Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп’ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента	*Садиченка Данила Валерійовича*
	(ПІБ)
академічної групи	*122М-22-1*
	(шифр)
спеціальності	*122 Комп’ютерні науки*
	(код і назва спеціальності)
освітньої програми	*«122 Комп’ютерні науки»*
	(назва освітньої програми)
на тему:	*Дослідження інтерактивної чат платформи для організації комунікації між користувачами з використанням геолокації*

___________________ *Д.В. Садиченко*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
розділів кваліфікаційної роботи				
спеціальний	*доц.Спірінцев В.В.*			
економічний				

Рецензент				

Нормоконтролер	*проф. Лактіонов І.С.*			

Дніпро
2023

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

______________ І.М. Удовик
(підпис) (прізвище, ініціали)

«   » __________ 20 23 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

студенту *122м-22-1* *Садиченка Данила Валерійовича*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Дослідження інтерактивної чат платформи для організації комунікації між користувачами з використанням геолокації*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 09.10.2023 р. № 1227-с.

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процес прийому, передачі та обробки інформації в режимі реального часу, в системах побудованих на основі протоколу комунікації WebSocket.

Предмет досліджень – методи організації роботи інтерактивної чат-платформи з використанням геолокації та протоколу комунікації WebSocket.

Мета НДР – підвищення якості спілкування та знайомства між новими користувачами, за рахунок розробки і використання чат-додатку, де використання геолокації виступає в якості ключової функціональної особливості.

Вихідні дані для проведення роботи – теоретичні та експериментальні дослідження, основи систем прийому, передачі та обробки інформації в режимі реального часу, побудованих на основі протоколу комунікації WebSocket.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Новизна запропонованих рішень полягає в тому, що запропоновано новий підхід для підвищення якості спілкування та знайомств користувачів в мережі Інтернет з використанням геолокації. Розроблена інтерактивна чат-платформа, на відміну від існуючих, враховує геолокацію користувача, що дозволяє підвищити якість спілкування за рахунок додавання користувача до чату з іншими користувачами які знаходяться в тій же самій країні або місті.

Практична цінність результатів полягає в тому, що запропонований чат-додаток дозволяє полегшити спілкування та знайомства в нових для користувача країнах та містах.

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Результати досліджень мають бути подані у вигляді, що дозволяє побачити та оцінити безпосереднє використання геолокації користувача для покращення якості спілкування та знайомств в мережі Інтернет. В результаті роботи повинен бути розроблений додаток інтерактивної чат платформи для організації комунікації та знайомств між користувачами на основі використання геолокації.

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – Кінець**
Аналіз теми та постановка задачі	12.09.2020-30.09.2023
Аналіз засобів створення програмного забезпечення для проектування інтерактивної чат платформи для для організації комунікації між користувачами з використанням геолокації.	01.10.2020-15.11.2023
Розробка програмного забезпечення та дослідження ефективності запропонованих рішень.	16.11.2020-04.12.2023

6 РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект від реалізації результатів роботи очікується позитивним завдяки підвищенню якості спілкування між новими користувачами в різних країнах та містах, що в свою чергу зменшить потенційні витрати користувачів при отриманні різноманітної інформації стосовно країни або міста в якому розташован користувач.

Соціальний ефект від реалізації результатів роботи очікується позитивним завдяки застосуванню геолокації в якості ключової функціональної особливості розробляємого додатку, що в свою дозволяє покращити спілкування користувача в новій для нього країні.

7 ДОДАТКОВІ ВИМОГИ

Завдання видав	____________ (підпис)	доц. Спірінцев В.В. (посада, прізвище, ініціали)
Завдання прийняв до виконання	____________ (підпис)	Садиченко Д.В. (прізвище, ініціали)

Дата видачі завдання: *09.10.2023 р.*

Термін подання кваліфікаційної роботи до ЕК: *04.12.2023*

РЕФЕРАТ

Пояснювальна записка: 116 стор., 25 рис., 2 додатка, 24 джерел.

Об'єкт дослідження: процес прийому, передачі та обробки інформації в режимі реального часу, в системах побудованих на основі протоколу комунікації WebSocket.

Предмет дослідження: методи організації роботи інтерактивної чат-платформи з використанням геолокації та протоколу комунікації WebSocket.

Мета роботи: підвищення якості спілкування та знайомства між новими користувачами, за рахунок розробки і використання чат-додатку, де використання геолокації виступає в якості ключової функціональної особливості.

Методи дослідження базуються на основних принципах системного аналізу, функціонального аналізу, теорії баз даних. Використано методи структурного моделювання, теоретичні основи проектування реляційних баз даних, теоретичні основи побудови сховищ даних.

Новизна отриманих результатів полягає в тому, що запропоновано новий підхід для підвищення якості спілкування та знайомств користувачів в мережі Інтернет з використанням геолокації. Розроблена інтерактивна чат-платформа, на відміну від існуючих, враховує геолокацію користувача, що дозволяє підвищити якість спілкування за рахунок додавання користувача до чату з іншими користувачами які знаходяться в тій же самій крані або місті.

Практична цінність результатів полягає в тому, що запропонований чат-додаток дозволяє полегшити спілкування та знайомства в нових для користувача країнах та містах за рахунок використання геолокації в якості ключової функціональної особливості.

Область застосування. Розроблена система може використовуватися будь яким користувачем для поліпшення якості спілкування в умовах знаходження у новій для користувача країні або місті.

Значення роботи та висновки. Використання геолокації як ключової функціональної особливості розробленого чат-додатку має великий вплив на покращення якості спілкування та знайомства між користувачами, дозволяючи швидко находити співрозмовників в умовах перебування в новій для користувача країні.

Прогнози щодо розвитку досліджень. Покращити розроблений додаток, додавши можливість відправляти медіа файли в чат кімнатах, та можливість писати іншим користувачам приватні повідомлення.

Список ключових слів: додаток, програмна реалізація, HTML, WebSocket, API, HTTP, React.

ABSTRACT

Explanatory note: 116 pages, 25 figures, 2 applications, 24 sources.

Object of research: the process of real-time information reception, transmission, and processing in systems built on the WebSocket communication protocol.

Subject of research: Methods for organizing the operation of an interactive chat platform using geolocation and WebSocket communication protocol.

Purpose of Master's thesis: to enhance the quality of communication and networking among new users through the development and use of a chat application, where geolocation serves as a key functional feature.

Research methods are based on the fundamental principles of systems analysis, functional analysis, and database theory. Structural modeling methods, theoretical foundations of relational database design, and theoretical foundations of data warehousing are applied.

Originality of research is in the the fact that a new approach has been proposed to enhance the quality of communication and user networking on the Internet using geolocation. The developed interactive chat platform, unlike existing ones, takes into account the user's geolocation, allowing to improve the quality of communication by adding the user to a chat with others who are in the same country or city.

The practical value of the results is that the proposed chat application facilitates communication and networking in new countries and cities for the user.

Scope of application. The developed system can be used by any user to improve the quality of communication while in a new country or city for the user.

The value of the work and conclusions. The use of geolocation as a key functional feature in the developed chat application has a significant impact on improving the quality of communication and networking among users, allowing them to quickly find conversation partners in a new country.

Research forecast and development. Enhance the developed application by adding the ability to send media files in chat rooms and the ability to send private messages to other users.

Keywords: application, software implementation, HTML, WebSocket, API, HTTP, React.

ЗМІСТ

РЕФЕРАТ 4
ABSTRACT 5
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 8
ВСТУП 9
РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 13
1.1. Загальні відомості з предметної галузі 13
1.1.1. Історія розвитку чатів та месенджерів 13
1.1.2. Вплив чатів та месенджерів на сучасний світ 15
1.1.3. Технології для розробки чат-платформ та месенджерів 16
1.2. Аналіз існуючих інтерактивних чат-платформ та месенджерів 19
1.3. Висновки до першого розділу 24
РОЗДІЛ 2 МЕТОДИ ТА ТЕХНОЛОГІЇ ВИРІШЕННЯ ЗАДАЧІ 26
2.1. Аналіз та порівняння технологій React, Vue.js та Angular 26
2.1.1. Бібліотека React 26
2.1.2. Фрейморк Vue.js 29
2.1.3. Фреймворк Angular 31
2.1.4. Порівняння React, Angular та Vue.js 33
2.2. Мова програмування Node.js та фреймворк Express.js 36
2.3. WebSocket та бібліотека Socket.io 39
2.4. PostgreSQL 41
2.5. Висновки до другого розділу 42
РОЗДІЛ 3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ЙОГО ЗАСТОСУВАННЯ 44
3.1. Опис структури системи та алгоритмів її функціонування 44
3.2. Опис інтерфейсу користувача 59

3.3. Дослідження ефективності роботи розробленого додатку 66
ВИСНОВКИ 70
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 72
ДОДАТОК А 75
ДОДАТОК Б 112
ДОДАТОК В 114
ДОДАТОК Г 116

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

HTTP - HyperText Transfer Protocol

HTML - HyperText Markup Language

CSS - Cascading Style Sheets

DOM - Document Object Model

API - Application Programming Interface

SPA - Single Page Application

JSX - JavaScript XML

XML - Extensible Markup Language

AJAX - Asynchronous JavaScript And XML

ВСТУП

Сучасний світ відзначається стрімким розвитком інформаційних технологій, що неминуче впливає на нашу поведінку та спосіб спілкування. Завдяки швидким змінам та поширенню мобільних пристроїв, люди тепер мають змогу легко знаходити спільну мову та об'єднуватися в однакових інтересах. Однак, при всій розкоші інтернет-спілкування часто відсутній той контекст, який може забезпечити зустрічі у реальному житті. Це змушує нас задуматися про нові способи зміцнення в'язків та знайомств у віртуальному просторі.

В рамках цієї кваліфікаційної роботи пропонується дослідження і розробка інтерактивної чат-платформи з елементами месенджера, спрямованої на полегшення знайомств та спілкування між користувачами. Інноваційним аспектом цієї платформи є використання геолокації, що дозволить користувачам знаходити співрозмовників у своєму регіоні, об'єднуючи віртуальний та реальний світи. Це створює унікальну можливість не лише спілкуватися онлайн, але й легко організовувати зустрічі в реальному житті, розширюючи коло знайомств та соціальних контактів.

Актуальність теми. Актуальність даної теми визначається насамперед сучасними тенденціями у сфері інтернет-спілкування та знайомств. Додавання до чату потенційних співрозмовників на основі геолокації відкриває нові можливості для знайомств та спілкування, особливо для тих, хто прагне об'єднувати віртуальний та реальний світи. Актуальність теми також підкреслюється реаліями сучасного ринку чат-рішень, де, незважаючи на розмаїття пропозицій, використання геолокації у контексті спілкування залишається недостатньо висвітленим. Існує багато чат-платформ, але більшість із них не використовують геолокацію як ключовий елемент для забезпечення якісних і цікавих інтернет-спілкувань та знайомств. Впровадження такої функціональності в чат-додаток може відкрити нові можливості для користувачів, сприяючи об'єднанню віртуального та реального світів, що стає

особливо важливим у контексті сучасних тенденцій інтернет-спілкування та знайомств.

Мета та завдання дослідження. Ця кваліфікаційна робота ставить за мету дослідження підвищення якості спілкування та знайомства між новими користувачами, за рахунок розробки і використання чат-додатку, де використання геолокації виступає в якості ключової функціональної особливості. Основна ідея полягає в створенні інтерактивної платформи для спілкування та знайомств, яка дозволить користувачам взаємодіяти на основі їхнього місцеположення.

Для досягнення мети дослідження необхідно розв'язати наступні задачі:

- Аналіз існуючих рішень. Провести аналіз існуючих інтерактивних чат-платформ та месенджерів, а також дослідити їхні функціональні можливості та особливості використання геолокації для організації комунікації.
- Розробка концепції платформи. Розробити концепцію інтерактивної чат-платформи, яка включає в себе використання геолокації для розташування користувачів у комунікативних чатах та створення власних чат-кімнат.
- Вибір технологій і інструментів. Обрати необхідні технології та інструменти для розробки веб-платформи, зокрема вибрати мову програмування, фреймворки, базу даних та підхід для роботи з геолокацією.
- Розробка основного функціоналу. Створити основний функціонал платформи, включаючи авторизацію користувачів, розміщення їх у комунікативних чатах в залежності від геолокації та можливість створення власних чат-кімнат.
- Інтерфейс користувача. Розробити зручний інтерфейс користувача, який дозволить користувачам легко взаємодіяти з платформою та використовувати її функціонал.

- Розробити систему контролю доступу, яка буде обмежувати доступ користувачів до функціональності чи контенту за певними умовами.
- Забезпечити взаємодію клієнту з сервером використовуючи HTTP та WebSocket.

Об'єктом дослідження є процес прийому, передачі та обробки інформації в режимі реального часу, в системах побудованих на основі протоколу комунікації WebSocket.

Предметом дослідження є методи організації роботи інтерактивної чат-платформи з використанням геолокації та протоколу комунікації WebSocket.

Методи дослідження базуються на основних принципах системного аналізу, функціонального аналізу, теорії баз даних. Використано методи структурного моделювання, теоретичні основи проектування реляційних баз даних, теоретичні основи побудови сховищ даних.

Новизна цієї кваліфікаційної роботи полягає в тому, що запропоновано новий підхід для підвищення якості спілкування та знайомств користувачів в мережі Інтернет з використанням геолокації. Розроблена інтерактивна чат-платформа, на відміну від існуючих, враховує геолокацію користувача, що дозволяє підвищити якість спілкування за рахунок додавання користувача до чату з іншими користувачами які знаходяться в тій же самій крані або місті.

Практична значення цієї роботи полягає в тому, що розроблена чат-платформа може бути використана в різних сферах та мати конкретний позитивний вплив на життя користувачів та суспільства в цілому. Основні аспекти практичної цінності цієї роботи включають:

- Зручний інструмент для знайомств та спілкування. Розроблена платформа надає користувачам можливість знаходити і спілкуватися з новими людьми, які знаходяться в їхньому регіоні, сприяючи розширенню соціального кола і створенню нових дружб та зв'язків.
- Підтримка інтересів і прагнень. Додаток може стати цінним інструментом для подорожуючих, людей, які шукають спів-мешканців або подібних інтересів, а також для організації подій та зустрічей в реальному світі.

- Вплив на культурні та соціальні взаємодії. За допомогою цієї платформи можуть розвиватися нові культурні і соціальні ініціативи, які сприяють розширенню кругу друзів і розумінню інших культур.
- Застосування для бізнесу. Розроблена платформа може бути використана для просування подій, брендів, ресторанів, та інших підприємств, що залежать від взаємодії з локальною аудиторією.

Цей підхід відкриває нові можливості для людей, які шукають спільноту або нових друзів в конкретному регіоні, для подорожуючих, які хочуть знайти місцевих екскурсоводів або співмешканців для спільного відпочинку, і для багатьох інших сценаріїв.

Особистий внесок автора. Було створено інтерактивну чат-платформу, головною функціональною особистістю якої є використання геолокації для автоматичного додавання користувача для місцевих чатів. Також було здійснено дослідження ефективності роботи розробленого додатку в умовах великого навантаження за використанням бібліотеки Artillery.

Структура та обсяг кваліфікаційної роботи. Відповідно до мети, завдань, і предмета дослідження, кваліфікаційна робота складається з реферату, вступу, трьох основних розділів і висновків, списку використаних джерел та 4 додатків. Загальний обсяг роботи містить 116 сторінок друкованого тексту, із них основної частини - 13 сторінок з 2 рис., спеціальної – 44 сторінок з 23рис., списку використаних джерел з 24 найменуванням на 2 сторінках, 4 додатках на 41 сторінках.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Загальні відомості з предметної галузі

В сучасному світі зростає роль веб-платформ для соціальної комунікації та знайомств, яка в значній мірі визначається розвитком інформаційних технологій та доступністю мобільних пристроїв. Споживачі стають все більше залежними від цифрових рішень для задоволення своїх потреб у спілкуванні та знайомствах. Відсутність географічних обмежень та доступ до різноманітних функцій роблять веб-платформи ідеальними інструментами для задоволення цих потреб.

1.1.1. Історія розвитку чатів та месенджерів

Зародження чатів та месенджерів пов'язане з розвитком Інтернету та технологічними досягненнями. Перші інтернет-чати з'явилися в кінці 20-го століття та надали користувачам можливість обмінюватися текстовими повідомленнями в реальному часі. За допомогою чатів стали можливі спонтанні обговорення, спілкування з друзями та навіть знайомства з незнайомими користувачами.

Однак справжній прорив в соціальному спілкуванні став можливим завдяки розвитку месенджерів, таких як ICQ, AIM, MSN Messenger та інших. Запуск ICQ у 1996 році відкрив шлях для мільйонів користувачів для обміну миттєвими повідомленнями. Це був перший крок до масового прийому месенджерів, які зробили соціальне спілкування більш доступним та зручним.

ICQ (Internet Relay Chat) — це один з перших інтернет-месенджерів, який був розроблений і випущений 1996 року компанією Mirabilis. ICQ швидко став популярним і на піку своєї популярності в 2000 році мав понад 300 мільйонів

користувачів у всьому світі. ICQ пропонував широкий спектр функцій, включаючи:

– обмін текстовими повідомленнями;
– обмін файлами;
– створення групових чатів;
– відеочат;

ICQ був важливим кроком у розвитку соціальних мереж, оскільки він зробив спілкування в Інтернеті більш доступним і зручним.

AIM (AOL Instant Messenger) — це месенджер, розроблений компанією AOL. AIM був запущений у 1997 році і швидко став популярним, особливо серед користувачів AOL.

AIM пропонував такі функції, як:

– обмін текстовими повідомленнями;
– обмін файлами;
– створення групових чатів;
– відеочат;

AIM був одним із найпопулярніших месенджерів у 1990-х і на початку 2000-х років.

MSN Messenger — це месенджер, розроблений компанією Microsoft. MSN Messenger був запущений у 1999 році і швидко став популярним, особливо серед користувачів Windows.

MSN Messenger пропонував такі функції, як:

– Обмін текстовими повідомленнями;
– Обмін файлами;
– Створення групових чатів;
– Відеочат;

MSN Messenger був одним із найпопулярніших месенджерів у 2000-х роках.

Інші месенджери. Крім ICQ, AIM і MSN Messenger, існувала низка інших популярних месенджерів у 1990-х і 2000-х роках.

До них належать:

- Yahoo! Messenger;
- QQ;
- Skype;
- Viber;

Ці месенджери пропонували широкий спектр функцій, включаючи обмін текстовими повідомленнями, обмін файлами, створення групових чатів і відеочат. Вони зробили спілкування в Інтернеті більш доступним і зручним для мільйонів людей у всьому світі.

1.1.2. Вплив чатів та месенджерів на сучасний світ

Чати та месенджери значно змінили спосіб, яким люди спілкуються та взаємодіють у цифровому віці. Вони стали невід'ємною частиною нашого повсякденного життя та великого бізнесу. Окрім того, вони значно полегшили зв'язок між людьми на великій відстані та сприяють розвитку міжнародних стосунків.

Месенджери також здійснили значний вплив на розвиток мобільних технологій та додали інноваційних функцій, таких як голосові та відеодзвінки, шифрування повідомлень та багато інших. Вони відкрили двері для розробки нових додатків та сервісів, що робить їх невід'ємною частиною сучасної цифрової екосистеми.

На цьому етапі нашої кваліфікаційної роботи ми прагнемо розкрити загальні відомості з предметної галузі та визначити основні аспекти, які будуть розглянуті у подальших розділах нашої роботи. У подальших розділах буде детально розглянуто теоретичну основу, методологічний підхід, аналіз системи та проведення власних досліджень з розробки інтерактивної чат-платформи.

1.1.3. Технології для розробки чат-платформ та месенджерів

Основні платформи, які використовуються для розробки інтерактивних чат-платформ та месенджерів - це WEB та мобільні пристрої. Веб-розробники використовують такі технології, як HTML, CSS та JavaScript, для створення інтерфейсу користувача та забезпечення функціональності веб-платформ. Додатки для мобільних пристроїв зазвичай розробляються за допомогою мов програмування, таких як Java або Swift, в залежності від платформи (Android або iOS).

Для забезпечення реального часу обміну повідомленнями і синхронізації користувачів, часто використовуються технології WebSocket або HTTP-запити до серверу. Більшість сучасних чат-платформ також використовують бази даних для зберігання повідомлень та інформації про користувачів.

Ці технології є ключовими для створення інтерактивних та надійних чат-платформ та месенджерів, які можуть задовольняти потреби сучасних користувачів у спілкуванні та знайомствах.

Технологія WebSocket. WebSocket - це протокол з'єднання між клієнтом та сервером, який дозволяє спрощений двонаправлений обмін даними в режимі реального часу через одне постійне з'єднання. Він був розроблений для покращення спілкування на веб-сайтах та додатках, де необхідно миттєво обмінюватися повідомленнями та іншою інформацією без зайвого оновлення сторінки [1].

Як WebSocket використовується в чатах:

- Встановлення з'єднання: Коли користувач входить в чат або заводить новий діалог, його браузер або додаток встановлює з'єднання (handshake) WebSocket з сервером чату (рис. 1.1).
- Двонаправлене спілкування: Після встановлення з'єднання обидві сторони можуть надсилати повідомлення одна одній без зайвих HTTP-запитів. Це дозволяє миттєво обмінюватися текстовими повідомленнями, файлами, зображеннями та іншими даними.

– Спрощений протокол: WebSocket використовує спрощений протокол, що зменшує навантаження на мережу та сервер. Це дозволяє зменшити затримку та покращити продуктивність.

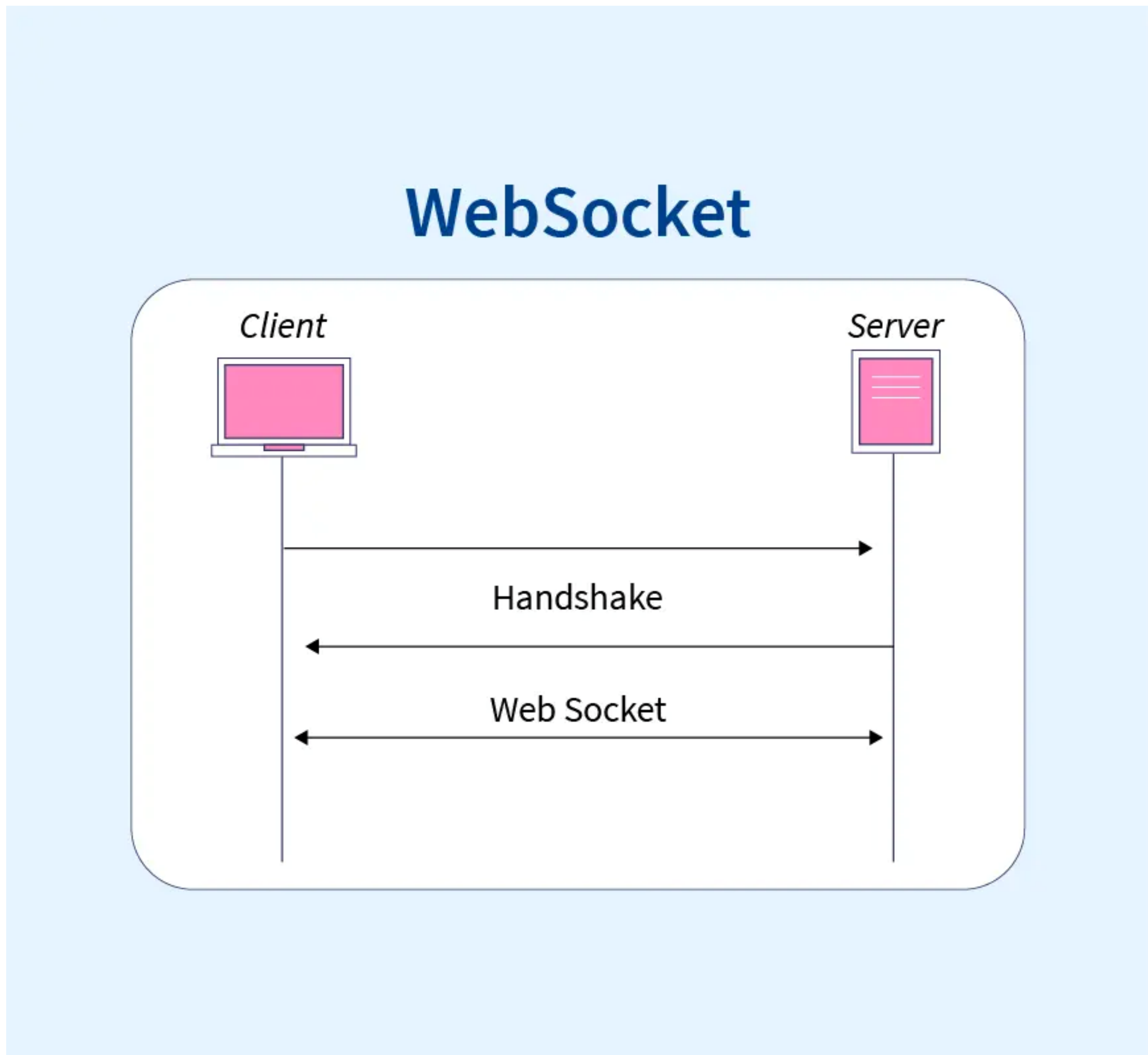

Рис. 1.1. Приклад роботи WebSocket

WebSocket став необхідною технологією для реалізації інтерактивних функцій в чат-платформах. Він дозволяє користувачам отримувати нові повідомлення миттєво, без необхідності оновлення сторінки, що робить спілкування більш зручним і приємним.

WebSocket також допомагає зменшити навантаження на сервер та мережу, що дозволяє платформі працювати ефективно, навіть при великій кількості одночасних підключень.Недоліки WebSocket:

- Постійне з'єднання: WebSocket підтримує постійні з'єднання між клієнтом і сервером, що може призводити до значного споживання ресурсів і збільшувати навантаження на сервер.
- Блокування: Якщо одне з'єднання WebSocket блокується з будь-якої причини, це може вплинути на інші з'єднання.
- Не підходить для всіх завдань: WebSocket ідеально підходить для реалізації реального часу, але не завжди є оптимальним вибором для всіх видів веб-застосунків.

Балансувальник навантаження. Балансувальник навантаження (Load Balancer) - це сервер або програмне обладнання, яке розподіляє запити від клієнтів рівномірно між доступними серверами, що виконують однаковий функціонал [2]. Основна мета Балансувальника навантаження - забезпечити надійність та високу продуктивність, а також розподіл навантаження між серверами для запобігання перевантаженню або відмовам [3].

Застосування Балансувальника навантаження разом з WebSocket.

- Розподіл трафіку: WebSocket може генерувати значну кількість з'єднань, і Балансувальник навантаження допомагає рівномірно розподілити цей трафік між різними серверами WebSocket (рис. 1.2).
- Масштабованість: Використання Балансувальника навантаження дозволяє легко додавати нові сервери WebSocket для забезпечення масштабованості системи.
- Висока доступність: В розподіленому середовищі, Балансувальник навантаження може перенаправляти трафік на доступний сервер, якщо один із серверів WebSocket відмовляє.
- Завантаження ресурсів: WebSocket може займати значну кількість ресурсів, і використання Балансувальника дозволяє розподілити це навантаження між серверами.

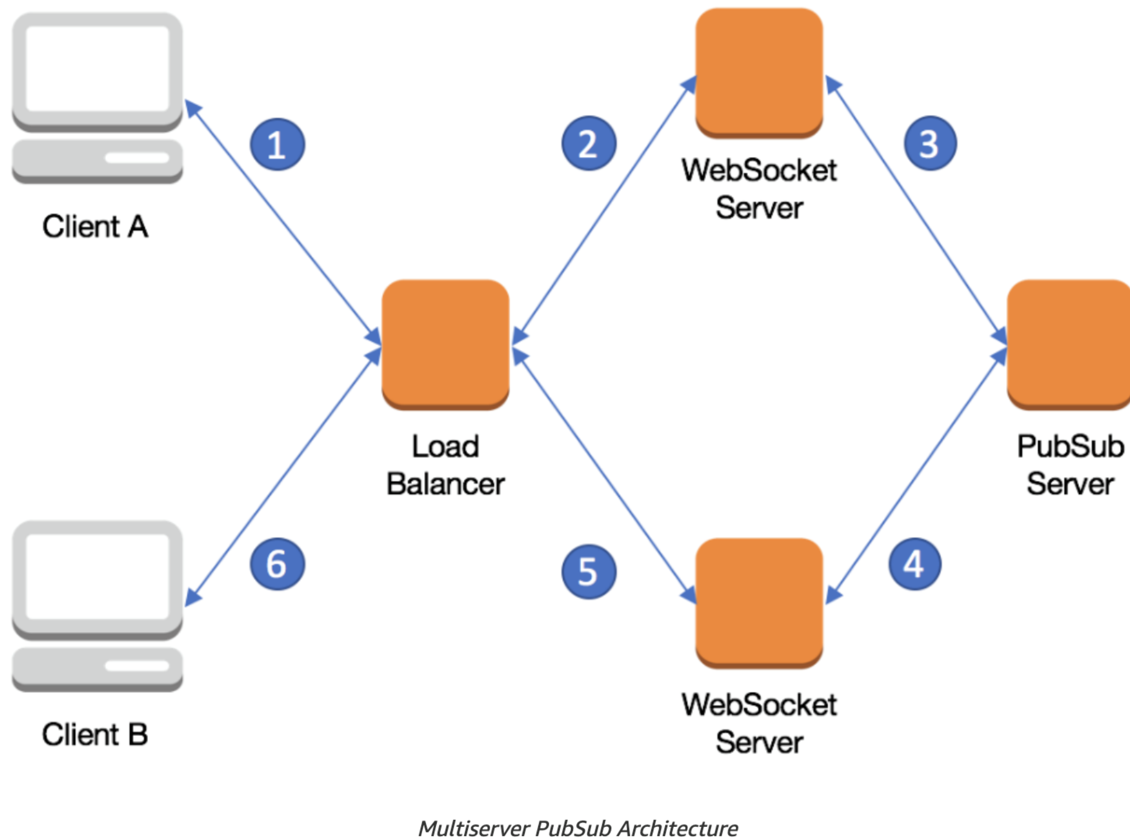

Рис. 1.2. Приклад Роботи Load Balancer з WebSocket

1.2. Аналіз існуючих інтерактивних чат-платформ та месенджерів

Сучасний світ переповнений різноманітними способами комунікації, але із ростом різноманітних платформ та месенджерів з'являються нові виклики та можливості для спілкування та знайомств між користувачами. У рамках дослідження, присвяченого розробці інтерактивної чат-платформи з елементами месенджера, особлива увага приділяється аналізу існуючих рішень.

З одного боку, швидкість розвитку цифрових технологій дозволяє нам бути завжди на зв'язку, спілкуватися з користувачами з різних кутків світу та розширювати можливості спілкування. Однак із зростанням кількості платформ та месенджерів виникає необхідність в пошуку нових шляхів для знайомства та комунікації.

Метою даного розділу є проведення аналізу існуючих чат-платформ та месенджерів та дослідження їх функціональних можливостей та особливостей

використання геолокації для організації комунікації. Даний аналіз є важливим кроком у розробці нового продукту, що дозволить забезпечити конкурентоспроможність та зручність користувачам.

Розглянемо деякі існуючі рішення та платформи для знайомств та спілкування з використанням геолокації та чат-елементів, розглядаючи їх переваги та недоліки:

WhatsApp. WhatsApp - це безкоштовний месенджер, який дозволяє користувачам обмінюватися текстовими повідомленнями, фотографіями, відео та голосовими повідомленнями. Він також пропонує функції групового чату, дзвінків та відеодзвінків. WhatsApp доступний для пристроїв Android, iOS, Windows Phone та Mac.

Месенджер був створений Брайаном Ектоном і Яном Кумом у 2009 році. У 2014 році його придбала компанія Facebook за 19 мільярдів доларів США. WhatsApp має понад 2 мільярди активних користувачів у всьому світі, що робить його одним із найпопулярніших месенджерів у світі.

WhatsApp є популярним вибором для користувачів, які хочуть надсилати текстові повідомлення та обмінюватися медіафайлами безкоштовно. Він також пропонує функції, які роблять його зручним для групового спілкування та здійснення дзвінків.

Переваги:

- Зручний та популярний месенджер.
- Безкоштовний обмін текстовими повідомленнями, фотографіями, відео та голосовими повідомленнями.
- Надсилає місцезнаходження користувача.

Недоліки:

- Орієнтований на особисте спілкування.
- Немає функціоналу автоматичного приєднання до локально створених кімнат

Facebook Messenger. Messenger - це безкоштовний месенджер, який дозволяє користувачам обмінюватися текстовими повідомленнями,

фотографіями, відео та голосовими повідомленнями. Він також пропонує функції групового чату, дзвінків та відеодзвінків, а також функції для бізнес-користувачів. Messenger доступний для пристроїв Android, iOS, Windows Phone та Mac.

Messenger був створений компанією Facebook у 2008 році як додаток до соціальної мережі Facebook. У 2011 році він був відокремлений у самостійний продукт. Messenger має понад 2,5 мільярди активних користувачів у всьому світі, що робить його одним із найпопулярніших месенджерів у світі.

Messenger є популярним вибором для користувачів, які хочуть надсилати текстові повідомлення та обмінюватися медіафайлами безкоштовно. Він також пропонує функції, які роблять його зручним для групового спілкування та здійснення дзвінків.

Messenger також надає можливість знаходити користувачів із спільними інтересами та переглядати їх місцезнаходження на карті.

Переваги:

- Інтеграція зі світовою соціальною мережею.
- Безкоштовний обмін текстовими повідомленнями, фотографіями, відео та голосовими повідомленнями.
- Можливість знаходити користувачів зі спільними інтересами та дивитися їхнє місцезнаходження.
- Функції для бізнес-користувачів, такі як бізнес-чат, чат-боти та корпоративні чати.

Недоліки:

- Переважно орієнтований на спілкування з вже знайомими людьми.
- Немає функціоналу автоматичного приєднання до локально створених кімнат.

Telegram. Telegram - це безкоштовний месенджер, який дозволяє користувачам обмінюватися текстовими повідомленнями, фотографіями, відео та голосовими повідомленнями. Він також пропонує функції групового чату, дзвінків та відеодзвінків, а також функції для бізнес-користувачів і розробників.

Telegram доступний для пристроїв Android, iOS, Windows, macOS, Linux та інших.

Месенджер був створений Ніколаєм і Павлом Дуровими у 2013 році. Telegram має понад 700 мільйонів активних користувачів у всьому світі, що робить його одним із найпопулярніших месенджерів у світі.

Telegram є популярним вибором для користувачів, які хочуть надсилати текстові повідомлення та обмінюватися медіафайлами безкоштовно. Він також пропонує функції, які роблять його зручним для групового спілкування та здійснення дзвінків.

Переваги:

- Безкоштовний обмін текстовими повідомленнями, фотографіями, відео та голосовими повідомленнями.
- Висока конфіденційність та шифрування повідомлень.
- Відкриті групи та канали для спільнот та інтересів.

Недоліки:

- Геолокація в основному використовується для публічних послуг, а не для знайомств.

MeetMe. MeetMe - це соціальна мережа, яка дозволяє користувачам спілкуватися один з одним у текстових чатах, відеочатах та під час живих трансляцій. MeetMe доступний для пристроїв Android, iOS, Windows і Mac.

Мережа була заснована в 2005 році як MyYearbook. У 2011 році вона була перейменована на MeetMe. MeetMe має понад 100 мільйонів активних користувачів у всьому світі.

MeetMe є популярним вибором для користувачів, які хочуть спілкуватися з новими людьми та знаходити нових друзів. Він також пропонує функції, які роблять його зручним для групового спілкування та організації заходів.

Переваги:

- Текстові чати. Користувачі можуть спілкуватися один з одним у текстових чатах один на один або в групах.

- Можливість знаходити і зустрічати нових людей, використовуючи геолокацію.
- Велика активна аудиторія зі світовим покриттям.

Недоліки:

- Через використання геолокації у режимі реального часу: додаток потребує обережності, оскільки може бути використаний не лише для знайомств.
- MeetMe — це соціальна мережа, а не месенджер.
- Багато підроблених акаунтів створених заради фішингу

Slack. Slack - це корпоративний месенджер, який дозволяє користувачам обмінюватися текстовими повідомленнями, файлами та посиланнями в групах. Slack доступний для пристроїв Android, iOS, Windows, macOS та Linux.

Месенджер був заснований в 2013 році Стефаном Феллоу і Філіпом Доддсом. Slack має понад 10 мільйонів активних користувачів у всьому світі.

Slack є популярним вибором для команд, які хочуть спілкуватися та співпрацювати в режимі реального часу. Він також пропонує функції, які роблять його зручним для управління проектами та організації командної роботи.

Переваги:

- Користувачі можуть обмінюватися файлами, такими як документи, презентації та зображення.
- Ефективна комунікація на робочих місцях та спільна робота.
- Slack пропонує функції для організації командної роботи, такі як канали, чат-боти та програми.
- Багато інтеграцій з іншими інструментами.

Недоліки:

- Slack це корпоративний месенджер, а не месенджер для особистого використання. Це означає, що Slack пропонує більше функцій для командного спілкування та співпраці, ніж месенджери для особистого використання.

- Відсутність будь якого функціоналу пов’язаного з використанням геолокації користувачів.

Discord. Discord - це платформа для голосового та текстового спілкування, призначена для геймерів та інших спільнот. Discord доступний для пристроїв Android, iOS, Windows, macOS, Linux та інших.

Платформа була заснована в 2015 році Джеремі Стівенсом, Мартіном Харкнесом і Ентоні Картером. Discord має понад 150 мільйонів активних користувачів у всьому світі.

Discord є популярним вибором для геймерів, оскільки він пропонує функції, які роблять його зручним для спілкування та співпраці під час гри. Він також використовується іншими спільнотами, такими як кіберспортсмени, художники, музиканти та інші.

Переваги:

- Висока якість голосового чату для геймерів.
- Можливість створення та приєднання до груп та спільнот.
- Користувачі можуть створювати та використовувати чат-ботів для автоматизації завдань та надання інформації.

Недоліки:

- Орієнтований на геймерську аудиторію, не для загальних знайомств.
- Немає функціоналу автоматичного приєднання до локально створених кімнат.

1.3. Висновки до першого розділу

У першому розділі, попередній аналіз показав, що існуючі рішення мають свої переваги та обмеження.

WhatsApp є популярним вибором для користувачів, які хочуть надсилати текстові повідомлення та обмінюватися медіафайлами безкоштовно. Він пропонує зручний інтерфейс та широкий спектр функцій, таких як групові чати, дзвінки та відеодзвінки.

Facebook Messenger є популярним вибором для користувачів, які хочуть спілкуватися з друзями та родиною, які використовують Facebook. Він пропонує зручний інтерфейс та широкий спектр функцій, таких як групові чати, дзвінки та відеодзвінки.

Telegram є популярним вибором для користувачів, які цінують приватність та безпеку. Він пропонує енкрипцію кінця в кінець для всіх повідомлень, а також широкий спектр функцій, таких як групові чати, дзвінки та відеодзвінки.

Discord є популярним вибором для геймерів та інших спільнот. Він пропонує широкий спектр функцій для голосового та текстового спілкування, а також для організації спільнот.

Багато з існуючих чат-платформ спрямовані на вже знайомих користувачів або використовують геолокацію переважно для інших цілей, таких як комерційна або робоча комунікація. Однак на мою думку існує потреба в платформі, яка дозволить користувачам знайомитися, спілкуватися та об'єднуватися в чат кімнати на основі їхнього місцезнаходження, при цьому не боячись що їх геолокація буде використана у шкідливих цілях

Розробляєма інтерактивна чат-платформа буде відкритою для аудиторії всіх видів та вікових груп. Вона надасть можливість знаходити нових друзів незалежно від місцезнаходження, спілкуватися та організовувати події на основі геолокації. Такий підхід дозволить покращити досвід користувачів і сприяти новим знайомствам, незалежно від того, чи це особисті, робочі, чи спільні заходи.

В наступному розділі буде проведено аналіз існуючих технологій та методів для вирішення поставленої задачі.

РОЗДІЛ 2
МЕТОДИ ТА ТЕХНОЛОГІЇ ВИРІШЕННЯ ЗАДАЧІ

2.1. Аналіз та порівняння технологій React, Vue.js та Angular

2.1.1. Бібліотека React

React - це бібліотека для розробки інтерфейсів користувача, створена компанією Facebook. Вона дозволяє розробникам будувати веб-додатки з великою кількістю інтерактивних елементів, які забезпечують ефективне та динамічне відображення даних [4].

React був розроблений і випущений компанією Facebook із початковим випуском у травні 2013 року. Його створення було викликане необхідністю вирішення проблем з розробкою великих інтерфейсів користувача для односторінкових додатків на стороні клієнта.

Однією з ключових постанов, яка лежить в основі React, є використання virtual DOM (рис. 2.1). Він був розроблений для того, щоб полегшити оновлення елементів інтерфейсу та забезпечити оптимальну продуктивність [5].

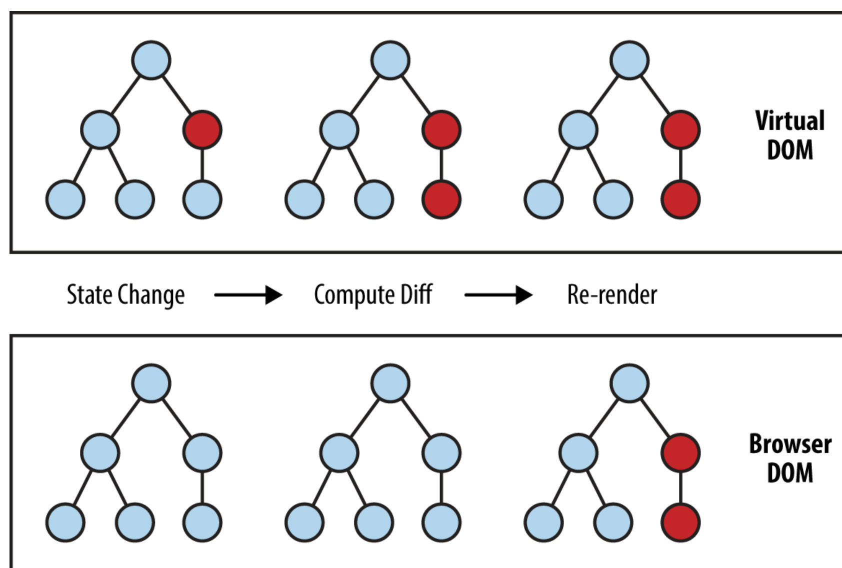

Рис. 2.1. Принцип роботи Virtual DOM

У 2015 році Facebook випустив бібліотеку React Native, яка дозволяє використовувати React для розробки мобільних додатків. Це рішення дозволило розробникам використовувати ті ж принципи та компоненти для побудови як веб-додатків, так і мобільних додатків. Протягом наступних років React став дуже популярним серед розробників, і велика кількість веб-додатків та компаній вибирають його для своїх проектів. У 2018 році була анонсована нова версія React, React 16, яка принесла численні вдосконалення та нововведення, такі як підтримка "hooks" для функціональних компонентів.

React також відомий своєю активною спільнотою розробників та великою кількістю сторонніх бібліотек та інструментів, що допомагають полегшити розробку.

Основні принципи React:

- Компоненти: React базується на концепції компонентів. Компонент - це невелика ізольована частина коду, яка має свою логіку та може бути повторно використана. Компоненти можуть бути класовими або функціональними.
- Віртуальний DOM (Virtual DOM): React використовує віртуальний DOM (рис. 2.1) для оптимізації швидкості оновлення веб-сторінки. Замість безпосередньої маніпуляції реальним DOM, React спочатку оновлює віртуальний DOM і потім визначає мінімальні зміни, які потрібно внести в реальний DOM.
- Односторінкова архітектура (SPA): React дозволяє легко створювати односторінкові додатки, де зміни відбуваються без перезавантаження сторінки. Це допомагає покращити швидкість та користувацький досвід.
- JSX (JavaScript XML): React використовує JSX, який є розширенням синтаксису JavaScript. JSX дозволяє описувати структуру UI за допомогою тегів, схожих на HTML, що полегшує читання та розуміння коду.

- Односторінкова архітектура (SPA): React дозволяє легко створювати односторінкові додатки, де зміни відбуваються без перезавантаження сторінки. Це допомагає покращити швидкість та користувацький досвід.

До найбільш переваг React відносять:

- Розробка веб-додатків: React широко використовується для створення веб-додатків з великою кількістю інтерактивних елементів, таких як соціальні мережі, панелі керування, інтернет-магазини тощо.
- Мобільна розробка: З допомогою React Native, який є фреймворком для мобільної розробки на основі React, розробники можуть створювати мобільні додатки для платформ Android та iOS, використовуючи знання React.
- Односторінкові додатки (SPA): React часто використовується для створення односторінкових додатків, де весь контент завантажується один раз, і подальші зміни відбуваються без перезавантаження сторінки.
- Розширення функціональності існуючих проектів: React може бути легко інтегрована в існуючі проекти, дозволяючи розробникам поетапно впроваджувати новий функціонал без повного переписування коду.
- Зворотна сумісність (Backward Compatibility): React зазвичай дотримується принципу зворотної сумісності, що означає, що нові версії бібліотеки можуть коректно обробляти код, написаний для попередніх версій. Це сприяє гладкому оновленню для розробників.
- Стабільність API: Офіційне API React підтримується та залишається стабільним протягом тривалого часу. Це означає, що основні концепції та інтерфейси, які використовують розробники, залишаються стабільними протягом тривалого періоду.

2.1.2. Фрейморк Vue.js

Vue.js — JavaScript-фреймворк для створення інтерфейсів користувача (UI). Він був розроблений Еван Ю в 2014 році як невелика бібліотека, але швидко став популярним і перетворився на повноцінний фреймворк.

Vue.js використовує модель MVVM (Model-View-ViewModel) для створення UI (рис.2.2). В цій моделі дані зберігаються в моделях, а відображення даних здійснюється за допомогою візуальних елементів, які називаються представленнями. Vue.js забезпечує зв'язок між моделями та представленими за допомогою реактивного зв'язування даних [6].

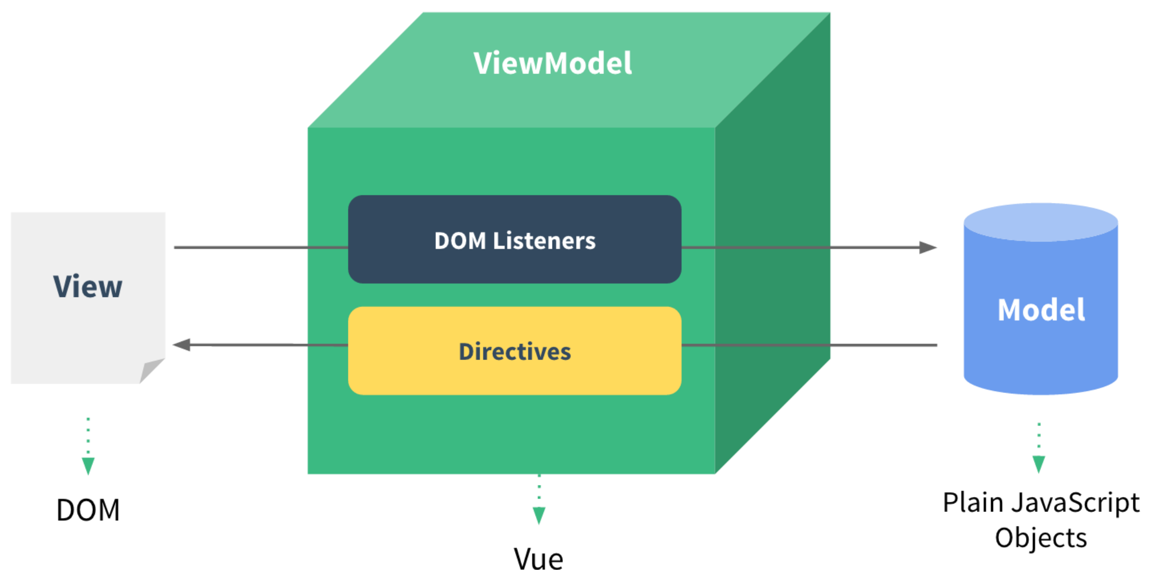

Рис. 2.2. Vue.js використовує модель MVVM (Model-View-ViewModel)

До основних елементів Vue.js належать [5]:

- Комоненти.
- Шаблони.
- Реактивне зв'язування даних(Data binding).
- Директиви.

Поговоримо детальніше про кожен з цих елементів.

Комоненти. Компоненти — це основні будівельні блоки Vue.js. Вони дозволяють створювати повторювані і організовані елементи UI. У Vue.js кожен компонент є просто Vue-екземпляром. Компоненти утворюють вкладену деревоподібну ієрархію (рис. 2.3), яка представляє інтерфейс вашого додатку. Вони можуть бути створені за допомогою користувацького конструктора, отриманого з Vue.extend, але більш декларативним підходом є реєстрація їх за допомогою Vue.component(id, constructor).

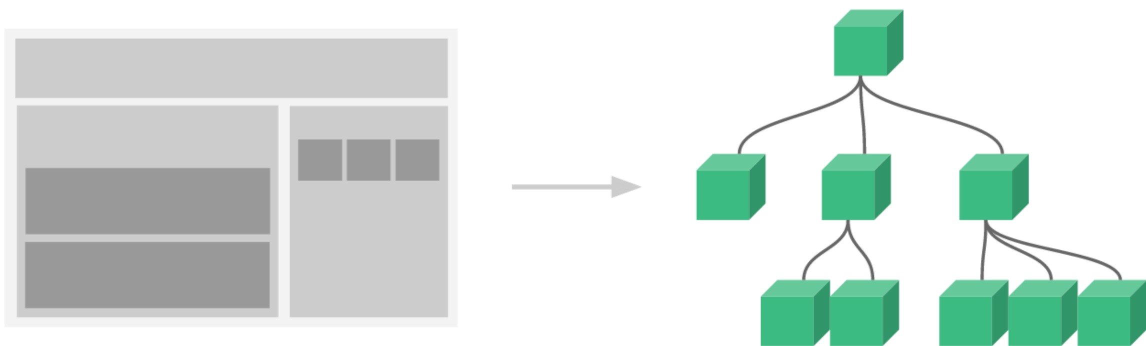

Рис. 2.3. Дерево компонентів Vue.js

Шаблони. Шаблони Vue.js — це HTML-код, який використовується для визначення зовнішнього вигляду компонента. Вони можуть містити такі елементи, як:

- HTML-теги. HTML-теги використовуються для створення візуальних елементів компонента.
- Дата-зв'язки. Дата-зв'язки використовуються для зв'язування даних з візуальними елементами.
- Скрипти. Скрипти використовуються для додавання поведінки до компонента.

Реактивне зв'язування даних. Реактивне зв'язування даних (Data binding) — це механізм, який дозволяє зв'язувати дані з візуальними елементами (рис.2.4). У Vue.js data binding реалізується за допомогою атрибутів v-bind.

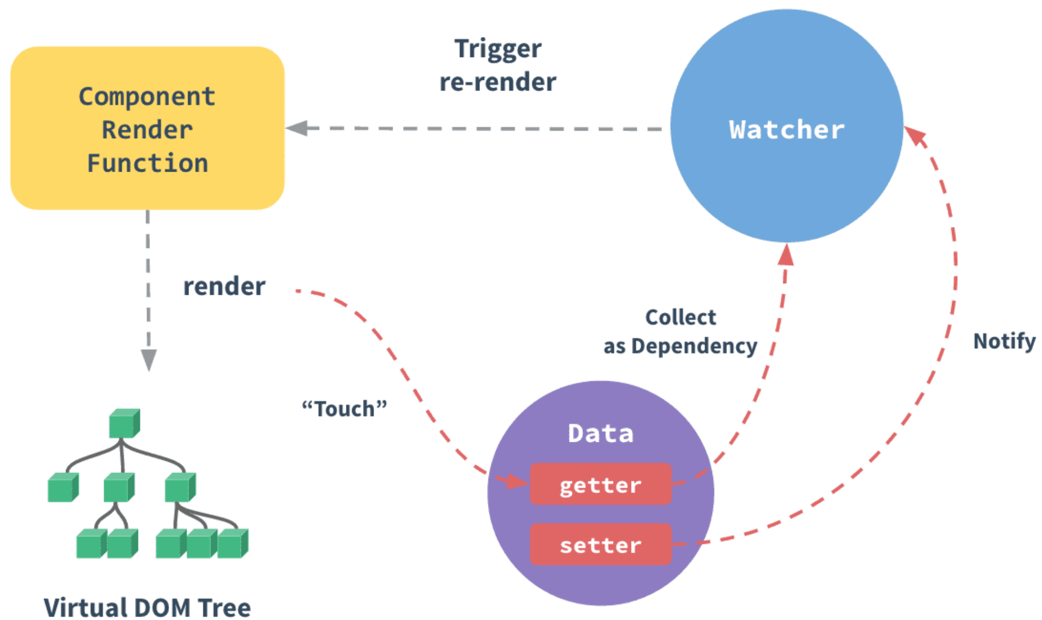

Рис. 2.4. Реактивне зв'язування даних у Vue.js

Директиви. Директиви — це префіксовані HTML-атрибути, які інструктують Vue.js виконувати певні дії з DOM-елементом.

Директиви є спеціальними HTML-атрибутами, що починаються з префіксу v-. Вони інструктують Vue.js виконувати певні дії на пов'язаному DOM-елементі. Наприклад, директива v-text, коли її застосовують до елемента <div>, синхронізує властивість textContent елемента з значенням JavaScript-змінної, названої message. Це означає, що будь-які зміни, внесені до змінної message, автоматично відображатимуться в змісті елемента <div>.

Директиви можна використовувати для виконання різноманітних операцій над DOM, включаючи встановлення атрибутів, копіювання елементів та прикріплення обробників подій.

2.1.3. Фреймворк Angular

Angular — це повнофункціональний фреймворк для створення веб-додатків, написаний на TypeScript [7]. Він був розроблений компанією Google в

2010 році як заміна для свого попереднього фреймворка AngularJS. Перша версія Angular була випущена в 2016 році.

Angular має ряд унікальних особливостей, які відрізняють його від інших фреймворків для розробки веб-додатків:

- Реактивне зв'язування даних — Angular використовує реактивний дата-зв'язування, яке забезпечує синхронізацію даних і візуальних елементів у режимі реального часу.
- Компоненти — Angular використовує компоненти для створення візуальних елементів і поведінки додатку. Компоненти є повторюваними і організованими, що робить код більш зрозумілим і універсальним.
- Директиви — Angular використовує директиви для виконання різних дій з DOM-елементами. Директиви можуть використовуватися для додавання поведінки до візуальних елементів, а також для стилізації та форматування.
- Сервіси — Angular використовує сервіси для доступу до зовнішніх даних і функціональності. Сервіси можна використовувати для абстрагування від деталей реалізації, а також для забезпечення повторюваності коду.
- Рутинні процеси — Angular використовує рутинні процеси для виконання повторюваних завдань. Рутинні процеси можна використовувати для зменшення кількості коду, який потрібно писати.

Angular має ряд переваг, які роблять його популярним вибором для розробки веб-додатків:

- Реактивний дата-зв'язування забезпечує синхронізацію даних і візуальних елементів у режимі реального часу. Це робить додатки більш динамічними і інтерактивними.
- Компоненти є повторюваними і організованими, що робить код більш зрозумілим і універсальним.
- Директиви дають розробникам широкий спектр можливостей для стилізації, форматування та розширення функціональності візуальних елементів.

- Сервіси забезпечують повторюваність коду і абстрагують від деталей реалізації.
- Рутинні процеси зменшують кількість коду, який потрібно писати.

Angular також має ряд недоліків, які слід враховувати при виборі фреймворка для розробки веб-додатків:

- Angular може бути складним для вивчення і використання. Він має широкий спектр функцій, які можуть бути важкими для розуміння.
- Angular вимагає певного досвіду в JavaScript і TypeScript.
- Angular може бути повільним для великих і складних додатків.

Angular — це потужний і універсальний фреймворк для розробки веб-додатків. Він має широкий спектр функцій, які роблять його хорошим вибором для створення складних і динамічних додатків.

Angular підходить для розробників, які хочуть створювати складні і динамічні веб-додатки. Він також підходить для розробників, які хочуть використовувати реактивний дата-зв'язування і компоненти.

2.1.4. Порівняння React, Angular та Vue.js

React, Vue і Angular — це три найпопулярніші фреймворки для розробки веб-додатків. У кожного з них є свої сильні і слабкі сторони, які слід враховувати при виборі фреймворка для конкретного проекту.

React — це бібліотека для створення інтерфейсів користувача (UI). Він використовує реактивний дата-зв'язування для синхронізації даних і візуальних елементів у режимі реального часу. React є відносно легким фреймворком, який легко вивчити і використовувати.

Vue — це фреймворк для створення UI, який використовує реактивний дата-зв'язування. Він подібний до React, але має більш просту структуру і меншу кількість функцій. Vue є хорошим вибором для розробників, які хочуть створити простий і динамічний UI.

Angular — це повнофункціональний фреймворк для створення веб-додатків. Він використовує реактивний дата-зв'язування, компоненти, директиви, сервіси та рутинні процеси. Angular є потужним фреймворком, який підходить для створення складних і динамічних додатків.

Ось порівняння React, Vue і Angular за основними критеріями:

Критерій	React	Vue	Angular
Вага	Легкий	Легкий	Середній
Легкість вивчення	Легкий	Легкий	Середній
Реактивний дата-зв'язування	Так	Так	Так
Компоненти	Так	Так	Так
Директиви	Ні	Так	Так
Сервіси	Ні	Так	Так
Рутинні процеси	Ні	Ні	Так
Застосування	UI	UI	UI, веб-додатки
Кому підходить	Розробники, які хочуть створити простий і динамічний UI	Розробники, які хочуть створити простий і динамічний UI	Розробники, які хочуть створити складний і динамічний UI

Рис. 2.5. Порівняння React, Vue та Angular

На основі трендів Stack Overflow (рис. 2.6), опитів Stack Overflow (рис. 2.7) ми можемо констатувати, що на сьогоднішній день React являється найпопулярнішою бібліотекою на розробки веб додатків у світі [8].

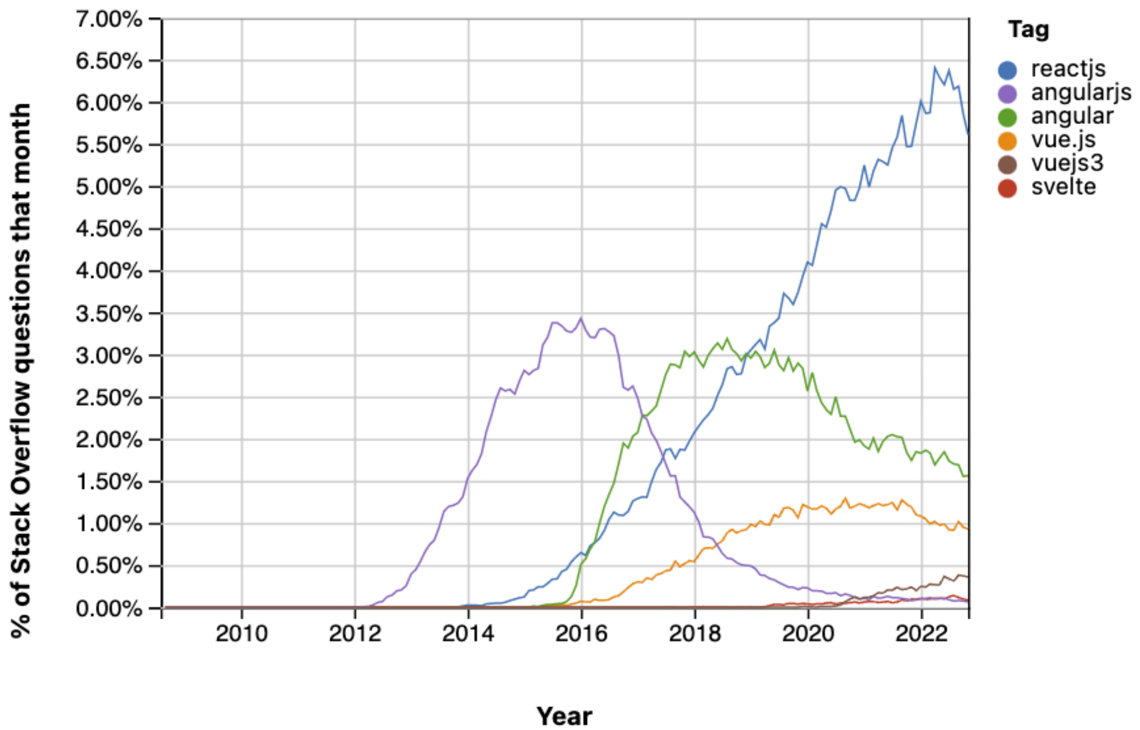

Рис. 2.6. Популярність реакту на базі трендів Stack Overflow

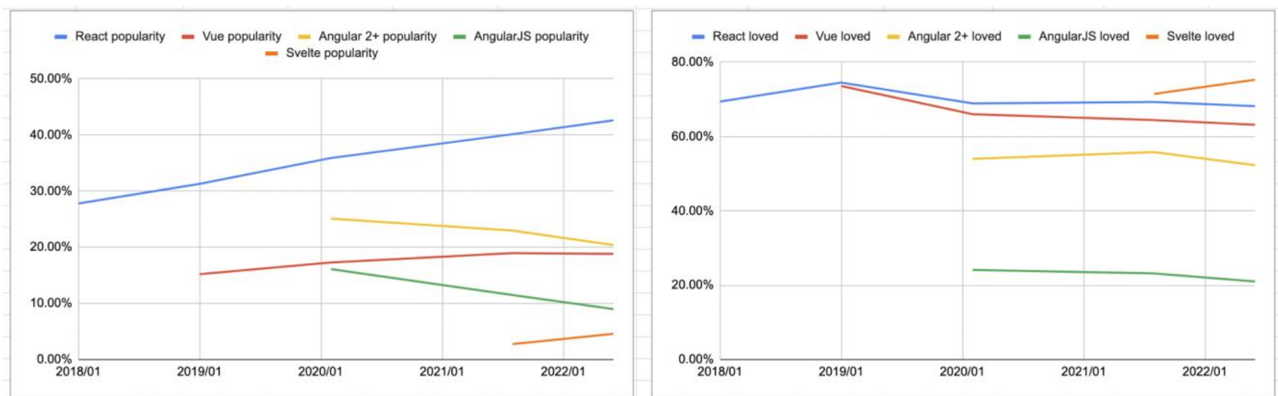

Рис. 2.7. Популярність реакту на базі опитів активних користувачів Stack Overflow

На фоні конкурентів, таких як Angular та Vue.js, React вирізняється своєю гнучкістю та простотою. Angular, хоча також потужний, має більш жорсткий і складний синтаксис, що може вимагати більше часу для вивчення. Vue.js, з іншого боку, вважається більш легким і простим для використання, але йому може бракувати екосистеми та розширеного функціоналу, які надає React.

На основі попереднього аналізу можна заключити, що React — це найкращий вибір для більшості проектів. Він легкий, простий у вивченні і

використанні, і використовує реактивний дата-зв'язування для синхронізації даних і візуальних елементів у режимі реального часу.

Ось кілька аргументів на користь вибору React:

- React є легким фреймворком, який легко вивчити і використовувати. Це робить його хорошим вибором для розробників-початківців, а також для команд, які хочуть швидко і ефективно створювати веб-додатки.
- React використовує реактивний дата-зв'язування. Це забезпечує синхронізацію даних і візуальних елементів у режимі реального часу, що робить додатки більш динамічними і інтерактивними.
- React має велику і активну спільноту. Це означає, що розробники можуть знайти допомогу та підтримку в разі потреби.

Однією з головних переваг React є його велика та активна спільнота розробників, що призводить до постійного розвитку та підтримки. Розширена екосистема бібліотек та інструментів, таких як Redux для управління станом додатку, також робить React привабливим вибором для розробників.

Звичайно, у кожного фреймворка є свої сильні і слабкі сторони. React підходить не для всіх проектів. Наприклад, Angular може бути кращим вибором для розробки складних і динамічних додатків. Однак для більшості проектів React є хорошим вибором.

2.2. Мова програмування Node.js та фреймворк Express.js

Node.js є виконавчим середовищем для JavaScript, яке дозволяє виконувати код JavaScript на сервері. Він був створений Райаном Далем та вперше випущений у 2009 році. Основною ідеєю було розширення можливостей JavaScript за межі веб-браузера, дозволяючи використовувати його для створення серверної частини додатків [9].

Основні переваги Node.js:

- Асинхронність та Події: Node.js використовує асинхронний, подійний механізм обробки, що дозволяє ефективно взаємодіяти з багатьма запитами одночасно, що особливо важливо для серверів.
- Швидкодія: Завдяки використанню двигуна V8 від Google, Node.js володіє вражаючою швидкодією виконання JavaScript-коду.
- Спільна мова для фронтенду та бекенду: Використання однієї мови (JavaScript) як на фронтенді, так і на бекенді полегшує обмін коду та ресурсів між розробниками.
- Розширені можливості за допомогою модулів: Node.js має велику кількість модулів, які дозволяють розробникам легко розширювати функціональність своїх додатків.
- Активна спільнота: Існує велика та активна спільнота розробників Node.js, яка постійно розвиває та підтримує цей проект.

Використання Node.js на бекенді, коли фронтенд розроблений на JavaScript, принесе кілька переваг:

- Спільна Мова: Якщо фронтенд і бекенд написані на одній мові (JavaScript), це спрощує обмін кодом, навчання розробників та спільну роботу між обома частинами системи.
- Спільний Код та Модулі: Функції та модулі, написані на JavaScript для фронтенду, можна використовувати також на бекенді, що полегшує розробку та підтримку.
- Ефективне використання Навичок: Розробники, які вже володіють навичками роботи з JavaScript, можуть застосовувати свої знання при створенні інтерактивних та швидких веб-додатків.

Express.js: Легкий та Гнучкий Фреймворк для Node.js. Express.js є одним з найпопулярніших фреймворків для створення серверних додатків на базі Node.js. Деякі переваги використання Express.js включають[9]:

- Легкість використання: Express.js розроблений так, щоб бути простим та зрозумілим, одночасно надаючи достатній рівень абстракції для ефективної розробки.

- Природна підтримка Middleware: Express.js активно використовує концепцію middleware, що дозволяє легко додавати та налаштовувати функціональність серверу.
- Маршрутизація: Express.js надає потужні засоби для визначення та обробки маршрутів, що робить його ідеальним для реалізації API та обробки запитів.
- Розширені Можливості: Якщо базового функціоналу Express.js вам недостатньо, ви можете легко використовувати додаткові модулі та розширювати фреймворк.
- Велика Спільнота та Документація: Існує обширна спільнота користувачів Express.js, а також детальна та доступна документація, що полегшує вивчення та використання фреймворку.

Використання комбінації React та Express для розробки чат-додатку є високо оптимальним рішенням, що поєднує потужність фронтенду та бекенду в єдиному стеку технологій. React, як бібліотека для інтерфейсу користувача, надає гнучкість, компонентний підхід та швидкодію, що є ключовими для розробки інтерактивного та динамічного інтерфейсу чату.

З іншого боку, Express, як легкий та гнучкий фреймворк для Node.js, стає потужним інструментом для створення серверної частини додатку. Він надає зручні засоби для реалізації обробки запитів, маршрутизації та роботи з middleware, що дозволяє з легкістю створювати та підтримувати бекенд чат-системи.

Однією з ключових переваг такого підходу є спільна мова програмування (JavaScript/TypeScript) для обох сторін - клієнтської та серверної, що полегшує розробку, підтримку та спілкування в команді. Використання React і Express дозволяє розробникам створювати сучасні та ефективні чат-додатки, забезпечуючи швидкий відгук та високу продуктивність.

Загальний висновок полягає в тому, що використання React та Express у спільному проекті створює потужну та зручну інфраструктуру для розробки чат-

додатку, що відповідає сучасним стандартам та вимогам до користувацького досвіду.

2.3. WebSocket та бібліотека Socket.io

WebSocket - це протокол зв'язку, який надає двосторонній, повно-дуплексний канал зв'язку між клієнтом та сервером через веб-браузер або інші платформи. Ця технологія особливо корисна для розробки чат-платформ та інших веб-додатків, де реально-часовий обмін даними є важливим [1].

Основні риси та принципи роботи WebSocket:

- Дуплексний зв'язок: WebSocket дозволяє обмінюватися даними між клієнтом та сервером в обидві сторони одночасно. Це відрізняє його від традиційного HTTP, яке є напів-дуплексним, тобто дані можуть передаватися тільки в одному напрямку в певний момент.
- Спрощений протокол рукостискання: Процес встановлення з'єднання у WebSocket спрощений порівняно з HTTP. Ініціатор з'єднання робить запит, і якщо сервер приймає, то вони обирають дуплексне з'єднання.
- Real-time оновлення: WebSocket ідеально підходить для сценаріїв реального часу, таких як чат-платформи, онлайн-ігри, трансляції тощо. Дані можуть бути миттєво передані всім підключеним клієнтам без необхідності постійних запитів до сервера.
- Менше затримок: У порівнянні з підходом "попит-відповідь" у традиційних HTTP-запитах, WebSocket дозволяє зменшити затримки, оскільки немає потреби в створенні нового з'єднання для кожного запиту.

Як це працює:

- Рукостискання (Handshake): Клієнт відправляє запит на сервер для встановлення WebSocket-з'єднання, і якщо сервер приймає, вони встановлюють з'єднання. Цей процес відомий як рукостискання.
- Повно-дуплексне з'єднання: Після встановлення з'єднання обидві сторони можуть відправляти дані одне одному в реальному часі.

- Фрейми та Повідомлення: Дані передаються у вигляді "фреймів". Фрейми можуть бути текстовими, бінарними чи іншими форматами. Інформація відправляється у вигляді повідомлень.
- Закриття з'єднання: З'єднання може бути закрите як клієнтом, так і сервером. Також може бути можливість автоматичного перепідключення.

Socket.IO як реалізація WebSocket. Socket.IO - це бібліотека для реалізації веб-сокетів (WebSocket) та інших транспортних механізмів в реальному часі для веб-додатків. Вона дозволяє побудувати багатокористувацькі, реально-часові додатки з використанням протоколів, таких як WebSocket, Polling, та інших [10].

Основні характеристики та функціонал Socket.IO:

- WebSocket та Інші Транспорти: Socket.IO намагається використовувати WebSocket як найоптимальніший транспорт, але в разі, якщо він недоступний, автоматично переходить на інші транспортні механізми, такі як AJAX Long Polling, Server-Sent Events тощо.
- Швидкість та Надійність: Socket.IO вбудовує механізми перепідключення та автоматичного виправлення помилок, що забезпечує стійкість та надійність з'єднання в умовах різних мережевих умов.
- Канали та Події: Socket.IO підтримує концепцію каналів, що дозволяє групувати клієнтів та обмінюватися повідомленнями між ними. Крім того, інтерфейс подій дозволяє визначати та обробляти різні типи подій на клієнті та сервері.
- Проста Інтеграція з Express: Socket.IO може легко інтегруватися з фреймворком Express для створення повноцінних веб-додатків.
- Підтримка Різних Клієнтських та Серверних Платформ: Socket.IO має клієнтські бібліотеки для JavaScript, а також серверні бібліотеки для Node.js, Python, Java, та інших мов програмування.
- Крос-доменні Запити: Socket.IO підтримує крос-доменні запити (Cross-Origin Resource Sharing - CORS), що дозволяє взаємодіяти з різними доменами та піддоменами.

2.4. PostgreSQL

PostgreSQL, також відомий як "Postgres," є відкритою та об'єктно-реляційною системою управління базами даних (СУБД). Він був розроблений з прицілом на стандарти та повну сумісність з мовою SQL. PostgreSQL визначається своєю надійністю, гнучкістю, та високою продуктивністю [11].

Основні Характеристики PostgreSQL:

- Реляційна Система Управління Базами Даних (СУБД): PostgreSQL використовує модель реляційної бази даних, що дозволяє організувати дані у вигляді таблиць з відносинами між ними.
- Повна Сумісність із SQL:PostgreSQL дотримується стандартів мови SQL, забезпечуючи користувачам знайомий та однорідний інтерфейс для роботи з даними.
- Множинні Типи Даних: в PostgreSQL можна використовувати різноманітні типи даних, включаючи числові, рядкові, дати, географічні, та інші. Це дозволяє зберігати та обробляти різноманітні дані.
- Розширені Функціональність та Засоби Запитів: PostgreSQL надає багато функціональних можливостей, таких як транзакції, підтримка індексів, використання засобів повідомлень, включення тригерів, та інші.
- Підтримка JSON та JSONB: За допомогою розширення JSON та JSONB, PostgreSQL ставиться в один ряд із сучасними тенденціями в зберіганні та опрацюванні документоорієнтованих даних.
- Паралельні Запити та Масштабованість:В останніх версіях PostgreSQL додано підтримку паралельних запитів, що поліпшує продуктивність високонавантажених систем. Також існують засоби для горизонтального та вертикального масштабування.
- Гнучкість та Розширюваність:PostgreSQL дозволяє розширювати його функціональність за допомогою розширень та власних функцій.
- Безпека Даних: Система надає різні засоби безпеки, включаючи ролі та права доступу, що дозволяє детально налагодити доступ до бази даних.

– Активна Спільнота та Підтримка: З великою та активною спільнотою користувачів та розробників, PostgreSQL забезпечує швидке реагування на проблеми та підтримку нових технологій.

2.5. Висновки до другого розділу

У другому розділі в ході проведеного аналізу було обрано стек технологій для розробки чат-додатку, який включає React для фронтенду, Express.js, Socket.IO для бекенду та PostgreSQL для бази даних, представляє собою комплексне та оптимальне рішення. Використання цих технологій дозволяє ефективно вирішувати ключові завдання розробки чат-платформи.

Socket.IO в поєднанні з Express.js надає механізм реального часу, що є критично важливим для чат-додатків. Здатність миттєво відправляти та отримувати повідомлення забезпечує інтерактивність та покращує користувацький досвід. Також це поєднання технологій дозволяє легко масштабувати бекенд чат-додатку відповідно до зростаючих потреб користувачів. Гнучкість цих технологій робить їх підходящими для різних розмірів та типів чат-платформ.

React, який використовується для фронтенду, дозволяє створювати динамічний та ефективний інтерфейс чат-додатку. Компонентна архітектура React сприяє легкості розробки, підтримці та розширенню функціональності.

Використання PostgreSQL як системи управління базами даних забезпечує надійність, стабільність та гнучкість для зберігання та управління даними чат-додатку. Реляційна модель бази даних інтегрується з ефективним засобом зберігання JSON, що робить її ідеальною для обробки чат-повідомлень та користувацьких даних.

Також слід зазначити, що всі використовувані технології (React, Express.js, Socket.IO, PostgreSQL) мають великі та активні спільноти, що гарантує швидку підтримку, вирішення проблем та доступність нових функцій.

Обраний стек технологій вирішує ключові виклики розробки чат-додатку, забезпечуючи швидку та ефективну обмін даними в реальному часі, надійну роботу з базою даних та забезпечуючи гнучкість та масштабованість для майбутнього розвитку платформи.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ЙОГО ЗАСТОСУВАННЯ

3.1. Опис структури системи та алгоритмів її функціонування

Для реалізації поставленої задачі першим кроком було поділено розробку чат платформи на п’ять кроків:

1. Налаштування структури бази даних, та ORM для з’єднання бази та серверу.
2. Налаштування системи авторизації користувачів, та налаштування ролей доступу до чат кімнат.
3. Налаштування Socket.io, а саме сокетних кімнат, відправки повідомлень в ці кімнати, та зв’язок з клієнтською частиною додатку.
4. Створення користувацького інтерфейсу чат платформи, та з’єднання його з серверною частиною додатку.
5. Створення middleware для обробки геолокації користувача з подальшим додаванням його до місцевого чату.

Далі ми розглянемо кожен з цих пунктів більш детальніше.

Налаштування структури бази даних. Першим кроком при розробці будь якого продукту є проектування бази даних, і ми в нашій роботі не стали відходити від цього принципу, і почали саме с бази даних. Нами була обрана реляційна база даних PostgreSQL, через її надійність, гнучкість, та високу продуктивність.

Схема бази даних виглядає наступним чином (рис. 3.1).

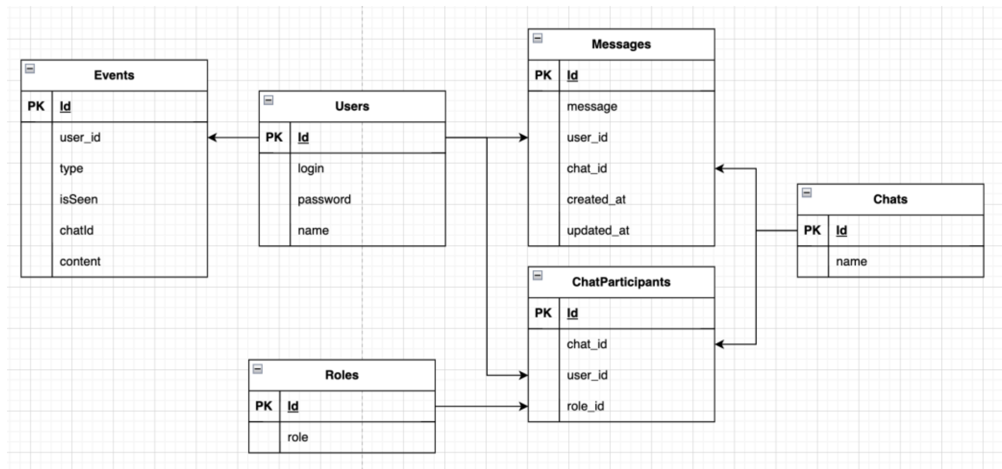

Рис. 3.1. Схема бази даних

Дивлячись на схему відображену на рис. 3.1 можна виділити наступні сутності:

- Users - ця таблиця містить інформацію про користувачів системи.
- Chats - ця таблиця містить інформацію про чати.
- Messages - ця таблиця містить інформацію про повідомлення в чатах.
- Events - ця таблиця містить інформацію про події в чатах.
- ChatParticipants - ця таблиця містить інформацію про учасників чатів.
- Roles - ця таблиця містить інформацію про роль учасника в чаті.

Таблиця Users має зв'язок один-до-багатьох з таблицею ChatParticipants. Це означає, що кожен користувач може брати участь в одному або декількох чатах. Зв'язок між таблицями Users і ChatParticipants задається за допомогою поля UserId в таблиці ChatParticipants. Це поле є зовнішнім ключем, який посилається на первинний ключ Id в таблиці Users. Це означає, що кожний запис в таблиці ChatParticipants повинен мати унікальне значення в полі UserId, яке також є унікальним значенням в таблиці Users.

Таблиця Chats має зв'язок один-до-багатьох з таблицею Messages. Це означає, що в кожному чаті може бути відправлено один або кілька повідомлень. Зв'язок між таблицями Chats і Messages задається за допомогою поля ChatId в таблиці Messages. Це поле є зовнішнім ключем, який посилається на первинний ключ Id в таблиці Chats. Це означає, що кожний запис в таблиці Messages

повинен мати унікальне значення в полі ChatId, яке також є унікальним значенням в таблиці Chats.

Таблиця Chats має зв'язок один-до-багатьох з таблицею Events. Це означає, що в кожному чаті може відбутися один або кілька подій. Зв'язок між таблицями Chats і Events задається за допомогою поля ChatId в таблиці Events. Це поле є зовнішнім ключем, який посилається на первинний ключ Id в таблиці Chats. Це означає, що кожен запис в таблиці Events повинен мати унікальне значення в полі ChatId, яке також є унікальним значенням в таблиці Chats.

Ці зв'язки дозволяють нам об'єднувати інформацію з різних таблиць. Наприклад, ми можемо використовувати зв'язок між таблицями Users і ChatParticipants, щоб знайти всі чати, в яких бере участь користувач. Ми можемо використовувати зв'язок між таблицями Chats і Messages, щоб знайти всі повідомлення, відправлені в чат. Ми можемо використовувати зв'язок між таблицями Chats і Events, щоб знайти всі події, що відбулися в чаті.

Для того щоб приєднати базу даних до серверної частини додатку я буду використовувати ORM sequelize та провайдер pg. Програмна реалізація цього виглядає наступним чином:

Створюємо окремий файл з даними необхідними для приєднання до бази даних:

```
const { Sequelize } = require('sequelize');
module.exports = new Sequelize(
  process.env.DB_NAME,
  process.env.DB_USER,
  process.env.DB_PASSWORD,
  {
    dialect: 'postgres',
    host: process.env.DB_HOST,
    port: process.env.DB_PORT,
    logging: false,
  }
);
```

Далі імпортуємо екземпляр Sequelize до файлу сервера, та ініціюємо приєднання до бази даних:

```
const sequelize = require('./db');
```

```
    …
const start = async () => {
  try {
// Аутентифікація в базі
    await sequelize.authenticate();
// Оновлення таблиць в базі даних, якщо відбулись будь-які видозміни моделей таблиць
    await sequelize.sync({ alter: true });
    httpServer.listen(PORT, () =>
      console.log(`Server started on port ${PORT}`)
    );
    // countries.map(async (country) => {
    //   const chat = await models.Chat.create({ name: country });
    // });
  } catch (e) {
    console.log(e);
  }
};
```

Далі створюємо уявлення таблиць баз даних у вигляді моделей Sequelize.

Таблиця користувачів:

```
const User = sequelize.define('user', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  login: { type: DataTypes.STRING, unique: true, allowNull: false },
  password: { type: DataTypes.STRING, allowNull: false },
  name: { type: DataTypes.STRING, allowNull: false },
});
```

Таблиця ролей:

```
const Role = sequelize.define('role', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  role: {
    type: DataTypes.STRING,
    allowNull: false,
    unique: true,
  },
});
```

Таблиця чатів:

```
const Chat = sequelize.define('chat', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
```

```
    name: { type: DataTypes.STRING, allowNull: false, unique:
true },
    image: { type: DataTypes.STRING, allowNull: true },
  });
```

Таблиця повідомлень:

```
  const Message = sequelize.define('message', {
    id: { type: DataTypes.INTEGER, primaryKey: true,
autoIncrement: true },
    message: { type: DataTypes.STRING, allowNull: false },
  });
```

Таблиця учасників чата:

```
  const ChatParticipants = sequelize.define(
    'chatParticipants',
    {
      id: { type: DataTypes.INTEGER, primaryKey: true,
autoIncrement: true },
      unreadMessages: {
        type: DataTypes.INTEGER,
        allowNull: false,
        defaultValue: 0,
      },
    },
    {
      tableName: 'chat_participants',
      indexes: [
        {
          unique: true,
          fields: ['userId', 'chatId'],
        },
      ],
    }
  );
```

Таблиця івентів:

```
  const Event = sequelize.define('event', {
    id: { type: DataTypes.INTEGER, primaryKey: true,
autoIncrement: true },
    type: { type: DataTypes.STRING, allowNull: false },
    content: { type: DataTypes.STRING, allowNull: true },
    isSeen: { type: DataTypes.BOOLEAN, allowNull: false,
defaultValue: false },
  });
```

Налаштування зв’язку Message та User:

```
User.hasMany(Message);
Message.belongsTo(User);
```

Налаштування зв’язку Event та User:

```
User.hasMany(Event);
Event.belongsTo(User);
```

Налаштування зв’язку Event та Chat:

```
Chat.hasMany(Event);
Event.belongsTo(Chat);
```

Налаштування зв’язку Message та Chat:

```
Chat.hasMany(Message);
Message.belongsTo(Chat);
```

Налаштування зв’язку ChatParticipants та User:

```
User.hasMany(ChatParticipants);
ChatParticipants.belongsTo(User);
```

Налаштування зв’язку ChatParticipants та Chat:

```
Chat.hasMany(ChatParticipants);
ChatParticipants.belongsTo(Chat);
```

Налаштування зв’язку ChatParticipants та Role:

```
Role.hasMany(ChatParticipants);
ChatParticipants.belongsTo(Role);
```

Авторизація користувачів. Першим кроком для розробки системи авторизації користувачів нами було створено схему роботи авторизації в рамках нашої чат платформи (рис. 3.2).

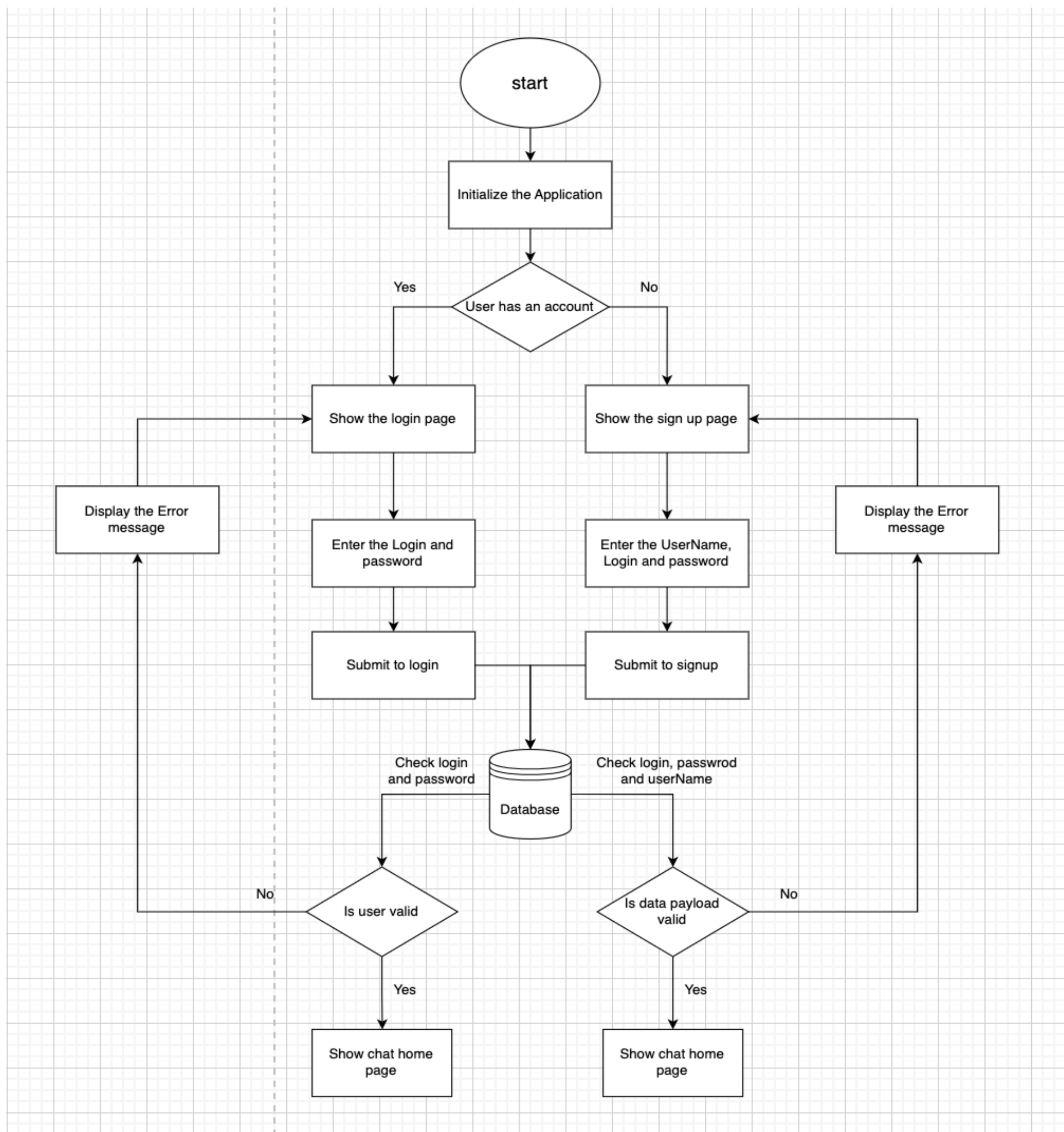

Рис. 3.2 – Схема авторизації користувачів

Схема авторизації користувача на рис. 3.2 відображає наступні кроки:

1. Ініціалізація програми. На цьому етапі програма завантажується та запускається.
2. Перевірка наявності облікового запису. На цьому етапі перевіряється, чи є у користувача обліковий запис у системі. Якщо є, то користувач переходить на сторінку входу. Якщо немає, то користувач переходить на сторінку реєстрації.
3. Відображення сторінки входу. На цьому етапі користувачеві відображається сторінка входу, де він може ввести свій логін і пароль.

4. Введення логіна та пароля. На цьому етапі користувач вводить свій логін і пароль.
5. Перевірка правильності логіна та пароля. На цьому етапі програма перевіряє, чи є введений логін і пароль правильними. Якщо так, то користувач переходить на головну сторінку чату. Якщо ні, то відображається повідомлення про помилку.
6. Відображення сторінки реєстрації. На цьому етапі користувачеві відображається сторінка реєстрації, де він може створити новий обліковий запис.
7. Введення імені користувача, логіна та пароля. На цьому етапі користувач вводить своє ім'я користувача, логін і пароль.
8. Перевірка правильності імені користувача, логіна та пароля. На цьому етапі програма перевіряє, чи є введені ім'я користувача, логін і пароль правильними. Якщо так, то користувач переходить на головну сторінку чату. Якщо ні, то відображається повідомлення про помилку.
9. Відображення головної сторінки чату. На цьому етапі користувачеві відображається головна сторінка чату, де він може спілкуватися з іншими користувачами.

Діаграма потоку на рис. 3.2 є прикладом простого процесу реєстрації. У більш складних системах процес реєстрації може включати в себе додаткові кроки, такі як підтвердження адреси електронної пошти або відправлення коду підтвердження на мобільний телефон.

У нашому додатку програмна реалізація авторизації користувачів здійснюється за допомогою JWT та бібліотеки bcrypt для хешування паролів. Надійність та захищеність облікових даних забезпечуються шифруванням та генерацією токенів. Давайте розглянемо основні етапи авторизації, представлені у нашому контролеру користувачів.

Функція для генерації JWT токена:

```
const generateJwt = (id, login) => {
  return jwt.sign({ id, login }, process.env.SECRET_KEY,
{ expiresIn: '24h' });
```

```
};
```

Клас контролера користувачів який зберігає в собі усе методи роботи з користувачами та їхніми записами:

```
class UserController {
  // Обробник реєстрації нового користувача
  async signUp(req, res, next) {
    // Отримання облікових даних з тіла запиту
    const { login, password, name } = req.body;
    console.log(req.body);
    // Перевірка наявності обов'язкових полів
    if (!login || !password) {
      return next(ApiError.badRequest('Wrong login or
password'));
    }
    // Перевірка наявності імені користувача
    if (!name) {
      return next(ApiError.badRequest('Please provide correct
name'));
    }
    // Перевірка, чи не існує користувача з таким же логіном
    const candidate = await User.findOne({ where:
{ login } });
    if (candidate) {
      return next(
        ApiError.badRequest('User with the same login already
exists')
      );
    }
    // Хешування паролю та створення нового користувача
    const hashPassword = await bcrypt.hash(password, 5);
    const user = await User.create({ login, password:
hashPassword, name });
    // Генерація та відправлення JWT токена у відповідь
    const token = generateJwt(user.id, user.login);
    return res.json({
      token,
      userName: user.name,
      userId: user.id,
      login: user.login,
    });
  }
  // Обробник входу користувача
  async login(req, res, next) {
    const { login, password } = req.body;
    // Пошук користувача за логіном
    const user = await User.findOne({ where: { login } });
    if (!user) {
      return next(ApiError.internal('User not found'));
    }
    // Порівняння хешу паролю з введеним паролем
```

```
      let comparePassword = bcrypt.compareSync(password,
user.password);
      if (!comparePassword) {
        return next(ApiError.internal('Wrong password'));
      }
      // Генерація та відправлення JWT токена у відповідь
      const token = generateJwt(user.id, user.login);
      return res.json({
        token,
        userName: user.name,
        userId: user.id,
        login: user.login,
      });
    }
    // Обробник перевірки токена
    async check(req, res, next) {
      // Генерація нового токена на основі інформації з
існуючого токена
      const token = generateJwt(req.user.id, req.user.login);
      return res.json({ token });
    }
  }
  // Експорт екземпляру класу UserController
  module.exports = new UserController();
```

Налаштування Socket.io. Як було зазначено у розділі методів та технологій вирішення задач наш бекенд сервер розроблений з використанням Express.js для реалізації HTTP серверу та Socket.io для реалізації WebSocket з’єднання. Програмна реалізація виглядає наступним чином:

```
// Підключення необхідних модулів та бібліотек
require('dotenv').config();
const express = require('express');
const { createServer } = require('http');
const { Server } = require('socket.io');

// Створення HTTP сервера
const PORT = process.env.PORT || 5000;
const app = express();
const httpServer = createServer(app);

// Використання middleware для авторизації та обробки
виняткових ситуацій
app.use(cors());
app.use(express.json());
app.use('/api', router);
app.use(errorHandler);
```

```
      // Створення WebSocket серверу за допомогою бібліотеки
Socket.io
      const io = new Server(httpServer, {
        cors: {
          origin: '*',
        },
      });
      // Налаштування логіки створення та приєднання клієнтів до
сокетних кімнат
      io.on('connection', (socket) => {
        console.log('Connected to socket.io');
        socket.on('setup', (userId) => {
          socket.join(userId);
          console.log(socket.rooms);
          socket.emit('connected');
        });
      // Подія приєднання до кімнати
        socket.on('join chat', (room) => {
          socket.join(room);
          console.log('User Joined Room: ' + room);
        });
      //Подія виходу з кімнати
        socket.on('left chat', (room) => {
          socket.join(room);
          console.log('User Left Room: ' + room);
        });
        socket.on('typing', (room) =>
socket.in(room).emit('typing'));
        socket.on('stop typing', (room) =>
socket.in(room).emit('stop typing'));
      // Подія створення нового повідомлення
        socket.on('new message', (newMessageRecieved) => {
          var chat = newMessageRecieved.chat;

          if (!chat.chatParticipants) return
console.log('chat.users not defined');

          chat.chatParticipants.forEach((user) => {
            if (user.userId === newMessageRecieved.userId) return;

            socket.in(user.userId).emit('message recieved',
newMessageRecieved);
            socket.in(user.userId).emit('new messages unreaded',
user);
          });
        });
      // Подія проглядання повідомлення
        socket.on('read messages', async ({ userId, chatId }) => {
          const chatParticipant = await
models.ChatParticipants.update(
            { unreadMessages: 0 },
            {
              where: {
```

```
          userId,
          chatId,
        },
        returning: true,
      }
    );

    socket.emit('message was read', ...chatParticipant[1]);
  });
  socket.on('disconnect', () => {
    console.log('disconnect');
  });
});
```

Створення користувацького інтерфейсу чат платформи, та з'єднання його з серверною частиною додатку. Для розробки клієнтської частини чат платформи була обрана бібліотека React. Розроблений додаток був з використання строго режиму, який рекомендовано використовувати авторами React.

```
ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Маршрутизація відбувається за допомогою бібліотеки react-router-dom [19]. В нашому додатку буде 4 сторінки: сторінка реєстрації, сторінка логіну, сторінка всіх чатів, та динамічна сторінка обраного чату. Програмна реалізація виглядає наступним чином:

```
// Створюємо компонент роутера
export default function Router() {
// Створюємо об'єкт роутера
  const router = createBrowserRouter([
    {
      path: '/',
      element: (
        <RedirectIfNotAuthenticated>
          <ChatListPage />
        </RedirectIfNotAuthenticated>
      ),
      children: [
```

```
        {
          path: 'chats/:chatId',
          element: <ChatPage />,
        },
      ],
      // ErrorBoundary: NotFoundPage,
    },
    {
      path: '/login',
      element: (
        <RedirectIfAuthenticated>
          <LoginPage />
        </RedirectIfAuthenticated>
      ),
    },
    {
      path: '/signup',
      element: (
        <RedirectIfAuthenticated>
          <SignUpPage />
        </RedirectIfAuthenticated>
      ),
    },
  ]);
  return <RouterProvider router={router} />;
}
```

Для того, щоб дізнатися геолокацію користувача ми будемо використовувати ip клієнта, який за допомогою сервіса ipapi буде повертати інформацію щодо міста та країни користувача звідки був здійснений запит до серверу. Потім ця інформація буде відправлятися в заголовках кожного запиту до серверу в полі під назвою location. Виглядає це наступним чином:

```
// Створюємо змінну де пізніше будемо зберігати локацію
користувача
let location: any;
// Створюємо екземпляр декоратора для запитів
const axiosInstance = axios.create({
  baseURL: baseURI + '/api',
  headers: {
    'Cache-Control': 'no-cache',
    Pragma: 'no-cache',
    Expires: '0',
    API_SECRET_KEY: import.meta.env.VITE_BASE_API_KEY,
  },
});

axiosInstance.interceptors.request.use(async function
(config: any) {
```

```
      // Отримуємо з локального сховища браузера токен авторизації
        const authData =
JSON.parse(localStorage.getItem('authData') || '{}');
      // Якщо токен авторизації існує - додаємо його до кожного
запиту на сервер
        if (authData?.token) {
          config.headers.Authorization = 'Bearer ' +
authData.token;
        }

      // Якщо локація не існує отримуємо локацію с сервісу ipapi
        if (!location) {
          location = await (await
fetch('https://ipapi.co/json/')).json();
        }
          // Додаємо до кожного запиту до серверу локацію
користувача
        config.headers.location = location.country_name;

        return config;
      });
```

Створення middleware для обробки геолокації користувача з подальшим додаванням його до місцевого чату. Завдяки системи middleware в express, ми маємо можливість створити middleware, або інакше кажучи проміжну функцію яка буде виконуватися до будь-яких контролерів. В цій функції ми будемо парсити дані с запиту дістаючи токен авторизації та локацію користувача. Якщо токен валідний то далі ми перевіряємо чи є поточний користувач учасником місцевого чату. Програмна реалізація виглядає наступним чином:

```
      // Імпортуємо необхідні модулі
     const jwt = require('jsonwebtoken');
      const { Chat, Role, ChatParticipants, Event } =
require('../models/models');
      const { ROLES } = require('../constants/roles');

      module.exports = async function (req, res, next) {
      // Якщо метод API запиту options, то пропускаємо всі
перевірки
        if (req.method === 'OPTIONS') {
          next();
        }
        try {
      // Перевірка токена авторизації на валідність
```

```
          const token = req.headers.authorization.split(' ')[1];
          if (!token) {
      // Якщо токен не валідний, повертаємо помилку з кодом 401
            return res.status(401).json({ message: 'Not
authorized });
          }
          const decoded = jwt.verify(token,
process.env.SECRET_KEY);
          req.user = decoded;

      // Отримуємо локацію користувача
          const chatName = req.headers.location;
          const userId = decoded.id;
      // Шукаємо місцевий чат по цій локації
          const chat = await Chat.findOne({ where: { name:
chatName } });
          if (chat) {
      // Якщо чат знайдено перевіряємо чи являється поточний
користувач учасником місцевого чату
            const foundParticipant = await
ChatParticipants.findOne({
              where: {
                userId,
                chatId: chat.id,
              },
            });
      // Якщо поточний користувач вже учасник чата, то пропускаємо
виконання функції
            if (foundParticipant) {
              return next();
            }

            const role = await Role.findOne({ where: { role:
ROLES.USER } });
      // Якщо поточний користувач не являється учасником чата -
добавляємо його у місцевий чат з ролю USER
            await ChatParticipants.create({
              userId,
              chatId: chat.id,
              roleId: role.id,
            });
      // Створюємо івент, що користувач був доданий до місцевого
чата, щоб потім по цьому івенту показати модальне вікно з
інформацією щодо додавання на клієнті
            await Event.create({
              type: 'CHAT_ADDITION',
              chatId: chat.id,
              userId,
            });
          }
          next();
        } catch (e) {
          res.status(401).json({ message: 'Not authorized' });
```

```
    }
};
```

3.2. Опис інтерфейсу користувача

За замовчуванням, перше що баче користувач – екран авторизації додатку (рис. 3.3). Користувач має можливість як ввести логін і пароль, так і перейти до екрану реєстрації (рис. 3.4) натиснувши посилання “Sign up”.

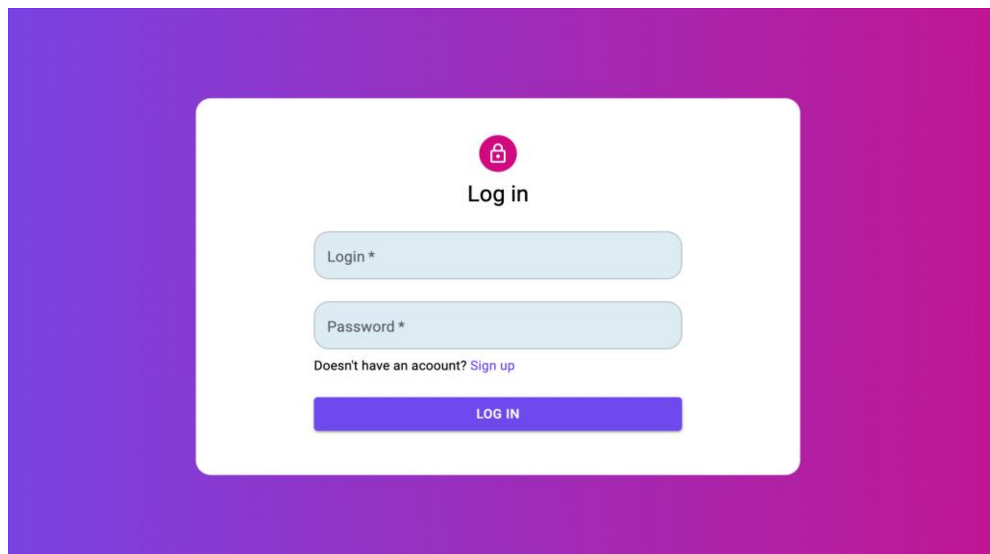

Рис. 3.3. Екран авторизації

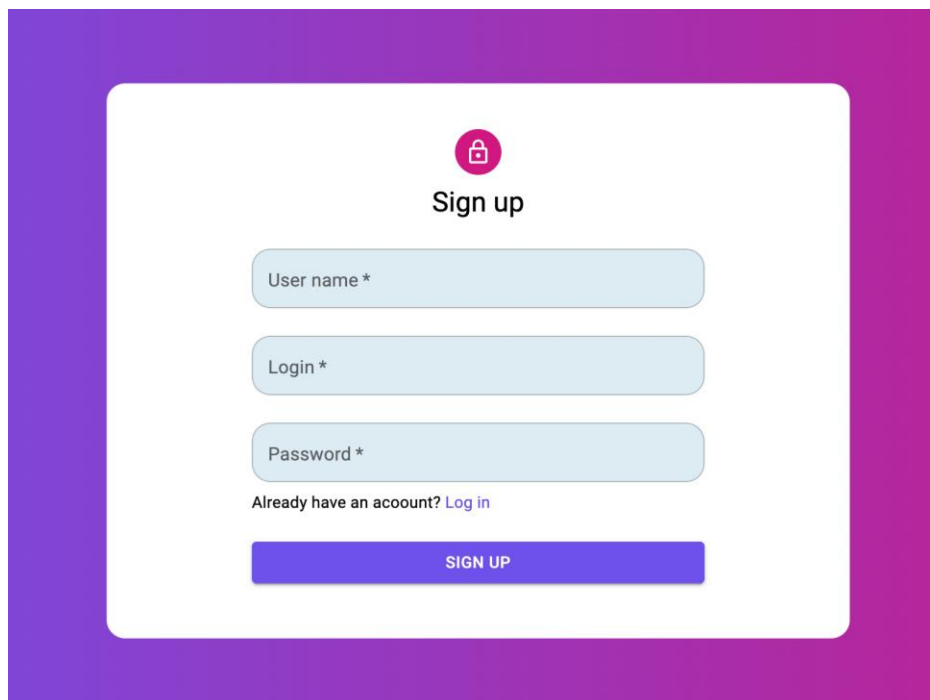

Рис. 3.4. Екран реєстрації

Також користувач завжди може повернутися з екрана реєстрації (рис. 3.4) до екрану авторизації (рис. 3.3) натиснувши посилання “Log in”.

Після того як користувач ввів необхідні дані для реєстрації (рис. 3.5), та натиснув кнопку “Sign up”, після успішної реєстрації його зустріне модальне вікно про те що він був доданий до чату його локації (рис. 3.6).

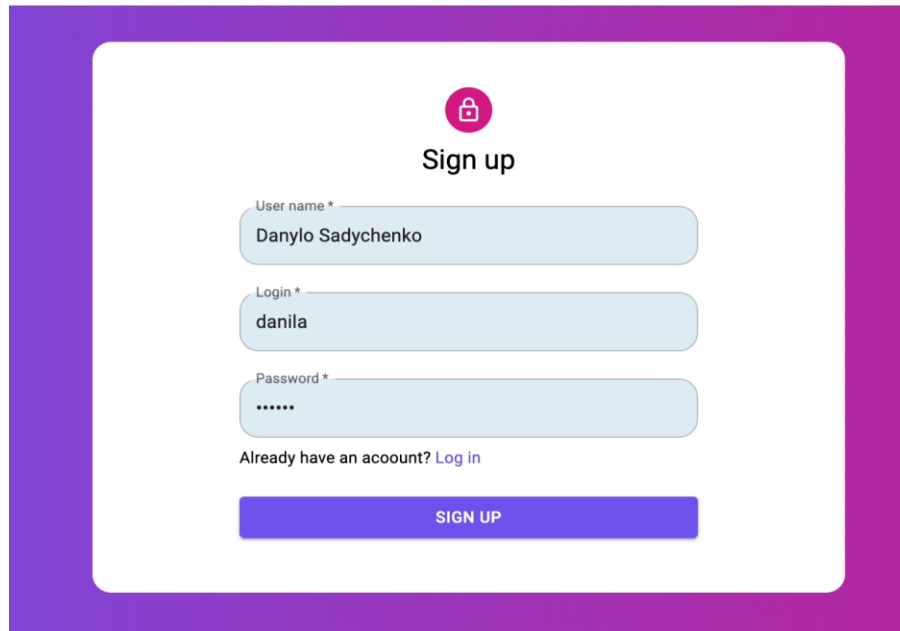

Рис. 3.5. Заповнення реєстраційної форми

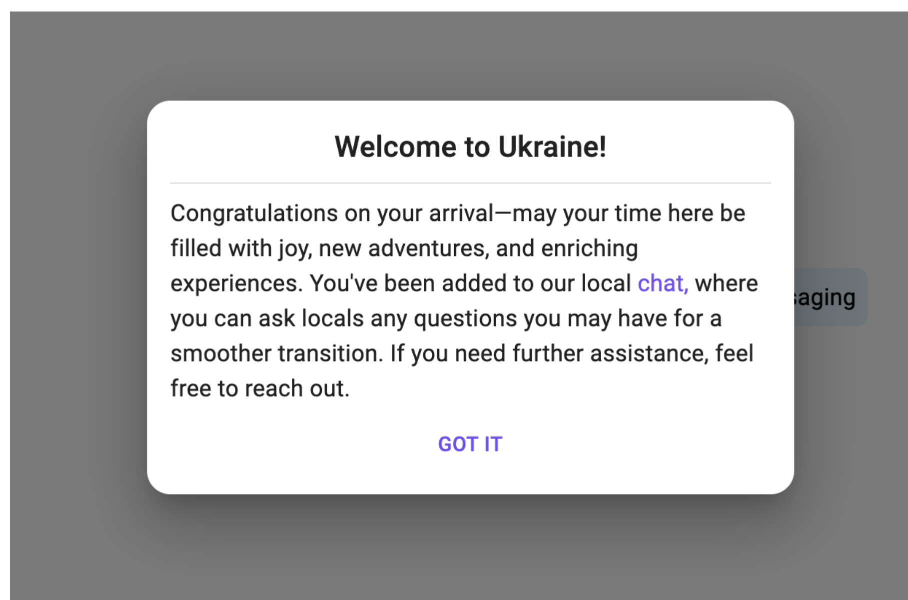

Рис. 3.6. Модальне вікно додавання до чату

Після того як модальне вікно буде закрите, користувач буде навігований на сторінку місцевого чату, до якого він щойно був доданий (рис. 3.7).

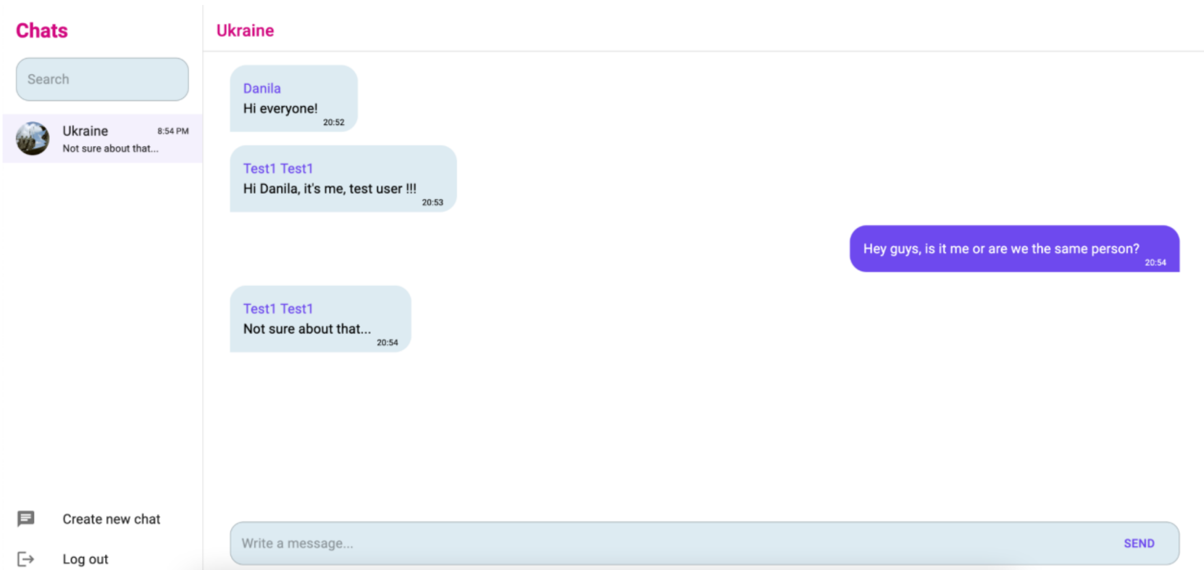

Рис. 3.7. Інтерфейс авторизованного користувача

Також натиснувши на “Create new chat” кнопку, користувач побаче модальне вікно створення нового чату (рис. 3.8), де йому буде потрібно обов’язково вести назву чата та вести посилання на картинку чата.

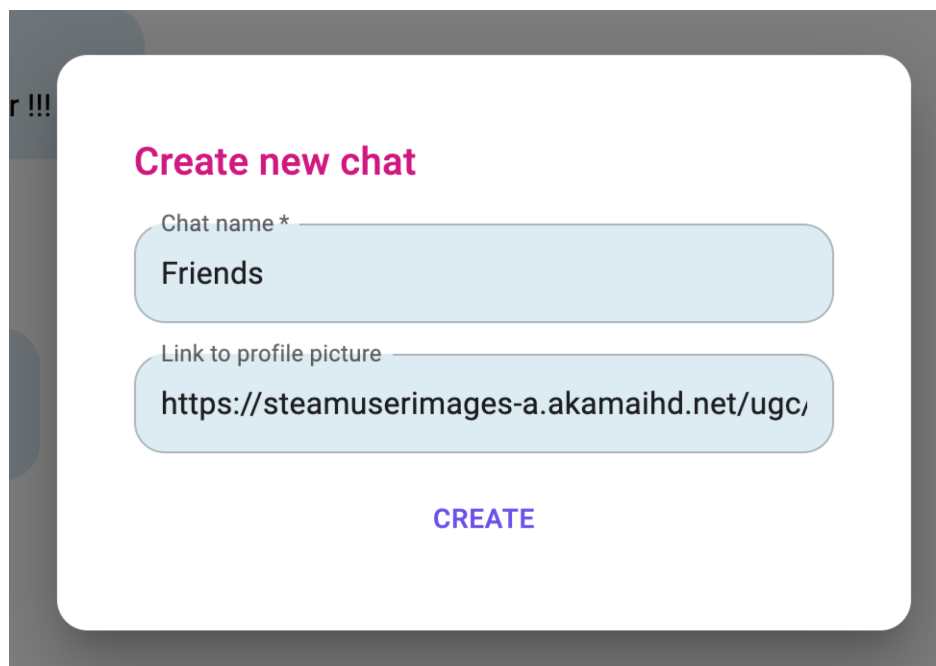

Рис. 3.8. Модальне вікно створення нового чату

Після створення чату, користувач буде перенаправлен на сторінку нового чату, де не буде повідомлень, і де він буде єдиним учасником. Натиснувши на названня чату, користувач побаче інформацію щодо чату, малюнок чату, кількість учасників, та форми для додавання нових учасників (рис. 3.9).

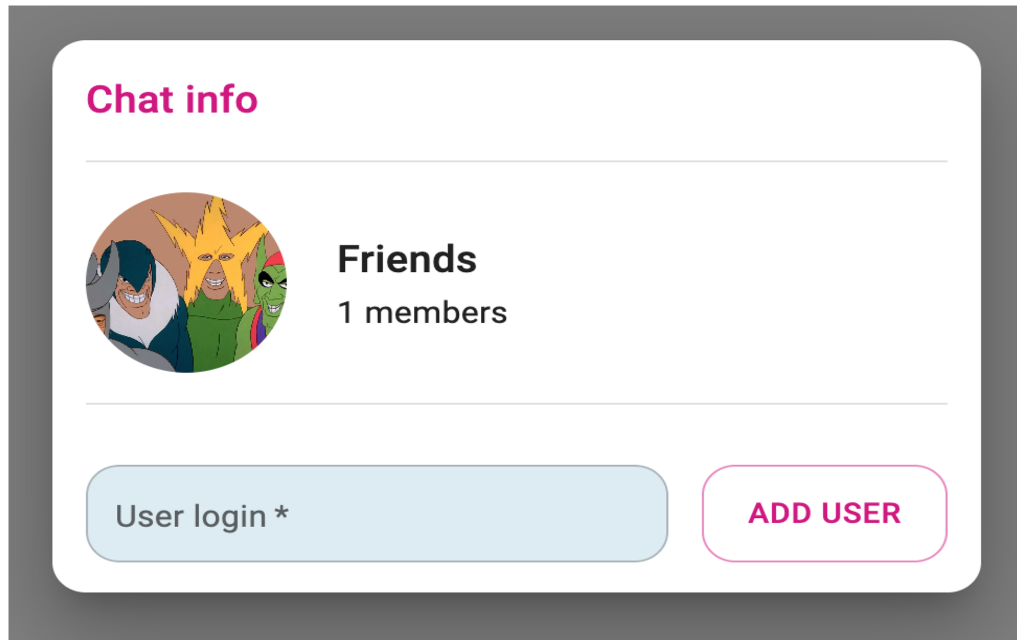

Рис. 3.9. Інформація щодо чат кімнати

Після заповнення текстового поля та натиснувши кнопку “Add user”, якщо введений користувач існує - він буде доданий до чату, якщо доданий користувач не існує, буде відображення відповідна помилка (рис. 3.10).

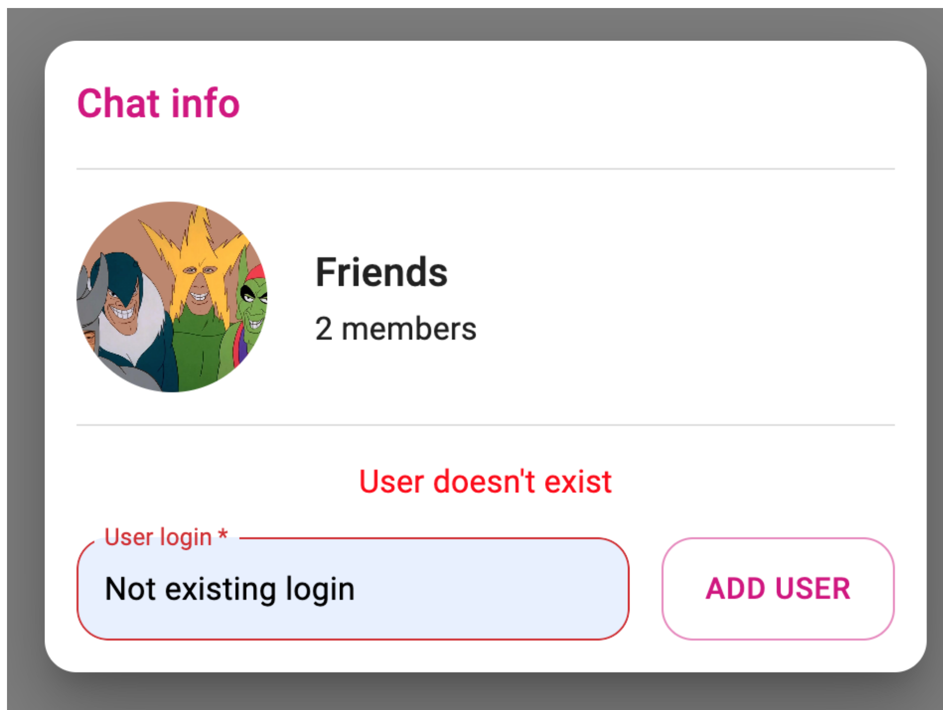

Рис. 3.10. Додавання до чату не існуючого користувача

Раніше доданий користувач до чату теперь бачить новий чат, до якого віе був доданий, також як і одне непрочитане повідомлення в новому чаті (рис. 3.11).

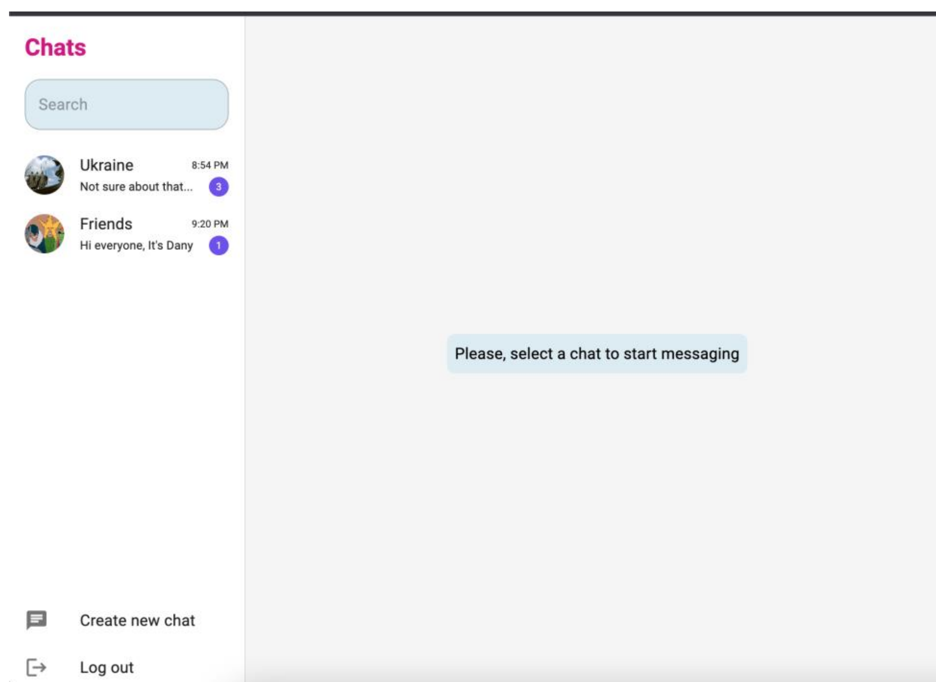

Рис. 3.11. Відображення нового чату і непрочитаного повідомлення

Також доданий до чату користувач може переглядати повідомлення та відправляти свої. (рис. 3.12)

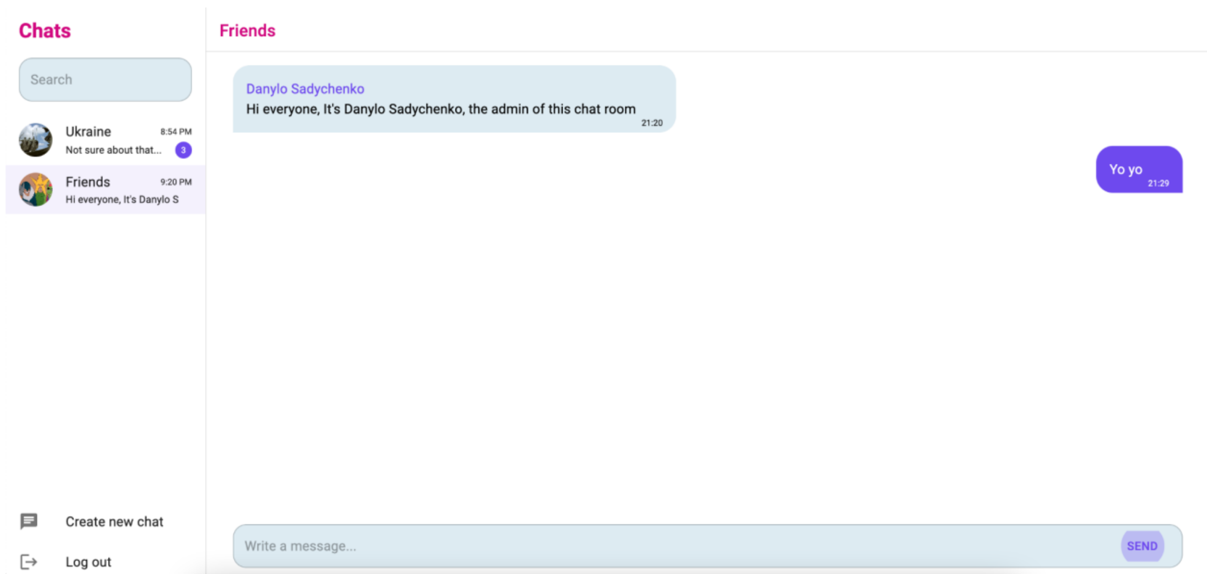

Рис. 3.12. Переглядання повідомлень та відправка власних у новому чаті

Слід зазначити що користувач що був доданий до чату, має роль USER, тому він не має можливості додавати нових користувачів до чату. (рис. 3.13)

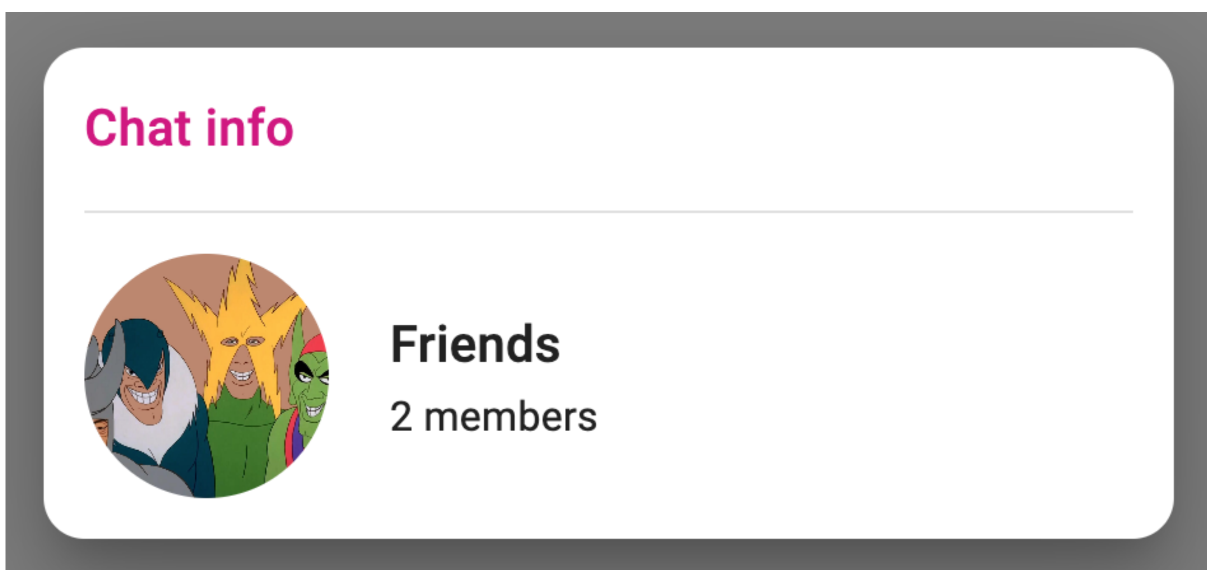

Рис. 3.13. Для звичайного користувача відсутнє поле для додавання нових користувачів до чату

Також використовуючи поле для пошуку чатів, користувач має можливість знайти чати що йому до вподоби (рис. 3.14).

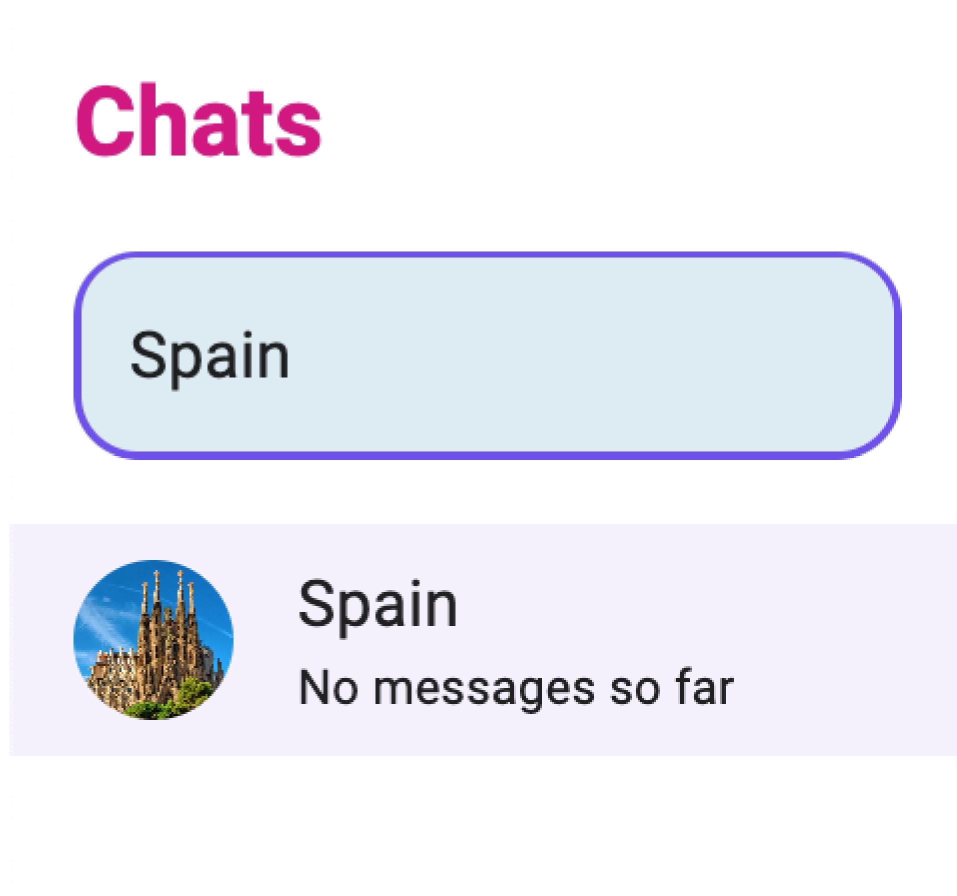

Рис. 3.14. Пошук нових чатів

Перед тим як мати можливість відправляти повідомлення, користувач зобов'язаний вступити до чату натиснувши кнопку “Join chat” (рис. 3.15).

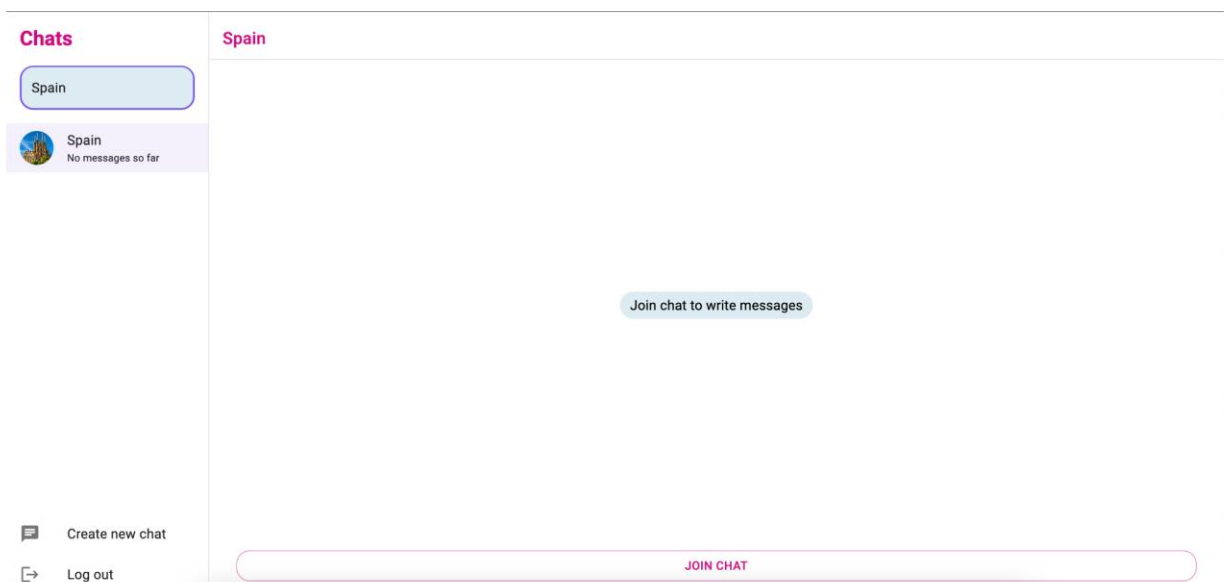

Рис. 3.15. Зовнішній вигляд чату в якому немає користувача

3.3. Дослідження ефективності роботи розробленого додатку

Навантажувальний тест був проведений для оцінки продуктивності та масштабованості системи в умовах різних змодельованих сценаріїв користувачів. Artillery, потужний інструмент для навантажувального тестування з відкритим вихідним кодом, був використаний для імітації реальної поведінки користувачів та взаємодій із системою.

Основними цілями цього навантажувального тесту були:

- Оцінка здатності системи обробляти зростаючу кількість одночасних з'єднань.
- Ефективне управління випуском подій.
- Оцінка загальної продуктивності в умовах тривалого навантаження.

Скрипт тесту, реалізований за допомогою Artillery, включає серію дій користувачів, таких як налаштування з'єднання, приєднання та вихід з чатів, відправлення повідомлень, читання повідомлень і відключення. Ці дії були розроблені для імітації взаємодій користувачів із системою у реальному світі.

Шляхом налаштування таких параметрів, як швидкість надходження, час обмірковування та частота випуску подій, ми намагалися відтворити сценарії навантаження, які система може зустріти під час пікового використання. Тривалість тесту, швидкість нових з'єднань та мінливість частоти випуску подій були ретельно налаштовані, щоб отримати уявлення про поведінку системи під навантаженням.

Результати цього навантажувального тесту будуть використані для:

- Виявлення потенційних вузьких місць.
- Оптимізації продуктивності системи.
- Забезпечення масштабованості системи для задоволення потреб зростаючої бази користувачів.

Завдяки використанню Artillery очікується отримати комплексне розуміння можливостей системи обробляти одночасні з'єднання та керувати подіями.

Програмна реалізація тесту навантаження виглядає наступним чином:

```
config:
  target: 'http://localhost:5000'
  phases:
    - duration: 300 # Increase the test duration to 5 minutes
      arrivalRate: 40 # Increase the arrival rate to 40 new connections per second
  engines:
    socketio-v3: {}

scenarios:
  - name: Load Test Scenario
    engine: socketio-v3
    flow:
      # Introduce a small think time before starting
      - think: 1

      # Emit a 'setup' event to simulate user connection and setup
      - emit:
          channel: 'setup'
          data: '1'

      # Introduce a pause/think time between actions
      - think: 5

      # Emit a 'join chat' event to simulate users joining a chat room
      - emit:
          channel: 'join chat'
          data: '4'

      # Introduce a pause/think time between actions
      - think: 5

      # Emit a 'new message' event to simulate users sending messages
      - emit:
          channel: 'new message'
          data:
            {
              userId: '1',
              chat: { chatParticipants: [{ userId: '2', chatId: '4' }] },
            }

      # Introduce a pause/think time between actions
      - think: 5

      # Emit a 'read messages' event to simulate users reading messages
```

```
- emit:
    channel: 'read messages'
    data: { userId: '1', chatId: '4' }

# Introduce a pause/think time between actions
- think: 5

# Emit a 'disconnect' event to simulate users disconnecting
- emit: 'disconnect'
```

Після роботи тесту навантаження ми отримали наступний звіт відображений на (рис. 3.16).

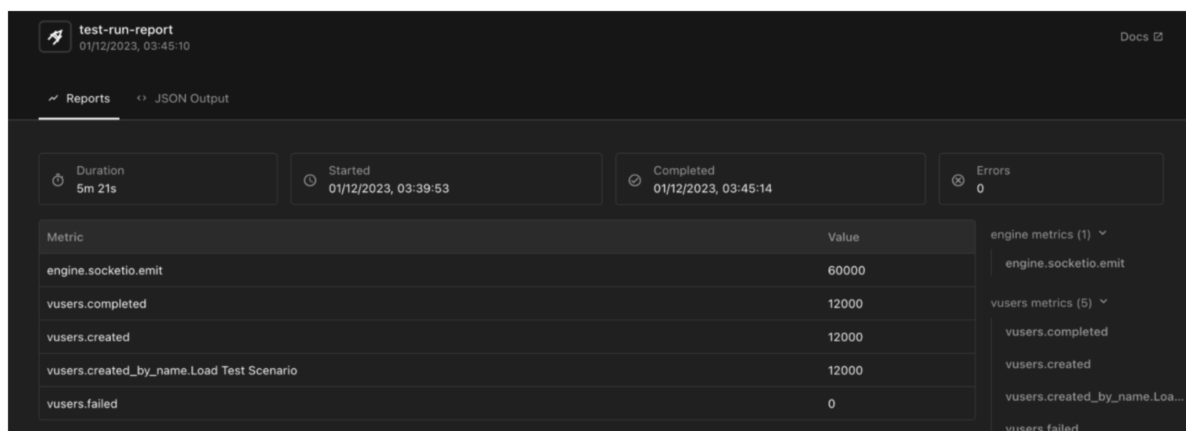

Рис. 3.16. Звіт щодо тестування навантаження

Можемо зробити наступні висновки з тесту на основі наданого на (рис. 3.16) звіту:

1. Загальний огляд тесту:
    1.1. Загальна кількість віртуальних користувачів, створених: 12,000.
    1.2. Загальна кількість віртуальних користувачів, завершених: 12,000.
    1.3. Загальна кількість віртуальних користувачів, неуспішних: 0.
2. Темпи відсилання подій:
    2.1. Темп відсилання подій Socket.IO: 186 подій на секунду.
3. Розподіл тривалості сесій:
    3.1. Мінімальна тривалість сесії: 21,001.9 мілісекунд.
    3.2. Максимальна тривалість сесії: 21,490.5 мілісекунд.
    3.3. Середня тривалість сесії: 21,051.3 мілісекунд.

3.4. Медіана тривалості сесії (p50): 20,958.1 мілісекунд.
3.5. 75-й відсоток (p75): 20,958.1 мілісекунд.
3.6. 90-й відсоток (p90): 20,958.1 мілісекунд.
3.7. 95-й відсоток (p95): 20,958.1 мілісекунд.
3.8. 99-й відсоток (p99): 21,381.5 мілісекунд.
3.9. 99.9-й відсоток (p999): 21,381.5 мілісекунд.

4. Тривалість тесту:
   4.1. Приблизно 3,700 секунд (з першого до останнього лічильника та метрики).
5. Спостереження:
   5.1. Система добре справилася з навантаженням, не було повідомлень про неуспіхи серед 12,000 завершених віртуальних користувачів.
   5.2. Темп відсилання подій Socket.IO залишався стабільним на рівні 186 подій на секунду.
   5.3. Розподіл тривалості сесій свідчить про стабільну продуктивність, із більшістю сесій, які потрапляють в медіану та відсотки.

ВИСНОВКИ

На основі проведеного аналізу існуючих інтерактивних чат-платформ та месенджерів, а також дослідження їхніх функціональних можливостей та особливостей використання геолокації для організації комунікації, можна зробити висновки, що існуючі рішення, хоч і відповідають певним потребам, мають свої обмеження.

Враховуючи ці обмеження, наша кваліфікаційна робота спрямована на створення інтерактивної чат-платформи для знайомств та спілкування між користувачами з використанням геолокації, з практичною метою підвищення якості спілкування та знайомства між новими користувачами, за рахунок використання геолокації в якості ключової функціональної особливості.

Технологічний стек для розробки включає сучасні фреймворки та мови програмування, такі як React, Express, Socket.io, PostgreSQL, Sequelize, забезпечуючи високий рівень продуктивності та гнучкість.

Узагальнюючи, розробка інтерактивної чат-платформи, яка акцентує на нових знайомствах та використовує геолокацію для автентичної та цікавої взаємодії, є важливим кроком у напрямку створення продукту, що задовольнить вимоги сучасного користувача, сприяючи активній та різноманітній комунікації.

Під час виконання даної кваліфікаційної роботи були виконані
наступні задачі:

1. Аналіз існуючих рішень.
2. Розробка концепції платформи.
3. Вибір технологій і інструментів.
4. Розробка основного функціоналу.
5. Розробка інтерфейсу користувача.
6. Розробка системи контролю доступу, яка буде обмежувати доступ користувачів до функціональності чи контенту за певними умовами.
7. Налаштування взаємодії клієнту з сервером використовуючи HTTP та WebSocket.

Також було проведено дослідження ефективності роботи додатку за підсумками якого було доведено, що додаток спроможний витримувати навантаження в 186 подій в секунду, що можна прирівняти до одночасної відправи повідомлення 186 користувачами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. “WebSocket Protocol”: [Електронний ресурс]. – Режим доступу: https://www.wallarm.com/what/a-simple-explanation-of-what-a-websocket-is (дата звернення 09.10.2023).
2. “What Is Load Balancing?”: [Електронний ресурс]. – Режим доступу: https://www.nginx.com/resources/glossary/load-balancing/ (дата звернення 09.10.2023).
3. “Load Balancing Algorithm Explained”: [Електронний ресурс]. – Режим доступу: https://aws.amazon.com/what-is/load-balancing/ (дата звернення 09.10.2023).
4. “React Reference Overview”: [Електронний ресурс]. – Режим доступу: https://react.dev/reference/react (дата звернення 12.11.2023).
5. “Virtual DOM and Internals”: [Електронний ресурс]. – Режим доступу: https://legacy.reactjs.org/docs/faq-internals.html#:~:text=What%20is%20the%20Virtual%20DOM,This%20process%20is%20called%20reconciliation. (дата звернення 12.11.2023).
6. “Concepts Overview”: [Електронний ресурс]. – Режим доступу: https://012.vuejs.org/guide/. (дата звернення 12.11.2023).
7. “What is Angular?”: [Електронний ресурс]. – Режим доступу: https://angular.io/guide/what-is-angular. (дата звернення 12.11.2023).
8. “Stack Overflow Trends”: [Електронний ресурс]. – Режим доступу: https://insights.stackoverflow.com/trends?tags=reactjs%2Cvue.js%2Cangular%2Csvelte%2Cangularjs%2Cvuejs3 (дата звернення 12.11.2023).
9. “Express/Node introduction”: [Електронний ресурс]. – Режим доступу: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction (дата звернення 14.11.2023).
10. “Introduction”: [Електронний ресурс]. – Режим доступу: https://socket.io/docs/v4/ (дата звернення 14.11.2023).

11. “Why Use PostgreSQL For Your Next Project?”: [Електронний ресурс]. – Режим доступу: https://cleancommit.io/blog/why-use-postgresql-for-your-next-project/#:~:text=PostgreSQL's%20main%20benefit%20is%20it's,of%20writing%20or%20updating%20data. (дата звернення 14.11.2023).
12. “Socket.IO with Node.Js + Express” : [Електронний ресурс]. – Режим доступу: https://medium.com/kocfinanstech/socket-io-with-node-js-express-5cc75aa67cae (дата звернення 15.11.2023).
13. “Real time communication with Socket.IO and Node.js” : [Електронний ресурс]. – Режим доступу: https://dev.to/admirnisic/real-time-communication-with-socketio-and-nodejs-3ok2 (дата звернення 16.11.2023).
14. “Create a Basic Server with Express.js”: [Електронний ресурс]. – Режим доступу: https://medium.com/geekculture/create-a-basic-server-with-express-js-really-basic-but-delicious-c5cceaca1c60 (дата звернення 19.11.2023).
15. “How to Build a Secure Server with Node.js and Express and Upload Images with Cloudinary”: [Електронний ресурс]. – Режим доступу: https://www.freecodecamp.org/news/build-a-secure-server-with-node-and-express/ (дата звернення 19.11.2023).
16. “What I learned about WebSockets by building a real-time chat application using Socket.IO.”: [Електронний ресурс]. – Режим доступу: https://medium.com/@bootcampmillionaire/what-i-learned-about-websockets-by-building-a-real-time-chat-application-using-socket-io-3d9e163e504 (дата звернення 19.11.2023).
17. “How to Build a Chat App with WebSockets and Node.js”: [Електронний ресурс]. – Режим доступу: https://www.cometchat.com/tutorials/how-to-build-a-chat-app-with-websockets-and-node-js (дата звернення 19.11.2023).
18. “Building a WebSocket Chat service for Cloud Run”: [Електронний ресурс]. – Режим доступу: https://cloud.google.com/run/docs/tutorials/websockets (дата звернення 19.11.2023).

19. “Feature overview”: [Електронний ресурс]. – Режим доступу: https://reactrouter.com/en/main/start/overview#feature-overview (дата звернення 20.11.2023).
20. “Exploring User Authentication Methods in Express Node Server”: [Електронний ресурс]. – Режим доступу: https://medium.com/@himalpgautam/exploring-user-authentication-methods-in-express-node-server-e7704775ef4e (дата звернення 21.11.2023).
21. “PART-2: User Authentication with JWT and Express.js in Node.js: A JOIful Journey”: [Електронний ресурс]. – Режим доступу: https://vivekumar08.medium.com/part-2-user-authentication-with-jwt-and-express-js-in-node-js-a-joiful-journey-51565ae376b8 (дата звернення 21.11.2023).
22. “Sequelize.js Best Practices”: [Електронний ресурс]. – Режим доступу: https://climbtheladder.com/10-sequelize-js-best-practices/ (дата звернення 23.11.2023).
23. “Sequelize using Postgres dialect”: [Електронний ресурс]. – Режим доступу: https://medium.com/@jatin.jain_69313/sequelize-using-postgres-dialect-d342b6c64092 (дата звернення 23.11.2023).
24. “Git: Understanding the basics.”: [Електронний ресурс]. – Режим доступу: https://medium.com/@onejohi/git-understanding-the-basics-ba004a20dacc. (дата звернення 23.11.2023).

ДОДАТОК А

ЛІСТИНГ ПРОГРАМИ

```
// index.js головний файл серверу
require('dotenv').config();
const express = require('express');
const { createServer } = require('http');
const { Server } = require('socket.io');
const sequelize = require('./db');
const models = require('./models/models');
const cors = require('cors');
const router = require('./routes/index');
const errorHandler = require('./middleware/ErrorHandlingMiddleware');
const path = require('path');
const { countries } = require('./countries');

const PORT = process.env.PORT || 5000;

const app = express();
const httpServer = createServer(app);

app.use(cors());
app.use(express.json());
app.use('/api', router);

app.use(errorHandler);

const io = new Server(httpServer, {
  cors: {
    origin: '*',
  },
});
io.on('connection', (socket) => {
  console.log('Connected to socket.io');
  socket.on('setup', (userId) => {
    socket.join(userId);
    console.log(socket.rooms);
    socket.emit('connected');
  });

  socket.on('join chat', (room) => {
    socket.join(room);
    console.log('User Joined Room: ' + room);
  });

  socket.on('left chat', (room) => {
    socket.join(room);
    console.log('User Left Room: ' + room);
  });
  socket.on('typing', (room) => socket.in(room).emit('typing'));
  socket.on('stop typing', (room) => socket.in(room).emit('stop typing'));

  socket.on('new message', (newMessageRecieved) => {
    var chat = newMessageRecieved.chat;

    if (!chat?.chatParticipants) return console.log('chat.chatParticipants not
defined');

    chat.chatParticipants.forEach((user) => {
```

```
      if (user.userId === newMessageRecieved.userId) return;

      socket.in(user.userId).emit('message recieved', newMessageRecieved);
      socket.in(user.userId).emit('new messages unreaded', user);
    });
  });
  socket.on('read messages', async ({ userId, chatId }) => {
    const chatParticipant = await models.ChatParticipants.update(
      { unreadMessages: 0 },
      {
        where: {
          userId,
          chatId,
        },
        returning: true,
      }
    );

    socket.emit('message was read', ...chatParticipant[1]);
  });
  socket.on('disconnect', () => {
    console.log('disconnect');
  });
});

const start = async () => {
  try {
    await sequelize.authenticate();
    await sequelize.sync({ alter: true });
    httpServer.listen(PORT, () =>
      console.log(`Server started on port ${PORT}`)
    );
    // countries.map(async (country) => {
    //   const chat = await models.Chat.create({ name: country });
    // });
  } catch (e) {
    console.log(e);
  }
};

start();
// db.js файл для приєднання до бази даних
const { Sequelize } = require('sequelize');

module.exports = new Sequelize(
  process.env.DB_NAME, // Название БД
  process.env.DB_USER, // Пользователь
  process.env.DB_PASSWORD, // ПАРОЛЬ
  {
    dialect: 'postgres',
    host: process.env.DB_HOST,
    port: process.env.DB_PORT,
    logging: false,
  }
);
// chatRouter.js
const Router = require('express');
const chatController = require('../controllers/chatController');
const authMiddleware = require('../middleware/authMiddleware');
const router = new Router();

//authMiddleware
router.post('/create', authMiddleware, chatController.createChat);
router.get('/:id', authMiddleware, chatController.getChat);
```

```
router.get('/', authMiddleware, chatController.getAllChats);
router.post('/addParticipant', authMiddleware, chatController.addParticipant);
router.post('/joinChat', authMiddleware, chatController.joinChat);

module.exports = router;

// eventRouter.js
const Router = require('express');
const authMiddleware = require('../middleware/authMiddleware');
const eventController = require('../controllers/eventController');
const router = new Router();

//authMiddleware
router.get('/', authMiddleware, eventController.getUserEvents);
router.post('/acknowledgeEvent', authMiddleware,
eventController.acknowledgeEvent);

module.exports = router;

//index.js
const Router = require('express');
const router = new Router();
const userRouter = require('./userRouter');
const chatRouter = require('./chatRouter');
const roleRouter = require('./roleRouter');
const messageRouter = require('./messageRouter');
const eventRouter = require('./eventRouter');

router.use('/user', userRouter);
router.use('/chat', chatRouter);
router.use('/role', roleRouter);
router.use('/message', messageRouter);
router.use('/event', eventRouter);

module.exports = router;

//messageRouter.js
const Router = require('express');
const router = new Router();
const authMiddleware = require('../middleware/authMiddleware');
const messageController = require('../controllers/messageController');

router.post('/createMessage', authMiddleware, messageController.createMessage);
router.post('/getAllMessages', authMiddleware,
messageController.getAllMessages);
router.post('/editMessage', authMiddleware, messageController.editMessage);

module.exports = router;

//roleRouter.js
const Router = require('express');
const router = new Router();
const roleController = require('../controllers/roleController');
const authMiddleware = require('../middleware/authMiddleware');

router.post('/createRole', authMiddleware, roleController.createRole);

module.exports = router;

//userRouter.js
const Router = require('express')
const router = new Router()
const userController = require('../controllers/userController')
const authMiddleware = require('../middleware/authMiddleware')
```

```
router.post('/signUp', userController.signUp)
router.post('/login', userController.login)
router.get('/auth', authMiddleware, userController.check)

module.exports = router

//models.js
const sequelize = require('../db');
const { DataTypes } = require('sequelize');

const User = sequelize.define('user', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  login: { type: DataTypes.STRING, unique: true, allowNull: false },
  password: { type: DataTypes.STRING, allowNull: false },
  name: { type: DataTypes.STRING, allowNull: false },
});

const Role = sequelize.define('role', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  role: {
    type: DataTypes.STRING,
    allowNull: false,
    unique: true,
  },
});

const Chat = sequelize.define('chat', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  name: { type: DataTypes.STRING, allowNull: false, unique: true },
  image: { type: DataTypes.STRING, allowNull: true },
});

const Message = sequelize.define('message', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  message: { type: DataTypes.STRING, allowNull: false },
});

const ChatParticipants = sequelize.define(
  'chatParticipants',
  {
    id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
    unreadMessages: {
      type: DataTypes.INTEGER,
      allowNull: false,
      defaultValue: 0,
    },
  },
  {
    tableName: 'chat_participants',
    indexes: [
      {
        unique: true,
        fields: ['userId', 'chatId'],
      },
    ],
  }
);

const Event = sequelize.define('event', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  type: { type: DataTypes.STRING, allowNull: false },
  content: { type: DataTypes.STRING, allowNull: true },
  isSeen: { type: DataTypes.BOOLEAN, allowNull: false, defaultValue: false },
```

```
});

User.hasMany(Message);
Message.belongsTo(User);

User.hasMany(Event);
Event.belongsTo(User);

Chat.hasMany(Event);
Event.belongsTo(Chat);

Chat.hasMany(Message);
Message.belongsTo(Chat);

User.hasMany(ChatParticipants);
ChatParticipants.belongsTo(User);

Chat.hasMany(ChatParticipants);
ChatParticipants.belongsTo(Chat);

Role.hasMany(ChatParticipants);
ChatParticipants.belongsTo(Role);

module.exports = {
  User,
  Role,
  Chat,
  Message,
  ChatParticipants,
  Event,
};

// authMiddleware.js
const jwt = require('jsonwebtoken');
const { Chat, Role, ChatParticipants, Event } = require('../models/models');
const { ROLES } = require('../constants/roles');

module.exports = async function (req, res, next) {
  if (req.method === 'OPTIONS') {
    next();
  }
  try {
    const token = req.headers.authorization.split(' ')[1];
    if (!token) {
      return res.status(401).json({ message: 'Не авторизован' });
    }
    const decoded = jwt.verify(token, process.env.SECRET_KEY);
    req.user = decoded;

    const chatName = req.headers.location;
    const userId = decoded.id;
    const chat = await Chat.findOne({ where: { name: chatName } });
    if (chat) {
      const foundParticipant = await ChatParticipants.findOne({
        where: {
          userId,
          chatId: chat.id,
        },
      });

      if (foundParticipant) {
        return next();
      }
```

```
      const role = await Role.findOne({ where: { role: ROLES.USER } });

      await ChatParticipants.create({
        userId,
        chatId: chat.id,
        roleId: role.id,
      });

      await Event.create({
        type: 'CHAT_ADDITION',
        chatId: chat.id,
        userId,
      });
    }
    next();
  } catch (e) {
    res.status(401).json({ message: 'Не авторизован' });
  }
};

// ErrorHandlingMiddleware.js
const ApiError = require('../error/ApiError');

module.exports = function (err, req, res, next) {
  if (err instanceof ApiError) {
    return res.status(err.status).json({ message: err.message });
  }
  return res.status(500).json({ message: 'Непредвиденная ошибка!' });
};

// ApiError.js
class ApiError extends Error {
  constructor(status, message) {
    super();
    this.status = status;
    this.message = message;
  }

  static badRequest(message) {
    return new ApiError(400, message);
  }

  static internal(message) {
    return new ApiError(500, message);
  }

  static forbidden(message) {
    return new ApiError(403, message);
  }
}

module.exports = ApiError;

//chatController.js
const ApiError = require('../error/ApiError');
const {
  Chat,
  ChatParticipants,
  Role,
  User,
  Message,
  Event,
} = require('../models/models');
const { ROLES } = require('../constants/roles');
```

```
class ChatController {
  async createChat(req, res, next) {
    const { name, image } = req.body;
    const userId = req.user.id;

    if (!name) {
      return next(ApiError.badRequest('Please provide correct chat name'));
    }
    const newChat = await Chat.create({ name, image });
    const role = await Role.findOne({ where: { role: ROLES.ADMIN } });

    await ChatParticipants.create({
      userId,
      chatId: newChat.id,
      roleId: role.id,
    });
    const chat = await Chat.findOne({
      where: { id: newChat.dataValues.id },
      include: [
        {
          model: ChatParticipants,
          as: 'chatParticipants',
          include: [{ model: Role }, { model: User }],
        },
        {
          model: Message,
          include: [
            {
              model: User,
              attributes: ['id', 'login', 'name', 'createdAt', 'updatedAt'],
            },
          ],
        },
      ],
    });
    return res.json(chat);
  }
  async getChat(req, res, next) {
    const { id } = req.params;
    const chat = await Chat.findOne({
      where: { id },
      include: [
        {
          model: ChatParticipants,
          as: 'chatParticipants',
          include: [{ model: Role }, { model: User }],
        },
        {
          model: Message,
          include: [
            {
              model: User,
              attributes: ['id', 'login', 'name', 'createdAt', 'updatedAt'],
            },
          ],
        },
      ],
    });
    return res.json(chat);
  }

  async getAllChats(req, res, next) {
    const userId = req.user.id;
```

```
  const chats = await Chat.findAll({
    include: [
      {
        model: ChatParticipants,
        as: 'chatParticipants',
        include: [
          { model: Role },
          {
            model: User,
            attributes: ['id', 'login', 'name', 'createdAt', 'updatedAt'],
          },
        ],
        // where: {
        //   userId: [userId],
        // },
      },
      {
        model: Message,
      },
    ],
  });
  return res.json(chats);
}

async addParticipant(req, res, next) {
  const { chatId, login } = req.body;
  const userId = req.user.id;

  const isUserAdminOfTheChat = await ChatParticipants.findOne({
    where: { userId, chatId },
    include: [{ model: Role }],
  });

  if (isUserAdminOfTheChat.role.role !== ROLES.ADMIN) {
    return next(ApiError.forbidden('Forbidden'));
  }

  const userToBeAdded = await User.findOne({ where: { login } });

  if (!userToBeAdded) {
    return next(ApiError.badRequest(`User doesn't exist`));
  }

  if (userToBeAdded.dataValues.id === userId) {
    return next(ApiError.badRequest(`You cant add yourself`));
  }

  console.log('userToBeAdded', userToBeAdded.dataValues.id);

  const foundParticipant = await ChatParticipants.findOne({
    where: {
      chatId,
      userId: userToBeAdded.dataValues.id,
    },
  });

  console.log('foundParticipant', foundParticipant);

  if (foundParticipant) {
    return next(ApiError.badRequest('User already joined this chat'));
  }

  const role = await Role.findOne({ where: { role: ROLES.USER } });
```

```
    const chatParticipants = await ChatParticipants.create({
      userId: userToBeAdded.dataValues.id,
      chatId,
      roleId: role.id,
    });
    return res.json(chatParticipants);
  }
  async joinChat(req, res, next) {
    const { chatId } = req.body;
    const userId = req.user.id;

    const foundParticipant = await ChatParticipants.findOne({
      where: {
        userId,
        chatId,
      },
    });

    if (foundParticipant) {
      return next(ApiError.badRequest('User already joined this chat'));
    }

    const role = await Role.findOne({ where: { role: ROLES.USER } });

    const chatParticipants = await ChatParticipants.create({
      userId,
      chatId,
      roleId: role.id,
    });
    return res.json({ chatParticipants });
  }
}

module.exports = new ChatController();

// eventController.js
const ApiError = require('../error/ApiError');
const { Event, Chat } = require('../models/models');
const { ROLES } = require('../constants/roles');

class EventController {
  async getUserEvents(req, res, next) {
    const userId = req.user.id;
    const events = await Event.findAll({
      where: { userId, isSeen: false },
      include: [{ model: Chat }],
    });
    res.json(events);
  }
  async acknowledgeEvent(req, res, next) {
    const { eventId } = req.body;
    await Event.update(
      { isSeen: true },
      {
        where: { id: eventId },
      }
    );
    res.json();
  }
}

module.exports = new EventController();

// messageController.js
```

```
const ApiError = require('../error/ApiError');
const { Chat, Message, ChatParticipants, User } = require('../models/models');
const countries = require('../constants/countries.json');
const { Op } = require('sequelize');

class MessageController {
  async createMessage(req, res, next) {
    const { message, chatId } = req.body;
    const userId = req.user.id;
    if (!message) {
      return next(ApiError.internal('Provide correct message'));
    }
    if (!chatId) {
      return next(ApiError.internal('Provide correct chatId'));
    }

    const foundChat = await Chat.findOne({ where: { id: chatId } });
    if (!foundChat) {
      return next(
        ApiError.internal(`Chat with the id ${chatId} doesn't exist`)
      );
    }

    const createdMessage = await Message.create({ message, userId, chatId });
    const chatParticipants = await ChatParticipants.findAll({
      where: {
        chatId: chatId,
        [Op.not]: {
          userId: userId,
        },
      },
    });
    chatParticipants.forEach((participant) => {
      participant.increment('unreadMessages');
      participant.save();
    });
    const messageWithRelations = await Message.findOne({
      where: { id: createdMessage.id },
      include: [
        { model: Chat, include: [{ model: ChatParticipants }] },
        { model: User },
      ],
    });

    return res.json(messageWithRelations);
  }

  async getAllMessages(req, res, next) {
    const { chatId, limit, offset } = req.body;
    if (!chatId) {
      return next(ApiError.internal('Provide correct chatId'));
    }

    const foundChat = await Chat.findOne({ where: { id: chatId } });
    if (!foundChat) {
      return next(
        ApiError.internal(`Chat with the id ${chatId} doesn't exist`)
      );
    }

    const messages = await Message.findAll({
      where: { chatId },
      offset,
      limit,
```

```
    });

    return res.json({ messages });
  }

  async editMessage(req, res, next) {
    const { messageId, messageText } = req.body;
    if (!messageId) {
      return next(ApiError.internal('Provide correct messageId'));
    }
    if (!messageText) {
      return next(ApiError.internal('Provide correct messageText'));
    }

    const foundMessage = await Message.findOne({ where: { id: messageId } });
    if (!foundMessage) {
      return next(
        ApiError.internal(`Message with the id ${chatId} doesn't exist`)
      );
    }

    foundMessage.message = messageText;
    await foundMessage.save();

    return res.json({ foundMessage });
  }
}

module.exports = new MessageController();

// roleController.js
const ApiError = require('../error/ApiError');
const { Role } = require('../models/models');

class RoleController {
  async createRole(req, res, next) {
    const { role } = req.body;
    if (!role) {
      return next(ApiError.internal('Provide correct role name'));
    }
    const foundRole = await Role.findOne({ where: { role } });

    if (foundRole) {
      return next(ApiError.internal(`Role ${role} already exists`));
    }
    const createdRole = Role.create({ role });
    return res.json({ createdRole });
  }
}

module.exports = new RoleController();

// userController.js
const ApiError = require('../error/ApiError');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const { User } = require('../models/models');

const generateJwt = (id, login) => {
  return jwt.sign({ id, login }, process.env.SECRET_KEY, { expiresIn: '24h' });
};

class UserController {
  async signUp(req, res, next) {
```

```
    const { login, password, name } = req.body;
    console.log(req.body);
    if (!login || !password) {
      return next(ApiError.badRequest('Wrong login or password'));
    }
    if (!name) {
      return next(ApiError.badRequest('Please provide correct name'));
    }
    const candidate = await User.findOne({ where: { login } });
    if (candidate) {
      return next(
        ApiError.badRequest('User with the same login already exists')
      );
    }
    const hashPassword = await bcrypt.hash(password, 5);
    const user = await User.create({ login, password: hashPassword, name });
    const token = generateJwt(user.id, user.login);
    return res.json({
      token,
      userName: user.name,
      userId: user.id,
      login: user.login,
    });
  }

  async login(req, res, next) {
    const { login, password } = req.body;
    const user = await User.findOne({ where: { login } });
    if (!user) {
      return next(ApiError.internal('User not found'));
    }
    let comparePassword = bcrypt.compareSync(password, user.password);
    if (!comparePassword) {
      return next(ApiError.internal('Wrong password'));
    }
    const token = generateJwt(user.id, user.login);
    return res.json({
      token,
      userName: user.name,
      userId: user.id,
      login: user.login,
    });
  }

  async check(req, res, next) {
    const token = generateJwt(req.user.id, req.user.login);
    return res.json({ token });
  }
}

module.exports = new UserController();

// roles.js
const ROLES = {
  ADMIN: 'admin',
  USER: "user"
};
module.exports = { ROLES };


// main.tsx головний файл клієнтської частини додатку
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './components/App';
```

```
import './global.css';

ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);

// theme.ts файл стилей додатку
import { createTheme } from '@mui/material';

export const authPageBackgroundStyles = {
  width: '100%',
  height: '100%',
  background:
    'linear-gradient(90deg, rgba(107,74,241,1) 0%, rgba(205,11,132,1) 100%)',
};

export const styleInputDefault = {
  fontSize: '16px',
  height: 'auto',
  lineHeight: 1,
  background: "#DCEBF2",
  border: 'none',
  outline: 'none',

  '&.MuiInputBase-inputSizeSmall': {
    fontSize: '16px',
  },
};
export const inputContainerDefaultStyle = {
  borderRadius: 4,
  overflow: "hidden"
};

export const theme = createTheme({
  palette: {
    primary: {
      main: '#6E48ED',
    },
    secondary: {
      main: '#D20980',
    },
  },
  components: {
    MuiInputBase: {
      defaultProps: {
        sx: {
          ...inputContainerDefaultStyle,
          input: {
            ...styleInputDefault,
          },
        },
      },
    },
  },
});

// router.tsx
import { RouterProvider, createBrowserRouter } from 'react-router-dom';
import RedirectIfNotAuthenticated from './RedirectIfNotAuthenticated';
import ChatListPage from '../pages/ChatListPage';
import LoginPage from '../pages/LoginPage';
import RedirectIfAuthenticated from './RedirectIfAuthenticated';
```

```
import ChatPage from '../pages/ChatPage';
import SignUpPage from '../pages/SignUpPage';

export default function Router() {
  const router = createBrowserRouter([
    {
      path: '/',
      element: (
        <RedirectIfNotAuthenticated>
          <ChatListPage />
        </RedirectIfNotAuthenticated>
      ),
      children: [
        {
          path: 'chats/:chatId',
          element: <ChatPage />,
        },
      ],
      // ErrorBoundary: NotFoundPage,
    },
    {
      path: '/login',
      element: (
        <RedirectIfAuthenticated>
          <LoginPage />
        </RedirectIfAuthenticated>
      ),
    },
    {
      path: '/signup',
      element: (
        <RedirectIfAuthenticated>
          <SignUpPage />
        </RedirectIfAuthenticated>
      ),
    },
  ]);
  return <RouterProvider router={router} />;
}

// RedirectIfAuthenticated.tsx
import { FC, PropsWithChildren } from 'react';
import { Navigate } from 'react-router-dom';

const RedirectIfAuthenticated: FC<PropsWithChildren> = ({ children }) => {
  const redirectTo = '/';
  const isLoggedIn =
    JSON.parse(localStorage.getItem('authData') || '{}') || false;

  return isLoggedIn.token ? <Navigate to={redirectTo} replace /> : children;
};

export default RedirectIfAuthenticated;

// RedirectIfNotAuthenticated.tsx
import { FC, PropsWithChildren } from 'react';
import { Navigate } from 'react-router-dom';

const RedirectIfNotAuthenticated: FC<PropsWithChildren> = ({ children }) => {
  const redirectTo = '/login';
  const isLoggedIn =
    JSON.parse(localStorage.getItem('authData') || '{}') || false;

  return !isLoggedIn.token ? <Navigate to={redirectTo} replace /> : children;
```

```
};

export default RedirectIfNotAuthenticated;

// ChatListPage.tsx
import { Box, Stack, Typography } from '@mui/material';
import { useEffect, useState } from 'react';
import { useNavigate, useOutlet } from 'react-router-dom';
import { acknowledgeEvent, getAllChats, getUserEvents } from '../../api';
import ChatList from '../../components/ChatList';
import UserAddedToChatModal from '../../components/UserAddedToChatModal';
import { Chat } from '../ChatPage';

function ChatPlaceholder() {
  return (
    <Stack
      justifyContent={'center'}
      alignItems={'center'}
      sx={{ width: '100%', height: '100%' }}>
      <Typography sx={{ p: 1, background: '#DCEBF2', borderRadius: 2 }}>
        Please, select a chat to start messaging
      </Typography>
    </Stack>
  );
}

export interface Event {
  id: number;
  createdAt: string;
  updatedAt: string;
  chatId: string;
  chat: Chat;
  content: string | null;
}

export default function ChatListPage() {
  const outlet = useOutlet();

  const [chatList, setChatList] = useState<Chat[]>([]);
  const [chatsWithUser, setChatsWithUser] = useState<Chat[]>([]);
  const [filteredChats, setFilteredChats] = useState<Chat[]>([]);
  const [userEvents, setUserEvents] = useState<Event[]>([]);
  const [isOpen, setIsOpen] = useState(false);
  const authData = JSON.parse(localStorage.getItem('authData') || '{}');
  const navigate = useNavigate();

  function handleSubmit(chatId: number, eventId: number) {
    navigate(`chats/${chatId}`);
    acknowledgeEvent({ eventId });
    setIsOpen(false);
  }

  function searchChat(searchText: string) {
    if (searchText === '') {
      setFilteredChats(chatsWithUser);
    } else {
      setFilteredChats(
        chatList.filter((chat) => chat.name.includes(searchText))
      );
    }
  }

  useEffect(() => {
    setFilteredChats(chatsWithUser);
```

```
  }, [chatsWithUser]);

  useEffect(() => {
    console.log(filteredChats);
  }, [filteredChats]);

  useEffect(() => {
    getAllChats().then((res: Chat[]) => {
      const userChats = res.filter((chat) =>
        chat.chatParticipants.find((user) => user.userId === authData?.userId)
      );
      setChatList(res);
      setChatsWithUser(userChats);
      setFilteredChats(userChats);
    });
    getUserEvents().then((res) => {
      if (res?.length > 0) {
        setIsOpen(true);
      }
      setUserEvents(res);
    });
  }, []);

  return (
    <Box sx={{ display: 'flex' }}>
      <ChatList
        chatList={filteredChats}
        setChatList={setChatsWithUser}
        searchChat={searchChat}
      />
      <Box
        component='main'
        sx={{
          backgroundColor: (theme) =>
            theme.palette.mode === 'light'
              ? theme.palette.grey[100]
              : theme.palette.grey[900],
          flexGrow: 1,
          height: '100vh',
          overflow: 'auto',
        }}>
        {outlet || <ChatPlaceholder />}
      </Box>
      {userEvents.length > 0 && (
        <UserAddedToChatModal
          isOpen={isOpen}
          event={userEvents[0]}
          onClose={() => setIsOpen(false)}
          onSubmit={handleSubmit}
        />
      )}
    </Box>
  );
}

// ChatPage.tsx
import { CircularProgress, Divider, Stack, Typography } from '@mui/material';
import { useEffect, useState } from 'react';
import { useNavigate, useParams } from 'react-router-dom';
import { addParticipantToTheChat, getChatbyId, joinChat } from '../../api';
import ChatForm from '../../components/ChatForm';
import ChatInfoModal from '../../components/ChatInfoModal';

export interface Chat {
```

```
  id: number;
  name: string;
  image: string | null;
  createdAt: string;
  updatedAt: string;
  chatParticipants: ChatParticipants[];
  messages: Message[];
}
export interface Message {
  id: number;
  message: string;
  createdAt: string;
  updatedAt: string;
  userId: number;
  chatId: number;
  user: User;
}
export interface ChatParticipants {
  id: number;
  unreadMessages: number;
  createdAt: string;
  updatedAt: string;
  userId: number;
  chatId: number;
  roleId: number;
  role: Role;
  user: User;
}
export interface Role {
  id: number;
  createdAt: string;
  updatedAt: string;
  role: string;
}
export interface User {
  id: number;
  login: string;
  name: string;
  createdAt: string;
  updatedAt: string;
}

export default function ChatPage() {
  const { chatId } = useParams();
  const navigate = useNavigate();
  const [chatData, setChatData] = useState<Chat | null>(null);
  const [isLoading, setIsLoading] = useState(false);
  const [messages, setMessages] = useState<Message[]>([]);
  const [isOpen, setIsOpen] = useState(false);
  const authData = JSON.parse(localStorage.getItem('authData') || '{}');
  const participant = chatData?.chatParticipants.find(
    (chat) => (chat.userId === authData?.userId)
  );

  useEffect(() => {
    if (chatId && +chatId) {
      console.log('asd');
      setIsLoading(true);
      getChatbyId({ id: chatId })
        .then((res) => {
          setChatData(res);
          setMessages(res.messages);
        })
        .finally(() => setIsLoading(false));
```

```
  } else {
    console.log('asd');
    navigate('/');
  }
}, [chatId]);

function handleJoinChat(chatId: string) {
  joinChat({ chatId }).then((res: ChatParticipants) => {
    if (chatData) {
      setChatData({
        ...chatData,
        chatParticipants: [...chatData.chatParticipants, res],
      });
    }
  });
}

function handleSubmit(newParticipant: ChatParticipants) {
  if (chatData) {
    setChatData({
      ...chatData,
      chatParticipants: [...chatData.chatParticipants, newParticipant],
    });
    setIsOpen(false);
  }
}

return (
  <Stack
    sx={{ height: '100%', width: '100%', background: '#fff' }}
    divider={<Divider />}>
    <Stack
      height={56}
      p={2}
      onClick={() => setIsOpen(true)}
      sx={{ cursor: 'pointer' }}>
      <Typography
        variant='h6'
        sx={{ color: (theme) => theme.palette.secondary.main }}>
        {chatData?.name}
      </Typography>
    </Stack>
    <Stack flexGrow={1}>
      {/* {isLoading ? (
        <Stack
          height={'100%'}
          alignItems={'center'}
          justifyContent={'center'}>
          <CircularProgress />
        </Stack>
      ) : ( */}
      <ChatForm
        messages={messages}
        chatId={chatId!}
        setMessages={setMessages}
        isParticipant={!!participant}
        onJoinChat={handleJoinChat}
      />
      {/* )} */}
    </Stack>
    {chatData && participant && (
      <ChatInfoModal
        isOpen={isOpen}
        chat={chatData}
```

```
          onClose={() => setIsOpen(false)}
          participant={participant}
          onSubmit={handleSubmit}
        />
      )}
    </Stack>
  );
}

// LoginPage.tsx
import { FormEvent, useEffect, useState } from 'react';
import Avatar from '@mui/material/Avatar';
import Button from '@mui/material/Button';
import TextField from '@mui/material/TextField';
import Box from '@mui/material/Box';
import LockOutlinedIcon from '@mui/icons-material/LockOutlined';
import Typography from '@mui/material/Typography';
import { login } from '../../api';
import { useNavigate } from 'react-router-dom';
import { CircularProgress, Stack } from '@mui/material';
import { authPageBackgroundStyles } from '../../theme/theme';

export default function LoginPage() {
  const navigate = useNavigate();
  const [userLogin, setUserLogin] = useState('');
  const [userPassword, setUserPassword] = useState('');
  const [errorMessage, setErrorMessage] = useState('');
  const [isLoading, setIsLoading] = useState(false);

  const handleSubmit = (event: FormEvent<HTMLFormElement>) => {
    event.preventDefault();
    setIsLoading(true);
    setErrorMessage('');
    login({ login: userLogin, password: userPassword })
      .then((res) => {
        if (res.data.token) {
          localStorage.setItem('authData', JSON.stringify(res?.data));
          setIsLoading(false);
          navigate('/');
        }
      })
      .catch((error) => {
        setIsLoading(false);
        setErrorMessage(
          error?.response?.data?.message || 'Something went wrong, try again'
        );
      });
  };

  useEffect(() => {
    setErrorMessage('');
  }, [userLogin, userPassword]);

  function handleLoginChange(e: any) {
    setUserLogin(e.target.value);
  }
  function handlePasswrodChange(e: any) {
    setUserPassword(e.target.value);
  }

  return (
    <Stack
      component='main'
      alignItems={'center'}
```

```
justifyContent={'center'}
sx={authPageBackgroundStyles}>
<Box
  sx={{
    maxWidth: '640px',
    width: '100%',
    background: '#fff',
    p: 4,
    borderRadius: 4,
    display: 'flex',
    flexDirection: 'column',
    justifyContent: 'center',
    alignItems: 'center',
  }}>
  <Avatar sx={{ m: 1, bgcolor: 'secondary.main' }}>
    <LockOutlinedIcon />
  </Avatar>
  <Typography component='h1' variant='h5'>
    Log in
  </Typography>

  <Box
    component='form'
    onSubmit={handleSubmit}
    sx={{ mt: 1, maxWidth: '390px' }}>
    <TextField
      value={userLogin}
      onChange={handleLoginChange}
      error={!!errorMessage}
      disabled={isLoading}
      margin='normal'
      required
      fullWidth
      id='login'
      label='Login'
      name='login'
      autoComplete='login'
      autoFocus
    />
    <TextField
      value={userPassword}
      onChange={handlePasswrodChange}
      error={!!errorMessage}
      disabled={isLoading}
      margin='normal'
      required
      fullWidth
      name='password'
      label='Password'
      type='password'
      id='password'
      autoComplete='current-password'
    />
    {errorMessage && (
      <Typography mt={2} textAlign='center' color='red' variant='body2'>
        {errorMessage}
      </Typography>
    )}
    {isLoading && (
      <Stack alignItems={'center'}>
        <CircularProgress />
      </Stack>
    )}
    <Typography variant='body2'>
```

```
            Doesn't have an acoount?{' '}
            <Typography
              component='span'
              sx={{
                color: (theme) => theme.palette.primary.main,
                cursor: 'pointer',
              }}
              onClick={() => {
                navigate('/signup');
              }}
              variant='body2'>
              Sign up
            </Typography>
          </Typography>
          <Button
            type='submit'
            fullWidth
            variant='contained'
            disabled={isLoading}
            sx={{ mt: 3, mb: 2 }}>
            Log In
          </Button>
        </Box>
      </Box>
    </Stack>
  );
}

// SignUpPage.tsx
import { FormEvent, useEffect, useState } from 'react';
import Avatar from '@mui/material/Avatar';
import Button from '@mui/material/Button';
import TextField from '@mui/material/TextField';
import Box from '@mui/material/Box';
import LockOutlinedIcon from '@mui/icons-material/LockOutlined';
import Typography from '@mui/material/Typography';
import { signUp } from '../../api';
import { useNavigate } from 'react-router-dom';
import { CircularProgress, Stack } from '@mui/material';
import { authPageBackgroundStyles } from '../../theme/theme';

export default function SignUpPage() {
  const navigate = useNavigate();
  const [userLogin, setUserLogin] = useState('');
  const [userPassword, setUserPassword] = useState('');
  const [userName, setUserName] = useState('');
  const [errorMessage, setErrorMessage] = useState('');
  const [isLoading, setIsLoading] = useState(false);

  const handleSubmit = (event: FormEvent<HTMLFormElement>) => {
    event.preventDefault();
    setIsLoading(true);
    setErrorMessage('');
    signUp({ login: userLogin, password: userPassword, name: userName })
      .then((res) => {
        if (res.data.token) {
          localStorage.setItem('authData', JSON.stringify(res?.data));
          setIsLoading(false);
          navigate('/');
        }
      })
      .catch((error) => {
        setIsLoading(false);
        setErrorMessage(
```

```
        error?.response?.data?.message || 'Something went wrong, try again'
      );
    });
  };

  useEffect(() => {
    setErrorMessage('');
  }, [userLogin, userPassword]);

  function handleLoginChange(e: any) {
    setUserLogin(e.target.value);
  }
  function handleUserNameChange(e: any) {
    setUserName(e.target.value);
  }
  function handlePasswrodChange(e: any) {
    setUserPassword(e.target.value);
  }

  return (
    <Stack
      component='main'
      alignItems={'center'}
      justifyContent={'center'}
      sx={authPageBackgroundStyles}>
      <Box
        sx={{
          maxWidth: '640px',
          width: '100%',
          background: '#fff',
          p: 4,
          borderRadius: 4,
          display: 'flex',
          flexDirection: 'column',
          justifyContent: 'center',
          alignItems: 'center',
        }}>
        <Avatar sx={{ m: 1, bgcolor: 'secondary.main' }}>
          <LockOutlinedIcon />
        </Avatar>
        <Typography component='h1' variant='h5'>
          Sign up
        </Typography>

        <Box
          component='form'
          onSubmit={handleSubmit}
          sx={{ mt: 1, maxWidth: '390px' }}>
          <TextField
            value={userName}
            onChange={handleUserNameChange}
            error={!!errorMessage}
            disabled={isLoading}
            margin='normal'
            required
            fullWidth
            id='userName'
            label='User name'
            placeholder='Foo Bar'
            name='userName'
            autoComplete='userName'
            autoFocus
          />
          <TextField
```

```
          value={userLogin}
          onChange={handleLoginChange}
          error={!!errorMessage}
          disabled={isLoading}
          margin='normal'
          required
          fullWidth
          id='login'
          label='Login'
          name='login'
          autoComplete='login'
          autoFocus
        />
        <TextField
          value={userPassword}
          onChange={handlePasswrodChange}
          error={!!errorMessage}
          disabled={isLoading}
          margin='normal'
          required
          fullWidth
          name='password'
          label='Password'
          type='password'
          id='password'
          autoComplete='current-password'
        />
        <Typography variant='body2'>
          Already have an acoount?{' '}
          <Typography
            component='span'
            sx={{
              color: (theme) => theme.palette.primary.main,
              cursor: 'pointer',
            }}
            onClick={() => {
              navigate('/login');
            }}
            variant='body2'>
            Log in
          </Typography>
        </Typography>
        {errorMessage && (
          <Typography mt={2} textAlign='center' color='red' variant='body2'>
            {errorMessage}
          </Typography>
        )}
        {isLoading && (
          <Stack alignItems={'center'}>
            <CircularProgress />
          </Stack>
        )}
        <Button
          type='submit'
          fullWidth
          variant='contained'
          disabled={isLoading}
          sx={{ mt: 3, mb: 2 }}>
          Sign up
        </Button>
      </Box>
    </Box>
  </Stack>
);
```

```
}

// SocketContext.tsx
import React, {
  PropsWithChildren,
  createContext,
  useContext,
  useEffect,
  useState,
} from 'react';
import { io, Socket } from 'socket.io-client';

interface SocketContextProps {
  socket: Socket;
}

const SocketContext = createContext<SocketContextProps | undefined>(undefined);

const SocketProvider: React.FC<PropsWithChildren> = ({ children }) => {
  const [socket] = useState<Socket>(io(import.meta.env.VITE_APP_BASE_URI));
  const authData = JSON.parse(localStorage.getItem('authData') || '{}');
  useEffect(() => {
    socket.emit('setup', authData?.userId);
  }, []);

  return (
    <SocketContext.Provider value={{ socket }}>
      {children}
    </SocketContext.Provider>
  );
};

const useSocket = () => {
  const context = useContext(SocketContext);
  if (!context) {
    throw new Error('useSocket must be used within a SocketProvider');
  }
  return context;
};

export { SocketProvider, useSocket };

// App.tsx
import '@fontsource/roboto/300.css';
import '@fontsource/roboto/400.css';
import '@fontsource/roboto/500.css';
import '@fontsource/roboto/700.css';
import Router from '../../router';
import { ThemeProvider } from '@mui/material';
import { theme } from '../../theme/theme';
import { SocketProvider } from '../../context/SocketContext';

function App() {
  return (
    <SocketProvider>
      <ThemeProvider theme={theme}>
        <Router />
      </ThemeProvider>
    </SocketProvider>
  );
}

export default App;
```

```
// ChatCreationModal.tsx
import { Button, Dialog, Stack, TextField, Typography } from '@mui/material';
import { Chat } from '../../pages/ChatPage';
import { useState } from 'react';
import { createChat } from '../../api';

interface ChatCreationModalProps {
  isOpen: boolean;
  onClose: () => void;
  onSubmit: (chat: Chat) => void;
}

export default function ChatCreationModal({
  isOpen,
  onClose,
  onSubmit,
}: ChatCreationModalProps) {
  const [chatName, setChatName] = useState('');
  const [chatPicture, setChatPicture] = useState('');
  const [isLoading, setIsLoading] = useState(false);
  const [errorMessage, setErrorMessage] = useState('');

  function handleSubmit(e: any) {
    setIsLoading(true);
    e.preventDefault();
    createChat({ chatName, chatPicture })
      .then((res: Chat) => {
        setChatName('');
        setChatPicture('');
        onSubmit(res);
      })
      .catch((error) => {
        console.error(error);
        setErrorMessage(error.message);
      })
      .finally(() => {
        setIsLoading(false);
      });
  }

  return (
    <Dialog
      open={isOpen}
      PaperProps={{
        sx: {
          borderRadius: 4,
          p: 5,
        },
      }}
      fullWidth={true}
      maxWidth='xs'
      onClose={onClose}>
      <Stack component={'form'} spacing={2} onSubmit={handleSubmit}>
        <Typography
          variant='h6'
          sx={{ color: (theme) => theme.palette.secondary.main }}>
          Create new chat
        </Typography>
        <TextField
          value={chatName}
          onChange={({ target }) => setChatName(target.value)}
          error={!!errorMessage}
          disabled={isLoading}
          margin='normal'
```

```
          required
          fullWidth
          id='chatName'
          label='Chat name'
          name='chatName'
          autoComplete='chatName'
          autoFocus
        />
        <TextField
          value={chatPicture}
          onChange={({ target }) => setChatPicture(target.value)}
          error={!!errorMessage}
          disabled={isLoading}
          margin='normal'
          fullWidth
          id='chatPicture'
          label='Link to profile picture'
          name='chatPicture'
          autoComplete='chatPicture'
        />
        <Button type='submit'>Create</Button>
      </Stack>
    </Dialog>
  );
}

// ChatForm.tsx
import { Button, Stack, TextField, Typography } from '@mui/material';
import { useEffect, useRef, useState } from 'react';
import { Message } from '../../pages/ChatPage';
import dayjs from 'dayjs';
import { createMessage } from '../../api';
import { useSocket } from '../../context/SocketContext';

interface ChatFormProps {
  messages: Message[];
  setMessages: any;
  chatId: string;
  isParticipant: boolean;
  onJoinChat: (chatId: string) => void;
}

export default function ChatForm({
  messages,
  chatId,
  setMessages,
  isParticipant,
  onJoinChat,
}: ChatFormProps) {
  const [enteredMessage, setEnteredMessage] = useState('');
  const [socketConnected, setSocketConnected] = useState(false);
  const [typing, setTyping] = useState(false);
  const [istyping, setIsTyping] = useState(false);
  const authData = JSON.parse(localStorage.getItem('authData') || '{}');
  const { socket } = useSocket();

  const chatRef = useRef();

  useEffect(() => {
    socket.on('connected', () => setSocketConnected(true));
    socket.on('typing', () => setIsTyping(true));
    socket.on('stop typing', () => setIsTyping(false));
    return () => {
      socket.off('connected');
```

```
    socket.off('typing');
    socket.off('stop typing');
  };
}, []);

useEffect(() => {
  socket.emit('join chat', chatId);
  socket.emit('read messages', { chatId, userId: authData?.userId });

  socket.on('message recieved', (newMessageRecieved) => {
    if (chatId == newMessageRecieved.chat.id) {
      setMessages((prevState: Message[]) => [
        ...prevState,
        newMessageRecieved,
      ]);
    }
  });

  return () => {
    socket.emit('left chat', chatId);
    socket.off('message recieved');
  };
}, [chatId]);

useEffect(() => {
  //@ts-expect-error asd
  chatRef?.current?.scrollIntoView({ behavior: 'smooth' });
}, [messages]);

const sendMessage = async (event: any) => {
  if (enteredMessage) {
    // socket.emit('stop typing', chatId);
    try {
      setEnteredMessage('');
      const createdMessage = await createMessage({
        message: enteredMessage,
        chatId,
      });
      console.log(createdMessage);
      socket.emit('new message', createdMessage);
      setMessages([...messages, createdMessage]);
    } catch (error) {
      console.log(error);
      // toast({
      //   title: "Error Occured!",
      //   description: "Failed to send the Message",
      //   status: "error",
      //   duration: 5000,
      //   isClosable: true,
      //   position: "bottom",
      // });
    }
  }
};

function handleJoinChat() {
  onJoinChat(chatId);
}

return (
  <Stack
    px={4}
    height='100%'
    position={'relative'}
```

```
justifyContent={'flex-end'}>
{messages.length > 0 ? (
  <Stack
    sx={{
      position: 'absolute',
      maxHeight: 'calc(100% - 64px)',
      left: '32px',
      right: '32px',
      bottom: 0,
      top: 0,
      overflow: 'scroll',
      // justifyContent: 'flex-end',
    }}
    py={2}
    spacing={2}>
    {messages.map((message, index, array) => {
      const isAuthtor = message.userId === authData?.userId;
      const isSameAuthorOfThePreviousMessage =
        array[index - 1]?.userId === message.userId;
      const currentMessageDay = dayjs(message.createdAt).startOf('day');
      const previousMessageDay = dayjs(
        array[index - 1]?.createdAt
      ).startOf('day');
      const daysDifference = currentMessageDay.diff(
        previousMessageDay,
        'day'
      );
      return (
        <Stack key={message.id}>
          {daysDifference !== 0 && (
            <Typography
              sx={{
                fontSize: 12,
                alignSelf: 'center',
                py: 0.5,
                px: 1.5,
                mb: 2,
                background: (theme) => theme.palette.secondary.main,
                color: 'white',
                borderRadius: 8,
              }}>
              {dayjs(message.createdAt).format('MMMM D')}
            </Typography>
          )}
          <Stack
            sx={{
              position: 'relative',
              maxWidth: '600px',
              alignSelf: isAuthtor ? 'end' : 'start',
              p: 2,
              pr: 6,
              mt:
                isSameAuthorOfThePreviousMessage && daysDifference < 1
                  ? -1.5
                  : 0,

              borderRadius: 5,
              ...(isAuthtor
                ? { borderBottomRightRadius: 0 }
                : { borderBottomLeftRadius: 0 }),
              background: isAuthtor
                ? (theme) => theme.palette.primary.main
                : '#DCEBF2',
              color: isAuthtor ? '#fff' : '#000',
```

```
              }}>
              {!isAuthtor && (
                <Typography
                  sx={{
                    color: (theme) => theme.palette.primary.main,
                  }}>
                  {message.user?.name}
                </Typography>
              )}

              <Typography>{message?.message}</Typography>
              <Typography
                sx={{
                  fontSize: '10px',
                  position: 'absolute',
                  bottom: 4,
                  right: 16,
                }}>
                {dayjs(message?.createdAt).format('H:mm')}
              </Typography>
            </Stack>
          </Stack>
        );
      })}
      <Stack
        mt={0}
        ref={(el) => {
          //@ts-expect-error asd
          chatRef.current = el;
        }}
      />
    </Stack>
  ) : (
    <Stack
      justifyContent={'center'}
      alignItems={'center'}
      sx={{ width: '100%', height: '100%' }}>
      <Typography
        sx={{ py: 0.5, px: 1.5, background: '#DCEBF2', borderRadius: 4 }}>
        {isParticipant
          ? 'Write your first message'
          : 'Join chat to write messages'}
      </Typography>
    </Stack>
  )}

  <Stack pb={2} position={'relative'} justifyContent={'center'}>
    {isParticipant ? (
      <>
        {' '}
        <TextField
          sx={{ my: 0 }}
          value={enteredMessage}
          onChange={({ target }) => setEnteredMessage(target.value)}
          // error={!!errorMessage}
          // disabled={isLoading}
          margin='normal'
          required
          fullWidth
          id='enteredMessage'
          name='enteredMessage'
          autoComplete='enteredMessage'
          placeholder='Write a message...'
          autoFocus
```

```
          />
          <Button
            variant='text'
            onClick={sendMessage}
            sx={{
              color: (theme) => theme.palette.primary.main,
              position: 'absolute',
              right: 16,
              '&:hover': {
                background: 'transparent',
              },
            }}>
            Send
          </Button>
        </>
      ) : (
        <Button
          onClick={handleJoinChat}
          sx={{ borderRadius: 4 }}
          color='secondary'
          variant='outlined'>
          Join chat
        </Button>
      )}
    </Stack>
  </Stack>
);
}

// ChatInfoModal.tsx
import {
  Avatar,
  Button,
  Dialog,
  Divider,
  Stack,
  TextField,
  Typography,
} from '@mui/material';
import { useState } from 'react';
import { Chat, ChatParticipants } from '../../pages/ChatPage';
import { addParticipantToTheChat } from '../../api';

interface ChatInfoModalProps {
  isOpen: boolean;
  chat: Chat;
  onClose: () => void;
  onSubmit: (arg: ChatParticipants) => void;
  participant: ChatParticipants;
}

export default function ChatInfoModal({
  isOpen,
  chat,
  participant,
  onClose,
  onSubmit,
}: ChatInfoModalProps) {
  console.log('participant', participant);
  const [enteredLogin, setEnteredLogin] = useState('');
  const [errorMessage, setErrorMessage] = useState('');
  function handleSubmit(e: any) {
    e.preventDefault();
    if (enteredLogin) {
```

```
    addParticipantToTheChat({ chatId: chat?.id, login: enteredLogin })
      .then((res: ChatParticipants) => {
        onSubmit(res);
      })
      .catch((res) => {
        setErrorMessage(res.message);
      });

    // setEnteredLogin('');
  }
}
function handleClose() {
  onClose();
  setEnteredLogin('');
  setErrorMessage('');
}
return (
  <Dialog
    open={isOpen}
    PaperProps={{
      sx: {
        borderRadius: 4,
        p: 2,
      },
    }}
    fullWidth={true}
    maxWidth='xs'
    onClose={handleClose}>
    <Stack
      component={'form'}
      spacing={2}
      onSubmit={handleSubmit}
      divider={<Divider />}>
      <Typography
        variant='h6'
        sx={{ color: (theme) => theme.palette.secondary.main }}>
        Chat info
      </Typography>
      <Stack direction={'row'} alignItems={'center'} spacing={3}>
        <Avatar
          sx={{ width: 96, height: 96 }}
          src={chat?.image || ''}
          alt={chat?.name}
        />
        <Stack>
          <Typography variant='h6'>{chat?.name}</Typography>
          <Typography>{chat?.chatParticipants.length} members</Typography>
        </Stack>
      </Stack>
      {participant.role.role.toLowerCase() === 'admin' && (
        <>
          <Typography textAlign={'center'} color='red'>
            {errorMessage}
          </Typography>
          <Stack direction={'row'} alignItems={'center'} spacing={2}>
            <Stack flexGrow={1}>
              <TextField
                sx={{ my: 0 }}
                value={enteredLogin}
                onChange={({ target }) => {
                  setErrorMessage('');
                  setEnteredLogin(target.value);
                }}
                error={!!errorMessage}
```

```
                  // disabled={isLoading}
                  margin='normal'
                  required
                  fullWidth
                  id='enteredLogin'
                  label='User login'
                  name='enteredLogin'
                  autoComplete='enteredLogin'
                  autoFocus
                />
              </Stack>
              <Stack>
                <Button
                  size='large'
                  variant='outlined'
                  color='secondary'
                  sx={{ borderRadius: 4, height: 52 }}
                  type='submit'>
                  Add user
                </Button>
              </Stack>
            </Stack>
          </>
        )}
      </Stack>
    </Dialog>
  );
}

// ChatList.tsx
import {
  List,
  ListItemButton,
  ListItemIcon,
  ListItemText,
  Stack,
  TextField,
  Typography,
  styled,
} from '@mui/material';
import MuiDrawer from '@mui/material/Drawer';
import { useEffect, useState } from 'react';
import LogoutIcon from '@mui/icons-material/Logout';
import ChatIcon from '@mui/icons-material/Chat';
import { useNavigate } from 'react-router-dom';
import { Chat } from '../../pages/ChatPage';
import ChatListItem from '../ChatListItem';
import ChatCreationModal from '../ChatCreationModal';
import { useDebounce } from 'use-debounce';

const Drawer = styled(MuiDrawer, {
  shouldForwardProp: (prop) => prop !== 'open',
})(({ theme }) => ({
  '& .MuiDrawer-paper': {
    position: 'relative',
    whiteSpace: 'nowrap',
    width: drawerWidth,
    transition: theme.transitions.create('width', {
      easing: theme.transitions.easing.sharp,
      duration: theme.transitions.duration.enteringScreen,
    }),
    boxSizing: 'border-box',
  },
}));
```

```
const drawerWidth: number = 240;

interface ChatListProps {
  chatList: Chat[];
  setChatList: (arg: any) => void;
  searchChat: (arg: any) => void;
}

export default function ChatList({
  chatList,
  setChatList,
  searchChat,
}: ChatListProps) {
  const [searchText, setSearchText] = useState('');
  const [debouncedSearchText] = useDebounce(searchText, 500);
  const [isOpen, setIsOpen] = useState(false);
  const navigate = useNavigate();

  function logout() {
    localStorage.clear();
    navigate('/login');
  }
  useEffect(() => {
    searchChat(searchText);
  }, [debouncedSearchText]);

  function handleSubmit(newChat: Chat) {
    navigate(`/chats/${newChat.id}`);
    setChatList((prevList: Chat[]) => [...prevList, newChat]);
    setIsOpen(false);
  }

  return (
    <Drawer variant='permanent' open={true} sx={{ maxHeight: '100%' }}>
      <Stack p={2} pb={0}>
        <Typography
          sx={{
            color: (theme) => theme.palette.secondary.main,
            fontWeight: 'bold',
          }}
          variant='h5'>
          Chats
        </Typography>
        <TextField
          value={searchText}
          onChange={({ target }) => setSearchText(target.value)}
          // error={!!errorMessage}
          // disabled={isLoading}
          margin='normal'
          required
          fullWidth
          id='searchText'
          placeholder='Search'
          name='searchText'
          autoComplete='searchText'
          autoFocus
        />
      </Stack>
      <Stack
        flexGrow={1}
        justifyContent={'end'}
        sx={{ height: '100%', position: 'relative' }}>
        <List
```

```
          sx={{
            position: 'absolute',
            maxHeight: 'calc(100% - 94px)',
            overflow: 'scroll',
            top: 0,
            right: 0,
            left: 0,
            bottom: 0,
          }}>
          <Stack>
            {chatList.map((chat) => (
              <ChatListItem key={chat.id} chat={chat} />
            ))}
          </Stack>
        </List>
        <Stack>
          <ListItemButton onClick={() => setIsOpen(true)}>
            <ListItemIcon>
              <ChatIcon />
            </ListItemIcon>
            <ListItemText primary='Create new chat' />
          </ListItemButton>
          <ListItemButton onClick={logout}>
            <ListItemIcon>
              <LogoutIcon />
            </ListItemIcon>
            <ListItemText primary='Log out' />
          </ListItemButton>
        </Stack>
      </Stack>
      <ChatCreationModal
        isOpen={isOpen}
        onClose={() => setIsOpen(false)}
        onSubmit={handleSubmit}
      />
    </Drawer>
  );
}

// ChatListItem.tsx
import { Link } from 'react-router-dom';
import { Chat } from '../../pages/ChatPage';
import {
  Avatar,
  Box,
  ListItemButton,
  ListItemIcon,
  Stack,
  Typography,
} from '@mui/material';
import dayjs from 'dayjs';
import { useEffect, useState } from 'react';
import { useSocket } from '../../context/SocketContext';

interface ChatListItemProps {
  chat: Chat;
}

export default function ChatListItem({ chat }: ChatListItemProps) {
  const [chatState, setChatState] = useState(chat);
  const { socket } = useSocket();

  const authData = JSON.parse(localStorage.getItem('authData') || '{}');
```

```
  const participant = chatState.chatParticipants.find(
    (participant) => participant?.userId === authData.userId
  );

  const unreadMessages = participant?.unreadMessages;

  const isSelected = location.pathname === `/chats/${chat.id}`;

  useEffect(() => {
    socket.on('message was read', (newChatParticipant) => {
      console.log(newChatParticipant);
      setChatState((prevState) => {
        const filteredChatParticipants = prevState.chatParticipants.filter(
          (participant) => {
            return participant?.id !== newChatParticipant?.id;
          }
        );

        return {
          ...prevState,
          chatParticipants: [...filteredChatParticipants, newChatParticipant],
        };
      });
    });
    // socket.on('new messages unreaded', (newChatParticipant) => {
    //   if (chatState.id == newChatParticipant.chatId) {
    //     setChatState((prevState) => {
    //       const filteredChatParticipants = prevState.chatParticipants.filter(
    //         (participant) => {
    //           return participant.id !== newChatParticipant.id;
    //         }
    //       );
    //       return {
    //         ...prevState,
    //         chatParticipants: [...filteredChatParticipants,
newChatParticipant],
    //       };
    //     });
    //   }
    // });
    return () => {
      socket.off('message was read');
    };
  }, []);

  const time = chatState.messages[chatState.messages.length - 1]?.createdAt;

  return (
    <Link key={chatState.id} to={`/chats/${chatState.id}`}>
      <ListItemButton selected={isSelected}>
        <ListItemIcon>
          <Avatar alt={chatState.name} src={chatState.image || ''} />
        </ListItemIcon>
        <Stack
          sx={{ position: 'relative', width: '100%', overflow: 'hidden' }}
          justifyContent={'center'}>
          <Stack
            direction='row'
            justifyContent={'space-between'}
            alignItems={'center'}>
            <Typography>{chatState.name}</Typography>
            <Typography sx={{ fontSize: '10px' }}>
              {time && dayjs(time).format('h:mm A')}
            </Typography>
```

```
          </Stack>
          <Stack
            direction='row'
            justifyContent={'space-between'}
            alignItems={'center'}>
            <Typography sx={{ fontSize: '12px', overflow: 'hidden', mr: 2 }}>
              {chatState.messages[chatState.messages.length - 1]?.message ||
                'No messages so far'}
            </Typography>
            {unreadMessages && unreadMessages > 0 ? (
              <Typography
                sx={{
                  minWidth: 20,
                  minHeight: 20,
                  display: 'flex',
                  alignItems: 'center',
                  justifyContent: 'center',

                  borderRadius: '50%',
                  fontSize: '10px',
                  background: isSelected
                    ? '#fff'
                    : (theme) => theme.palette.primary.main,
                  color: isSelected
                    ? (theme) => theme.palette.primary.main
                    : '#fff',
                }}>
                {unreadMessages}
              </Typography>
            ) : null}
          </Stack>
        </Stack>
      </ListItemButton>
    </Link>
  );
}

// UserAddedToChatModal.tsx
import { Button, Dialog, Divider, Stack, Typography } from '@mui/material';
import { Chat } from '../../pages/ChatPage';
import { Event } from '../../pages/ChatListPage';

interface UserAddedToChatModalProps {
  isOpen: boolean;
  event: Event;
  onClose: () => void;
  onSubmit: (chatId: number, eventId: number) => void;
}

export default function UserAddedToChatModal({
  isOpen,
  event,
  onClose,
  onSubmit,
}: UserAddedToChatModalProps) {
  return (
    <Dialog
      open={isOpen}
      PaperProps={{
        sx: {
          borderRadius: 4,
          p: 2,
        },
      }}
```

```
      fullWidth={true}
      maxWidth='xs'
      onClose={onClose}>
      <Stack spacing={1}>
        <Typography textAlign={'center'} variant='h6'>
          Welcome to {event.chat.name}!
        </Typography>
        <Divider />
        <Typography>
          Congratulations on your arrival—may your time here be filled with joy,
          new adventures, and enriching experiences. You've been added to our
          local{' '}
          <Typography
            component={'span'}
            sx={{
              color: (theme) => theme.palette.primary.main,
              cursor: 'pointer',
            }}
            onClick={() => onSubmit(event.chat.id, event.id)}>
            {' '}
            chat,{' '}
          </Typography>
          where you can ask locals any questions you may have for a smoother
          transition. If you need further assistance, feel free to reach out.
        </Typography>
        <Button onClick={() => onSubmit(event.chat.id, event.id)}>
          Got it
        </Button>
      </Stack>
    </Dialog>
  );
}
```

ДОДАТОК Б

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ДНІПРОВСЬКА ПОЛІТЕХНІКА»

Факультет інформаційних технологій
Кафедра програмного забезпечення комп'ютерних систем

ВІДГУК

Наукового керівника Спірінцева В'ячеслава Васильовича, к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

На кваліфікаційну роботу
студента Садиченка Данила Валерійовича
(прізвище, ім'я, по батькові)

курсу II групи 122м-22-1
спеціальності 122 Комп'ютерні науки

на тему	Дослідження інтерактивної чат платформи для організації комунікації між користувачами з використанням геолокації

Актуальність теми Актуальність даної теми визначається насамперед сучасними тенденціями у сфері інтернет-спілкування та знайомств. Додавання до чату потенційних співрозмовників на основі геолокації відкриває нові можливості для знайомств та спілкування, особливо для тих, хто прагне об'єднувати віртуальний та реальний світи. Актуальність теми також підкреслюється реаліями сучасного ринку чат-рішень, де, незважаючи на розмаїття пропозицій, використання геолокації у контексті спілкування залишається недостатньо висвітленим.

Мета досліджень Полягає у підвищення якості спілкування та знайомства між новими користувачами, за рахунок розробки і використання чат-додатку, де використання геолокації виступає в якості ключової функціональної особливості.

Коротка характеристика розділів роботи Перший розділ роботи присвячений аналізу предметної області в якому розглядаються технології для вирішення постановленої задачі та аналіз вже існуючих рішень чат-платформ. Другий розділ містить аналітичний огляд та порівняння методів та технологій

рішень постановленої задачі під час якого був обраний технологічний стек розробляємої чат-платформи. Третій розділ містить в собі опис розробленої чат-платформи та дослідження ефективності роботи додатку під час збільшеного навантаження на систему через велику кількість одночасних користувачів.

Практичне значення роботи полягає в тому, що запропонований чат-додаток дозволяє полегшити спілкування та знайомства в нових для користувача країнах та містах за рахунок використання геолокації в якості ключової функціональної особливості.

Зауваження та недоліки Недостатньо глибокий аналіз наявних бекенд технологій для вирішення поставленої задачі. Відсутність можливості відправляти приватні повідомлення іншим користувачам

Висновки та оцінка Магістром було проведено розробку та дослідження ефективності роботи розробленої чат-платформи з використанням геолокації. Під час виконання магістерської кваліфікаційної роботи студент Садиченко Д.В. проявив себе грамотним, кваліфікованим спеціалістом здатним приймати самостійно складні технічні рішення. Вважаю, що магістерська кваліфікаційна робота заслуговує оцінку 85 «добре», а Садиченко Д.В. – присвоєння кваліфікації «магістра» з комп'ютерних наук.

Науковий керівник ______________________________

(прізвище, ім'я, по батькові, посада, місце роботи)

«___»_____________20__ р. ____________________

(підпис)

ДОДАТОК В

РЕЦЕНЗІЯ
на кваліфікаційну роботу

студента Садиченка Данила Валерійовича
(прізвище, ім'я, по батькові)

курсу II групи 122м-22-1
кафедри програмного забезпечення комп'ютерних систем
спеціальності 122 Комп'ютерні науки
Тема роботи Дослідження інтерактивної чат платформи для організації комунікації між користувачами з використанням геолокації

Стисла характеристика розділів роботи Перший розділ роботи присвячений огляду проблеми, та аналізу предметної області в якому розглядаються технології для вирішення постановленої задачі та аналіз вже існуючих рішень. Другий розділ містить аналітику та в порівняння методів та технологій рішень постановленої задачі. У третьому Розділі був розроблений та досліджений на ефективність роботи чат-додаток.

Пропозиції, внесені студентом, рівень їх наукового обґрунтування В даній кваліфікаційній роботі студентом надано декілька пропозицій щодо вирішення поставлених задач. Кожна з пропозицій була обґрунтована та підкріплена науковими даними.

Практичне значення роботи полягає в тому, що запропонований чат-додаток дозволяє полегшити спілкування та знайомства в нових для користувача країнах та містах за рахунок використання геолокації в якості ключової функціональної особливості.

Якість оформлення роботи Магістерська кваліфікаційна робота, яку подано на рецензію, виконана у повному обсязі у встановлений термін. Робота є добре структурованою та достатньо проілюстрованою. Викладена основна суть проблеми, що вирішується в ході виконання роботи, і шляхів її вирішення.

Недоліки в роботі відсутність приватних повідомлень, відсутність можливості відправи медіа файлів.

Загальний висновок Магістерська кваліфікаційна робота виконана у
(підготовленість студента до самостійної роботи як спеціаліста)

відповідності з завданням із дотриманням всіх вимог.

Оцінка магістерської роботи Робота заслуговує оцінки «добре», а студент Садиченко Д.В. — присвоєння кваліфікації «магістра» з комп'ютерних наук.

Рецензент ______________________________

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

«___»______________20__ р. ____________________

(підпис)

ДОДАТОК Г

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім'я файла	**Опис**
Пояснювальні документи	
Диплом_Садиченко.doc	Пояснювальна записка роботи. Документ Word.
Диплом_Садиченко.pdf	Пояснювальна записка роботи в форматі PDF
Програма	
Program.rar	Архів. Містить коди програми і откомпільовану програму
Презентація	
Презентація_Садиченко.ppt	Презентація роботи