МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інтелектуальних інформаційних систем

_____ Євген СІДЕНКО

«____»_____ 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

РОЗРОБКА ДЕСКТОПНОГО ЧАТ-ЗАСТОСУНКУ З РОЗШИРЕНИМИ МОЖЛИВОСТЯМИ

Спеціальність 122 Комп'ютерні науки

Освітня програма «Комп'ютерні науки»

Здобувач _____ Андрій КУРИЛО

«___»_____ 2026 р.

Керівник д-р техн. наук, професор _____Олександр ГОЖИЙ

«___»_____ 2026 р.

Миколаїв – 2026

Чорноморський національний університет імені Петра Могили

Факультет	Комп’ютерних наук
Кафедра	Інтелектуальних інформаційних систем
Рівень вищої освіти	Перший (бакалаврський)
Освітній ступень	Бакалавр
Спеціальність	122 Комп’ютерні науки
Освітня програма	Комп’ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем

_____ Євген СІДЕНКО

«____» _____ 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу здобувача
Курило Андрія Миколайовича

1. Тема кваліфікаційної роботи: Розробка десктопного чат-застосунку з розширеними можливостями

Керівник роботи: Гожий Олександр Петрович, професор кафедри інтелектуальних інформаційних систем, доктор технічних наук, професор.

Затверджена наказом ЧНУ ім. Петра Могили від «25» грудня 2025 р. № 353.

2. Строк представлення кваліфікаційної роботи «___» червня 2026 р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні: очікуваним результатом є працюючий десктопний чат-застосунок для обміну повідомленнями у реальному часі, який має функцію збереження історії чатів та безпечну авторизацію користувачів.

4. Перелік питань, що підлягають розробці: Порівняльний аналіз аналогів, аналіз предметної області, обґрунтування стеку технологій, проєктування файлової структури та бази даних, програмна реалізація, тестування системи.

5. Перелік графічних матеріалів: презентація

Керівник роботи _____ Олександр ГОЖИЙ
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Андрій КУРИЛО
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата видачі завдання «21» грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН
кваліфікаційної роботи

Тема: Розробка десктопного чат-застосунку з розширеними можливостями

№	Найменування роботи	Початок	Закінчення	Примітки
1	Отримання завдання на виконання КР	21.12.2025	24.12.2025	Виконано
2	Аналіз предметної області та постановка задачі	25.12.2025	30.01.2026	Виконано
3	Огляд літературних джерел за темою кваліфікаційної роботи, порівняння аналогів	31.01.2026	01.03.2026	Виконано
4	Огляд існуючих аналогів десктопних чат-застосунків	02.03.2026	01.04.2026	Виконано
5	Реалізація обраних технологій, (Electron, React, NestJS), з аналізом результатів	02.04.2026	24.05.2026	Виконано
6	Перший попередній захист КР на засіданні комісії кафедри	25.05.2026	25.05.2026	Виконано
7	Корегування роботи за результатами попереднього захисту	26.05.2026	04.06.2026	Виконано
8	Другий попередній захист КР на засіданні комісії кафедри	05.06.2026	05.06.2026	Виконано
9	Доробка та остаточне оформлення КР	06.06.2026	14.06.2026	Виконано
10	Подання КР, її електронної копії та інших документів (відгуку, рецензії) до захисту	15.06.2026	19.06.2026	Виконано

Керівник роботи _____ Олександр ГОЖИЙ
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Андрій КУРИЛО
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата складання календарного плану «29» січня 2026 р.

АНОТАЦІЯ

до кваліфікаційної роботи

здобувача групи 401 ЧНУ ім. Петра Могили

Курило Андрія Миколайовича

на тему: «**РОЗРОБКА ДЕСКТОПНОГО ЧАТ-ЗАСТОСУНКУ З РОЗШИРЕНИМИ МОЖЛИВОСТЯМИ»**

Актуальність даної роботи полягає у зростаючому попиті на швидкі та надійні засоби комунікації, а також у необхідності створення десктопного чат-застосунку, який забезпечує миттєву доставку повідомлень у реальному часі, не перевантажує ресурси операційної системи та надає незалежну від браузера програму зі швидким відгуком інтерфейсу.

Об’єктом роботи є процес комунікації для обміну повідомленнями в режимі реального часу.

Предметом роботи є засоби розробки та програмні технології для реалізації вебсокетів, побудови клієнт-серверної архітектури і розгортання програмного продукту.

Метою роботи є створення захищеного десктопного месенджера на базі технології WebSocket для забезпечення надійного та швидкого спілкування між користувачами.

В результаті виконання роботи було розроблено повноцінний клієнт-серверний застосунок, який забезпечує реєстрацію, авторизацію користувачів, створення приватних і групових чатів, миттєвий обмін повідомленнями через WebSocket, відображення статусів онлайн та індикатора набору тексту, а також надійне збереження історії комунікації у базі даних MongoDB. Система є кросплатформною та використовує сучасний стек технологій - Electron та React на клієнті, NestJS та Socket.IO на сервері.

Дана робота складається з чотирьох розділів.

У першому розділі проведено аналіз предметної області, огляд існуючих аналогів та сформульовано основні вимоги до системи.

Другий розділ присвячений проєктуванню клієнт-серверної архітектури, розробці UML-діаграм варіантів використання, діяльності та класів.

У третьому розділі обґрунтовано вибір технологічного стеку, описано архітектуру клієнтської та серверної частин, організацію бази даних, а також рішення для хмарного розгортання на платформі Render.

У четвертому розділі наведено програмну реалізацію ключових модулів, описано дизайн застосунку та проведено тестування основних сценаріїв, що підтвердило працездатність системи.

Загальний обсяг роботи – 82 сторінок. Кваліфікаційна робота містить 3 додатків, 20 рисунків, 16 таблиць, 30 джерел посилання.

Ключові слова: десктопний чат-застосунок, обмін повідомленнями, автентифікація, клієнт-серверна архітектура, WebSocket, Electron, React, NestJS, MongoDB.

ABSTRACT

to the qualification work by the student of the group 401 of Petro Mohyla Black Sea National University

Kurylo Andrii

“DEVELOPMENT OF A DESKTOP CHAT APPLICATION WITH ADVANCED FEATURES”

The relevance of this work lies in the growing demand for fast and reliable means of communication, as well as in the need to create a desktop chat application that ensures instant delivery of messages in real time, does not overload the operating system’s resources, and provides a browser-independent program with a responsive interface.

The object of this work is the communication process for real-time message exchange.

The subject of this work is development tools and software technologies for implementing WebSockets, building a client-server architecture, and deploying the software product.

The goal of this work is to create a secure desktop messenger based on WebSocket technology to ensure reliable and fast communication between users.

As a result of this work, a full-fledged client-server application was developed that provides user registration and authorization, the creation of private and group chats, instant messaging via WebSocket, the display of online statuses and a typing indicator, as well as reliable storage of communication history in a MongoDB database. The system is cross-platform and utilizes a modern technology stack - Electron and React on the client side, NestJS and Socket.IO on the server side.

This thesis consists of four chapters.

The first chapter analyzes the subject area, reviews existing analogues, and formulates the main requirements for the system.

The second chapter is devoted to the design of the client-server architecture, the development of UML diagrams for use cases, activities, and classes.

The third chapter justifies the choice of the technology stack, describes the architecture of the client and server components, the database organization, as well as solutions for cloud deployment on the Render platform.

The fourth chapter presents the software implementation of key modules, describes the application design, and conducts testing of key scenarios, which confirmed the system’s functionality.

The total length of the thesis is 82 pages. The thesis contains 3 appendices, 20 figures, 16 tables, and 30 references.

Keywords: desktop chat application, messaging, authentication, client-server architecture, WebSocket, Electron, React, NestJS, MongoDB.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ОС – Операційна система

СУБД – система управління базами даних

API – Application Programming Interface

E2EE – End-to-End Encryption

HTTPS – Hyper Text Transfer Protocol Secure

UML – Unified Modeling Language

JWT – JSON Web Token

NoSQL – Not Only SQL

npm – Node Package Manager

SPA – Single Page Application

SQL – Structured Query Language

TCP – Transmission Control Protocol

WSS – WebSocket Secure

ВСТУП

У сучасному світі з розширенням впливу мережі Інтернет і розвитком інформаційних технологій, виникає велика проблема щодо швидкої та надійної передачі інформації на великі відстані. Месенджери для передачі інформації, на сьогодні використовуються більшою частиною людей, як для особистого спілкування, так і для її обміну у великих групах. Саме тому, попит на засоби для комунікації з кожним роком буде зростати.

Основна частина чат-застосунків, орієнтована на браузерну архітектуру, де розробники покладаються на спрощення розгортання продукту, але ціна цього – підвищене використання оперативної пам’яті, та зниження продуктивності на малопотужних пристроях, і як наслідок обмеження інтеграції з можливостями операційної системи.

Основними провайдерами на сучасному ринку комунікацій, є Telegram, Meta, Discord та Microsoft, які постійно інтегрують нові функції та вдосконалюють власні платформи. Саме вони визначають мейнстрім щодо розвитку галузі у найближчі роки. Проте, серед їхніх продуктів можна виділити надмірну кількість зайвих нововведень, через що їхні програми стають вимогливими до ресурсів ОС.

Актуальність даної теми зумовлена необхідністю розробки десктопного чат-застосунку, здатного забезпечити миттєву доставку повідомлень та підтримувати стабільну передачу повідомлень у режимі реального часу. При умові, що розроблена архітектура буде добре скомпонована, тоді дійсно можна створити застосунок, який не буде перевантажувати комп’ютер. Також, завдяки цьому, можна буде отримати незалежну від браузера програму зі швидким відгуком інтерфейсу.

Об’єктом роботи виступає процес комунікації для обміну повідомленнями у реальному часі.

Предметом роботи є засоби розробки та програмні технології, необхідні для впровадження вебсокетів та розгортання продукту в глобальній мережі.

Метою роботи є розробка та впровадження згаданих технологій для створення десктопного месенджеру на базі вебсокетів для надійного спілкування.

Для досягнення поставленої мети було поставлено такі завдання:

- аналіз предметної галузі;
- огляд існуючих аналогів;
- обґрунтування технологічного стеку проєктування структури бази даних;
- реалізація серверної логіки, графічного інтерфейсу та передачі даних у real-time;
- тестування створеної системи.

Декларація про використання ШІ. Під чат підготовки наукової роботи було використано інструмент Gemini 3.1 Pro Thinking. Розкриття факту делегування завдань генеративному ШІ

Автори заявляють про використання генеративного ШІ у процесі дослідження та підготовки рукопису. Відповідно до таксономії GAIDeT (2025), наведені нижче завдання були делеговані інструментам генеративного ШІ за повного людського нагляду:

- пошук і систематизація літератури;
- оптимізація коду;
- вичитування та редагування;
- резюмування тексту;
- адаптація та коригування емоційного тону;
- оцінювання якості;
- рекомендації.

Повну відповідальність за фінальний рукопис несе автор.

Інструменти генеративного ШІ не зазначаються як автори та не несуть відповідальності за кінцеві результати.

Декларацію подав: Курило Андрій Миколайович.

1 АНАЛІЗ ДЕСКТОПНИХ ЧАТ-ЗАСТОСУНКІВ

1.1 Аналіз системи та основних структурних особливостей

Десктопний чат-застосунок, що розробляється, являтиме собою розподілену систему типу «клієнт-сервер», яка буде орієнтована на забезпечення швидкої текстової комунікації. Головною архітектурною особливістю спроєктованого рішення стане його середовище виконання – на відміну від популярних веб-орієнтованих аналогів, застосунок функціонуватиме як повноцінний незалежний процес операційної системи.

Завдяки використанню кросплатформних технологій програма матиме нативну підтримку на базі Windows, macOS та Linux. Таке структурне рішення дозволить реалізувати глибоку інтеграцію з програмними інтерфейсами ОС – забезпечити автономну роботу у фоновому режимі, налаштувати виведення системних сповіщень, а також гарантувати більш оптимізоване споживання оперативної пам’яті комп’ютера порівняно з браузерними клієнтами.

В основу побудови системи буде покладено класичну трирівневу архітектурну модель, що гарантуватиме суворе розмежування відповідальності та бізнес-логіки між програмними вузлами:

- першим рівнем виступатиме клієнтська частина, завданням якої буде обробка графічного інтерфейсу, організація зручної взаємодії з користувачем та локальна обробка інтерфейсних подій;
- другий рівень – серверна частина, що виконуватиме роль центрального комунікаційного вузла. Цей рівень аутентифікуватиме клієнтів, забезпечуватиме перевірку безпеки з’єднань, керуватиме активними сесіями та маршрутизуватиме потоки повідомлень;
- третім етапом стане рівень зберігання даних, який гарантуватиме надійне, структуроване та безпечне збереження всієї історії листування і профілів користувачів.

Ключовою технічною особливістю майбутнього застосунку стане гібридний підхід до організації мережевої взаємодії. Для виконання базових разових операцій реєстрації, авторизації та завантаження початкових станів буде передбачено використання традиційної архітектури REST API на базі протоколу HTTP. Однак для основного функціоналу – обміну повідомленнями в режимі реального часу, буде реалізовано технологію WebSocket.

Цей протокол створюватиме постійно відкритий двосторонній канал зв’язку, що дозволить серверу миттєво надсилати нові повідомлення клієнтам без необхідності регулярного опитування з їхнього боку, тим самим мінімізуючи мережеві затримки.

Структура бази даних також проєктуватиметься з урахуванням специфіки сучасних систем обміну повідомленнями. Обрана нереляційна документо-орієнтована модель дозволить максимально гнучко зберігати інформацію про чати та повідомлення у вигляді JSON-подібних структур.

Такий підхід дасть змогу в майбутньому легко розширювати функціонал, додавати статуси прочитання, типи чатів, підтримку вкладень чи перебудови загальної архітектури бази даних.

1.2 Огляд існуючих десктопних чат-застосунків

Для визначення основних вимог до розроблюваного програмного продукту, а також для об’єктивного виявлення слабких місць існуючих аналогів, було проведено ґрунтовний огляд сучасного ринку систем обміну повідомленнями.

Цей огляд має на меті не лише проаналізувати поточний стан індустрії, але й визначити, які саме архітектурні рішення, сильні та слабкі сторони існуючих аналогів програмних продуктів, безпосередньо впливають на їхню популярність серед користувачів.

На сьогодні ринок пропонує велику кількість рішень, але доцільно розглянути найбільш популярні десктопні клієнти – Telegram [1], Discord [2] та Signal [3]. Для виявлення ключових переваг та недоліків кожного з клієнтів,

проведено їх огляд із подальшим зведенням даних у описові таблиці.

Telegram

Telegram – це багатофункціональний кросплатформний месенджер, побудований на основі клієнт-серверної архітектури. Він набув широкої популярності завдяки високій швидкості передачі даних, що забезпечується найкращим криптографічним протоколом MTProto, оптимізованим для роботи навіть при нестабільному інтернет-з'єднанні.

Таблиця 1.1 – Опис Telegram

Назва	Telegram
Архітектура	Децентралізована хмарна архітектура з використанням власного протоколу MTProto
Тип застосунку	Кросплатформний месенджер з акцентом на швидкість та безпеку
Мови програмування	C++ (основний клієнт і сервер), Java/Kotlin (Android), Swift (iOS)
Базовий функціонал	Приватні та групові чати, аудіо- та відеодзвінки, обмін файлами, хмарне зберігання
Розширені можливості	– секретні чати з наскрізним шифруванням (E2EE) та самознищенням повідомлень; – канали для публікацій, боти для автоматизації, групи до 200 000 учасників; – голосові чати та відеотрансляції для великих груп.
Безпека	E2EE лише для «секретних чатів» та голосових/відеодзвінків. Звичайні хмарні чати та групові листування не мають E2EE. Використовується власний протокол MTProto.
Переваги	– одна з найшвидших систем доставки повідомлень; – підтримка файлів будь-якого формату та розміру до 2 ГБ з преміум; – велика кількість клієнтів з відкритим вихідним кодом.
Недоліки	– E2EE обмежене лише «секретними чатами», звичайні хмарні чати не мають наскрізного шифрування; – обов'язкова прив'язка до номера телефону.
Посилання	https://telegram.org [1]

Підсумовуючи наведені характеристики вище (табл. 1.1), можна зазначити, що Telegram є потужною комунікаційною платформою з високою швидкодією та гнучкою хмарною синхронізацією. Використання оптимізованого протоколу MTProto забезпечує системі стійкість при великих навантаженнях, швидку передачу файлів та ефективну роботу гіганських групових чатів.

Проте така архітектура має суттєвий компроміс у сфері приватності, оскільки стандартні повідомлення не захищені E2EE, залишаючи серверу доступ до історії листування.

Крім того, жорстка прив’язка облікового запису до номера мобільного телефону суттєво обмежує анонімність користувачів. Зважаючи на ці архітектурні та безпекові обмеження, доцільним є подальше дослідження альтернативних рішень на ринку десктопних месенджерів.

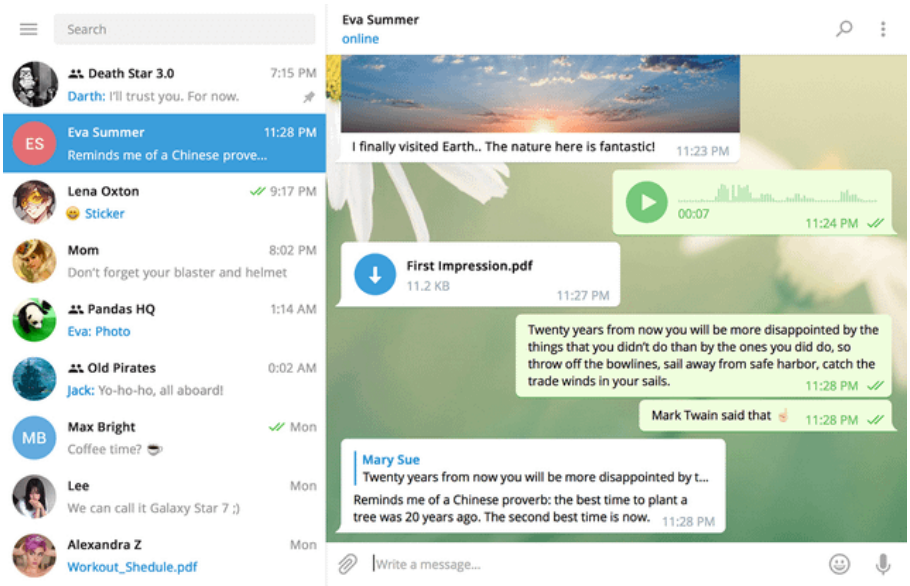

Рисунок 1.1 – Огляд інтерфейсу Telegram [1]

Discord

Discord – це багатофункціональна комунікаційна платформа, що початково розроблялася для геймерів, але поступово стала універсальним інструментом для створення тематичних спільнот. Основна архітектурна особливість полягає в організації спілкування через «сервери», які об’єднують текстові, голосові та

відеоканали. Завдяки використанню технології WebRTC [13], платформа забезпечує високоякісний зв’язок у реальному часі з мінімальними затримками.

Таблиця 1.2 – Опис Discord

Назва	Discord
Архітектура	Централізована клієнт-серверна архітектура, оптимізована для роботи в реальному часі.
Тип застосунку	Кросплатформна система для спільнот
Мови програмування	Elixir, Rust, Go для високонавантаженої серверної частини; JavaScript/TypeScript для клієнтської частини
Базовий функціонал	Текстові, голосові та відеоканали на серверах; обмін файлами та зображеннями; особисті повідомлення та групові чати до 10 учасників
Розширені можливості	− організація серверів із гнучкою системою ролей та ієрархії для учасників спільнот; − демонстрація екрану та стрімінг контенту в реальному часі; − потужна система інтеграцій та ботів для автоматизації задач.
Безпека	Шифрування даних при передачі (TLS). Наскрізне шифрування (E2EE) відсутнє
Переваги	− найкраща система ролей та прав доступу для керування великими групами; − низька затримка голосового та відеозв’язку завдяки спеціалізованим серверам.
Недоліки	− відсутність E2EE; − середнє споживання оперативної пам’яті через використання Electron.
Посилання	https://discord.com [2]

Аналіз платформи Discord показує, що її головною перевагою є потужний інструментарій для управління масштабними спільнотами завдяки гнучкій системі ролей та розширеній інтеграції ботів. Використання технології WebRTC [13] забезпечує високу якість голосових і відеотрансляцій із мінімальною затримкою, що робить систему ідеальною для групової взаємодії в реальному часі.

Однак суттєвим недоліком месенджера залишається повна відсутність E2EE, що значно знижує рівень конфіденційності користувацького листування.

Крім того, клієнтський застосунок платформи відрізняється середнім споживанням оперативної пам’яті через специфіку неоптимізованого фреймворку Electron [4].

Таким чином, незважаючи на чудову продуктивність серверної частини, висока ресурсоємність десктопного клієнта та відкритість даних для сервера роблять Discord не ідеальним варіантом для легкого й приватного обміну повідомленнями.

Рисунок 1.2 – Огляд інтерфейсу Discord [2]

Signal

Signal – це орієнтований на конфіденційність месенджер з відкритим вихідним кодом, який вважається світовим стандартом безпечного обміну даними. Основна архітектурна особливість платформи полягає у використанні найкращого на даний час протоколу наскрізного шифрування, що гарантує доступ до інформації лише безпосереднім учасникам діалогу.

Таблиця 1.3 – Опис Signal

Назва	Signal
Архітектура	Централізована клієнт-серверна архітектура з акцентом на конфіденційність
Тип застосунку	Кросплатформний месенджер для безпечного спілкування
Мови програмування	Java (Android), Swift (iOS), JavaScript/TypeScript (Desktop), Rust/Go (серверні компоненти).
Базовий функціонал	Приватні та групові чати, аудіо- та відеодзвінки, обмін файлами та зображеннями.
Розширені можливості	– наскрізне шифрування (E2EE) за замовчуванням для всіх повідомлень, дзвінків і файлів; – зникаючі повідомлення з гнучкими налаштуваннями таймера; – підтримка signal protocol, що вважається «золотим стандартом» криптографії.
Безпека	Найвищий рівень безпеки – E2EE для всіх типів комунікації за замовчуванням. Мінімальний збір метаданих. Використовується відкритий та перевірений Signal Protocol.
Переваги	– найвищий рівень конфіденційності, мінімальний збір метаданих; – усі вихідні коди клієнтів та сервера є відкритими;
Недоліки	– для desktop-версії потрібен спочатку зареєстрований мобільний клієнт;
Посилання	https://signal.org [3]

Аналіз платформи Signal показує, що її головною перевагою є безпрецедентний рівень приватності завдяки використанню еталонного криптографічного алгоритму Signal Protocol. E2EE та мінімальний збір метаданих роблять цю систему загальновизнаним світовим стандартом безпечного спілкування.

Однак суттєвим недоліком месенджера залишається відсутність повністю автономної десктопної версії, оскільки для її роботи вимагається обов’язкова попередня реєстрація через мобільний застосунок.

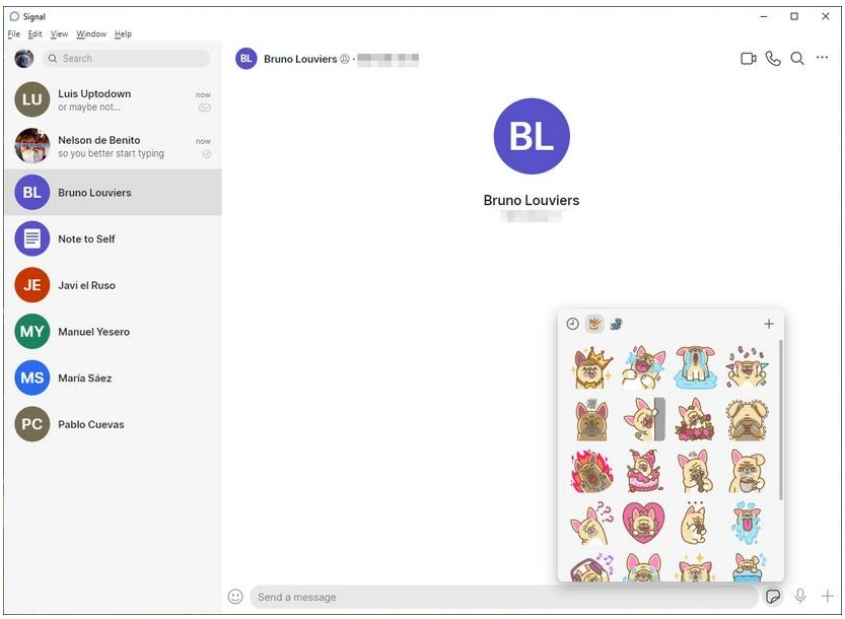

Рисунок 1.3 – Огляд інтерфейсу Signal [3]

Таким чином, незважаючи на найкращу на ринку модель криптографічного захисту, архітектурна залежність від мобільних пристроїв та телефонних номерів робить Signal недостатньо гнучким рішенням для незалежної десктопної комунікації.

1.3 Постановка задачі для розробки десктопного чат-застосунку

Головною метою роботи є розробка програмного забезпечення для двостороннього обміну текстовими повідомленнями в реальному часі на базі клієнт-серверної взаємодії.

Об’єктом роботи виступає безпосередньо процес організації мережевої комунікації та передачі даних між віддаленими вузлами. Відповідно, предметом роботи є програмні методи та технологічні засоби, необхідні для реалізації кросплатформних віконних застосунків та серверних інтерфейсів обміну повідомленнями.

Для досягнення поставленої мети необхідно вирішити комплекс взаємопов’язаних завдань. Насамперед вимагається провести ґрунтовний аналіз предметної області, дослідити наявні аналоги на ринку комунікаційного програмного забезпечення та визначити їхні переваги й недоліки.

На основі отриманих даних необхідно сформулювати вимоги до майбутньої системи, яка повинна підтримувати постійне відкрите з'єднання без необхідності регулярних повторних запитів до сервера для забезпечення миттєвої доставки повідомлень.

Також важливим завданням є забезпечення ефективного збереження історії листування користувачів та швидкого пошуку інформації у базі даних. Окрім того, обов'язковою вимогою є впровадження механізмів захисту конфіденційних даних шляхом безпечної авторизації.

Процес розробки програмного продукту потребує чіткої формалізації вимог до його кінцевого функціоналу. Для систематизації постановки задачі основні вимоги до розроблюваного чат-застосунку зведено у таблицю 1.4.

Таблиця 1.4 – Основні вимоги та завдання для розроблюваного чат-застосунку

Категорія вимог	Зміст задачі та вимоги до реалізації
Мережева взаємодія	Забезпечення постійного двостороннього зв'язку між клієнтом та сервером, миттєва передача даних у реальному часі без затримок.
Серверна логіка	Обробка реєстрації та авторизації користувачів, прийом та маршрутизація потокових повідомлень, управління доступом до сховища даних.
Клієнтська частина	Створення кросплатформного графічного інтерфейсу, відображення історії повідомлень, підтримка темної теми, редагування профілю. Розширені можливості - надсилання файлів, перегляд зображень, сповіщення про нові повідомлення, індикатор друку контакту, статус «онлайн/офлайн».
Безпека та надійність	Хешування паролів, автентифікація за JWT. Перевірка членства в чаті.

Успішне визначення вимог у постановці задачі завдань дозволить створити повноцінний програмний продукт, готовий до встановлення та безперебійної експлуатації кінцевими користувачами.

Завдяки впровадженню технології WebSocket система гарантуватиме високу швидкість обміну даними, а кросплатформна архітектура забезпечить її універсальність.

Висновки до розділу 1

У першому розділі було проведено загальний аналіз предметної області та проведено огляд існуючих систем обміну повідомленнями, що підтвердило доцільність створення кросплатформного месенджера. Визначено, що майбутня система будуватиметься за трирівневою моделлю з чітким розмежуванням обов’язків між клієнтом, сервером та базою даних. Окрему увагу приділено механізму постійного двостороннього з’єднання, яке гарантує доставку повідомлень у реальному часі без необхідності періодичного опитування сервера, суттєво знижуючи навантаження на мережу.

На основі цих досліджень було сформульовано остаточну постановку задачі та класифіковано функціональні вимоги до програмного продукту, що формує необхідне підґрунтя для переходу до етапу проєктування архітектури системи у наступному розділі.

2 МОДЕЛЮВАННЯ СТРУКТУРИ ДЕКСТОПНОГО ЧАТ-ЗАСТОСУНКУ

2.1 Розробка загальної структурної схеми чат-застосунку

На основі вимог, сформульованих у попередньому розділі, було розроблено та змодельовано загальну архітектуру десктопного чат-застосунку. Система будується за класичною трирівневою моделлю, яка складатиметься з клієнтського рівня, десктопний застосунок на базі Electron [4] та React [3], серверного рівня, NestJS [8] із підтримкою Socket.IO [6], та рівня зберігання даних. Така структура дозволяє чітко розмежувати функціональність, де клієнт відповідає за відображення інтерфейсу та обробку дій користувача, сервер забезпечує виконання бізнес-логіки та маршрутизацію повідомлень, а база даних слугує централізованим сховищем інформації. Клієнтська частина проєктується на двох основних рівнях.

Перший рівень - це оболонка на базі фреймворку Electron, яка відповідає за створення нативного вікна застосунку, обробку системних сповіщень, підтримку фонової роботи в треї та функцію автозапуску.

Другий рівень - це React-застосунок, який відповідає безпосередньо за побудову графічного інтерфейсу. Компоненти користувацького інтерфейсу будуть логічно поділені на такі блоки:

- форма реєстрації та авторизації;
- список активних чатів;
- робоча область для перегляду історії та надсилання повідомлень;
- панель пошуку контактів;
- вікно налаштувань профілю користувача.

Для забезпечення роботи в режимі реального часу використовується Socket.IO-клієнт, який створює постійне з’єднання з сервером та дозволяє обмін подіями між учасниками певного чату.

Серверна частина реалізована на платформі NestJS [8] та структурована на чотири логічні модулі.

Перший модуль відповідає за реєстрацію нових користувачів та вхід у систему. Він обробляє дані, що передаються від клієнта до сервера, перевіряє унікальність електронної пошти, хешує паролі за допомогою bcrypt, створює нові записи в базі даних, а також генерує та перевіряє JWT-токени для забезпечення авторизації.

Другий модуль керує профілями користувачів, забезпечуючи збереження та оновлення інформації про імена, аватари й статуси присутності. Він також дозволяє шукати користувачів за іменем.

Третій модуль відповідає за створення чатів. Він генерує нові документи в базі даних, додає учасників до розмов, призначає адміністраторів для групових бесід та надає можливість редагувати склад учасників – додавати або видаляти їх. Окрім цього, даний модуль обробляє надсилання та отримання повідомлень. Він отримує дані від відправника, зберігає їх у колекції повідомлень у MongoDB з позначками часу та посиланням на певний чат, після чого автоматично ініціює доставку.

Четвертий модуль реалізує шлюз WebSocket [17] на основі Socket.IO [6]. Саме він керує клієнтськими підключеннями, підписує користувачів на події відповідних чатів, забезпечує доставку нових повідомлень у режимі реального часу всім учасникам, а також відстежує підключення та відключення для миттєвого оновлення статусів присутності.

Модулі автентифікації та профілів формують надійний фундамент для захищеного зберігання користувацьких даних і управління сесіями. Своєю чергою, тісна взаємодія модуля чатів зі шлюзом WebSocket дозволяє не лише структуровано зберігати історію листування в MongoDB [7], але й миттєво доставляти повідомлення та оновлювати статуси користувачів без додаткового навантаження на сервер.

Такий підхід робить програмний код легко масштабованим і у майбутньому кожен із перелічених модулів можна розширювати незалежно від інших.

Для наочного відображення архітектури було побудовано діаграму компонентів системи (рис. 2.1).

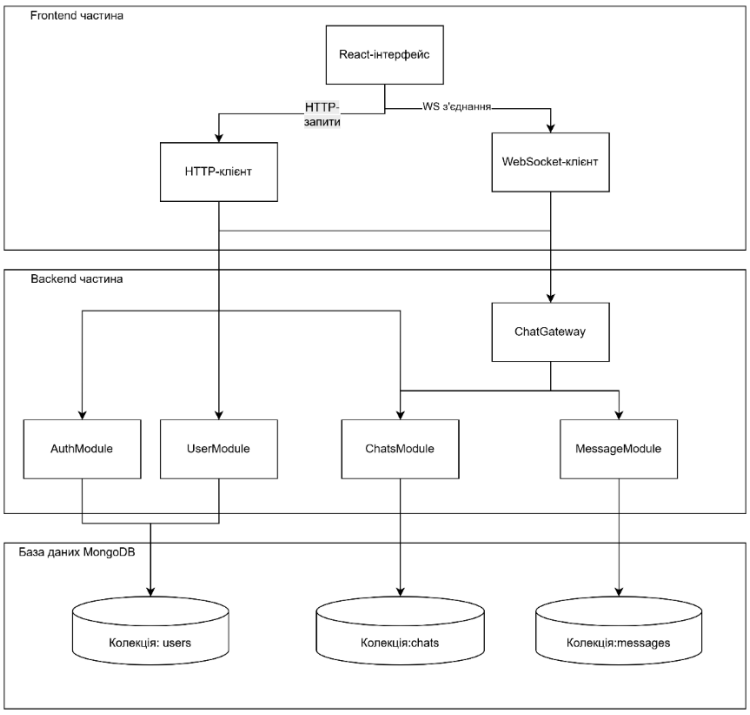

Рисунок 2.1 – Діаграма компонентів системи

На діаграмі відображено три основні вузли, зокрема Desktop Client, Backend Server та базу даних. Клієнт за допомогою HTTP/HTTPS-запитів взаємодіє з REST API сервера для здійснення авторизації та отримання історії повідомлень, тоді як через WebSocket-з’єднання забезпечується взаємодія з Gateway для обміну повідомленнями в режимі реального часу.

Сервер, своєю чергою, через Mongoose [11] здійснює взаємодію з MongoDB для збереження та подальшого отримання даних.

2.2 Визначення сценаріїв використання та поведінки системи

Після формування загальної архітектурної моделі системи виникає об’єктивна необхідність деталізувати сценарії її практичного використання.

Проєктування функціональної поведінки застосунку здійснюється шляхом розробки варіантів її використання – Use Cases. Цей етап моделювання дозволяє не

лише перерахувати доступні функції, але й суворо формалізувати взаємодії кінцевих користувачів із клієнтським та серверним рівнями системи.

У загальному вигляді базовий функціонал системи включає процеси реєстрації, авторизації за допомогою токенів, пошуку контактів у базі даних, ініціалізації приватних і групових чатів, надсилання та м'якого видалення повідомлень, а також управління налаштуваннями профілю.

Для деталізації найважливіших системних процесів розроблено покрокові сценарії використання.

Нижче (табл. 2.1) наведено алгоритм первинної взаємодії користувача із застосунком - проходження реєстрації. Цей прецедент ілюструє процес валідації вхідних форм, криптографічного захисту пароля та генерації авторизаційних JWT-токенів.

Таблиця 2.1 - Use Case проходження реєстрації

Use Case Name	Проходження реєстрації в системі
Level	Ціль користувача
Primary Actor	Новий користувач (відвідувач), який вперше відкрив застосунок
Stakeholders and Interests	Відвідувач хоче створити власний обліковий запис для отримання можливості спілкуватися з іншими учасниками мережі
Preconditions	Користувач не має активного сеансу авторизації, програму щойно запущено
Success Guarantee	Обліковий запис успішно створено, відомості про власника збережено в колекції users бази даних MongoDB, клієнт отримує валідний JWT-токен та автоматичний доступ до системи
Main process	– відвідувач відкриває програму та на стартовому екрані бачить панель входу; – обирає опцію «Зареєструватися» для створення нового облікового запису; – графічний інтерфейс відображає форму з такими полями - ім'я, електронна пошта, пароль, повторення пароля; – відвідувач заповнює всі обов'язкові поля та підтверджує дію кнопкою реєстрації; – клієнтська частина проводить локальну валідацію; – у разі успішної локальної валідації клієнт відправляє HTTP POST-запит із даними до серверної частини (API); – серверний модуль перевіряє унікальність вказаної пошти шляхом пошуку збігів у колекції users бази даних MongoDB; – у разі унікальності пошти сервер пропускає пароль через функцію криптографічного хешування bcrypt;

Кінець таблиці 2.1

	– сервер формує та зберігає новий документ користувача з хешованим паролем до колекції users; – сервер генерує токен доступу (JWT) на основі ID нового користувача та відправляє його у відповіді клієнту; – клієнтський застосунок зберігає токен локально, і відвідувач автоматично перенаправляється до головного комунікаційного вікна програми.
Frequency of Occurrence	Одноразово для кожного нового користувача

Після успішного створення облікового запису та проходження авторизації користувач отримує доступ до головного вікна програми зі списком активних чатів та контактів. На цьому етапі базовою функцією є ініціалізація нових діалогів.

Для розгляду іншого ключового Use Case системи доцільно взяти сценарій створення групової розмови.

У таблиці 2.2 детально описано алгоритм взаємодії користувача та системи під час виконання саме цього процесу.

Таблиця 2.2 – Use Case формування групової бесіди

Use Case Name	Формування групової бесіди
Scope	Програма для обміну повідомленнями
Level	Ціль користувача
Primary Actor	Зареєстрований учасник (організатор)
Stakeholders and Interests	Учасник бажає створити спільний простір для одночасного листування з декількома людьми
Preconditions	Користувач пройшов процедуру входу
Success Guarantee	– організатор натискає кнопку «Створити групу»; – інтерфейс відображає вікно з рядком імені та списком доступних контактів; – організатор вводить назву розмови та вибирає потрібних людей; – організатор підтверджує дію кнопкою «Створити»; –

Кінець таблиці 2.2

	– програма переконується, що ім’я не пусте, а кількість вибраних людей щонайменше два; – сервер генерує новий документ у колекції чатів репозиторію mongodb з тегом «Група»; – ідентифікатори всіх запрошених включаючи організатора, вносяться в масив members; – організатор автоматично отримує права керування; – кожен підключений учасник отримує сигнал через постійний канал websocket про додавання до щойно створеної групи; – у списку доступних діалогів для кожного запрошеного з’являється новий запис.
Frequency of Occurrence	Регулярно

Як видно з наведеного сценарію, процес формування групової бесіди поєднує в собі як традиційну архітектуру для збереження структури чату в базі даних, так і подієво-орієнтовану модель WebSocket для оновлення клієнтських інтерфейсів у реальному часі.

2.3 Логіка роботи основних функціональних процесів

Після визначення загальних вимог було вирішено деталізувати внутрішню механіку розроблюваного застосунку. У цьому розділі здійснюється перехід від абстрактного опису функцій та створення Use Case до конкретного проєктування.

Основне завдання цього етапу полягає в покроковій демонстрації процесу обробки системою дій користувача, починаючи від клієнтського інтерфейсу і закінчуючи логікою бекенду та операціями з базою даних.

Для візуалізації паралельних процесів і умовних розгалужень було створено UML-діаграми діяльності. Цей інструмент моделювання дозволяє врахувати асинхронну природу клієнт-серверної архітектури та відобразити логіку маршрутизації даних.

Завдяки їм можна наочно простежити життєвий цикл ключових функцій системи – від ініціації події в інтерфейсі користувача до її обробки на рівні бекенду. Для комплексного розуміння роботи системи нижче було змодельовано процеси.

Процес надсилання повідомлення. Оскільки базовою функцією будь-якого месенджера є обмін інформацією, найперше розглянуто алгоритм надсилання текстового повідомлення (рис. 2.2).

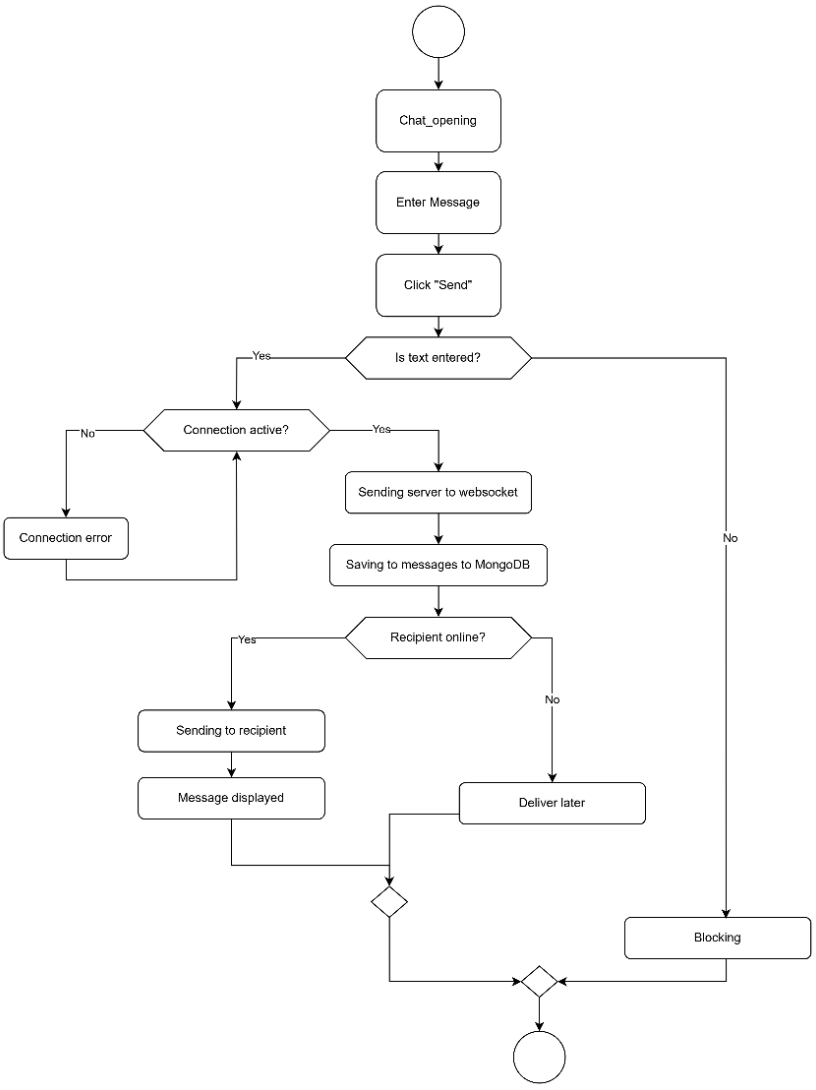

Рисунок 2.2 – Діаграма для діяльності надсилання повідомлення

Процес видалення повідомлення з групового чату. Наступним важливим етапом є управління контентом, яке вимагає перевірки прав учасників бесіди під час видалення повідомлень (рис. 2.3).

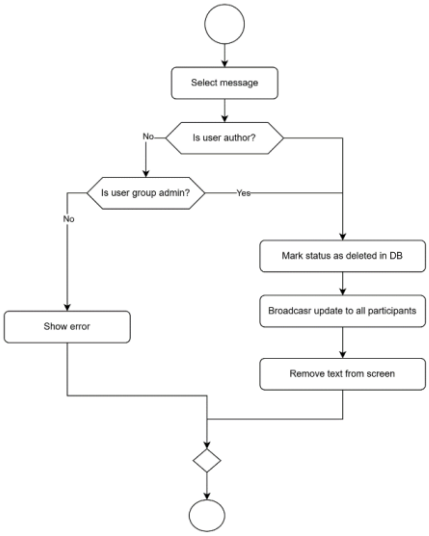

Рисунок 2.3 – Діаграма для діяльності видалення повідомлення з групового чату

Процес створення групового чату. Третім ключовим алгоритмом, який вимагає синхронізації даних між багатьма клієнтами, є створення нового групового чату.

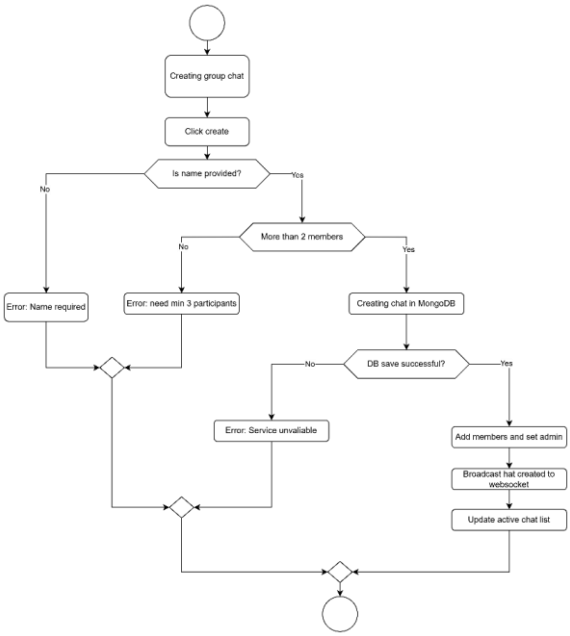

Рисунок 2.4 – Діаграма для діяльності створення групового чату

2.4 Розробка структури класів системи

Розробка діаграми класів (рис. 2.5) є одним із ключових етапів побудови архітектури програмного забезпечення. Вона надає графічну візуалізацію

структури системи, відображаючи класи, їхні атрибути, методи та взаємозалежності.

Це дозволяє визначити основні компоненти системи, спроєктувати їхню взаємодію ще на початкових етапах розробки, а також виступає зв'язуючою ланкою між логічною моделлю бази даних і реалізацією функціональних можливостей на серверній частині.

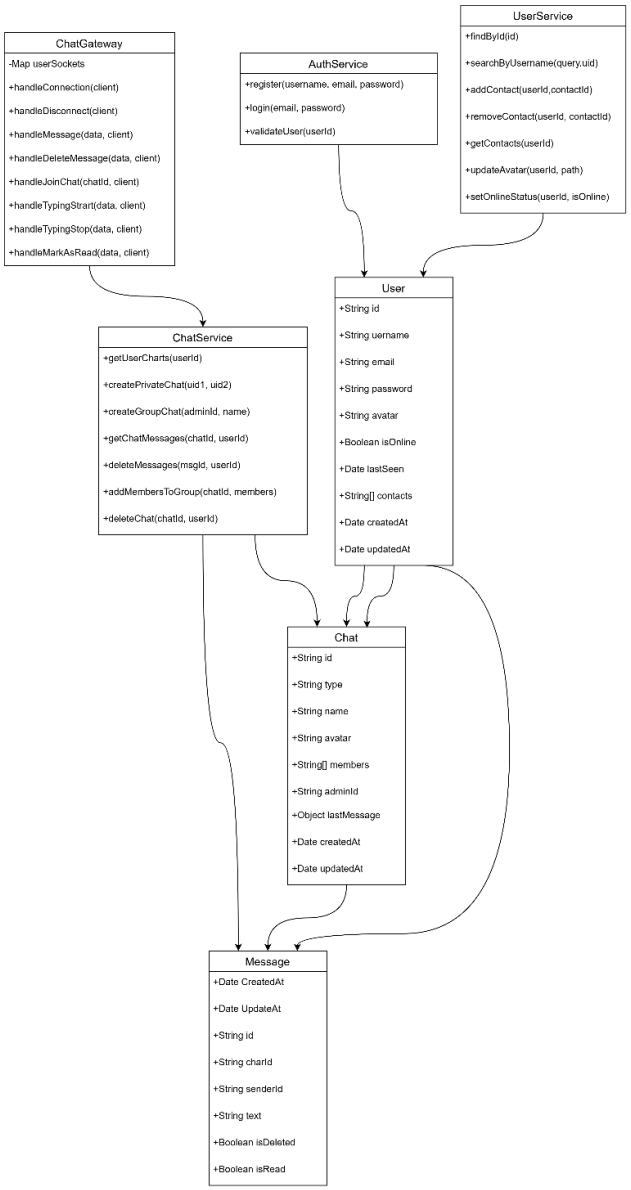

Рисунок 2.5 – Діаграма класів десктопного чат-застосунку

Відповідно до наведеної вище діаграми, основні класи системи, що проєктується, логічно розділятимуться на моделі даних та сервіси бізнес-логіки:

– клас користувача «User» – основна сутність системи, яка міститиме унікальний ідентифікатор id, унікальні ім’я користувача username та електронну пошту email, посилання на аватар avatar і зберігатиме статусні дані, зокрема мережевий статус isOnline та час останньої активності lastSeen, причому зв’язок із контактами реалізовуватиметься через масив contacts, що міститиме посилання на інші об’єкти користувачів;

– клас чату «Chat», що описуватиме логіку приватних діалогів та групових розмов, де атрибут type визначатиме тип бесіди, при цьому для групових чатів додатково заповнюватимуться поля назви name, зображення avatar та ідентифікатора адміністратора adminId, також він міститиме масив ідентифікаторів учасників members, а для оптимізації завантаження списку чатів зберігатиметься об’єкт останнього повідомлення lastMessage замість завантаження всього масиву історії листування;

– клас повідомлення «Message», що відображатиме одиницю інформаційного обміну, міститиме текстовий вміст, зовнішні ключі ідентифікатора чату chatId і відправника senderId, а також булеві статуси прочитання isRead та видалення isDeleted.

Сервіси бізнес-логіки (NestJS Providers)

Проєктовані сервіси взаємодіятимуть між собою через механізми впровадження залежностей Dependency Injection, що забезпечуватиме слабку зв’язаність компонентів і полегшуватиме модульне тестування. До складу цієї функціональної групи належатимуть:

– сервіс безпеки та автентифікації AuthService, який відповідатиме за реєстрацію нових облікових записів за допомогою методу register, авторизацію через login та перевірку валідності користувача через validateUser;

– сервіс управління користувачами UsersService, що інкапсулюватиме логіку пошуку за ім’ям, керування списком контактів за допомогою методів

addContact та removeContact, оновлення аватара профілю та синхронізацію мережевого статусу через setOnlineStatus;

– сервіс управління чатами ChatsService, який керуватиме всією логікою повідомлень, зокрема створенням приватних та групових кімнат, отриманням історії листування, додаванням учасників до групи, а також м’яким видаленням окремих повідомлень і цілих бесід;

– контролер мережевої взаємодії в реальному часі на базі WebSocket, у якому для відстеження активних сесій використовуватиметься структура userSockets, причому цей компонент оброблятиме події підключення, надсилання та видалення повідомлень, приєднання до кімнати через handleJoinChat, індикацію набору тексту через події typing:start і typing:stop, а також відмітки про прочитання через message:markAsRead.

Запропонована структура класів забезпечуватиме чітке розмежування між моделями збереження даних та сервісами бізнес-логіки, що закладе надійну архітектурну основу для подальшої програмної реалізації чат-застосунку.

Висновки до розділу 2

У другому розділі спроєктовано трирівневу клієнт-серверну архітектуру чат-застосунку з чітким розмежуванням інтерфейсу, бізнес-логіки та збереження даних. Розроблено сценарії використання для авторизації та групової комунікації, а також діаграми діяльності ключових процесів.

Побудовано логічну структуру даних у вигляді UML-діаграми класів із визначенням основних сутностей та їхніх зв’язків. Отримана модель є основою для подальшого проєктування системи.

3 АРХІТЕКТУРА ТА ТЕХНОЛОГІЧНЕ ЗАБЕЗПЕЧЕННЯ ЧАТ-ЗАСТОСУНКУ

3.1 Архітектура програмної системи

На етапі практичної реалізації проєкту розробляється загальна архітектура чат-застосунку, яка базується на класичній трирівневій клієнт-серверній моделі. Сучасна архітектура вимагає глибокого аналізу структурних патернів перед початком написання коду, оскільки від правильно закладеного фундаменту залежить подальша працездатність, продуктивність та можливості масштабування готового продукту.

При проєктуванні системи закладається чіткий розподіл обчислювального навантаження та функціональних обов’язків між трьома основними складовими, якими є інтерфейсна частина. що розташовується на десктопному клієнті на базі фреймворку Electron [4], мережевий рівень із серверною логікою на базі NestJS [8] та рівень збереження даних у вигляді нереляційної СУБД MongoDB [7].

Такий підхід забезпечує високу модульність програмного рішення, адже кожен компонент функціонує як незалежний об’єкт. Це дозволяє дотримуватися принципу єдиної відповідальності на рівні архітектури, що суттєво полегшує процес написання коду, ізольованого тестування окремих модулів та подальшого розширення системи.

Більше того, відокремлення клієнтської частини від серверної дозволяє в майбутньому легко інтегрувати нові платформи, наприклад мобільний застосунок, без необхідності переписування серверного ядра.

Клієнт-серверна архітектура є найбільш доцільною та безпечною моделлю для системи обміну повідомленнями з кількох ключових причин:

– надійність та централізованість даних, оскільки усі повідомлення, історія чатів та персональні налаштування користувачів зберігаються на захищеному сервері, тому користувач не втратить свою інформацію навіть у разі перевстановлення клієнтського застосунку чи втрати апаратного пристрою;

– безпека та контроль доступу, завдяки чому конфіденційна інформація та паролі хешуються і перевіряються виключно на бекенді, причому клієнт ніколи не отримує прямого доступу до бази даних, а сервер надійно перевіряє права доступу користувача за допомогою криптографічних токенів перед тим, як надіслати йому історію конкретного чату;

– безперервна синхронізація, за якої база даних виступає єдиним джерелом правди, що гарантує ідентичне відображення актуальної інформації на всіх підключених пристроях користувача.

Альтернативні архітектурні рішення, такі як однорангові мережі, були проаналізовані та відкинуті під час проєктування. Хоча P2P-архітектура забезпечує високий рівень децентралізації, вона потребує надзвичайно складних механізмів виявлення вузлів і, найголовніше, не забезпечує можливості гарантованої доставки повідомлень у тих випадках, коли пристрій отримувача тимчасово не підключений до мережі.

У класичній клієнт-серверній моделі сервер виступає надійним посередником, який накопичує повідомлення та автоматично доставляє їх клієнту одразу після відновлення з’єднання.

3.2 Клієнтська частина застосунку

Проєктування та розробка клієнтської частини сучасних десктопних систем вимагає використання інструментів, здатних забезпечити високу швидкість створення користувацького інтерфейсу, кросплатформність та безперебійний доступ до системних ресурсів.

Традиційні підходи до розробки десктопних застосунків, тобто нативна розробка, передбачають написання окремого коду для кожної операційної системи, наприклад на C# для Windows або Swift для macOS, що суттєво уповільнює цикл розробки та ускладнює підтримку проєкту.

З метою оптимізації цього процесу було обрано підхід на основі сучасних вебтехнологій. Головним інструментом для перенесення вебсередовища на десктоп

виступає фреймворк Electron [4], а безпосередньо для побудови графічного інтерфейсу застосовується бібліотека React [5]. Electron – це платформа з відкритим вихідним кодом, створена компанією GitHub [16].

Вона дозволяє розробляти повноцінні кросплатформні застосунки за допомогою JavaScript, HTML та CSS. Під капотом ця технологія є симбіозом двох потужних компонентів, браузерного рушія рендерингу Chromium та серверного середовища виконання Node.js [10].

Завдяки інтеграції з Node.js програма отримує прямий доступ до апаратних ресурсів комп'ютера. На відміну від звичайної вкладки у браузері, застосунок на базі Electron здатен безпосередньо взаємодіяти з файловою системою операційної системи через File System API, працювати з мережевими протоколами та викликати нативні API, наприклад для створення іконок у треї, управління вікнами або виклику системних сповіщень.

Схематичне зображення загальної архітектури фреймворку Electron наведено на рисунку 3.1.

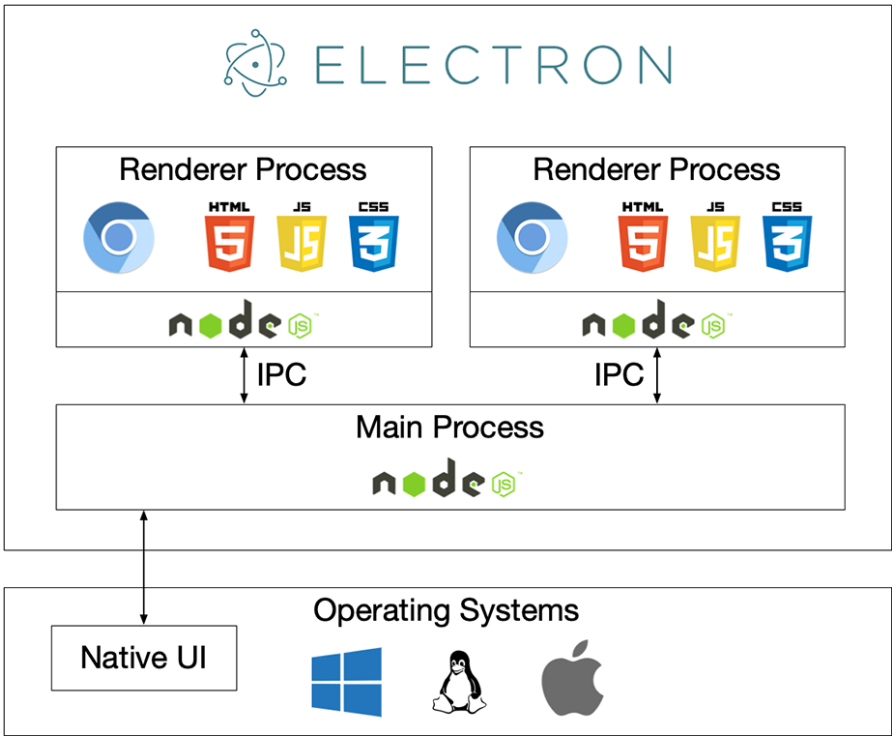

Рисунок 3.1 – Загальна архітектура фреймворку Electron

Архітектура функціонування Electron базується на строгій ізоляції та використанні двох основних типів процесів:

– головного процесу (Main Process), який є точкою входу в програму, працює у середовищі Node.js, не має візуального інтерфейсу та відповідає за управління життєвим циклом застосунку, створення системних вікон і взаємодію з ядром операційної системи;

– процесів рендерингу (Renderer Processes), що відповідають виключно за відображення графічного інтерфейсу, причому кожне вікно застосунку запускається у власному процесі рендерингу, який поводить себе як окрема вебсторінка, а з міркувань безпеки ці процеси не мають прямого доступу до Node.js і спілкуються з головним процесом через спеціальний міст – механізм міжпроцесної взаємодії (Inter-Process Communication).

Вибір Electron супроводжується певними компромісами. Основні сильні та слабкі сторони цієї технології можна розділити на дві категорії:

– переваги, серед яких використання єдиної кодової бази для різних операційних систем (Code Reusability), прямий доступ до масштабної екосистеми модулів npm та ідентичне відображення інтерфейсу на всіх платформах завдяки вбудованому рушію Chromium;

– недоліки, що включають значний розмір кінцевого виконуваного файлу через постачання Chromium разом із додатком та підвищене споживання оперативної пам’яті порівняно з програмами, які написані мовами нижчого рівня.

Порівняння Electron з актуальними інструментами для створення десктопних застосунків наведено у таблиці 3.1.

Таблиця 3.1 – Порівняльна таблиця Electron з аналогами

Характеристика	Electron	Tauri	NW.js
Базова архітектура	Chromium + Node.js	Системний WebView + Rust	Chromium + Node.js
Розмір кінцевого білда	Значний (від 80 МБ)	Мінімальний (від 5 МБ)	Значний

Кінець таблиці 3.1

Використання пам’яті	Високе	Низьке	Високе
Підтримка Web API	Повна та стандартизована	Залежить від ОС користувача	Неповний

Аналіз даних таблиці 3.1 показує, що хоча Electron споживає більше системних ресурсів, його стабільність та екосистема роблять його стандартом де-факто для створення складних корпоративних систем обміну повідомленнями.

Водночас Electron виступає лише надійним середовищем, яке об’єднує вебтехнології з операційною системою, але не має власних вбудованих інструментів для малювання елементів вікна.

Тому безпосередньо всередині його процесу рендерингу для створення візуальної частини застосовується бібліотека React. Це декларативна JavaScript-бібліотека з відкритим кодом, створена інженерами компанії Meta для побудови швидких, масштабованих та інтерактивних інтерфейсів користувача. Офіційний логотип фреймворку React представлено нижче (рис. 3.2).

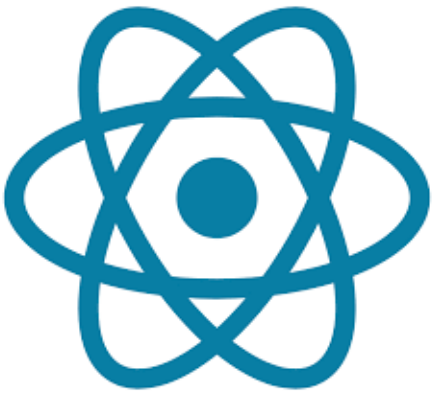

Рисунок 3.2 – Логотип фреймворку React [5]

Вибір React зумовлений його здатністю ефективно управляти складними станами програми, що є критично важливим для комунікаційних додатків з великою кількістю асинхронних подій. Її використання у клієнтській частині пояснюється високою швидкістю оновлення елементів інтерфейсу, зручною

компонентною організацією та можливістю ефективно працювати зі складним станом застосунку.

Для системи обміну повідомленнями це має особливе значення, оскільки чат-застосунок повинен оперативно реагувати на появу нових повідомлень, зміну статусів користувачів та інші події в режимі реального часу.

До основних особливостей React, які визначають його ефективність у розробці клієнтських застосунків, належать такі:

– використання проміжного віртуального представлення інтерфейсу, що дозволяє зменшити кількість прямих звернень до реального DOM;

– побудова інтерфейсу з окремих незалежних компонентів, кожен з яких реалізує певну частину функціоналу, що сприяє повторному використанню коду та значно полегшує модульне тестування;

– використання хуків, що забезпечує гнучке керування локальним станом і життєвим циклом компонента;

– декларативний опис бажаного стану інтерфейсу, при якому бібліотека самостійно виконує його оновлення відповідно до змін даних;

– передача даних виключно в одному напрямку - від батьківських компонентів до дочірніх;

– використання спеціального JSX-синтаксису.

Порівняльний аналіз React з іншими популярними аналогами наведено нижче у таблиці 3.2.

Таблиця 3.2 – Порівняння популярних JS-бібліотек для побудови інтерфейсів

Характеристика	React	Vue.js	Angular
Архітектурний підхід	Бібліотека, орієнтована на UI	Гнучкий frontend framework	Повноцінний frontend framework
Основний синтаксис	JSX	HTML-шаблони + директиви	HTML + TypeScript
Складність інтеграції	Легка	Дуже легка	Висока

Кінець таблиці 3.2

Потік даних	Односпрямований	Переважно двоспрямований	Двоспрямований
Підхід до масштабування	Добре підходить для великих SPA	Зручний для малих і середніх проєктів	Орієнтований на великі корпоративні системи
Поріг входження	Середній	Низький	Високий
Гнучкість вибору бібліотек	Висока	Висока	Нижча, оскільки багато рішень уже вбудовано

Вибір React є критично важливим для чат-застосунку, бо вони вимагають миттєвого оновлення інтерфейсу при надходженні нових текстових даних або зміні статусів користувачів. Усе це генерує величезну кількість мікрооновлень екрану. Завдяки механізму Virtual DOM, React здатний ефективно обробляти сотні мутацій за секунду, забезпечуючи високу плавність роботи користувацького інтерфейсу.

3.3 Серверна частина системи

Проєктування серверної частини для систем обміну повідомленнями вимагає інструментів, здатних витримувати високі навантаження, забезпечувати швидку обробку великої кількості паралельних асинхронних подій та підтримувати постійні з’єднання з тисячами клієнтів одночасно.

Сервер має працювати стабільно та безперебійно, оскільки саме він відповідає за маршрутизацію трафіку, авторизацію та бізнес-логіку. Тому для реалізації програмної логіки було обрано сучасний фреймворк NestJS, який функціонує на базі платформи Node.js і використовує мову програмування TypeScript.

NestJS – це прогресивний фреймворк для створення ефективних, надійних і масштабованих серверних додатків. Його головна особливість полягає в тому, що він привносить сувору архітектуру у світ Node.js [10].

Рисунок 3.3 – Логотип фреймворку NestJS [8]

Архітектурною основою NestJS є глибоке використання патерну впровадження залежностей (Dependency Injection). Цей механізм дозволяє відокремити бізнес-логіку від контролерів маршрутизації, забезпечуючи високий рівень інкапсуляції. Завдяки нативній підтримці TypeScript, типізація на етапі компіляції автоматично нівелює більшість помилок, пов'язаних із несумісністю типів даних (Type Safety), що безпосередньо впливає на надійність обробки інформації користувачів.

Крім того, NestJS використовує асинхронну модель вводу-виводу (Non-blocking I/O), успадковану від Node.js, що робить його ідеальним інструментом для додатків, які залежать від великої кількості мережевих запитів, а не від важких математичних обчислень. Порівняння популярних архітектурних підходів для створення бекенду в середовищі Node.js наведено у таблиці 3.3.

Таблиця 3.3 – Порівняльний аналіз серверних технологій на базі Node.js

Характеристика	NestJS	Express.js	Koa.js
Архітектурний підхід	Модульний	Мінімалістичний	Мінімалістичний
Інтеграція WebSocket	Має вбудований модуль Gateways	Потребує ручного налаштування	Потребує ручного налаштування
Підтримка TypeScript	Нативна	Підтримується через встановлення сторонніх пакетів	Підтримується через встановлення сторонніх пакетів
Складність освоєння	Висока	Низька	Середня

Як видно з даних таблиці 3.3, NestJS має найсильніші позиції завдяки вбудованій підтримці архітектурних патернів. Окремої уваги вимагає механізм доставки даних. Оскільки система обміну повідомленнями вимагає миттєвої доставки даних, класичного протоколу HTTP, який працює за принципом одноразового запиту та відповіді недостатньо. Технологічний стек сервера доповнено протоколом WebSocket, за допомогою бібліотеки Socket.IO [6].

WebSocket встановлює постійне TCP-з’єднання між клієнтом і сервером. Це дозволяє бекенду самостійно ініціювати відправку подій, а саме нових повідомлень, статусів "Online/Offline", у будь-який момент часу без необхідності постійного опитування сервера клієнтом.

Це радикально зменшує мережеве навантаження та усуває затримки, роблячи спілкування дійсно миттєвим. Бібліотека Socket.IO додатково забезпечує механізми автоматичного відновлення з’єднання при втраті мережі та підтримує логіку "кімнат" для ізольованої відправки даних учасникам конкретного чату.

3.4 Організація бази даних

Організація рівня збереження даних є фундаментальним аспектом створення будь-якого комунікаційного застосунку. Зважаючи на специфіку чат-систем, де повідомлення генеруються з величезною швидкістю та можуть мати різноманітну структуру, класичні реляційні бази даних часто створюють «вузькі місця» продуктивності. Саме тому для реалізації проєкту було вирішено використати документо-орієнтовану нереляційну СУБД MongoDB [7].

На відміну від реляційних СУБД, MongoDB належить до класу NoSQL-рішень і зберігає інформацію у гнучкому форматі BSON (Binary JSON). Замість жорстких таблиць і рядків вона оперує колекціями та документами. Такий підхід забезпечує підтримку динамічної схеми даних, завдяки чому кожен документ у колекції може мати власний набір полів, який за потреби легко розширювати без впливу на інші записи.

Рисунок 3.4 – Логотип NoSQL MondgoDB [7]

Використання документо-орієнтованої бази даних MongoDB для розробки чат-застосунку забезпечує низку критично важливих властивостей. Насамперед це гнучкість схеми даних. Подальше розширення функціоналу, наприклад, додавання масиву реакцій чи статусів прочитання до вже існуючих повідомлень, не вимагає глобальних змін структури всієї бази через міграції.

Крім того, завдяки вкладеним структурам, з’являється можливість зберігати пов’язану інформацію безпосередньо всередині одного документа. Це повністю усуває необхідність виконання складних і повільних операцій злиття, які є характерними для класичних SQL-баз, що суттєво пришвидшує завантаження історії листування.

Ще однією вагомою перевагою є нативна підтримка горизонтального масштабування. З огляду на те, що обсяги даних у чатах постійно зростають, MongoDB дозволяє автоматично розподіляти інформацію між кількома фізичними серверами. Це оптимізує швидкість обробки запитів та гарантує високу відмовостійкість системи у разі збоїв.

Для наочного обґрунтування вибору технології збереження даних нижче подано порівняльний аналіз популярних СУБД.

Таблиця 3.4 – Порівняння реляційних та нереляційних СУБД для комунікаційних систем

Критерій оцінки	MongoDB (NoSQL)	PostgreSQL (SQL)	Redis (In- Memory NoSQL)
Модель даних	Документо-орієнтована	Реляційна	Ключ-значення
Структура схеми	Динамічна	Жорстка	Відсутня
Швидкість обробки неструктурованих даних	Дуже висока	Середня	Максимальна
Основне призначення в системі	Основне сховище	Транзакційне сховище	Кешування сесій

Аналіз даних таблиці 3.4 підтверджує доцільність вибору MongoDB як основного сховища даних (Primary Database). На відміну від жорстко структурованої PostgreSQL та орієнтованого на кешування Redis, вона ідеально поєднує високу швидкість обробки неструктурованої інформації з надійністю її довгострокового збереження, що робить її найефективнішим рішенням для систем обміну повідомленнями.

3.5 Розгортання та технологічне забезпечення системи

Процес переходу від локальної розробки до стадії робочої експлуатації вимагає ретельного планування інфраструктури розгортання. У цьому підрозділі розглядається комплекс технологічних рішень та інструментів, які забезпечують контроль версій коду, налаштування серверного середовища та глобальну доступність системи обміну повідомленнями в мережі Інтернет.

3.5.1 Використання хостингу для серверної частини

Однією з ключових проблем при розробці систем обміну повідомленнями в реальному часі є обмеження локального середовища. На етапі розробки серверна частина запускається на локальній машині, що робить протокол WebSocket доступним виключно в межах однієї локальної мережі.

У такій конфігурації клієнти, які знаходяться в інших мережах, не можуть встановити WebSocket-з’єднання з сервером, що унеможливлює повноцінну комунікацію між користувачами.

Для вирішення цієї проблеми та забезпечення глобальної доступності системи виникає необхідність розміщення серверної частини на публічному хмарному хостингу. В якості платформи для розгортання бекенду було обрано сервіс Render [13].

Вибір платформи Render обґрунтовується декількома критично важливими для проєкту факторами, до яких належать:

– нативна підтримка WebSocket, адже на відміну від багатьох безкоштовних платформ, які обривають довгі TCP-з’єднання, Render підтримує постійні з’єднання, що є обов’язковою умовою для коректної роботи Socket.IO;

– наявність безкоштовного тарифу, що дозволяє розгорнути вебсервіс на базі Node.js без фінансових витрат на етапі тестування та захисту дипломного проєкту;

– автоматичне розгортання Continuous Deployment, завдяки якому сервіс самостійно підтягує нові зміни з репозиторію і безперебійно перезапускає сервер;

– вбудована підтримка SSL-сертифікатів, оскільки платформа автоматично генерує та оновлює сертифікати безпеки, що гарантує захищене HTTPS- та WSS-з’єднання між клієнтом і сервером без додаткових налаштувань.

Для порівняння з іншими аналогами було створено порівняльну таблицю (табл. 3.5).

Таблиця 3.5 – Порівняння хмарних платформ для розгортання Node.js серверів

Платформа	Підтримка WebSocket	Безкоштовний тариф (Free Tier)	Тип інфраструктури
Render	Повна підтримка	Присутній (з обмеженням RAM)	Контейнери

Кінець таблиці 3.5

Heroku	Повна підтримка	Відсутній (лише платні)	Контейнери
Vercel	Не підтримується	Присутній	Serverless Functions
AWS EC2	Повна підтримка	Обмежений на 1 рік	Віртуальні машини

Аналіз таблиці 3.6 підтверджує, що Render є найбільш доцільним та економічно вигідним вибором для розміщення серверної частини чат-застосунку.

3.5.2 Налаштування середовища розгортання

Процес налаштування середовища розгортання на платформі Render [13] передбачає конфігурацію низки параметрів, необхідних для безпечної та безперебійної роботи NestJS-сервера. Організація середовища базується на таких кроках:

– налаштування змінних оточення (Environment Variables), за якого конфіденційні дані, як-от секретний ключ для підпису JWT-токенів та URI-рядок підключення до хмарної бази даних MongoDB Atlas, не зберігаються у відкритому коді, а вказуються безпосередньо в панелі управління Render, що гарантує захист від витоку даних;

– конфігурація CORS (Cross-Origin Resource Sharing), яка дозволяє приймати HTTP-запити та WebSocket-підключення від клієнта, оскільки десктопний застосунок та сервер знаходяться на різних доменах;

– визначення команд збірки (Build Commands), де для платформи Render вказуються специфічні інструкції, зокрема npm install для завантаження залежностей та npm run build для компіляції TypeScript-коду у виконуваний JavaScript-код перед запуском.

Після успішного розгортання сервер отримує публічну HTTPS-адресу, до якої десктопний клієнт звертається для авторизації та встановлення глобального WebSocket-з’єднання.

3.5.3 Додаткові інструменти розробки

Для забезпечення ефективного життєвого циклу розробки програмного забезпечення застосовуються спеціалізовані інструменти контролю версій, написання коду та автоматизації процесів.

Фундаментальною складовою організації робочого процесу є використання розподіленої системи контролю версій Git. Використання цієї технології є обов'язковим індустріальним стандартом сучасної інженерії, оскільки воно гарантує цілісність кодової бази протягом усього періоду створення програмного продукту.

Крім базових можливостей, система контролю версій Git [15] дозволяє реалізувати такі важливі сценарії, як створення тегів для релізів, використання git stash для тимчасового збереження незавершених змін та інтерактивне ребейз для очищення історії комітів. Це особливо корисно при підтримці довготривалих проєктів із кількома версіями.

Для наочності основні команди, що використовувалися під час розробки чат-застосунку, зведено в таблицю 3.6.

Таблиця 3.6 – Основні команди Git, застосовані при розробці проєкту

Команда	Призначення
git init	Ініціалізація локального репозиторію
git add .	Додавання всіх файлів (клієнтської та серверної частин) до індексу для наступного коміту
git remote add origin https://github.com/...	Прив'язка локального репозиторію до віддаленого сховища на GitHub
git push origin main	Відправлення змін до віддаленого сховища GitHub
git commit -m ""	Подальші коміти з конкретними описами виправлень або нових функцій

Використання саме цих команд забезпечило чисту історію проєкту та спростило виявлення помилок при інтеграції клієнтської та серверної частин.

У контексті розробки чат-застосунку система Git забезпечує виконання таких ключових завдань:

– локальна фіксація повної історії всіх змін кодової бази, що дозволяє детально відстежувати кожен етап еволюції проєкту;

– створення ізольованих гілок для безпечного написання нового функціоналу без ризику впливу на основну, стабільну версію системи;

– можливість швидкого відновлення попередніх працездатних станів застосунку (Rollback) у випадку виникнення критичних архітектурних помилок.

Віддаленим сховищем та платформою для співпраці виступає хмарний сервіс GitHub [16]. Він виконує роль централізованого хабу для надійного збереження вихідного коду, унеможливлюючи його втрату при локальних збоях обладнання. Крім того, наявність репозиторію на GitHub дозволяє реалізувати базові принципи безперервного розгортання (Continuous Deployment).

Будь-яке злиття нового коду з основною гілкою репозиторію автоматично відслідковується хостингом Render [13], що ініціює збірку та перезапуск серверної частини без ручного втручання розробника.

Основним інструментом для безпосереднього написання програмного коду виступає інтегроване середовище розробки Visual Studio Code. Завдяки своїй модульності та легковажності, це середовище стало стандартом для веб- та десктопної розробки.

Вбудована підтримка мови TypeScript , розширена система автодоповнення та наявність інтегрованого термінала дозволяють зручно виконувати системні команди та керувати файловою структурою. Додатково використання інтегрованих інструментів форматування, розширення Prettier забезпечує дотримання єдиного стилю написання коду впродовж усього процесу розробки.

3.6 Оптимізація продуктивності десктопного чат-застосунку

Під час розробки кросплатформного десктопного чат-застосунку на основі вебтехнологій постає питання продуктивності. Невдала архітектура легко

призводить до надмірного споживання пам’яті, затримок інтерфейсу та нестабільності при великих навантаженнях. Тому в межах проєкту було передбачено низку практичних рішень, спрямованих на оптимізацію як клієнтської, так і серверної частин, а також мережевої взаємодії.

Основні заходи оптимізації, реалізовані в системі:

– пагінація повідомлень – завантаження історії чату порціями по 50 повідомлень із можливістю підвантаження старіших під час прокручування;

– робота з базою даних – використання документо-орієнтованої моделі MongoDB, вибірка даних без зайвих полів і обмеження кількості результатів пошуку;

– розділення глобального стану на окремі сховища – кожен компонент отримує лише ті дані, які йому дійсно потрібні, що зменшує кількість зайвих оновлень інтерфейсу;

– асинхронна обробка всіх мережевих операцій – HTTP-запити, WebSocket-з’єднання та завантаження файлів не блокують головний потік, тому інтерфейс залишається плавним;

– маршрутизація WebSocket-подій через кімнати – повідомлення, індикатор набору тексту та позначки прочитання отримують лише ті клієнти, які перебувають у конкретному чаті;

– ліниве завантаження зображень і кешування – аватари підвантажуються лише за потреби, а браузерне кешування уникає повторних запитів.

Насамперед, у серверному модулі ChatsService реалізовано пагінацію повідомлень за допомогою параметрів page та limit. Метод getChatMessages спочатку рахує загальну кількість повідомлень у чаті, а потім повертає лише потрібний діапазон. Завдяки цьому клієнт може поступово підвантажувати старіші повідомлення під час прокручування вікна.

Щодо бази даних MongoDB, то тут застосовано кілька простих, але ефективних прийомів. У всіх запитах до колекції користувачів свідомо виключається поле пароля, щоб не передавати зайву службову інформацію.

Документо-орієнтована модель даних дозволяє отримувати всю інформацію про чат одним запитом без операцій об’єднання, як у реляційних базах.

Крім того, в пошуку контактів обмежено кількість результатів двадцятьма записами, що запобігає поверненню надто великої вибірки при неповному або нечіткому запиті.

На клієнтській стороні для керування станом використано бібліотеку Zustand. Глобальний стан розділено на три незалежні сховища – дані авторизації, чати та повідомлення та контакти, онлайн-статуси. Кожен компонент підписується лише на ті дані, які йому реально потрібні. Наприклад, список чатів оновлюється тільки при зміні масиву chats, а вікно листування – при появі нового повідомлення в активному чаті.

Така селективна підписка дозволяє уникнути зайвих рендерів і зберігає інтерфейс чуйним. Додатково завдяки middleware persist дані авторизації автоматично зберігаються в localStorage, що зменшує кількість початкових HTTP-запитів після перезапуску програми.

Усі мережеві операції – відправлення повідомлень, завантаження аватарів, ініціалізація сокета, виконуються асинхронно з використанням async/await. Під час тривалих операцій показується простий індикатор завантаження, що не дратує, але дає зрозуміти, що система працює.

Велику роль у швидкодії реального часу відіграє правильне використання WebSocket. У серверному шлюзі ChatGateway кожен клієнт автоматично додається до «кімнат», що відповідають ідентифікаторам чатів, учасником яких він є. Завдяки цьому повідомлення, події про набір тексту та позначки прочитання надсилаються виключно тим користувачам, які перебувають у конкретному чаті. Це різко скорочує непотрібний мережевий трафік. Крім того, самі події містять лише мінімально необхідний набір полів, без зайвих вкладених об’єктів.

Наостанок, для зображень застосовано ліниве завантаження та покладання на стандартне браузерне кешування. Адреси аватарів формуються окремою утилітою, яка враховує базовий URL сервера. Браузер сам вирішує, коли запитати картинку

повторно, а коли взяти з кешу. При цьому на клієнті не створюються додаткові копії зображень у пам’яті – все відображається через звичайні теги <img>. Це дає змогу знизити споживання оперативної пам’яті та зробити роботу застосунку більш стабільною.

Отже, комплекс перелічених заходів – пагінація, оптимізована робота з БД, розділений стан, асинхронність, кімнати в WebSocket та ліниве завантаження зображень – забезпечує стабільну роботу програми, плавність інтерфейсу та комфортну роботу навіть при інтенсивному листуванні.

У майбутньому продуктивність можна буде додатково підвищити за допомогою індексів, наприклад в MongoDB, за полями chatId та createdAt, мемоїзації важких React-компонентів та стиснення WebSocket-трафіку, однак на поточному етапі реалізованих рішень цілком достатньо для ефективного використання кінцевими користувачами.

Висновки до розділу 3

У третьому розділі було виконано комплексне проєктування архітектури десктопного чат-застосунку та сформовано його технологічну базу. У процесі роботи розроблено багаторівневу клієнт-серверну модель, яка чітко розмежовує зони відповідальності між графічним інтерфейсом, серверною логікою та сховищем даних.

Для реалізації клієнтської частини обрано зв’язку фреймворків Electron та React , що забезпечило кросплатформність та швидкість рендерингу візуальних компонентів. Серверну інфраструктуру побудовано на базі NestJS у поєднанні з нереляційною базою даних MongoDB [7], а для забезпечення двосторонньої комунікації в режимі реального часу інтегровано протокол WebSocket [17]. Додатково було спроєктовано середовище хмарного розгортання на платформі Render [13] та організовано процес контролю версій і безперервної інтеграції за допомогою Git та GitHub [16].

У результаті проведеної роботи вдалося створити надійний технологічний фундамент, який задовольняє всі сучасні вимоги до систем обміну повідомленнями. Завдяки розробленим рішенням з оптимізації продуктивності, таким як курсорна пагінація повідомлень, розділення глобального стану на окремі сховища і використання WebSocket-кімнат, досягнуто високої швидкодії застосунку та суттєво знижено споживання системних ресурсів.

Побудована архітектура гарантує безперебійну синхронізацію інформації, безпечне зберігання даних користувачів та здатність системи ефективно масштабуватися. Загалом, сформована інфраструктура повністю вирішує поставлені задачі для розробки програми і створює оптимальні умови для успішного переходу до фінального етапу – безпосередньої програмної реалізації проєкту.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ЧАТ-ЗАСТОСУНКУ

4.1 Опис структури клієнтської частини

Структура frontend частини

Клієнтська частина розробилялась як SPA, де маршрутизація, управління станом та обробка бізнес-логіки виконуються локально, мінімізуючи зайві звернення до сервера. Файлова структура проєкту суворо розділена за функціональним призначенням. Частина файлової структури директорія frontend/ містить такі ключові архітектурні вузли:

- src/components/– директорія для найбільш часто використовуваних логічних модулів , такі як чати, контакти, макети тощо;
- src/pages/– контейнерні компоненти, які відповідають за життєвий цикл окремих маршрутів застосунку;
- src/store/– глобальний стан застосунку;
- src/utils/ – інкапсульовані утиліти для налаштування HTTP-запитів та WebSocket-комунікації;
- src/config/конфігураційні файли, зокрема env.ts, де зберігаються змінні середовища для динамічного вибору сервера;
- electron/– системний код для роботи десктопної оболонки;
- index.html та main.tsx точки входу для запуску React-додатку.

У клієнтській частині логіка роботи з даними відділена від компонентів відображення за допомогою системи глобального стану Zustand. Стан розділено на три незалежні модулі. authStore.ts відповідає за збереження JWT-токена та даних сесії користувача, usersStore.ts кешує словники мережевих статусів контактів, а chatStore.ts інкапсулює логіку управління повідомленнями, зберігаючи дані у форматі хеш-таблиць.

Для виконання HTTP-запитів до REST API застосовується бібліотека Axios із налаштованими інтерцепторами. Ці інтерцептори автоматично додають токен авторизації до заголовків усіх запитів.

Процес створення десктопної оболонки керується файлом main.js. Під час запуску він ініціалізує системний процес і підвантажує скомпільований код React. З метою безпеки архітектури прямий доступ фронтенду до системних API Node.js заблоковано.

main.js

frontend > electron > main.js > waitForVite

```
function createWindow() {
  try {
    mainWindow = new BrowserWindow({
      width: 1200,
      height: 780,
      minWidth: 900,
      minHeight: 600,
      frame: false,
      titleBarStyle: 'hidden',
      show: false,
      webPreferences: {
        preload: path.join(__dirname, 'preload.js'),
        nodeIntegration: false,
        contextIsolation: true,
        sandbox: true,
      },
      backgroundColor: '#1c1b19',
    });

    const isDev = process.argv.includes('--dev');

    if (isDev) {
      mainWindow.loadURL('http://localhost:5173');
      setTimeout(() => {
        mainWindow.webContents.openDevTools();
      }, 1000);
    } else {
      mainWindow.loadFile(path.join(__dirname, '../dist/index.html'));
    }

    mainWindow.webContents.on('did-finish-load', () => {
      mainWindow.show();
    });

    mainWindow.webContents.on('did-fail-load', (event, errorCode, errorDescrip
      console.error('Load error:', errorCode, errorDescription);
    });

    mainWindow.webContents.on('crashed', () => {
      console.error('WebContents crashed');
    });
```

Рисунок 4.1 – Конфігурація головного процесу Electron

Наведена конфігурація забезпечує повну ізоляцію React-коду від системних API Node.js, що підвищує безпеку застосунку. Завдяки механізму IPC та об’єкту window.electronAPI віконні операції виконуються без прямого доступу до десктопних ресурсів.

Реалізовані можливості Electron у десктопному чат-застосунку

У розробленому застосунку Electron використано не лише як обгортку для веб-інтерфейсу, а й для забезпечення глибокої інтеграції з операційною системою.

Завдяки двопроцесній архітектурі вдалося розмежувати відповідальність, де логіка інтерфейсу виконується в ізольованому середовищі, а системні виклики проходять через безпечний міст IPC.

Усі комунікації між React-застосунком та Electron [4] відбуваються через об’єкт window.electronAPI, який експонується в прелоад-скрипті. На основі файлів main.js та preload.js було реалізовано низку нативних функцій, до яких належать:

– керування вікном за допомогою методів minimize(), maximize() та close() через IPC-канали, завдяки чому користувач може керувати вікном так само, як будь-якою іншою десктопною програмою;

– робота із системним треєм, що передбачає відображення іконки після запуску та контекстного меню з пунктами «Показати вікно» і «Вийти», причому під час спроби закрити вікно програма не завершується, а ховається в трей, що є нормальною практикою для більшості сучасних месенджерів, а клік на іконку показує або приховує застосунок;

– генерація нативних системних сповіщень через виклик та створення нового екземпляра Notification у головному процесі, при цьому в компоненті App.tsx цей механізм використовується під час отримання нового повідомлення, якщо вікно не у фокусі згідно з перевіркою isWindowFocused();

– виклик нативного контекстного меню для полів, що блокує браузерне контекстне меню для елементів input та textarea і замінює його на системне з пунктами «Копіювати», «Вставити», «Вирізати» й «Вибрати все», підвищуючи тим самим зручність роботи з текстом;

– забезпечення автономності від браузера завдяки вбудованому рушію Chromium та ізоляції від браузерних розширень, де високий рівень безпеки

досягається шляхом використання параметрів nodeIntegration: false, contextIsolation: true та sandbox: true;

– обробка життєвого циклу застосунку, у межах якої реалізовано коректне завершення роботи за допомогою прапора isQuitting, а також обробку подій before-quit, window-all-closed та activate для macOS, що гарантує відсутність «висячих» процесів після закриття програми.

Таким чином, завдяки Electron вдалося створити повноцінний десктопний месенджер, який поводиться як нативний застосунок, підтримує трей, сповіщення, керування вікном та контекстне меню.

Усі перелічені функції працюють без змін на популярних операційних системах Windows, Linux та macOS, що підтверджує кросплатформність рішення.

Реалізована архітектура з ізольованим прелоад-скриптом гарантує безпеку - веб-частина не має прямого доступу до Node.js [10], а всі системні виклики чітко контролюються. Це робить програму зручною для щоденного використання без потреби тримати браузер відкритим, а також дозволяє в майбутньому легко додавати нові нативні функції (наприклад, гарячі клавіші або автозапуск)

WebSocket комунікація у клієнтській частині

Критично важливою складовою клієнтської архітектури є програмна реалізація синхронізації даних у режимі реального часу. За ініціалізацію та підтримку безперебійного обміну даними відповідає файл utils/socket.ts.

Основна логіка інкапсульована у функції initSocket, яка приймає JWT-токен і встановлює стабільне двостороннє з’єднання з сервером через бібліотеку socket.io-client. У конфігурації підключення передбачено автоматичне відновлення зв’язку, у випадку збоїв мережі клієнт самостійно здійснює реконект із динамічною затримкою.

socket.ts M

frontend > src > utils > socket.ts > getSocket

```ts
import { io, Socket } from 'socket.io-client';

let socket: Socket | null = null;

export const initSocket = (token: string): Socket => {
  let socketUrl: string;

  const viteSocketUrl = import.meta.env.VITE_SOCKET_URL;
  if (viteSocketUrl) {
    socketUrl = viteSocketUrl;
  } else {
    const hostname = window.location.hostname;
    const protocol = window.location.protocol === 'https:' ? 'https' : 'http';
    socketUrl = `${protocol}://${hostname}:3001`;
  }

  console.log('Connecting to socket:', socketUrl);

  socket = io(socketUrl, {
    auth: { token },
    transports: ['websocket', 'polling'],
    reconnection: true,
    reconnectionDelay: 1000,
    reconnectionDelayMax: 5000,
    reconnectionAttempts: 5,
    forceNew: false,
  });

  socket.on('connect_error', (error) => {
    console.error('Socket connection error:', error);
  });

  socket.on('disconnect', (reason) => {
    console.warn('Socket disconnected:', reason);
  });

  return socket;
};
```

Рисунок 4.2 – Фрагмент коду ініціалізації WebSocket-клієнта

Після встановлення WebSocket-з’єднання клієнтський застосунок ініціалізує глобальні слухачі подій, які безпосередньо мутують дані у Zustand-сховищах. Під час надходження події message:new програма викликає метод додавання даних у chatStore.messages, що ініціює перенаправлення компонента чату.

Синхронізація видаленої інформації обробляється через події message:deleted та chat:deleted, які автоматично очищують відповідні записи у словниках стану.

4.2 Реалізація серверної частини та маршрутизації даних

Серверна частина реалізована на фреймворку NestJS [8] з використанням TypeScript [9]. NestJS обрано завдяки модульній архітектурі, вбудованій підтримці WebSocket та зручній інтеграції з MongoDB через Mongoose [11]. Файлова

структура проєкту суворо розділена за функціональним призначенням. Коренева директорія backend/ містить такі ключові архітектурні вузли:

- src/ – основний каталог з вихідним кодом;
- src/auth/ – модуль автентифікації містить контролери, сервіси, стратегія JWT;
- src/users/ – модуль роботи з користувачами, контролер, сервіс, схема User;
- src/chats/ – модуль управління чатами, контролер, сервіс, схема Chat;
- src/messages/ – модуль для завантаження файлів та схема Message;
- src/gateway/ – WebSocket шлюз для real-time комунікації;
- src/config/ – змінні середовища та конфігурації;
- main.ts – точка входу, де створюється NestJS застосунок, налаштовуються CORS, глобальний префікс API та статична роздача файлів;
- app.module.ts – кореневий модуль, який імпортує всі інші модулі та підключає Mongoose до MongoDB.

Кожен модуль слідує принципу розділення відповідальності. Наприклад, auth.module.ts імпортує JwtModule та PassportModule, реєструє контролер, сервіс і стратегію. chats.module.ts імпортує схеми Chat, Message, User і надає ChatsService. Така організація дозволяє легко тестувати окремі компоненти та масштабувати проєкт.

4.2.1 Програмна реалізація процесів авентифікації, авторизації та обробки даних

У застосунку реалізовано дворівневий механізм контролю доступу. Перший рівень – виконана на основі JWT. Другий рівень – авторизація, реалізована за допомогою гвардів, які аналізують роль або статус користувача. Передача даних між клієнтом і сервером виконується за класичною схемою «запит – обробка – відповідь» через REST API.

Маршрутизація та потік даних під час реєстрації

Клієнт надсилає POST-запит на ендпоінт /api/auth/register із тілом, що містить username, email та password. Запит приймає AuthController, який викликає метод register сервісу AuthService. Сервіс, використовуючи інжектовану модель User, перевіряє унікальність email та username в колекції MongoDB.

У разі успіху пароль хешується bcrypt, створюється новий документ користувача, а метод JwtService.sign генерує JWT з ідентифікатором користувача. Токен разом із даними профілю повертається клієнту. Цей токен зберігається на фронтенді, додається до заголовка Authorization: Bearer, при кожному наступному запиті.

Рисунок 4.3 – Фрагмент коду auth.service.ts з методами register та login

Процес авторизації включає у себе такі етапи:

– отримання даних форми від клієнта;
– перевірка наявності користувача з такою поштою чи ім’ям у бд;
– хешування пароля за допомогою bcrypt ;
– збереження нового об’єкта user у mongodb;

– створення jwt-токена, який містить ідентифікатор користувача та його ім’я» або ще простіше: «генерація токена з даними користувача;

– повернення токена та публічних даних користувача - id, username, email, avatar.

Процес логіну відрізняється від реєстрації. Клієнт надсилає POST-запит на /api/auth/login з email та password. AuthController делегує виклик методу login сервісу. Сервіс шукає користувача за email, а потім порівнює переданий пароль з хешем за допомогою bcrypt.compare.

При збігу генерується новий JWT-токен і повертається клієнту. У разі невідповідності викидається виняток UnauthorizedException з повідомленням «Invalid credentials». Далі фронтенд зберігає токен і переходить до захищених маршрутів.

Після отримання JWT-токена клієнт додає його до заголовка Authorization при кожному запиті. На сервері процес перевірки токена складається з наступних етапів:

– jwtauthguard активує стратегію jwtstrategy;

– стратегія перевіряє підпис і термін дії токена за допомогою секретного ключа, якщо токен невалідний – повертається помилка 401 unauthorized;

– з токена вилучається поле sub, яке передається в метод validate;

– authservice.validateuser() отримує з mongodb об’єкт користувача і підставляє його в req.user, роблячи доступним у контролерах.

Рисунок 4.4 – Фрагмент коду jwt.strategy.ts

Дана стратегія є типовим прикладом реалізації механізму JWT у NestJS. Вона не лише перевіряє цілісність токена, але й забезпечує плавну інтеграцію з системою авторизації застосунку.

Завдяки такому підходу код контролерів залишається чистим, оскільки вся логіка валідації винесена в окремий спеціалізований клас.

4.2.2 Логіка управління протоколом WebSocket

Для реалізації двостороннього зв’язку в реальному часі в застосунку використовується бібліотека Socket.IO [6], інтегрована з фреймворком NestJS. Основний модуль, що відповідає за цю функціональність, знаходиться у файлі chat.gateway.ts.

Він містить клас ChatGateway, який за допомогою декоратора @WebSocketGateway оголошується як WebSocket-шлюз. Цей шлюз виконує кілька критично важливих завдань - аутентифікацію клієнтів за допомогою JWT-токена, управління кімнатами для адресної доставки повідомлень, обробку вхідних подій таких як надсилання повідомлень, видалення, позначка прочитання, індикатор

набору тексту, а також оновлення статусу «онлайн/остання активність» у базі даних.

Клас ChatGateway реалізує інтерфейси OnGatewayConnection та OnGatewayDisconnect, завдяки чому отримує можливість виконувати логіку при підключенні та відключенні кожного клієнта. У конструкторі через механізм Dependency Injection отримуються JwtService для роботи з токенами та Mongoose-моделі Message, Chat і User для доступу до бази даних.

Ключовим моментом є використання механізму кімнат Socket.IO, де клієнт успішно підключається, сервер знаходить усі чати, учасником яких є користувач, і для кожного такого чату викликає client.join(chatId). Це дозволяє надалі надсилати події лише тим клієнтам, які перебувають у відповідній кімнаті, що значно зменшує мережевий трафік і підвищує продуктивність.

Основним методом обробки повідомлень є handleMessage, який реагує на подію message:send. Він перевіряє членство користувача в чаті, створює запис у колекції Message, оновлює lastMessage чату та розсилає отримане повідомлення всім учасникам за допомогою this.server.to(chatId).emit.

Також шлюз обробляє події ретрансляції індикатора набору тексту, позначеннь повідомлень видаленими, позначення повідомлень прочитаними та додаванням нового чату для учасників.

Завдяки інтеграції з JWT кожне підключення проходить безпечну автентифікацію, а статуси онлайн/офлайн оновлюються автоматично.

```
TS chat.gateway.ts M, M
backend > src > gateway > TS chat.gateway.ts > ChatGateway
15  export class ChatGateway implements OnGatewayConnection, OnGatewayDisconnect {
27    async handleConnection(client: Socket) {
28      try {
29        const token = client.handshake.auth?.token || client.handshake.headers?.authorization?.split(' ')[1];
30        if (!token) {
31          this.logger.warn('Connection attempt without token');
32          client.disconnect();
33          return;
34        }
35        const payload = this.jwtService.verify(token);
36        client.data.userId = payload.sub;
37        this.userSockets.set(payload.sub, client.id);
38        await this.userModel.findByIdAndUpdate(payload.sub, { isOnline: true });
39        this.server.emit('user:online', { userId: payload.sub, isOnline: true });
40        const chats = await this.chatModel.find({ members: payload.sub });
41        chats.forEach(chat => client.join(chat._id.toString()));
42        this.logger.debug(`User ${payload.sub} connected. Total clients: ${this.server.engine.clientsCount}`);
43      } catch (error) {
44        this.logger.error('Connection error:', error);
45        client.disconnect();
46      }
47    }
48
49    async handleDisconnect(client: Socket) {
50      try {
51        const userId = client.data.userId;
52        if (userId) {
53          this.userSockets.delete(userId);
54          await this.userModel.findByIdAndUpdate(userId, { isOnline: false, lastSeen: new Date() });
55          this.server.emit('user:online', { userId, isOnline: false, lastSeen: new Date() });
56          this.logger.debug(`User ${userId} disconnected`);
57        }
58      } catch (error) {
59        this.logger.error('Disconnect error:', error);
60      }
61    }
```

Рисунок 4.5 – Фрагмент файлу chat.gateway.ts

Після наведеного рисунка (рис. 4.5) варто зазначити, що метод handleConnection є критичним для безпеки функціонування WebSocket-підключень. Саме він виконує автентифікацію клієнта та автоматично додає його до кімнат усіх доступних чатів. Такий підхід дозволяє серверу миттєво сповіщати всіх учасників про зміни статусу та доставляти повідомлення без зайвих HTTP-запитів, що є основою роботи реального часу.

Окрім базових подій надсилання та отримання повідомлень, у системі реалізовано низку допоміжних WebSocket-подій, які значно підвищують зручність користування застосунком. Насамперед це індикатор набору тексту. Коли користувач починає вводити повідомлення, клієнт надсилає подію typing:start з ідентифікатором чату, а сервер транслює її всім іншим учасникам бесіди, які бачать спливаюче повідомлення «хтось друкує…».

При зупинці введення або надсиланні повідомлення клієнт ініціює typing:stop, і індикатор зникає. Для запобігання надмірним мережевим запитам на клієнті реалізовано таймаут, тобто повторна подія typing:start надсилається не частіше ніж раз на дві секунди.

Ще однією важливою можливістю є оновлення імені користувача в реальному часі. Коли користувач змінює своє ім’я в налаштуваннях профілю, сервер не лише оновлює відповідний запис у базі даних, але й знаходить усі чати, учасником яких є цей користувач.

Для кожного такого чату через WebSocket розсилається подія user:nameChanged, яка містить нове ім’я та ідентифікатор користувача. Клієнти, які перебувають у цих чатах, миттєво оновлюють відображення імені у списку чатів та в заголовку вікна листування, що уникає потреби перезавантажувати застосунок.

Також у системі реалізовано розширену модель прав на видалення повідомлень та так зване «м’яке видалення». Звичайний користувач може видалити лише власні повідомлення, тоді як адміністратор групового чату має право видаляти будь-яке повідомлення в групі. Процес видалення ініціюється подією message:delete, після чого сервер перевіряє права, а потім застосовує «м’яке видалення».

Вміст повідомлення замінюється на текст «Повідомлення видалено», а файли та посилання на них очищуються. Сам документ у базі даних залишається, але отримує позначку isDeleted: true. Це дозволяє зберегти історію чату для адміністрування, однак приховує зміст від звичайних користувачів.

Після успішного видалення сервер розсилає подію message:deleted усім учасникам чату. Якщо видалене повідомлення було останнім у цьому чаті, клієнт автоматично підтягує попереднє повідомлення, щоб коректно відображати останню активність.

4.3 Опис дизайну чат-застосунку

Інтерфейс чат-застосунку виконаний у сучасному мінімалістичному стилі з підтримкою темної теми. Головне вікно поділяється на три основні області – бічна панель навігації з іконками таки розділів як чати, контакти, пошук, налаштування.

Сторінка пошуку людей (рис. 4.5) текстове поле з для введення пошукового запиту. Після введення мінімум двох символів система відображає знайдені

облікові записи, для кожного результату показуються аватар, ім’я користувача, електронна пошта та кнопка «Додати».

Якщо користувач уже є в списку контактів, кнопка змінює стан на «Додано» та стає неактивною. У нижній частині сторінки розміщено підказку про необхідність переходу до розділу чатів для початку спілкування.

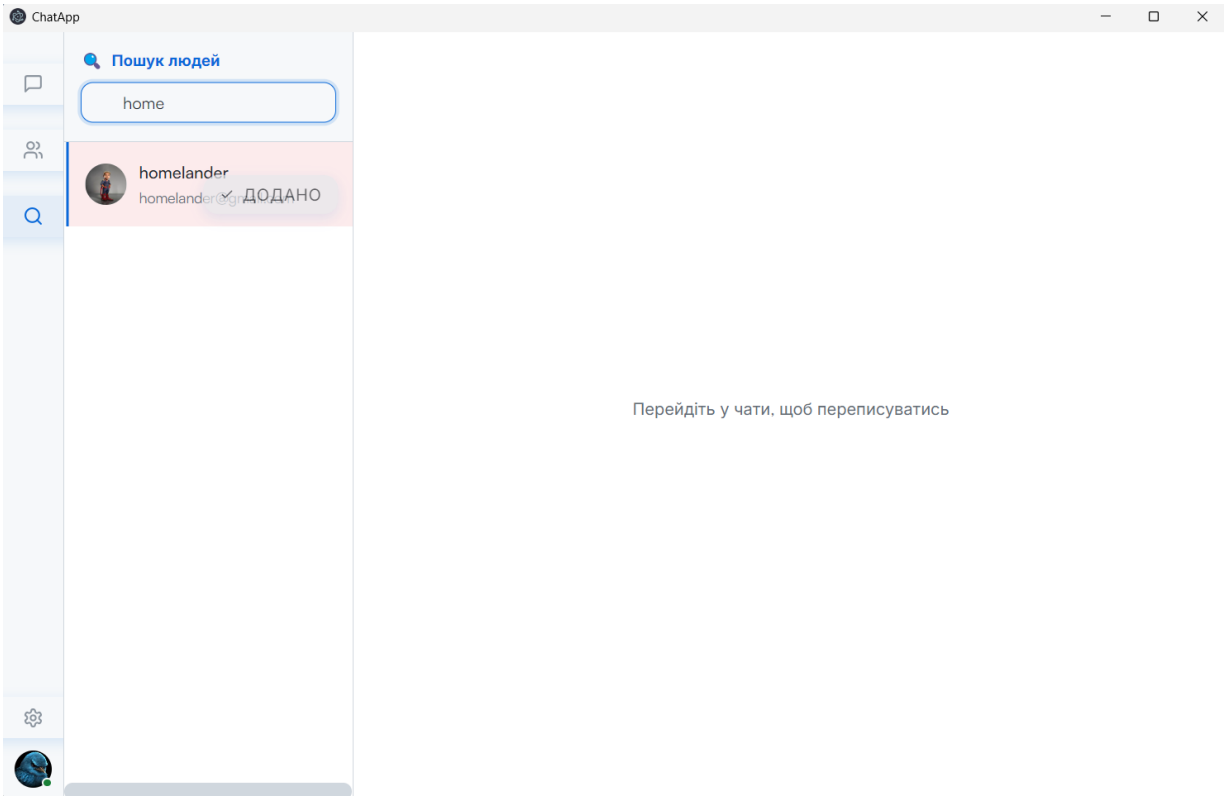

Рисунок 4.6 – Сторінка пошуку друзів

Модальне вікно створення групи (рис. 4.6) надає можливість ввести назву групи та обрати учасників зі списку наявних контактів. Кожен контакт супроводжується аватаром, іменем та електронною поштою.

Візуальне виділення обраних учасників, а також відображення їхньої кількості які є у контактах, в заголовку секції та на кнопці створення забезпечує зворотний зв’язок. Кнопка для створення групи активується лише після того, як обрано хоча б одного учасника та введено назву групи, що запобігає створенню неповних або порожніх груп.

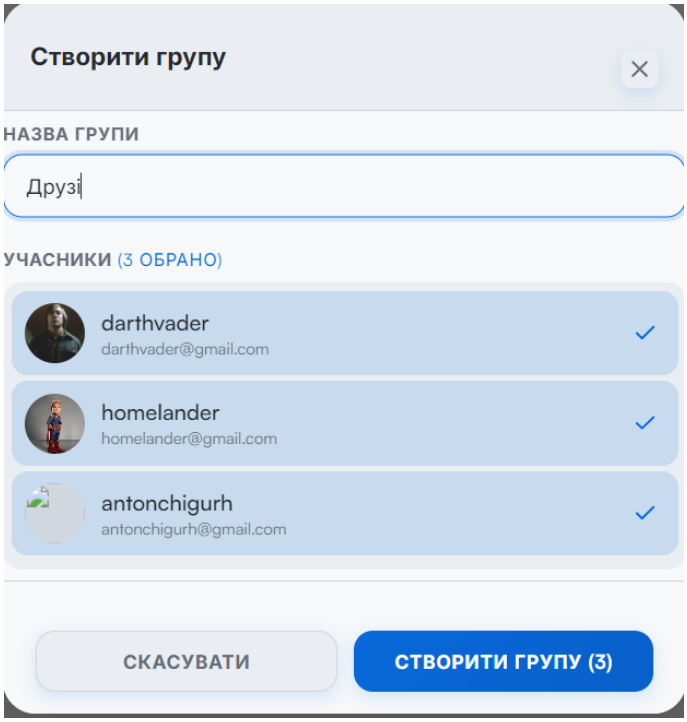

Рисунок 4.7 – Вікно створення групи

Головний екран чату (рис. 4.7) складається з двох колонок. Ліва колонка містить список усіх чатів користувача - групових та приватних. Для кожного чату відображається аватар, ім’я співрозмовника або назва групи, останнє повідомлення та час його надсилання. Непрочитані повідомлення виділяються червоним колом з кількістю непрочитанних повідомлень у ньому.

Права колонка відображає активний чат, повідомлення інших учасників вирівняні ліворуч і мають світло-сірий фон, власні повідомлення вирівняні праворуч і мають градієнтний синій фон. Така організація інтерфейсу забезпечує інтуїтивно зрозумілу навігацію та швидкий доступ до функцій спілкування.

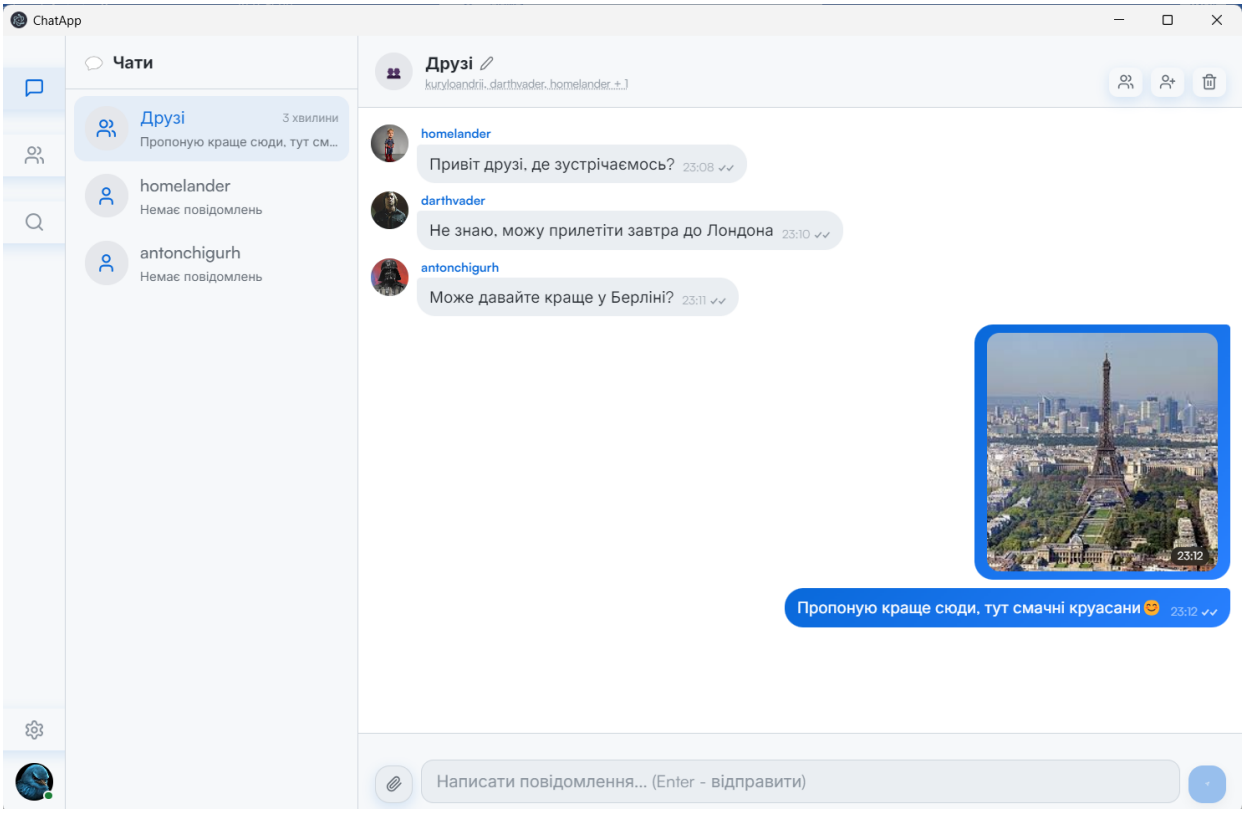

Рисунок 4.8 – Головний екран чату

Отже, розроблений інтерфейс чат-застосунку поєднує сучасний мінімалістичний дизайн з інтуїтивно зрозумілою навігацією. Темна тема, зручне компонування лівої панелі чатів та правої області листування, а також візуальне виділення власних та чужих повідомлень роблять використання програми комфортним навіть для нових користувачів.

Інтерфейс застосунку підтримує два візуальні режими – темний і світлий. Уся кольорова гама винесена в CSS-змінні у файлі globals.css. При перемиканні теми в компоненті SettingsPage змінюється атрибут data-theme на кореневому елементі html, що автоматично оновлює значення всіх змінних. Обрана тема зберігається в localStorage і відновлюється при наступному запуску програми в main.tsx.

Такий підхід унеможливлює мерехтіння під час завантаження, оскільки тема застосовується до рендеру React-компонентів.

У темному режимі використано приглушені кольори з низьким контрастом для фон та яскраві акценти, що зменшує навантаження на зір при тривалій роботі вночі. Світлий режим наслідує стиль сучасних месенджерів із білими картками та

м’якими тінями. Користувач може перемикати тему в будь-який момент без перезавантаження застосунку, що підвищує загальну зручність.

Додаткові елементи, такі як індикатор набору тексту та контекстне меню, суттєво розширюють функціональність без перевантаження екрана. Таким чином, результати цього розділу свідчать про вдалу реалізацію клієнтської частини, яка повністю відповідає сучасним вимогам до UX/UI комунікаційних додатків.

4.4 Тестування працездатності чат-застосунку

Для підтвердження коректності роботи всіх основних функціональних модулів було проведено функціональне тестування. У ході тестування перевірялися сценарії реєстрації нового користувача, надсилання повідомлень у реальному часі, створення групових чатів, а також завантаження зображень через drag & drop. Кожен тест-кейс фіксував очікувану поведінку системи при правильних діях та реакцію на типові помилки, наприклад порожні поля чи перевищення розміру файлу. Результати перевірки зведено в таблиці 4.1 – 4.4.

Таблиця 4.1 – Реєстрація нового користувача

Діючі особи	Неавторизований користувач, система
Мета	Створити обліковий запис для доступу до головного екрану чат-застосунку
Передумова	Користувач відкрив застосунок, бачить сторінку AuthPage з формою реєстрації.
Успішний сценарій: – користувач вводить електронну пошту, ім’я та пароль. – користувач натискає кнопку «sign up»; – система надсилає запит post /auth/register на сервер; – сервер перевіряє унікальність email та валідність полів, створює нового користувача; – сервер повертає об’єкт { user, token }; – система зберігає дані в localstorage (zustand persist) та виконує автоматичний вхід; – користувач перенаправляється на головний екран зі списком чатів.	

Кінець таблиці 4.1

Помилкові дії 1a. Користувач залишив порожнім одне з полів, ввів пароль коротше шести символів або ім’я коротше трьох символів.
Результат: сервер повертає помилку 400, система відображає червоне повідомлення про помилку у формі, реєстрація не виконується.

Таблиця 4.2 – Надсилання текстового повідомлення в реальному часі

Діючі особи	Відправник (А), отримувач (Б), система
Мета	Перевірити миттєву доставку повідомлення через WebSocket-з’єднання
Передумова	Обидва користувачі авторизовані, перебувають у спільному приватному чаті, ChatWindow активний, сокет-з’єднання встановлено (функція initSocket)
Успішний сценарій: – користувач А вводить текст у поле для вводу; – користувач А натискає клавішу enter або кнопку «send»; – спрацьовує обробник handlesend, який викликає socket.emit; – сервер отримує подію, зберігає повідомлення в базі даних, додає _id та createdat; – сервер транслює подію message:new усім учасникам чату; – на клієнтах А та Б викликається addmessage(msg) та updatechatlastmessage(...); – повідомлення відображається у вікні чату, у користувача А – праворуч, у користувача Б – ліворуч; – стрічка повідомлень автоматично прокручується вниз.	
Помилкові дії 1a. Користувач А намагається надіслати повідомлення, що складається лише з пробілів. 1b. Під час надсилання відбувається розрив WebSocket-з’єднання (відсутній інтернет).	
Результат: socket.emit не може відправити повідомлення, у консолі фіксується помилка.	

Таблиця 4.3 – Надсилання текстового повідомлення в реальному часі

Діючі особи	Авторизований користувач, система
Мета	Переконатися, що група створюється коректно, учасники отримують сповіщення, а чат з’являється у списку всіх запрошених

Кінець таблиці 4.3

Передумова	Користувач має список контактів мінімальна кількість котрих, не менше двох осіб, перебуває на головному екрані, WebSocket-з'єднання активне
Успішний сценарій: – користувач натискає кнопку «Створити групу»; – система відображає модальне вікно з полем «Назва групи» та списком контактів; – організатор вводить назву, обирає двох контактів та натискає «Створити»; – клієнт надсилає POST-запит /api/chats/group із масивом memberIds; – сервер перевіряє права, створює документ Chat і додає організатора як adminId; – сервер через WebSocket надсилає інформацію про новий чат усім обраним учасникам; – у кожного учасника в лівій панелі миттєво з'являється нова група; – організатор перенаправляється у вікно щойно створеної групи.	
Помилкові дії 1а. Організатор не ввів назву групи або не вибрав жодного учасника	
Результат: кнопка «Створити» неактивна, система показує підказку «Заповніть усі поля»	

Таблиця 4.4 – Тестування відправлення зображення

Діючі особи	Зареєстрований користувач (відправник), система, отримувач
Мета	Перевірити можливість надсилання зображення шляхом перетягування файлу у вікно повідомлень, коректне відображення мініатюри та сповіщення інших учасників чату.
Передумова	Користувач авторизований, активний чат (приватний або груповий) відкрито, WebSocket-з'єднання встановлено. Файл-зображення розміром до 10 МБ знаходиться у файловій системі.
Успішний сценарій: – користувач відкриває папку з файлом; – перетягує файл мишею у центральну область вікна чату ; – система візуально підсвічує зону прийому; – після відпускання миші файл завантажується у робочу область; – файл надсилається на сервер через post /messages/upload; –	

Кінець таблиці 4.4

– сервер зберігає файл у директорії uploads/messages/ та повертає json із messageurl, filename, filetype; – клієнт викликає socket.emit з отриманим url; – сервер створює запис у колекції message з полем fileurl, а text залишає порожнім; – подія message:new транслюється всім учасникам чату; – у вікні чату замість тексту відображається мініатюра зображення ; – при натисканні на мініатюру відкривається модальне вікно imageviewer з можливістю масштабування та закриття по escape; – учасник чату отримує системне сповіщення (якщо вікно не в фокусі) з текстом «зображення».
Помилкові дії 1a. Користувач перетягує файл розміром більше 10 МБ. 1c. Під час завантаження зникає інтернет-з'єднання.
Результат: – система не починає завантаження, показує спливаюче повідомлення «Файл не може перевищувати 10 МБ» ; – Axios запит переривається помилкою, індикатор зникає, поле вводу розблоковується. Користувач отримує повідомлення «Не вдалося завантажити файл».

Результати тестування, наведені в таблицях 4.1 – 4.4, підтверджують повну працездатність основних функціональних модулів чат-застосунку. Система коректно обробляє реєстрацію та авторизацію користувачів, забезпечує миттєву доставку повідомлень через WebSocket-з'єднання, а також адекватно реагує на типові помилки введення або збої мережі.

Створення групових чатів відбувається без затримок, а сповіщення про нову групу миттєво отримують усі запрошені учасники. Додатково перевірено механізм завантаження зображень через drag & drop – він працює стабільно на всіх підтримуваних операційних системах, а модальний переглядач забезпечує зручне масштабування.

Успішне виконання всіх тестових сценаріїв свідчить про високий рівень надійності розробленого програмного забезпечення. Виявлені недоліки, зокрема

відсутність локального кешування невідправлених повідомлень при розриві зв’язку, не є критичними та можуть бути усунені в наступних версіях продукту. Загалом, за результатами тестування, застосунок готовий до впровадження та експлуатації в реальних умовах.

Висновки до розділу 4

У даному розділі було розглянуто практичну реалізацію клієнтської частини чат-застосунку, її дизайн та тестування. На основі поставлених вимог у попередніх розділах, розроблено інтерфейс користувача, який складається з бічної панелі навігації, списку чатів, вікна переписки, сторінок пошуку контактів, налаштувань та авторизації. Застосунок підтримує темну тему, адаптивне компонування, сучасний мінімалістичний дизайн із заокругленими кутами, градієнтними кнопками та чіткою візуальною ієрархією.

Реалізовано основні функціональні модулі – реєстрацію та вхід користувачів, створення приватних та групових чатів, надсилання текстових повідомлень і файлів у реальному часі через WebSocket-з’єднання, редагування профілю імені та аватара, пошук і додавання контактів, а також адміністрування груп.

Для забезпечення надійності та коректності роботи було проведено тестування ключових сценаріїв – реєстрації, надсилання повідомлень, створення групових чатів, пошуку та додавання контактів. У ході тестування підтверджено стабільну роботу застосунку при правильних діях користувача, а також виявлено реакцію системи на типові помилки.

Таким чином, результати практичної реалізації та тестування підтверджують, що розроблений чат-застосунок відповідає заявленим вимогам, є функціонально повним, зручним у використанні та готовим до експлуатації.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи бакалавра було розроблено десктопний чат-застосунок з підтримкою обміну повідомленнями в режимі реального часу, що відповідає сучасним вимогам до надійності та зручності користування.

Для реалізації системи використано сучасний стек технологій - на стороні клієнта – Electron, React та TypeScript, на стороні сервера – NestJS, MongoDB та Socket.IO.

У ході виконання роботи було проведено аналіз предметної галузі, визначено вимоги до системи, розроблено архітектуру клієнт-серверної взаємодії, спроєктовано UML-діаграми варіантів використання та структурні схеми бази даних. Реалізовано повний цикл розробки програмного забезпечення – від проєктування до розгортання системи на хмарній платформі Render з використанням системи контролю версій Git та платформи GitHub.

Розроблена система забезпечує реєстрацію та автентифікацію користувачів, керування контактами, створення особистих і групових чатів, надсилання й отримання повідомлень у режимі реального часу, а також відображення статусів онлайн та індикатора набору тексту.

У результаті виконання кваліфікаційної роботи було набуто практичного досвіду з проєктування клієнт-серверних застосунків, роботи з WebSocket-з'єднаннями в реальному часі, побудови REST API та організації автентифікації на основі токенів. Отримані знання та навички можуть бути застосовані при подальшій розробці та вдосконаленні системи.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Telegram. Офіційний сайт десктопного месенджера. https://telegram.org/ (дата звернення: 04.05.2026).

2. Discord Developer Documentation. https://discord.com/developers/docs/intro (дата звернення: 15.05.2026).

3. Signal. Офіційний сайт десктопного месенджера. https://signal.org/ (дата звернення: 18.05.2026).

4. Electron. Офіційна документація. https://www.electronjs.org/docs/latest/ (дата звернення: 21.01.2026).

5. React. Офіційна документація. https://react.dev/reference/react (дата звернення: 22.02.2026).

6. Socket.IO. Офіційна документація. https://socket.io/docs/v4/ (дата звернення: 03.03.2026).

7. MongoDB. Офіційна документація. https://www.mongodb.com/docs/ (дата звернення: 03.02.2026).

8. NestJS. Офіційна документація. https://docs.nestjs.com/ (дата звернення: 16.03.2026).

9. TypeScript Handbook. https://www.typescriptlang.org/docs/ (дата звернення: 13.03.2026).

10. Node.js. Офіційна документація. https://nodejs.org/en/docs/ (дата звернення: 04.04.2026).

11. Mongoose. Офіційна документація. https://mongoosejs.com/docs/ (дата звернення: 15.04.2026).

12. JWT (JSON Web Token) Introduction. https://jwt.io/introduction (дата звернення: 25.04.2026).

13. Render. Офіційна документація. URL: https://render.com/docs (Дата звернення: 08.06.2026).

14. WebRTC API / MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API (дата звернення: 01.05.2026).

15. Git Documentation. https://git-scm.com/doc (дата звернення: 21.05.2026).

16. GitHub Docs. https://docs.github.com/ (дата звернення: 22.05.2026).

17. WebSocket API / MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API (дата звернення: 23.05.2026).

18. The WebSocket Protocol (RFC 6455). https://datatracker.ietf.org/doc/html/rfc6455 (дата звернення: 23.05.2026).

19. JSON Web Token (JWT) (RFC 7519). https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 24.05.2026).

20. Fetch API / MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API (дата звернення: 24.05.2026).

21. Window: localStorage property / MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage (дата звернення: 24.05.2026).

22. bcrypt / npm. https://www.npmjs.com/package/bcrypt (дата звернення: 25.05.2026).

23. Zustand Documentation. https://zustand.docs.pmnd.rs/ (дата звернення: 25.05.2026).

24. React Router Documentation. https://reactrouter.com/ (дата звернення: 26.05.2026).

25. npm Docs. https://docs.npmjs.com/ (дата звернення: 26.05.2026).

26. MDN Web Docs. https://developer.mozilla.org/ (дата звернення: 27.05.2026).

27. HTML Living Standard. //html.spec.whatwg.org/ (дата звернення: 27.05.2026).

28. ECMAScript Language Specification. https://tc39.es/ecma262/ (дата звернення: 28.05.2026).

29. CSS Working Group Specification. https://www.w3.org/Style/CSS/ (дата звернення: 28.05.2026).

30. HTTP Semantics. https://datatracker.ietf.org/doc/html/rfc9110 (дата звернення: 29.05.2026).

ДОДАТОК А

Ініціалізація WebSocket-клієнта

```
import { io, Socket } from 'socket.io-client';


let socket: Socket | null = null;
export const initSocket = (token: string): Socket => {
  let socketUrl: string;

  const viteSocketUrl = import.meta.env.VITE_SOCKET_URL;
  if (viteSocketUrl) {
    socketUrl = viteSocketUrl;
  } else {
    const hostname = window.location.hostname;
    const protocol = window.location.protocol === 'https:' ? 'https' : 'http';
    socketUrl = `${protocol}://${hostname}:3001`;
  }

  console.log('Connecting to socket:', socketUrl);

  socket = io(socketUrl, {
    auth: { token },
    transports: ['websocket', 'polling'],
    reconnection: true,
    reconnectionDelay: 1000,
    reconnectionDelayMax: 5000,
    reconnectionAttempts: 5,
    forceNew: false,
  });

  socket.on('connect_error', (error) => {
    console.error('Socket connection error:', error);
  });

  socket.on('disconnect', (reason) => {
    console.warn('Socket disconnected:', reason);
  });
  return socket;
};

export const getSocket = (): Socket | null => socket;

export const disconnectSocket = () => {
  if (socket) { socket.disconnect(); socket = null; }
```

ДОДАТОК Б

Методи видалення, оновлення групи, видалення учасника

```
import { Injectable, ForbiddenException, NotFoundException } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Model } from 'mongoose';
import { Chat, ChatDocument } from './chat.schema';
import { Message, MessageDocument } from '../messages/message.schema';
@Injectable()
export class ChatsService {
  constructor(
    @InjectModel(Chat.name) private chatModel: Model<ChatDocument>,
    @InjectModel(Message.name) private messageModel: Model<MessageDocument>,
  ) {}
  async getUserChats(userId: string) {
    return this.chatModel.find({ members: userId }).sort({ updatedAt: -1 });
  }
  async createPrivateChat(userId: string, targetUserId: string) {
    const existing = await this.chatModel.findOne({
      type: 'private',
      members: { $all: [userId, targetUserId], $size: 2 },
    });
    if (existing) return existing;
    return this.chatModel.create({
      type: 'private',
      members: [userId, targetUserId],
    });
  }
  async createGroupChat(adminId: string, name: string, memberIds: string[]) {
    const members = [...new Set([adminId, ...memberIds])];
    return this.chatModel.create({
      type: 'group',
      name,
      members,
      adminId,
    });
  }
  async getChatMessages(chatId: string, userId: string, page: number = 1, limit: number = 50) {
    const chat = await this.chatModel.findById(chatId);
    if (!chat || !chat.members.includes(userId)) throw new ForbiddenException();
    const skip = (page - 1) * limit;
    const total = await this.messageModel.countDocuments({ chatId });
    const messages = await this.messageModel
      .find({ chatId })
      .sort({ createdAt: -1 })
      .skip(skip)
      .limit(limit)
      .exec();
    return {
      messages: messages.reverse(),
      pagination: {
        page,
        limit,
        total,
        pages: Math.ceil(total / limit),
      },
    };
  }
async deleteMessage(messageId: string, userId: string) {
```

```
  const msg = await this.messageModel.findById(messageId);
  if (!msg) throw new ForbiddenException('Message not found');
  const chat = await this.chatModel.findById(msg.chatId);
  if (!chat) throw new ForbiddenException('Chat not found');
  const isAdmin = chat.type === 'group' && chat.adminId === userId;
  const isAuthor = msg.senderId === userId;
  if (!isAdmin && !isAuthor) throw new ForbiddenException('Not allowed to delete this message');
    msg.isDeleted = true;
  msg.text = 'Повідомлення видалено';
  msg.fileUrl = null;
  msg.fileType = null;
  msg.fileName = null;
  await msg.save();
  return msg;
}
  async addMembersToGroup(chatId: string, adminId: string, newMemberIds: string[]) {
    const chat = await this.chatModel.findById(chatId);
    if (!chat || chat.adminId !== adminId) throw new ForbiddenException('Not admin');
    const updated = await this.chatModel.findByIdAndUpdate(
      chatId,
      { $addToSet: { members: { $each: newMemberIds } } },
      { new: true },
    );
    return updated;
  }
  async deleteChat(chatId: string, userId: string) {
    const chat = await this.chatModel.findById(chatId);
    if (!chat || !chat.members.includes(userId)) throw new ForbiddenException('Not member of this chat');

    await this.messageModel.deleteMany({ chatId });
    await this.chatModel.findByIdAndDelete(chatId);
    return { success: true };
  }
  async updateGroupName(chatId: string, userId: string, newName: string) {
    const chat = await this.chatModel.findById(chatId);
    if (!chat) throw new NotFoundException('Chat not found');
    if (chat.type !== 'group') throw new ForbiddenException('Not a group chat');
    if (!chat.members.includes(userId)) throw new ForbiddenException('Not a member');
    return this.chatModel.findByIdAndUpdate(chatId, { name: newName }, { new: true });
  }
  async removeMemberFromGroup(chatId: string, adminId: string, memberIdToRemove: string) {
    const chat = await this.chatModel.findById(chatId);
    if (!chat) throw new NotFoundException('Chat not found');
    if (chat.type !== 'group') throw new ForbiddenException('Not a group');
    if (chat.adminId !== adminId) throw new ForbiddenException('Only admin can remove members');
    if (adminId === memberIdToRemove) throw new ForbiddenException('Admin cannot remove themselves');
    const updated = await this.chatModel.findByIdAndUpdate(
      chatId,
      { $pull: { members: memberIdToRemove } },
      { new: true }
    );
    return updated;
  }
  async leaveGroup(chatId: string, userId: string) {
    const chat = await this.chatModel.findById(chatId);
    if (!chat || !chat.members.includes(userId)) throw new ForbiddenException('Not a member');
    if (chat.adminId === userId) throw new ForbiddenException('Admin cannot leave, must delete group or transfer
ownership');
    await this.chatModel.findByIdAndUpdate(chatId, { $pull: { members: userId } });
    return { success: true };
}
```

ДОДАТОК В

WebSocket та глобальні обробники

```
import React, { useEffect, useState } from 'react';
import { useAuthStore } from './store/authStore';
import { useChatStore } from './store/chatStore';
import { useUsersStore } from './store/usersStore';
import { initSocket, disconnectSocket, getSocket } from './utils/socket';
import api from './utils/api';
import TitleBar from './components/layout/TitleBar';
import Sidebar from './components/layout/Sidebar';
import ChatList from './components/chat/ChatList';
import ChatWindow from './components/chat/ChatWindow';
import ContactsList from './components/contacts/ContactsList';
import SearchPage from './pages/SearchPage';
import SettingsPage from './pages/SettingsPage';
import AuthPage from './pages/AuthPage';
import './styles/globals.css';

export default function App() {
  const { user, token, updateUser } = useAuthStore();
  const { setChats, addChat, addMessage, deleteMessage, updateChatLastMessage } = useChatStore();
  const { setContacts, setOnlineStatus, updateContactName, getUserById } = useUsersStore();
  const [activePage, setActivePage] = useState('chats');

  useEffect(() => {
    const handleNativeContextMenu = (e: MouseEvent) => {
      const target = e.target as HTMLElement;
      if (target.tagName === 'INPUT' || target.tagName === 'TEXTAREA') {
        e.preventDefault();
        (window as any).electronAPI?.showContextMenu(e.clientX, e.clientY);
      }
    };
    document.addEventListener('contextmenu', handleNativeContextMenu);
    return () => document.removeEventListener('contextmenu', handleNativeContextMenu);
  }, []);

  useEffect(() => {
    if (!token || !user) return;

    api.get('/chats').then(r => setChats(r.data)).catch(console.error);
    api.get('/users/contacts').then(r => setContacts(r.data)).catch(console.error);

    const socket = initSocket(token);
    const electronAPI = (window as any).electronAPI;

    socket.on('connect', () => {
      console.log('Socket connected');
    });

    socket.on('message:new', async (msg: any) => {
      addMessage(msg);
      updateChatLastMessage(msg.chatId, {
        text: msg.text,
        senderId: msg.senderId,
        createdAt: msg.createdAt,
      });

      const currentUser = useAuthStore.getState().user;
```

```
    if (msg.senderId !== currentUser?._id && electronAPI) {
      let isFocused = true;
      if (electronAPI.isWindowFocused) {
        isFocused = await electronAPI.isWindowFocused();
      }
      if (!isFocused && electronAPI.showNotification) {
        const sender = getUserById(msg.senderId);
        const senderName = sender?.username || 'Користувач';
        electronAPI.showNotification(senderName, msg.text || 'Нове повідомлення');
      }
    }
  });

  socket.on('message:deleted', ({ messageId, chatId }: any) => {
    deleteMessage(chatId, messageId);

    const chat = useChatStore.getState().chats.find(c => c._id === chatId);
    const messages = useChatStore.getState().messages[chatId] || [];
    const deletedMsg = messages.find(m => m._id === messageId);
    if (chat && chat.lastMessage && deletedMsg && chat.lastMessage.createdAt === deletedMsg.createdAt) {
      const prevMsg = [...messages].reverse().find(m => !m.isDeleted && m._id !== messageId);
      if (prevMsg) {
        updateChatLastMessage(chatId, {
          text: prevMsg.text,
          senderId: prevMsg.senderId,
          createdAt: prevMsg.createdAt,
        });
      } else {
        updateChatLastMessage(chatId, {
          text: 'Повідомлення видалено',
          senderId: '',
          createdAt: new Date().toISOString(),
        });
      }
    }
  });

  socket.on('chat:created', (chat: any) => {
    addChat(chat);
  });

  socket.on('user:online', ({ userId, isOnline, lastSeen }: any) => {
    setOnlineStatus(userId, isOnline, lastSeen);
  });

  socket.on('user:nameChanged', ({ userId, username }: any) => {
    updateContactName(userId, username);
    if (userId === user._id) {
      updateUser({ username });
    }
  });

  return () => {
    disconnectSocket();
  };
}, [token, user?._id]);

return (
  <div className="app-shell">
    {!user || !token ? (
      <AuthPage />
```

```
    ) : (
      <div className="app-body">
        <Sidebar activePage={activePage} setActivePage={setActivePage} />
        <div className="app-panel">
          {activePage === 'chats' && <ChatList />}
          {activePage === 'contacts' && <ContactsList setActivePage={setActivePage} />}
          {activePage === 'search' && <SearchPage />}
          {activePage === 'settings' && <SettingsPage />}
        </div>
        {activePage === 'chats' && (
          <div className="app-main">
            <ChatWindow />
          </div>
        )}
        {activePage !== 'chats' && (
          <div className="app-main--empty">
            <div style={{ color: 'var(--color-text-faint)', textAlign: 'center' }}>
              <div style={{ fontSize: 48, marginBottom: 16 }}>💬</div>
              <p style={{ fontSize: 'var(--text-sm)' }}>Перейдіть у чати, щоб переписуватись</p>
            </div>
          </div>
        )}
      </div>
    )}
  </div>
 );
}
```