Міністерство освіти і науки України
Львівський національний університет імені Івана Франка
Факультет електроніки та комп’ютерних технологій
Кафедра системного проектування

КУРСОВА РОБОТА

“Розробка веб месенджеру”

Виконавець:
студент групи ФеІм-12с
Рокуш Д. В.
Науковий керівник:
доц. Демків Л. С.

Львів – 2023

Анотація

Ця курсова робота присвячена розробці веб месенджеру, який містить реєстрацію, авторизацію, надсилання та прийняття повідомлень у режимі реального часу, відображення статусу користувача та сповіщення про отримання нових повідомлень з використанням технологій React, Node.js та Socket.io, що має на меті створення зручного та функціонального інструменту для обміну повідомленнями через Інтернет.

Зокрема, React використовується для створення клієнтського інтерфейсу, Node.js виступає основою серверної частини, а Socket.io забезпечує двосторонній зв'язок між клієнтом та сервером у реальному часі.

Abstract

This course work is devoted to the development of a web messenger that includes registration, authorization, sending and receiving messages in real time, displaying user status and notifications of new messages using React, Node.js and Socket.io technologies, with the aim of creating a convenient and functional tool for messaging over the Internet.

In particular, React is used to create the client interface, Node.js is the basis of the server side, and Socket.io provides two-way communication between the client and the server in real time.

ЗМІСТ

ВСТУП…….. 4
1 ПОНЯТТЯ МЕСЕНДЖЕР ТА ОСНОВНІ ФУНКЦІЇ……. 5
1.1 Визначення поняття месенджер…….. 5
1.2 Еволюція месенджерів……. 6
1.3 Функції та переваги месенджерів……... 8
1.4 Роль месенджерів в сучасному суспільстві…… 10
1.5 Приклади та порівняння сучасних месенджерів……... 13
1.6 Шифрування повідомлень в месенджерах……. 16
2 ТЕХНОЛОГІЇ РОЗРОБКИ ВЕБ МЕСЕНДЖЕРА……... 18
2.1 Вибір програмного середовища…….. 18
2.2 Фреймворки та бібліотеки……... 21
2.3 База даних…….. 23
3 РОЗРОБКА МЕСЕНДЖЕРА…… 26
3.1 Реєстрація і авторизація…….. 26
3.2 Розробка клієнтського інтерфейсу……. 32
3.3 Розробка серверної частини…… 39
3.4 Тестування……. 43
ВИСНОВКИ…….. 48
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ……. 49
ДОДАТОК А……. 50

ВСТУП

У сучасному цифровому світі комунікація між людьми є невіддільною частиною нашого повсякденного життя. Завдяки швидкому розвитку технологій інформації і зв'язку, ми отримали нові можливості для спілкування та обміну повідомленнями. Одним з ключових інструментів цієї комунікації є месенджери.

Перш за все, поняття месенджер охоплює широкий спектр додатків та сервісів, які дозволяють надсилати повідомлення в режимі реального часу. Вони перетворилися на невіддільну складову сучасної комунікації і забезпечують зручність, швидкість та ефективність спілкування.

Ця курсова робота присвячена розробці веб месенджера, який має на меті реалізацію основного функціонала комунікації, а саме реєстрацію та авторизацію користувачів, надсилання та прийняття повідомлень в режимі онлайн, відображення статусу користувачів та надання сповіщень про отримані повідомлення. Для розробки месенджера використовуються такі технології як React, Node.js, Socket.io.

1. ПОНЯТТЯ МЕСЕНДЖЕР ТА ОСНОВНІ ФУНКЦІЇ

1.1 Визначення поняття месенджер

Месенджер є програмним засобом для обміну повідомленнями між користувачами в режимі реального часу. Це комунікаційний інструмент, який дозволяє людям спілкуватися, надсилати повідомлення, обмінюватися медіафайлами та вести групові чати.

Основна мета месенджерів полягає в забезпеченні швидкого і зручного спілкування на відстані. Вони дозволяють користувачам обмінюватися інформацією незалежно від часових та географічних обмежень. Завдяки месенджерам, люди можуть легко спілкуватися з друзями, родичами, колегами та іншими людьми навіть за тисячі кілометрів відстані.

Месенджери надають широкий спектр функцій, включаючи текстові повідомлення, відео та аудіозв'язок, передачу файлів, створення групових чатів, можливість надсилати голосові повідомлення, наліпки та багато іншого. Деякі месенджери також підтримують розширені функції, такі як виклики на телефонні номери, платежі, розсилки повідомлень та інтеграцію з іншими сервісами.

Завдяки постійному розвитку технологій, месенджери стають все більші інтегрованими, надаючи користувачам більше можливостей для спілкування та обміну інформацією. Вони стали не тільки засобом особистої комунікації, але й важливим інструментом для бізнесу, командної роботи та організації подій.

1.2 Еволюція месенджерів

Месенджери мають довгу історію розвитку, починаючи від простих систем обміну повідомленнями до сучасних розширених платформ комунікації. Їх еволюція відбувалася відповідно до змін потреб користувачів та зростання технологічних можливостей.

Спочатку месенджери були базовими текстовими системами, які дозволяли надсилати короткі повідомлення між користувачами. Прикладом таких ранніх месенджерів можуть бути ICQ, AOL Instant Messenger та MSN Messenger. Вони були популярними в 90-х та початку 2000-х років і дозволяли користувачам обмінюватися текстовими повідомленнями та вести приватні чати.

Згодом, з появою широкосмугового Інтернету та розширення можливостей мобільних пристроїв, месенджери отримали нові функціональні можливості. Були додані можливості надсилати медіафайли (фотографії, відео), створювати групові чати та використовувати смайлики та наліпки для вираження емоцій. Прикладами таких месенджерів є WhatsApp, Viber та Telegram, які здобули велику популярність серед користувачів по всьому світу.

У сучасному суспільстві месенджери стали не тільки засобом особистої комунікації, але й важливим інструментом для бізнесу та командної роботи. Деякі месенджери, такі як Slack та Microsoft Teams, спеціально розроблені для спілкування та співпраці в робочих командах, забезпечуючи можливості обміну файлами, організацію відеоконференцій та інтеграцію з іншими формаційними системами. Такі месенджери дозволяють ефективно координувати проєкти, обмінюватися документами, стежити за завданнями та спілкуватися з колегами у реальному часі, незалежно від їх місцеперебування.

Значний вплив на еволюцію месенджерів мали й соціальні мережі. Багато популярних соціальних мереж, таких як Facebook, Twitter та Instagram, додали власні внутрішні системи обміну повідомленнями, щоб дозволити користувачам спілкуватися один з одним безпосередньо на платформі соціальної мережі. Це посилило популярність месенджерів та розширило їх функціонал.

За останні роки з'явилися також нові месенджери, що використовують шифрування та фокусуються на приватності користувачів. Наприклад, Signal та Telegram забезпечують високий рівень безпеки та приватності, шифруючи повідомлення та надаючи користувачам контроль над своїми даними.

У майбутньому можна очікувати подальшого розвитку месенджерів, зокрема впровадження нових технологій, таких як штучний інтелект для покращення функціональності та персоналізації спілкування, використання віртуальної та доповненої реальності для створення нових способів комунікації та забезпечення більш імерсивного досвіду.

Еволюція месенджерів продовжується, і вони стають все більш важливими засобами спілкування та співпраці у нашому сучасному суспільстві.

1.3 Функції та переваги месенджерів

Переваги використання месенджерів включають:

Зручність та доступність: Месенджери забезпечують зручний та простий спосіб комунікації. Вони доступні на різних платформах, таких як смартфони, планшети та комп'ютери, що дозволяє користувачам з легкістю спілкуватися незалежно від пристрою, яким вони користуються.

Миттєве доставлення повідомлень: Месенджери надають можливість миттєво надсилати та отримувати повідомлення. Це дозволяє швидко спілкуватися зі своїми контактами, отримувати важливу інформацію в реальному часі та бути на зв'язку навіть на великій відстані.

Ефективність комунікації: Месенджери допомагають полегшити комунікацію, зменшити час на розв’язання питань та координацію дій. Вони дозволяють обмінюватися ідеями, ділитися документами та робити важливі рішення у спільних чатах, що сприяє ефективності командної роботи.

Глобальний зв'язок: Месенджери зривають бар'єри між місцеперебуваннями, дозволяючи спілкуватися з людьми з усього світу. Вони відкривають можливість зв'язку з рідними, друзями, колегами та новими знайомими, зближуючи людей незалежно від географічних відстаней.

Мультимедійні можливості: Месенджери надають можливість відправляти мультимедійний контент, такий як фотографії, відео та аудіо.

Синхронізація на різних пристроях: Багато месенджерів підтримують синхронізацію повідомлень та історії чату між різними пристроями. Це означає, що ви можете почати розмову на смартфоні, а потім продовжити її на комп'ютері без втрати даних або прогресу.

Вартість та доступність: Багато месенджерів пропонують безкоштовну основну функціональність, що робить їх доступними для широкого кола користувачів. Це зменшує витрати на комунікацію і робить її доступною для всіх, хто має доступ до Інтернету.

Бізнес-функції: Деякі месенджери надають спеціальні функції для бізнес-комунікації, такі як інтеграція з інструментами проєктного управління, обмін файлами, створення віртуальних конференцій та інше. Це полегшує комунікацію в межах бізнесу та сприяє покращенню робочих процесів.

Збереження історії повідомлень: Месенджери часто зберігають історію повідомлень, що дозволяє вам переглянути попередні розмови та повернутися до них в разі потреби. Це може бути корисно для відстеження деталей, пошуку інформації або ведення документації.

Забезпечення приватності та безпеки: Багато месенджерів надають важливі функції приватності та захисту даних. Вони дозволяють користувачам керувати своєю конфіденційністю, встановлювати обмеження на доступ до свого профілю та повідомлень, а також шифрувати комунікацію для запобігання несанкціонованому доступу до інформації.

Інтеграція з іншими сервісами: Деякі месенджери надають можливість інтегруватися з іншими платформами та сервісами, такими як соціальні мережі, електронна пошта, хмарні сховища та інші. Це полегшує обмін інформацією, співпрацю та доступ до різних інструментів без потреби перемикатися між додатками.

Підтримка додаткових функцій: Деякі месенджери пропонують додаткові функції, такі як голосові та відеодзвінки, віртуальні чат-боти, мобільні платежі та інше. Це розширює можливості користувачів і дозволяє використовувати месенджери не лише для текстового спілкування, але й для різноманітних потреб.

Загалом, месенджери є потужними засобами комунікації, які забезпечують швидку, зручну та ефективну спілкування, незалежно від місцеперебування. Вони відкривають безліч можливостей для особистих, професійних та бізнес-цілей.

1.4 Роль месенджерів в сучасному суспільстві

Месенджери стали не лише засобом швидкої комунікації, але й платформою для обміну інформацією, вираження емоцій, ведення ділових переговорів та проведення віртуальних зустрічей.

У сучасному суспільстві месенджери використовуються в різних сферах життя. Вони стали невіддільною частиною особистої комунікації, дозволяючи людям легко знаходити зв'язок з рідними та друзями, незалежно від географічних відстаней. Месенджери також активно використовуються в бізнесі для підтримки комунікації між колегами, клієнтами та партнерами. Вони полегшують організацію процесів, спільне вирішення завдань та забезпечують швидкий обмін інформацією.

Крім того, месенджери впливають на соціальний аспект суспільства, змінюючи спосіб взаємодії між людьми. Вони стимулюють активну спільноту, об'єднують людей за спільними інтересами, сприяють розвитку нових форм комунікації та спілкування.

Роль месенджерів в сучасному суспільстві є надзвичайно важливою, оскільки вони сприяють покращенню комунікаційних процесів, зменшенню бар'єрів для спілкування та сприяють зближенню людей. Вони є потужними інструментами, що допомагають нам побудувати міцні зв'язки, обмінятися ідеями та інформацією, співпрацювати та розвивати особисті та професійні навички.

Зазвичай месенджери використовуються людьми як засіб особистого спілкування. Вони допомагають зберігати зв'язок з рідними, друзями та колегами, дозволяючи обмінюватись фотографіями, відео, голосовими повідомленнями та іншими файлами.

Також месенджери використовують для створення групових чатів, які об’єднують людей за певними інтересами, цілями, хобі або професіональними сферами.

Месенджери використовуються для ведення бізнесу. Вони дозволяють спілкуватися зі співробітниками, клієнтами партнерами, полегшуючи обмін інформацією, координацію проєктів та прийняття рішень.

Месенджери стають платформою для розвитку електронної комерції, дозволяючи бізнесам спілкуватися зі своїми клієнтами, надавати консультації, приймати замовлення та проводити транзакції прямо в чаті. Вони дозволяють підприємствам залучати нових клієнтів, пропонувати послуги та товари через чат-боти, проводити рекламні кампанії та надавати індивідуальну підтримку клієнтів.

Деякі месенджери включають функції з використання криптовалюти та фінансових послуг. Вони дозволяють користувачам здійснювати перекази грошей, зберігати цифрові активи та отримувати фінансову інформацію прямо в месенджері.

Є месенджери які дозволяють користувачам створювати чат-ботів наділених різними функціями, таких як: конвертація зображень та відео файлів, редагування зображень та відео за допомогою штучного інтелекту, вивід інформації в реальному часі з потрібних для користувача джерел, вивід транзакцій користувачів з блокчейну.

Месенджери стають платформою для споживання медійного контенту. Вони надають можливість відправляти та отримувати фотографії, відео, аудіо та інші мультимедійні матеріали, а також споживати новини, розважальний та освітній контент.

Загалом, месенджери відіграють різноманітні ролі в сучасному суспільстві, полегшуючи комунікацію, сприяючи соціальній активізації та розвитку бізнесу. Вони є важливим інструментом для зміцнення зв'язків між лідерами та членами громадськості, сприяють інноваціям та зручності в повсякденному житті. Завдяки їхнім функціям і можливостям, месенджери продовжують розширювати свої можливості і вплив на суспільство.

Багато месенджерів використовують розумні алгоритми та штучний інтелект для покращення користувацького досвіду. Вони надають функції автоматичного перекладу, розпізнавання мови, голосових команд, рекомендацій та інші інноваційні можливості.

Безпека та приватність є важливими аспектами месенджерів. Багато з них пропонують шифрування повідомлень, можливості видалення повідомлень, управління налаштуваннями конфіденційності та інші функції, що допомагають користувачам зберігати контроль над своєю приватністю та безпекою.

Месенджери часто інтегруються з іншими сервісами та платформами, такими як соціальні мережі, електронна пошта, календарі, хмарні сховища та інші. Це дозволяє користувачам зручно комунікувати та співпрацювати, використовуючи різноманітні інструменти і сервіси в одному місці.

1.5 Приклади та порівняння сучасних месенджерів

Сучасні месенджери пропонують безліч варіантів для спілкування та обміну повідомленнями. Ось кілька прикладів та порівняння деяких популярних месенджерів.

WhatsApp:

- Одним з найпопулярніших месенджерів, який надає широкі можливості спілкування.
- Забезпечує миттєве доставлення повідомлень та мультимедійний обмін.
- Підтримує відео- та голосові дзвінки.
- Захищений шифруванням повідомлень, що забезпечує приватність користувачів.
- Доступний на різних платформах, включаючи смартфони та комп'ютери.

Telegram:

- За популярністю наближається до WhatsApp і пропонує подібний функціонал.
- Забезпечує миттєве доставлення повідомлень та мультимедійний обмін.
- Підтримує відео та голосові дзвінки.
- Має потужні інструменти для організації групових чатів та каналів.
- Приватність користувачів важлива для Telegram, і він також пропонує шифрування повідомлень.
- Можливість синхронізації контактів та повідомлень між пристроями.
- Надає можливість відправляти голосові повідомлення та використовувати наліпки.

Viber:

- Популярний месенджер зі широким функціоналом та соціальними можливостями.

- Забезпечує миттєве доставлення повідомлень, а також можливість викликів.
- Має вбудовані функції для створення групових чатів та відеоконференцій.
- Надає можливість відправляти голосові повідомлення та використовувати наліпки.
- Можливість синхронізації контактів та повідомлень між пристроями.

Facebook Messenger:

- Інтегрований з соціальною мережею Facebook і надає зручну комунікацію з друзями та контактами з Facebook.
- Забезпечує миттєве доставлення повідомлень та можливість обміну медіафайлами.
- Підтримує відео та голосові виклики.
- Має різні функції, такі як наліпки, емодзі, фільтри для фотографій.
- Надає можливість створювати групові чати та організовувати події.

Всі розглянуті месенджери наділені такими функціями, як голосові та відео дзвінки, забезпечують миттєве доставлення повідомлень. Кожен з них має свої переваги і підходить для різних потреб користувачів. Вибір залежить від особистих уподобань, функцій, які вам потрібні, і пріоритетів, таких як приватність, зручність та соціальне спілкування.

Серед популярних професійних месенджерів, які використовуються для комунікації та співпраці в бізнес-середовищі, можна виділити Slack та Microsoft Teams.

Slack є одним з найбільш відомих та поширених професійних месенджерів. Він надає зручні можливості для створення робочих просторів та каналів для комунікації в команді або проєкті. Slack підтримує надсилання повідомлень, спільну роботу над документами, інтеграцію з іншими інструментами та багато іншого.

Microsoft Teams є частиною платформи Office 365 та використовується для комунікації та співпраці в команді або організації. Він надає можливості для групових чатів, відеозв'язку, спільної роботи над документами та інтеграції з іншими інструментами Microsoft, такими як Word, Excel, SharePoint тощо.

1.6 Шифрування повідомлень в месенджерах

Сучасні месенджери використовують різні методи і алгоритми шифрування для захисту повідомлень користувачів.

До прикладу Telegram використовує комбінацію різних методів шифрування для кодування повідомлень Основні методи шифрування, що використовуються в Telegram, включають:

Шифрування точка-точка (End-to-End Encryption, E2EE): Telegram використовує E2EE для захисту конфіденційності повідомлень між використовувачами. Однак, важливо відзначити, що E2EE в Telegram не ввімкнено за замовчуванням і доступний лише в режимі "Секретні чати". У секретних чатах всі повідомлення шифруються на пристрої відправника і розшифровуються тільки на пристрої отримувача.

Симетричне шифрування: Для забезпечення безпеки повідомлень, які не передаються через секретні чати, Telegram використовує симетричне шифрування. Кожне повідомлення шифрується за допомогою унікального симетричного ключа, який обмінюється між відправником і отримувачем. Після цього повідомлення передається зашифрованим і розшифровується на пристрої отримувача з використанням спільного ключа.

AES шифрування: Для симетричного шифрування Telegram використовує алгоритм AES (Advanced Encryption Standard) з ключем довжиною 256 біт. AES є одним з найбільш поширених і надійних алгоритмів шифрування.

RSA шифрування: Для обміну симетричним ключем між відправником і отримувачем Telegram використовує асиметричний RSA алгоритм шифрування. За допомогою RSA ключа відправник шифрує симетричний ключ, який тоді може бути розшифрований отримувачем з використанням його власного приватного ключа.

E2EE шифрування це такий метод захисту даних від несанкціонованого перегляду та зміни, за якого доступ до повідомлень або даних мають тільки ті

користувачі, які беруть участь у спілкуванні. Це шифрування гарантує недоступність будь-якої призначеної для користувача інформації (текст, відео і звук, файли, що передаються) навіть серверам, які беруть участь у передачі даних. Шифрування і розшифрування повідомлень відбувається безпосередньо на тих пристроях користувачів, які використовуються в даний момент у сеансі зв'язку. Таким чином, ніхто крім одержувача не зможе прочитати або підмінити передані дані. Захист не поширюється на метадані, наприклад, дата і час надсилання повідомлення не будуть приховані.

E2EE шифрування відбувається по такому принципу (рис.1.1):

- На початку, при створенні сеансу зв'язку, додаток-одержувач генерує два ключі: публічний і закритий.
- Публічний ключ відправляється на сервер. Цей ключ використовується тільки для шифрування і тому його наявність жодним чином не дає змоги серверу прочитати повідомлення відправника. Тут важливо, щоб сервер не міг підмінити пару закритого і публічного ключів на свою.
- Додаток-відправник завантажує з сервера публічний ключ для шифрування повідомлення.
- Зашифроване повідомлення надсилається на сервер.
- Додаток-одержувач завантажує повідомлення і розшифровує його за допомогою свого закритого ключа.

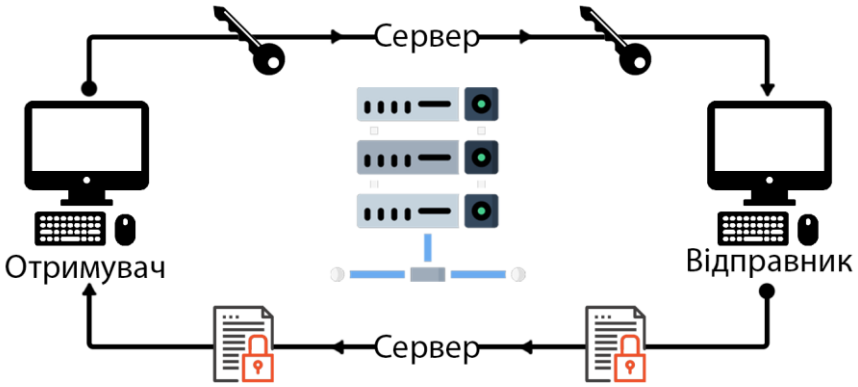

Рис.1.1 E2EE шифрування

2. ТЕХНОЛОГІЇ РОЗРОБКИ МЕСЕНДЖЕРА

2.1 Вибір програмного середовища

Вибір програмного середовища є важливим кроком при розробці веб-додатків, і в даній роботі я обрав Node.js. Ось деякі причини, чому я вважаю Node.js відповідним вибором для розробки веб месенджера, порівняно з іншими середовищами:

1. Ефективність та масштабованість: Node.js базується на подіях та неблокуючому вводі/виводі (non-blocking I/O), що робить його дуже ефективним та масштабованим. Він здатний обробляти багато одночасних з'єднань без блокування, що особливо важливо для месенджерів з великою кількістю одночасних користувачів.

2. Швидкодія: Node.js використовує двигун V8, розроблений Google, який забезпечує високу швидкодію виконання JavaScript-коду. Це дозволяє розробникам побудувати швидкі та реактивні веб-додатки.

3. Єдність мови: Node.js дозволяє використовувати JavaScript як мову програмування як на стороні сервера, так і на стороні клієнта. Це спрощує розробку, оскільки нема потреби вивчати нову мову програмування для серверної частини. Крім того, це дозволяє легко перевикористовувати код та модулі між сервером та клієнтом.

4. Розширюваність: Node.js має велику кількість модулів та бібліотек, доступних через менеджер пакетів npm. Це дозволяє розробникам швидко використовувати готові рішення та розширювати функціонал своїх додатків.

Хоча Node.js має багато переваг, варто враховувати і деякі його обмеження:

Одно поточність: Оскільки Node.js працює в одному потоці, це може призвести до проблем з обробкою важких обчислювальних завдань. Якщо веб месенджер потребує інтенсивних обчислень, може знадобитися розподілення

завдань на кілька процесів або використання додаткових модулів для виконання таких завдань.

Потреба у додаткових модулях: Хоча Node.js має велику кількість модулів та бібліотек, деякі функціональні можливості можуть вимагати встановлення додаткових пакетів. Це може збільшити складність розробки та підтримки проєкту.

Недостатність документації: Порівняно з іншими середовищами, деякі аспекти Node.js можуть мати менш докладну документацію або бути менш покритими прикладами. Це може призвести до додаткових зусиль для зрозуміння та використання деяких функцій або патернів розробки.

Менш підходить для важкостартових завдань: Node.js зазвичай показує найкращі результати в веб-додатках з великою кількістю одночасних з'єднань. Для менш вимогливих до швидкодії додатків або додатків, які потребують великих обчислень при запуску, інші середовища, такі як Python або Ruby, можуть бути більш слушними варіантами.

Враховуючи всі ці фактори, я обрав Node.js для розробки веб месенджера, оскільки він пропонує ефективну та масштабовану платформу для обробки багатьох одночасних з'єднань, має швидкодію та єдність мови з клієнтською частиною додатка. Також, наявність великої кількості пакетів та активна спільнота розробників робить його зручним із погляду розширення функціональності та забезпечує багато ресурсів для підтримки та вирішення проблем.

Порівнюючи з іншими середовищами, наприклад Python та Ruby, Node.js має деякі переваги. На відміну від Python, який є мовою загального призначення, Node.js був спеціально розроблений для розробки веб-додатків. Він має більш простий та зрозумілий синтаксис, що полегшує розробку. Крім того, Node.js використовує JavaScript, яка є однією з найпоширеніших мов програмування для веб-розробки, що означає доступність великої кількості ресурсів та знань для підтримки.

У порівнянні з Ruby, Node.js відрізняється вищою швидкодією та ефективністю завдяки використанню подій та неблокуючого вводу-виводу. Це особливо важливо для месенджерів, які потребують миттєвої обробки повідомлень та швидкого оновлення стану. Крім того, Node.js має активну спільноту розробників, що забезпечує широкий вибір пакетів та бібліотек для розробки.

Загалом, вибір Node.js для розробки веб месенджера мотивований його ефективністю, швидкодією, єдністю мови з клієнтською частиною та доступністю ресурсів для підтримки. Використання Node.js дозволить розробити потужний та масштабований месенджер, який зможе відповідати вимогам сучасного веб-середовища.

2.2 Фреймворки та бібліотеки

Для написання клієнтської частини коду я використав React, це потужний JavaScript фреймворк для розробки користувацького інтерфейсу. Його основною перевагою є можливість створювати компоненти, які дозволяють організовувати логіку та представлення окремих частин веб-додатка. React використовує віртуальний DOM для ефективного оновлення інтерфейсу та рендерингу змін. React дозволяє розробникам створювати великі вебзастосунки, які використовують дані, котрі змінюються з часом, без перезавантаження сторінки. Його мета полягає в тому, щоб бути швидким, простим, масштабованим. Використання React у розробці веб месенджера дозволило мені легко управляти компонентами та забезпечити ефективну роботу з даними.

Для надсилання та приймання повідомлень в реальному часі було використано бібліотеку Socket.IO. Це JavaScript-бібліотека для веб-додатків і обміну даними в реальному часі. Складається з двох частин: клієнтської, яка запускається в браузері, і серверної для node.js. Обидва компоненти мають схоже API. Подібно до node.js, Socket.IO подієво-орієнтована.

Socket.IO головним чином використовує протокол WebSocket, але якщо потрібно, використовує інші технології, наприклад Flash Socket, AJAX Long Polling, AJAX Multipart Stream, надаючи той самий інтерфейс. Крім того, що Socket.IO можна використовувати як оболонку для WebSocket, вона містить багато інших функцій таких, як зберіганням даних, пов'язаних із кожним клієнтом, і асинхронним введенням та виведенням.

За допомогою Socket.IO можна реалізувати аналітику в реальному часі, багатокористувацькі ігри, обмін миттєвими повідомленнями та спільну роботу з документами в реальному часі.

Socket.IO досить популярний, його використовують Microsoft, Yammer, Zendesk, Trello і багато інших організацій для створення систем реального часу. Socket.IO працює на основі подій.

Використання Socket.io у веб месенджері дозволило мені реалізувати миттєве відправлення та отримання повідомлень між користувачами без необхідності постійного оновлення сторінки.

Bootstrap - це популярний HTML, CSS та JavaScript фреймворк для швидкої та простої розробки адаптивних вебінтерфейсів. Він містить набір готових стилів, компонентів та шаблонів, що допомагають забезпечити єдність дизайну та покращити користувацький досвід.

Використання Bootstrap у розробці веб месенджера дозволило мені зекономити час на розробці дизайну месенджера, завдяки готовим компонентам.

Також в роботі була використана бібліотека Moment, завдяки якій здійснюється показ часу відправленого повідомлення. Moment є однією з найпопулярніших бібліотек для роботи з датами та часом в JavaScript.

Основна причина обрання Moment полягає в його простоті використання та багатому функціоналі. Вона надає зручні методи для форматування, парсингу, маніпулювання та відображення дат та часу. Завдяки цьому, можна легко виконувати операції з датами, такі як отримання різниці між двома часовими мітками, перетворення дати у різні формати або показ поточного часу у зручному для користувача вигляді.

Moment також надає можливості локалізації, що дозволяє відображати дати та час у форматі, зрозумілому для користувачів з різних країн. Це особливо важливо у месенджерах, де користувачі можуть бути з різних часових зон.

Загалом, використання бібліотеки Moment у розробці месенджера допомагає забезпечити точне та зручне відображення часу відправлення повідомлень. Вона спрощує роботу з датами та часом, забезпечує гнучкість та локалізацію, що робить її ідеальним вибором для цієї задачі.

2.3 База даних

У курсовій роботі важливою складовою частиною є база даних. База даних використовується для збереження та управління інформацією про користувачів, повідомлення, чати що потрібні для роботи месенджера.

Одним з популярних варіантів бази даних, який можна використовувати для розробки веб-додатків, є реляційна база даних, наприклад MySQL або PostgreSQL. Реляційні бази даних добре підходять для збереження структурованої інформації та забезпечують можливості для виконання складних запитів та оптимізації даних.

Однак, для розробки месенджера, де важлива швидкість обміну повідомленнями в реальному часі, може бути доцільно розглянути використання нереляційної бази даних, такої як MongoDB або Firebase Realtime Database. Нереляційні бази даних зазвичай пропонують швидку запис та зчитування даних, а також масштабованість, що є важливими аспектами для месенджера.

У випадку використання Node.js в якості серверної технології, можна використовувати модулі для взаємодії з базою даних, такі як mongoose для MongoDB або Sequelize для реляційних баз даних. Ці модулі дозволяють спростити роботу з базою даних, надають ORM (Object-Relational Mapping) для легкої взаємодії з даними та допомагають забезпечити безпеку даних.

Вибір бази даних залежить від конкретних потреб та вимог проєкту. Реляційні бази даних підходять для проєктів зі складною структурою даних та складними запитами, тоді як нереляційні бази даних, зокрема MongoDB, можуть бути корисними для проєктів, де потрібно зберігати та отримувати дані швидко, особливо в реальному часі. MongoDB використовує формат документів JSON-подібний BSON, що дозволяє зберігати дані у гнучкому форматі. Вона підтримує реплікацію та шардування, що дозволяє горизонтально масштабувати базу даних.

Для курсової роботи я обрав базу даних MongoDB. Ось кілька аргументів, чому саме ця база даних є хорошим вибором для написання месенджера.

Гнучкість даних: MongoDB використовує документо орієнтовану модель, де дані зберігаються у форматі JSON-подібного документа. Це дозволяє зберігати дані у гнучкому та невимушеному форматі, без потреби визначення жорсткої схеми. У випадку месенджера, де повідомлення можуть мати різну структуру або містити різні типи даних (текст, зображення, відео і т.д.), гнучкість MongoDB дозволяє легко зберігати такі дані.

Швидкість та масштабованість: MongoDB пропонує швидку та ефективну роботу з даними. Вона може працювати на різних рівнях консистентності даних, що дозволяє досягти високої швидкодії та низького часу відповіді. Крім того, MongoDB підтримує реплікацію та шардування, що дозволяє горизонтально масштабувати базу даних, щоб відповідати зростальним потребам месенджера.

Реальний час: MongoDB підтримує можливість спостерігати зміни даних в реальному часі. Це може бути корисно для месенджера, де користувачі очікують миттєвого доставлення та відображення повідомлень. За допомогою механізму відстеження змін (change streams) MongoDB, можна реалізувати функціональність реального часу, яка сповіщає клієнтів про оновлення даних.

Простота розробки: MongoDB має широкий набір драйверів та бібліотек для підтримки різних мов програмування, включаючи Node.js. Це робить його дуже доступним для розробників, які вже працюють з Node.js. Драйвери та бібліотеки надають зручний API для взаємодії з базою даних, що спрощує розробку інтеграції з MongoDB.

Горизонтальна масштабованість: MongoDB дозволяє розподіляти дані на кілька серверів (шардів), що дозволяє горизонтально масштабувати базу даних. Це особливо важливо для проєктів, які мають велику кількість користувачів або потребують високої доступності та швидкості. Масштабованість MongoDB дозволяє легко розширювати ресурси та оброблювати зростаючий обсяг даних без перебудови архітектури бази даних.

Спільнота та підтримка: MongoDB має велику та активну спільноту розробників, що надає доступ до багатої документації, навчальних матеріалів та форумів. Це означає, що можна легко знайти відповіді на запитання та отримати підтримку у випадку проблем. Крім того, MongoDB є комерційно підтримуваним продуктом, що означає, що ви можете отримати підтримку та консультації від розробників компанії MongoDB Inc.

3. РОЗРОБКА МЕСЕНДЖЕРА

3.1 Реєстрація і авторизація

Для реєстрації та авторизації користувачів використовується контекст “AuthContext”, який містить необхідні функції та стан для керування реєстрацією та авторизацією. Основні кроки для реєстрації та авторизації користувача описані нижче.

Реєстрація користувача

- В компоненті Register користувач вводить свої дані, такі як ім'я, електронна пошта та пароль.
- Після заповнення полів користувач натискає кнопку "Зареєструвати".
- У функції “registerUser” дані форми зчитуються та передаються в POST-запит до сервера для обробки.
- На сервері виконується перевірка наявності користувача з введеною електронною поштою. Якщо користувач з такою поштою уже існує, повертається помилка.
- Виконується перевірка валідності даних, таких як коректність електронної пошти та надійності пароля.
- Пароль хешується перед збереженням у базі даних для забезпечення безпеки.
- Якщо всі перевірки пройдені успішно, користувача зберігається у базі даних, і йому надсилається JWT-токен для авторизації.
- Інформація про користувача та отриманий токен зберігаються у локальному сховищі (localStorage).

Авторизація користувача

- В компоненті “Login” користувач вводить свою електронну пошту та пароль.
- Після заповнення полів користувач натискає кнопку "Вхід".

- У функції loginUser дані форми зчитуються та передаються в POST-запит до сервера для обробки.
- На сервері виконується перевірка коректності введених даних.
- Якщо введені дані вірні, користувачу надсилається JWT-токен для авторизації входу.
- Отриманий токен зберігається у локальному сховищі (localStorage).
- Користувача авторизовано і отримані дані про нього зберігаються у стані за допомогою “setUser”.

Після реєстрації або авторизації користувача інформація про нього зберігається у стані за допомогою setUser.

У разі виходу користувача з системи виконується видалення його даних з локального сховища та стану за допомогою localStorage.removeItem("User") та setUser(null).

Цей функціонал реалізований у React компонентах Login та Register за допомогою використання контексту AuthContext, який надає доступ до необхідних функцій та стану для реєстрації та авторизації користувачів. Також присутні функції updateRegisterInfo та updateLoginInfo, які дозволяють оновлювати дані форми під час введення користувачем.

На серверній стороні реєстрація та авторизація користувача виконується за допомогою виклику відповідних функцій registerUser та loginUser. У цих функціях виконуються перевірки даних та взаємодія з базою даних (MongoDB) за допомогою моделі користувача userModel.

Приклад роботи реєстраціії та авторизації:

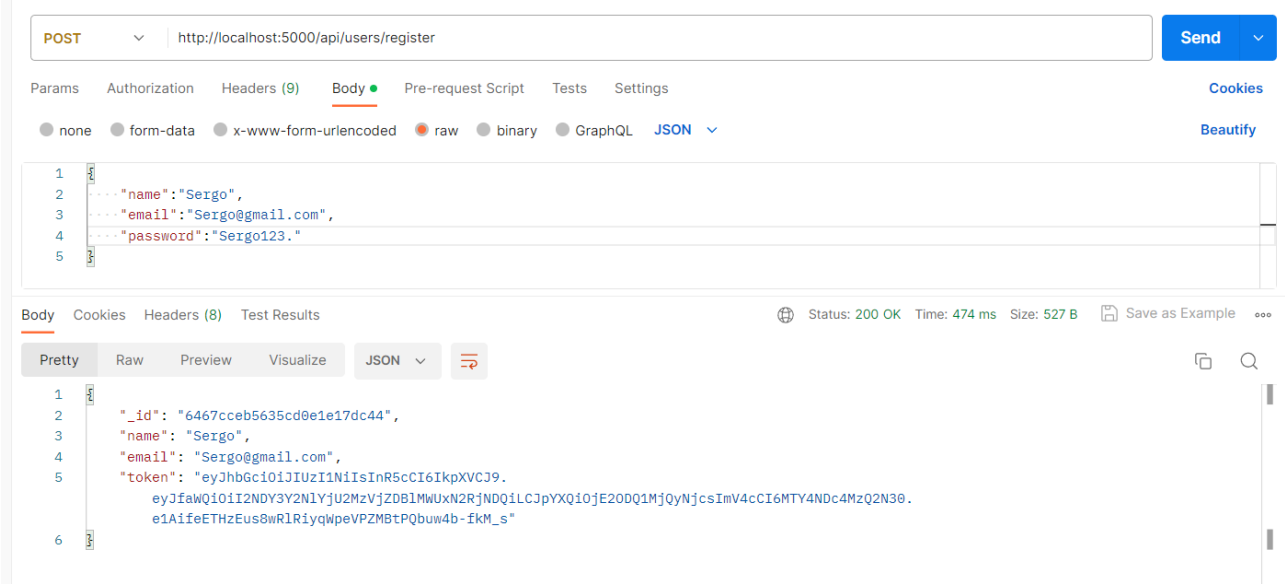

Рис. 3.1 Виконання реєстрації користувача

З рис.3.1 видно, як проходить реєстрація користувача, на вхід подається ім’я користувача, електронна пошта та пароль. А на виході отримуємо id користувача, ім’я, електронну пошту та токен. Пароль записується в базу даних в хешованому вигляді, це видно на рис.3.2.

Рис. 3.2 Вигляд даних користувача в базі даних

На наступних рисунках 3.3 – 3.6 можна бачити які помилки будуть виводитись користувачу, якщо він неправильно заповнить поля для вводу при реєстрації. На рис 3.7 продемонстровано, що буде якщо неправильно ввести логін або пароль.

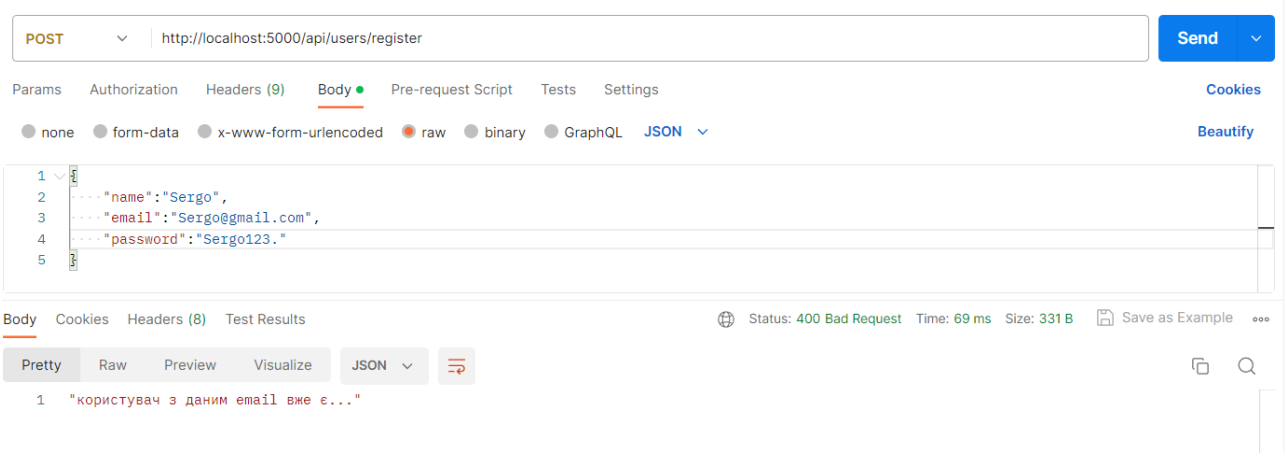

Рис.3.3 Реєстрація з використанням електронної пошти яка уже є в базі даних(без інтерфейсу)

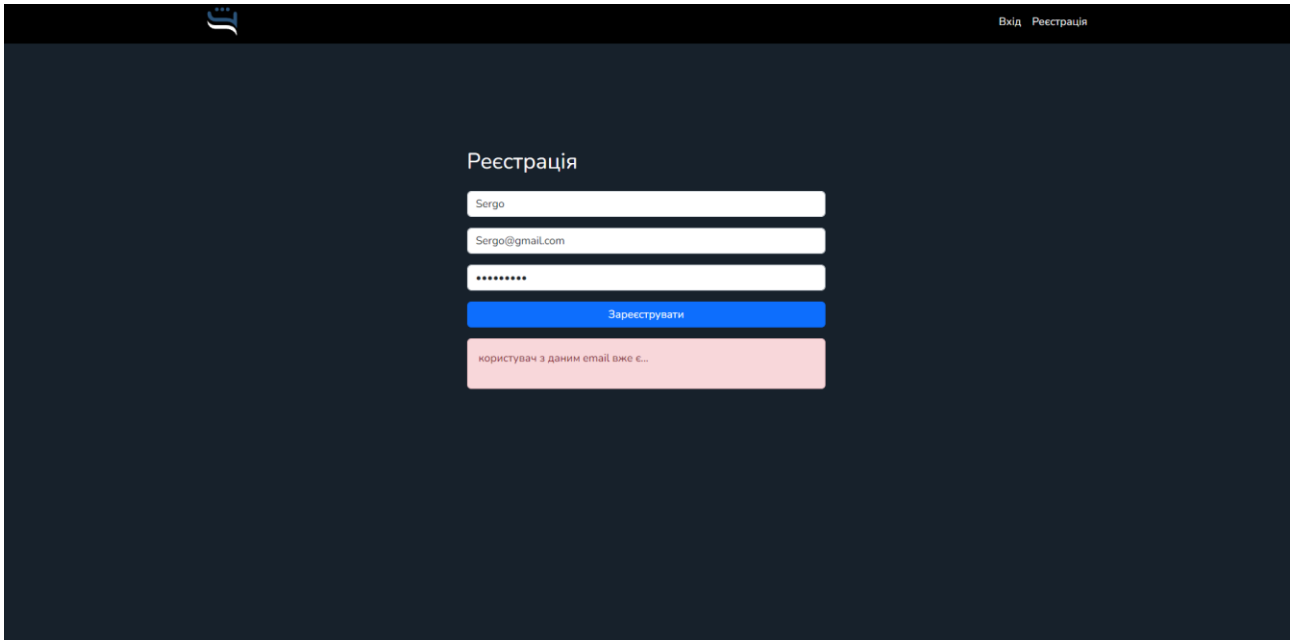

Рис.3.4 Реєстрація з використанням електронної пошти яка уже є в базі даних(з інтерфейсом)

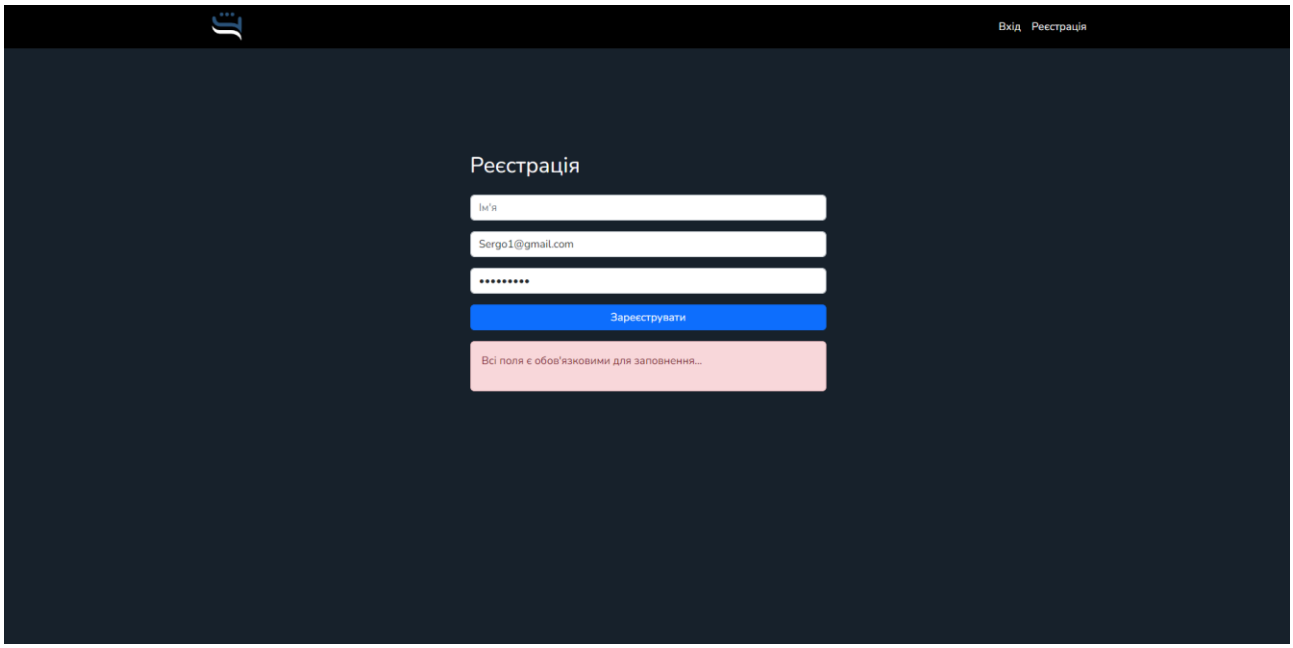

Рис.3.5 Реєстрація з пропущеним полем

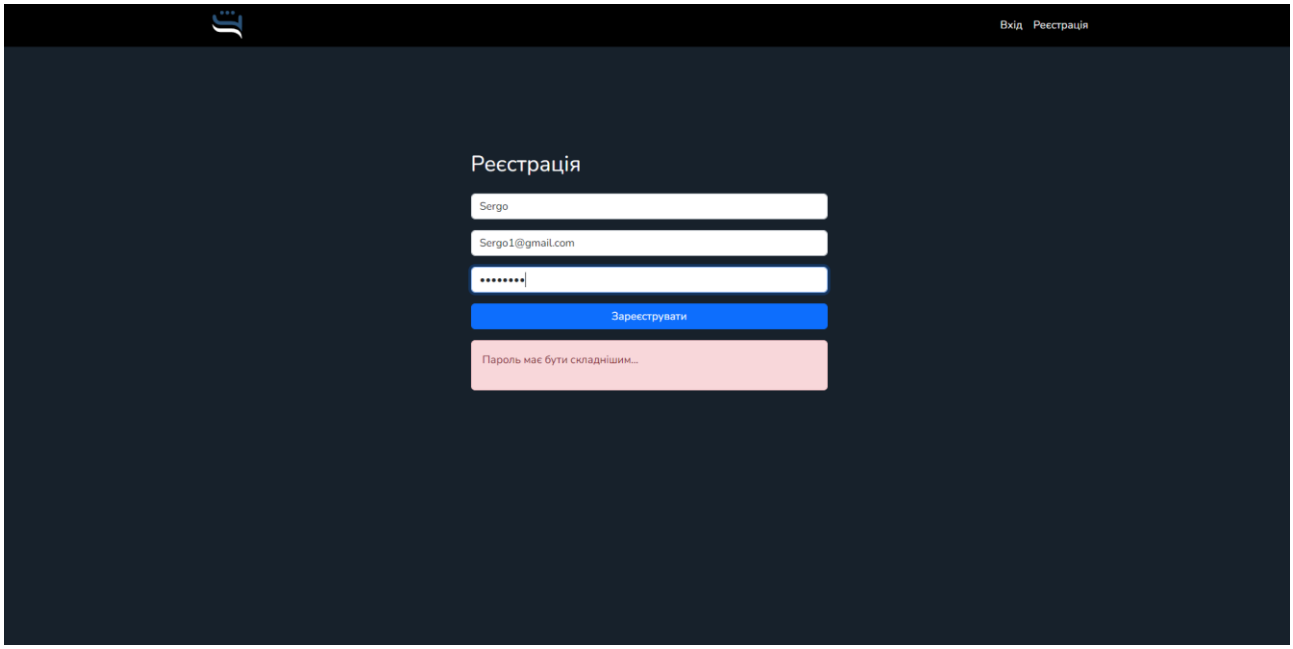

Рис.3.6 Реєстрація з легким паролем

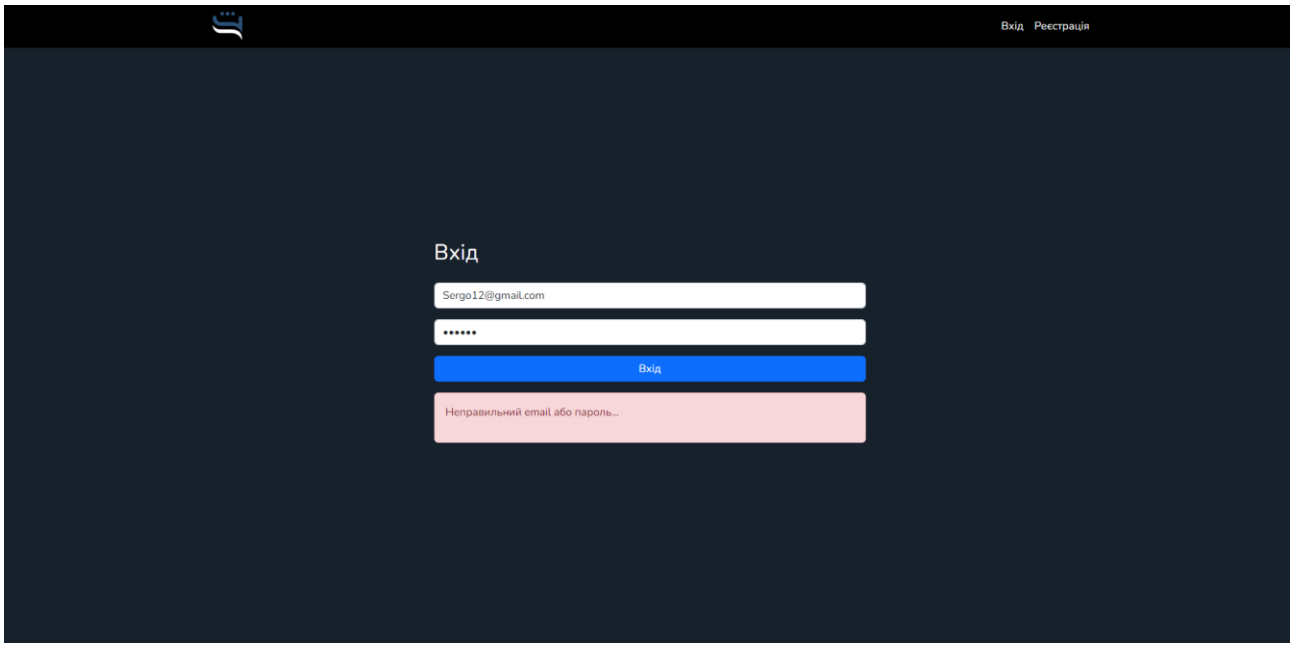

Рис.3.7 Логін з неправильною електронною поштою або паролем

Якщо не виникло жодних помилок і реєстрація або авторизація пройшла успішно, відкриється веб месенджер рис 3.8.

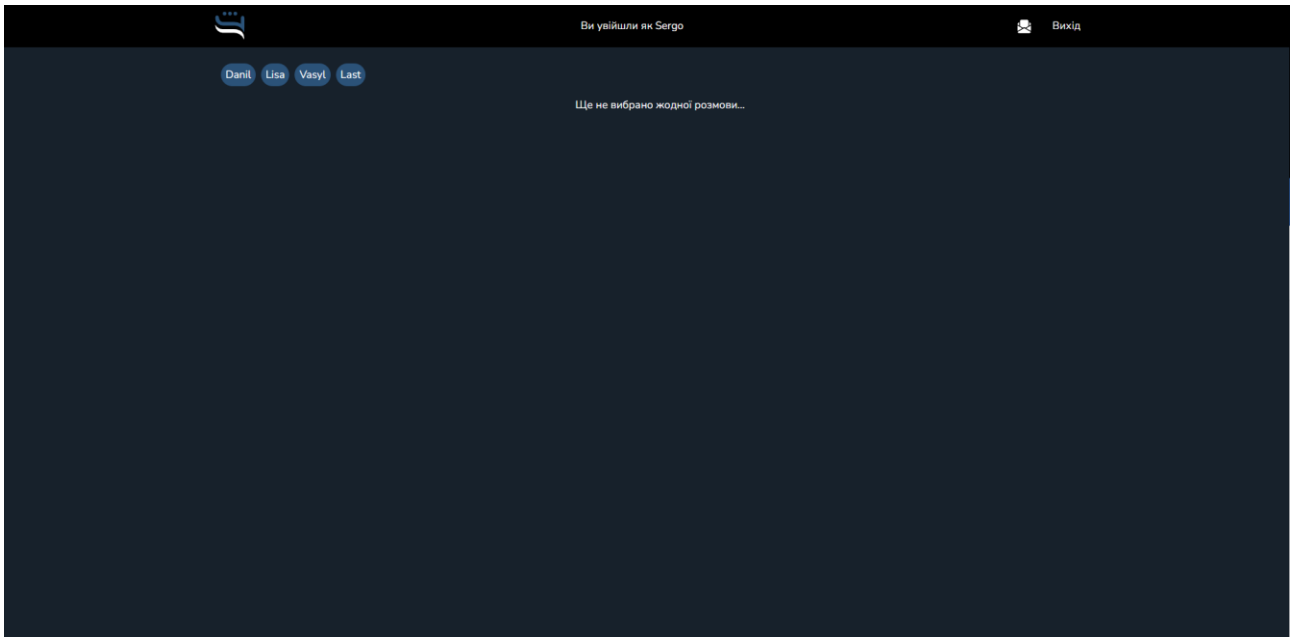

Рис.3.8 Вигляд месенджера після реєстрації нового користувача

3.2 Розробка клієнтського інтерфейсу

Компонент NavBar:

- Використовується хук useContext для отримання інформації про користувача та функції logoutUser з контексту AuthContext.
- У лівій частині панелі розміщений заголовок, який є посиланням на домашню сторінку.
- По центрі розміщена інформація про користувача(якщо користувача авторизовано).
- У правій частині панелі розміщені навігаційні посилання.
- В залежності від статусу користувача відображаються різні посилання:
    - Якщо користувача авторизовано (user має значення), відображаються компонент Notification для сповіщень, посилання “Вихід” рис 3.9.
    - Якщо користувач не авторизований (user не має значення), відображаються посилання на сторінки входу та реєстрації рис 3.10.

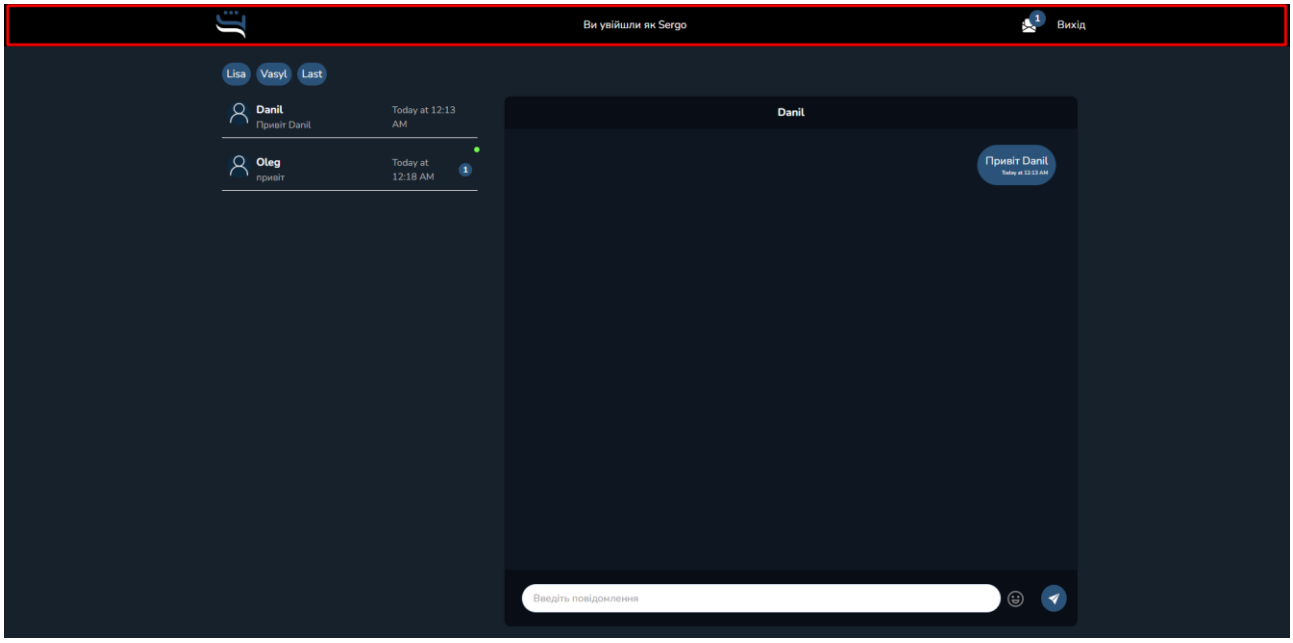

Рис.3.9 Червоним прямокутником виділено компонент NavBar

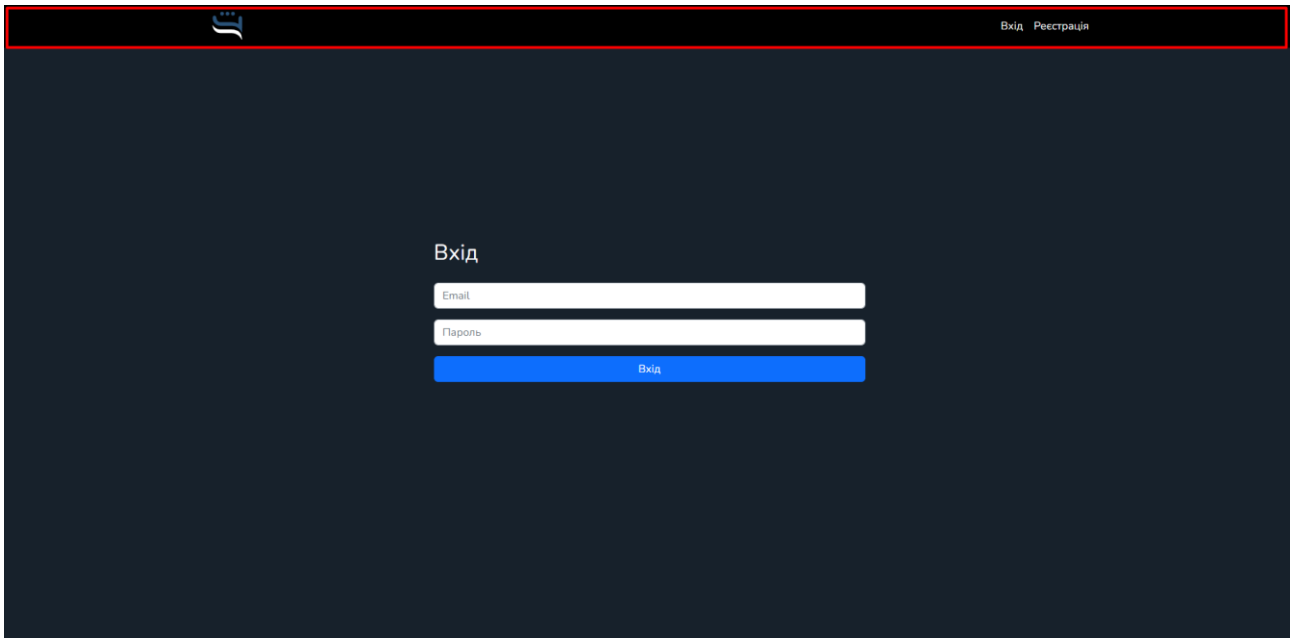

Рис.3.10 Червоним прямокутником виділено компонент NavBar

Компонент PotentialChats:

- Цей компонент відображає потенційних користувачів, з якими можна розпочати новий чат рис 3.11.
- Він використовує контекст AuthContext для отримання інформації про поточного користувача.
- Також він використовує контекст ChatContext для отримання списку потенційних користувачів та функції створення нового чату.
- Компонент створює блок для кожного користувача.
- При натисканні на користувача створюється новий чат з обраним користувачем і відкривається вікно чату.
- Якщо користувач, з яким можна створити чат, є онлайн, то до блоку додається клас “user-online” для позначення статусу онлайн.

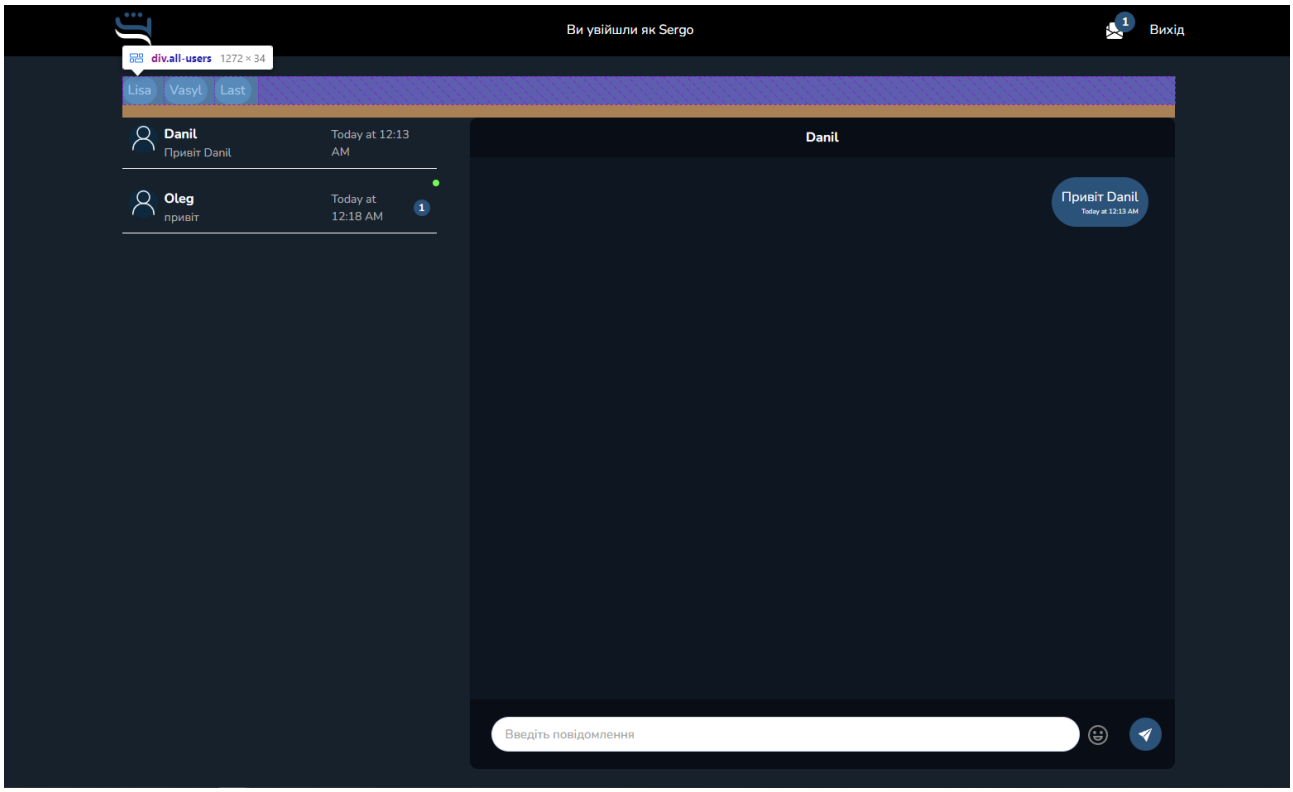

Рис.3.11 прямокутником виділено компонент PotentialChats

Компонент UserChat:

- Він імпортує Stack з react-bootstrap для створення розміщення компонентів.
- Він також імпортує кілька хуків та контекстів, які використовуються у компоненті.
- Компонент отримує інформацію про чат інформацію про поточного користувача.
- Він використовує хук useFetchRecipientUser для отримання інформації про користувача, отримувача і повідомлень у даному чаті.
- Він також використовує контекст ChatContext для отримання списку онлайн користувачів, повідомлень і функцій для роботи з ними.
- Компонент використовує функцію unreadNotificationsFunc для підрахунку непрочитаних повідомлень.
- Він фільтрує непрочитані повідомлення, які були надіслані користувачем, з яким ведеться чат.

- Компонент відображає фото профілю користувача, ім'я користувача, останнє повідомлення зі скороченим текстом, дату останнього повідомлення та позначки онлайн статусу.
- Якщо є непрочитані повідомлення від цього користувача, вони відображаються у вигляді числа в кружечку на синьому фоні (this-user-notifications).
- Якщо користувач-отримувач є онлайн, відображається зелена крапка (user-online).

У підсумку, цей компонент відображає окремий блок з інформацією про чат користувача, включаючи його фото, ім'я, останнє повідомлення, онлайн статус.

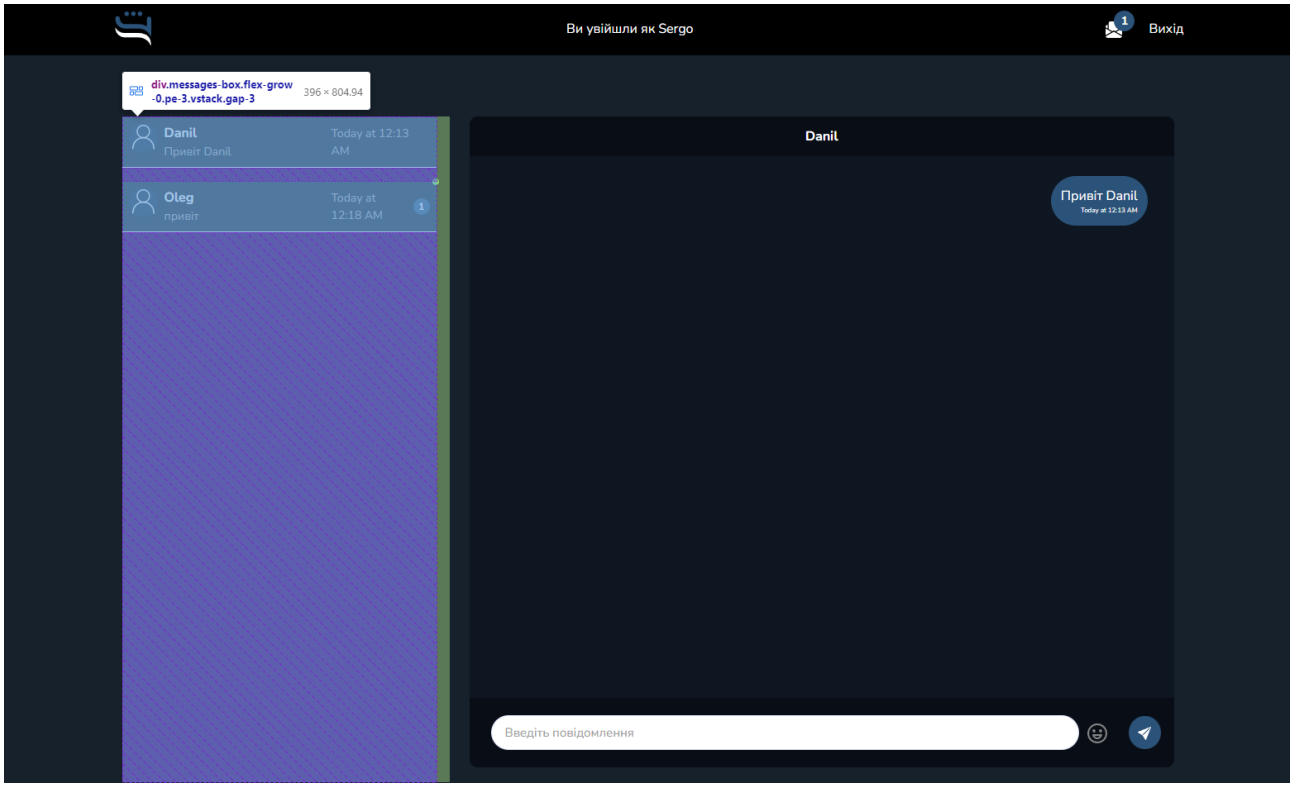

Рис.3.12 прямокутником виділено компонент UserChat

Компонент ChatBox:

- Цей компонент відповідає за відображення повідомлень у чаті з обраним користувачем.

- Використовується контекст AuthContext для отримання поточного користувача.
- Відображає список повідомлень у чаті з відміткою про відправника, текстом повідомлення та часом його надсилання.
- Використовується контекст ChatContext для отримання поточного чату, повідомлень, надсилання повідомлень.
- Містить поле введення повідомлення та кнопку надсилання.
- Автоматично прокручує до найновіших повідомлень при їх отриманні.
- Компонент має умовні відображення в залежності від наявності даних: якщо не вибрано жодної розмови, він відображає повідомлення "Ще не вибрано жодної розмови..."; якщо триває завантаження повідомлень, він відображає повідомлення "Завантаження чату...".
- При натисканні на кнопку "Надіслати", викликається функція sendTextMessage, яка передає текстове повідомлення, інформацію про поточного користувача та ідентифікатор поточного чату для відправлення повідомлення

Отже, цей компонент відображає чатове вікно, яке містить повідомлення між поточним користувачем та обраним користувачем у поточному чаті. Користувач може вводити повідомлення та надсилати його. Історія повідомлень відображається у вигляді списку, а нові повідомлення автоматично прокручуються до найновіших.

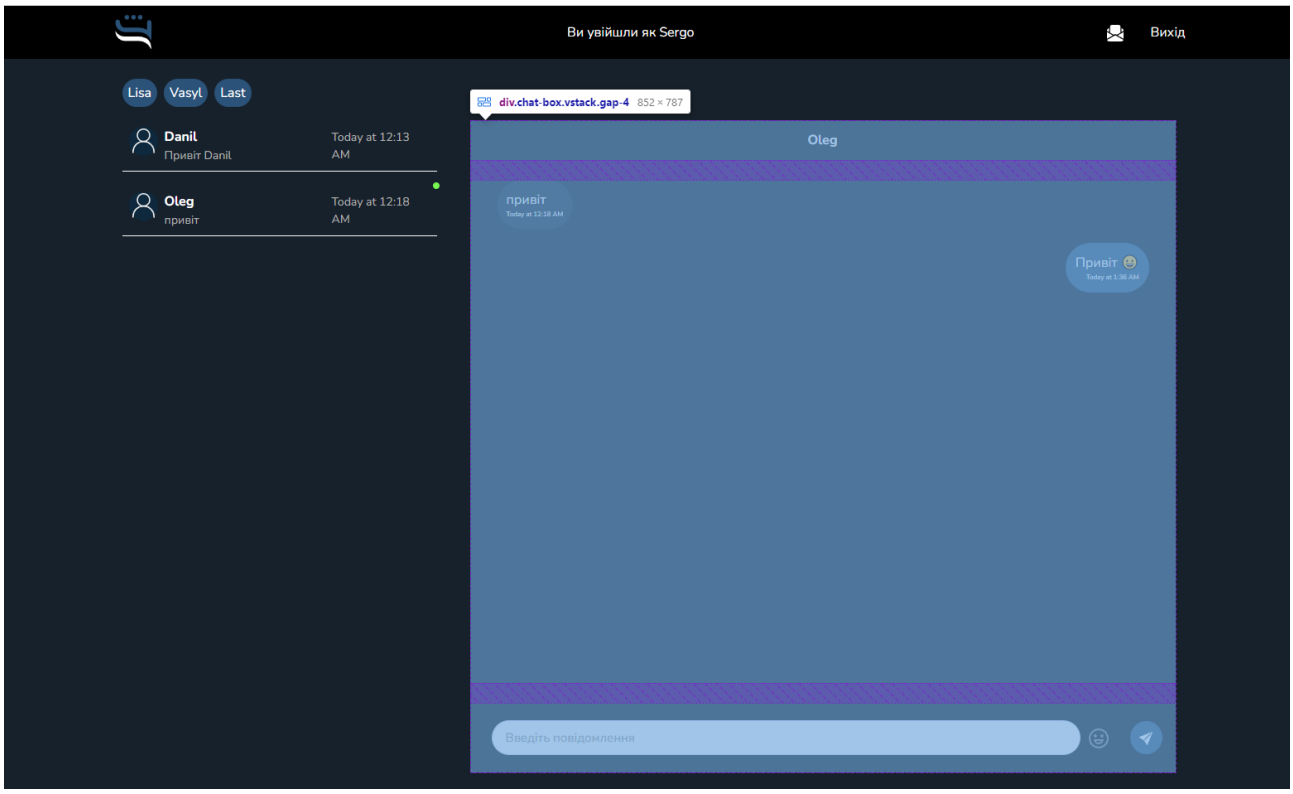

Рис.3.13 прямокутником виділено компонент ChatBox

Компонент Notifications відображає панель сповіщень рис 3.14, яка містить список непрочитаних повідомлень для користувача. Основна функціональність цього компонента включає наступне:

- Компонент отримує доступ до контексту ChatContext та AuthContext за допомогою хуків useContext. Він отримує дані про користувача, повідомлення, чати та всіх користувачів.
- За допомогою функції unreadNotificationsFunc з модуля unreadNotifications, відбувається обробка повідомлень і формування списку непрочитаних повідомлень.
- Повідомлення змінюються за допомогою функції map, де додається ім'я відправника до кожного повідомлення за допомогою allUsers.find(user => user._id === n.senderId). Змінені повідомлення зберігаються в змінну modifiedNotifications.
- Компонент містить іконку сповіщень, кількість непрочитаних повідомлень та блок сповіщень.

- При натисканні на іконку сповіщень панель сповіщень відкривається або закривається, залежно від значення isOpen.
- Якщо немає непрочитаних повідомлень (unreadNotifications.length === 0), панель сповіщень не відображається.
- Клік на "Прочитано" відзначає всі повідомлення як прочитані за допомогою функції markAllNotificationsAsRead.
- Компонент також відображає список повідомлень, де кожне повідомлення має назву відправника, дату та час надходження повідомлення.
- При кліку на повідомлення воно відзначається як прочитане, викликається функція `markNotificationAsRead` та закривається панель сповіщень (setIsOpen(false)).

Цей компонент забезпечує відображення сповіщень для користувача, можливість позначення їх як прочитаних та взаємодію з повідомленнями.

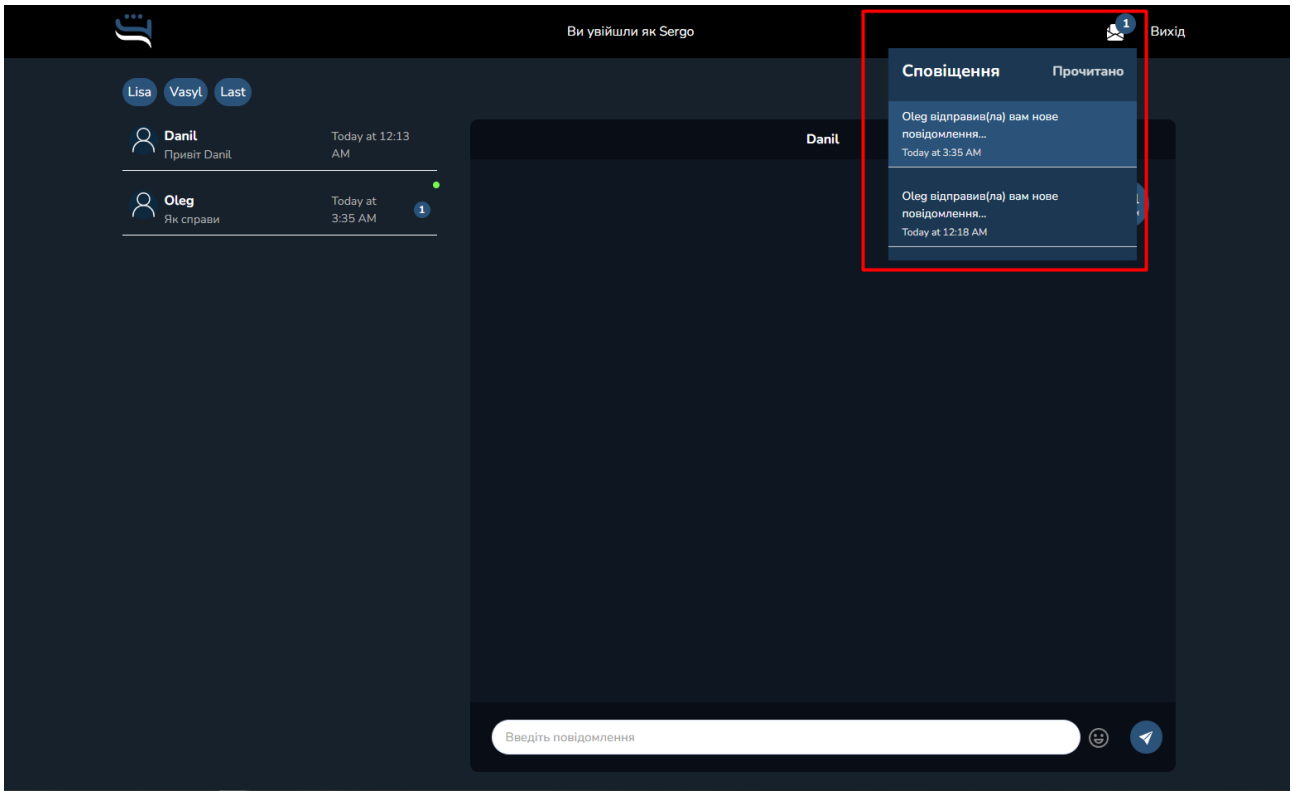

Рис.3.14 прямокутником виділено компонент Notifications

3.3 Розробка серверної частини

У компоненті userController використовується модель userModel, яка відповідає за взаємодію з колекцією користувачів у базі даних MongoDB. Додатково, використовуються пакети bcrypt, validator і jsonwebtoken для хешування паролів, перевірки даних та генерації JWT-токенів.

createToken - це функція, яка отримує ідентифікатор користувача і створює JWT-токен з використанням секретного ключа.

registerUser - це функція-контролер, яка отримує дані з запиту (name, email, password) і реєструє нового користувача на основі цих даних. Функція перевіряє, чи існує користувач з таким же email, чи всі поля заповнені, чи email є дійсним та, чи пароль є достатньо складним. Пароль хешується за допомогою bcrypt перед збереженням у базі даних. Після успішної реєстрації, функція створює JWT-токен за допомогою createToken і повертає об'єкт з ідентифікатором користувача, ім'ям, email та токеном у відповіді.

loginUser - це функція-контролер, яка отримує дані з запиту (email, password) і аутентифікує користувача. Функція шукає користувача за вказаним email, перевіряє правильність пароля з використанням bcrypt, і якщо аутентифікація успішна, створює JWT-токен за допомогою createToken і повертає об'єкт з ідентифікатором користувача, ім'ям, електронною поштою та токеном у відповіді.

findUser - це функція-контролер, яка отримує userId з параметрів запиту і шукає користувача з вказаним userId у базі даних. Після знаходження користувача, функція повертає об'єкт з інформацією про користувача у відповіді.

getUsers - це функція-контролер, яка отримує всіх користувачів з бази даних і повертає їх у відповіді.

Усі ці функції обробляють помилки, які можуть виникнути під час взаємодії з базою даних чи обробки запиту.

Модель userModel використовує схему userSchema, яка містить поля для зберігання даних користувача, такі як name, email та password.

Компонент userRoute відповідає за маршрутизацію пов’язану з користувачами в додатку.

Компонент chatController відповідає за обробку запитів, пов'язаних з чатами, і взаємодію з моделлю chatModel.

Функція createChat отримує дані відправника (senderId) і отримувача (receiverId) з запиту. Вона перевіряє, чи вже існує чат між цими користувачами. Якщо чат вже існує, повертає його. Якщо чат не існує, створює новий чат за допомогою моделі chatModel і зберігає його в базі даних. Повертає створений чат у відповідь.

Функція userChats отримує userId з параметрів запиту. Вона шукає всі чати, в яких користувач з userId є учасником. Повертає знайдені чати у відповідь.

Функція findChat отримує firstId(відправник) та secondId(отримувач) з параметрів запиту. Вона шукає чат, в якому обидва користувачі з firstId і secondId є учасниками. Повертає знайдений чат у відповідь.

Усі ці функції обробляють помилки, якщо вони виникають під час виконання запиту, і повертають відповідні статуси та повідомлення про помилку у відповідь.

Компонент chatController експортує ці функції, щоб їх можна було використовувати в інших частинах серверного додатка, наприклад, в маршрутах.

chatModel створює модель на основі схеми chatSchema. Вона надає доступ до методів, які дозволяють взаємодіяти Chat, таким як створення нового запису, пошук, оновлення.

Компонент chatRoure являє собою мартшутизатор, який визначає шляхи для обробки різних запитів, пов’язаних з чатами.

Компонент messageController відповідає за обробку повідомлень в додатку Спочатку, імпортується модель повідомлень (chatId, senderId, text).

Далі, оголошуються дві функції:

createMessage: Ця функція обробляє POST-запит для створення нового повідомлення. Вона отримує chatId, senderId і text з запиту, створює новий об'єкт повідомлення з цими даними, і зберігає його у базі даних. Якщо операція успішна, функція повертає статус 200 та створене повідомлення у форматі JSON. У разі помилки, повертається статус 500 і об'єкт помилки у форматі JSON.

getMessages: Ця функція обробляє GET-запит для отримання всіх повідомлень у певному чаті. Вона отримує chatId з параметрів запиту, шукає всі повідомлення з вказаним chatId за допомогою методу find(), і повертає їх у форматі JSON зі статусом 200. Якщо виникає помилка, повертається статус 500 і об'єкт помилки у форматі JSON.

Компонент messageRoute відповідає за маршрутизацію пов’язану з повідомленнями в додатку.

Важливою частиною серверної складови становить Socket.IO. Саме за допомогою його виконується надсилання та прийняття повідомлень в реальному часі та відстежування користувачів які знаходяться в мережі.

Спочатку, імпортується Server з пакета Socket.IO. Створюється об'єкт io із вказанням параметрів.

Визначається масив onlineUsers, який зберігатиме інформацію про користувачів, які знаходяться в онлайні.

addNewUser - обробляє запит на додавання нового користувача. Перевіряється, чи користувач вже присутній в onlineUsers, якщо ні, то його додаватиметься до масиву onlineUsers. Потім, емітується подія getOnlineUsers, щоб надіслати список онлайн користувачів всім підключеним клієнтам.

sendMessage - обробляє запит на відправлення повідомлення. Перевіряється, чи отримувач повідомлення знаходиться в onlineUsers. Якщо так, то повідомлення та повідомлення про сповіщення надсилаються до відповідного клієнта.

disconnect - обробляє подію відключення клієнта. Видаляє відповідного користувача з onlineUsers і емітує подію getOnlineUsers, щоб надіслати оновлений список онлайн користувачів.

3.4 Тестування

Перевірка функціонала веб месенджера. На рис 3.15 можна побачити користувача Vasyl, який зараз онлайн. При натисненні на цього користувача у нас з'являється новий чат рис 3.16, у користувача з яким ми створили чат також створюється чат. При натисненні на цей чат він відкривається в правому вікну рис 3.17.

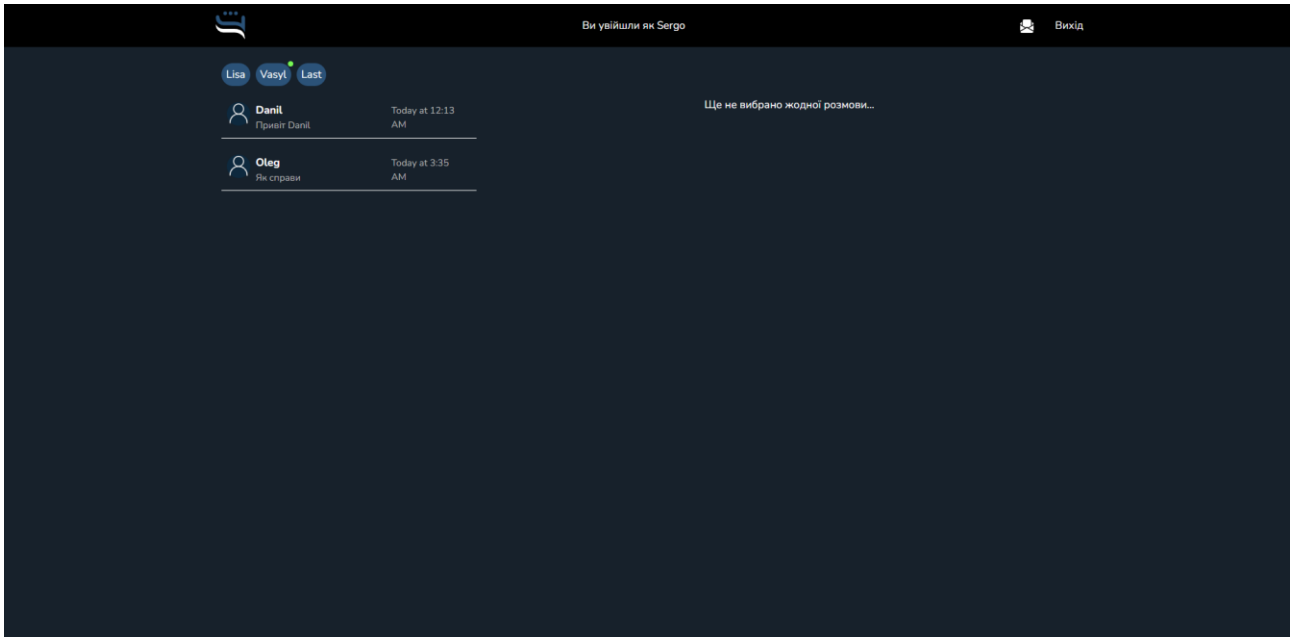

Рис. 3.15 створення нового чату з користувачем

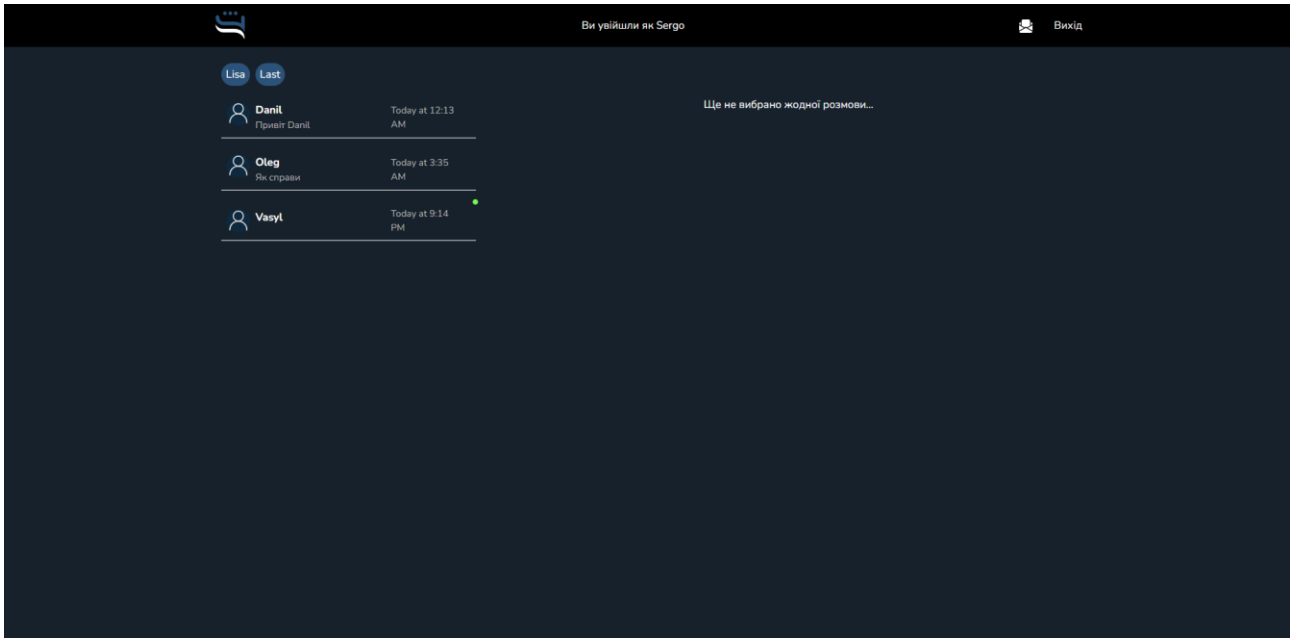

Рис. 3.16 створення нового чату з користувачем

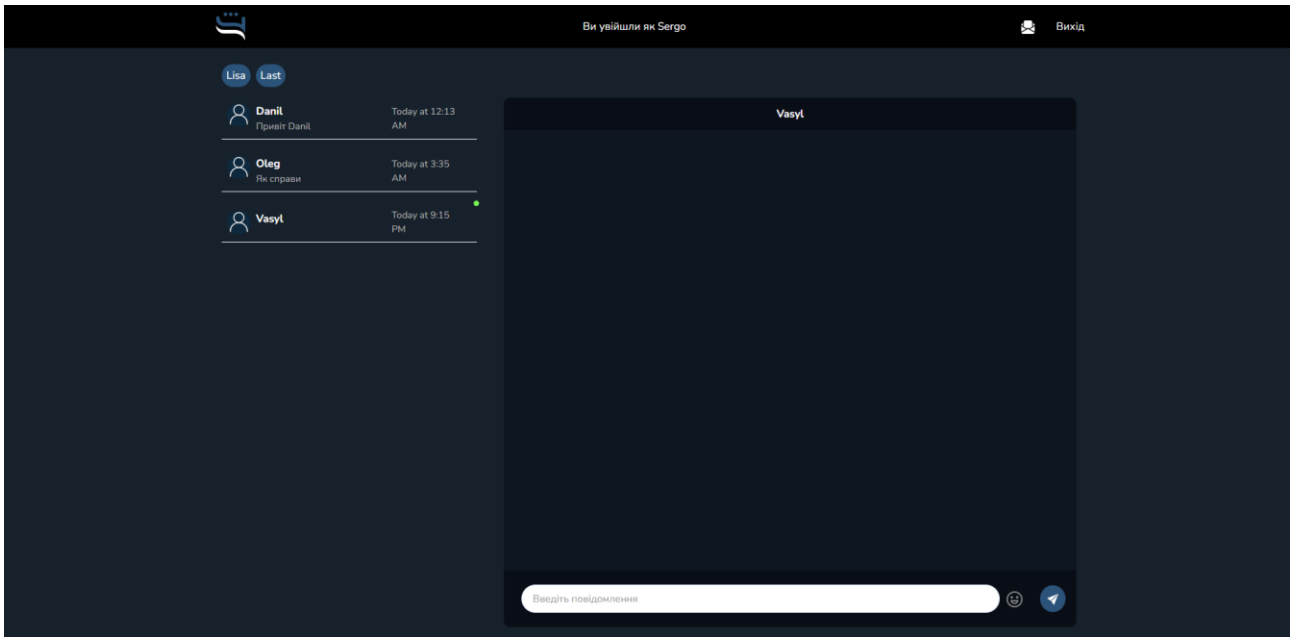

Рис. 3.17 відкриття чату

На рис 3.18 зображено початок спілкування користувачів, все працює добре, повідомлення приходять без затримки.

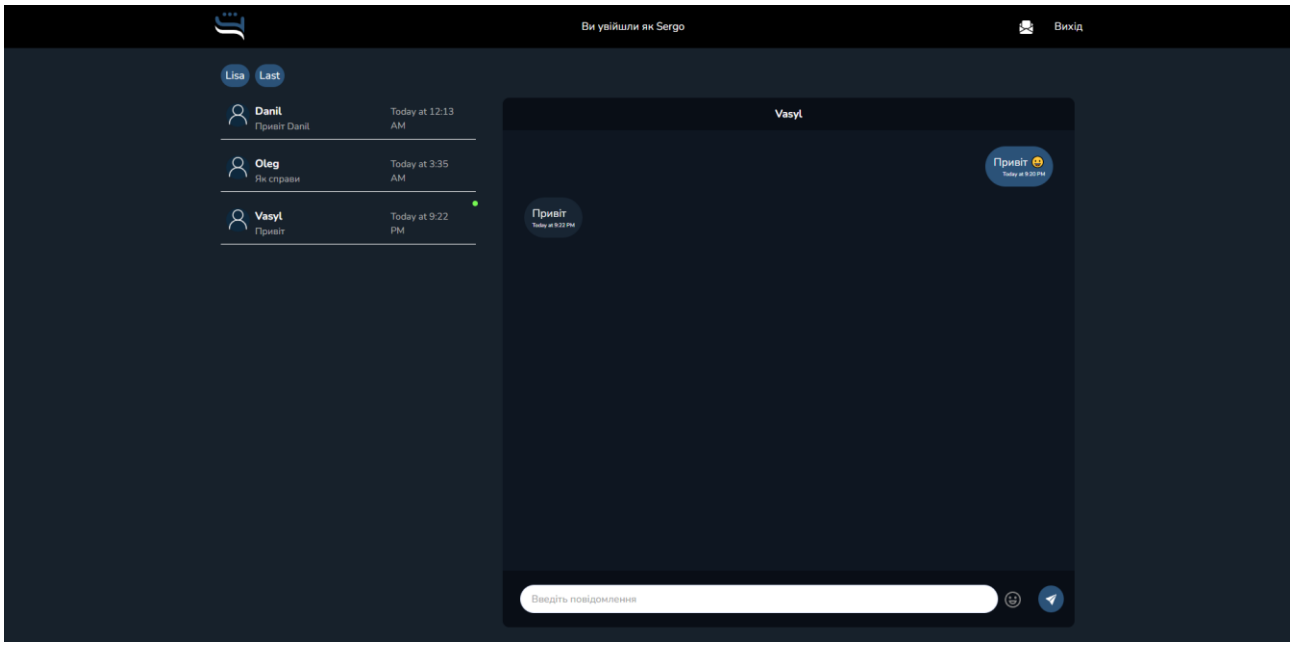

Рис. 3.18 надсилання та прийняття повідомлень

На рис 3.19 користувач Sergo перейшов в інший чат та отримав повідомлення від користувача Vasyl. Сповіщення про повідомлення відображається біля користувача Vasyl та у правому верхньому куті біля кнопки “Вихід”. Також біля користувача Vasyl відображається початок його

повідомлення, час відправлення та онлайн статус. На рис 3.20 користувач Oleg надіслав повідомлення, але вони не відобразились як сповіщення, бо чат з цим користувачем був відкритий. На рис 3.21 користувач Sergo перейшов в інший чат, а користувачі Oleg та Vasyl надіслали йому повідомлення. На рисунку видно кількість сповіщень біля кожного користувача та підсумовані сповіщення в правому верхньому куті програми.

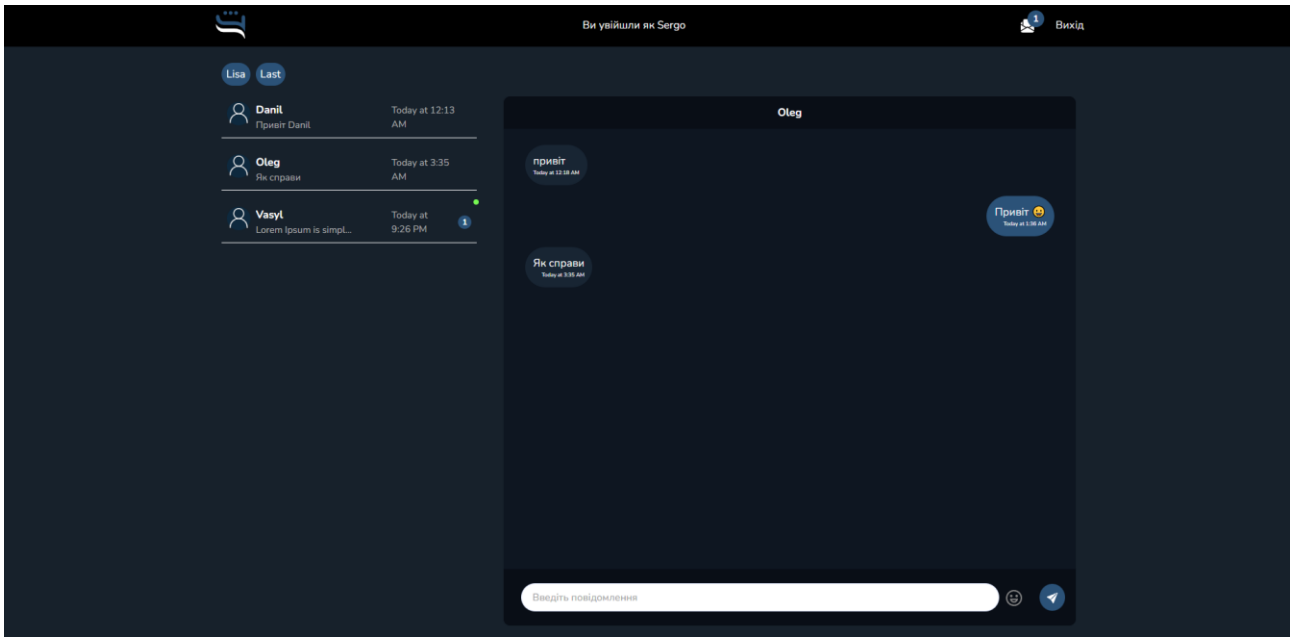

Рис. 3.19 Отримання сповіщень

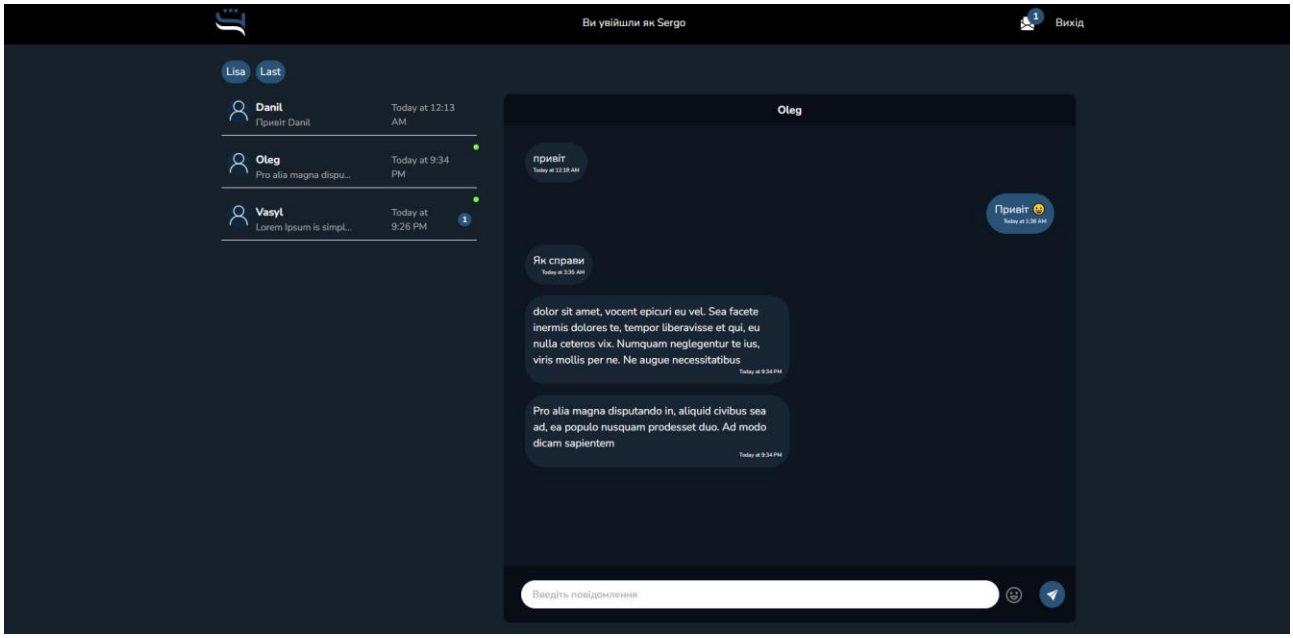

Рис. 3.20 Отримання сповіщень

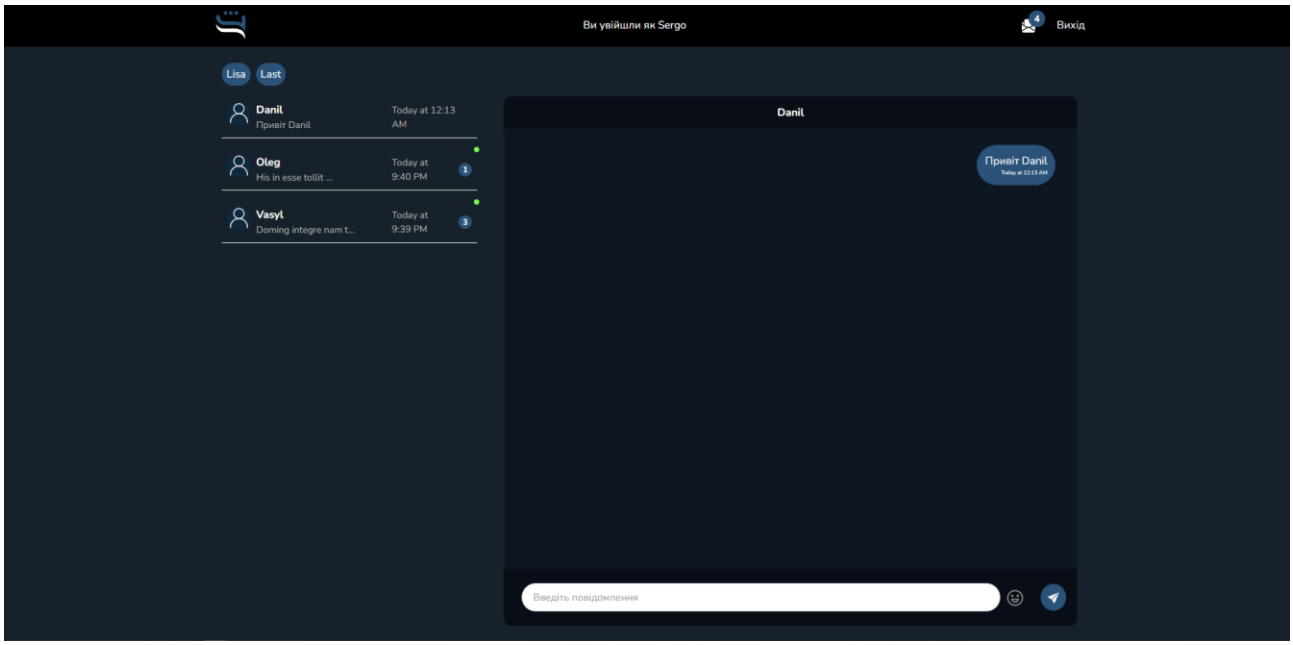

Рис. 3.20 Отримання сповіщень

При натисканні на користувача від якого маємо сповіщення, відкривається чат, пропадають сповіщення і чат прогортається до останнього повідомлення рис 3.21. Також сповіщення можна прочитати натиснувши на іконку повідомлення рис 3.22 та натиснути на кнопку “Прочитано” рис 3.23, або просто натиснути на повідомлення і відкриється чат з користувачем, від якого прийшло сповіщення.

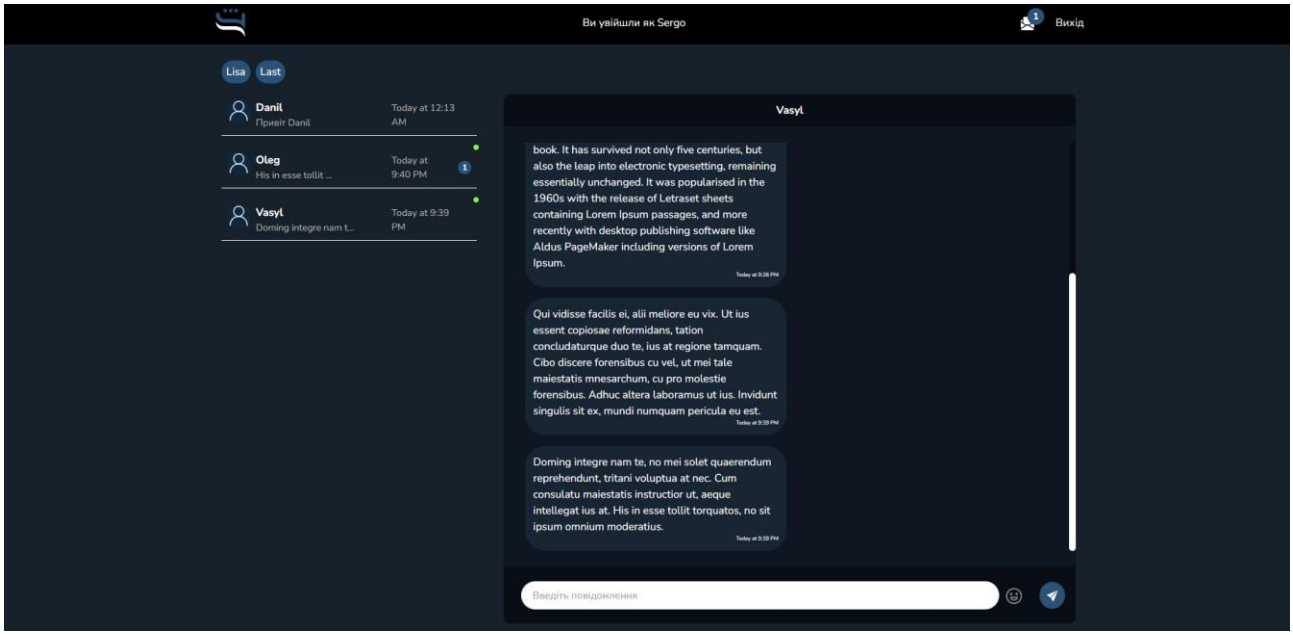

Рис. 3.21 Прочитування сповіщень

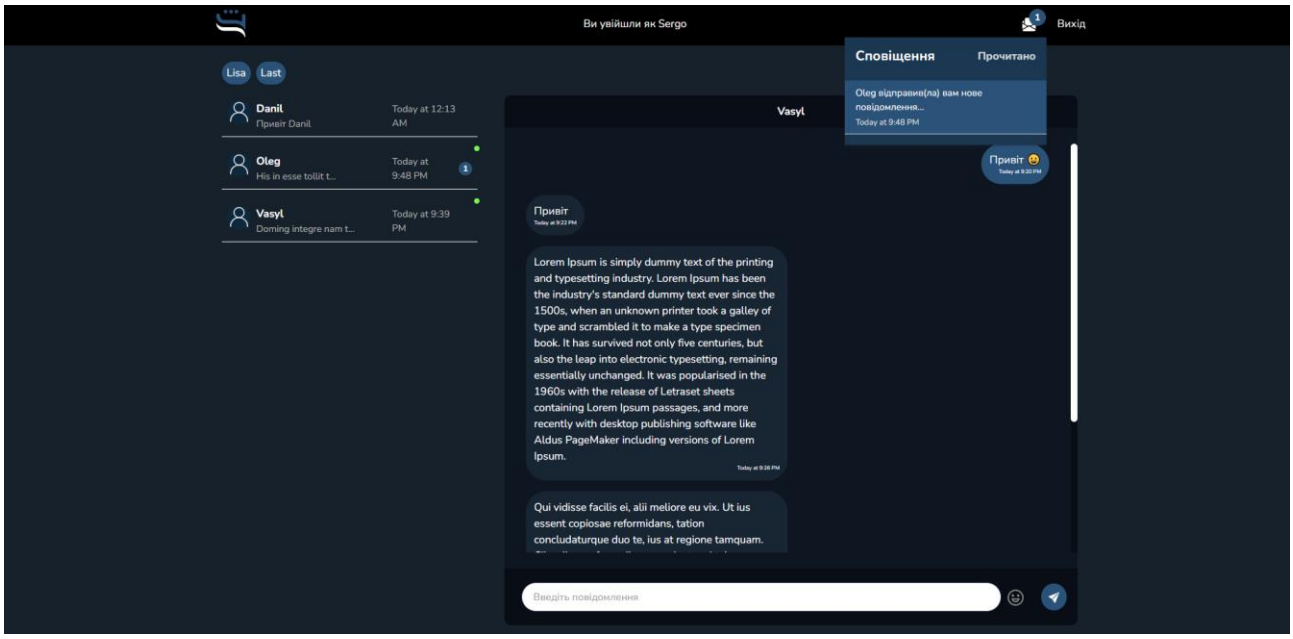

Рис. 3.22 Прочитування сповіщень

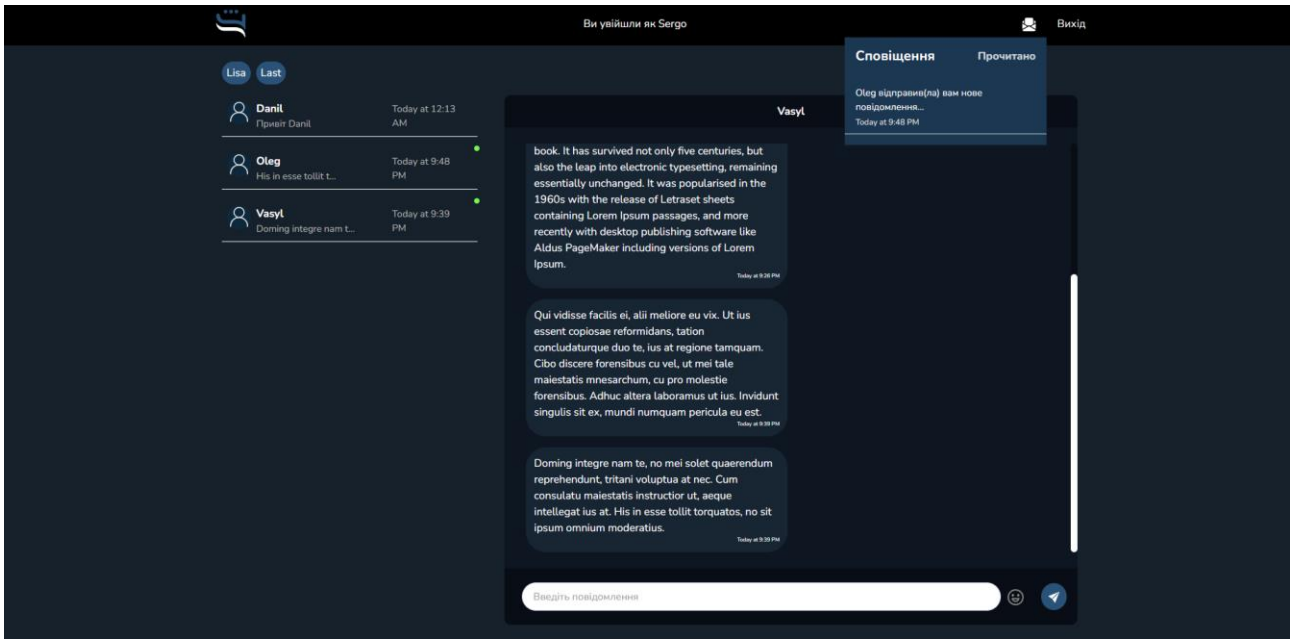

Рис. 3.23 Прочитування сповіщень

При тестуванні веб месенджера, всі реалізовані функції працюють правильно. Повідомлення приходять без затримки і без потреби оновлення сторінки, правильно відображається онлайн статус користувачів, приходять сповіщення та створюються нові чати.

ВИСНОВКИ

У цій курсовій роботі було розглянуто поняття месенджерів та їхню роль у сучасному суспільстві. Були визначені основні їх функції та переваги, а також проведено порівняння деяких сучасних месенджерів.

Для розробки веб месенджера було вибрано програмне середовище, фреймворки та бібліотеки. Розроблена клієнтська та серверна частини, виконане тестування на рівні користувача.

В результаті курсової роботи був розроблений веб месенджер, який дозволяє користувачам створювати облікові записи, входити в систему, обмінюватися повідомленнями та вести комунікацію з іншими користувачами. Розроблений месенджер має базовий функціонал, що включає реєстрацію, авторизацію, створення повідомлень, відображення активних користувачів та сповіщень. В подальшій роботі над месенджером можна реалізувати один з підходів шифрування повідомлень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Миттєві повідомлення [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/Миттєві_повідомлення
2. Документація Socket.IO [Електронний ресурс]. – Режим доступу: https://socket.io/docs/v4
3. Документація React [Електронний ресурс]. – Режим доступу: https://uk.legacy.reactjs.org/docs/getting-started.html#api-reference
4. Документація MongoDB [Електронний ресурс]. – Режим доступу: https://www.mongodb.com/docs/manual/
5. Про Node.js [Електронний ресурс]. – Режим доступу: https://nodejs.org/uk/about

ДОДАТОК А

Клієнтська частина коду

NavBar.jsx:

```
import { useContext } from "react";
import { Container, Nav, Navbar, Stack} from "react-bootstrap";
import { Link } from "react-router-dom";
import { AuthContext } from "../context/AuthContext";
import Notification from "./chat/Notification";
import logo from "../assets/logo.svg";


const NavBar = () => {
   const {user, logoutUser} = useContext(AuthContext);


   return <Navbar bg="black" className="mb-4" style={{height:"4rem"}}>
      <Container>
         <h2>
            <Link to="/" className="link-light text-decoration-none">
               <img src={logo} height="55px" />
            </Link>
         </h2>
         {user && < span>Ви увійшли як {user?.name}</span>}
         <Nav>
            <Stack direction="horizontal" gap={3}>
               {
                  user && (<>
                  <Notification/>
                  <Link onClick={() => logoutUser()} to="/login" className="link-light text-decoration-none">
                     Вихід
                  </Link>
                  </>)
               }
               {
                  !user && (<>
                  <Link to="/login" className="link-light text-decoration-none">
                     Вхід
                  </Link>
                  <Link to="/register" className="link-light text-decoration-none">
                     Реєстрація
                  </Link>
               </>)
               }

            </Stack>
         </Nav>
      </Container>
   </Navbar>
};

export default NavBar;
```

ChatBox.jsx:

```
import { useContext, useEffect, useRef, useState } from "react";
import { AuthContext } from "../../context/AuthContext";
import { ChatContext } from "../../context/ChatContext";
import { useFetchRecipientUser } from "../../hooks/useFetchRecipient";
```

```
import { Stack } from "react-bootstrap";
import moment from "moment";
import InputEmoji from "react-input-emoji";

const ChatBox = () => {
    const { user } = useContext(AuthContext);
    const { currentChat, messages, sendTextMessage, isMessagesLoading } =
      useContext(ChatContext);
    const { recipientUser } = useFetchRecipientUser(currentChat, user);
    const [textMessage, setTextMessage] = useState("");
    const scroll = useRef();

    useEffect(() => {
      scroll.current?.scrollIntoView({ behavior: "smooth" });
    }, [messages]);

    if (!recipientUser)
      return (
        <p style={{ textAlign: "center", width: "100%" }}>
          Ще не вибрано жодної розмови...
        </p>
      );
    if (isMessagesLoading)
      return (
        <p style={{ textAlign: "center", width: "100%" }}>Завантаження чату...</p>
      );
    return (
      <Stack gap={4} className="chat-box">
        <div className="chat-header">
          <strong>{recipientUser?.name}</strong>
        </div>
        <Stack gap={3} className="messages" id="element">
          {messages &&
            messages?.map((message, index) => (
              <Stack
                className={`${
                  message?.senderId === user?._id
                    ? "message self align-self-end flex-grow-0"
                    : "message align-self-start flex-grow-0"
                }`}
                key={index}
                ref={scroll}
              >
                <span>{message.text}</span>
                <span className="message-footer">
                  {moment(message.createdAt).calendar()}
                </span>
              </Stack>
            ))}
        </Stack>
        <Stack  direction="horizontal" className="chat-input flex-grow-0" gap={3}>
          <InputEmoji
            value={textMessage}
            onChange={setTextMessage}
            fontFamily="nunito"
            borderColor="rgba(72, 112, 223, 0.2)"
            placeholder="Введіть повідомлення"
          />
          <button
            className="send-btn"
            onClick={() =>
              sendTextMessage(textMessage, user, currentChat._id, setTextMessage)
            }
          >
```

```
          <svg
            xmlns="http://www.w3.org/2000/svg"
            width="16"
            height="16"
            fill="currentColor"
            className="bi bi-send-fill"
            viewBox="0 0 16 16"
          >
            <path d="M15.964.686a.5.5 0 0 0-.65-.65L.767 5.855H.766l-.452.18a.5.5 0 0 0-.082.887l.41.26.001.002 4.995
3.178 3.178 4.995.002.002.26.41a.5.5 0 0 0 .886-.083l6-15Zm-1.833 1.89L6.637 10.07l-.215-.338a.5.5 0 0 0-.154-.154l-
.338-.215 7.494-7.494 1.178-.471-.47 1.178Z" />
          </svg>
        </button>
      </Stack>
    </Stack>
  );
};
export default ChatBox;
```

Notification.jsx:

```
import { useContext, useState } from "react";
import { ChatContext } from "../../context/ChatContext";
import { AuthContext } from "../../context/AuthContext";
import { unreadNotificationsFunc } from "../../utils/unreadNotifications";
import moment from "moment";

const Notification = () => {
   const [isOpen, setIsOpen] = useState(false);
   const { user } = useContext(AuthContext);
   const {notifications, userChats, allUsers, markAllNotificationsAsRead, markNotificationAsRead} =
useContext(ChatContext);

   const unreadNotifications = unreadNotificationsFunc(notifications);
   const modifiedNotifications = notifications.map((n) => {
      const sender = allUsers.find(user => user._id === n.senderId );

      return{
         ...n,
         senderName: sender?.name,
      };
   });

   console.log("un", unreadNotifications);
   console.log("mn", modifiedNotifications);

   return (
      <div className="notifications">
         <div className="notifications-icon" onClick={() => setIsOpen(!isOpen)}>
            <svg xmlns="http://www.w3.org/2000/svg" width="20" height="20" fill="currentColor" className="bi bi-
envelope-paper-fill" viewBox="0 0 16 16">
               <path fill-rule="evenodd" d="M6.5 9.5 3 7.5v-6A1.5 1.5 0 0 1 4.5 0h7A1.5 1.5 0 0 1 13 1.5v6l-3.5 2L8 8.75l-
1.5.75ZM1.059 3.635 2 3.133v3.753L0 5.713V5.4a2 2 0 0 1 1.059-1.765ZM16 5.713l-2 1.173V3.133l.941.502A2 2 0 0
1 16 5.4v.313Zm0 1.16-5.693 3.337L16 13.372v-6.5Zm-8 3.199 7.941 4.412A2 2 0 0 1 14 16H2a2 2 0 0 1-1.941-1.516L8
10.072Zm-8 3.3 5.693-3.162L0 6.873v6.5Z" />
            </svg>
            {unreadNotifications?.length === 0 ? null : (
               <span className="notification-count">
                  <span>{unreadNotifications?.length}</span>
               </span>
            )}
         </div>
```

```
    {isOpen ? (
      <div className="notifications-box">
        <div className="notifications-header">
          <h3>Сповіщення</h3>
          <div
            className="mark-as-read"
            onClick={() => markAllNotificationsAsRead(notifications) }

          >
            Прочитано
          </div>
        </div>
        {modifiedNotifications?.length === 0 ? (
          <span className="notification">Сповіщень не знайдено...</span>
        ) : null}
        {modifiedNotifications &&
          modifiedNotifications.map((n, index) => (
            <div
              key={index}
              className={n.isRead ? `notification` : `notification not-read`}
              onClick={() => {
                markNotificationAsRead(n, userChats, user, notifications);
                setIsOpen(false);
              }}
            >
              <span>{`${n.senderName} відправив(ла) вам нове повідомлення...`}</span>
              <span className="notification-time">
                {moment(n.date).calendar()}
              </span>
            </div>
          ))}
      </div>
    ) : null}

      </div>

  );
};

export default Notification;
```

PotentialChats.jsx:

```
import { useContext } from "react";
import { ChatContext } from "../../context/ChatContext";
import { AuthContext } from "../../context/AuthContext";

const PotentialChats = () => {
   const {user} = useContext(AuthContext)
   const {potentialChats, createChat, onlineUsers  } = useContext(ChatContext);

   // console.log("pchats", potentialChats);
   return (<>
      <div className="all-users">
         {potentialChats && potentialChats.map((u, index) => {
            return (
               <div
                  className="single-user"
                  key={index}
                  onClick={() => createChat(user._id, u._id)}
               >
```

```
                {u.name}
                <span className={
                    onlineUsers?.some((user) => user?.userId === u?._id) ? "user-online" : ""}></span>
            </div>
        );

    })}
  </div>
</>
);
};

export default PotentialChats;
```

UserChat.jsx:

```
import { Stack } from "react-bootstrap";
import { useFetchRecipientUser } from "../../hooks/useFetchRecipient";
import avatar from "../../assets/avarter.svg"
import { useContext } from "react";
import { ChatContext } from "../../context/ChatContext";
import { unreadNotificationsFunc } from "../../utils/unreadNotifications";
import { useFecthLatestMessage } from "../../hooks/useFetchLatestMessages";
import moment from "moment";

const UserChat = ({chat, user}) => {
  const {recipientUser} = useFetchRecipientUser(chat, user);
  const {onlineUsers, notifications, markThisUserNotificationsAsRead} = useContext(ChatContext);
  const {latestMessage} = useFecthLatestMessage(chat);


  const unreadNotifications = unreadNotificationsFunc(notifications);
  const thisUserNotifications = unreadNotifications?.filter(
    n => n.senderId == recipientUser?._id
  )

  const isOnline = onlineUsers?.some((user) => user?.userId === recipientUser?._id);

  const truncateText = (text) =>{
    let shortText = text.substring(0, 20);

    if(text.length > 20){
      shortText = shortText + "...";
    }

    return  shortText;
  };

  return (
  <Stack
  direction="horizontal"
  gap={3}
  className="user-card align-items-center p-2 justify-content-between "
  role = "button"
  onClick={() =>{
    if(thisUserNotifications?.length !== 0){
      markThisUserNotificationsAsRead(
        thisUserNotifications,
        notifications
      );
    }
  }}
  >
```

```
        <div className="d-flex">
          <div className="me-2">
            <img src={avatar} height="35px"/>
          </div>
          <div className="text-content">
            <div className="name">{recipientUser?.name}</div>
            <div className="text">{
              latestMessage?.text && (
                <span>{truncateText(latestMessage?.text)}</span>
              )
            }</div>
          </div>
        </div>
        <div className="d-flex flex-column align-items-end"></div>
        <div className="date">{moment(latestMessage?.createdAt).calendar()}
        </div>
        <div>
          <div className= {thisUserNotifications?.length > 0 ? "this-user-notifications" : ""}>
            {thisUserNotifications?.length > 0 ? thisUserNotifications?.length : "" }
          </div>
          <span className={isOnline ? "user-online": "" }></span>
        </div>
    </Stack>
    );
};

export default UserChat;
```

context/AuthContext.jsx:

```
import { createContext, useCallback, useEffect, useState } from "react";
import { baseUrl, postRequest } from "../utils/services";

export const AuthContext = createContext();
export const AuthContextProvider = ({ children }) => {
    const [user, setUser] = useState(null);
    const [registerError, setRegisterError] = useState(null);
    const [isRegisterLoading, setIsRegisterLoading] = useState(false);
    const [registerInfo, setRegisterInfo] = useState({
      name: "",
      email: "",
      password: "",
    });
    const [loginError, setLoginError] = useState(null);
    const [isLoginLoading, setIsLoginLoading] = useState(false);
    const [loginInfo, setLoginInfo] = useState({
      email: "",
      password: "",
    });

    console.log("Userr:", user);
    console.log("registerError:", registerError);
    console.log("isRegisterLoading:", isRegisterLoading);
    console.log("loginError:", loginError);
    console.log("isLoginLoading:", isLoginLoading);

    useEffect(() => {
      const user = localStorage.getItem("User");

      setUser(JSON.parse(user));
    }, []);

    const updateRegisterInfo = useCallback((info) => {
      setRegisterInfo(info);
```

```
}, []);

const updateLoginInfo = useCallback((info) => {
  setLoginInfo(info);
}, []);

const registerUser = useCallback(
  async (e) => {
    e.preventDefault();

    setIsRegisterLoading(true);
    setRegisterError(null);

    const response = await postRequest(
      `${baseUrl}/users/register`,
      JSON.stringify(registerInfo)
    );

    setIsRegisterLoading(false);

    if (response.error) {
      return setRegisterError(response);
    }

    localStorage.setItem("User", JSON.stringify(response));
    setUser(response);
  },
  [registerInfo]
);

const loginUser = useCallback(
  async (e) => {
    e.preventDefault();

    setIsLoginLoading(true);
    setLoginError(null);

    const response = await postRequest(
      `${baseUrl}/users/login`,
      JSON.stringify(loginInfo)
    );

    setIsLoginLoading(false);

    if (response.error) {
      return setLoginError(response);
    }

    localStorage.setItem("User", JSON.stringify(response));
    setUser(response);
  },
  [loginInfo]
);

const logoutUser = useCallback(() => {
  localStorage.removeItem("User");
  setUser(null);
}, []);

return (
  <AuthContext.Provider
    value={{
      user,
      registerUser,
```

```
        loginUser,
        registerInfo,
        updateRegisterInfo,
        loginInfo,
        updateLoginInfo,
        loginError,
        isLoginLoading,
        registerError,
        isRegisterLoading,
        logoutUser,
      }}
    >
      {children}
    </AuthContext.Provider>
  );
};
```

context/ChatContext.jsx:

```
import { createContext } from "react";
import { useState, useEffect, useCallback } from "react";
import { baseUrl, getRequest, postRequest } from "../utils/services";
import { io } from "socket.io-client";
// import { async } from "react-input-emoji";

export const ChatContext = createContext();

export const ChatContextProvider = ({ children, user }) =>{
    const [userChats, setUserChats] = useState(null);
    const [isUserChatsLoading, setIsUserChatsLoading] = useState(false);
    const [userChatsError, setUserChatsError] = useState(null);
    const [potentialChats, setPotentialChats] = useState([]);
    const [currentChat, setCurrentChat] = useState(null);
    const [messages, setMessages] = useState(null);
    const [messagesError, setMessagesError] = useState(null);
    const [isMessagesLoading, setIsMessagesLoading] = useState(false);
    const [sendTextMessageError, setSendTextMessageError] = useState(null);
    const [newMessage, setNewMessage] = useState(null);
    const [socket, setSocket] = useState(null);
    const [onlineUsers, setOnlineUsers] = useState([]);
    const [notifications, setNotifications] = useState([]);
    const[allUsers, setAllUsers] = useState([]);

    // console.log("messages", messages);
    console.log("onlineUsers", onlineUsers);
    // console.log("Notifications", notifications);

    //встановлення сокету

    useEffect (() => {
        const newSocket = io("http://localhost:3377");
        setSocket (newSocket);

        return () =>{
            newSocket.disconnect();
        };
    }, [user]);


    // добавлення юзерів онлайн
    useEffect(() => {
        if(socket === null ) return
        socket.emit("addNewUser", user?._id);
        socket.on("getOnlineUsers", (res) => {
```

```
        setOnlineUsers(res);
    });

    return () => {
        socket.off("getOnlineUsers");
    };
  }, [socket] );

  // відправлення повідомлень
  useEffect(() => {
    if(socket === null ) return;

    const recipientId = currentChat?.members.find((id) => id !== user?._id);

    socket.emit("sendMessage", {...newMessage, recipientId});
  }, [newMessage] );

  //отримання повідомлень і оповіщень

useEffect(() => {
  if (socket === null) return;

  socket.on("getMessage", (res) => {
    if (currentChat?._id !== res.chatId) return;

    setMessages((prev) => [...prev, res]);
  });

  socket.on("getNotification", (res) => {
    const isChatOpen = currentChat?.members.some((Id) => Id === res.senderId);

    if (isChatOpen) {
      setNotifications((prev) => [{ ...res, isRead: true }, ...prev]);
    } else {
      setNotifications((prev) => [res, ...prev]);
    }
  });

  return () => {
    socket.off("getMessage");
    socket.off("getNotification");
  };
}, [socket, currentChat]);



  useEffect(() => {

    const getUsers = async () =>{
        const response = await getRequest(`${baseUrl}/users`);

        if(response.error){
            return console.log("Error fetching users", response);
        }
        const pChats = response.filter((u) =>{
            let isChatCreated = false;
            if(user?._id === u._id) return false;

            if(userChats){
                isChatCreated = userChats?.some((chat) =>{
                    return chat.members[0] === u._id || chat.members[1] === u._id;
                });
            };
```

```
            return !isChatCreated;

        });
        setPotentialChats(pChats);
        setAllUsers(response);

    };

    getUsers();

},[userChats]);

    useEffect(() => {
        const getUserChats = async() =>{
            if(user?._id){

                setIsUserChatsLoading(true);
                setUserChatsError(null);

                const response = await getRequest(`${baseUrl}/chats/${user?._id}`);

                setIsUserChatsLoading(false);

                if(response.error){
                    return setUserChatsError(response);
                }

                setUserChats(response);
            }
        };
         getUserChats();
    }, [user, notifications]);


    useEffect(() => {
        const getMessages = async() =>{


                setIsMessagesLoading(true);
                setMessagesError(null);

                const response = await getRequest(`${baseUrl}/messages/${currentChat?._id}`);

                setIsMessagesLoading(false);

                if(response.error){
                    return setMessagesError(response);
                }

                setMessages(response);
        };
        getMessages();
    }, [currentChat]);

    const sendTextMessage = useCallback( async (textMessage, sender, currentChatId, setTextMessage) =>{
        if (!textMessage) return console.log("You type something...");

        const response = await postRequest(`${baseUrl}/messages`, JSON.stringify({
            chatId: currentChatId,
            senderId: sender._id,
            text: textMessage
        }));

        if(response.error){
```

```
            return setSendTextMessageError(response);
        }

        setNewMessage(response);
        setMessages((prev) => [...prev, response]);
        setTextMessage("");

    }, []);


    const updateCurrentChat = useCallback((chat) => {
        setCurrentChat(chat);
    },[]);

    const createChat = useCallback( async (senderId, receiverId)=>{
        const response = await postRequest(
            `${baseUrl}/chats/`,
            JSON.stringify({
            senderId,
            receiverId
        })
        );
        if(response.error){
            return console.log("Error creating chat", response);
        }
        setUserChats((prev) => [...prev, response]);
    },[]);

const markAllNotificationsAsRead = useCallback((notifications) => {
    const mNotifications = notifications.map(n =>
        {return {...n, isRead: true};
     });

     setNotifications(mNotifications);
}, []);

const markNotificationAsRead = useCallback ((n, userChats, user, notifications) =>{
    //пошук відкритого чату
    const desiredChat = userChats.find(chat =>{
        const chatMembers =[user._id , n.senderId];
        const isDesiredChat = chat?.members.every((member) => {
            return chatMembers.includes(member);
        });

        return isDesiredChat;
    });
    //позначити прочитаними сповіщення

    const mNotifications = notifications.map(el =>{
        if(n.senderId === el.senderId){
            return {...n, isRead: true};
        }else{
            return el;
        }

    });

    updateCurrentChat(desiredChat);
    setNotifications(mNotifications);
}, [] );

const markThisUserNotificationsAsRead = useCallback ((thisUserNotifications, notification) => {
```

```
      const mNotifications = notification.map(el => {
        let notification;

        thisUserNotifications.forEach(n => {
          if(n.senderId === el.senderId){
            notification = {...n, isRead: true};
          }else{
            notification = el;
          }
        });

        return notification;
      });

      setNotifications(mNotifications);
    }, []);

    return (
    <ChatContext.Provider
     value={{
      userChats,
      isUserChatsLoading,
      userChatsError,
      potentialChats,
      createChat,
      updateCurrentChat,
      messages,
      isMessagesLoading,
      messagesError,
      currentChat,
      sendTextMessage,
      onlineUsers,
      notifications,
      allUsers,
      markAllNotificationsAsRead,
      markNotificationAsRead,
      markThisUserNotificationsAsRead,


    }}
    >
      {children}
    </ChatContext.Provider>
    );
};
```

hooks/useFetchLatestMessages.js:

```
import { useContext, useEffect, useState } from "react";
import { ChatContext } from "../context/ChatContext";
import { baseUrl, getRequest } from "../utils/services";

export const useFecthLatestMessage = (chat) => {
  const { newMessage, notifications } = useContext(ChatContext);
  const [latestMessage, setLatestMessage] = useState(null);

  useEffect(() => {
    const getMessages = async () => {
      const response = await getRequest(`${baseUrl}/messages/${chat?._id}`);

      if (response.error) {
        return console.log("Error getting messages...", error);
      }
```

```
      const lastMessage = response[response?.length - 1];

      setLatestMessage(lastMessage);
    };
    getMessages();
  }, [newMessage, notifications]);

  return { latestMessage };
};
```

hooks/useFetchRecipient.js:

```
import { useEffect, useState } from "react";
import { baseUrl, getRequest } from "../utils/services";
export const useFetchRecipientUser = (chat, user) => {
    const [recipientUser, setRecipientUser] = useState(null);
    const [error, setError] = useState(null);

    const recipientId = chat?.members.find((id) => id !== user?._id);

    useEffect(() => {
      const getRecipientUser = async () => {
        if (!recipientId) return null;

        const response = await getRequest(`${baseUrl}/users/find/${recipientId}`);

        if (response.error) {
          return setError(error);
        }

        setRecipientUser(response);
      };
      getRecipientUser();
    }, [recipientId]);

    return { recipientUser };
  };
```

pages/Chat.jsx:

```
import { useContext } from "react";
import { Container, Stack } from "react-bootstrap";
import ChatBox from "../components/chat/ChatBox";
import { AuthContext } from "../context/AuthContext";
import { ChatContext } from "../context/ChatContext";
import UserChat from "../components/chat/UserChat";
import PotentialChats from "../components/chat/PotentialChats";

const Chat = () => {
    const {user}= useContext(AuthContext);

    const {userChats,isUserChatsLoading,updateCurrentChat }= useContext(ChatContext);

    return ( <Container>
        <PotentialChats/>
        {userChats?.lenth < 1 ? null : (
        <Stack direction="horizontal" gap={4} className="align-items-start">
            <Stack className="messages-box flex-grow-0 pe-3" gap={3}>
                {isUserChatsLoading && <p>Завантаження чатів...</p>}
                {userChats?.map((chat,index) => {
                    return(
                        <div key={index} onClick={() => updateCurrentChat(chat)}>
                            <UserChat chat={chat} user={user}/>
```

```
                </div>
              );
            })}
          </Stack>
          <ChatBox />
        </Stack>
      )}
    </Container>
  );
};

export default Chat;
```

pages/Login.jsx:

```
import { useContext } from "react";
import { Alert, Button, Col, Form, Row, Stack } from "react-bootstrap";
import { AuthContext } from "../context/AuthContext";

const Login = () => {
  const { loginInfo, updateLoginInfo, loginUser, loginError, isLoginLoading } =
    useContext(AuthContext);

  return (
    <>
      <Form onSubmit={loginUser}>
        <Row
          style={{
            height: "100vh",
            justifyContent: "center",
            paddingTop: "20%",
          }}
        >
          <Col xs={6}>
            <Stack gap={3}>
              <h2>Вхід</h2>

              <Form.Control
                type="email"
                placeholder="Email"
                onChange={(e) =>
                  updateLoginInfo({ ...loginInfo, email: e.target.value })
                }
                value={loginInfo.email}
              />
              <Form.Control
                type="password"
                placeholder="Пароль"
                onChange={(e) =>
                  updateLoginInfo({
                    ...loginInfo,
                    password: e.target.value,
                  })
                }
                value={loginInfo.password}
              />
              <Button  variant="primary" type="submit" disabled={isLoginLoading}>
                {isLoginLoading ? "Виконується..." : "Вхід"}
              </Button>

              {loginError?.error && (
                <Alert variant="danger">

                  <p>{loginError?.message}</p>
```

```
            </Alert>
          )}
        </Stack>
      </Col>
    </Row>
  </Form>
 </>
);
};

export default Login;
```

pages/Register.jsx:

```
import { useContext } from "react";
import {Alert, Button, Form, Row, Col, Stack, FormControl} from "react-bootstrap";
import { AuthContext } from "../context/AuthContext";

const Register = () => {

  const {registerInfo, updateRegisterInfo, registerUser, registerError, isRegisterLoading} = useContext(AuthContext);

  return <>
  <Form onSubmit={registerUser}>
    <Row style={{
      height: "100vh",
      justifyContent:"center",
      paddingTop: "10%"
    }}>
      <Col xs={5}>
      <Stack gap={3}>
        <h2>Реєстрація</h2>

        <FormControl
          type="text"
          placeholder="Ім'я"
          onChange={(e) =>
            updateRegisterInfo({...registerInfo, name: e.target.value})
          }
        />
        <FormControl
          type="email"
          placeholder="Email"
          onChange={(e) =>
            updateRegisterInfo({...registerInfo, email: e.target.value})
          }
        />
        <FormControl
          type="password"
          placeholder="Пароль"
          onChange={(e) =>
            updateRegisterInfo({...registerInfo, password: e.target.value})
          }
        />
        <Button variant="primary" type="submit">
          {isRegisterLoading ? "Створення акаунту" : "Зареєструвати"}
        </Button>
          {
            registerError?.error && <Alert variant="danger">
            <p>{registerError?.message}</p>
          </Alert>
          }
```

```
          </Stack>
          </Col>
        </Row>
    </Form>
    </>;
}

export default Register;
```

App.jsx:

```
import{Routes, Route, Navigate} from "react-router-dom";
import Chat from "./pages/Chat";
import Register from "./pages/Register";
import Login from "./pages/Login";
import "bootstrap/dist/css/bootstrap.min.css";
import {Container} from "react-bootstrap"
import NavBar from "./components/NavBar";
import { useContext } from "react";
import { AuthContext } from "./context/AuthContext";
import { ChatContextProvider } from "./context/ChatContext";

function App() {
  const {user} = useContext(AuthContext);
 return (
  <>
  <ChatContextProvider user= {user}>
  <NavBar />
  <Container>
    <Routes>
      <Route path="/" element = {user ? <Chat /> : <Login/>} />
      <Route path="/register" element = {user ? <Chat /> : <Register/>} />
      <Route path="/login" element = {user ? <Chat /> : <Login/>} />
      <Route path="*" element = {<Navigate to="/"/>}/>
    </Routes>
  </Container>
  </ChatContextProvider>
  </>
 );
}


export default App;
```

index.css:

```
@import
url("https://fonts.googleapis.com/css2?family=Nunito:
wght@400;700&display=swap");

body {
    color: rgb(255, 255, 255);
  font-family: "Nunito", sans-serif;
  background: rgb(23, 33, 43);
}

/* All users */
.all-users {
  display: flex;
  margin-bottom: 1rem;
}

.single-user {
  background: #2b5278;
  color: rgb(255, 255, 255);
  cursor: pointer;
  padding: 5px 7px;
  border-radius: 20px;
  margin-right: 0.5rem;
  position: relative;
}

/* User Card */
.user-card {
  color: rgb(255, 255, 255);
  position: relative;
  border-bottom: 1px solid rgb(255, 255, 255);
  width: 380px;
```

```
}

.user-card .text-content {
  width: 170px;
}

.user-card .name {
  font-weight: 700;
}

.user-card .text,
.user-card .date {
  font-size: 14px;
  color: rgb(180, 180, 180);
}

.user-online {
  display: inline-block;
  height: 8px;
  width: 8px;
  border-radius: 50%;
  background: rgb(119, 248, 83);
  position: absolute;
  top: -3px;
  right: -3px;
  z-index: 2;
}

/* Chat Box */
.chat-box {
  max-height: calc(100vh - 10rem);
  overflow-y: auto;
  background: rgb(14, 22, 33);
  border-radius: 10px;
}

.messages-box {
  height: 85vh;
}

/* Chat Header */
.chat-header {
  display: flex;
  align-items: center;
  justify-content: center;
  padding: 0.75rem;
  color: white;
  background: rgb(9, 14, 22);
}

/* Messages */
.messages {
  max-height: calc(100vh - 15rem);
  overflow-y: auto;
  padding: 0 2rem;
}

.message {
  color: white;
  background: #182533;
  padding: 0.75rem;
  border-radius: 30px;
  max-width: 50%;
}

.message.self {
  color: white;
  background: #2b5278;
}

.message-footer {
  font-size: 8px;
  align-self: flex-end;
  font-weight: 400;
}

/* Chat Input  */
.chat-input {
  width: 100%;
  background: rgb(9, 14, 22);
  padding: 1rem;
}
.send-btn {
  border: none;
  background: #2b5278;
  color: white;
  height: 40px;
  width: 40px;
  border-radius: 50%;
}

/* Notification */
.notifications {
  position: relative;
}

.notifications-icon {
  color: white;
  cursor: pointer;
  position: relative;
  margin-right: 1rem;
}

.notifications-box {
  max-height: 50vh;
  width: 300px;
  position: absolute;
  top: 2rem;
  right: 0;
  box-shadow: rgba(50, 50, 93, 0.25) 0px 2px 5px -1px,
    rgba(0, 0, 0, 0.3) 0px 1px 3px -1px;
  background:#1c3751;
  color: white;
  z-index: 5;
}

.notifications-header {
  padding: 1rem;
  padding-bottom: 0;
  display: flex;
  justify-content: space-between;
}

.notifications-box h3 {
  font-weight: 700;
  font-size: 20px;
}
```

```
.mark-as-read {
  cursor: pointer;
  font-weight: 700;
  opacity: 0.8;
}

.notification {
  font-size: 14px;
  margin: 1rem 0;
  padding-bottom: 0.2rem;
  border-bottom: 1px solid rgb(207, 207, 207);
  display: flex;
  flex-direction: column;
  cursor: pointer;
  padding: 0.5rem 1rem;
}

.notification.not-read {
  background: #2b5278;
}

.notification-time {
  margin-top: 0.2rem;
  font-size: 12px;
  color: #e0e0e0;
}

.notification-count {
  display: flex;
  background: #2b5278;
  height: 25px;
  width: 25px;
  font-size: 14px;
  font-weight: 700;
  border-radius: 50%;
  align-items: center;
  justify-content: center;
  position: absolute;
  top: -10px;
  right: -15px;
}

.this-user-notifications {
  display: flex;
  background: #2b5278;
  height: 20px;
  width: 20px;
  font-size: 12px;
  font-weight: 700;
  border-radius: 50%;
  align-items: center;
  justify-content: center;
}


#element::-webkit-scrollbar {
  width: 10px;
}

#element::-webkit-scrollbar-track {
  background-color:rgb(9, 14, 22);
  border-radius: 10px;
}

#element::-webkit-scrollbar-thumb {
  border-radius: 10px;
  background: #ffffff; ;
}
```

main.jsx:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import './index.css';
import { BrowserRouter } from 'react-router-dom';
import { AuthContextProvider } from './context/AuthContext';

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

Серверна частина

Controllers/chatController.js:

```
const chatModel = require("../Models/chatModel");
```

```
const createChat = async (req, res) => {
  const { senderId, receiverId } = req.body;
  try {
    // перевірка чи чат вже є
    const chat = await chatModel.findOne({
      members: { $all: [senderId, receiverId] },
    });

    if (chat) return res.status(200).json(chat);

    const newChat = new chatModel({
      members: [senderId, receiverId],
    });

    const response = await newChat.save();
    res.status(200).json(response);
  } catch (error) {
    res.status(500).json(error);
  }
};

const userChats = async (req, res) => {
  const userId = req.params.userId;

  try {
    const chats = await chatModel.find({
      members: { $in: [userId] },
    });

    res.status(200).json(chats);
  } catch (error) {
    res.status(500).json(error);
  }
};

const findChat = async (req, res) => {
  const firstId = req.params.firstId;
  const secondId = req.params.secondId;

  try {
    const chat = await chatModel.findOne({
      members: { $all: [firstId, secondId] },
    });

    res.status(200).json(chat);
  } catch (error) {
    res.status(500).json(error);
  }
};

module.exports = { createChat, userChats, findChat };
```

Controllers/messageController.js:

```
const messageModel = require("../Models/messageModel");

const createMessage = async (req, res) => {
  const { chatId, senderId, text } = req.body;

  const message = new messageModel({
    chatId,
    senderId,
    text,
```

```
  });

  try {
    const response = await message.save();
    res.status(200).json(response);
  } catch (error) {
    res.status(500).json(error);
  }
};

const getMessages = async (req, res) => {
  const { chatId } = req.params;

  try {
    const messages = await messageModel.find({ chatId });
    res.status(200).json(messages);
  } catch (error) {
    res.status(500).json(error);
  }
};

module.exports = { createMessage, getMessages };
```

Controllers/userController.js:

```
const userModel = require("../Models/usermodel");
const bcrypt = require("bcrypt");
const validator = require("validator");
const jwt = require("jsonwebtoken");

const createToken = (_id) =>{
    const jwtkey = process.env.JWT_SECRET_KEY;

    return jwt.sign({_id}, jwtkey, {expiresIn: "3d"});
}

const registerUser = async (req, res) => {

    try{
        const { name, email, password } = req.body;

        let user = await userModel.findOne({ email });

        if (user)
         return res.status(400).json("користувач з даним email вже є...");

        if (!name || !email || !password)
         return res.status(400).json("Всі поля є обов'язковими для заповнення...");

        if (!validator.isEmail(email))
         return res.status(400).json("Email повинен бути дійсним...");

        if (!validator.isStrongPassword(password))
         return res.status(400).json("Пароль має бути складнішим...");

         user = new userModel({name, email, password});

         const salt = await bcrypt.genSalt(10);
         user.password = await bcrypt.hash(user.password,salt);

         await user.save();

         const token = createToken(user._id);
```

```
        res.status(200).json({_id: user._id, name, email, token});
    }catch(error){
        console.log(error)
        res.status(500).json(error);
    }

};

const loginUser = async(req, res) =>{
    const{email, password} = req.body;

    try{
        let user = await userModel.findOne({ email });

        if(!user) return res.status(400).json("Неправильний email або пароль...");

        const isValidPassword = await bcrypt.compare(password, user.password);

        if (!isValidPassword) return res.status(400).json("Неправильний email або пароль...");

        const token = createToken(user._id);

        res.status(200).json({_id: user._id, name: user.name, email, token});

    }catch(error){
        console.log(error);
        res.status(500).json(error);

    }
};

const findUser = async (req, res) => {
    const userId = req.params.userId;

    try {
      const user = await userModel.findById(userId);

      res.status(200).json(user);
    } catch (error) {
      res.status(500).json(error);
    }
  };

  const getUsers = async (req, res) => {
    try {
      const users = await userModel.find();

      res.status(200).json(users);
    } catch (error) {
      res.status(500).json(error);
    }
  };

module.exports = {registerUser, loginUser, findUser, getUsers};
```

Models/chatModel.js:

```
const mongoose = require("mongoose");

const chatSchema = new mongoose.Schema(
  {
    members: Array,
  },
  {
```

```
    timestamps: true,
  }
);

const chatModel = mongoose.model("Chat", chatSchema);

module.exports = chatModel;
```

Models/messageModel.js:

```
const mongoose = require("mongoose");

const messageSchema = new mongoose.Schema(
  {
    chatId: String,
    senderId: String,
    text: String,
  },
  {
    timestamps: true,
  }
);

const Message = mongoose.model("Message", messageSchema);

module.exports = Message;
```

Models/userModel.js:

```
const mongoose = require("mongoose");

const userSchema = new mongoose.Schema(
{
    name: {type: String, required: true, minlength: 3, maxlenght: 30},
    email: {
        type: String,
        required: true,
        minlenght: 3,
        maxlenght: 200,
        unique: true,
    },
    password: {type: String, required: true, minlenght: 3, maxlenght: 1024 },

},
{
    timestamps: true,
}
);

const userModel = mongoose.model("User", userSchema);

module.exports = userModel;
```

Routes/chatRoute.js:

```
const express = require("express");
const {
  createChat,
  userChats,
  findChat,
} = require("../Controllers/chatController");

const router = express.Router();
```

```
router.post("/", createChat);
router.get("/:userId", userChats);
router.get("/find/:firstId/:secondId", findChat);

module.exports = router;
```

Routes/messageRoute.js:

```
const express = require("express");
const {
  createMessage,
  getMessages,
} = require("../Controllers/messageController");

const router = express.Router();

router.post("/", createMessage);
router.get("/:chatId", getMessages);

module.exports = router;
```

Routes/userRoute.js:

```
const express = require("express");
const {registerUser, loginUser, findUser, getUsers} = require("../Controllers/userController");

const router = express.Router();


router.post("/register", registerUser);
router.post("/login", loginUser);
router.get("/find/:userId", findUser);
router.get("/", getUsers);

module.exports = router;
```

index.js:

```
const express = require("express");

const cors = require("cors");

const mongoose = require("mongoose");

const userRoute = require("./Routes/userRoute");

const chatRoute = require("./Routes/chatRoute");

const messageRoute = require("./Routes/messageRoute");


const app = express();

require("dotenv").config();


app.use(express.json());

app.use(cors());

app.use("/api/users", userRoute);

app.use("/api/chats", chatRoute);

app.use("/api/messages", messageRoute);
```

```
app.get("/", (req, res) =>{
   res.send("Welcome our chat app APIs..")
});



const  port = process.env.PORT || 5000 ;
const  uri = process.env.ATLAS_URI ;

app.listen(port, (req, res) =>{
   console.log(`Server running on port..: ${port}`);
});

mongoose
   .connect(uri, {
   useNewUrlParser: true,
   useUnifiedTopology: true
   })
   .then(() => console.log("MongoDB connection established"))
   .catch((error) => console.log("MongoDB connection failed: ", error.message) );
```

socket/index.js:

```
const { Server } = require("socket.io");
const io = new Server({ cors: "http://localhost:5173" });

let onlineUsers = [];
io.on("connection", (socket) => {

  socket.on("addNewUser", (userId) => {
    !onlineUsers.some((user) => user.userId === userId) &&
      onlineUsers.push({
        userId,
        socketId: socket.id,
      });

    console.log("Connected Users:", onlineUsers);


    io.emit("getOnlineUsers", onlineUsers);
  });

  socket.on("sendMessage", (message) => {
    const user = onlineUsers.find(
      (user) => user.userId === message.recipientId
    );
```

```
    if (user) {
      console.log("sending message and notification");
      io.to(user.socketId).emit("getMessage", message);
      io.to(user.socketId).emit("getNotification", {
        senderId: message.senderId,
        isRead: false,
        date: new Date(),
      });
    }
  });

  socket.on("disconnect", () => {
    onlineUsers = onlineUsers.filter((user) => user.socketId !== socket.id);
    console.log("User Disconnected:", onlineUsers);

    io.emit("getOnlineUsers", onlineUsers);
  });
});

io.listen(3377);
```