**ДИПЛОМНА РОБОТА**

**ДР.122.042.001.ПЗ**

**КУЗМІНСЬКИЙ РУСЛАН ОЛЕКСАНДРОВИЧ**

**2025**

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЖИТОМИРСЬКИЙ АГРОТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ

ВІДДІЛЕННЯ «Інженерна інфраструктура та комп'ютерні науки»

ЦИКЛОВА КОМІСІЯ СПЕЦІАЛЬНОСТІ «КОМП’ЮТЕРНІ НАУКИ»

**ПОЯСНЮВАЛЬНА ЗАПИСКА**

до дипломної роботи освітнього ступеня «фаховий молодший бакалавр»

за спеціальністю 122 «Комп’ютерні науки»

(освітня програма 122 «Комп’ютерні науки»)

на тему:

**«Розробка веб-месенджера з можливістю обміну повідомленнями в локальній мережі для корпоративного використання»**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студента | 4 | курсу | П-42 | групи |
| Кузмінського Руслана Олександровича |
| (ПІБ) |
| Керівник | Данилюк Ілля Володимирович |
| Рецензент | Можаровський Сергій Володимирович |
|  |
| Національна шкала |  |
| Кількість балів |  | Оцінка: ECTS |  |
|  |  |  |  |
| Члени комісії |  | Ісаєв А.М. |
|  | (підпис) | (прізвище та ініціали) |
|  |  | Данилюк І.В. |
|  | (підпис) | (прізвище та ініціали) |
|  |  | Устименко Л.М. |
|  | (підпис) | (прізвище та ініціали) |

 Устименко Я.І.

|  |  |
| --- | --- |
| (підпис) | (прізвище та ініціали) |

Житомир – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЖИТОМИРСЬКИЙ АГРОТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ

ВІДДІЛЕННЯ «Інженерна інфраструктура та комп'ютерні науки»

ЦИКЛОВА КОМІСІЯ СПЕЦІАЛЬНОСТІ «КОМП’ЮТЕРНІ НАУКИ»

**«ЗАТВЕРДЖУЮ»**

Голова циклової комісії спеціальності

«Комп’ютерні науки»

 Сергій МОЖАРОВСЬКИЙ

«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_202\_\_ р.

**ЗАВДАННЯ**

**на дипломну роботу**

Здобувач вищої освіти: **КУЗМІНСЬКИЙ Руслан Олександрович**

Керівник роботи: **ДАНИЛЮК Ілля Володимирович**

Тема роботи: **«Розробка веб-месенджера з можливістю обміну повідомленнями в локальній мережі для корпоративного використання»**, затверджена наказом закладу вищої освіти від «27» грудня 2023 р., №532у.

Вихідні дані для роботи: створити універсальній веб-додаток для інтернет-магазину ігор, яка забезпечує користувачам зручний інтерфейс для перегляду, купівлі та відгуків про ігри.

|  |  |  |  |
| --- | --- | --- | --- |
| Розділ | Консультант | Завданнявидав | Завдання прийняв |
| 1 | Ілля данилюк | 01-02-2024 | 01-02-2024 |
| 2 | Ілля данилюк | 18-03-2024 | 18-03-2024 |
| 3 | Ілля данилюк | 23-04-2024 | 23-04-2024 |

Консультанти з дипломної роботи із зазначенням розділів, що їх стосується:

**Календарний план**

|  |  |  |  |
| --- | --- | --- | --- |
| № з/п | Етап роботи | Термін виконання | Примітка |
| 1 | Формулювання мети, визначення завдань розробки | 01 лютого 2025 | Виконано |
| 2 | Планування розробки | 14 лютого 2025 | Виконано |
| 3 | Проєктування користувацької частини сайту | 01 березня 2025 | Виконано |
| 4 | Проєктування дизайну проєкту | 11 березня 2025 | Виконано |
| 5 | Розробка алгоритмів  | 28 березня 2025 | Виконано |
| 6 | Перевірка роботи алгоритмів | 10 квітня 2025 | Виконано |
| 7 | Проєктування функцій проєкту | 23 квітня 2025 | Виконано |
| 8 | Налагодження взаємодії між елементами | 30 квітня 2025 | Виконано |
| 9 | Розробка дизайну для різних пристроїв | 10 травня 2025 | Виконано |
| 10 | Тестування всіх функцій сайту | 20 травня 2025 | Виконано |
| 11 | Формулювання мети, визначення завдань розробки | 30 травня 2025 | Виконано |
| 12 | Попередній захист роботи | 06 червня 2025 |  |

Здобувач фахової передвищої освіти\_\_\_\_\_\_\_\_\_\_ Руслан КУЗМІНСЬКИЙ

Керівник \_\_\_\_\_\_\_\_\_\_ Ілля ДАНИЛЮК

**РЕФЕРАТ**

Записка: 59 стор., 10 рис., 10 джерел.
ВЕБ-ЧАТ, WEBSOCKET, NODE.JS, HTML, CSS, JAVASCRIPT, КЛІЄНТ-СЕРВЕР, КОРПОРАТИВНА КОМУНІКАЦІЯ

Об'єкт дослідження — процеси інтерактивної комунікації між користувачами в реальному часі за допомогою вебтехнологій.
Мета роботи — розробка функціонального корпоративного веб-чату, який забезпечує миттєвий обмін повідомленнями між користувачами без використання баз даних і без потреби в авторизації.

Методи дослідження — аналіз існуючих рішень у сфері веб-комунікацій, побудова клієнт-серверної архітектури, застосування протоколу WebSocket, моделювання структури інтерфейсу та реалізація за допомогою HTML, CSS і JavaScript.

Результати — реалізовано веб-додаток для локального чату між користувачами. Розроблено WebSocket-сервер на Node.js, побудовано клієнтську частину для відображення повідомлень. Система працює без зберігання повідомлень у базі даних, не потребує реєстрації, а повідомлення обробляються в реальному часі. Чат підтримує багатокористувацький режим, відображає авторство повідомлень та має індикатор з’єднання.

ABSTRACT

The thesis consists of a fully functional web-based corporate chat system and supporting documentation. Documentation volume: 59 pages, 10 figures, 10 sources.
KEY WORDS: WEB CHAT, WEBSOCKET, NODE.JS, HTML, CSS, JAVASCRIPT, CLIENT-SERVER, REAL-TIME COMMUNICATION.

The subject of research is the process of real-time communication between users using modern web technologies.
The aim is to develop a lightweight corporate web chat that enables instant message exchange without using databases or user authentication.

Research methods include analysis of existing communication tools, development of client-server architecture, implementation of WebSocket protocol, user interface design, and programming with HTML, CSS, and JavaScript.

Results: a local real-time chat system has been developed. It includes a Node.js WebSocket server and a client interface for message interaction. The system does not store messages permanently, does not require login, and works over WebSocket for fast message delivery. Each message displays the sender’s name, and the system supports multiple users and connection status display.

**ЗМІСТ**

[ВСТУП 5](#_Toc200538631)

[РОЗДІЛ 1. АНАЛІЗ ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ ВЕБ-МЕССЕНДЖЕРА 6](#_Toc200538632)

[1.1. Постановка основної задачі проєкту 6](#_Toc200538633)

[1.2. Вибір технологій для розробки веб-месенджера 8](#_Toc200538634)

[Висновки до розділу 1 10](#_Toc200538635)

[РОЗДІЛ 2. ПРОЄКТУВАННЯ КОРПОРАТИВНОГО ВЕБ-МЕСЕНДЖЕРА 11](#_Toc200538636)

[2.1. Аналіз вимог та потреб месенджера 11](#_Toc200538637)

[2.2. Функціональні вимоги 13](#_Toc200538638)

[2.3. Нефункціональні вимоги 15](#_Toc200538639)

[2.4. Дослідження конкурентів 17](#_Toc200538640)

[2.5. Визначення вимог до користувацького інтерфейсу 19](#_Toc200538641)

[2.6. Дизайн інтерфейсу користувача 21](#_Toc200538642)

[Висновки до розділу 2 23](#_Toc200538643)

[РОЗДІЛ 3. РОЗРОБКА КОРПОРАТИВНОГО ВЕБ-МЕСЕНДЖЕРА 24](#_Toc200538644)

[3.1. Структура проєкту 24](#_Toc200538645)

[3.2. Структура месенджера 26](#_Toc200538646)

[3.3. Технічна реалізація корпоративного веб-чату 31](#_Toc200538647)

[Висновки до розділу 3 33](#_Toc200538648)

[ВИСНОВКИ 34](#_Toc200538649)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 36](#_Toc200538650)

[ДОДАТКИ 37](#_Toc200538651)

# ВСТУП

Сучасний світ інформаційних технологій вимагає від програмного забезпечення не лише високої продуктивності та безпеки, а й здатності забезпечувати ефективну комунікацію між користувачами. У корпоративному середовищі особливо важливим є наявність інструментів, які дозволяють працівникам миттєво обмінюватися повідомленнями, координувати дії, обговорювати завдання та швидко передавати важливу інформацію. Одним із найефективніших рішень у цьому контексті є веб-чати — засоби онлайн-комунікації, що працюють у браузері та не потребують встановлення додаткового програмного забезпечення.

Популярні месенджери, такі як Slack, Microsoft Teams, Discord, Telegram та інші, вже давно довели ефективність формату текстового спілкування у корпоративному просторі. Однак більшість із них вимагає авторизації, зберігає дані на сторонніх серверах і має складну архітектуру, що може бути недоцільним або зайвим у невеликих організаціях або в межах внутрішніх мереж. Існує потреба у легкому, незалежному рішенні, яке можна швидко налаштувати, розгорнути локально й використовувати без зовнішніх залежностей.

В основі реалізації лежить клієнт-серверна архітектура: користувачі працюють з веб-інтерфейсом у браузері (HTML, CSS, JavaScript), а обробка повідомлень здійснюється на сервері (Node.js) за допомогою WebSocket — протоколу, що дозволяє встановити постійне з’єднання між клієнтом і сервером. Повідомлення передаються у форматі JSON без збереження в базі даних, що значно спрощує архітектуру та полегшує підтримку системи.

Мета цієї роботи — створити простий, функціональний веб-чат, що працює в реальному часі, з інтерфейсом, зрозумілим для кінцевого користувача, без потреби в реєстрації чи збереженні даних. У процесі реалізації проєкту буде проаналізовано вимоги до подібних систем, досліджено наявні рішення, сформовано технічні вимоги, спроєктовано структуру та інтерфейс, а також здійснено програмну реалізацію з подальшим тестуванням.

#  РОЗДІЛ 1. АНАЛІЗ ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ ВЕБ-МЕССЕНДЖЕРА

## 1.1. Постановка основної задачі проєкту

У сучасному світі цифрових технологій інформаційна взаємодія між людьми відіграє ключову роль. Особливо важливе значення вона набуває у рамках організацій, підприємств та онлайн-спільнот, де щоденна комунікація потребує ефективних, швидких та безпечних інструментів. Одним із таких рішень є корпоративний веб-чат, який дозволяє співробітникам, учасникам проєктів або підписникам блогів оперативно обмінюватися інформацією, координувати дії та зберігати єдиний простір комунікації.

Існуючі засоби спілкування, такі як електронна пошта або зовнішні месенджери, мають низку обмежень у контексті інтеграції до внутрішніх систем, контролю доступу, захисту даних та кастомізації. Також використання сторонніх сервісів для внутрішнього спілкування іноді створює загрозу конфіденційності або порушення внутрішньої політики компанії. Тому створення власного веб-чату, який повністю адаптований під специфіку платформи (наприклад, блогу або корпоративного порталу), є актуальним завданням для ІТ-фахівців.

У цьому дипломному проєкті розглядається задача створення веб-чату, інтегрованого в середовище блогу. Такий чат має забезпечувати миттєвий обмін повідомленнями між автором контенту та читачами, а також між самими користувачами платформи. Зручність, швидкодія, інтуїтивний інтерфейс, захист персональних даних та масштабованість — основні вимоги, яким має відповідати розроблене рішення.

Особливістю цього проєкту є використання вебтехнологій — клієнт реалізується за допомогою мови програмування C# у середовищі .NET, а серверна частина — на платформі Node.js. Такий підхід дозволяє забезпечити гнучкість у розгортанні, сумісність із сучасними браузерами та хорошу продуктивність. Крім того, застосовується обмін повідомленнями у форматі JSON, що є зручним і ефективним способом передачі структурованих даних.

З технічної точки зору, чат повинен підтримувати наступні функції: авторизація користувача, введення імені для ідентифікації в чаті, відображення історії повідомлень у межах сесії, фільтрація повідомлень за автором, та виключення власних повідомлень із відображення (щоб не дублювати відправлену інформацію). Окрема увага приділяється стабільності мережевого підключення — у разі втрати зв’язку користувач повинен отримати відповідне повідомлення.

Метою дипломного проєкту є реалізація на практиці повного циклу розробки системи обміну повідомленнями: від формулювання вимог і проєктування структури застосунку до реалізації та тестування функціональної системи. Також у процесі реалізації важливо забезпечити можливість подальшого розширення функціоналу, наприклад, додавання кімнат, історії чату, підтримки файлів чи мультимедіа.

Крім функціональної частини, система має бути зручною для кінцевого користувача. Тому важливим етапом у реалізації є розробка логічної та візуальної структури інтерфейсу, яка дозволяє швидко зрозуміти принцип роботи веб-чату без необхідності звертання до інструкцій. У розробці інтерфейсу особливу роль відіграють принципи UX/UI дизайну, які повинні бути застосовані з урахуванням сучасних вебстандартів.

Таким чином, основна задача проєкту — створити сучасну, ефективну, зручну та безпечну систему корпоративного веб-чату, яка буде інтегрована у платформу блогу та відповідатиме як технічним, так і користувацьким вимогам.

## 1.2. Вибір технологій для розробки веб-месенджера

Вибір технологій — один з найважливіших етапів розробки будь-якого програмного забезпечення, особливо коли йдеться про застосунки, що працюють у реальному часі та повинні забезпечувати стабільну взаємодію між користувачами. У контексті створення корпоративного веб-чату для блогу основними критеріями вибору технологій є: швидкодія, підтримка двосторонньої комунікації (real-time), кросплатформеність, безпечність передачі даних та легкість у розгортанні й підтримці.

З урахуванням цих критеріїв було обрано архітектуру типу «клієнт-сервер», у якій клієнтська частина реалізована за допомогою мови програмування C# у середовищі .NET, а серверна частина — на основі JavaScript з використанням Node.js. Така комбінація дозволяє поєднати високопродуктивне середовище для побудови мережевих сервісів (Node.js) із зручною для розробки десктопних клієнтів мовою C#.

Node.js був обраний як серверна платформа через його асинхронну, подієво-орієнтовану архітектуру, яка ідеально підходить для задач, пов’язаних із обробкою великої кількості одночасних з’єднань. Бібліотека ws у Node.js забезпечує підтримку WebSocket — сучасного протоколу, який дозволяє відкривати постійне двостороннє з’єднання між клієнтом і сервером. Це дозволяє забезпечити миттєву доставку повідомлень, без необхідності опитування сервера або перезавантаження сторінки.

Для клієнтської частини було використано C# у рамках класичної Windows Forms-програми або .NET Console App, що дозволяє легко організувати графічний інтерфейс, обробку подій і підтримку підключення до WebSocket-сервера. C# має гарну підтримку роботи з мережею, асинхронним кодом та JSON-серіалізацією, що робить його зручним інструментом для реалізації клієнта.

Для обміну повідомленнями між клієнтом і сервером обрано формат JSON, оскільки він є простим, читабельним для людини та ефективним у передачі структурованих даних. Повідомлення передаються у вигляді об’єктів, які містять ім’я користувача, текст повідомлення та, за потреби, мітку часу.

На стороні сервера реалізовано логіку приймання та пересилання повідомлень усім підключеним клієнтам, а також базову валідацію даних. Сервер прослуховує порт, очікує вхідні з’єднання та обробляє події, пов’язані з надсиланням або відключенням клієнтів. Кожне нове повідомлення, що надходить на сервер, передається усім іншим учасникам чату в режимі реального часу.

Особливу увагу при виборі технологій було приділено безпеці. Хоча в межах даного проєкту не використовується шифрування (через локальний характер використання), архітектура передбачає можливість додавання HTTPS/SSL, авторизації через токени та обмеження доступу до WebSocket API.

Також важливо зазначити, що обрані технології дозволяють масштабувати систему. Node.js має можливість розгортання на хмарних сервісах, підтримує балансування навантаження та горизонтальне масштабування, що дозволяє використовувати ту саму архітектуру при переході від невеликої мережі до повноцінного вебсервісу.

Загалом, обраний стек технологій — C#, .NET, Node.js, WebSocket і JSON — забезпечує надійне та ефективне функціонування корпоративного веб-чату та дозволяє повністю реалізувати поставлену у проєкті мету: швидке, зручне та безпечне спілкування користувачів у рамках однієї вебплатформи.

## Висновки до розділу 1

У першому розділі було здійснено аналіз актуальності розробки корпоративного веб-чату для інтеграції у блогову платформу. Розглянуто основні проблеми, які виникають у процесі цифрової комунікації, а також обґрунтовано потребу у створенні власного рішення замість використання сторонніх месенджерів. Основними причинами такого підходу є бажання забезпечити конфіденційність, контроль над функціональністю, адаптивність до внутрішніх процесів, а також можливість масштабування і подальшого розвитку.

Чітко сформульовано мету проєкту — створити ефективний, простий у використанні та безпечний інструмент для обміну повідомленнями між користувачами в межах блогу. Основна задача передбачає реалізацію системи, яка забезпечить двосторонню комунікацію в реальному часі, не потребуватиме складної конфігурації та буде достатньо гнучкою для подальшого розширення функціоналу.

У процесі вибору технологій було розглянуто низку сучасних рішень, що використовуються для реалізації real-time додатків. Обрано поєднання технологій Node.js для серверної частини та C# з .NET для створення клієнта. Node.js надає потужні можливості для роботи з великою кількістю одночасних з’єднань завдяки асинхронній подієвій архітектурі, а C# забезпечує зручне середовище розробки десктопних застосунків і просту інтеграцію з WebSocket.

Додатково обґрунтовано вибір протоколу WebSocket для миттєвої передачі повідомлень і формату JSON для структурування даних. Це рішення є сучасним стандартом у сфері обміну даними між клієнтом і сервером, дозволяє легко масштабувати систему та інтегрувати її з іншими сервісами.

Таким чином, розділ 1 сформував теоретичну та технічну основу для реалізації проєкту. Отримані результати дозволяють перейти до етапу проєктування архітектури та інтерфейсу веб-чату, що буде розглянуто в наступному розділі.

# РОЗДІЛ 2. ПРОЄКТУВАННЯ КОРПОРАТИВНОГО ВЕБ-МЕСЕНДЖЕРА

## 2.1. Аналіз вимог та потреб месенджера

Успішне створення веб-чату неможливе без попереднього аналізу вимог до системи. Веб-чат є інструментом, що дозволяє користувачам обмінюватися повідомленнями в режимі реального часу, а також забезпечує швидкий зворотній зв’язок, інтерактивність та соціальну активність у межах платформи. У випадку інтеграції до блогу, чат відіграє додаткову роль — він стає каналом для миттєвого обговорення публікацій, обміну думками між авторами та читачами, а також підвищує залученість користувачів.

Основними учасниками системи є дві категорії користувачів: читачі блогу та автори контенту. Важливою вимогою є забезпечення для обох сторін зручного доступу до функціоналу чату без потреби встановлення додаткових додатків. Отже, чат має бути легко доступним через звичні для користувача засоби — браузер або вбудований компонент вебзастосунку.

Важливим є також збереження простоти використання. Чат не повинен перевантажувати користувача зайвими функціями — інтерфейс має бути інтуїтивно зрозумілим: поле для введення повідомлення, список отриманих повідомлень, індикатор статусу мережі. Усі ці елементи мають працювати швидко, стабільно та без потреби в складних інструкціях.

Ще однією важливою потребою є забезпечення мінімального навантаження на систему при значній кількості одночасних користувачів. У цьому контексті необхідно уникати рішень на основі довгих HTTP-запитів або частого опитування сервера. Оптимальним варіантом є використання WebSocket-з’єднання, що дозволяє тримати постійний канал зв’язку між клієнтом і сервером, надсилаючи й приймаючи повідомлення миттєво.

Урахування вимог до безпеки також є критично важливим. Незважаючи на те, що в рамках цього проєкту не реалізовано складної авторизації, система має бути готова до впровадження обмежень доступу, фільтрації вмісту повідомлень та захисту від спаму. Важливо забезпечити, щоб дані користувача (наприклад, ім’я, яке він вказує) не могли бути використані третіми особами без згоди.

З технічного боку, чат має бути незалежним від конкретної реалізації блогу. Він повинен функціонувати як окремий компонент, який можна вбудувати у будь-який вебпроєкт, що дозволить повторно використовувати рішення в інших системах.

Ще однією важливою вимогою є гнучкість у налаштуванні. Адміністратор або розробник має мати можливість змінювати базові параметри чату: порт сервера, допустимі обсяги повідомлень, кількість користувачів тощо.

Таким чином, проведений аналіз показав, що для реалізації ефективного веб-чату необхідно враховувати технічні, функціональні, інтерфейсні та безпекові вимоги. Усі ці чинники впливають на вибір архітектури, підходів до реалізації та вибору технологій, що буде докладніше розглянуто у наступних підрозділах.

## 2.2. Функціональні вимоги

Функціональні вимоги до системи є ключовими параметрами, які визначають, що саме повинен робити веб-чат з точки зору користувача. Вони охоплюють усі можливості, які мають бути реалізовані у застосунку для забезпечення його ефективного функціонування, зручності у використанні та відповідності поставленим цілям. Основною функцією корпоративного веб-чату є надання користувачам можливості швидкої, двосторонньої комунікації в режимі реального часу. Особливістю системи є її простота та орієнтація на миттєвий обмін повідомленнями без додаткових ускладнень у вигляді реєстрації, складної автентифікації або багаторівневого доступу.

Після запуску клієнтської частини програма має ініціювати сеанс взаємодії із сервером. На цьому етапі користувач повинен ввести своє ім’я, яке використовуватиметься в подальшій комунікації. Це дозволяє ідентифікувати автора кожного повідомлення, а також дає змогу іншим користувачам бачити, хто саме є відправником тієї чи іншої репліки. Після введення імені, користувач потрапляє до головного вікна чату, яке складається з поля введення повідомлення, кнопки відправки та області відображення повідомлень.

Введене повідомлення повинно миттєво надсилатися на сервер за допомогою протоколу WebSocket. На відміну від класичних HTTP-запитів, WebSocket-з’єднання дозволяє підтримувати постійний канал комунікації, що забезпечує негайну доставку інформації без затримок. Після того як повідомлення обробляється на сервері, воно розсилається всім підключеним користувачам, окрім відправника. Таким чином досягається ефект "живого" спілкування, де кожен учасник отримує повідомлення у режимі реального часу.

Історія повідомлень формується динамічно, лише під час активної сесії користувача. Це означає, що після закриття програми або її перезапуску історія очищується. Такий підхід дозволяє не зберігати зайві дані, зменшує вимоги до зберігання та дозволяє уникнути питань, пов’язаних із захистом конфіденційної інформації. У випадку масштабування системи або її інтеграції до більших проєктів можливо реалізувати збереження історії за допомогою бази даних, але в межах поточного проєкту реалізується лише тимчасове зберігання повідомлень у межах однієї сесії.

Однією з важливих функціональних особливостей є відфільтровування власних повідомлень. Тобто, коли користувач відправляє повідомлення, воно не повинно повертатися назад до нього як вхідне. Це досягається через порівняння імені користувача у вхідному повідомленні з локально збереженим ім’ям на клієнті. Якщо вони збігаються — повідомлення ігнорується. Таким чином досягається уникнення дублювання, а чат виглядає логічно: користувач бачить лише те, що надсилають інші.

Ще однією важливою функцією є виведення системного повідомлення у випадку втрати зв’язку із сервером. Система має контролювати стабільність WebSocket-з’єднання і реагувати у разі його розриву. У такій ситуації на екран користувача виводиться попередження про те, що з’єднання з сервером втрачено, що дозволяє уникнути непорозумінь і дає змогу користувачеві вчасно вжити заходів (перевірити мережу, перезапустити програму тощо).

Незважаючи на те, що система у своєму базовому вигляді підтримує лише текстові повідомлення, її архітектура дозволяє реалізувати розширення — додавання вкладень, емодзі, підтримку кількох кімнат, приватних повідомлень тощо. Однак навіть у мінімальному функціональному наборі реалізовані всі базові можливості, які роблять веб-чат повноцінним засобом комунікації.

Таким чином, функціональні вимоги до корпоративного веб-чату визначають його як простий, ефективний і зручний інструмент обміну повідомленнями, який забезпечує основну мету — швидке спілкування між користувачами у межах єдиного вебпростору.

## 2.3. Нефункціональні вимоги

Нефункціональні вимоги є не менш важливою складовою процесу проєктування та розробки програмного забезпечення, ніж функціональні. Якщо функціональні вимоги відповідають на питання «що система повинна робити», то нефункціональні — «як вона повинна це робити». У межах створення корпоративного веб-чату нефункціональні вимоги визначають характеристики продуктивності, масштабованості, зручності використання, безпеки та сумісності.

Одним із ключових нефункціональних параметрів є продуктивність. Веб-чат повинен забезпечувати швидку передачу повідомлень між усіма підключеними користувачами, навіть у випадку високого навантаження. Затримка в обміні повідомленнями повинна бути мінімальною і не перевищувати 1–2 секунд при нормальному з’єднанні. Завдяки використанню WebSocket-протоколу, який забезпечує постійне з’єднання між клієнтом і сервером, досягається висока швидкодія, що дозволяє передавати повідомлення практично миттєво.

Іншою важливою вимогою є масштабованість системи. Розроблений веб-чат повинен бути здатен обслуговувати не лише локальне використання в межах кількох пристроїв, а й при необхідності — розширюватися до більших мереж з десятками чи сотнями користувачів. Архітектура на основі Node.js дозволяє досягти цього завдяки подієвій моделі обробки запитів та можливості горизонтального масштабування серверної частини.

Зручність використання — ще один важливий аспект. Інтерфейс чату повинен бути інтуїтивно зрозумілим для користувача з мінімальним досвідом. Усі основні функції, такі як введення імені, написання та надсилання повідомлень, мають бути логічно розміщені в межах застосунку. Веб-чат має бути готовим до використання одразу після запуску без додаткових налаштувань або реєстрацій. Завдяки цьому досягається простота впровадження у робоче середовище або блог.

Надійність і стійкість до збоїв також належать до ключових нефункціональних вимог. Система повинна стабільно працювати при нормальних умовах експлуатації, а також адекватно реагувати на непередбачувані події, такі як втрата з’єднання, недоступність сервера або некоректне повідомлення. У таких випадках користувач повинен отримувати відповідне повідомлення про помилку, а система — намагатися відновити роботу без втрати стабільності.

Безпека — ще один важливий аспект, навіть якщо система не передбачає зберігання конфіденційних даних. Мінімальні вимоги до безпеки включають захист від спаму (наприклад, обмеження довжини повідомлень), можливість фільтрації неприпустимого вмісту (в майбутньому), а також можливість легко інтегрувати механізми автентифікації, якщо такі вимагатимуться. Передача даних через WebSocket може бути захищена SSL/TLS у випадку, якщо сервер буде розгортатися в мережі Інтернет.

Сумісність системи є критично важливою з огляду на різноманітність клієнтських середовищ. Клієнтська частина повинна мати можливість запуску на різних операційних системах (передусім Windows), а серверна — бути кросплатформною. Node.js дозволяє запускати сервер на будь-якому середовищі, яке підтримує JavaScript, включаючи Linux, macOS та Windows. Це надає розробнику велику гнучкість у виборі хостингу та середовища розгортання.

Також система повинна бути легко підтримуваною. Весь код має бути структурованим і задокументованим, а конфігураційні параметри — доступними для редагування. Це дозволяє іншим розробникам або адміністраторам легко адаптувати систему до нових умов або потреб.

Підсумовуючи, нефункціональні вимоги задають високий стандарт якості для розроблюваного веб-чату. Їх урахування на ранніх етапах проєктування дозволяє створити рішення, яке буде не лише функціональним, але й надійним, масштабованим, зручним і готовим до подальшого розвитку.

## 2.4. Дослідження конкурентів

Для ефективного проєктування і створення корпоративного веб-чату важливо провести аналіз існуючих рішень, які вже представлені на ринку. Дослідження конкурентів дозволяє не лише визначити найкращі практики, а й виявити недоліки, які варто уникати при розробці власної системи. Аналіз допомагає зрозуміти очікування користувачів, вимоги до зручності інтерфейсу, функціональні особливості та технічні параметри, що мають вирішальне значення у подальшій розробці.

У сучасному цифровому просторі існує велика кількість чат-систем, які можуть бути інтегровані у вебсайти або використовуватись як окремі сервіси. Серед найбільш відомих конкурентів можна виділити Slack, Discord, Telegram Web, Rocket.Chat, Mattermost та інші системи, що підтримують як публічне, так і приватне корпоративне спілкування. Більшість із них побудовані за принципом модульності та мають розширений функціонал, включаючи підтримку файлів, пошуку, приватних повідомлень, сповіщень, ботів тощо.

Наприклад, платформи Slack і Microsoft Teams надають повноцінні рішення для корпоративної комунікації, включаючи чати, відеодзвінки, інтеграцію з іншими сервісами та гнучкі засоби керування доступом. Однак такі системи зазвичай мають закритий код, потребують реєстрації, централізованого управління та залежні від зовнішніх серверів, що унеможливлює повний контроль над даними. Крім того, складність їхньої структури ускладнює інтеграцію у невеликі вебпроєкти, особливо в ті, де необхідно реалізувати лише базовий чат без надлишкових можливостей.

Існують також більш відкриті рішення, такі як Rocket.Chat або Mattermost, які мають відкритий вихідний код і можуть бути адаптовані під потреби конкретного проєкту. Ці платформи підтримують розгортання на власних серверах, а також мають можливість інтеграції з CMS, CRM та іншими системами. Проте навіть у них реалізація повноцінного веб-чату вимагає встановлення серверного програмного забезпечення, додаткової конфігурації, та у багатьох випадках — значного ресурсу на навчання персоналу або адміністрування.

Якщо розглядати вбудовані чати у блогових платформах, таких як Disqus або LiveChat, то вони мають іншу специфіку. Вони більше орієнтовані на коментування публікацій, а не на живе спілкування в режимі реального часу. Вони часто не підтримують багатосторонню комунікацію між читачами, мають значні затримки у відображенні нових повідомлень та вимагають авторизації через сторонні акаунти (Google, Facebook тощо).

Таким чином, аналіз показує, що наявні рішення або надто складні, або недостатньо адаптовані для інтеграції у просту блогову платформу. Більшість з них мають або зайвий функціонал, або складні механізми розгортання, що ускладнює їх використання у невеликих проектах. Водночас вони демонструють найкращі практики, які варто врахувати при створенні власного чату: миттєва доставка повідомлень, простий інтерфейс, можливість масштабування, а також підтримка зворотного зв’язку.

Власний веб-чат, який розробляється в межах цього проєкту, спрямований на усунення виявлених недоліків конкурентів — забезпечення повного контролю, простоти впровадження, відкритості архітектури та гнучкості у налаштуваннях. Завдяки цьому він може бути легко інтегрований у будь-яку блогову систему без втрати швидкодії, стабільності чи зручності для кінцевого користувача.

## 2.5. Визначення вимог до користувацького інтерфейсу

Користувацький інтерфейс (UI) є однією з найважливіших складових будь-якого застосунку. Саме від його зручності, логічності та візуальної привабливості залежить перше враження користувача, рівень задоволення від роботи із системою та загальна ефективність взаємодії. У рамках розробки корпоративного веб-чату інтерфейс повинен забезпечувати швидкий доступ до основного функціоналу, бути простим і мінімалістичним, а також відповідати стандартам сучасного UX/UI дизайну.

Основною вимогою до користувацького інтерфейсу є інтуїтивність. Користувач не повинен витрачати час на вивчення інструкцій або шукати приховані функції. Усе управління повинно бути очевидним з перших секунд запуску. Інтерфейс має складатися з трьох основних елементів: поля введення імені (на етапі входу), основного вікна чату з областю повідомлень і текстовим полем для введення повідомлення, а також кнопки для його надсилання.

Важливою вимогою є наявність відображення імені користувача, яке автоматично додається до кожного повідомлення. Це забезпечує персоніфікацію спілкування, дає змогу легко ідентифікувати співрозмовників та сприяє загальному впорядкуванню бесіди. Повідомлення повинні відображатися у зрозумілому форматі: спочатку ім’я відправника, далі — текст повідомлення. Кольорове виділення власних та чужих повідомлень може бути корисним для кращої візуалізації.

Інтерфейс також має повідомляти користувача про технічний стан з’єднання із сервером. У разі втрати з’єднання має з’являтися відповідне повідомлення або індикатор. Це допоможе уникнути ситуацій, коли користувач намагається надсилати повідомлення, не знаючи, що чат недоступний. У стабільному стані індикатор має бути непомітним, але з можливістю швидкого реагування у разі збою.

Ще одним важливим елементом є обробка помилок користувача. Наприклад, система повинна запобігати надсиланню порожніх повідомлень або повідомлень, що перевищують максимально допустиму довжину. Повідомлення про помилку мають бути зрозумілими, короткими та ненав’язливими, щоб не відволікати від основного функціоналу.

Інтерфейс також повинен бути адаптивним до різних розмірів вікна або екрана. Це дозволить користувачам комфортно працювати з чатом як на повноекранному десктопі, так і в меншому вікні браузера. Використання адаптивної верстки або елементів, що масштабуються, значно покращить користувацький досвід.

З естетичної точки зору інтерфейс має бути візуально легким, не перевантаженим графічними елементами, але водночас не надто аскетичним. Бажано використовувати спокійні кольори, чіткі шрифти, мінімалістичні кнопки та логічну структуру блоків. Важливо, щоб зовнішній вигляд був узгоджений із загальним дизайном блогу, в який інтегрується чат.

Підсумовуючи, інтерфейс веб-чату має відповідати таким вимогам: бути простим, функціональним, адаптивним, зрозумілим та приємним у використанні. Такий підхід дозволяє мінімізувати бар’єр входу для нових користувачів, а також забезпечує стабільну та ефективну взаємодію в межах корпоративної платформи.

## 2.6. Дизайн інтерфейсу користувача

Після визначення основних вимог до інтерфейсу постає питання про реалізацію його дизайну — зовнішнього вигляду, розташування елементів, кольорової гами та загальної візуальної структури. Дизайн інтерфейсу є не лише естетичним компонентом, але і важливою складовою взаємодії між людиною та системою. Правильний дизайн покращує ефективність користувача, знижує ймовірність помилок та створює позитивне враження від використання програмного забезпечення.

У рамках створення корпоративного веб-чату дизайн орієнтується на мінімалізм, зручність та швидкість сприйняття. Основне завдання — не відволікати користувача від головної функції — обміну повідомленнями. Тому було вирішено використовувати класичне трисекційне розміщення елементів: верхній інформативний блок з іменем користувача, центральну частину — область виводу повідомлень та нижню панель для введення нового повідомлення з кнопкою надсилання.

Типова область виводу повідомлень реалізована у вигляді вертикального скролового контейнера з автоматичним прокручуванням до останнього повідомлення. Повідомлення виводяться у форматі "Ім’я користувача: текст повідомлення", з можливістю стилізувати власні повідомлення відмінним кольором фону. Це дозволяє візуально розділяти чужі повідомлення від своїх і швидше орієнтуватись у діалозі.

Поле введення повідомлення має бути достатньо широким, аби дозволити користувачу набирати текст без дискомфорту. Важливо, щоб після натискання Enter або кнопки «Надіслати» повідомлення моментально з’являлося в чаті. Дизайн кнопки повинен чітко вказувати на її призначення — простий напис «Надіслати» або піктограма у вигляді стрілки, знайома користувачам з месенджерів.

Додатковим елементом інтерфейсу є інформаційний рядок або індикатор стану мережевого з’єднання. Його задача — повідомити користувача про втрату з’єднання із сервером у разі збою. У нормальному режимі він прихований або виглядає як зелений індикатор. У разі розриву — змінюється на червону позначку з відповідним текстом, наприклад: «Втрачено зв’язок з сервером».

Візуальний стиль інтерфейсу витриманий у світлих нейтральних кольорах — білому, сірому, синьому, щоб забезпечити читабельність тексту та зниження втоми очей при тривалому використанні. Використано простий шрифт без засічок (наприклад, Segoe UI або Arial), що дозволяє зберігати однорідність інтерфейсу з іншими сучасними вебдодатками.

Інтерфейс реалізований з урахуванням адаптивності — при зміні розмірів вікна всі елементи масштабуються відповідно до розміру екрану, що дозволяє використовувати веб-чат як на десктопах, так і на планшетах чи в невеликих вікнах браузера. Завдяки цьому веб-чат може органічно інтегруватися у будь-який сучасний блог або внутрішню корпоративну систему.

Дизайн інтерфейсу враховує можливість подальшого розширення функціоналу — передбачені місця для додавання нових елементів (сповіщення, реакції, панелі з контактами або групами). Такий підхід забезпечує гнучкість системи та її готовність до масштабування без кардинальної зміни структури вікна.

Загалом, дизайн користувацького інтерфейсу реалізовано відповідно до сучасних принципів зручності, простоти, функціональності та ефективності. Це дозволяє користувачам швидко адаптуватися до системи, легко сприймати повідомлення та виконувати необхідні дії без зайвого навантаження.

## Висновки до розділу 2

У другому розділі було детально розглянуто проєктування корпоративного веб-чату як окремого компонента, що має бути інтегрований у блогову платформу. Проведено системний аналіз вимог до системи, у тому числі функціональних та нефункціональних. Визначено, що для забезпечення ефективного функціонування веб-чату необхідно реалізувати підтримку обміну повідомленнями в режимі реального часу, ідентифікацію користувачів, фільтрацію власних повідомлень, стабільну роботу при втраті з’єднання та можливість адаптації до різних сценаріїв використання.

Значну увагу приділено дослідженню існуючих аналогів і конкурентів. Було встановлено, що більшість наявних чат-платформ є або надто складними в інтеграції до простих блогових систем, або мають надлишкову функціональність і вимагають залежності від сторонніх серверів чи сервісів. Це підтвердило доцільність розробки власного веб-чату, що повністю відповідає потребам конкретного середовища.

Визначено вимоги до користувацького інтерфейсу, який повинен бути простим, адаптивним, інтуїтивно зрозумілим і водночас функціональним. Сформульовано основні принципи дизайну, що базуються на мінімалізмі, логіці розміщення елементів та естетичному оформленні.

Таким чином, результати другого розділу закладають надійну теоретичну і концептуальну основу для реалізації практичної частини проєкту — безпосередньої розробки веб-чату. У наступному розділі розглядатимуться структура програмного забезпечення, особливості реалізації серверної та клієнтської частин, а також результати тестування функціональності.

# РОЗДІЛ 3. РОЗРОБКА КОРПОРАТИВНОГО ВЕБ-МЕСЕНДЖЕРА

## 3.1. Структура проєкту

Розроблений веб-чат побудовано за принципом клієнт-серверної архітектури. Це означає, що застосунок поділяється на дві основні частини: клієнтську (інтерфейс користувача) та серверну (обробка з’єднань, маршрутизація повідомлень, передача даних). Такий підхід дозволяє легко масштабувати систему, змінювати кожну з її частин незалежно, а також забезпечити гнучкість у виборі платформи для розміщення.

Фізично проєкт реалізовано у вигляді окремої теки, що містить усі необхідні компоненти для запуску і роботи чату. Усі файли згруповано відповідно до їх функціонального призначення.

Основні файли та їх призначення:

* **index.html** — головна HTML-сторінка, яка відображає інтерфейс чату. Містить елементи введення імені, поле для повідомлень та список чату.
* **style.css** — файл стилів, у якому реалізовано візуальне оформлення інтерфейсу: кольори, розміри, шрифти, розміщення елементів.
* **client.js** — логіка клієнта, яка керує підключенням до сервера через WebSocket, надсиланням та прийомом повідомлень, а також відображенням у браузері.
* **server.js** — серверна частина, написана на Node.js. Приймає WebSocket-з’єднання, отримує повідомлення від клієнтів і транслює їх усім підключеним учасникам (окрім відправника).
* **package.json** — конфігураційний файл Node.js-проєкту. Містить інформацію про залежності (наприклад, бібліотеку ws) та команди для запуску сервера.
* **favicon.ico** (необов’язковий) — піктограма вкладки браузера для покращення візуального сприйняття застосунку.

Проєкт не використовує базу даних, тому не містить схем або SQL-файлів. Усі повідомлення існують лише під час активної сесії. Така структура підходить для невеликих блогових або локальних рішень, а також є хорошою основою для подальшого розширення — наприклад, інтеграції авторизації, історії повідомлень, різних каналів чату тощо.



**Рисунок 3.1.1**

## 3.2. Структура месенджера

Структура веб-чату визначає внутрішню організацію взаємодії між його клієнтською та серверною частинами. Веб-чат реалізовано за класичною схемою: сервер на Node.js обслуговує клієнтів, які підключаються до нього через протокол WebSocket. Вся система функціонує без бази даних, а тому спрощується логіка зберігання, зменшується затримка та полегшується тестування. Передача повідомлень здійснюється у форматі JSON.

Клієнтська частина

Клієнтська частина веб-чату розташована у файлах index.html, client.js та style.css. HTML-розмітка визначає структуру елементів:

поле введення імені користувача;



**Рисунок 3.2.1**

блок повідомлень:



**Рисунок 3.2.2**

текстове поле введення;



**Рисунок 3.2.3**

кнопка «Надіслати»;



**Рисунок 3.2.4**

Усі елементи мають відповідні ідентифікатори для керування ними з JavaScript-коду.

Скрипт client.js реалізує логіку обробки подій: при запуску користувачеві пропонується ввести ім’я, яке зберігається у змінній на час сесії. Після цього встановлюється WebSocket-з’єднання з сервером, і користувач отримує доступ до чату. Кожне надіслане повідомлення надсилається у форматі JSON на сервер і відображається на стороні клієнта після повернення. Повідомлення інших користувачів автоматично додаються у список повідомлень, якщо автор не є поточним користувачем.

Серверна частина

Серверна частина представлена у файлі server.js. На базі бібліотеки ws створено WebSocket-сервер, який прослуховує підключення на вказаному порту. Після встановлення з’єднання кожен клієнт додається до масиву активних підключень. При отриманні повідомлення від одного з клієнтів сервер здійснює розсилку цього повідомлення усім іншим підключеним клієнтам

(окрім відправника).

Повідомлення на сервері також обробляються у форматі JSON. Сервер не зберігає жодних даних — ні історії, ні інформації про користувачів. Це означає, що після перезапуску сервера всі сеанси втрачаються, що відповідає концепції короткотривалого, легкого чату для блогів або внутрішнього використання.

Архітектурна схема взаємодії

Схематично логіку можна представити так:

* 1. Користувач відкриває index.html у браузері.



**Рисунок 3.2.5**

* 2. Скрипт client.js запитує ім’я користувача та ініціює WebSocket-з’єднання.



**Рисунок 3.2.6**



**Рисунок 3.2.7**

* 3. Користувач вводить повідомлення та натискає кнопку надсилання.



**Рисунок 3.2.8**

* 4. Повідомлення у JSON-форматі надсилається на server.js.
* 5. Сервер передає це повідомлення іншим підключеним клієнтам.
* 6. Кожен клієнт, який отримав повідомлення, відображає його у своєму вікні чату.



**Рисунок 3.2.9**

Така структура дозволяє зберегти простоту системи, мінімізувати затримки при передачі даних та забезпечити гнучкість у подальшому розширенні — зокрема, додавання функцій зберігання історії, реєстрації користувачів, авторизації, а також підтримки групових або приватних чатів.

## 3.3. Технічна реалізація корпоративного веб-чату

Технічна реалізація веб-чату передбачає створення двох основних компонентів: клієнтської частини, яка забезпечує інтерфейс користувача, і серверної частини, яка відповідає за обробку повідомлень і управління з’єднаннями між користувачами. Для взаємодії між клієнтом і сервером використано протокол WebSocket — сучасний стандарт для двостороннього обміну даними в режимі реального часу.

Клієнтська частина

Клієнтська частина веб-чату реалізована за допомогою стандартних вебтехнологій: HTML, CSS та JavaScript. Головна сторінка (index.html) містить поле для введення імені, блок чату з виводом повідомлень, поле для вводу нового повідомлення та кнопку надсилання. Стилізація інтерфейсу виконана через style.css, де визначено зовнішній вигляд елементів: кольори, шрифти, розміщення блоків тощо.

JavaScript-файл client.js відповідає за взаємодію з WebSocket-сервером. Після запуску скрипт виконує наступні дії:

* запитує ім’я користувача через prompt();
* встановлює WebSocket-з’єднання з сервером за допомогою об’єкта WebSocket();
* слухає події onmessage, щоб відображати отримані повідомлення;
* обробляє подію натискання кнопки або клавіші Enter для надсилання нового повідомлення у форматі JSON.

Кожне повідомлення має наступну структуру:

{

 "username": "Ім’я користувача",

 "message": "Текст повідомлення"

}

Повідомлення, отримані від сервера, перевіряються: якщо ім’я відправника збігається з локальним ім’ям користувача, то таке повідомлення не відображається (оскільки воно вже є на екрані у відправника). Це дозволяє уникнути дублювання. Також реалізовано перевірку з’єднання з сервером: при його втраті на екран виводиться відповідне повідомлення.

Серверна частина

Сервер реалізовано за допомогою Node.js та бібліотеки ws, яка дозволяє швидко налаштувати WebSocket-з’єднання без потреби у складній конфігурації. У файлі server.js створюється сервер, який прослуховує визначений порт (наприклад, 3000 або 8080) та приймає нові з’єднання.

Після встановлення з’єднання з клієнтом сервер додає це підключення до внутрішнього списку. Коли один з користувачів надсилає повідомлення, сервер приймає його, парсить (перетворює з JSON), і надсилає усім іншим клієнтам, що залишаються підключеними. Повідомлення не кешуються, не зберігаються у базі даних і не записуються у файл — кожна сесія є незалежною.

Важливою особливістю реалізації є відсутність потреби в авторизації або реєстрації. Це спрощує використання чату та дозволяє застосовувати його в обмежених мережах — наприклад, у рамках локального підприємства, блогової платформи або навчального проєкту.

Запуск системи

Для запуску системи необхідно встановити Node.js та бібліотеку ws командою npm install ws. Після цього сервер запускається через node server.js, а HTML-сторінку можна відкрити у будь-якому сучасному браузері. Всі підключені користувачі на одному сервері можуть бачити повідомлення один одного в реальному часі.

Можливості розширення

Архітектура реалізації дозволяє легко розширювати функціонал чату: додавати групи, історію, медіафайли, авторизацію через логін/пароль або OAuth, інтеграцію з базами даних (наприклад, MongoDB) та інші можливості. Поточна реалізація є базовим модулем для ефективного прототипування та створення кастомізованих рішень.

## Висновки до розділу 3

У третьому розділі було розглянуто практичну реалізацію корпоративного веб-чату, починаючи зі структури проєкту, опису компонентів та завершуючи детальною технічною реалізацією клієнтської і серверної частин. Система створена за принципом клієнт-серверної архітектури з використанням сучасних вебтехнологій — HTML, CSS, JavaScript для клієнта та Node.js з WebSocket-бібліотекою для сервера.

Завдяки такому підходу досягнуто високої швидкодії, простоти використання, гнучкості та можливості масштабування. Всі компоненти логічно структуровані: інтерфейс користувача максимально спрощений, передбачена мінімальна взаємодія для входу та початку спілкування, а сервер обробляє повідомлення з мінімальними затримками. Вся передача даних між клієнтами здійснюється в режимі реального часу з використанням формату JSON, що дозволяє легко інтегрувати або модифікувати функціональність у майбутньому.

Основною перевагою реалізованого рішення є незалежність від зовнішніх сервісів, простота розгортання та здатність працювати у межах локальної мережі. Усі користувачі можуть взаємодіяти між собою без збереження даних на сервері, що забезпечує додатковий рівень приватності.

Таким чином, розроблений веб-чат повністю відповідає поставленим технічним та функціональним вимогам, є надійним, стабільним у роботі та може бути легко адаптований під специфіку різних проєктів — від блогових платформ до внутрішніх корпоративних систем.

# ВИСНОВКИ

У межах виконаного проєкту було розроблено повноцінну систему корпоративного веб-чату, яка функціонує в режимі реального часу та може бути інтегрована у блогову платформу або внутрішнє корпоративне середовище. Розв’язання задачі базувалося на принципах ефективності, простоти використання, технологічної гнучкості та можливості подальшого масштабування.

У першому розділі було сформульовано основну мету розробки — створення засобу швидкої комунікації між користувачами, який би не потребував складної інфраструктури, авторизації або реєстрації, але забезпечував би достатній рівень взаємодії та стабільності роботи. Було обґрунтовано вибір клієнт-серверної архітектури, протоколу WebSocket та стеку технологій, які найбільш відповідають поставленим вимогам.

У другому розділі було проаналізовано потреби цільової аудиторії, визначено функціональні та нефункціональні вимоги до системи, досліджено конкурентні рішення та їхні обмеження. На основі цього сформовано чітке технічне бачення системи, яке дозволило перейти до розробки без перевитрат часу на перебудову архітектури. Визначення вимог до користувацького інтерфейсу та його дизайн дали змогу створити зручну та інтуїтивну взаємодію з користувачем.

У третьому розділі було реалізовано усі функціональні компоненти системи. Створено клієнтську частину з HTML-інтерфейсом та JavaScript-логікою, а також серверну частину на Node.js, яка обробляє WebSocket-з’єднання. Реалізація дозволяє користувачам надсилати повідомлення з автоматичним додаванням імені, отримувати повідомлення від інших, контролювати статус з’єднання та уникати дублювання власних повідомлень.

Результатом виконаної роботи є програмна система, яка є повністю самостійною, не залежить від сторонніх серверів чи сервісів. Для своєї роботи не потребує бази даних, що робить її легкою в налаштуванні та розгортанні, також підтримує обмін повідомленнями у режимі реального часу завдяки WebSocket. Має простий і зручний інтерфейс, адаптований для різних користувачів і може працювати як у локальній мережі, так і в Інтернеті, залежно від налаштувань сервера.

Створений веб-чат може бути використаний у блозі для оперативного обговорення публікацій, а також як база для розширених корпоративних комунікаційних інструментів. У перспективі його можна доповнити авторизацією, збереженням історії, підтримкою медіа, створенням кімнат і тем для обговорення. Таким чином, результати проєкту мають практичне значення та можуть бути адаптовані до різних сфер застосування.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Марк Л. Мерфі. *Програмування на Java для Android. Книга для розробників мобильных приложений*. – Дніпро: «Баланс Бізнес Букс», 2018. – 496 с.
2. Давид Флэнаган. *JavaScript. Подробное руководство*. – СПб.: Символ-Плюс, 2021. – 1088 с.
3. Гудман Д., Новик Дж. *HTML5 и CSS3. Разработка сайтов для любых браузеров и устройств*. – СПб.: Питер, 2020. – 416 с.
4. MDN Web Docs. <https://developer.mozilla.org/>
5. Node.js Documentation. <https://nodejs.org/en/docs/>
6. WS — WebSocket library for Node.js. <https://github.com/websockets/ws>
7. Can I use WebSockets? <https://caniuse.com/websockets>
8. Брауде Е.Ю. *Архитектура клиент-серверных приложений*. – М.: Диалектика, 2022. – 368 с.
9. ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.
10. Eric Elliott. *Programming JavaScript Applications*. – O'Reilly Media, 2014.

# ДОДАТКИ

Додаток А

**Лістинг index.html**

<!DOCTYPE html>

<html>

 <head>

 <title>Corporate IRC</title>

 <style>

 @import url("https://fonts.googleapis.com/css2?family=VT323&display=swap");

 @import url("https://fonts.googleapis.com/css2?family=Press+Start+2P&display=swap");

 body {

 margin: 0;

 padding: 0;

 height: 100vh;

 font-family: "VT323", monospace;

 background: #1a1a1a;

 color: #cfd8dc;

 display: flex;

 }

 .chat-container {

 flex: 1;

 display: flex;

 flex-direction: column;

 min-width: 0;

 }

 #messages {

 flex: 1;

 padding: 20px;

 overflow-y: auto;

 border-right: 2px solid #4a4a4a;

 }

 .message {

 margin: 8px 0;

 animation: fadeIn 0.3s ease-in;

 }

 .username {

 color: #82e0aa;

 margin-right: 8px;

 font-weight: bold;

 }

 .timestamp {

 color: #616161;

 font-size: 0.8em;

 margin-left: 8px;

 }

 .notification {

 color: #f5a623;

 text-align: center;

 font-style: italic;

 margin: 10px 0;

 }

 .input-container {

 display: flex;

 padding: 10px;

 background: #3a3a3a;

 border-top: 2px solid #4a4a4a;

 }

 #messageInput {

 flex: 1;

 padding: 12px;

 background: #4a4a4a;

 border: none;

 color: #cfd8dc;

 font-family: inherit;

 font-size: 1em;

 margin-right: 10px;

 border-radius: 4px;

 resize: none;

 }

 #sendButton {

 padding: 12px 24px;

 background: #2ecc71;

 border: none;

 color: #1a1a1a;

 font-family: "Press Start 2P", cursive;

 font-size: 0.8em;

 cursor: pointer;

 transition: all 0.2s;

 border-radius: 4px;

 text-shadow: 0 1px 0 #1d6a3d;

 box-shadow: 0 2px 0 #1d6a3d;

 }

 #sendButton:hover {

 background: #27ae60;

 }

 .user-list {

 width: 20%;

 background: #2d2d2d;

 border-left: 2px solid #4a4a4a;

 padding: 20px;

 overflow-y: auto;

 }

 .user-item {

 display: flex;

 align-items: center;

 margin: 8px 0;

 }

 .user-status {

 width: 10px;

 height: 10px;

 background: #2ecc71;

 border-radius: 50%;

 margin-right: 10px;

 flex-shrink: 0;

 }

 .modal {

 display: flex;

 justify-content: center;

 align-items: center;

 position: fixed;

 top: 0;

 left: 0;

 width: 100%;

 height: 100%;

 background: rgba(0, 0, 0, 0.8);

 }

 .modal-content {

 background: #3a3a3a;

 padding: 20px;

 border-radius: 8px;

 width: 300px;

 text-align: center;

 }

 .modal-input {

 width: calc(100% - 22px);

 padding: 10px;

 margin: 10px 0;

 background: #4a4a4a;

 border: none;

 color: #cfd8dc;

 font-family: inherit;

 font-size: 1em;

 border-radius: 4px;

 }

 @keyframes fadeIn {

 from {

 opacity: 0;

 transform: translateY(5px);

 }

 to {

 opacity: 1;

 transform: translateY(0);

 }

 }

 ::-webkit-scrollbar {

 width: 12px;

 }

 ::-webkit-scrollbar-track {

 background: #4a4a4a;

 }

 ::-webkit-scrollbar-thumb {

 background: #6a6a6a;

 border-radius: 6px;

 }

 </style>

 </head>

 <body>

 <div class="chat-container">

 <div id="messages"></div>

 <div class="input-container">

 <textarea

 id="messageInput"

 placeholder="Type your message..."

 autocomplete="off"

 rows="1"

 ></textarea>

 <button id="sendButton" disabled>SEND</button>

 </div>

 </div>

 <div class="user-list" id="userList"></div>

 <div id="usernameModal" class="modal">

 <div class="modal-content">

 <h2 style="color: #2ecc71; margin-bottom: 20px">ENTER USERNAME</h2>

 <input

 type="text"

 id="usernameInput"

 class="modal-input"

 maxlength="15"

 autofocus

 />

 <div id="errorMessage" style="color: red; height: 20px"></div>

 <button

 onclick="setUsername()"

 style="

 margin-top: 15px;

 padding: 8px 16px;

 background: #2ecc71;

 border: none;

 color: #1a1a1a;

 cursor: pointer;

 "

 id="connectButton"

 >

 CONNECT

 </button>

 </div>

 </div>

 <script>

 // DOM Elements - DECLARE THESE AT THE TOP

 const messagesDiv = document.getElementById("messages");

 const userListDiv = document.getElementById("userList");

 const sendButton = document.getElementById("sendButton");

 const messageInput = document.getElementById("messageInput");

 const usernameModal = document.getElementById("usernameModal");

 const usernameInput = document.getElementById("usernameInput");

 const errorMessage = document.getElementById("errorMessage");

 const connectButton = document.getElementById("connectButton");

 // Connection Variables

 let isUsernamePending = false;

 let pendingUsername = null; // Add missing declaration

 let reconnectAttempts = 0;

 let currentUserList = [];

 let ws = null;

 function connect() {

 ws = new WebSocket("ws://localhost:8080");

 ws.onopen = () => {

 connectButton.disabled = false;

 errorMessage.textContent = "";

 if (pendingUsername) {

 setUsername(pendingUsername);

 pendingUsername = null;

 }

 reconnectAttempts = 0;

 };

 ws.onerror = (error) => {

 console.error("WebSocket Error:", error);

 errorMessage.textContent = "Connection failed. Check server status.";

 connectButton.textContent = "CONNECT";

 connectButton.disabled = false;

 isUsernamePending = false;

 };

 ws.onclose = () => {

 errorMessage.textContent = "Disconnected. Reconnecting...";

 connectButton.disabled = true;

 setTimeout(() => {

 if (reconnectAttempts < 5) {

 reconnectAttempts++;

 connect();

 } else {

 errorMessage.textContent = "Failed to reconnect. Refresh page.";

 }

 }, 2000);

 };

 ws.onmessage = (event) => {

 const data = JSON.parse(event.data);

 if (data.type === "error") {

 if (data.message.includes("wait")) {

 // Handle spam error specifically

 errorMessage.textContent = data.message;

 setTimeout(() => {

 errorMessage.textContent = "";

 }, 2000);

 } else {

 // Handle other errors

 errorMessage.textContent = data.message;

 }

 if (isUsernamePending) {

 connectButton.textContent = "CONNECT";

 connectButton.disabled = false;

 isUsernamePending = false;

 }

 } else if (data.type === "user\_list") {

 if (isUsernamePending) {

 usernameModal.style.display = "none";

 connectButton.textContent = "CONNECT";

 connectButton.disabled = false;

 isUsernamePending = false;

 }

 currentUserList = data.users;

 renderUserList();

 } else if (data.type === "message") {

 const messageDiv = document.createElement("div");

 messageDiv.className = "message";

 messageDiv.innerHTML = `

 <span class="timestamp">[${new Date(

 data.timestamp

 ).toLocaleTimeString()}]</span>

 <span class="username">${data.username}:</span>

 ${data.text.replace(/</g, "<").replace(/>/g, ">")}

 `;

 messagesDiv.appendChild(messageDiv);

 messagesDiv.scrollTop = messagesDiv.scrollHeight;

 } else if (data.type === "notification") {

 const notificationDiv = document.createElement("div");

 notificationDiv.className = "notification";

 notificationDiv.textContent = `${data.message}`;

 messagesDiv.appendChild(notificationDiv);

 messagesDiv.scrollTop = messagesDiv.scrollHeight;

 }

 };

 }

 connect();

 function setUsername(usernameParam) {

 const username = (usernameParam || usernameInput.value).trim();

 if (!username || isUsernamePending) return;

 if (ws.readyState === WebSocket.OPEN) {

 isUsernamePending = true;

 connectButton.textContent = "CONNECTING...";

 connectButton.disabled = true;

 errorMessage.textContent = "";

 ws.send(JSON.stringify({ type: "set\_username", username }));

 } else {

 errorMessage.textContent = "Waiting for server connection...";

 pendingUsername = username;

 connectButton.textContent = "CONNECT";

 connectButton.disabled = true;

 }

 }

 usernameInput.addEventListener("input", () => {

 const username = usernameInput.value.trim();

 const isValid = /^[a-zA-Z0-9\_]{3,15}$/.test(username);

 connectButton.disabled = !isValid || isUsernamePending;

 });

 usernameInput.addEventListener("keypress", (e) => {

 if (e.key === "Enter") setUsername();

 });

 // Add cooldown visual feedback

 let sendCooldown = false;

 const COOLDOWN\_MS = 3000; // 3-second cooldown, must match server

 function sendMessage() {

 if (sendCooldown) return;

 const text = messageInput.value.trim();

 if (text && ws.readyState === WebSocket.OPEN) {

 ws.send(JSON.stringify({ type: "message", text }));

 messageInput.value = "";

 sendButton.disabled = true;

 messageInput.rows = 1;

 messageInput.style.height = "auto";

 messageInput.focus();

 // Visual cooldown feedback

 sendCooldown = true;

 sendButton.textContent = "WAIT...";

 setTimeout(() => {

 sendCooldown = false;

 sendButton.textContent = "SEND";

 sendButton.disabled = messageInput.value.trim() === "";

 }, COOLDOWN\_MS);

 }

 }

 function renderUserList() {

 userListDiv.innerHTML = "";

 currentUserList.sort().forEach((user) => {

 const userDiv = document.createElement("div");

 userDiv.className = "user-item";

 userDiv.innerHTML = `

 <div class="user-status"></div>

 <div class="user-name">${user}</div>

 `;

 userListDiv.appendChild(userDiv);

 });

 }

 sendButton.addEventListener("click", sendMessage);

 messageInput.addEventListener("input", () => {

 sendButton.disabled = messageInput.value.trim() === "";

 messageInput.rows = 1;

 messageInput.style.height = "auto";

 messageInput.style.height = messageInput.scrollHeight + "px";

 });

 messageInput.addEventListener("keypress", (e) => {

 if (e.key === "Enter" && !e.shiftKey) {

 e.preventDefault();

 sendMessage();

 }

 });

 usernameInput.focus();

 </script>

 </body>

</html>

**Лістинг package.json**

{

 "name": "ruslan-diploma",

 "version": "1.0.0",

 "main": "server.js",

 "scripts": {

 "test": "echo \"Error: no test specified\" && exit 1",

 "start": "node server.js"

 },

 "keywords": [],

 "author": "",

 "license": "ISC",

 "dependencies": {

 "ws": "^8.18.2"

 },

 "description": ""

}

**Лістинг server.js**

const http = require("http");

const fs = require("fs");

const path = require("path");

const WebSocket = require("ws");

const server = http.createServer((req, res) => {

 let filePath = "." + req.url;

 if (filePath === "./") filePath = "./index.html";

 const extname = path.extname(filePath);

 let contentType = "text/html";

 switch (extname) {

 case ".js":

 contentType = "text/javascript";

 break;

 case ".css":

 contentType = "text/css";

 break;

 case ".json":

 contentType = "application/json";

 break;

 case ".png":

 contentType = "image/png";

 break;

 case ".jpg":

 contentType = "image/jpg";

 break;

 }

 fs.readFile(filePath, (err, content) => {

 if (err) {

 res.writeHead(err.code === "ENOENT" ? 404 : 500);

 res.end(err.code === "ENOENT" ? "File not found" : "Server error");

 } else {

 res.writeHead(200, { "Content-Type": contentType });

 res.end(content, "utf-8");

 }

 });

});

const wsServer = new WebSocket.Server({ server });

const clients = new Map();

const messageCooldowns = new Map();

const COOLDOWN\_MS = 3000;

wsServer.on("connection", (socket) => {

 console.log("New client connected");

 socket.on("message", (message) => {

 const data = JSON.parse(message.toString());

 if (data.type === "set\_username") {

 const username = data.username.trim();

 if (!username) {

 socket.send(

 JSON.stringify({ type: "error", message: "Username cannot be empty" })

 );

 return;

 }

 if (!/^[a-zA-Z0-9\_]{3,15}$/.test(username)) {

 socket.send(

 JSON.stringify({

 type: "error",

 message:

 "Username must be 3-15 characters (letters, numbers, underscores)",

 })

 );

 return;

 }

 if (Array.from(clients.values()).includes(username)) {

 socket.send(

 JSON.stringify({ type: "error", message: "Username already taken" })

 );

 return;

 }

 clients.set(socket, username);

 sendUserList(socket);

 broadcastUserList();

 broadcast({

 type: "notification",

 message: `${username} joined the chat`,

 });

 } else if (data.type === "message") {

 const now = Date.now();

 const lastMessageTime = messageCooldowns.get(socket);

 if (lastMessageTime && now - lastMessageTime < COOLDOWN\_MS) {

 socket.send(

 JSON.stringify({

 type: "error",

 message: `Please wait ${COOLDOWN\_MS / 1000} seconds between messages`,

 })

 );

 return;

 }

 messageCooldowns.set(socket, now);

 const user = clients.get(socket);

 if (user) {

 broadcast({

 type: "message",

 username: user,

 text: data.text,

 timestamp: new Date().toISOString(),

 });

 }

 }

 });

 socket.on("close", () => {

 const username = clients.get(socket);

 if (username) {

 clients.delete(socket);

 messageCooldowns.delete(socket);

 broadcastUserList();

 broadcast({

 type: "notification",

 message: `${username} left the chat`,

 });

 }

 });

});

function broadcast(data, excludeSocket) {

 for (const client of clients.keys()) {

 if (client !== excludeSocket && client.readyState === WebSocket.OPEN) {

 client.send(JSON.stringify(data));

 }

 }

}

function broadcastUserList() {

 const users = Array.from(clients.values()).sort();

 broadcast({

 type: "user\_list",

 users: users,

 });

}

function sendUserList(socket) {

 const users = Array.from(clients.values()).sort();

 socket.send(

 JSON.stringify({

 type: "user\_list",

 users: users,

 })

 );

}

server.listen(8080, "0.0.0.0", () => {

 console.log("Server running on port 8080");

});