ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Веб додаток для навчання студентів на базі WebSocket»

на здобуття освітнього ступеня магістра
зі спеціальності 122 Комп'ютерні науки
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерні науки
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Андрій ПІЛОСЯН
(підпис) *(Ім'я, ПРІЗВИЩЕ здобувача)*

Виконав:
здобувач вищої освіти Андрій ПІЛОСЯН
група КНДМ-62

Керівник: Віталій КОТЕЛЯНЕЦЬ
науковий ступінь, вчене звання к.т.н.

Рецензент:
науковий ступінь, вчене звання
_____
(Ім'я, ПРІЗВИЩЕ)

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерних наук

Ступінь вищої освіти Магістр

Спеціальність Комп'ютерні науки

Освітньо-професійна програма Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедрою Комп'ютерних наук

_____ Віктор ВИШНІВСЬКИЙ
«_____» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Пілосяну Андрію Андрійовичу
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «Веб додаток для навчання студентів на базі WebSocket»

керівник кваліфікаційної роботи Віталій КОТЕЛЯНЕЦЬ к.т.н.,
(Ім'я, ПРІЗВИЩЕ науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024р. №320

2. Строк подання кваліфікаційної роботи «15» грудня 2024р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, методи інтерактивної взаємодії в веб-додатках, використання технології WebSocket для двосторонньої комунікації, методи розробки серверної та клієнтської частин веб-додатків.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   4.1. Аналіз сучасних технологій та інструментів для створення веб-додатка на базі протоколу WebSocket.
   4.2. Проектування архітектури веб-додатка та підбір технологій для реалізації.

4.3 Розробка веб-додатка для навчання студентів з використанням протоколу WebSocket, які включають розробку функціоналу авторизації, обміну повідомленнями, створення та управління чатами, інтеграції медіафайлів, управління чорним списком.
4.4. Тестування та оптимізація продуктивності системи, зокрема запитів до бази даних і ефективного використання серверних ресурсів.

5. Перелік графічного матеріалу: *презентація*
5.1 Приклади використання програмних конструкцій TypeScript, React, Redux
5.2 Скріншоти та приклади інтерфейсу клієнтської частини веб-додатка.
5.3 Ілюстрації, які демонструють роботу веб додатку, та його функціональність
5.4 Головні компоненти створеної системи.
6. Дата видачі завдання«16» вересня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	16.09-27.09.24	Виконано
2	Аналіз методів інтерактивної взаємодії та технологій WebSocket	30.09-11.10.24	Виконано
3	Проектування архітектури веб-додатка та підбір технологій реалізації	14.10-25.10.24	Виконано
4	Розробка серверної частини веб-додатка	28.10-08.11.24	Виконано
5	Розробка клієнтської частини веб-додатка	11.11-22.12.24	Виконано
6	Реалізація головних компонентів створеної системи	25.11-29.11.24	Виконано
7	Оформлення роботи: вступ, висновки, реферат	02.12-06.12.24	Виконано
8	Розробка демонстраційних матеріалів	09.12-13.12.24	Виконано

Здобувач вищої освіти _____ Андрій ПІЛОСЯН
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Віталій КОТЕЛЯНЕЦЬ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 63 стор., 57 рис., 12 джерел.

Наукове завдання – розробка веб-додатка для навчання студентів на базі WebSocket з високошвидкісною комунікацією в реальному часі.

Мета роботи – підвищення ефективності навчального процесу через впровадження інтерактивного веб-додатка на базі WebSocket, що забезпечує високошвидкісну двосторонню взаємодію між студентами та викладачами в режимі реального часу з мінімальними затримками.

Об'єкт дослідження – процес інтерактивної взаємодії між студентами та викладачами у веб-додатку на базі WebSocket.

Предмет дослідження – технології створення інтерактивних веб-додатків з використанням WebSocket для двосторонньої комунікації в реальному часі, орієнтованої на підтримку процесу навчання.

Методи дослідження – порівняльний аналіз технологій для комунікації в реальному часі, проектування архітектури веб-додатка, функціональне тестування з акцентом на затримки, оптимізацію продуктивності та використання серверних ресурсів.

Короткий зміст роботи: У роботі проаналізовано сучасні технології та інструменти створення веб-додатків на базі WebSocket та порівнюються з традиційними методами обміну даними (Polling, Long Polling, SSE). Розроблено архітектуру веб-додатка для двосторонньої комунікації між клієнтом і сервером. Реалізовано функціонал веб-чату, що включає обмін повідомленнями, створення чатів, інтеграцію медіафайлів, чорний список та авторизацію за токенами. Проведено тестування і оптимізацію продуктивності, зокрема запитів до бази даних та використання серверних ресурсів.

Ключові слова: WebSocket, SocketIO веб-додаток, React, Redux, TypeScript, Node.js, інтерактивне навчання, база даних MySQL, оптимізація, тестування.

ABSTRACT

Text part of the master's qualification work: 63 pages, 57 pictures, 12 sources.

Scientific task is to develop of a web application for student learning based on WebSocket with high-speed real-time communication.

Goal of the work is to enhancing the efficiency of the educational process through the implementation of an interactive web application based on WebSocket, ensuring high-speed two-way interaction between students and teachers in real time with minimal delays.

Object of research – the process of interactive interaction between students and teachers in a WebSocket-based web application.

Subject of research – technologies for creating interactive web applications using WebSocket for two-way real-time communication, aimed at supporting the educational process.

Research methods – comparative analysis of real-time communication technologies, design of the web application's architecture, functional testing focusing on delays, performance optimization, and efficient use of server resources.

Summary of the work: The work analyzes modern technologies and tools for creating WebSocket-based web applications and compares them with traditional data exchange methods (Polling, Long Polling, SSE). The architecture of the web application for two-way communication between the client and server was developed. The web chat functionality was implemented, including message exchange, chat creation, media file integration, a blacklist, and token-based authorization. Performance testing and optimization were carried out, particularly regarding database queries and server resource usage.

Keywords: WebSocket, SocketIO, web application, React, Redux, TypeScript, Node.js, interactive learning, MySQL database, optimization, testi

ЗМІСТ

ВСТУП

Сучасні технології відіграють ключову роль у розвитку освітнього процесу, створюючи нові можливості для інтерактивного навчання. Інтерактивні веб-додатки є важливими інструментами взаємодії між викладачами та студентами, забезпечуючи швидкий обмін інформацією, гнучкість та зручність використання.

Особливо актуальною ця тема стає в умовах зростання ролі дистанційного та змішаного навчання, що потребує надійних цифрових платформ для забезпечення ефективної комунікації між учасниками освітнього процесу.

Об’єктом дослідження є процес інтерактивної взаємодії між студентами та викладачами у веб-додатку на базі WebSocket. Предметом дослідження виступають технології створення інтерактивних веб-додатків з використанням WebSocket для двосторонньої комунікації в реальному часі, орієнтованої на підтримку процесу навчання.

Метою роботи є підвищення ефективності навчального процесу через впровадження інтерактивного веб-додатка на базі WebSocket, що забезпечує високошвидкісну двосторонню взаємодію між студентами та викладачами в режимі реального часу з мінімальними затримками.

Для досягнення поставленої мети визначено такі основні завдання:

1. Провести аналіз сучасних технологій та інструментів для створення веб-додатків із використанням WebSocket.
2. Розробити архітектуру веб-додатка, яка включає клієнтську та серверну частини.
3. Реалізувати функціонал веб-додатка, зокрема:
   - авторизацію користувачів за допомогою токенів;
   - обмін повідомленнями;
   - створення та управління чатами;
   - інтеграцію медіафайлів;
   - управління чорним списком.

4. Провести тестування клієнтської та серверної частин для перевірки їхньої надійності та продуктивності.

5. Оптимізувати запити до бази даних для підвищення продуктивності та забезпечити ефективне використання серверних ресурсів.

Для досягнення поставленої мети розроблено веб-додаток, який покращує взаємодію між викладачами та студентами завдяки функціям миттєвого обміну повідомленнями, інтерактивним груповим чатам, управлінню користувачами (включаючи чорний список) та інтеграції медіафайлів для зручного обміну навчальними матеріалами.

1 АНАЛІЗ ТЕХНОЛОГІЙ ТА ІНСТРУМЕНТІВ ДЛЯ РОЗРОБКИ ВЕБ ДОДАТКІВ НА БАЗІ WEBSOCKET

1.1 Огляд сучасних технологій для створення інтерактивних веб-додатків

Інтерактивні веб-додатки — це веб-застосунки, що забезпечують динамічну взаємодію користувача та системи в реальному часі. Відмінною рисою таких застосунків є можливість виконання дій без необхідності повного перезавантаження сторінки, що значно покращує користувацький досвід, роблячи його плавним та безперервним.

На початкових етапах розвитку інтернету статичні веб-сторінки повністю задовольняли потреби користувачів, оскільки інформація переважно надавалася у вигляді тексту, зображень чи простих гіперпосилань. Проте зі збільшенням кількості користувачів, розширенням доступу до інтернету та зростанням вимог до інтерактивності веб-середовища стало очевидно, що статичних сторінок недостатньо. Люди почали очікувати більшої функціональності, швидшого доступу до інформації та можливості взаємодії з веб-застосунками у режимі реального часу.

Наприклад, раніше для надсилання форми користувачеві потрібно було чекати повного оновлення сторінки, що створювало незручності. Сьогодні інтерактивні веб-додатки вирішують ці проблеми завдяки технологіям, які дозволяють обмінюватися даними з сервером у фоновому режимі, наприклад, через WebSocket або AJAX.

Крім того, інтерактивні веб-додатки стали важливою складовою для багатьох сфер життя: від онлайн-шопінгу та соціальних мереж до навчальних платформ і сервісів для організації роботи. Їхнє поширення обумовлено тим, що сучасні користувачі цінують швидкість, зручність і багатофункціональність.

Таким чином, еволюція від статичних до інтерактивних веб-застосунків відображає не лише технічний прогрес, а й зміни у поведінці користувачів, їхніх очікуваннях та запитах до веб-простору.

До основних характеристик інтерактивних веб додатків входять:

– Динамічна взаємодія в інтерактивних веб-додатках досягається завдяки використанню сучасних технологій, таких як AJAX, WebSocket та інших. Коли використовується AJAX, клієнтський додаток надсилає асинхронний запит до сервера, не перезавантажуючи сторінку. Сервер, у свою чергу, може звернутися до бази даних для отримання або оновлення даних і повернути відповідь клієнту. Наприклад, при зміні статусу замовлення на сайті сервер оновлює інформацію в базі даних і повертає новий статус на клієнтську частину, який відображається без оновлення сторінки (рис. 1.1). WebSocket забезпечує двосторонній постійний зв’язок між клієнтом і сервером. Після встановлення такого з'єднання клієнт може підписуватися на певні події або оновлення від сервера. Наприклад, сервер може в режимі реального часу надсилати повідомлення в чаті або оновлення даних, таких як курс валют, без необхідності повторного запиту з боку клієнта (рис. 1.2). Клієнт, у свою чергу, також може відправляти дані на сервер у будь-який момент, забезпечуючи ефективний обмін інформацією.

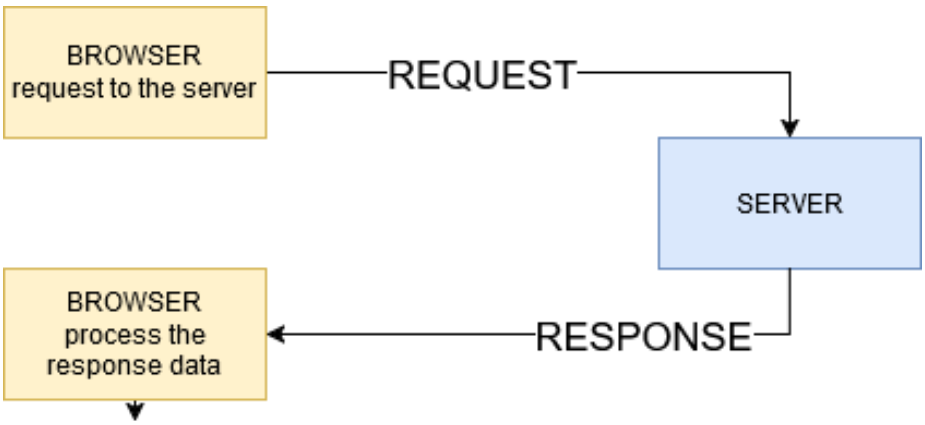


Рисунок 1.1 – Приклад роботи AJAX

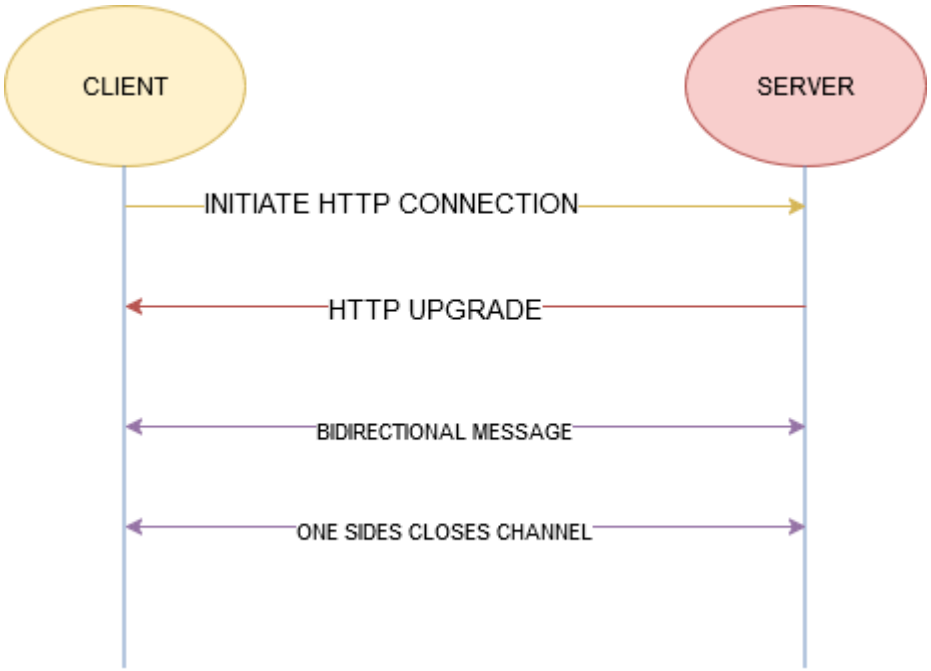


Рисунок 1.2 – Приклад роботи WebSocket

– Реактивність означає, що інтерфейс миттєво реагує на дії користувача, забезпечуючи швидкий і плавний користувацький досвід. Це досягається використанням сучасних бібліотек та фреймворків, таких як React, Angular або Vue.js. Вони дозволяють створювати динамічні компоненти, які оновлюються без перезавантаження сторінки. На прикладі React взаємодія з користувачем реалізується через оновлення так званого віртуального DOM. Коли користувач взаємодіє з додатком, React не оновлює весь інтерфейс, а лише ті його частини, які зазнали змін. Це значно зменшує навантаження на браузер і прискорює роботу додатка. Приклад оновлення стану у React рис. 1.3 [11]. До переваг реактивності можна віднести: швидку взаємодію, оскільки вона відбувається не через перемальовування усього дерева, а через зміну локальних частин дерева, які саме зазнали змін у процесі взаємодії користувача з візуальною частиною веб-додатку. Також це компонентний підхід (рис. 1.4) [12], який дає змогу, розбивати великі частини інтерфейсу на відповідні компоненти, для подальшого їх використання в інших частинах додатку. Це дає змогу зменшити кількість коду, що впливає також на продуктивність веб-додатку.

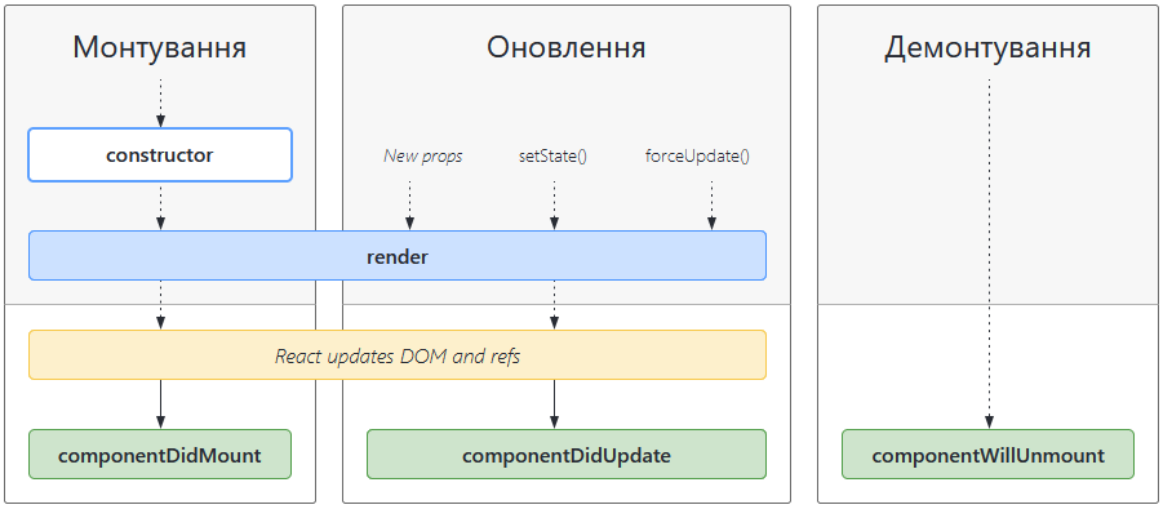


Рисунок 1.3 – Зміна стану у React компонентах

```
1 class Welcome extends React.Component {
2   render() {
3     return <h1>Привіт, {this.props.name}</h1>;
4   }
5 }
```

Рисунок 1.4 – Код приклад компонента на React

– Мультимедійність це важлива складова інтерактивних веб-додатків, яка забезпечує можливість відтворення різних типів контенту, таких як аудіо, відео, зображення, анімації тощо. Вона розширює можливості користувачів, дозволяючи їм слухати музику, переглядати відео або взаємодіяти з візуальними матеріалами безпосередньо у браузері.

Сучасні веб-додатки активно використовують мультимедіа для підвищення зручності й залучення користувачів. Наприклад, навчальні платформи інтегрують відеолекції, аудіокоментарі та інтерактивні візуалізації, щоб полегшити сприйняття матеріалу. У соціальних мережах мультимедійність є основним інструментом для створення та обміну контентом.

Приклади інтерактивних веб-додатків

1. Веб-додатки для обміну повідомленнями, такі як Telegram Web або WhatsApp Web, забезпечують миттєву синхронізацію даних між користувачами.

Повідомлення надсилаються та отримуються в реальному часі, незалежно від пристрою.

2. Наприклад, Agar.io чи Slither.io, де гравці взаємодіють у реальному часі через веб-браузер. Висока швидкість обміну даними забезпечує плавну ігрову динаміку.

3. Інструменти, як Trello або Asana, дозволяють користувачам працювати над спільними проєктами, відстежуючи зміни в завданнях у реальному часі. Наприклад, перетягування карток на дошці моментально відображається у всіх учасників проєкту.

4. Сервіси на кшталт Zoom, Google Meet або інтерактивні навчальні платформи з тестами та завданнями дають змогу вчитися в реальному часі. Викладачі та учні можуть обмінюватися матеріалами, відповідати на запитання або проводити опитування без затримок.

5. Онлайн-платформи для моніторингу фінансів або трейдингу, наприклад, Binance чи Robinhood, дозволяють користувачам отримувати дані про курси валют, акцій або криптовалют у реальному часі, що є критичним для прийняття швидких рішень.

6. Такі сервіси, як Google Maps або Waze, надають можливість не лише планувати маршрути, а й отримувати актуальні дані про затори, аварії чи ремонти доріг. Ця інформація автоматично оновлюється для всіх користувачів.

Базові технології для побудови інтерактивних веб-додатків:

– HTML – структура веб-сторінки.

– CSS – оформлення зовнішнього вигляду сторінки.

– JavaScript (JS) – інтерактивність та динаміка на сторінці.

– Node.js – серверна платформа для запуску JavaScript на сервері.

– Express – фреймворк для створення серверів на Node.js.

– AJAX- комунікація із сервером через запити зі сторони клієнта.

– WebSocket – протокол для постійного двостороннього з'єднання між сервером і клієнтом, що забезпечує передачу даних у реальному часі.

1.2 Особливості протоколу WebSocket та його переваги

Основною відмінністю між протоколом WebSocket та традиційними HTTP-запитами є можливість встановлення двостороннього зв'язку в реальному часі через постійно активне з'єднання, яке ініціюється за допомогою HTTP-запиту з заголовком Upgrade. Це дозволяє серверу надсилати дані на клієнта без необхідності кожного разу чекати запиту. У порівнянні з HTTP, де кожен запит викликає відповідь (одностороння комунікація), WebSocket підтримує постійний зв'язок, що є критичним для таких додатків, як онлайн-ігри, чат-системи, фінансові інструменти тощо.

Процес встановлення з'єднання WebSocket:

1. Клієнт відправляє HTTP-запит на сервер із заголовками, що вказують на бажання встановити WebSocket-з'єднання. Це запит типу "Upgrade".
2. Сервер перевіряє запит і, якщо підтримує WebSocket, відповідає позитивно (HTTP 101 — Switching Protocols), підтверджуючи перехід на WebSocket-протокол. Якщо сервер не підтримує або відмовляється, з'єднання не встановлюється.
3. Після отримання позитивної відповіді від сервера клієнт та сервер переходять на двостороннє з'єднання через WebSocket. З цього моменту вони можуть надсилати повідомлення один одному без необхідності додаткових запитів або відповіді.

Таким чином, WebSocket дозволяє забезпечити інтерактивність в реальному часі, підтримуючи постійне з'єднання між клієнтом та сервером. Це зручно для додатків, де необхідна миттєва передача даних або оновлень, таких як чат, ігри чи біржові додатки.

Основні переваги WebSocket:

- Завдяки постійному з'єднанню, WebSocket дозволяє значно знизити час затримки між запитом і відповіддю, оскільки не потрібно повторно встановлювати з'єднання для кожного обміну даними.

- WebSocket підтримує двосторонній зв'язок, що дозволяє серверу ініціювати відправку даних на клієнта без запиту з його боку, забезпечуючи миттєву реакцію на події.
- Завдяки відсутності необхідності постійно ініціювати нові з'єднання, WebSocket знижує навантаження на сервер і клієнтську частину, що робить додатки більш ефективними у використанні ресурсів.
- WebSocket чудово підходить для масштабованих додатків, таких як онлайн-ігри чи чати, де потрібно одночасно обробляти багато з'єднань і передавати дані в реальному часі.
- Завдяки високій швидкості з'єднання та можливості обробки даних у реальному часі, WebSocket застосовують у багатьох сферах, від фінансових додатків до систем моніторингу.

Якщо порівнювати WebSocket з іншими методами, такими як Server-Sent Events (SSE), polling та long polling, то основні відмінності можна побачити у способах комунікації та ефективності.

Server-Sent Events (SSE) — це технологія, яка дозволяє здійснювати передачу даних від сервера до клієнта через стандартний HTTP-протокол, що є корисним для додатків, які потребують отримання оновлень у реальному часі. SSE є односпрямованим механізмом комунікації, що означає, що дані можуть передаватися лише від сервера до клієнта. На відміну від WebSocket, який забезпечує двосторонню комунікацію, SSE дозволяє лише серверу надсилати інформацію користувачеві без можливості для клієнта відправляти дані назад через той самий канал зв'язку.

Технологія SSE, що ґрунтується на використанні HTTP-протоколу, дозволяє ефективно передавати невеликі потоки даних на клієнтські пристрої без необхідності періодичних запитів від клієнта до сервера, що дозволяє знизити навантаження на сервер. Це особливо важливо для додатків, де потрібно регулярно отримувати нові дані або оновлення, наприклад, для новин, соціальних мереж або повідомлень, де зміст оновлюється часто.

Один із основних аспектів SSE полягає в тому, що під час підключення клієнт отримує потік даних, який не переривається до тих пір, поки зв'язок не буде завершено з обох сторін або не відбудеться відключення через неполадки в мережі. Таким чином, сервер може безперервно передавати дані клієнту, і клієнт отримує оновлення в реальному часі, не потребуючи активних запитів чи перезавантаження сторінки.

Однією з ключових переваг SSE є простота налаштування. Для того, щоб організувати підключення SSE, необхідно тільки встановити спеціальний заголовок на сервері для передачі даних у форматі "text/event-stream". Крім того, клієнт просто підключається до цього потоку через JavaScript, використовуючи вбудований об'єкт EventSource. Це значно спрощує процес інтеграції цієї технології у веб-додатки.

Однак, існують і певні обмеження SSE, які необхідно враховувати під час вибору цієї технології. Як було зазначено, SSE підтримує тільки односпрямовану комунікацію, тобто клієнт не може відправляти дані назад через цей канал (рис. 1.5). Якщо додаток вимагає двостороннього зв'язку, наприклад, для чатів, онлайн-ігор або додатків, що потребують інтерактивної комунікації, більш підходящим варіантом буде використання WebSocket. WebSocket дозволяє встановити постійне з'єднання між клієнтом і сервером, через яке обидві сторони можуть надсилати дані в будь-який час.

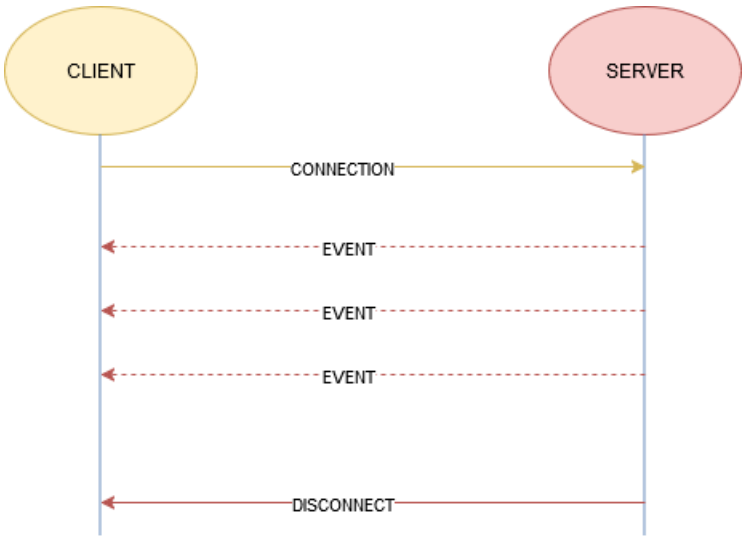


Рисунок 1.5 – Принцип роботи Server-Sent Events

Ще однією важливою особливістю SSE є обмеження на кількість одночасних з'єднань, яке може бути встановлено для кожного клієнта. Сервери можуть обмежити кількість відкритих потоків на клієнта через певні конфігурації, що може призвести до труднощів в обробці великої кількості клієнтів одночасно. У таких випадках варто розглянути використання інших технологій, таких як WebSocket, які можуть більш ефективно справлятися з великими обсягами трафіку і потребами в двосторонньому зв'язку.

Також слід зазначити, що, оскільки SSE використовує стандартний HTTP-протокол, він має хорошу сумісність із більшістю веб-браузерів і може використовувати існуючі механізми інфраструктури, такі як балансувальники навантаження або кешування HTTP-запитів. Це робить технологію більш простою в інтеграції з існуючими системами та забезпечує зручність в адмініструванні.

Серед переваг використання SSE можна виділити низьке навантаження на сервер і можливість легко інтегрувати потоки даних у веб-додатки без значних змін у серверній частині. Однак для більш складних застосунків, де важливе постійне двостороннє спілкування або потреба в низьких затримках передачі даних, краще використовувати технології на кшталт WebSocket, які можуть забезпечити більшу гнучкість у роботі з клієнтами.

Висновуючи, можна сказати, що Server-Sent Events є зручним і простим інструментом для створення додатків, що вимагають односпрямованої передачі даних у реальному часі, таких як новинні стрічки або системи сповіщень. Однак для складніших додатків, що потребують більш інтерактивної комунікації між клієнтом і сервером, варто звернутися до технологій, що підтримують двостороннє спілкування, таких як WebSocket.

Polling це метод, при якому клієнт періодично відправляє запити на сервер для отримання оновлень, що дозволяє зібрати нові дані після кожного запиту. Хоча цей спосіб має простоту в реалізації, його недоліки проявляються у значному навантаженні на сервер. З кожним новим запитом створюється окреме з'єднання з сервером, що збільшує використання ресурсів і може призвести до перевантаження при високій кількості одночасних підключень.

Цей метод також має більші затримки, оскільки дані приходять тільки після кожного нового запиту, що робить його менш придатним для задач, де необхідне оперативне оновлення інформації, як у реальному часі. Для прикладу, у випадку веб-чату або ігрових додатків, де взаємодія між клієнтом і сервером має бути безперервною і миттєвою, polling може бути занадто повільним та неефективним порівняно з іншими технологіями, такими як WebSocket або Server-Sent Events.

Крім того, в процесі polling відбувається постійне створення нових з'єднань, що може призводити до високих витрат на обробку запитів (рис. 1.6). Сервер кожен раз має обробляти нове підключення і відповідно до цього здійснювати додаткові операції на обробку запитів, навіть якщо зміни на сервері відсутні. Таким чином, це створює значне навантаження на інфраструктуру сервера, зокрема при високих частотах запитів або великій кількості користувачів, що може призвести до затримок у відповідях.

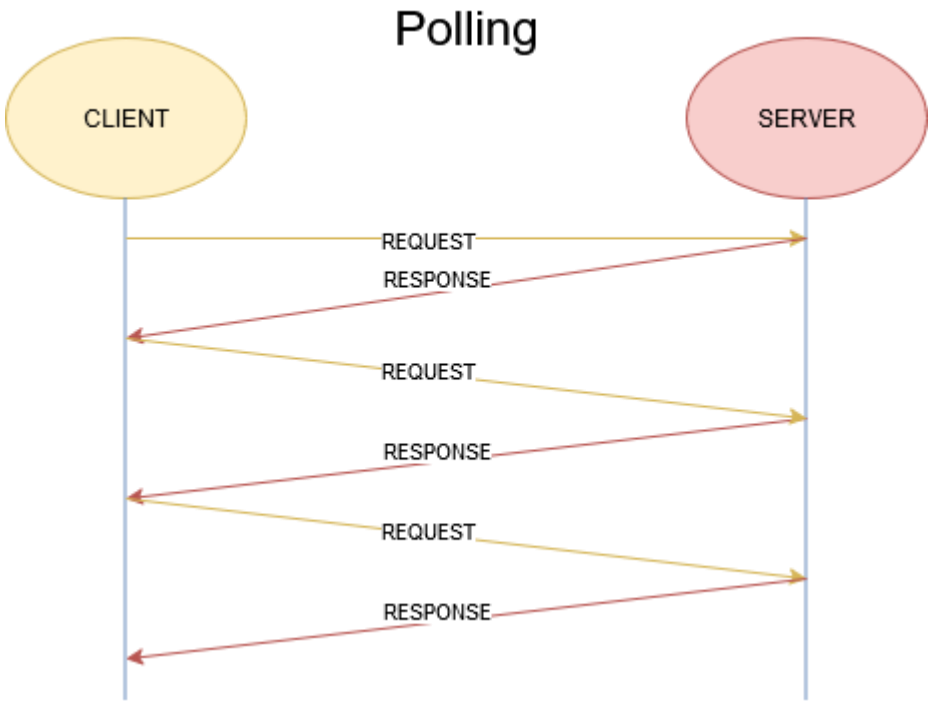


Рисунок 1.6 – Принцип роботи Polling

Хоча polling є доступним варіантом для простих сценаріїв, де немає потреби у постійному оновленні даних, для додатків, що потребують миттєвої реакції або обміну інформацією в реальному часі, використання polling може стати

неефективним. Тому для таких випадків зазвичай рекомендують використовувати більш сучасні технології, які дозволяють встановлювати постійне з'єднання між клієнтом і сервером, зменшуючи затримки і навантаження на сервер.

Long polling є вдосконаленою версією класичного polling, де клієнт відправляє запит на сервер і потім чекає відповіді. Однак, на відміну від звичайного polling, у разі відсутності нових даних сервер не відповідає негайно. Клієнт чекає, поки на сервері з'являться нові дані, і лише після цього отримує відповідь, що містить актуальну інформацію. Це дозволяє зменшити кількість запитів, коли нові дані не з'являються, і таким чином знизити навантаження на сервер.

Після того, як сервер надсилає відповідь клієнту, з'єднання закривається, і клієнт знову ініціює новий запит. Тобто, після кожної відповіді сервер і клієнт відновлюють процес, чекаючи нових оновлень. Така схема значно зменшує кількість запитів, оскільки запит буде відправлений лише тоді, коли є нові дані, а не через певні інтервали часу, як у класичному polling.

Проте, навіть із цією оптимізацією long polling все ще має певні недоліки. Одним із основних є високі затримки, оскільки клієнт чекає відповіді від сервера, що може бути неприязним для сценаріїв, де важлива миттєва реакція (рис. 1.7). Крім того, хоча сервер не відповідає на запити, коли немає нових даних, він все одно мусить утримувати з'єднання з клієнтом, поки чекає нових даних. Це вимагає додаткових ресурсів для обробки відкритих з'єднань, що збільшує навантаження на сервер, особливо при великій кількості одночасних користувачів.

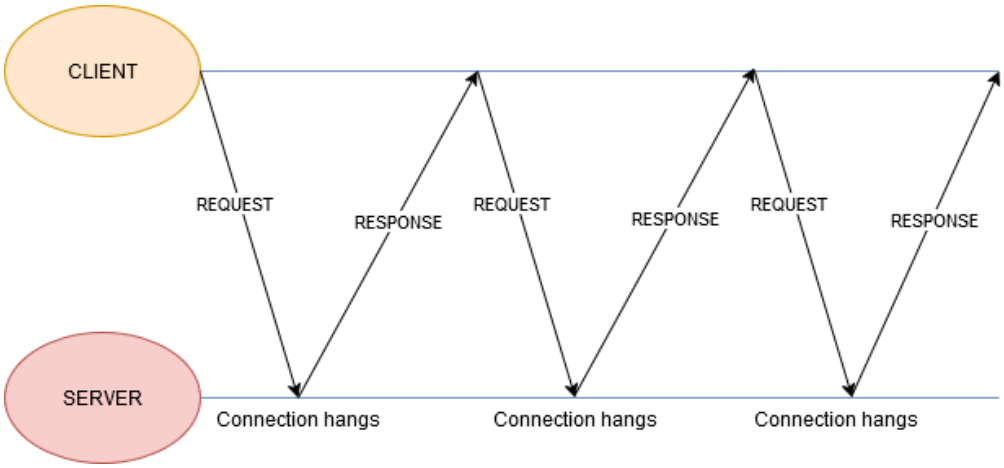


Рисунок 1.7– Принцип роботи Long Polling

Попри ці недоліки, long polling залишається кращим варіантом для сценаріїв, де необхідно отримувати дані в реальному часі, але де неможливо або неефективно використовувати постійно відкрите з'єднання, як це реалізовано в WebSocket. Цей метод оптимізує навантаження на сервер, але все одно не досягає такої ефективності, як більш сучасні технології для обміну даними в реальному часі.

На відміну від інших методів, WebSocket дозволяє встановити постійне двостороннє з'єднання між клієнтом і сервером. Це суттєво знижує затримку передачі даних і дозволяє серверу надсилати повідомлення клієнту в реальному часі без необхідності ініціювати нові запити чи чекати на них. Завдяки постійному з'єднанню та здатності до двостороннього обміну даними, WebSocket є значно ефективнішим для додатків, які потребують миттєвої взаємодії з мінімальними затримками, таких як чати, онлайн-ігри або системи сповіщень.

Водночас вибір між підходами, як-от polling, long polling, server-sent events і WebSocket, залежить від конкретного контексту використання й обмежень інфраструктури. Кожен з методів має свою область застосування. Наприклад, polling і long polling чудово підходять для більш простих додатків, де затримки не є настільки критичними, і де інфраструктура не дозволяє підтримувати складніші з'єднання. Однак ці підходи втрачають ефективність при високих навантаженнях. З іншого боку, server-sent events — це оптимальне рішення для додатків, які потребують лише одностороннього потоку даних від сервера до клієнта. Вони ідеально підходять для додатків, де важливо регулярно оновлювати інформацію на стороні клієнта, але зворотний зв'язок не є необхідним. Проте їхні можливості обмежені порівняно з WebSocket, що надає більшу гнучкість і можливості для інтерактивного обміну даними.

Зрештою, вибір методу залежить від специфіки проекту та технічних вимог. Вибір між polling, long polling, server-sent events та WebSocket визначається кількістю одночасних з'єднань, типом даних, що передаються, вимогами до затримок, а також можливостями серверної інфраструктури.

1.3 Аналіз стеку технологій для реалізації веб-додатка на базі WebSocket

Для розробки веб-додатка на основі WebSocket я вибрав сучасний стек технологій, що включає такі інструменти, як React, Redux, TypeScript, Tailwind CSS, Socket.IO для фронтенду та Node.js, Express.js, MySQL для бекенд-частини. Цей набір технологій дозволяє створити високопродуктивний, масштабований та інтерактивний веб-додаток, здатний ефективно працювати з великими обсягами даних і забезпечувати миттєву взаємодію з користувачами.

React є основним інструментом для побудови користувацького інтерфейсу. Він дозволяє створювати динамічні, взаємодіючі компоненти, які можуть окремо оновлюватися без необхідності перезавантаження всієї сторінки. Це значно покращує ефективність додатка та користувацький досвід, оскільки дозволяє мінімізувати затримки і витрати на перезавантаження. Завдяки компонентній архітектурі, React також спрощує розробку масштабованих додатків, де кожен компонент може бути окремо тестований та підтримуваний.

TypeScript в проекті додає додатковий рівень надійності завдяки строгій типізації. Це дозволяє виявляти помилки на етапі розробки, що значно знижує кількість багів і покращує стабільність додатка. Точні типи також сприяють зручності підтримки коду, що є важливим для довгострокових проектів.

Redux обирається для централізованого управління станом додатка, що дозволяє зберігати дані в глобальному сховищі. Це значно полегшує взаємодію між компонентами додатка, оскільки всі зміни стану управляються через одне джерело істини, що спрощує управління даними та синхронізацію між різними частинами додатка.

Tailwind CSS є ще одним важливим інструментом, який значно полегшує створення стильного та адаптивного інтерфейсу. Завдяки утилітним класам Tailwind CSS дозволяє швидко створювати стильові елементи, без необхідності писати великі обсяги CSS-коду. Це дозволяє зберігати проект чистим і лаконічним, забезпечуючи при цьому високу гнучкість у налаштуванні вигляду інтерфейсу.

Socket.IO на фронтенді дозволяє організувати двостороннє з’єднання між клієнтом і сервером, що є ключовим для додатків, де потрібна миттєва взаємодія в реальному часі, наприклад, для чатів або онлайн-ігор. Замість того, щоб постійно відправляти HTTP-запити, як у традиційних веб-додатках, Socket.IO використовує постійне з’єднання між клієнтом і сервером, що дозволяє миттєво обмінюватися даними без затримок.

На серверній стороні Node.js є оптимальним вибором завдяки своїй асинхронній природі, що дозволяє ефективно обробляти велику кількість одночасних з’єднань. Завдяки такій архітектурі, Node.js здатний забезпечити високу продуктивність і знизити час очікування запитів. Це особливо важливо для додатків в реальному часі, де потрібно оперативно обробляти велику кількість одночасних з’єднань і запитів.

Express.js додає зручність в організації серверної частини, дозволяючи легко реалізувати RESTful API для обробки запитів. Всі операції, що потребують взаємодії з базою даних, а також процеси авторизації та реєстрації, можуть бути легко реалізовані через цей фреймворк, що значно прискорює розробку.

Для зберігання даних використовується MySQL, оскільки ця реляційна база даних добре підходить для зберігання структурованих даних, таких як інформація про користувачів або історії чатів. MySQL забезпечує високу продуктивність і здатна швидко обробляти складні запити, що важливо для забезпечення швидкої роботи додатка, навіть при великій кількості користувачів.

Цей технологічний стек дозволяє ефективно поєднувати високі вимоги до продуктивності, масштабованості та інтерактивності, що є критичним для створення успішних веб-додатків. Використання WebSocket для реалізації двосторонньої взаємодії в реальному часі дозволяє забезпечити миттєвий обмін даними між користувачем та сервером без необхідності постійно ініціювати нові запити, що значно зменшує затримки і покращує швидкість реакції додатка. Завдяки цьому, користувачі можуть безперешкодно взаємодіяти з платформою, що є важливим для онлайн-курсів, чат-систем, відеоконференцій та інших інтерактивних освітніх функцій.

2 РОЗРОБКА АРХІТЕКТУРИ ВЕБ-ДОДАТКА

2.1 Проектування клієнтської частини веб-додатка

Клієнтська частина веб-додатка є основою інтерактивності та зручності користування системою. Вона відповідає за відображення даних, обробку дій користувача та інтеграцію з серверною частиною для обміну інформацією в реальному часі.

2.1.1 Використання React, Redux, TypeScript

Оскільки на клієнті використовується React, розробка здійснюється із застосуванням компонентного підходу, що передбачає створення клієнтської частини як сукупності незалежних і функціональних блоків. Такий підхід дозволяє значно спростити процес розробки, адже компоненти можуть бути перевикористані в різних частинах додатка, що зменшує кількість дубльованого коду та підвищує його підтримуваність.

Ключова перевага компонентів полягає у можливості динамічного оновлення їхнього стану (рис. 2.1). За допомогою React функцій або ж через інтеграцію Redux, стан компонентів можна змінювати відповідно до дій користувача чи інших умов. Це забезпечує гнучкість у побудові інтерфейсу та підтримці актуальності даних у реальному часі. Наприклад, якщо змінюється стан одного компонента, це автоматично оновлює відповідну частину інтерфейсу без необхідності перезавантажувати сторінку. Таким чином, компонентний підхід у поєднанні з інструментами управління станом, як-от Redux, значно підвищує ефективність розробки клієнтської частини веб-додатка.

```tsx
import React, { useState } from 'react';

type Item = {
  id: number;
  name: string;
  description: string;
};

const DynamicListComponent = () => {
  const [items, setItems] = useState<Item[]>([]);
  const [newItem, setNewItem] = useState<string>('');
  const [newDescription, setNewDescription] = useState<string>('');

  const handleAddItem = () => {
    if (newItem && newDescription) {
      setItems([...items, { id: Date.now(), name: newItem, description: newDescription }]);
      setNewItem('');
      setNewDescription('');
    }
  };

  const handleDeleteItem = (id: number) => {
    setItems(items.filter(item => item.id !== id));
  };

  return (
    <div>
      <input
        type="text"
        placeholder="Item Name"
        value={newItem}
        onChange={(e) => setNewItem(e.target.value)}
      />
      <input
        type="text"
        placeholder="Item Description"
        value={newDescription}
        onChange={(e) => setNewDescription(e.target.value)}
      />
      <button onClick={handleAddItem}>Add Item</button>

```

Рисунок 2.1 – Код компоненту react з динамічним станом

Для управління станом у веб-додатку застосовується Redux — ефективний інструмент, який забезпечує централізоване сховище даних. Завдяки цьому всі компоненти програми отримують доступ до єдиного джерела інформації, що значно спрощує взаємодію між ними та забезпечує контроль над змінами стану. Глобальне сховище організоване за допомогою спеціальних функцій, які називаються ред'юсерами (рис. 2.2). Вони відповідають за визначення того, як змінюється стан додатка у відповідь на певні дії користувача або системи.

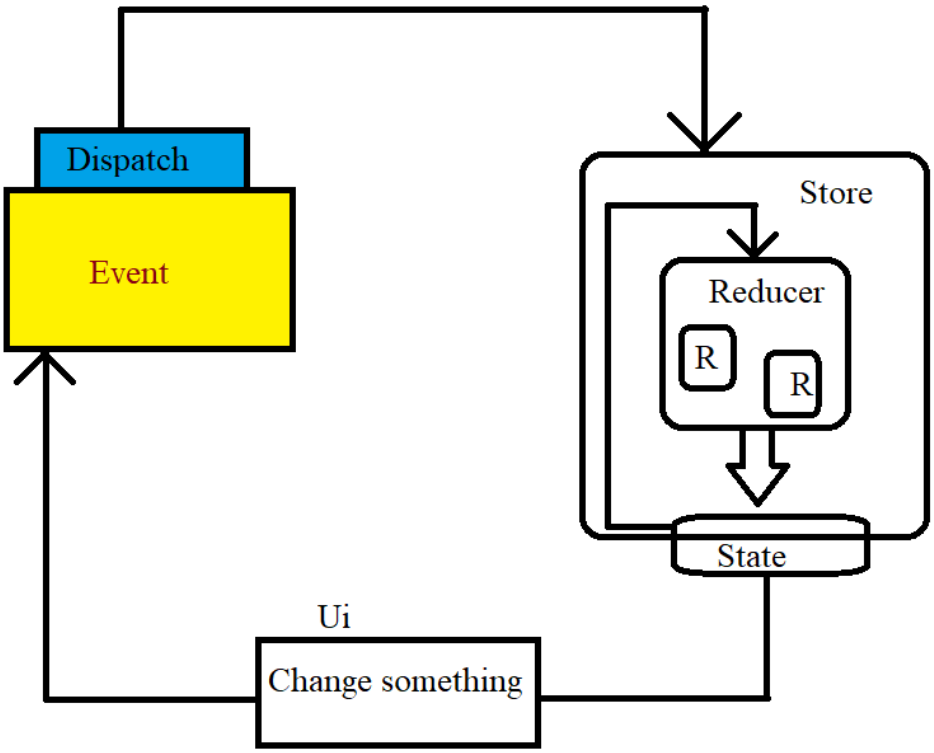


Рисунок 2.2 – Глобальне сховище з reducers

Однією з важливих особливостей Redux є можливість використовувати зрізи (slices) — компактні модулі, які об'єднують логіку роботи зі станом у зручну структуру. У межах цих зрізів зберігаються дії (actions), що виконують роль тригерів змін. Кожна дія представлена об'єктом, який містить тип події та, за необхідності, додаткову інформацію (рис. 2.3). Такий підхід дозволяє організувати роботу з даними у прозорий і зрозумілий спосіб, полегшуючи тестування, підтримку та масштабування додатка.

Рисунок 2.3 – Приклад action

Централізоване управління станом через Redux забезпечує узгодженість у роботі всієї системи. Завдяки єдиній точці доступу всі компоненти взаємодіють між собою без дублювання логіки або конфліктів у станах, що робить додаток більш стабільним і надійним.

Для забезпечення коректної роботи з даними у додатку використовується TypeScript, який дозволяє додати типізацію до кожного елемента коду. Завдяки цьому можна точно визначити, який тип даних очікує кожна частина програми, що мінімізує помилки під час виконання. TypeScript легко інтегрується як із бібліотекою React, так і з глобальним сховищем Redux, забезпечуючи можливість типізувати їх функції та структури.

У Redux типізація виконується через визначення типів для спеціальних хуків, таких як useDispatch і useSelector (рис. 2.4). Це робить ці хуки універсальними та готовими до використання у будь-якій частині програми. Вони не лише автоматично пов'язані з глобальним сховищем, але й забезпечують безпечну роботу з даними завдяки своїй типізованості. Такий підхід спрощує процес роботи з Redux, полегшує написання коду та підвищує його якість, роблячи додаток більш надійним та зрозумілим для розробників.

```
1 import { useDispatch, useSelector } from 'react-redux'
2 import type { TypedUseSelectorHook } from 'react-redux'
3 import type { RootState, AppDispatch } from '../redux/store'
4
5 // Use throughout your app instead of plain `useDispatch` and `useSelector`
6 export const useAppDispatch: () => AppDispatch = useDispatch
7 export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector
8
9
```

Рисунок 2.4 – Код типізації Redux хуків за допомогою TypeScript

2.1.2 Інтеграція Tailwind CSS та SCSS для стилізації

Для забезпечення зручного та ефективного стилювання в проекті було обрано Tailwind CSS, популярну утилітарну CSS-бібліотеку, яка дозволяє швидко створювати привабливі та адаптивні інтерфейси без необхідності писати складні стилі. Tailwind CSS працює на принципі використання класів, кожен з яких виконує конкретне завдання, таке як визначення кольору фону, шрифтів, розмірів елементів, відступів та інших властивостей. Цей підхід значно спрощує процес розробки, оскільки можна працювати безпосередньо в HTML-коді, використовуючи інтуїтивно зрозумілі класи для стилізації елементів.

Наприклад, для того щоб змінити фон елемента на синій, достатньо застосувати клас bg-blue-500, що відповідає конкретному відтінку синього кольору в бібліотеці. Якщо потрібно змінити колір тексту на білий, використовується клас text-white. Такий підхід дозволяє не лише значно пришвидшити процес стилізації, а й зберігати код максимально чистим, оскільки весь необхідний CSS застосовується через класи без потреби писати окремі правила у файлі стилів.

Tailwind CSS також надає великий потенціал для налаштування стилів відповідно до вимог проекту. Бібліотека дозволяє розширювати базові стилі, додаючи власні класи або коригуючи наявні значення, що дає більше гнучкості при роботі з проектом. Це можливо завдяки конфігураційному файлу tailwind.config.js (рис. 2.5), де можна визначити власні параметри для кольорів, розмірів шрифтів, відступів, а також налаштувати змінні для відображення елементів у різних темах або для адаптації до різних екранів.

```
/** @type {import('tailwindcss').Config} */
export default {
  content: ["./index.html", "./src/**/*.{js,ts,jsx,tsx}"],
  theme: {
    container: {
      center: true,
      padding: "1rem",
      screens: {
        xs: "100%",
        sm: "640px",
        md: "768px",
        lg: "1024px",
        xl: "1280px",
        "2xl": "1750px",
      },
    },

    extend: {
      fontFamily: {
        robo: "Roboto",
      },
```

Рисунок 2.5 – Налаштування Tailwind CSS

Наприклад, у конфігураційному файлі можна додати нові кольори або змінні, що відповідають фірмовим вимогам проекту, або змінити наявні значення для використання в усіх компонентах. Це дозволяє адаптувати дизайн і забезпечити уніфікований вигляд всього інтерфейсу, при цьому мінімізуючи потребу в переписуванні стилів у кожному окремому файлі.

Tailwind CSS також є дуже зручним для адаптивного дизайну. За допомогою утиліт можна швидко налаштовувати елементи для відображення на різних розмірах екранів, що особливо важливо для сучасних веб-додатків. Наприклад, класи на кшталт sm:w-full або md:px-8 дозволяють встановити різні параметри для мобільних пристроїв або десктопів, що забезпечує зручність використання на різних пристроях без потреби писати окремі медіа-запити.

Таким чином, використання Tailwind CSS в проекті дозволяє не тільки спростити процес стилізації, але й зробити його більш гнучким і масштабованим,

що особливо важливо в умовах розвитку великих веб-додатків. Це також дає змогу швидко змінювати вигляд елементів, не вносячи значних змін у загальну структуру проекту, та дає змогу легко підтримувати і модифікувати дизайн у майбутньому, оскільки всі стилі чітко структуровані і легко змінювані через конфігураційний файл.

Цей підхід дає змогу ефективно керувати проектом і знижує кількість дублювання коду, що позитивно впливає на підтримуваність та масштабованість проекту. Така система дозволяє розробникам швидко адаптуватися до змін у вимогах дизайну або функціональності, без необхідності кожного разу починати з нуля.

У поєднанні з SCSS Tailwind дозволяє ще більше гнучкості. SCSS дає можливість створювати вкладені стилі, що зручніше для написання більш складних компонентів (рис. 2.6). Вкладені стилі дозволяють краще організувати CSS-код, роблячи його чистішим і легшим для розуміння. Наприклад, при використанні SCSS можна структурувати стилі таким чином, щоб вони не повторювались і могли бути легше підтримувані в майбутньому. Крім того, використання SCSS з Tailwind дозволяє зберігати всі переваги обох інструментів, створюючи більш зручний процес стилізації та розвитку інтерфейсу.

```scss
nav {
  ul {
    margin: 0;
    padding: 0;

    li {
      list-style: none;
      a {
        text-decoration: none;
        color: $primary-color;
      }
    }
  }
}
```

Рисунок 2.6 – Синтаксис коду SCSS

2.2 Проектування серверної частини веб-додатка

2.2.1 Використання Node.js, Express та SocketIO

На сервері та клієнті в проекті використовуються принципи архітектури MVC, що розшифровується як модель-представлення-контролер. Ця архітектура є однією з найпоширеніших у програмній розробці, оскільки дозволяє чітко розділити програмну логіку на три основні компоненти: модель, представлення та контролер. Такий підхід забезпечує високу модульність, що полегшує розробку, тестування та подальше обслуговування програмного забезпечення.

Модель відповідає за дані та бізнес-логіку програми, включаючи роботу з базами даних, валідацію введених даних та обробку запитів. Представлення (View) використовується для відображення цих даних користувачу у вигляді веб-сторінок, форм або інших інтерфейсів. Воно отримує дані з моделі і відображає їх у зручному для користувача форматі, тим самим забезпечуючи зрозуміле та ефективне взаємодія з користувачем. Контролер же виступає як посередник між користувачем і системою, обробляючи запити, виконуючи відповідні дії, зокрема взаємодіючи з моделлю для отримання чи оновлення даних, і передаючи результат для подальшого відображення у вигляді.

Застосування MVC дозволяє розробникам працювати з окремими компонентами програми більш ефективно (рис. 2.7). Цей підхід зменшує залежності між компонентами, що в свою чергу спрощує їх модифікацію та обслуговування. Крім того, така архітектура дозволяє досягти високої гнучкості та масштабованості системи, оскільки кожен компонент може бути змінений незалежно від інших, що позитивно впливає на підтримку та подальший розвиток програмного забезпечення.

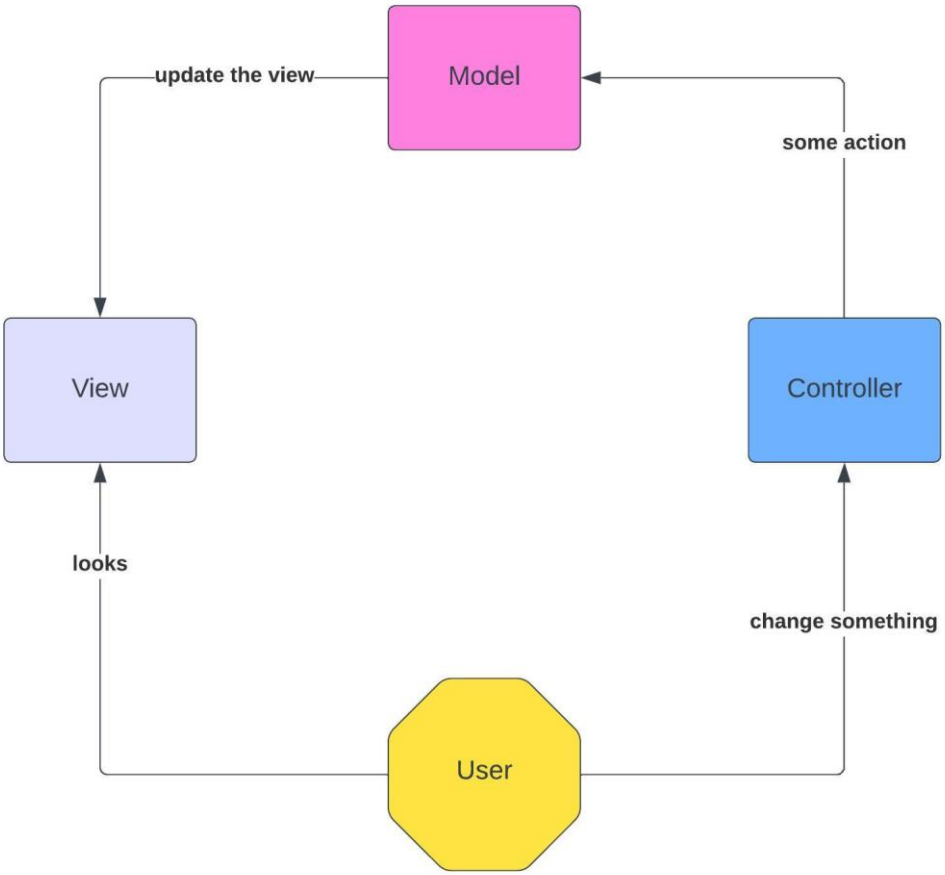


Рисунок 2.7 – MVC архітектура

Express фреймворк, який дозволяє швидко та гнучко будувати серверні додатки. Завдяки йому розробники можуть створювати потужні веб-застосунки та API, використовуючи мінімум коду. Express надає безліч корисних функцій, що полегшують розробку. Наприклад, він підтримує маршрутизацію, обробку запитів, інтеграцію з базами даних, а також має вбудовану підтримку шаблонів для рендерингу сторінок. Завдяки своїй простоті та масштабованості, Express став одним із найбільш використовуваних фреймворків для серверної частини на платформі Node.js.

Крім того, Express має значну кількість налаштувань, що дозволяють адаптувати його під різні потреби проекту. Розробники можуть налаштовувати різні параметри, що дозволяє значно підвищити продуктивність та гнучкість додатка, а також забезпечити більш зручну роботу з middleware, які є проміжними програмними компонентами, які можуть обробляти запити на різних етапах їх

проходження (рис. 2.8). Таким чином, Express є не тільки потужним інструментом для створення серверних додатків, але й дуже гнучким у налаштуванні під індивідуальні потреби кожного проекту (рис. 2.9).

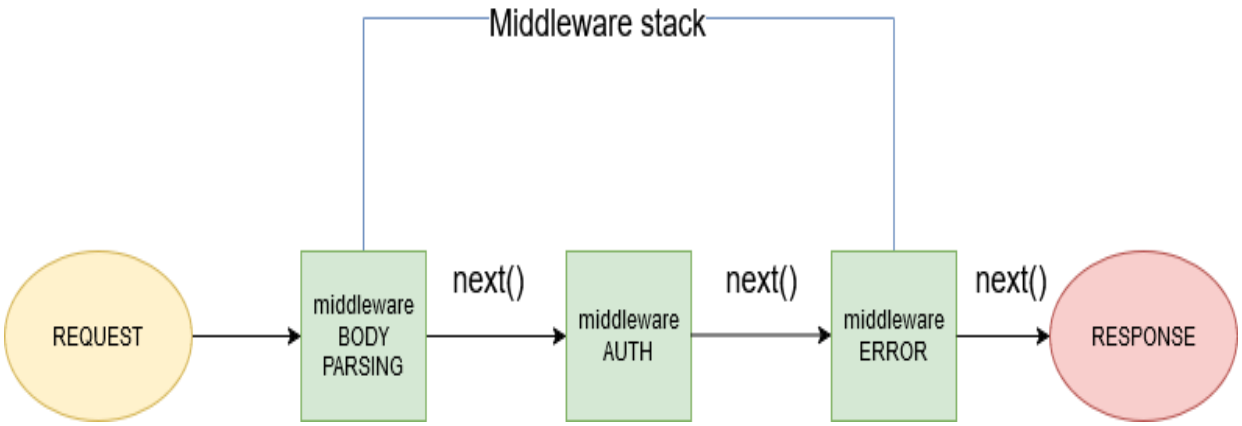


Рисунок 2.8 – Middleware stack

```
app.use(express.json({ extended: true }));
app.use(express.urlencoded({ extended: true }));
app.use("/uploads", express.static(path.join(__dirname, "uploads")));
app.use(cookieParser());
app.use(
  cors({
    credentials: true,
    origin: process.env.CLIENT_URL,
  })
);
async function startApp() {
  try {
    httpServer.listen(PORT, () =>
      console.log("SERVER STARTED ON PORT " + PORT)
    );
  } catch (error) {
    console.log(error);
  }
}
```

Рисунок 2.9 – Код express налаштування

SocketIO є потужною бібліотекою для реалізації зв'язку в реальному часі між клієнтом та сервером, і в даному випадку вона є основним інструментом для забезпечення обміну повідомленнями в веб-додатку. SocketIO побудована на базі

протоколу WebSocket, але має більше функціональних можливостей і значно спрощує роботу з реальним часом. Ця бібліотека дозволяє створювати стабільні, двосторонні з'єднання між сервером та клієнтом, забезпечуючи обмін даними без необхідності постійних запитів від клієнта.

SocketIO підтримує не лише WebSocket, але й інші технології, такі як long polling, що дозволяє забезпечити зв'язок навіть у випадках, коли WebSocket-з'єднання неможливо через обмеження мережі або брандмауери. Вона автоматично вибирає найкращий доступний метод для з'єднання, що робить її універсальним рішенням для більшості випадків.

Однією з ключових переваг використання SocketIO є її можливість підтримувати постійне двостороннє з'єднання між клієнтом і сервером. Це дозволяє серверу відправляти дані на клієнт без необхідності клієнту ініціювати запит, що особливо важливо для додатків, які працюють в реальному часі, таких як чат-програми або онлайн-ігри. Це забезпечує миттєвий обмін повідомленнями між користувачами без затримок.

Крім того, SocketIO надає готові інструменти для керування подіями, зручний API для роботи з повідомленнями, а також вбудовану підтримку кімнат і просторів, що дає можливість організовувати групові чати або приватні канали для передачі повідомлень між певними групами користувачів (рис. 2.10). Ці функції спрощують реалізацію складних сценаріїв комунікації, таких як повідомлення в групах або особисті чати.

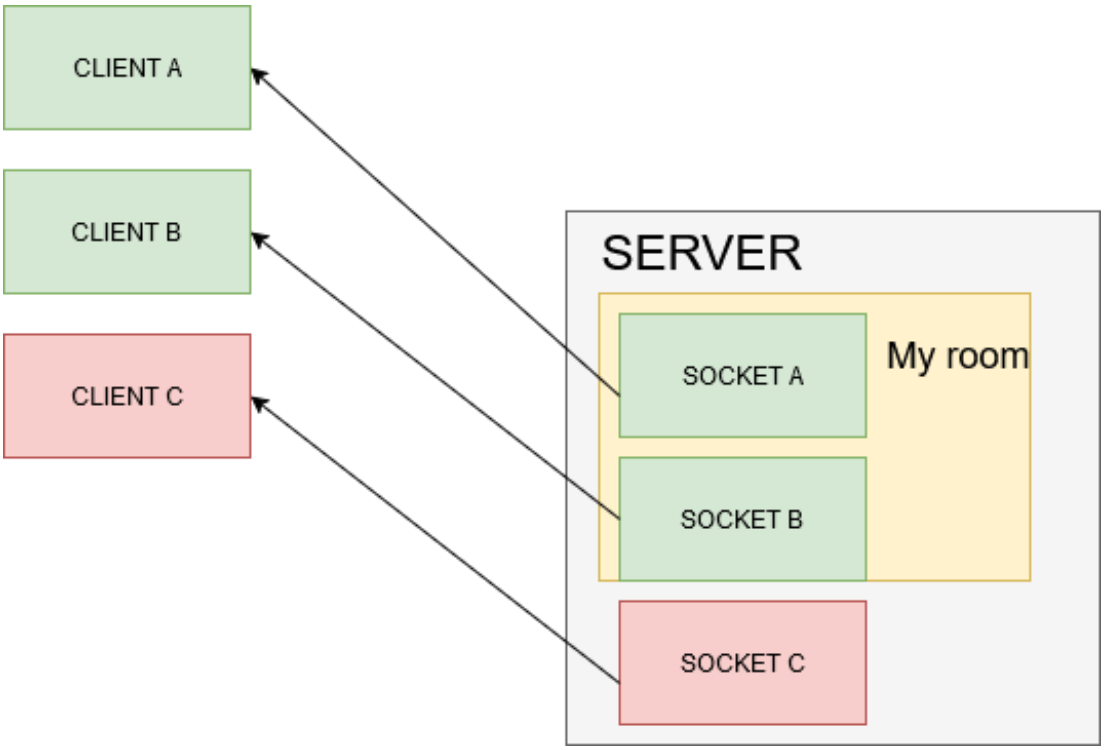


Рисунок 2.10 – Принцип роботи кімнат у SocketIO

Завдяки цим характеристикам, Socket.IO є дуже популярним інструментом для розробки додатків, що потребують обміну даними в реальному часі. Вона спрощує реалізацію таких функцій, як чати, сповіщення, оновлення даних на сторінці без перезавантаження та інші сценарії взаємодії в реальному часі. Socket.IO також має вбудовану підтримку для повторного підключення та управління з'єднаннями, що дозволяє автоматично відновлювати з'єднання, якщо воно було втрачено, та забезпечує стійкість роботи додатка. Це є важливою функціональністю для додатків, що працюють в умовах нестабільного з'єднання або мережевих збоїв. Завдяки такій здатності автоматично відновлювати з'єднання, користувачі можуть не відчувати значних перерв у роботі додатка, що підвищує їхній досвід користування. У разі відключення чи перебоїв у з'єднанні, додаток продовжує спроби відновлення зв'язку без втрати важливої інформації, що робить його більш надійним і стійким до збоїв.

Це дозволяє зберегти плавність користувацького досвіду, навіть якщо користувачі стикаються з технічними проблемами або поганою якістю з'єднання. Крім того, завдяки таким механізмам відновлення з'єднання, розробники можуть

зосередитися на розробці функціональності, не турбуючись про вплив зовнішніх факторів на роботу додатка.

2.2.2 Організація бази даних за допомогою MySQL

На серверній частині проекту була реалізована реляційна база даних MySQL, що є однією з найбільш популярних систем управління базами даних СУБД). Вона дозволяє ефективно працювати з великими об'ємами даних та забезпечує високу продуктивність завдяки використанню SQL (Structured Query Language) для виконання операцій із даними, таких як створення, оновлення, видалення та вибірка даних. MySQL підтримує транзакції, забезпечує цілісність даних та підтримує різноманітні механізми індексації для швидкого виконання запитів.

У цьому проекті були використані різні таблиці для збереження та обробки даних користувачів та їх взаємодії з додатком. Зокрема, таблиця "users" (рис. 2.11) зберігає основні дані користувачів, включаючи ім'я користувача, електронну пошту, пароль, статус та інші налаштування. Вона також містить зовнішній ключ, що посилається на таблицю "files" (рис. 2.12), де зберігаються файли аватарок користувачів.

```
CREATE TABLE "users" (
  "id" int NOT NULL AUTO_INCREMENT,
  "username" varchar(255) NOT NULL,
  "email" varchar(255) NOT NULL,
  "password_hash" varchar(255) NOT NULL,
  "avatar_id" int DEFAULT NULL,
  "bio" text,
  "status" enum('online','offline','away') DEFAULT 'offline',
  "role" enum('admin','user','moderator') DEFAULT 'user',
  "is_verified" tinyint(1) DEFAULT '0',
  "two_factor_enabled" tinyint(1) DEFAULT '0',
  "last_login_at" datetime DEFAULT NULL,
  "last_ip_address" varchar(255) DEFAULT NULL,
  "created_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  "updated_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  "phone" varchar(15) NOT NULL,
  "verification_code" varchar(6) DEFAULT NULL,
  "verification_code_expires_at" timestamp NULL DEFAULT NULL,
  PRIMARY KEY ("id"),
  UNIQUE KEY "username" ("username"),
  UNIQUE KEY "email" ("email"),
  UNIQUE KEY "phone" ("phone"),
  KEY "idx_users_username" ("username"),
  KEY "idx_users_email" ("email"),
  KEY "users_ibfk_1" ("avatar_id"),
  CONSTRAINT "users_ibfk_1" FOREIGN KEY ("avatar_id") REFERENCES "files" ("id") ON DELETE SET NULL
)
```

Рисунок 2.11 – Таблиця users

```
CREATE TABLE "files" (
  "id" int NOT NULL AUTO_INCREMENT,
  "user_id" int NOT NULL,
  "filename" varchar(255) NOT NULL,
  "original_name" varchar(255) NOT NULL,
  "file_type" varchar(50) NOT NULL,
  "file_size" int NOT NULL,
  "uploaded_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  "file_hash" char(64) DEFAULT NULL,
  PRIMARY KEY ("id"),
  UNIQUE KEY "file_hash" ("file_hash"),
  KEY "files_ibfk_1" ("user_id"),
  CONSTRAINT "files_ibfk_1" FOREIGN KEY ("user_id") REFERENCES "users" ("id") ON DELETE CASCADE
)
```

Рисунок 2.12 – Таблиця files

Таблиця "tokens" (рис. 2.13) використовується для зберігання токенів авторизації, які дозволяють користувачам залишатися в системі без необхідності вводити пароль на кожному запиті. Зокрема, таблиця містить refresh токени, що дозволяють користувачам залишатися в системі після закінчення терміну дії основного токена. Токени зберігаються разом із датою їх створення та терміном дії.

```
CREATE TABLE "tokens" (
  "id" int NOT NULL AUTO_INCREMENT,
  "user_id" int NOT NULL,
  "refresh_token" text NOT NULL,
  "created_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  "expires_at" timestamp NOT NULL,
  PRIMARY KEY ("id"),
  KEY "user_id" ("user_id"),
  CONSTRAINT "tokens_ibfk_1" FOREIGN KEY ("user_id") REFERENCES "users" ("id") ON DELETE CASCADE
)
```

Рисунок 2.13 – Таблиця tokens

Таблиця "messages" (рис. 2.14) зберігає всі повідомлення, надіслані користувачами в чатах. Кожне повідомлення містить інформацію про відправника, текст повідомлення, час створення та статус повідомлення (наприклад, "SENT", "READ"). Крім того, повідомлення можуть бути пов'язані з файлами, які зберігаються в таблиці "message_files" (рис. 2.15). Файли, прикріплені до

повідомлень, зберігаються в таблиці "files", що дозволяє обробляти та зберігати різні типи медіафайлів.

```
CREATE TABLE "messages" (
  "id" int NOT NULL AUTO_INCREMENT,
  "sender_id" int NOT NULL,
  "group_id" int DEFAULT NULL,
  "direct_chat_id" int DEFAULT NULL,
  "content" text,
  "created_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  "updated_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  "replied_message_id" int DEFAULT NULL,
  "is_edited" tinyint(1) DEFAULT '0',
  "deleted_at" timestamp NULL DEFAULT NULL,
  "status" enum('SENDING','SENT','DELIVERED','READ','FAILED') NOT NULL DEFAULT 'SENDING',
  "delivered_at" timestamp NULL DEFAULT NULL,
  "read_at" timestamp NULL DEFAULT NULL,
  "failed_at" timestamp NULL DEFAULT NULL,
  "failure_reason" varchar(255) DEFAULT NULL,
  PRIMARY KEY ("id"),
  KEY "idx_messages_created_at" ("created_at"),
  KEY "idx_messages_sender_id" ("sender_id"),
  KEY "idx_messages_direct_chat_id" ("direct_chat_id"),
  KEY "idx_messages_group_id" ("group_id"),
  CONSTRAINT "fk_messages_group_id" FOREIGN KEY ("group_id") REFERENCES "groups" ("id") ON DELETE CASCADE,
  CONSTRAINT "messages_ibfk_1" FOREIGN KEY ("sender_id") REFERENCES "users" ("id") ON DELETE CASCADE,
  CONSTRAINT "messages_ibfk_3" FOREIGN KEY ("direct_chat_id") REFERENCES "direct_chats" ("id") ON DELETE CASCADE,
  CONSTRAINT "messages_chk_1" CHECK (((`group_id` is not null) or (`direct_chat_id` is not null)))
)
```

Рисунок 2.14 – Таблиця messages

```
CREATE TABLE "message_files" (
  "message_id" int NOT NULL,
  "file_id" int NOT NULL,
  PRIMARY KEY ("message_id","file_id"),
  KEY "fk_message_files_file_id" ("file_id"),
  CONSTRAINT "fk_message_files_file_id" FOREIGN KEY ("file_id") REFERENCES "files" ("id") ON DELETE CASCADE,
  CONSTRAINT "fk_message_files_message_id" FOREIGN KEY ("message_id") REFERENCES "messages" ("id") ON DELETE CASCADE
)
```

Рисунок 2.15 – Таблиця message_files

У проекті також реалізовані функції групових чатів і приватних чатів. Таблиця "groups" (рис. 2.16) зберігає дані про групи, такі як назва групи, опис, ідентифікатор адміністратора та інші налаштування. Користувачі можуть бути додані до групи через таблицю "group_member" (рис. 2.17), де зберігається інформація про роль користувача в групі, час приєднання та інші атрибути.

```
CREATE TABLE "groups" (
  "id" int NOT NULL AUTO_INCREMENT,
  "name" varchar(255) NOT NULL,
  "description" text,
  "admin_id" int NOT NULL,
  "created_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  "updated_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  "icon_id" int DEFAULT NULL,
  PRIMARY KEY ("id"),
  KEY "admin_id" ("admin_id"),
  KEY "idx_rooms_name" ("name"),
  KEY "fk_groups_icon" ("icon_id"),
  CONSTRAINT "fk_groups_icon" FOREIGN KEY ("icon_id") REFERENCES "files" ("id") ON DELETE SET NULL,
  CONSTRAINT "groups_ibfk_1" FOREIGN KEY ("admin_id") REFERENCES "users" ("id") ON DELETE CASCADE
)
```

Рисунок 2.16 – Таблиця groups

```
CREATE TABLE "group_member" (
  "group_id" int NOT NULL,
  "user_id" int NOT NULL,
  "joined_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  "role" enum('admin','moderator','member') NOT NULL DEFAULT 'member',
  PRIMARY KEY ("group_id","user_id"),
  KEY "user_id" ("user_id"),
  KEY "idx_group_member_role" ("role"),
  CONSTRAINT "group_member_ibfk_1" FOREIGN KEY ("group_id") REFERENCES "groups" ("id") ON DELETE CASCADE,
  CONSTRAINT "group_member_ibfk_2" FOREIGN KEY ("user_id") REFERENCES "users" ("id") ON DELETE CASCADE
)
```

Рисунок 2.17 – Таблиця group_member

Таблиця "direct_chats" (рис. 2.18) містить інформацію про приватні чати між користувачами, де зберігається ідентифікатор кожного учасника чату. Інформація про заблокованих або вимкнених користувачів зберігається в таблицях "blocked_users" (рис. 2.19) і "muted_users" (рис. 2.20), що дозволяє користувачам контролювати, хто може з ними спілкуватися.

```
CREATE TABLE "direct_chats" (
  "id" int NOT NULL AUTO_INCREMENT,
  "user1_id" int NOT NULL,
  "user2_id" int NOT NULL,
  "created_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  "is_deleted" tinyint(1) DEFAULT '0',
  "deleted_at" timestamp NULL DEFAULT NULL,
  PRIMARY KEY ("id"),
  UNIQUE KEY "user1_id" ("user1_id","user2_id"),
  KEY "user2_id" ("user2_id"),
  CONSTRAINT "direct_chats_ibfk_1" FOREIGN KEY ("user1_id") REFERENCES "users" ("id") ON DELETE CASCADE,
  CONSTRAINT "direct_chats_ibfk_2" FOREIGN KEY ("user2_id") REFERENCES "users" ("id") ON DELETE CASCADE
)
```

Рисунок 2.18 – Таблиця direct_chats

```
CREATE TABLE "blocked_users" (
  "user_id" int NOT NULL,
  "blocked_user_id" int NOT NULL,
  "blocked_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY ("user_id","blocked_user_id"),
  KEY "blocked_user_id" ("blocked_user_id"),
  CONSTRAINT "blocked_users_ibfk_1" FOREIGN KEY ("user_id") REFERENCES "users" ("id") ON DELETE CASCADE,
  CONSTRAINT "blocked_users_ibfk_2" FOREIGN KEY ("blocked_user_id") REFERENCES "users" ("id") ON DELETE CASCADE
)
```

Рисунок 2.19 – Таблиця blocked_users

```
CREATE TABLE "muted_users" (
  "user_id" int NOT NULL,
  "muted_user_id" int NOT NULL,
  "muted_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY ("user_id","muted_user_id"),
  KEY "muted_user_id" ("muted_user_id"),
  CONSTRAINT "muted_users_ibfk_1" FOREIGN KEY ("user_id") REFERENCES "users" ("id") ON DELETE CASCADE,
  CONSTRAINT "muted_users_ibfk_2" FOREIGN KEY ("muted_user_id") REFERENCES "users" ("id") ON DELETE CASCADE
)
```

Рисунок 2.20 – Таблиця muted_users

Система також підтримує механізм друзів, де кожен запис у таблиці "friends" (рис. 2.21) представляє зв'язок між двома користувачами, з можливістю підтвердження або відхилення запиту на дружбу. Всі ці таблиці працюють разом, щоб забезпечити ефективну та масштабовану роботу додатка, що дозволяє зберігати й обробляти великі обсяги даних у реальному часі.

```
CREATE TABLE "friends" (
  "user_id" int NOT NULL,
  "friend_id" int NOT NULL,
  "status" enum('pending','accepted') DEFAULT 'pending',
  "created_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  "updated_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY ("user_id","friend_id"),
  KEY "friend_id" ("friend_id"),
  KEY "idx_friends_status" ("status"),
  CONSTRAINT "friends_ibfk_1" FOREIGN KEY ("user_id") REFERENCES "users" ("id") ON DELETE CASCADE,
  CONSTRAINT "friends_ibfk_2" FOREIGN KEY ("friend_id") REFERENCES "users" ("id") ON DELETE CASCADE
)
```

Рисунок 2.21 – Таблиця friends

MySQL також забезпечує високий рівень безпеки даних, підтримуючи шифрування, захищені з'єднання та точне налаштування доступу користувачів до даних, що робить її надійним вибором для побудови додатків, які працюють з чутливою інформацією. Усі ці можливості дозволяють створювати складні додатки з багатьма таблицями та зв'язками між ними.

2.2.3 Авторизація за допомогою токенів

JWT (JSON Web Token) — це стандарт для створення токенів, що використовуються для аутентифікації та авторизації користувачів. JWT складається з трьох частин: заголовка (header), корисного навантаження (payload) та підпису (signature).

Заголовок (header) є JSON-об'єктом, що містить метадані про токен, зокрема алгоритм, використаний для створення підпису, та тип токена. Корисне навантаження (payload) — це JSON-об'єкт, що містить інформацію про користувача або інші дані, необхідні для передачі. Підпис (signature) — це підпис, який генерується на основі заголовка та корисного навантаження з використанням секретного ключа, відомого лише серверу, що створив токен. Підпис додається до заголовка та корисного навантаження як третя частина JWT і гарантує, що токен не був змінений після його створення та що тільки сервер, який має секретний ключ, може перевірити його автентичність.

Це забезпечує захист від підробки токена та гарантує безпеку процесу автентифікації та авторизації. Для створення підпису використовуються алгоритми шифрування, такі як HMAC (Hash-based Message Authentication Code) або RSA (Rivest-Shamir-Adleman). Якщо підпис не відповідає очікуваному значенню, токен вважається недійсним і не буде прийнятий сервером (рис. 2.22).

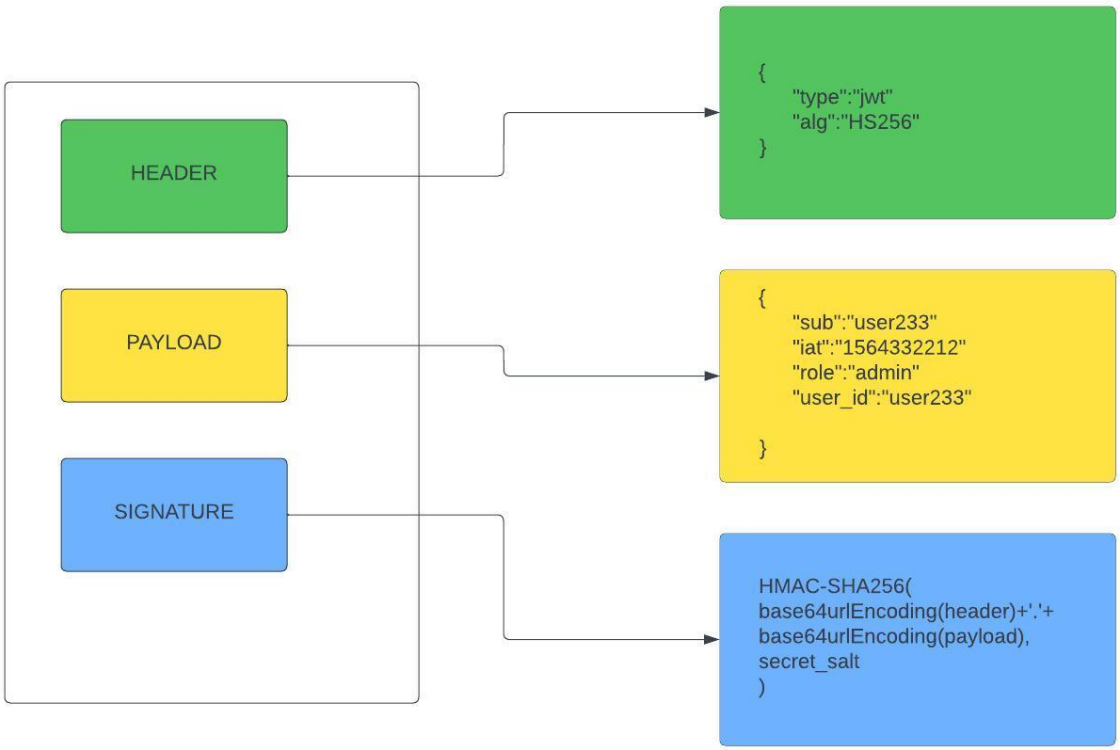


Рисунок 2.22 – JWT

Існує два види токенів: AccessToken та RefreshToken. AccessToken це токен, який передається під час кожного запиту до сервісу, містить інформацію про користувача та його доступ до ресурсів. Його термін дії обмежений, що підвищує безпеку, оскільки зменшується можливість зловживань. Після закінчення терміну дії AccessToken, користувач повинен знову пройти автентифікацію для отримання нового токена.

RefreshToken це токен, який використовується для отримання нового AccessToken після того, як попередній AccessToken втратив свою дійсність. RefreshToken має більший термін дії та передається тільки при запиті нового AccessToken. Користувач не має прямого доступу до RefreshToken, і він не передається при кожному запиті, що підвищує безпеку. Процес отримання нового AccessToken за допомогою RefreshToken називається Refresh Token Flow. Під час автентифікації користувач вводить логін та пароль, після чого сервіс генерує і повертає AccessToken та RefreshToken. AccessToken передається з кожним запитом, а RefreshToken зберігається на стороні клієнта. Коли AccessToken втрачає

актуальність, клієнт використовує RefreshToken для отримання нового AccessToken та оновлення даних користувача. Після цього процес повторюється, і нові AccessToken та RefreshToken знову передаються клієнту.

Перед створенням токенів сервер перевіряє користувача. Залежно від запиту, здійснюється або авторизація, або реєстрація. Наприклад, при авторизації, якщо користувач є в базі даних, система перевіряє правильність введених даних, таких як електронна пошта та пароль. Для перевірки пароля використовуються методи шифрування, які зазначені на сервері. Як вже зазначалося, перевірка пароля здійснюється шляхом порівняння введеного значення з тим, яке зберігається в базі даних, з використанням декодування (рис. 2.23).

```
async login(email, password) {
    const user = await UserModel.findOne("email", email);
    if (!user) {
      throw ApiError.BadRequest("User with this email couldn't be found", []);
    }
    const isPassEquals = await bcrypt.compare(password, user.password_hash);

    if (!isPassEquals) {
      throw ApiError.BadRequest("Incorrect password", []);
    }
    const userDto = new UserDto(user);
    const tokens = tokenService.generateToken({ ...userDto });

    await tokenService.saveToken(userDto.id, tokens.refreshToken);

    return {
      ...tokens,
      user: userDto,
    };
  }

```

Рисунок 2.23 – Код декодування паролю користувача

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ВЕБ-ДОДАТКА

3.1 Реалізація функціоналу веб-чату

Функціонал веб-додатка включає авторизацію користувачів через токени, обмін повідомленнями через WebSocket, створення та управління чатами, інтеграцію медіафайлів і управління чорним списком для блокування небажаних користувачів.

3.1.1 Авторизація користувачів через токени

Під час реєстрації користувача спочатку перевіряємо, чи існує вже такий користувач у базі даних. Якщо його немає, створюємо нового користувача, а якщо є, то виводимо помилку. Потім пароль, введений користувачем, шифрується перед збереженням у базі даних, щоб забезпечити його безпеку. Це дозволяє захистити пароль від можливих атак на базу даних, адже навіть при витоку даних пароль не буде використаний для авторизації. На основі пароля, електронної пошти, імені та телефона формуємо об'єкт користувача, після чого здійснюється реєстрація (рис. 3.1). Одночасно генерується токен, що містить інформацію про користувача та секретне слово, яке знає лише сервер (рис. 3.2). Цей секретний ключ служить для перевірки справжності токена. Він використовується під час генерації і перевірки токена, що дозволяє впевнитись у тому, що інформація не була змінена після того, як токен був створений. Таким чином, навіть якщо зловмисник отримає токен, без доступу до цього секретного ключа він не зможе його використати для несанкціонованого доступу. Токен надається клієнту і надалі використовується для аутентифікації в системі під час кожного запиту до сервера.

Рисунок 3.1 - Реєстрація користувача

```
generateToken(payload) {
    const accessToken = jwt.sign(payload, process.env.JWT_ACCESS_SECRET, {
      expiresIn: "30m",
    });
    const refreshToken = jwt.sign(payload, process.env.JWT_REFRESH_SECRET, {
      expiresIn: "30d",
    });
    return {
      accessToken,
      refreshToken,
    };
  }
```

Рисунок 3.2 - Код генерації токенів

У випадку авторизації достатньо надати лише електронну пошту та пароль (рис. 3.3). На сервері ці дані обробляються для виконання запитів до бази даних, де перевіряється існування відповідного користувача. Сервер зіставляє надані дані з інформацією, що зберігається в базі, перевіряючи, чи збігається введений пароль із збереженим.

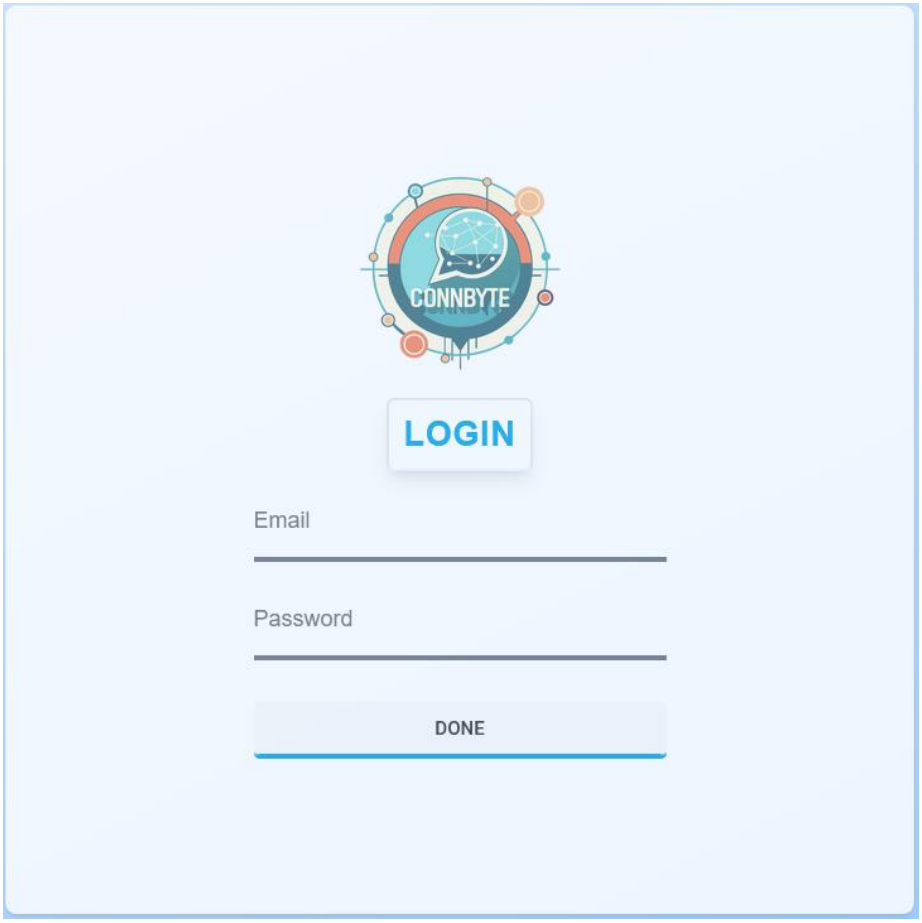


Рисунок 3.3 – Авторизація користувача

Після успішної реєстрації або входу користувача, система повертає об'єкт, що містить два токени "RefreshToken" та "AccessToken", а також дані користувача. Паралельно з цим сервер встановлює кукі, в які додається "RefreshToken".

Після входу в систему, в верхньому правому куті екрану відображаються аватар користувача (рис. 3.4). При оновленні сторінки відбувається автоматична перевірка на наявність токена доступу в локальному сховищі браузера (рис. 3.5). Якщо токен доступу знайдений, це свідчить про те, що користувач вже авторизувався або зареєструвався в обліковому записі. Інформація про токен передається на сервер через кукі, і сервер використовує ці дані для отримання нових токенів доступу та оновлення (рис. 3.6).

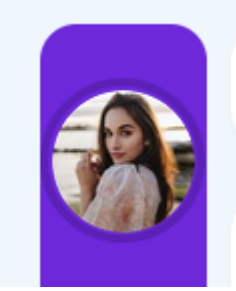


Рисунок 3.4 - Результат авторизації

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTcsInVzZXJuYW1lIjoia3JvbGFuNzIiLCJlbWFpbCI6Imtyb2xhbjcyQGdtYWlsLmNvbSIsInBob25lIjoiKzM4MDY3NDQzNTUyMSIsImJpbyI6bnVsbCwic3RhdHVzIjoib2ZmbGluZSIsInJvbGUiOiJ1c2VyIiwiaXNWZXJpZmllZCI6dHJ1ZSwidHdvRmFjdG9yRW5hYmxlZCI6ZmFsc2UsImxhc3RMb2dpbkF0IjpudWxsLCJsYXN0SXBBZGRyZXNzIjpudWxsLCJjcmVhdGVkQXQiOiIyMDI0LTExLTA2VDEzOjIzOjI1LjAwMFoiLCJpYXQiOjE3MzQyMDU1NTQsImV4cCI6MTczNDIwNzM1NH0.pEUlyoiRhw60Fg7Xe5UqUS0NSMMxgEStxjIBOsNyLCM
```

Рисунок 3.5 – Токен доступу

```
export const checkAuth = createAsyncThunk<IUser, void, { rejectValue: string }>(
  "authorization/user/checkAuth",
  async (_, thunkAPI) => {
    try {
      const response = await AuthService.checkAuth();
      setItem("token", response.data.accessToken);
      return response.data.user;
    } catch (error) {
      const message = handleApiError(error);
      return thunkAPI.rejectWithValue(message);
    }
  }
);
```

Рисунок 3.6 – Код перевірки чи була здійснена до цього аутентифікація

Під час реєстрації або авторизації, якщо користувач, ще не вводив код, то на вказану поштову адресу буде надіслано лист із кодом (рис. 3.7). Це реалізовано за допомогою бібліотеки Nodemailer, яка є зручним і потужним інструментом для відправлення електронної пошти з вашого додатку. Nodemailer дозволяє налаштовувати та ефективно керувати процесом відправки листів через SMTP-сервери, надаючи гнучкі можливості для інтеграції в будь-який проект.

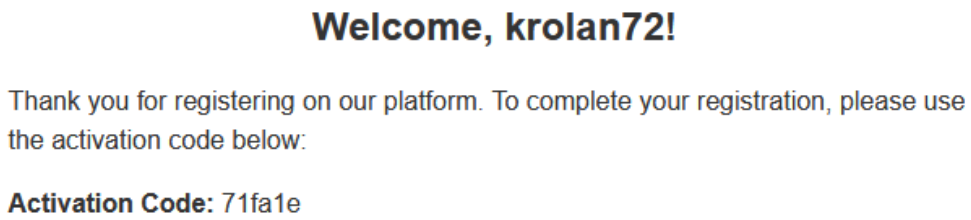


Рисунок 3.7 – Лист з кодом

Метод sendActivationMail використовує об'єкт transporter для відправки листа активації. Він приймає два параметри: адресу отримувача та активаційне посилання, яке буде включене в лист. Для формування листа застосовується HTML-шаблон, який містить плейсхолдери для динамічних даних, таких як активаційний код, ім’я. Після того як шаблон буде заповнений необхідною інформацією, лист відправляється через встановлене з'єднання з SMTP-сервером. SMTP (Simple Mail Transfer Protocol) — це стандартний протокол для передачі електронної пошти, що забезпечує надійний та безпечний процес доставки повідомлень між серверами електронної пошти. Використання SMTP дозволяє забезпечити правильну доставку листів, дотримуючись стандартних методів аутентифікації та перевірки.

Для створення з'єднання з SMTP-сервером використовується бібліотека Nodemailer. Об'єкт transporter містить усі налаштування для надсилання листів. Параметри підключення до сервера, такі як хост, порт, ідентифікатор користувача та пароль, зберігаються в конфігураційному файлі .env за допомогою бібліотеки dotenv, що гарантує безпеку збереження конфіденційної інформації.

Для коректної роботи цього коду необхідно налаштувати SMTP-сервер та використовувати дійсні дані аутентифікації, які можна отримати від постачальника електронної пошти або відповідного сервісу. У цьому випадку був обраний поштовий сервіс Google, тому було активовано двофакторну аутентифікацію для забезпечення безпеки доступу до облікового запису. Це обмеження було введене саме компанією Google для запобігання несанкціонованому доступу.

Настроювання SMTP-сервера Google передбачає кілька етапів. Спочатку необхідно мати обліковий запис Google і активувати двофакторну аутентифікацію для підвищення рівня безпеки доступу до акаунту. Для цього можна використовувати мобільний додаток Google Authenticator або подібні інструменти.

Наступним кроком є налаштування SMTP-сервера Google з такими параметрами: адреса сервера — "smtp.gmail.com", порт — 465 або 587 (залежно від використовуваного типу шифрування), тип шифрування — SSL або TLS. Для аутентифікації потрібно вказати адресу електронної пошти, основний пароль акаунту Google та створений пароль додатку.

3.1.2 Обмін повідомленнями

Обмін повідомленнями в веб-додатку досягається завдяки використанню бібліотеки Socket.IO, яка забезпечує двостороннє з'єднання між клієнтом і сервером у реальному часі. Вона побудована на основі WebSocket, але має додатковий функціонал, що робить інтеграцію простішою та забезпечує більшу гнучкість для обміну даними. Завдяки Socket.IO можна організувати ефективну та стабільну передачу повідомлень без необхідності перезавантаження сторінки.

Для налаштування серверної частини використовуємо бібліотеку Socket.IO, яка інтегрується з Express через HTTP сервер. При ініціалізації створюється об'єкт Server, який налаштовує параметри CORS для дозволу запитів з клієнта. Цей об'єкт отримує доступ до HTTP сервера та налаштовує з'єднання за допомогою Socket.IO. Однією з ключових переваг цієї бібліотеки є її підтримка різноманітних протоколів і механізмів для забезпечення стабільного з'єднання.

Цей код дозволяє ініціалізувати сервер, налаштувати дозволи для з'єднань і підключати middleware для автентифікації користувачів через токен. Після цього сервер приймає нові з'єднання, перевіряючи кожного користувача перед підключенням. Деталі налаштування з'єднання можна побачити на рис. 3.8.

```
const io = new Server(httpServer, {
    cors: {
      origin: process.env.CLIENT_URL,
      methods: ["GET", "POST"],
      credentials: true,
    },
  });
```

Рисунок 3.8 - Ініціалізація серверу

Після підключення нового користувача, сервер відслідковує події, такі як з'єднання (connection) і відключення (disconnect) (рис. 3.9). Кожен користувач отримує унікальний ідентифікатор сокета, який дозволяє організувати приватні кімнати для обміну повідомленнями. Користувачі підключаються до своїх "особистих кімнат" для обміну повідомленнями, що є важливим для організації приватних чатових сесій.

```
// Применяем middleware ко всем сокет-соединениям
io.use(socketAuthMiddleware);

io.on("connection", (socket) => {
  console.log("New socket connection Server:", socket.id);
  console.log("Authenticated User ID:", socket.userId);

  // Присоединяем пользователя к его личной комнате
  socket.join(`user_${socket.userId}`);
  console.log(`User ${socket.userId} joined room: user_${socket.userId}`);
  // Инициализируем все сокет-роуты
  initSocketRoutes(io, socket);
  SocketListeners.initListeners(socket, io);
  socket.on("disconnect", () => {
    console.log("Socket disconnected:", socket.id);
    // Опционально: можно удалить комнату при отключении
    socket.leave(`user_${socket.userId}`);
  });
});
```

Рисунок 3.9 - Код основні події

Для реалізації обміну повідомленнями сервер використовує декілька маршрутизаторів, які відповідають за обробку групових чатів, повідомлень, додавання друзів та інших функцій. Це дозволяє розділити різні аспекти обміну даними та зменшити навантаження на сервер. Деталі ініціалізації цих маршрутів можна побачити на рис. 3.10.

```
import initGroupRoutes from "../socket/group-socket-router.js";
import initNotificationRoutes from "../socket/notification-socket-router.js";
import initDirectChatRoutes from "../socket/direct-chat-socket-router.js";
import initFriendRoutes from "../socket/friend-socket-router.js";
import initUserRoutes from "../socket/user-socket-router.js";
import initUserRelationshipsRoutes from "../socket/user-relationship-socket-router.js";
import initMessageRoutes from "../socket/message-socket-router.js";
export default function initSocketRoutes(io, socket) {
  initGroupRoutes(io, socket);
  initNotificationRoutes(io, socket);
  initDirectChatRoutes(io, socket);
  initFriendRoutes(io, socket);
  initUserRoutes(io, socket);
  initUserRelationshipsRoutes(io, socket);
  initMessageRoutes(io, socket);
}
```

Рисунок 3.10 – Код маршрутів

Для автентифікації користувачів на сервері використовується проміжний процес socketAuthMiddleware. Цей механізм відповідає за перевірку токену, наданого клієнтом при підключенні до сервера через Socket.IO. Кожен клієнт, що підключається, повинен надати дійсний токен у запиті на підключення. Якщо токен валідний, сервер дозволяє підключення і додає дані користувача до сокет-об'єкта, що надає доступ до подальших операцій у системі. У разі, якщо токен відсутній або він не відповідає критеріям безпеки (наприклад, за терміном дії або структурою), сервер відмовляється від з'єднання, тим самим захищаючи систему від несанкціонованого доступу. Цей процес описано на рис. 3.11.

```
import TokenService from "../services/token-service.js";
import UserModel from "../models/user-model.js";

export default async function socketAuthMiddleware(socket, next) {
  try {
    // Получаем токен из заголовков или cookies
    const token =
      socket.handshake.auth.token ||
      socket.handshake.headers.authorization?.split(" ")[1];

    if (!token) {
      return next(new Error("Authentication error: No token provided"));
    }

    // Валидируем access token
    const userData = TokenService.validateAccessToken(token);

    if (!userData) {
      return next(new Error("Authentication error: Invalid token"));
    }

    // Проверяем существование пользователя в базе
    const user = await UserModel.findById(userData.id);

    if (!user) {
      return next(new Error("Authentication error: User not found"));
    }

    // Добавляем userId к сокету
    socket.userId = user.id;
```

Рисунок 3.11 – Код middleware для авторизації

Після успішної автентифікації користувач може підключатися до приватних кімнат для обміну повідомленнями. Цей процес забезпечує безпеку та конфіденційність, оскільки тільки авторизовані користувачі мають доступ до певних чатів або груп. Реалізація цього процесу через Socket.IO дозволяє підтримувати постійне двостороннє з'єднання між клієнтом і сервером, що є основою для миттєвого обміну повідомленнями в реальному часі. Такий підхід є надзвичайно важливим для чатів, онлайн-конференцій і будь-яких інших додатків, де необхідна швидка та безперервна взаємодія користувачів.

3.1.3 Створення та управління чатами

Веб-додаток реалізує можливість створення та управління чатами за допомогою обробки подій через Socket.IO. Цей функціонал дозволяє користувачам створювати особисті чати, приєднуватися до них, а також здійснювати керування чатами, наприклад, видаляти їх чи отримувати доступ до існуючих.

Для налаштування та обробки подій, пов'язаних з чатами, використовується спеціальний клас SocketListeners, який містить статичний метод initListeners (рис. 3.12). Цей метод викликає два набори слухачів: для особистих чатів та групових чатів. Всі ці слухачі ініціалізуються після встановлення з'єднання користувача з сервером. Така архітектура дозволяє централізовано обробляти всі події, що стосуються чатів, і розподіляти їх між різними типами чатів, а також знижує складність коду.

```
1 import directChatListeners from "./directChatListeners.js";
2 import groupListeners from "./groupListeners.js";
3 class SocketListeners {
4   constructor() {}
5   static async initListeners(socket, io) {
6     // Initialize listeners for direct chat and groups service
7     directChatListeners(socket, io);
8     groupListeners(socket, io);
9   }
10 }
11
12 export default SocketListeners;
13
```

Рисунок 3.12 – Код класу SocketListeners

У разі підключення користувача до сервера, цей користувач отримує список чатів, до яких він має доступ, з бази даних. Кожен чат прив'язується до окремої "кімнати" в Socket.IO, що дозволяє організувати ізольований простір для кожного чату, де користувач може отримувати повідомлення лише зі свого чату. Метод directChatListeners викликається при ініціалізації з'єднання і відповідно підключає сокет до кожної відповідної кімнати (рис. 3.13).

```
import DirectChatService from "../services/direct-chat-service.js";

const directChatListeners = async (socket, io) => {
  const userId = socket.userId; // Получите ID пользователя из сокета
  const chats = await DirectChatService.getUserChats(userId); // Извлеките чаты из базы данных
  console.log(chats);
  if (chats.length > 0) {
    chats.forEach((chat) => {
      socket.join(`chat_${chat.id}`); // Подключите сокет к каждой комнате
      console.log(`User ${socket.userId} joined chat: chat_${chat.id}`);
    });
  }
};
export default directChatListeners;

```

Рисунок 3.13 – Код слухач для персональних чатів

Також реалізовано обробку кількох подій для керування чатами, таких як створення нового чату, пошук чату за користувачами або видалення чату. Для кожної події сервер перевіряє коректність отриманих даних.

Наприклад, при створенні нового чату через подію "createChat", користувач передає ID іншого користувача, з яким хоче створити чат. Сервер виконує перевірку цих даних, використовуючи утиліту validateSocketData, що забезпечує їх відповідність необхідному формату. Якщо дані коректні, сервер викликає відповідний контролер, який обробляє запит та створює новий чат у базі даних.

Аналогічно, для подій на кшталт "deleteChat", "findChatByUsers" чи інших, сервер також здійснює валідацію вхідних даних перед виконанням відповідних операцій. Це дозволяє уникнути потенційних помилок або некоректних запитів. Наприклад:

- У події "deleteChat" сервер перевіряє, чи має користувач права на видалення зазначеного чату.
- У події "findChatByUsers" сервер переконується, що передані ідентифікатори користувачів дійсні й існують у системі.

Реалізація валідації даних через validateSocketData (рис. 3.14), забезпечує централізовану перевірку всіх подій, що спрощує підтримку коду та мінімізує ризики.

```
export function validateSocketData(socket, eventName, data, config) {
  try {
    const { requiredFields = [], typeMap = {}, customValidation } = config;

    // Проверка обязательных полей
    for (const field of requiredFields) {
      if (data[field] === undefined || data[field] === null) {
        throw new Error(
          `Field '${field}' is required for event '${eventName}'`
        );
      }
    }

    // Проверка типов
    for (const [field, expectedType] of Object.entries(typeMap)) {
      if (data[field] !== undefined) {
        const actualType = typeof data[field];
        const isValidType = checkType(data[field], expectedType);

        if (!isValidType) {
          throw new Error(
            `Invalid type for field '${field}'. Expected ${expectedType}, got ${actualType}`
          );
        }
      }
    }

    // Кастомная валидация
    if (customValidation) {
      customValidation(data);
    }

    return true;
  } catch (error) {
    socket.emit("validationError", {
      event: eventName,
      message: error.message,
    });
    return false;
  }
```

Рисунок 3.14 – Код валідації сокет даних

Користувачі можуть створювати нові чати за допомогою події "createChat". Це дозволяє створити чат між двома користувачами, після чого вони автоматично приєднуються до відповідної кімнати. Цей процес реалізується через обробник події на сервері, який обробляє дані і викликає необхідний метод у відповідному контролері. Після створення чату користувачі зможуть обмінюватися повідомленнями в реальному часі, що забезпечує інтерактивність і динамічний обмін інформацією.

Для групових чатів реалізовано подібний механізм, але з додатковими функціями для керування групами, запрошення користувачів, видалення учасників та інші операції. Усі ці функції також обробляються за допомогою відповідних слухачів подій і контролерів, що забезпечує гнучкість при управлінні як особистими, так і груповими чатами.

3.1.4 Інтеграція медіафайлів

Інтеграція медіафайлів у веб-додаток дозволяє користувачам передавати, зберігати та використовувати різні типи файлів, наприклад, зображення, документи чи інші медіафайли. Це робить систему більш зручною для користувачів, оскільки вони можуть додавати файли до своїх повідомлень або профілів.

У цьому проєкті для обробки завантаження файлів використовується бібліотека Multer. Вона дозволяє легко працювати з файлами, які передаються разом із запитом на сервер. Для початку була налаштована система збереження файлів на сервері. Було вирішено використовувати multer.diskStorage(), що дало змогу чітко вказати, де саме на сервері зберігатимуться завантажені файли, а також як формуватимуться їхні імена.

Щоб уникнути проблем із файлами, які можуть мати однакові імена, було реалізовано генерацію унікальних назв. Для цього використовується бібліотека UUID, яка створює унікальні ідентифікатори. Такий підхід гарантує, що навіть якщо різні користувачі завантажать файли з однаковими іменами, вони не замінять одне одного, оскільки отримають унікальні назви (рис. 3.15).

```
import * as uuid from "uuid";
import multer from "multer";
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, "/uploads");
  },
  filename: function (req, file, cb) {
    cb(
      null,
      `${file.fieldname}-${uuid.v4()}${path.extname(file.originalname)}`
    );
  },
});
const types = [""];
```

Рисунок 3.15 – Код налаштування multer

У цьому коді визначено, що файли будуть зберігатися в папці /uploads. Ім'я файлу формується з урахуванням його оригінального розширення, до якого додається унікальний ідентифікатор.

Для обмеження типів файлів, які можуть бути завантажені, використовується функція fileFilter. Вона перевіряє тип файлу за допомогою mimetype і дозволяє лише певні типи файлів, наприклад, зображення чи документи. Якщо файл не відповідає дозволеному формату, він не буде прийнятий на сервер.

Це дозволяє контролювати, які саме файли можуть бути завантажені в додаток і забезпечити безпеку, запобігаючи завантаженню шкідливих файлів. Фільтрації виглядає наступним чином (рис. 3.16).

```
function fileFilter(req, file, cb) {
  if (types.includes(file.mimetype)) {
    cb(null, true);
  } else {
    cb(null, false);
  }

  //   cb(new Error("I don't have a clue!"));
}
```

Рисунок 3.16 – Код фільтрація файлів

Після того, як файл завантажено, він зберігається на сервері. Для кожного файлу також створюється запис у базі даних. Використовуються функції, які перевіряють, чи не існує файл з таким самим хешем (щоб уникнути дублювання файлів), і якщо він вже є, файл не зберігається повторно. Якщо ж файл новий, його записують у базу, а також зберігають метадані, такі як ім’я, розмір, тип файлу та хеш.

Цей процес реалізовано в методах класу File, який відповідає за роботу з файлами. Клас дозволяє зберігати метадані файлів у базі даних, зокрема для іконок груп та інших медіафайлів. Це забезпечує можливість швидкого доступу до файлів за допомогою їх ідентифікаторів.

Медіафайли зберігаються не лише на сервері, але й у базі даних, де для кожного файлу зберігаються всі необхідні метадані, такі як ідентифікатор користувача, ім’я файлу, тип, розмір та інші атрибути. Крім того, сервер здатен обробляти запити на отримання файлів, наприклад, для відображення іконок груп або для завантаження файлів користувачами.

Ці функції реалізовані у методах класу File, які взаємодіють з базою даних і файловою системою для обробки файлів, що додаються користувачами. За допомогою цих методів можна отримати доступ до файлів через їх ідентифікатори або хеші, що дозволяє ефективно керувати медіафайлами в додатку. Завдяки таким методам класу File можна легко додавати, зберігати та знаходити медіафайли в системі, а також зберігати всю необхідну інформацію про файли для подальшого їх використання. Це дозволяє забезпечити ефективне управління контентом у веб-додатку. Код для цих операцій можна побачити на рис. 3.17.

```
async uploadAndCheckFiles(filesArray, senderId) {
    if (!filesArray || filesArray.length === 0) {
      return []; // Возвращаем пустой массив, если нет файлов
    }
    const fileIds = await Promise.all(
      filesArray.map(async (file) => {
        const { path: filePath, originalname, mimetype, size, filename } = file;
        const fileHash = await this.generateFileHash(filePath);

        return await FileModel.saveFileRecord(
          senderId,
          filename,
          originalname,
          mimetype,
          size,
          fileHash
        );
      })
    );

    return fileIds;
  }
```

Рисунок 3.17 – Код керування медіафайлами

3.1.5 Управління чорним списком

Управління чорним списком є важливою складовою частиною функціоналу веб-додатків, де взаємодія між користувачами може бути обмежена або регульована за допомогою певних механізмів, таких як блокування, розблокування, а також вимкнення звуків для окремих користувачів. Це дозволяє користувачам мати більший контроль над своєю взаємодією з іншими людьми в межах чату або соціальної мережі. У системах, де використовується реальний час для обміну повідомленнями, функції управління чорним списком повинні бути інтегровані так, щоб забезпечити миттєву реакцію на запити.

Блокування користувачів є однією з основних функцій чорного списку. Коли користувач блокує іншу особу, він більше не отримує повідомлення від заблокованого користувача. У системі реалізована подія для блокування користувачів, яка активується через сокет-зв'язок (рис. 3.18). Після надходження запиту на блокування, система перевіряє, чи є всі необхідні дані (наприклад,

ідентифікатор користувача, якого потрібно заблокувати). Після успішної валідації запиту виконується процес блокування, а дані про заблокованого користувача зберігаються в базі даних.

```
socket.on("blockUser", (data) => {
    const validationConfig = {
      requiredFields: ["targetUserId"],
      typeMap: {
        targetUserId: "number",
      },
    };

    if (validateSocketData(socket, "blockUser", data, validationConfig)) {
      userRelationshipsSocketController.blockUser({
        ...data,
        userId: socket.userId,
      });
    }
  });
```

Рисунок 3.18 – Код блокування користувача

Функція розблокування дозволяє користувачам відновити взаємодію з особами, яких вони раніше заблокували (рис. 3.19). Це дозволяє коригувати взаємодію в разі зміни обставин або помилкових блокувань. Процес розблокування дуже схожий на процес блокування і включає кілька етапів. Спочатку перевіряється правильність введених даних, щоб переконатися, що запит на розблокування походить від легітимного користувача. Далі здійснюється перевірка, чи є в базі даних інформація про блокування цієї особи, щоб уникнути непотрібних операцій. Якщо все коректно, то система виконує операцію розблокування, оновлюючи відповідні записи в базі даних.

```
socket.on("unblockUser", (data) => {
    const validationConfig = {
      requiredFields: ["targetUserId"],
      typeMap: {
        targetUserId: "number",
      },
    };

    if (validateSocketData(socket, "unblockUser", data, validationConfig)) {
      userRelationshipsSocketController.unblockUser({
        ...data,
        userId: socket.userId,
      });
    }
  });
```

Рисунок 3.19 – Код розблокування користувача

Окрім повного блокування, система також передбачає можливість тимчасово вимикати звук для певних користувачів (рис. 3.20), що дозволяє уникнути спаму або частих повідомлень від певних осіб без необхідності повного блокування. Вимкнення звуку застосовується, коли користувач хоче продовжувати мати доступ до чату, але не хоче отримувати звукові сповіщення. Це може бути корисним для чатових груп або для особистих чатів, де одні користувачі можуть надіслати багато повідомлень.

```
socket.on("muteUser", (data) => {
    const validationConfig = {
      requiredFields: ["targetUserId"],
      typeMap: {
        targetUserId: "number",
      },
    };

    if (validateSocketData(socket, "muteUser", data, validationConfig)) {
      userRelationshipsSocketController.muteUser({
        ...data,
        userId: socket.userId,
      });
    }
  });
```

Рисунок 3.20 – Код вимикання звуку користувача

Щоб забезпечити зручний доступ до інформації про користувачів, які були заблоковані або для яких вимкнено звук, система надає можливість переглядати ці списки. Користувачі можуть отримати список заблокованих осіб або тих, кого вони вимкнули для сповіщень. Це дозволяє мати повний контроль над своєю соціальною взаємодією.

Для реалізації всіх цих функцій використовуються різні сокет-сервіси, які обробляють події, що надходять від клієнтів. Кожен тип операції (блокування, розблокування, вимкнення звуків тощо) має свій окремий обробник, який відповідає за обробку запитів, перевірку даних і виконання необхідних змін у базі даних. Ці обробники також керують актуальністю даних і забезпечують правильне виконання операцій у реальному часі.

3.2 Тестування клієнтської та серверної частини

Під час тестування веб-додатку були протестовані основні функціональні можливості, такі як відправка повідомлень, створення груп, додавання друзів та управління чорним списком. Кожен з цих аспектів був ретельно перевірений для забезпечення стабільної та безпомилкової роботи додатку. Тестування було виконано у кілька етапів, щоб перевірити як серверну, так і клієнтську частину.

Першим етапом тестування було перевірка функціональності обміну повідомленнями, зокрема як у особистих чатах, так і в групових. Тестувалась можливість відправлення повідомлень (рис. 3.21), отримання їх у реальному часі, а також додавання вкладень до повідомлень, таких як медіафайли. Під час тестування перевірялось, чи коректно працюють WebSocket з'єднання при відправці та отриманні повідомлень, а також обробка помилок, що можуть виникнути при неналежному з'єднанні.

У результаті тестування було підтверджено, що всі повідомлення доставляються вчасно, а медіафайли коректно завантажуються на сервер, після чого користувачі можуть їх переглядати в чатах.

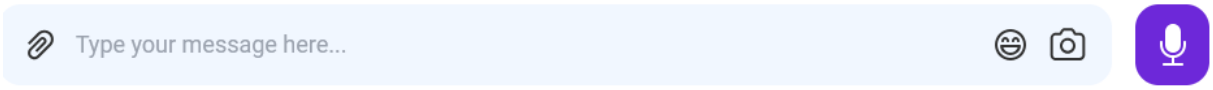


Рисунок 3.21 – Відправка повідомлення

Другим етапом було тестування функціональності створення груп та додавання учасників. Користувачі могли створювати нові групи, запрошувати до них інших користувачів і перевіряти, чи правильно працює система додавання учасників до групи. Під час тестування особлива увага була приділена таким аспектам, як коректне відображення списку учасників групи (рис. 3.22), можливість зміни прав адміністратора та додавання нових членів до існуючих груп. Також було перевірено, чи працює коректно система управління групами, зокрема видалення учасників та зміна назв груп.

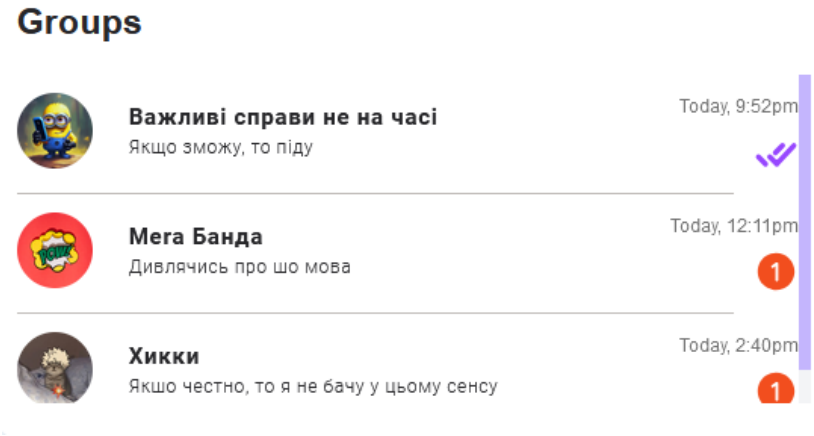


Рисунок 3.22 – Список груп

Окремо було протестовано функціональність додавання друзів, а також обробка запитів на блокування та розблокування користувачів. Користувачі мали можливість відправляти запити на дружбу (рис. 3.23), підтверджувати або відхиляти їх, а також блокувати або розблоковувати інших користувачів. Це тестування охоплювало всі основні сценарії взаємодії з друзями, включаючи блокування користувачів, щоб перевірити, чи коректно зберігаються ці дані в базі даних і чи працює функція відображення заблокованих користувачів у списку.

Особливу увагу було приділено перевірці роботи чорного списку, щоб переконатись, що після блокування користувача він не може надсилати повідомлення або додавати користувача в нові чати.

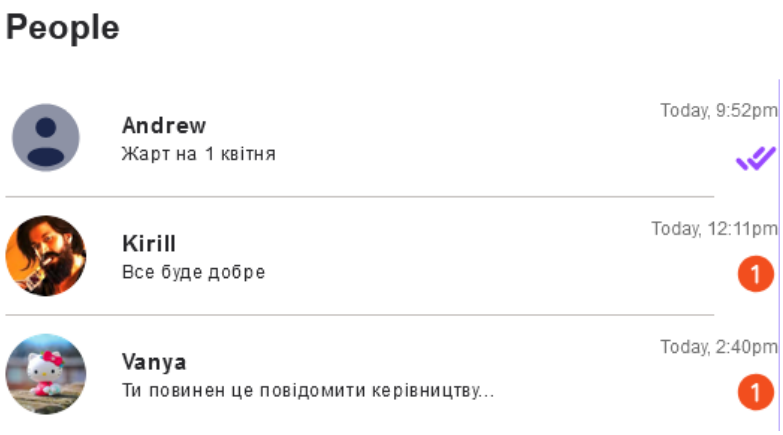


Рисунок 3.23 – Список друзів

Також було проведено тестування клієнтської та серверної частини на час затримки з метою порівняння ефективності різних технологій обміну даними в режимі реального часу: WebSocket, Polling, Long Polling та SSE (Server-Sent Events). Затримка, як ключовий показник швидкості передачі даних, відіграє вирішальну роль у забезпеченні миттєвого оновлення інформації в веб-додатках.

Для забезпечення об'єктивності тестування було створено спеціальне тестове середовище, що імітувало типові умови використання веб-додатку з різним рівнем навантаження. Вимірювання затримки здійснювалося шляхом фіксації часового інтервалу між відправленням запиту клієнтом та отриманням відповіді від сервера. Тестування проводилося в різних мережевих умовах та з різною кількістю одночасних з'єднань, що дозволило отримати повну картину продуктивності кожної з досліджуваних технологій.

Результати тестування чітко продемонстрували різницю в затримках між різними методами. WebSocket показав найнижчі показники затримки, що підтверджує його високу ефективність для додатків, що працюють в режимі реального часу (рис. 3.24).

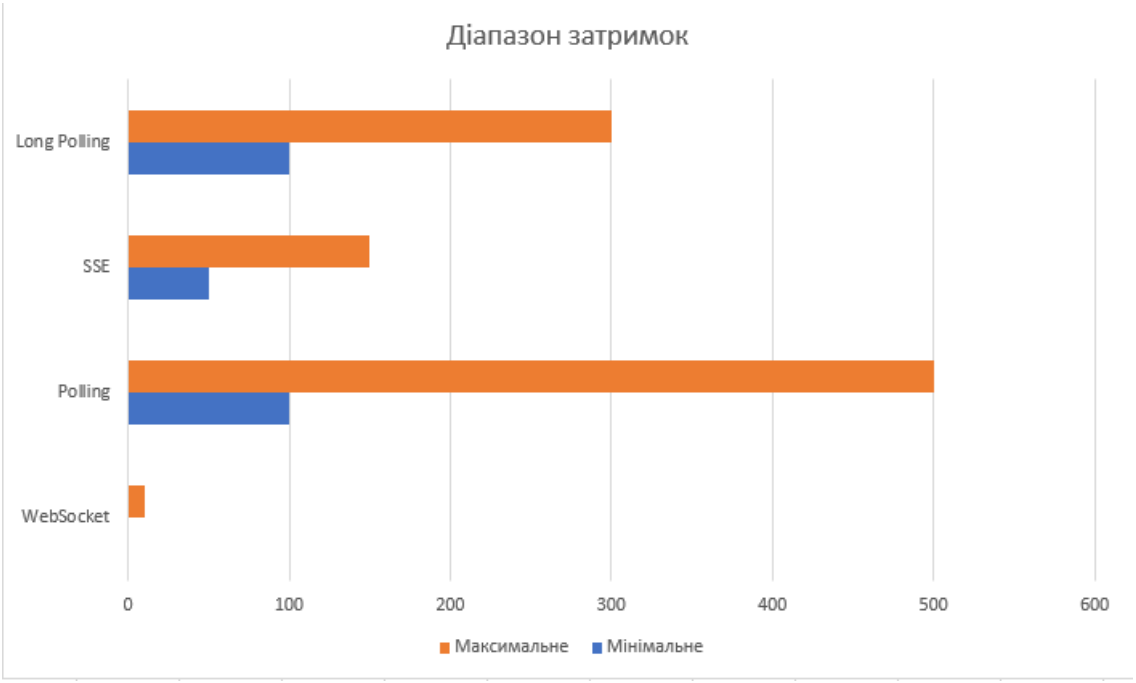


Рисунок 3.24 – Порівняння затримок

Зокрема, результати виглядають наступним чином:

- Порівняно з SSE: WebSocket продемонстрував зниження затримки на 70-80%. Цей значний показник підкреслює перевагу WebSocket у швидкості передачі даних, що є критичним для додатків, чутливих до часу, таких як онлайн-ігри, відеоконференції, системи миттєвих повідомлень та інші інтерактивні веб-сервіси.
- Порівняно з Polling та Long Polling: WebSocket досяг вражаючого зниження затримки на 90%. Це свідчить про значну перевагу WebSocket над традиційними методами опитування, які є менш ефективними для частих оновлень. Polling, через свою циклічну природу, завжди вносить значну затримку, що прямо пропорційна інтервалу опитування. Long Polling, хоча і є покращенням порівняно з Polling, все ж поступається WebSocket за швидкістю, оскільки передбачає встановлення нового з'єднання після кожного отримання даних.

Окрім вимірювання затримки, було також проведено тестування продуктивності системи при високих навантаженнях. Результати показали, що WebSocket демонструє значно кращу стійкість та відгук порівняно з іншими технологіями. Завдяки ефективному управлінню з'єднаннями, WebSocket здатний

ефективно обробляти велику кількість одночасних користувачів без значного збільшення затримок або втрати з'єднань. Polling та Long Polling, навпаки, показали значне погіршення продуктивності зі збільшенням навантаження, що виражалося у збільшенні затримок та кількості помилок. SSE показав кращу стійкість ніж Polling та Long Polling, але все ж поступався WebSocket.

Ці результати однозначно підтверджують, що WebSocket є оптимальним вибором для розробки веб-додатків, які потребують обміну даними в режимі реального часу та високої продуктивності при великих навантаженнях.

3.3 Оптимізація запитів та використання серверних ресурсів

У процесі розробки веб-додатку важливим аспектом є ефективне використання бази даних та оптимізація її запитів для досягнення найкращої продуктивності. Для цього необхідно враховувати кілька стратегій, зокрема створення індексів на важливих полях таблиць і впровадження транзакцій для забезпечення цілісності та швидкості виконання операцій. Всі ці заходи дозволяють значно знизити навантаження на сервер та забезпечити коректну і швидку роботу з даними навіть при великих обсягах інформації.

Одним із основних напрямків оптимізації стало створення індексів на найбільш часто використовуваних полях таблиць. Індекси дозволяють пришвидшити операції вибірки даних, що є особливо важливим у контексті веб-додатків з великою кількістю користувачів і повідомлень. Наприклад, таблиця повідомлень є однією з найбільш важливих для роботи веб-чату, і саме тому особлива увага була приділена її оптимізації (рис. 3.25).

```
CREATE TABLE "messages" (
  "id" int NOT NULL AUTO_INCREMENT,
  "sender_id" int NOT NULL,
  "group_id" int DEFAULT NULL,
  "direct_chat_id" int DEFAULT NULL,
  "content" text,
  "created_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  "updated_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  "replied_message_id" int DEFAULT NULL,
  "is_edited" tinyint(1) DEFAULT '0',
  "deleted_at" timestamp NULL DEFAULT NULL,
  "status" enum('SENDING','SENT','DELIVERED','READ','FAILED') NOT NULL DEFAULT 'SENDING',
  "delivered_at" timestamp NULL DEFAULT NULL,
  "read_at" timestamp NULL DEFAULT NULL,
  "failed_at" timestamp NULL DEFAULT NULL,
  "failure_reason" varchar(255) DEFAULT NULL,
  PRIMARY KEY ("id"),
  KEY "idx_messages_created_at" ("created_at"),
  KEY "idx_messages_sender_id" ("sender_id"),
  KEY "idx_messages_direct_chat_id" ("direct_chat_id"),
  KEY "idx_messages_group_id" ("group_id"),
  CONSTRAINT "fk_messages_group_id" FOREIGN KEY ("group_id") REFERENCES "groups" ("id") ON DELETE CASCADE,
  CONSTRAINT "messages_ibfk_1" FOREIGN KEY ("sender_id") REFERENCES "users" ("id") ON DELETE CASCADE,
  CONSTRAINT "messages_ibfk_3" FOREIGN KEY ("direct_chat_id") REFERENCES "direct_chats" ("id") ON DELETE CASCADE,
  CONSTRAINT "messages_chk_1" CHECK (((`group_id` is not null) or (`direct_chat_id` is not null)))
)
```

Рисунок 3.25 – Індекси таблиці повідомлень

Одним з індексів, який було додано в таблицю повідомлень, є індекс на поле created_at. Це значно прискорює виконання запитів, що фільтрують або сортують повідомлення за часом їх створення. Такі запити є дуже поширеними, оскільки при перегляді історії чатів або фільтрації нових повідомлень дуже важливо швидко отримати потрібну інформацію за певним часом. Індекс на поле created_at дозволяє не лише прискорити ці запити, а й значно знизити навантаження на сервер під час виконання таких операцій.

Ще одним важливим індексом є індекс на поле sender_id, яке є ідентифікатором користувача, що відправив повідомлення. Це оптимізує запити, що фільтрують повідомлення за відправником. Це дуже корисно, коли користувач хоче переглянути всі свої повідомлення або повідомлення конкретного користувача. Веб-додатки часто мають функціонал, що дозволяє переглядати всі повідомлення певної особи, і цей індекс дозволяє зробити такі запити набагато швидшими.

Індекс на полі direct_chat_id був доданий для прискорення вибірки повідомлень для певного чату. Це важливо, коли користувач хоче переглянути всі повідомлення у приватному чаті з іншим користувачем. Індекс на цьому полі

дозволяє швидко отримати всі повідомлення, що належать конкретному чату, що забезпечує високу швидкість роботи додатку в приватних чатах.

Також був створений індекс на полі group_id, що дозволяє ефективно обробляти запити, що вибирають повідомлення для певних груп. Групові чати, як правило, містять велику кількість повідомлень, і саме тому важливо мати можливість швидко завантажити всі повідомлення для конкретної групи користувачів. Індекс на цьому полі значно пришвидшує цей процес, а також знижує навантаження на сервер при роботі з великою кількістю запитів.

Наступним важливим аспектом є впровадження транзакцій для обробки декількох операцій, що включають маніпуляції з кількома таблицями одночасно. Транзакції допомагають забезпечити консистентність даних та їх цілісність при виконанні складних операцій. Використання транзакцій дозволяє групувати кілька запитів в одну операцію, що забезпечує атомарність всього процесу. Якщо якась з операцій не може бути виконана, усі зміни, що були виконані до цього, скасовуються, і це дозволяє уникнути часткових змін у базі даних, що є критичним для збереження даних, таких як повідомлення або файли.

Одним із прикладів застосування транзакцій є процес створення або оновлення повідомлень у чатах. Коли потрібно додати нове повідомлення, ця операція зазвичай включає кілька кроків: вставлення даних в таблицю повідомлень, оновлення історії чату, можливо, додавання файлів, що прикріплені до повідомлення, і оновлення стану чатів чи груп. Для того щоб ці зміни були виконані цілісно, без помилок і не призвели до непередбачених наслідків, було використано транзакції (рис. 3.26).

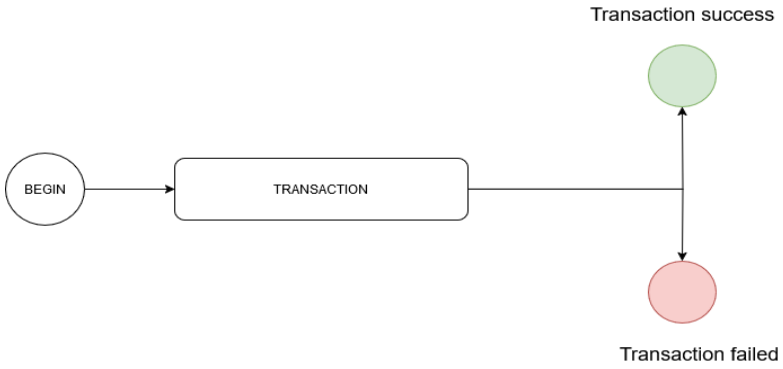


Рисунок 3.26 - Принцип транзакції

За допомогою транзакцій вдалося запобігти ситуаціям, коли частина змін була успішно виконана, а частина ні. Наприклад, якщо при додаванні повідомлення сталася помилка під час збереження файлів, транзакція дозволяє відкотити всі зміни і таким чином уникнути ситуацій, коли повідомлення додається, але файл не прикріплюється. Це важливо для підтримки цілісності даних і забезпечення стабільної роботи додатку.

Додатково, транзакції забезпечують кращу організацію роботи з кількома таблицями (рис. 3.27), що може бути важливим при обробці складних запитів, таких як робота з повідомленнями, користувачами, їх статусами та файлами.

```
async deleteMessagesByChatId(chatId, type = "direct") {
    const connection = await db.getConnection();
    try {
      await connection.beginTransaction();

      // Удаляем файлы сообщений
      await connection.execute(
        `DELETE mf FROM message_files mf
             JOIN messages m ON m.id = mf.message_id
             WHERE m.${type === "direct" ? "direct_chat_id" : "group_id"} = ?`,
        [chatId]
      );

      // Удаляем сами сообщения
      const [result] = await connection.execute(
        `DELETE FROM messages
             WHERE ${type === "direct" ? "direct_chat_id" : "group_id"} = ?`,
        [chatId]
      );

      await connection.commit();
```

Рисунок 3.27 – Код транзакції видалення повідомлення

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було реалізовано всі поставлені завдання, що дозволило досягти головної мети — підвищення ефективності навчального процесу через впровадження інтерактивного веб-додатка на базі WebSocket, що забезпечує високошвидкісну двосторонню взаємодію між студентами та викладачами в режимі реального часу з мінімальними затримками.

Під час роботи над проектом було проведено всебічний аналіз сучасних технологій та інструментів для розробки веб-додатків. Особливий акцент зроблено на протоколі WebSocket, який забезпечує постійне двостороннє з'єднання між клієнтом і сервером. У порівнянні з традиційними методами обміну даними, WebSocket продемонстрував суттєві переваги, зокрема високу швидкість обміну інформацією та зниження навантаження на сервери.

Проектування архітектури додатка включало розробку клієнтської та серверної частин, кожна з яких була реалізована з урахуванням сучасних принципів побудови веб-додатків. Для клієнтської частини було використано стек технологій, що включає React для побудови інтерфейсу, Redux для управління станом, TypeScript для типізації коду та Tailwind CSS для стилізації. Такий підхід забезпечив високу продуктивність, легкість у розробці та масштабуванні інтерфейсу.

Серверна частина була реалізована за допомогою Node.js, Express та Socket.IO. Для зберігання даних використано MySQL, що дозволило організувати структуроване та надійне зберігання інформації. Значну увагу приділено безпеці: реалізовано авторизацію користувачів із використанням токенів, що забезпечує захищений доступ до функціоналу додатка.

Однією з ключових частин роботи стало впровадження функціоналу веб-чату, а саме:

– Авторизацію користувачів, що передбачає перевірку даних на сервері, захист паролів і використання токенів для сесій.

– Обмін повідомленнями в режимі реального часу, що базується на двосторонній комунікації між клієнтом і сервером.

– Створення та управління чатами, які можуть бути як груповими, так і приватними.

– Інтеграцію медіафайлів, що дозволяє прикріплювати до повідомлень зображення, документи та інші типи файлів.

– Функціонал чорного списку, який забезпечує можливість блокування небажаних контактів.

Проведено комплексну перевірку функціоналу, що включала тестування авторизації, обміну повідомленнями, створення та управління чатами, інтеграцію медіафайлів, роботу з чорним списком і базою даних.

Оптимізація запитів до бази даних стала ще одним важливим досягненням. Завдяки впровадженню ефективних алгоритмів обробки даних та раціональному використанню серверних ресурсів вдалося знизити час обробки запитів і забезпечити стабільну роботу системи.

Порівняння ефективності WebSocket з іншими технологіями показало значні переваги. Порівняно з SSE, WebSocket продемонстрував зниження затримки на 70-80%. Порівняно з Polling та Long Polling, WebSocket дозволив знизити затримки на 90%. Polling створює додаткові затримки через циклічні запити, а Long Polling поступається через необхідність повторного встановлення з'єднань.

Таким чином, результати роботи свідчать про доцільність використання сучасних технологій, таких як React, Node.js та WebSocket, для створення інтерактивних навчальних платформ. Розроблений веб-додаток може бути впроваджений у навчальний процес для покращення взаємодії між викладачами та студентами. Він сприяє підвищенню ефективності навчання, забезпечує зручність використання та відповідає сучасним вимогам до цифрових освітніх рішень.

ПЕРЕЛІК ПОСИЛАНЬ

1. Russell J.T. Dyer. *Learning MySQL and MariaDB* [Електронний ресурс] : [Книга].
2. Romaniszyn K., Ogorzałek P. Automated mathematical models [Електронний ресурс] : [Журнал]. – Електронні дані. – Режим доступу: https://science.lpnu.ua/sites/default/files/journal-paper/2020/mar/21130/amm-2-27-33.pdf (дата звернення 01.12.2024).
3. «AJAX Introduction», W3Schools UA. [Електронний ресурс] : [Веб-сайт]. – Електронні дані. – Режим доступу: https://w3schoolsua.github.io/js/js_ajax_intro.html#gsc.tab=0 (дата звернення 01.12.2024).
4. Ilya Kantor. Learn JavaScript: WebSocket Protocol [Електронний ресурс] : [Посібник]. – Електронні дані. – Режим доступу: https://learn.javascript.ru/websocket (дата звернення 01.12.2024).
5. Tailwind Labs. Tailwind CSS Documentation [Електронний ресурс] : [Веб-сайт]. – Електронні дані. – Режим доступу: https://tailwindcss.com/docs/installation (дата звернення 01.12.2024).
6. SASS Documentation. [Електронний ресурс] : [Веб-сайт]. – Електронні дані. – Режим доступу: https://sass-lang.com/ (дата звернення 01.12.2024).
7. Spring Framework Guides: Serving Web Content. [Електронний ресурс] : [Посібник]. – Електронні дані. – Режим доступу: https://spring.io/guides/gs/serving-web-content (дата звернення 01.12.2024).
8. MySQL Documentation. [Електронний ресурс] : [Документація]. – Електронні дані. – Режим доступу: https://dev.mysql.com/doc/ (дата звернення 01.12.2024).
9. Percona Team. Optimizing MySQL Queries for Performance [Електронний ресурс] : [Стаття]. – Електронні дані. – Режим доступу: https://www.percona.com/blog/2018/07/25/optimizing-mysql-queries-for-performance/ (дата звернення 01.12.2024).

10. Socket.IO Documentation. [Електронний ресурс] : [Документація]. – Електронні дані. – Режим доступу: https://socket.io/docs/ (дата звернення 01.12.2024).
11. Wojtek Maj. React Lifecycle Methods Diagram [Електронний ресурс] : [Діаграма]. – Електронні дані. – Режим доступу: https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/ (дата звернення 01.12.2024).
12. React Documentation: Component Lifecycle Methods. [Електронний ресурс] : [Документація]. – Електронні дані. – Режим доступу: https://uk.legacy.reactjs.org/docs/react-component (дата звернення 01.12.2024).

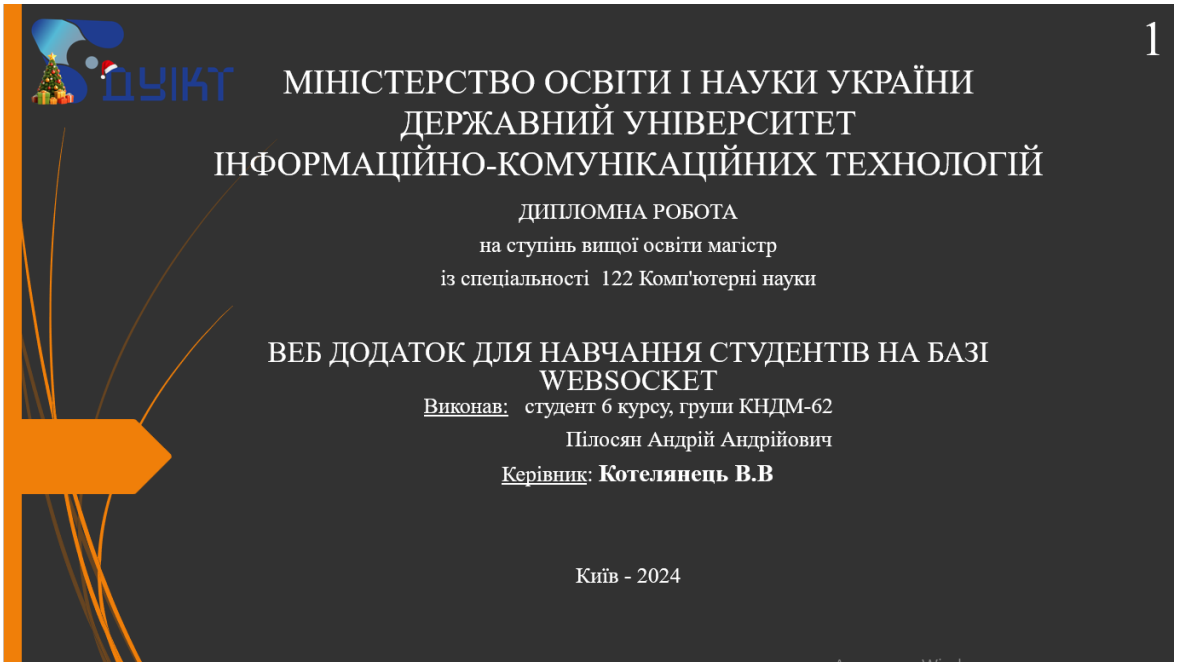
ДОДАТОК А. КЛАС FILE

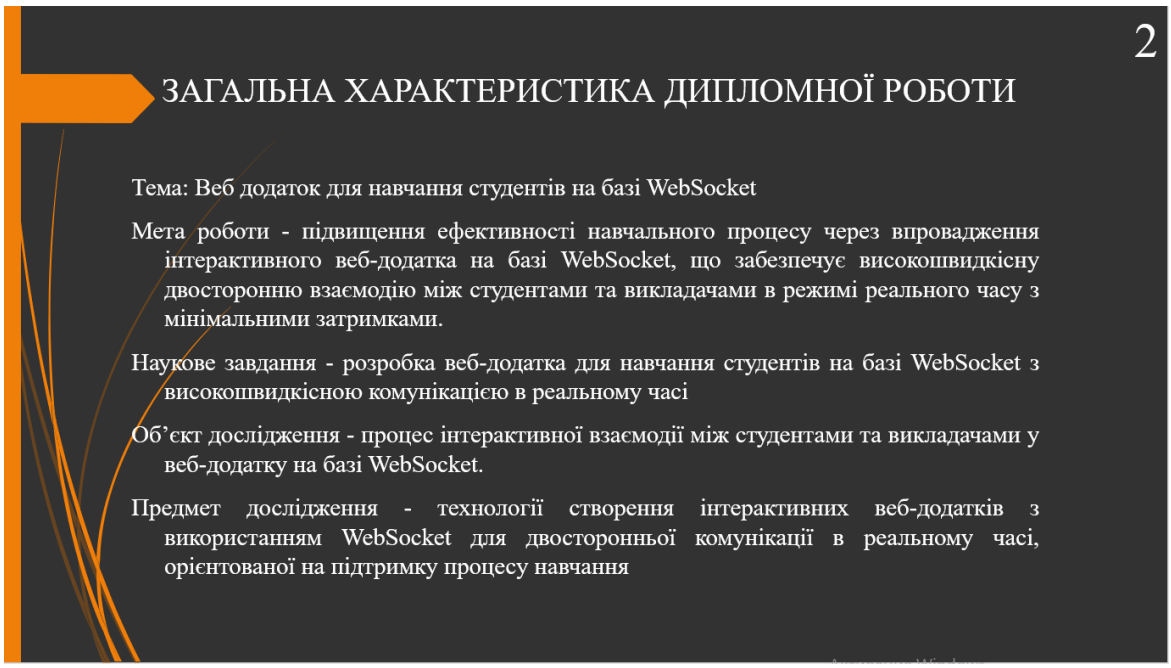
```
import db from "../db.js";
import ApiError from "../exceptions/api-error.js";
import fs from "fs";
import path from "path";
class File {
  async getGroupIcon(groupId) {
    try {
      const [file] = await db.execute(
        `SELECT f.* FROM files f
            JOIN groups g ON g.icon_id = f.id
            WHERE g.id = ?`,
        [groupId]
      );
      return file[0];
    } catch (error) {
      throw ApiError.BadRequest("Error fetching group icon", error);
    }
  }
  async saveIcon(userId, filename, originalName, mimetype, size, fileHash) {
    try {
      const [existingFile] = await db.execute(
        "SELECT id, filename FROM files WHERE file_hash = ? LIMIT 1",
        [fileHash]
      );
      if (existingFile.length !== 0) {
        await fs.promises.unlink(path.join("/uploads", filename));
        return existingFile[0].id;
```

```
      const [result] = await db.execute(
        "INSERT INTO files (user_id, filename, original_name, file_type, file_size, file_hash) VALUES (?, ?, ?, ?, ?, ?)",
        [userId, filename, originalName, mimetype, size, fileHash]
      );
      return result.insertId;
    } catch (error) {
      throw ApiError.BadRequest("Error in save icon", [error]);
    }
  }
  async saveFileRecord(
    senderId,
    filename,
    originalName,
    mimetype,
    size,
    fileHash
  ) {
    try {
      const [existingFile] = await db.execute(
        "SELECT id, filename FROM files WHERE file_hash = ? LIMIT 1",
        [fileHash]
      );
      if (existingFile.length !== 0) {
        await fs.promises.unlink(path.join("/uploads", filename));
        return existingFile[0].id;
      }
      const [result] = await db.execute(
        "INSERT INTO files (user_id, filename, original_name, file_type, file_size, file_hash, uploaded_at) VALUES (?, ?, ?, ?, ?, ?, NOW())",
```

```
        [senderId, filename, originalName, mimetype, size, fileHash]
      );
      return result.insertId;
    } catch (error) {
      if (error instanceof ApiError) {
        throw error;
      } else {
        throw ApiError.BadRequest("Error in save file", [error]);
      }
    }
  }
  async getFileById(fileId) {
    try {
      const [file] = await db.execute("SELECT * FROM files WHERE id = ?", [
        fileId,
      ]);
      if (file.length < 1) {
        throw ApiError.ForbiddenError(`File don't finded`);
      }
      return file[0];
    } catch (error) {
      if (error instanceof ApiError) {
        throw error;
      } else {
        throw ApiError.BadRequest("Database error or unknown error", [error]);
      }
    }
  }
}
export default new File();
```

ДОДАТОК Б. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



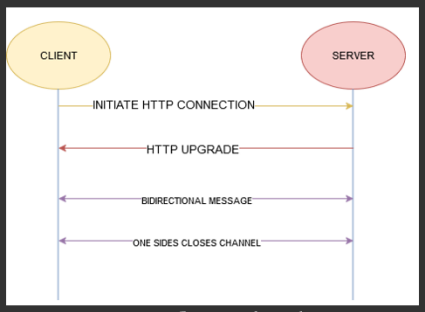


Основні завдання роботи:

1. Провести аналіз сучасних технологій та інструментів для створення веб-додатків із використанням WebSocket.
2. Розробити архітектуру веб-додатка, яка включає клієнтську та серверну частини.
3. Реалізувати функціонал веб-додатка
4. Провести тестування клієнтської та серверної частин для перевірки їхньої надійності та продуктивності.
5. Оптимізувати запити до бази даних для підвищення продуктивності та забезпечити ефективне використання серверних ресурсів.

Аналіз сучасних технологій

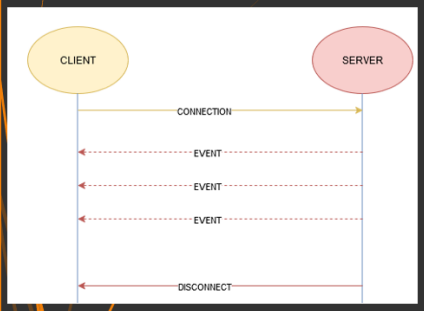
- Сучасні веб-додатки на базі WebSocket представляють собою потужний інструмент для створення інтерактивних систем з двосторонньою взаємодією в реальному часі. На відміну від традиційних HTTP-запитів, WebSocket забезпечує постійне з'єднання між клієнтом і сервером, що дозволяє миттєво обмінюватися даними без додаткових затримок.
- Основою таких додатків є динамічна взаємодія, яка реалізується через комбінацію AJAX для асинхронних запитів та WebSocket для постійного зв'язку. Це дозволяє створювати responsive-інтерфейси, які миттєво реагують на дії користувача без необхідності перезавантаження сторінки.
- Сучасний технологічний стек для розробки включає потужні інструменти як React для побудови користувацького інтерфейсу, Redux для управління станом, TypeScript для надійної типізації, та Socket.IO, Node.js з Express.js на серверній частині. Така комбінація технологій забезпечує високу продуктивність та масштабованість додатків.
- WebSocket має значні переваги порівняно з альтернативними підходами, такими як Server-Sent Events чи різні види polling. Протокол забезпечує справжню двосторонню комунікацію, мінімальні затримки та ефективне використання ресурсів, що робить його ідеальним вибором для чатів, онлайн-ігор, торгових платформ та інших додатків, де критична швидкість обміну даними.
- Така архітектура дозволяє створювати сучасні, швидкі та масштабовані веб-додатки, які відповідають зростаючим вимогам користувачів до інтерактивності та швидкодії онлайн-сервісів.



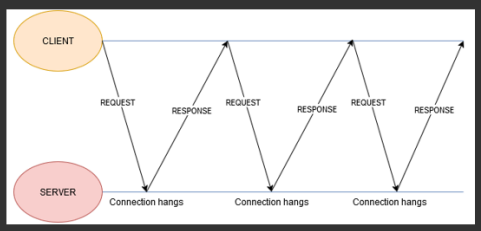
Приклад роботи WebSocket

Аналіз сучасних технологій

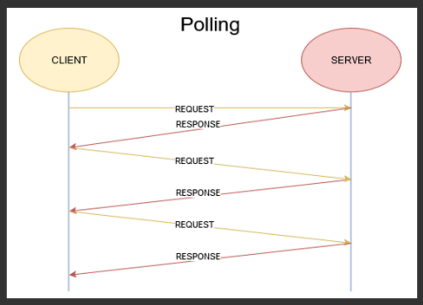
АЛЬТЕРНАТИВИ



Принцип роботи Server-Sent Events



Принцип роботи Long Polling

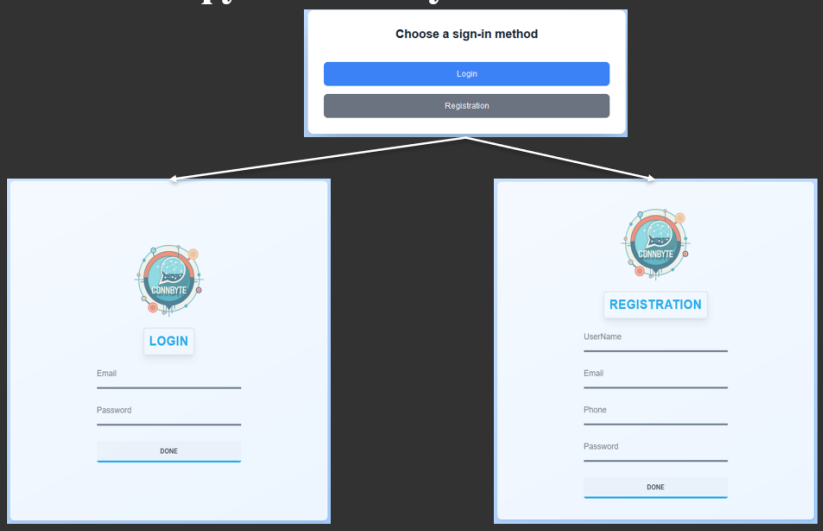


Принцип роботи Polling

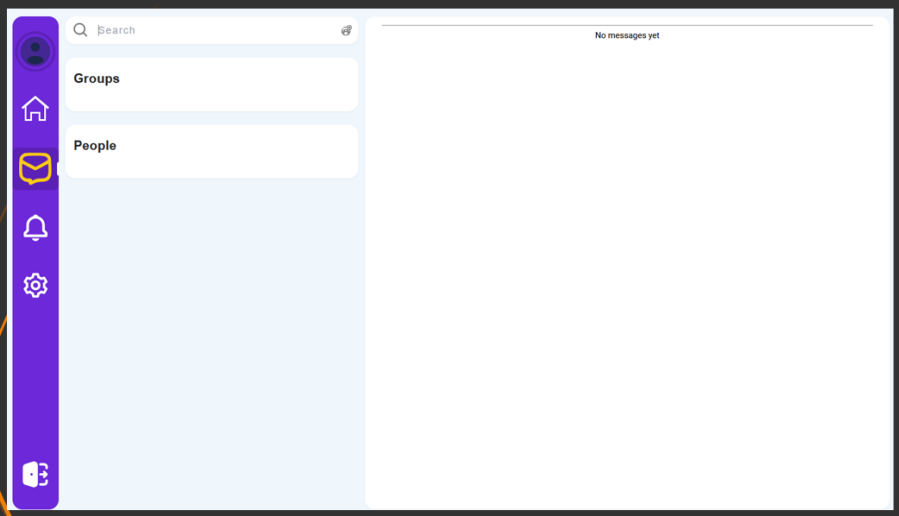
Архітектура веб-додатка

- Компоненти:
  - Клієнтська частина: React, Redux, TypeScript.
  - Серверна частина: Node.js, Express, Socket.IO.
  - База даних: MySQL.
- Особливості:
  - JWT для авторизації.
  - Підтримка групових і приватних чатів за допомогою Socket.IO.

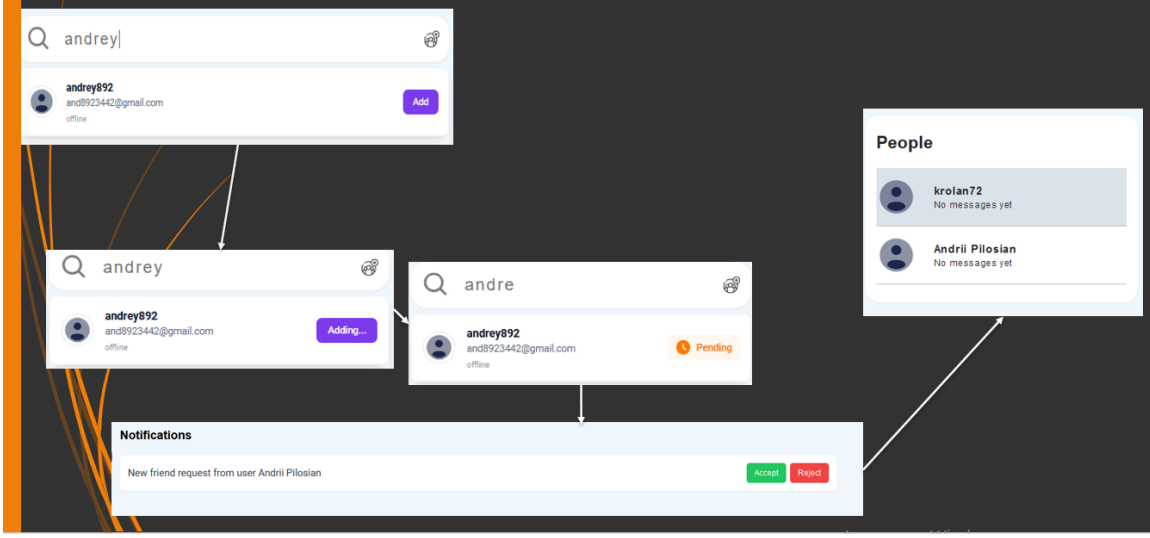
Реалізація функціоналу



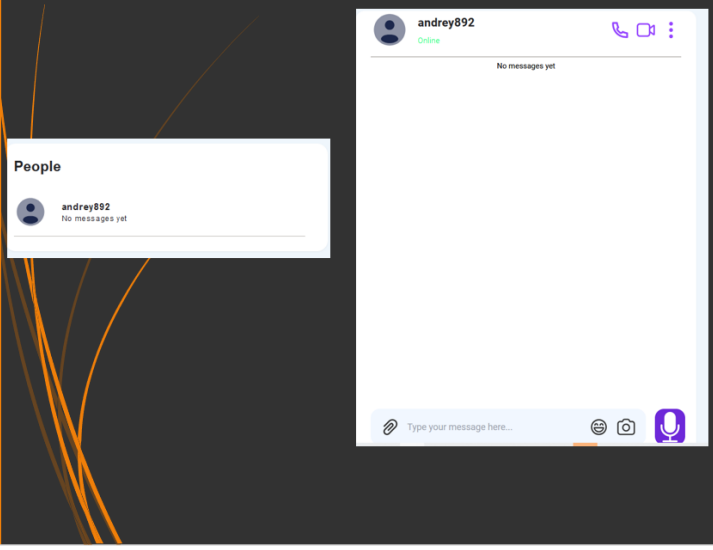
Реалізація функціоналу



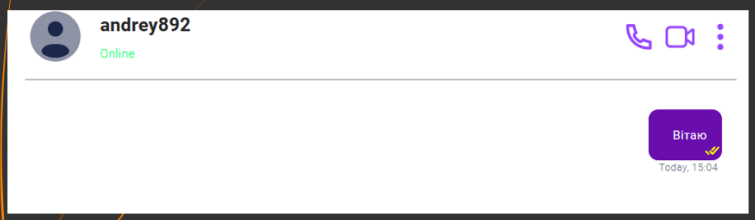
Реалізація функціоналу

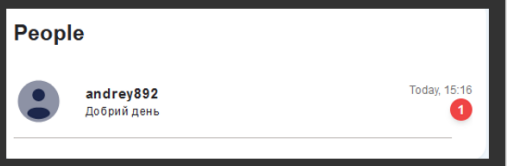


Реалізація функціоналу



Реалізація функціоналу





Реалізація функціоналу

SENDING

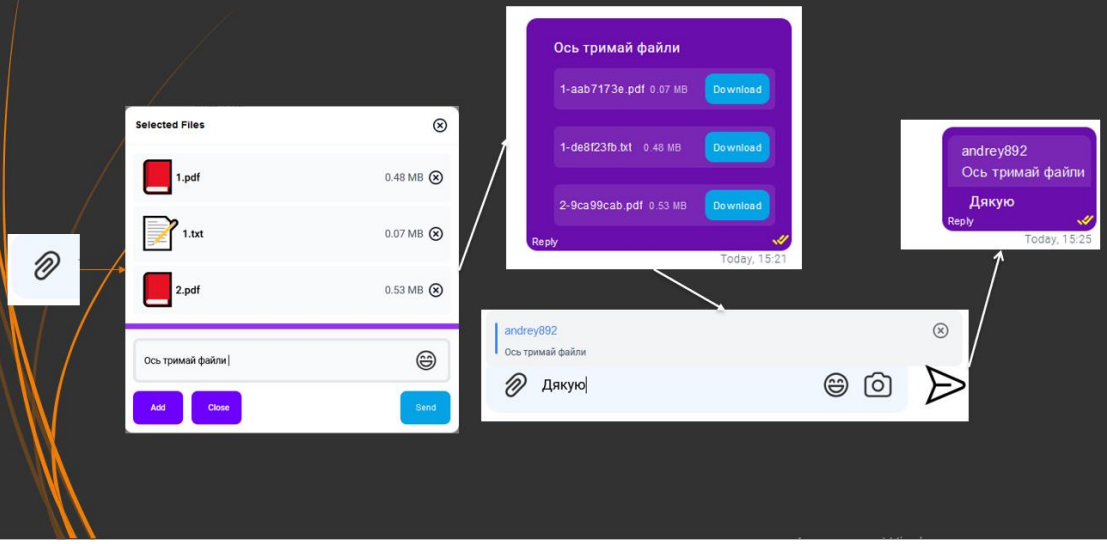
FAILED

SENT

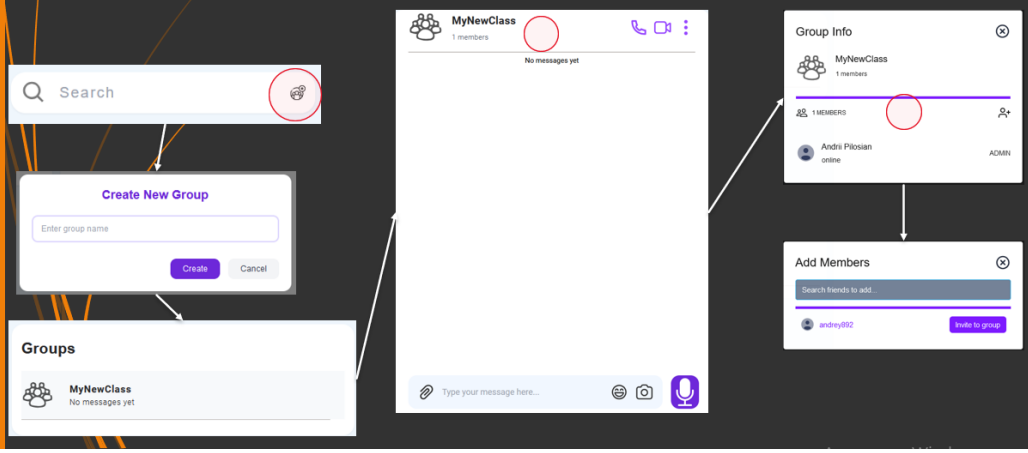
DELIVERED

READ

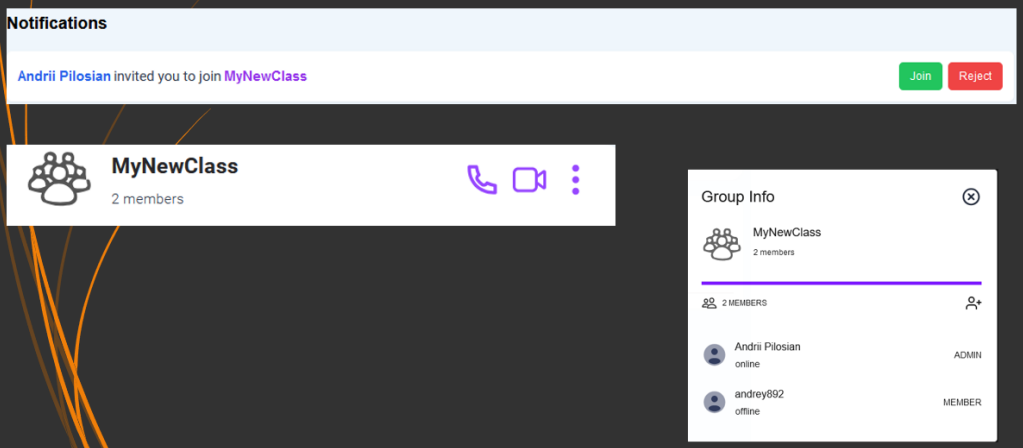
Реалізація функціоналу



Реалізація функціоналу



Реалізація функціоналу



Оптимізація запитів до бази данних

```
CREATE TABLE "messages" (
  "id" int NOT NULL AUTO_INCREMENT,
  "sender_id" int NOT NULL,
  "group_id" int DEFAULT NULL,
  "direct_chat_id" int DEFAULT NULL,
  "content" text,
  "created_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  "updated_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  "replied_message_id" int DEFAULT NULL,
  "is_edited" tinyint(1) DEFAULT '0',
  "deleted_at" timestamp NULL DEFAULT NULL,
  "status" enum('SENDING','SENT','DELIVERED','READ','FAILED') NOT NULL DEFAULT 'SENDING',
  "delivered_at" timestamp NULL DEFAULT NULL,
  "read_at" timestamp NULL DEFAULT NULL,
  "failed_at" timestamp NULL DEFAULT NULL,
  "failure_reason" varchar(255) DEFAULT NULL,
  PRIMARY KEY ("id"),
  KEY "idx_messages_created_at" ("created_at"),
  KEY "idx_messages_sender_id" ("sender_id"),
  KEY "idx_messages_direct_chat_id" ("direct_chat_id"),
  KEY "idx_messages_group_id" ("group_id"),
  CONSTRAINT "fk_messages_group_id" FOREIGN KEY ("group_id") REFERENCES "groups" ("id") ON DELETE CASCADE,
  CONSTRAINT "messages_ibfk_1" FOREIGN KEY ("sender_id") REFERENCES "users" ("id") ON DELETE CASCADE,
  CONSTRAINT "messages_ibfk_3" FOREIGN KEY ("direct_chat_id") REFERENCES "direct_chats" ("id") ON DELETE CASCADE,
  CONSTRAINT "messages_chk_1" CHECK ((((`group_id` is not null) or (`direct_chat_id` is not null))))
)
```

```
async deleteMessagesByChatId(chatId, type = "direct") {
    const connection = await db.getConnection();
    try {
      await connection.beginTransaction();

      // Удаляем файлы сообщений
      await connection.execute(
        `DELETE mf FROM message_files mf
             JOIN messages m ON m.id = mf.message_id
             WHERE m.${type === "direct" ? "direct_chat_id" : "group_id"} = ?`,
        [chatId]
      );

      // Удаляем сами сообщения
      const [result] = await connection.execute(
        `DELETE FROM messages
             WHERE ${type === "direct" ? "direct_chat_id" : "group_id"} = ?`,
        [chatId]
      );

      await connection.commit();
```

Висновки

У ході роботи реалізовано всі завдання, що дозволило досягти мети — створення інтерактивного веб-додатка для навчання з використанням WebSocket. Проведено аналіз технологій, розроблено архітектуру, реалізовано функціонал чату, включаючи авторизацію, обмін повідомленнями, управління чатами, інтеграцію медіа файлів і чорний список. Оптимізація бази даних та тестування підтвердили високу продуктивність і надійність системи. WebSocket продемонстрував значні переваги в швидкості обміну даними порівняно з іншими технологіями.