string | undefined>;
  loginGoogle: (credential: string) => Promise<void>;
  signupGoogle: (credential: string) => Promise<void>;
  logoutUser: () => Promise<void>;
  isUserLoggedIn: () => boolean;
}

type AuthProviderProps = { children: React.ReactNode };

const AuthContext = createContext<AuthContextType>({} as
AuthContextType);

export const AuthProvider = ({ children }: AuthProviderProps) =>
{
  const navigate = useNavigate();
  const [token, setToken] = useState<string | null>(null);
```

```
  const [user, setUser] = useState<UserProfile | null |
undefined>(undefined);
  const [isReady, setIsReady] = useState(false);

  // Fetch user profile and set token
  useEffect(() => {
    const abortController = new AbortController();

    AuthService.me({
      signal: abortController.signal,
    })
      .then((data) => {
        if (data) {
          setToken(data.token);
          setUser(data.user);
        }
      })
      .catch((e) => {
        console.log("[Me] Error:", e.message);
        setToken(null);
        setUser(null);
      });

    return () => abortController.abort();
  }, []);

  // Set up request interceptor
  useLayoutEffect(() => {
    const authInterceptor =
api.interceptors.request.use((config: any) => {
      config.headers.Authorization =
        !config._retry && token ? `Bearer ${token}` :
config.headers.Authorization;
      return config;
    });

    return () =>
api.interceptors.request.eject(authInterceptor);
  }, [token]);

  // Set up response interceptor
  useLayoutEffect(() => {
    const refreshInterceptor = api.interceptors.response.use(
      (response) => response,
      async (error) => {
        const originalRequest = error.config;

        if (
          error.response &&
          error.response.status === 401 &&
          originalRequest.url !== AuthRoutes.refreshToken
```

```
        ) {
          console.log("[Interceptor][Response] Error 401.");
          try {
            const response = await
api.post(AuthRoutes.refreshToken, {}, { withCredentials:
true });

            setToken(response.data.token);

            originalRequest.headers.Authorization = `Bearer $
{response.data.token}`;
            originalRequest._retry = true;

            console.log("[Interceptor][Response]: Token
refreshed");

            return api(originalRequest);
          } catch (e: any) {
            if (e.name !== "CanceledError") {
              console.log("[Interceptor][Response]: Token not
refreshed");

              setToken(null);
              setUser(null);
            }
          }
        }

        return Promise.reject(error);
      }
    );

    return () =>
api.interceptors.response.eject(refreshInterceptor);
  }, []);

  // Mark the app as ready only if user cannot be fetched from
server (missing refreshToken)
  // OR if user is logged in (refreshToken presented and user
info retrieved with JWT token)
  useEffect(() => {
    if (user === null || isUserLoggedIn()) {
      setIsReady(true);
    }
  }, [user]);

  const signupUser = async (request: SignupRequest, config?:
AxiosRequestConfig<any>) => {
    try {
      const data = await AuthService.signup(request, config);
      return data;
```

```
    } catch (e: any) {
      handleError(e);
      console.error("Unable to signup:", e.message);
    }
  };

  const loginUser = async (request: LoginRequest, config?:
AxiosRequestConfig<any>) => {
    try {
      const data = await AuthService.login(request, config);
      if (data) {
        setToken(data.token);
        setUser(data.user);
      }
    } catch (e: any) {
      handleError(e);
      console.error("Unable to login:", e.message);
    }
  };

  const signupGoogle = async (credential: string) => {
    try {
      const data = await GoogleService.signup(credential);
      if (data) {
        setToken(data.token);
        setUser(data.user);
      }
    } catch (e: any) {
      console.error("Unable to signup with google:", e.message);
      handleError(e);
    }
  };

  const loginGoogle = async (credential: string) => {
    try {
      const data = await GoogleService.login(credential);
      if (data) {
        setToken(data.token);
        setUser(data.user);
      }
    } catch (e: any) {
      console.error("Unable to login with google:", e.message);
      handleError(e);
    }
  };

  const isUserLoggedIn = () => !!user;

  const logoutUser = async () => {
    try {
      await AuthService.logout();
```

```
      setUser(null);
      setToken(null);
      navigate("/");
    } catch (e: any) {
      console.error("Unable to logout:", e.message);
    }
  };

  const value = useMemo(
    () => ({
      token,
      user,
      loginUser,
      signupUser,
      loginGoogle,
      signupGoogle,
      logoutUser,
      isUserLoggedIn,
    }),
    [token, user]
  );

  // Render children only when the app is ready
  return <AuthContext.Provider value={value}>{isReady ? children
: null}</AuthContext.Provider>;
};

export const useAuth = () => useCustomHook(AuthContext,
useAuth.name);
```

Код повідомлення Message.tsx:

```
import MessageContent from "./MessageContent";
import { ChatMessage } from "./{chatId}.types";
import MessageActions from "./MessageActions";
import UserAvatarLink from "./UserAvatarLink";
import { memo } from "react";

type MessageProps = {
  message: ChatMessage;
  isCurrentUser: boolean;
  isCurrentUserPrevious: boolean;
  showUserInfo: boolean;
};

const Message = ({ message, isCurrentUserPrevious,
isCurrentUser, showUserInfo }: MessageProps) => {
  console.count("Message render");
  const avatarSize = "64px";

  return (
```

```
    <li className={`flex gap-3 ${isCurrentUserPrevious ?
"pt-2" : "pt-8"} ${isCurrentUser ? "justify-end" : "justify-
start"}`}
    >
      {showUserInfo ? (
        <>
          <UserAvatarLink
            avatarUrl={message.user.avatarUrl}
            width={avatarSize}
            height={avatarSize}
            className={`rounded-full object-cover $
{isCurrentUser ? "order-2" : ""}`}
            userId={message.user.id}
          />

          <div className="flex flex-col gap-3 max-w-[80%]">
            <p
              className={`text-xl text-white text-opacity-80
flex gap-4 ${isCurrentUser ? "font-medium self-end" : ""
                }`}
            >
              {message.user.email}
            </p>
            <div
              className={`flex gap-2 items-center $
{isCurrentUser ? "justify-end" : "justify-start"
                }`}
            >
              {isCurrentUser && <MessageActions
messageId={message.id} />}
              <MessageContent messageChat={message} />
            </div>
          </div>
        </>
      ) : (
        <>
          <div
            style={{ width: avatarSize }}
            className={`rounded-full object-cover $
{isCurrentUser ? "order-2" : ""}`}
          />

          <div className="flex items-center gap-2 max-w-[80%]">
            {isCurrentUser && <MessageActions
messageId={message.id} />}
            <MessageContent messageChat={message} />
          </div>
        </>
      )}
    </li>
  );
```

```
};

export default memo(Message);
```

Код чату Chat.tsx:

```
import ChatHeader from "./ChatHeader";
import { MessageContextProvider } from "./MessageContext";
import ChatMessages from "./ChatMessages";
import { CountOfNewMessagesWrapper } from
"./CountOfNewMessages";
import MessageInputBlock from "./MessageInputBlock";
import { ChatUsersContextProvider } from "./ChatUsersContext";
import { ChatInfoContextProvider } from "./ChatInfoContext";
import { ChatMessagesContextProvider } from
"./ChatMessagesContext";

const Chat = () => {
  console.count("Chat render");
  return (
    <div
      className="flex flex-col bg-gradient-to-br from-slate-700
to-slate-900 lg:px-16 flex-grow"
      style={{ minHeight: "720px" }}
    >
      <ChatUsersContextProvider>
        <ChatInfoContextProvider>
          <ChatHeader />

          <ChatMessagesContextProvider>
            <CountOfNewMessagesWrapper />

            <MessageContextProvider>
              <ChatMessages />
              <MessageInputBlock />
            </MessageContextProvider>
          </ChatMessagesContextProvider>
        </ChatInfoContextProvider>
      </ChatUsersContextProvider>
    </div>
  );
};

export default Chat;
```

Код доступних чатів AvailableChats.tsx:

```
import { useEffect, useState } from "react";
import { ChatService } from "@src/services/api/ChatService";
import ChatPreview from "@src/pages/chats/ChatPreview";
import { LoaderScreen } from "@src/components/ui/Loader";
```

```
import Input from "@src/components/ui/Input";

export default function ChatsSection() {
  const [chats, setChats] = useState<{ id: string; title: string
}[]>([]);
  const [loading, setLoading] = useState(true);

  const [filter, setFilter] = useState("");

  function onFilterChange(value: string) {
    setFilter(value);
  }

  useEffect(() => {
    const abortController = new AbortController();
    ChatService.getChats({ pageNubmer: 1, pageSize: 9, filter:
filter }, { signal: abortController.signal })
      .then((data) => {
        setChats(data?.items);
        setLoading(false);
      })
      .catch((e) => console.error("Error getting chats:",
e.message));

    return () => abortController.abort();
  }, [filter]);

  if (loading) return <LoaderScreen />;

  return (
    <section className="py-20 px-6 bg-slate-600 text-white">
      <div className="max-w-screen-xl mx-auto flex flex-col
lg:gap-8 gap-4">
        <div className="text-center mb-12">
          <h2 className="text-4xl font-bold text-
white">Available chats</h2>
          <p className="text-lg opacity-80 mt-2">Discover new
chats and friends around the world.</p>
          <Input onChange={e => onFilterChange(e.target.value)}
className="mt-4" placeholder="Find chat by name" />
        </div>
        <div className="grid lg:grid-cols-3 md:grid-cols-2 grid-
cols-1 lg:gap-x-20 gap-x-10 gap-y-5">
          {chats?.map((value, index) => (
            <ChatPreview key={index} id={value.id}
title={value.title} />
          ))}
        </div>
      </div>
    </section>
  );
```

```
}
```

Код сторінки авторизації LoginPage.tsx:

```
import Logo from "@src/components/ui/Logo";
import { Link } from "react-router";
import LoginForm from "./LoginForm";

export default function LoginPage() {
  return (
    <div className="min-h-screen flex items-center justify-
center p-4 sm:p-8">
      <div className="grid md:grid-cols-2 items-center gap-16
max-w-6xl w-full">
        <div className="text-center md:text-left">
          <Logo className="flex md:justify-start justify-center
mt-10 lg:mt-0" scale={2} />
          <h2 className="text-3xl lg:text-4xl font-extrabold
text-white mt-8 bg-clip-text bg-gradient-to-r from-blue-400 to-
purple-500">
            Seamless Login for Exclusive Access
          </h2>
          <p className="text-sm text-gray-300 mt-6">
            Immerse yourself in a hassle-free login journey with
our intuitively designed login
            form. Effortlessly access your account.
          </p>
          <p className="text-sm text-gray-300 mt-6">
            Don't have an account? 
            <Link
              to="/signup"
              className="text-blue-400 font-semibold hover:text-
blue-300 transition-colors duration-200"
            >
              Register here
            </Link>
          </p>
        </div>

        <LoginForm className="max-w-md lg:mx-16 mt-8 mx-auto" />
      </div>
    </div>
  );
}
```

Код сторінки реєстрації SignupPage.tsx:

```
import { Facebook, Apple } from
"@src/components/svg/SVGAuthProviders";
import GoogleSignup from "@src/services/google/GoogleSignup";
```

```
import SignupForm from "./SignupForm";

export default function SignupPage() {
  return (
    <div className="min-h-screen flex items-center justify-
center lg:px-16 lg:py-32 pt-16">
      <div className="grid md:grid-cols-2 items-center gap-y-8
bg-white max-w-7xl w-full shadow-
[0_2px_10px_-3px_rgba(6,81,237,0.3)] lg:rounded-2xl overflow-
hidden">
        <div className="max-md:order-1 flex flex-col sm:p-8 p-4
bg-gradient-to-r from-slate-800 to-slate-500 w-full h-full">
          <div className="max-w-md space-y-12 mx-auto">
            <div>
              <h4 className="text-white text-lg font-
semibold">Create Your Account</h4>
              <p className="text-[13px] text-white mt-2">
                Welcome to our registration page! Get started by
creating your account.
              </p>
            </div>
            <div>
              <h4 className="text-white text-lg font-semibold">
                Simple &amp;amp; Secure Registration
              </h4>
              <p className="text-[13px] text-white mt-2">
                Our registration process is designed to be
straightforward and secure. We
                prioritize your privacy and data security.
              </p>
            </div>
            <div className="space-y-6">
              <button
                type="button"
                className="w-full px-5 py-2.5 flex items-center
justify-center rounded-md text-white text-base tracking-wider
font-semibold border-none outline-none bg-blue-600 hover:bg-
blue-700"
              >
                <Facebook />
                Continue with Facebook
              </button>
              {/* <button
              type="button"
              className="w-full px-5 py-2.5 flex items-center
justify-center rounded-md text-gray-800 text-base tracking-wider
font-semibold border-none outline-none bg-gray-100 hover:bg-
gray-200"
            >
              <Google />
              Continue with Google
```

```
            </button> */}

              <GoogleSignup />

              <button
                type="button"
                className="w-full px-5 py-2.5 flex items-center
justify-center rounded-md text-white text-base tracking-wider
font-semibold border-none outline-none bg-black hover:bg-[#333]"
              >
                <Apple />
                Continue with Apple
              </button>
            </div>
          </div>
        </div>
        <SignupForm />
      </div>
    </div>
  );
}
```