МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інтелектуальних інформаційних систем

_____Юрій КОНДРАТЕНКО

«____»_____ 20___ р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

«ВЕБЗАСТОСУНОК ДЛЯ КОРПОРАТИВНОЇ КОМУНІКАЦІЇ»

Спеціальність 122 Комп'ютерні науки

Освітня програма «Комп'ютерні науки»

Здобувач _____Микола РУДЬ

«___»_____ 20__ р.

Керівник ст. викладач _____Іван БУРЛАЧЕНКО

«___»_____ 20__ р.

Миколаїв 2025

Чорноморський національний університет імені Петра Могили
(повне найменування закладу вищої освіти)

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

ЗАТВЕРДЖУЮ
Завідувач кафедри інтелектуальних інформаційних систем
«Юрій КОНДРАТЕНКО»
«____» _____ 2025 р

ЗАВДАННЯ
на кваліфікаційну бакалаврську роботу здобувача

Рудь Микола Костянтинович
1. (прізвище, ім’я, по батькові здобувача)

1. Тема кваліфікаційної роботи «вебзастосунок для корпоративної комунікації».

Керівник роботи: Бурлаченко Іван Сергійович, ст. викладач.

(прізвище, ім’я, по батькові, посада, науковий ступінь, вчене звання)

Затверджена наказом ЧНУ ім. Петра Могили від «18» листопада 2025 р. № 321.

2. Строк прелставлення кваліфікаційної роботи «__» _____ 2025 р.

3. Очікуваним результатом роботи стане вебсистема комунікації, що забезпечує підтримку обміну повідомленнями в реальному часі, авторизацію користувачів, збереження історії переписок і простим інтерфейсом. Начальними справами є загальних вимогах до продуктивності, масштабованості і надійності системи, вибраного стека технологій (React, ExpressJS, MongoDB) і також концепція архітектури, орієнтованої на обробку великої кількості одночасних з'єднань.

4. Під час виконання роботи потрібно вивчити архітектурні методи щодо

будівництва подібних вебзастосунків, обґрунтувати розвиток технологій для клієнтської та серверної частини, розробити автентифікаційні механізми, забезпечити обмін повідомленнями у реальному часі, забезпечити зберігання та обробку даних, здійснити перевірку продуктивності та оцінити можливості масштабування системи.

5. Перелік графічних матеріалів: презентація, рисунки, таблиці.

Керівник роботи _____ Іван БУРЛАЧЕНКО
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Микола РУДЬ
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата видачі завдання «30» січня 2025 р.

.

КАЛЕНДАРНИЙ ПЛАН
кваліфікаційної роботи

Тема: «Вебзастосунок для корпоративної комунікації»

№	Найменування роботи	Початок	Закінчення	Примітки
1.	Отримання та затвердження завдання на виконання КР	18.12.2024	20.11.2024	
2.	Аналіз та, огляд сучасних засобів корпоративної комунікації	21.12.2024	10.01.2025	
3.	Вибір технологій та формування загальної архітектури рішення	11.01.2025	25.01.2025	
4.	Розробка моделі бази даних (MongoDB, Mongoose)	26.01.2025	09.02.2025	
5.	Реалізація серверної частини на Express.js з підтримкою REST API та Socket.IO	10.02.2025	01.03.2025	
6.	Розробка клієнтської частини вебзастосунку на React + TailwindCSS	02.03.2025	23.03.2025	
7.	Реалізація авторизації з JWT, захист даних, middleware	24.03.2025	05.04.2025	
8.	Інтеграція Cloudinary для обробки медіа-файлів	06.04.2025	12.04.2025	
9.	Підготовка графічних матеріалів, ER-діаграм	13.04.2025	19.04.2025	
10.	Оформлення пояснювальної записки, складання списку джерел	20.04.2025	30.04.2025	
11.	Завершення оформлення КР та презентації Створення презентації, оформлення додатків	01.05.2025	10.06.2025	
12.	Подання на захист КР електронної копії, відгуку та рецензії	11.06.2025	17.06.2025	

Керівник роботи _____ Іван БУРЛАЧЕНКО
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Микола РУДЬ
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата складання календарного плану
«30» січня 2025 р.

АНОТАЦІЯ

до кваліфікаційної роботи
здобувача групи 402 ЧНУ ім. Петра Могили

Рудя Миколи Костянтиновича

на тему: «**ВЕБЗАСТОСУНОК ДЛЯ КОРПОРАТИВНОЇ КОМУНІКАЦІЇ**»

Актуальність даного дослідження полягає в потребі створення надійного вебзастосунку для корпоративної комунікації, який здатен забезпечити ефективний обмін повідомленнями в реальному часі за умов високого навантаження. Це дозволить оптимізувати внутрішні бізнес-процеси, покращити взаємодію між співробітниками та підвищити продуктивність праці.

Об’єктом роботи є процеси транспортних вантажоперевезень і маршрутизації.

Предметом роботи є методи планування і оптимізації транспортних маршрутів із врахуванням часових обмежень.

Метою роботи є зменшення витрат на доставку вантажу і підвищення якості обслуговування клієнтів за рахунок використання часових обмежень та мета-евристичних методів.

У результаті виконання роботи було проаналізовано сучасні рішення у сфері корпоративної комунікації, визначено архітектуру клієнт-серверної системи, реалізовано REST API та WebSocket-з’єднання, впроваджено авторизацію користувачів і зберігання мультимедійних даних у хмарі.

Дана робота складається з чотирьох розділів. У першому розділі виконано аналіз предметної області та аналогічних рішень. У другому розглянуто моделі, методи і технології побудови вебсистем. У третьому описано процеси взаємодії між компонентами системи. У четвертому подано реалізацію клієнтської та серверної частин застосунку.

Загальний обсяг роботи – 123 сторінки. Кваліфікаційна робота містить 1 додаток, 116 рисунків, 5 таблиць і 43 джерела посилання.

Ключові слова: вебзастосунок, корпоративна комунікація, Socket.IO, JWT, React, Express.js, MongoDB, Cloudinary, REST API.

ABSTRACT

to the qualification work by the student of the group 402 of Petro Mohyla Black Sea National University

Rud Mykola Kostiantynovycha

«WEB APPLICATION FOR CORPORATE COMMUNICATION»

The relevance of this study lies in the need to create a reliable web application for corporate communication that is capable of ensuring effective real-time messaging under high load conditions. This will optimize internal business processes, improve interaction between employees, and increase productivity.

The object of the work is the processes of freight transportation and routing.

The subject of the work is methods for planning and optimizing transport routes, taking into account time constraints.

The goal of the work is to reduce freight delivery costs and improve customer service quality through the use of time constraints and meta-heuristic methods.

As a result of the work, modern solutions in the field of corporate communication were analyzed, the architecture of the client-server system was determined, REST API and WebSocket connections were implemented, user authorization and multimedia data storage in the cloud were introduced.

This work consists of four sections. The first section analyzes the subject area and similar solutions. The second section examines models, methods, and technologies for building web systems. The third section describes the processes of interaction between system components. The fourth section presents the implementation of the client and server parts of the application.

The total volume of the work is 123 pages. The qualification work contains 1 attachment, 116 figures, 5 tables, and 43 references.

Keywords: web application, corporate communication, Socket.IO, JWT, React, Express.js, MongoDB, Cloudinary, REST API.

ЗМІСТ

ВСТУП

Сьогоднішній світ неможливо уявити без цифрових інновацій, які пронизують кожен аспект суспільного життя – від особистих повідомлень до складних корпоративних процесів. Швидкість, з якою технології еволюціонують, вимагає від розробників не лише креативності, але і глибокого розуміння викликів, пов’язаних із обробкою великих обсягів даних. Наприклад, зростання аудиторії онлайн-сервісів призводить до необхідності розробляти системи, здатні працювати без збоїв навіть за умов мільйонних одночасних запитів.

Сучасні веб-платформи для онлайн-спілкування функціонують за принципом живого організму: кожен їхній "орган" – від інтерфейсу до серверних модулів – виконує чітко визначену роль, але лише у взаємодії вони забезпечують стабільну роботу. Секрет успіху таких систем криється не лише в технічній потужності, та архітектурній гармонії. Розробникам доводиться балансувати між суперечливими вимогами: наприклад, як залишити систему швидкою, коли до неї під’єднуються десятки тисяч нових користувачів одночасно, або як додати функціонал, не зупиняючи процесів, які вже запущені. Це нагадує ремонт літака під час польоту – ризиковано, але саме такі виклики роблять продукт конкурентноздатним.

Об’єктом роботи – є процес для обміну інформації між користувачами в реальному часі за допомогою вебзастосунку

Предмет роботи – методи, засоби проєктування та реалізація вебсистеми обміну повідомленнями з використанням авторизації, middleware, хмарних сервісів та технологій по типу socket.io.

Мета роботи – є розробка структурованої моделі та реалізувати систему обміну повідомленнями, яка буде забезпечувати ефективну, захищену та ефективну комунікацію між користувачами в реальному часі.

1 АНАЛІЗ ОСОБЛИВОСТЕЙ СТВОРЕННЯ ВЕБЗАСТОСУНКІВ ДЛЯ ОБМІНУ ПОВІДОМЛЕННЯМИ В РЕАЛЬНОМУ ЧАСІ

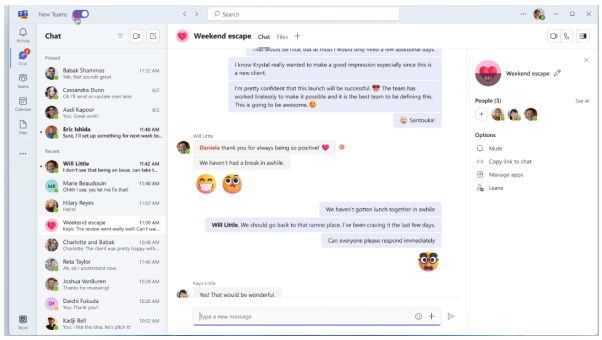
1.1 Особливості предметної області комунікаційних вебсистем

На сьогоднішній день, цифровізація безперервно рухається уперед, сервіси сучасних месенджерів вже стали платформами де ми не просто листуємося, а і переглядаємо новини, слухаємо музику, граємо в ігри, збираємо дані та думки людей за допомогою опитувань та вікторин (особливо в дуже популярних групах). Для розробників можна створювати ботів для пошуку книг до роботи з ШІ. Як користувачі, ми цим користуємося майже щодня. Окрім читання новинних стрічок, можна вести свій канал чи блог та набирати на цьому аудиторію. Можна використовувати сучасні месенджери як хмару, щоб зберігати там файли, фото та нотатки і швидко їх пересилати між пристроями, людям чи в групі. Брати участь у голосових або відеодзвінках, як один на один або в групах, створюючи таким чином meet. Таким чином, можна зробити висновок, що сьогодні месенджери, вже замінюють нам новинні сайти, плеєри по прослуховуванні музики та відео, облачні сервіси по типу Google Cloud. Месенджери не можуть замінити сервіси як Discord, Zoom, Google Meet або Microsoft Teams для онлайн спілкування, але сам факт, що сьогодні месенджери не просто платформа для спілкування, а повноцінний інструмент для інформаційного обміну, розваг та самоорганізації. Тому сьогоднішні сучасні месенджери повинні гарантувати безпеку даних, бо для когось сьогоднішні месенджери можуть бути і способом заробітку і при всьому цьому залишатись гарним, зрозумілим та зручним для користування. Тому у цій дипломній роботі буде розглянуто вебсистему систему для онлайн комунікації, у якої є назва PurrChat – це вебзастосунок для чатів, який може продемонструвати якісну та стабільну умову, навіть при великій кількості одночасних користувачів.

1.2 Аналіз сучасних аналогів і публікацій у сфері вебкомунікацій

Microsoft Teams — це найкращий додаток для обміну повідомленнями для вашої організації — робочий простір для співпраці та спілкування в режимі реального часу, зустрічей, спільного використання файлів і додатків і навіть випадкових емодзі! Все в одному місці, все відкрито, все доступно кожному.

а)



б)

Рисунок 1.1 – Логотип застосунку Microsoft Teams (а) та інтерфейс застосунку (б)

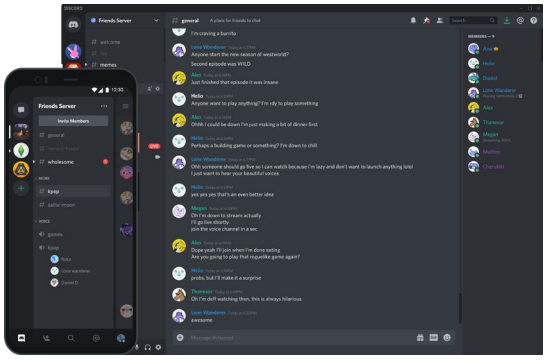
Discord — це програма для голосового, відео- та текстового чату, якою користуються десятки мільйонів людей віком від 13 років для спілкування та спілкування зі своїми спільнотами та друзями.

Люди щодня використовують Discord, щоб поговорити про багато речей, починаючи від мистецьких проєктів і сімейних поїздок до домашніх завдань і підтримки психічного здоров’я. Це дім для спільнот будь-якого розміру, але найбільш широко ним користуються невеликі та активні групи людей, які регулярно спілкуються.

Переважна більшість серверів є приватними місцями, доступними лише для запрошених, для груп друзів і спільнот, щоб залишатися на зв’язку та проводити час разом. Існують також більші, більш відкриті спільноти, які зазвичай зосереджені навколо конкретних тем, таких як популярні ігри, такі як Minecraft і Fortnite. Користувачі можуть контролювати, з ким вони взаємодіють і який їхній досвід у Discord.

Люди люблять Discord, тому що це дім для всіх їхніх спільнот і груп друзів. Це місце, де вони можуть бути собою та проводити час з іншими людьми, які поділяють їхні інтереси та хобі. Розмови в Discord ведуться людьми, які ви обираєте, і темами, на які ви обираєте.

а)

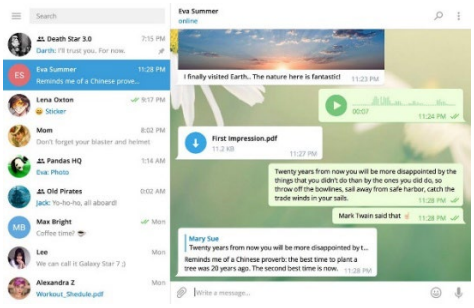


б)

Рисунок 1.2 – Логотип застосунку Discord (а) та інтерфейс застосунку (б)

Telegram, також відомий як Telegram Messenger, — це хмарна кросплатформна служба соціальних мереж і обміну миттєвими повідомленнями (IM). Спочатку він був запущений для iOS 14 серпня 2013 року та Android 20 жовтня 2013 року. Він дозволяє користувачам обмінюватися повідомленнями, ділитися медіафайлами та файлами, а також проводити приватні та групові голосові чи відеодзвінки, а також публічні прямі трансляції. Він доступний для Android, iOS, Windows, macOS, Linux і веб-браузерів. Telegram пропонує наскрізне шифрування під час голосових і відеодзвінків і, за бажанням, у приватних чатах, якщо обидва учасники використовують мобільний пристрій.

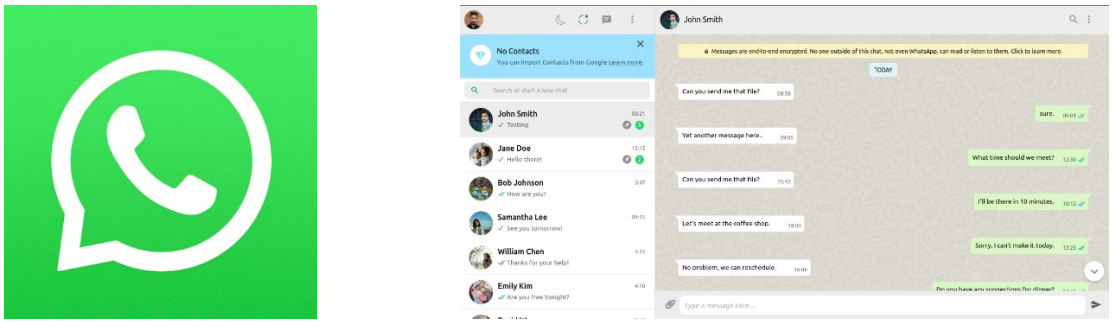
а)



б)

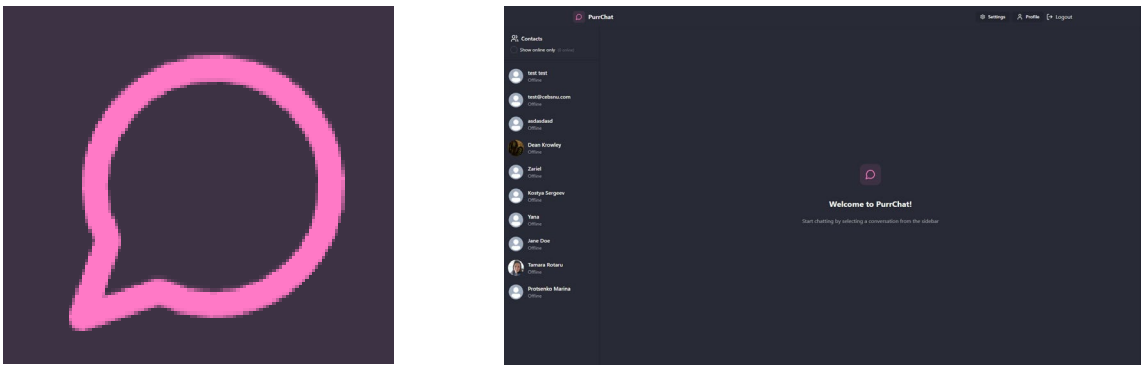
Рисунок 1.3 – Логотип застосунку Telegram (а) та інтерфейс застосунку (б)

Спочатку додаток WhatsApp замислювався як альтернатива SMS. Сьогодні сервіс підтримує надсилання та отримання даних у найрізноманітніших форматах: повідомлення, фото, відео, документи, геопозиція та аудіодзвінки. У WhatsApp ви ділитеся дуже особистими моментами, і саме тому ми зробили наскрізне шифрування невід'ємною частиною нашого застосунку. За кожним рішенням щодо розвитку сервісу стоїть наше прагнення створити умови для спілкування без перешкод з будь-якої точки світу.



а) б)

Рисунок 1.4 – Логотип застосунку Whatsapp (а) та інтерфейс застосунку (б)



а) б)

Рисунок 1.5 – Логотип застосунку PurrChat (а) та інтерфейс застосунку (б)

PurrChat – молодий проєкт з величезними амбіціями

1.3 Аналіз архітектури розподілених систем для оптимізації мережевої взаємодії

У найближчому майбутньому всі наші повсякденні речі та об'єкти будуть під'єднані до Інтернету й оснащені достатньою кількістю сенсорних, діючих і обробних можливостей, щоб використовувати всі потенційні переваги так званої парадигми Інтернету речей (IoT). Навіть найпростіші об'єкти стануть «розумними», оскільки вони будуть пов'язані з іншими об'єктами для обміну та збору даних із середовищ, у яких вони перебувають, тим самим прокладаючи шлях до нових прикладних служб, обчислювальних і комунікаційних сценаріїв.

У цьому контексті взаємодія між різними стандартами і технологіями зв'язку, як і раніше, є істотною проблемою, яку ми почали вирішувати, запропонувавши мобільний шлюз на базі смартфона, що діє як гнучкий і прозорий інтерфейс між різними пристроями IoT та Інтернетом. Представлена уніфікована, високорівнева і розширювана архітектура програмного забезпечення підтримує виявлення, контроль і управління опортуністичними пристроями IoT у поєднанні з функціями обробки, збору та поширення даних.

Був розгорнутий спеціальний випробувальний стенд на звичайних смартфонах з різними апаратними і програмними можливостями для оцінки реальної здійсненності розробленого рішення, що вимірює продуктивність системи з погляду споживання енергії, пам'яті та використання ЦП у сценаріях високого і низького навантаження. Згідно з отриманими результатами, реалізована архітектура програмного забезпечення для взаємодії декількох стандартів і декількох технологій являє собою скорочене використання апаратних ресурсів перед відносно високим значенням споживання енергії, в основному через одночасно активні радіоінтерфейси в поєднанні з невеликою ємністю акумулятора, що обмежує термін служби смартфона. Проте, представлений загальний підхід, як і раніше, примітний, оскільки цей останній аспект, найімовірніше, буде перевищено незабаром завдяки щоденним технологічним досягненням і іноваціям.

Висновки до розділу 1

У першому розділі досліджуються особливості розробки комунікаційних систем та їх порівняння із сучасними аналогами. Месенджери вже не просто додатки для листування — це цілі екосистеми з функціональністю соцмереж, хмарних середовищ, медіаплеєрів. Проєкт PurrChat демонструє підхід до створення веб-системи для комунікації. Ключові виклики, як-от мінімізація затримок, обробка великої кількості з’єднань і захист від загроз, вирішуються завдяки JWT-автентифікації, шифруванню bcryptjs та інтеграції хмарних сервісів (Cloudinary). Аналіз конкурентів (Microsoft Teams, Discord, Telegram, WhatsApp) підкреслив важливість оптимізації архітектури для розподілених систем. На відміну від них, PurrChat акцентує увагу на інтеграції додаткових функцій та підтримці SPA-архітектури для покращення user experience. Таким чином, підкреслено актуальність створення продуктів, що поєднують продуктивність, безпеку, гнучкість і зручність. Досвід, отриманий під час розробки PurrChat, був безцінним.

2 МОДЕЛІ, МЕТОДИ, АЛГОРИТМИ ТА ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ДЛЯ РОЗВ'ЯЗАННЯ ЗАДАЧІ

2.1 Базова архітектура для онлайн-системи

DOSN на основі DHT є суто плоскими (однорівневими) та неієрархічно структурованими системами (наприклад, PeerSoN використовує OpenDHT, Safebook використовує kademlia, Cachet, DECENT, a LibreSocial використовують FreePastry, WebP2P використовує Chord). плоских DHT-системах, таких як Chord, всі вузли , що беруть участь у мережі, організовані в єдину оверлейну мережу та відіграють рівну роль (вони несуть однакову відповідальність). У таких системах продуктивність пошуку є $O(log_2(N))$; N позначає кількість вузлів в оверлеї. Однак, DOSN , що використовують плоскі DHT-системи, не враховують багато важливих аспектів (автономність віртуальної організації та їхні конфліктні інтереси, принцип локальності). Оскільки OSN – це великомасштабні комп'ютерні мережі зі складною та динамічною структурою, великою кількістю користувачів та високим рівнем відтоку (часте підключення/відключення вузлів), збереження плоскої DHT-структури призводить до високих накладних витрат. Крім того, важливим параметром, який необхідно враховувати, є неоднорідність між вузлами. У плоских DHT -системах DOSN усі вузли обробляються однаково, на відміну від реального світу, де вузли мають різні можливості. Цілями запропонованої нами архітектури є: (1) Покращити процес пошуку даних користувача (профілів) шляхом мінімізації кількості переходів (пошук у невеликих оверлеях та кешування контенту в супер-вузлах) та зменшення часу пошуку (оптимізовано розмір таблиці пальців (2) Обмежити накладні витрати шляхом реплікації даних у найбільш доступних вузлах (найбільш доступних друзі, якщо вони існують). Для досягнення цих цілей наше рішення базується на організації вузлів в ієрархічному дворівневому (дворівневому) оверлеї. Розділивши плоску мережу на менші області або кластери, складність системи буде зменшена. Автори в виявили високий ступінь гетерогенності фізичних можливостей (потужність процесора , ємність сховища

тощо) та стабільності (період доступності вузлів у мережі P2P) вузлів-учасників у межах оверлею P2P. Хоча велика кількість вузлів залишається протягом короткого часу та має низькі можливості, невелика частина зазвичай має вищий час безвідмовної роботи та має відносно кращі фізичні можливості та стабільність. Ми можемо скористатися цією гетерогенністю при проєктуванні великих надійних систем для DOSN (це підвищує масштабованість мережі), де найпотужніші вузли формують оверлей верхнього рівня як кільце хорди (рівень 1), а кожен вузол виконує роль супер-оверлея для групи звичайних вузлів з низьким часом безвідмовної роботи, ці останні формують оверлей нижнього рівня (кільце хорди) та з'єднуються як під-оверлей верхнього рівня. Наша архітектура є ієрархічним оверлеєм на основі DHT, вона складається з двох рівнів, на яких вузли організовані в непересічні групи (кластери), і пропонує кілька важливих переваг порівняно з архітектурами DOSN на основі DHT, заснованими на плоскому DHT, як в:

− зменшення середньої кількості переходів для запиту пошуку через кількість кластерів, яка буде меншою за загальну кількість вузлів;

− зменшення накладних витрат на обслуговування: оскільки ієрархічна структура організовує вузли в кілька оверлеїв, менших за плоску, це дозволяє маршрутизувати повідомлення про обслуговування лише в одному з них. Це пришвидшить корекцію станів маршрутизації, одночасно зменшуючи кількість стабілізаційних повідомлень, що обробляються кожним вузлом;

− ізоляція відтоку: топологічні зміни в кластері, спричинені відтоком (з'єднання, виходи, відмови вузлів), не вплинуть на верхнє оверлейне ядро або інші кластери. Вони будуть розглядатися як локальні події, що впливають лише на один кластер, таким чином таблиці маршрутизації поза групою не зазнають впливу;

− мінімізація затримки мережі, коли вузли в одному кластері групуються за фізичною близькістю;

− сприяння розгортанню у великих масштабах шляхом надання кожному кластеру-учаснику адміністративної автономії та прозорості, оскільки будь-хто з них може обрати власний протокол накладання (одночасно можна

використовувати багато алгоритмів міжгрупового та внутрішньогрупового пошуку);

− кешування контенту: ієрархічна організація груп однорангових вузлів ідеально адаптована до кешування контенту , що може значно скоротити час відгуку та накладні витрати на зв'язок під час пошуку ресурсів, а також покращити продуктивність з точки зору доступності даних.

Середня кількість переходів, необхідних для локального запиту пошуку в кільці Chord з N вузлами, становить $O(log_2(M))$ де M – кількість вузлів у кластері. Різні дослідники зосереджуються на різних аспектах DOSN, таких як P2P-архітектури для DOSNs, DOSN рішення щодо проєктування (сховище, контроль доступу, механізми взаємодії та сигналізації), безпека та конфіденційність у DOSN. Ми не розглядаємо вичерпно всі проблеми, такі як проблема безпеки, але зосередимося на P2P-архітектурах для DOSN та питанні доступності даних, що є основною проблемою для DOSN . Водночас ми прагнемо мінімізувати накладні витрати, що виникають через часту міграцію реплік даних (репліки часто переміщуються через відтік у політиках реплікації). З міркувань продуктивності ми пропонуємо ієрархічне накладання на основі DHT з двома рівнями, ми обрали акорд DHT для кожного кластера нашої DOSN . У цьому накладенні верхній рівень складається зі стабільних вузлів з більшими можливостями, тоді як нижчий рівень містить менш стабільні вузли (звичайні вузли) з обмеженими ресурсами. Вартість створення та підтримки оверлею в цій архітектурі також менша, ніж у традиційних реалізаціях DHT. Фактично, структура даних (таблиці Finger, списки наступників) верхнього рівня оновлюватиметься рідше.

2.1.1 Огляд архітектури

Суперпірери формують кільце верхнього рівня (верхнє кільце) ієрархічної DHT та з'єднуються один з одним у кільцеподібну структуру (Chord), діючи як шлюзи між верхнім та нижчим рівнями цієї архітектури. Суперпірери зазвичай потужніші та стабільніші, ніж звичайні піри, і тому несуть більше відповідальності,

ніж звичайні. Суперпірери зазвичай вибираються на основі деяких показників, таких як потужність процесора, мережеве з'єднання, надійність тощо. Кожен суперпірер обробляє групу пірів нижчого рівня та відповідає за поширення запитів для пірів у своїй групі. Ми вважаємо, що суперпірери кешують копії найпопулярніших профілів (часто запитуваних) для обробки зовнішніх запитів, якщо власники цих профілів не в мережі. Щоб ці копії були актуальними, суперпірер періодично запитує оновлення кешованих профілів. Це зменшить кількість переходів та мережевого трафіку, вирішуючи більше зовнішніх запитів в оверлеї суперпірера, і значно покращить продуктивність пошуку.

Нижній рівень також являє собою структуровану P2P-мережу, що базується на протоколі Chord і складається з кластерів менш стабільних вузлів (учасників DOSN).

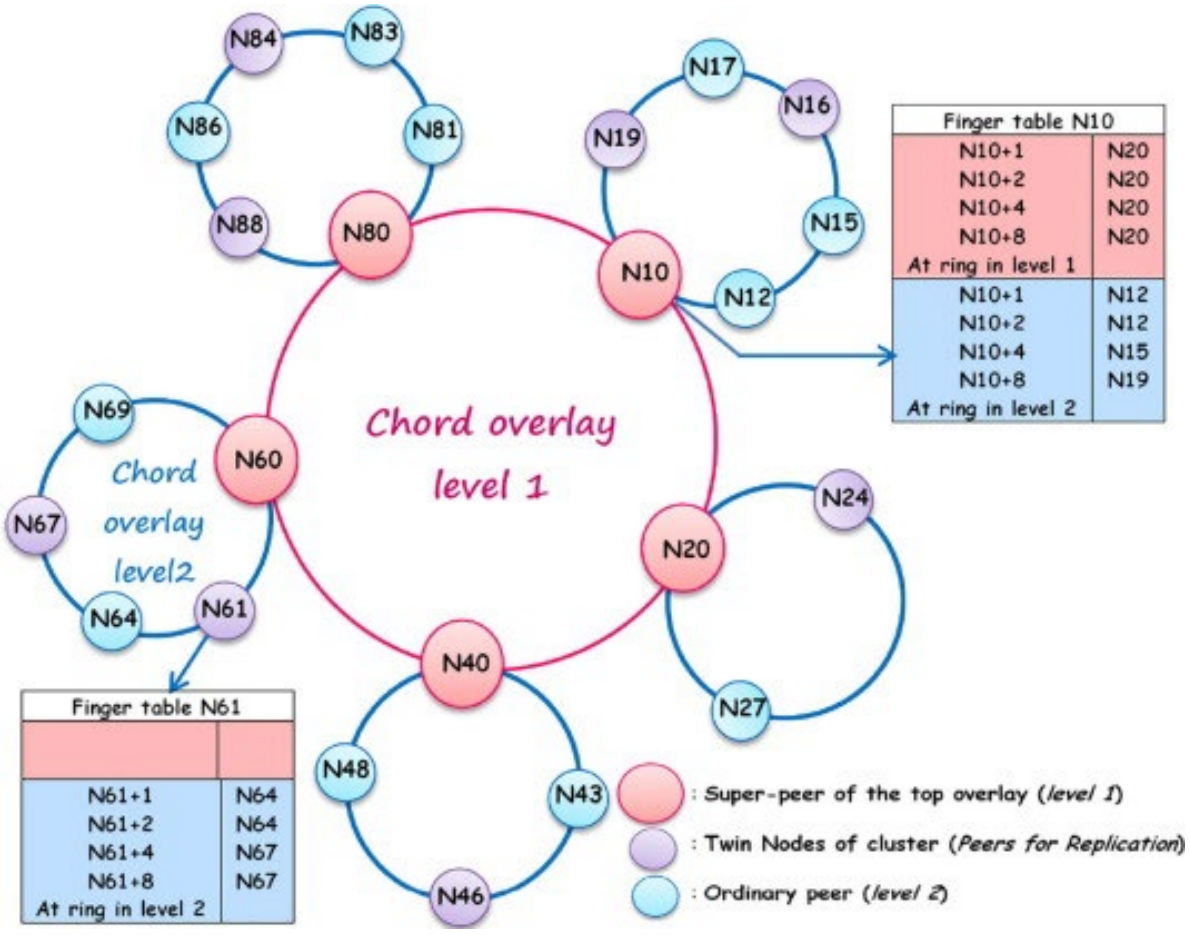


Рисунок 2.1 – Запропонована ієрархічна архітектура (DOSN 2-Tiers)[31].

Однорангові вузли в кожному кластері фізично та логічно близькі один до одного, що значно та ефективно зменшує затримку пошуку. Кожен вузол на цьому рівні належить лише до одного кластера. Усі кластери визначаються графом $G(C, E)$, де: $C = \{C_1, C_2, \dots, C_S\}$ представляє множину всіх кластерів, а E позначає множину віртуальних зв'язків між кластерами в C.

2.1.2 Маршрутизація запиту на пошук

Маршрутизацію поділяють на дві фази: внутрішньокластерну маршрутизацію та міжкластерну маршрутизацію:

− у внутрішньокластерній маршрутизації , де запитуваний ключ k (k – це профіль користувача) локально існує в тому ж кластері, процес пошуку буде таким самим, як і в протоколі Chord;

− при міжкластерній маршрутизації , де ключ k знаходиться в іншому кластері, запит на пошук передається супер-вузлами на DHT (до зовнішніх кластерів). Щоб знайти кластер, відповідальний за ключ k , вузол, який шукає ключ k, зв'язується з супер-вузлом у своєму кластері, який згодом пересилає запит на DHT до супер-вузла кластера призначення. Зрештою, якщо запитуваний ключ не знайдено на супер-вузлі кластера призначення, він пересилається всередині цього кластера.

Процес пошуку виконується в три кроки наступним чином:

− спочатку запит на пошук ключа k направляється до супер-вузла ініціюючого кластера. Якщо ключ k знайдено локально, пошук завершується, інакше виконуються кроки 2 та 3;

− по-друге, використовуючи алгоритм пошуку Chord, запит на пошук направляється до супер-вузла кластера, ідентифікатор якого є $id_{user} = successor(k)$ у верхньому рівні, щоб знайти супервузла кластера, відповідального за k . У цьому випадку ключ k можна знайти: (1) На супервузлі кластера, відповідального за k (коли це запитуваний вузол, який зберігає k, оскільки він також є звичайним вузлом, який має друзів), тому пошук успішно завершено. (2)

У кеші супервузла, якщо запитуваний вузол не підключений. Коли супервузл отримує запит на пошук для k , який існує в його кеші, він перевіряє, чи не підключений шуканий вузол, за допомогою повідомлення ping, якщо так, він відповідає на запит, надсилаючи копію профілю, інакше буде виконано крок (3);

– по-третє, запит на пошук далі передається до вузла другого рівня для пошуку всередині кластера, відповідального за k. Якщо вузол не знайдено в онлайн-статусі (він вийшов з DOSN), пошук буде здійснено для однієї з реплік його профілю, що зберігаються у вузлах реплікації кластера з офлайн-статусом, використовуючи метадані локалізації набору реплік Як показано (на рис. 2.2), запит на пошук ключа (198 | 104) ініційований вузлом A оверлею 2-го рівня, передається його супервузлу (70 | 190), цей ключ не належить до кластера 70 $id_{cluster}$ запиту є $198 \neq 70$), тому супер-рівень (70 | 190) направляє цей запит до вузла (198 | 104) його кластера, виконавши алгоритм пошуку акордів.

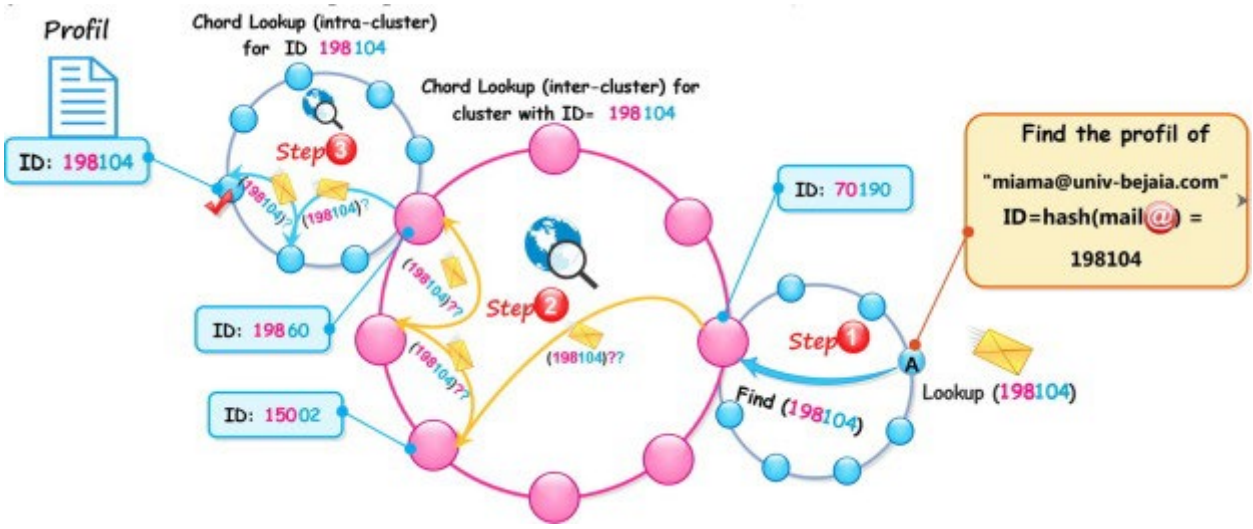


Рисунок 2.2 – Приклад процесу пошуку[31].

Алгоритм процесу пошуку Псевдокод у $DOSN$ $2-$ рівневий пошук k $(c_{n1}\ c_{n2})$[31]:

– початок;

– знайти ключ $k(c_{n1}, c_{n2})$ з $super-peer_{Querry\ldots k}$ в тому ж кластері /*

$super - peer_{Querry\ldots k}$ є супер-вузлом кластера, з якого здійснюється пошуковий запит для ключа k вихід;

– якщо [Ключ знаходиться в тому ж кластері;] $k, c_{n1} == super - peer_{Querry\ldots k}\ id_{cluster}$;

– $super - peer$ find_successor_level-2(k) /* Знайти наступника ключа використання find _ successor() алгоритму Chord;

– **інакше** [Ключ $k \notin$ до кластера рівня 2 запит спрямовується супер-вузлом рівня 1];

– super – peer *find_successor_2levels_DHT* (k);/*знайти кластер, де ∈ ключ k у накладці рівня 1 та наступнику ключаку своєму кластері;

– кінець, якщо;

– $super - peer_{Querry\ldots k}$ *find_successor_2levels_DHT* (k);

– якщо (k, $c_{n1} \in [super - peer_{Querry\ldots k}, successor - super - peer_{Querry\ldots k}]$);

– повернення $successor - super - peer_{\mathrm{Query\ldots k}}$;/*супервузівський вузол кластера, де ∈ ключ k знайдено;

– if [ключ/знайдено] $successor - super - peer_{\mathrm{Query\ldots k}}$ – це запитуваний вузол, який зберігає k;

– return (результат пошуку);

– else [k знаходиться в $successor - super - peer_{\mathrm{Query\ldots k}}$];

– $super - peer\ (k)$. find_successor_2 (k, c_{n2}); /* Використовуючи *find _ successor()* для Chord, знайдіть наступника тональності k супер-вузлом свого кластера з накладанням 2-го рівня;

– end, if;

– return [передати запит пошуку навколо накладання рівня 1];

– $super - peer_{lookup}$ = *closest_preceding_node (*k, $\mathrm{c_{n1}}$*);* *closest_preceding_node* процедури Chord вибирає з $super - peer_{Querry\ldots k}$ таблиця пальців, вузол з найбільшим ідентифікатором, що передує k, c_{n1};

- return $super - peer_{lookup}$ find_ successor_level_1 (k, c_{n1}); /* продовжити пошук $super - peer(k)$ за $super - peer_{lookup}$ накладання рівня 1;
- $super - peer(k)$. find_ successor_level_2 (k, c_{n2}); /* використовуючи find_successor() для Chord, знайти наступника тональності k у своєму кластері оверлею 2-го рівня;
- end, if;
- if (шуканий користувач не існує зі status_online);
- пошук однієї з реплік профілю зі status_offline;
- end, if;
- return (результат пошуку);
- end.

2.1.3 Маршрутизація запиту на пошук

Кожен вузол (або користувач) у *DOSN* представлений профілем, який є набором даних користувача. Вони можуть бути публічними, захищеними обмеженим доступом для підмножини друзів або приватними та недоступними для будь-кого (Дані в нашому DOSN мають таку ж класифікацію).

Щоб забезпечити доступність профілів своїх користувачів *(* публічного контенту), архітектури DECENT та Cachet використовують DHT для зберігання реплік профілів користувачів. Cachet розміщує репліки профілів у випадково вибраних вузлах, тоді як DECENT використовує набір друзів відповідального вузла за профіль (використовує Neighbor Replication).

У нашій архітектурі, щоб забезпечити високу доступність профілю користувача, ми використовуємо DHT як Cachet та DECENT, але пропонуємо стратегію реплікації, відмінну від тієї, що використовується в цих архітектурах. Ми підтримуємо репліки даних користувача в *R* найбільш доступних вузлах набору наступників кожного вузла, враховуючи аспект дружби, зберігаючи дані в дружніх вузлах щоб мінімізувати витрати на обслуговування порівняно з іншими архітектурами (Cachet та *DECENT*). Наша стратегія реплікації спрямована на

досягнення кращої доступності з оптимізованими накладними витратами, не нехтуючи аспектом безпеки.

2.2. Технології вебзастосунку для комунікації

Цей проєкт створений за допомогою сучасних технологій: серверна частина була реалізована за допомогою Express.js, а клієнтська частина була зроблена за допомогою React Js, для того, щоб обмін повідомленнями був у реальному часі використовується Socket.IO. Це JavaScript-бібліотека для вебсайтів та обміну даними в перший момент. Складаючись із двома частинами: клієнтською, вона запускається у браузері та серверною для node.js, що буде забезпечувати миттєву комунікацію в реальному часі. Для того, щоб була можливість розробляти таку систему для онлайн комунікації було зіткнено з великою кількістю викликів. Насамперед ключовим є створення способу для листування, з мінімальними затримками і захисту користувачів від збоїв та забезпечення достатнього захисту їх даних. Наступною проблемою буде зустрітись з величезною кількістю одночасних користувачів, зберігаючи при цьому швидкість та зручність інтерфейсу. У вебдодатку PurrChat ці виклики вирішує Socket.IO, який працює на базі WebSocket і він буде доповнювати вебсистему своїми механізмами для миттєвого з’єднання та обробки подій. Це підтримує швидкий зв’язок навіть, якщо будуть якісь тимчасові проблеми з мережею і навіть якщо WebSocket може бути не доступним за якихось причин, Socket.IO буде автоматично переводити користувачів на інші методи передачі даних, що буде покращувати user experience.

Також слід приділити увагу frontend частині. PurrChat використовує React - JavaScript-бібліотека з відкритим вихідним кодом для розроблення користувацьких інтерфейсів. React розробляють і підтримують Facebook, Instagram і спільнота окремих розробників і корпорацій. React можна використовувати для розробки односторінкових і мобільних додатків. Щоб реалізувати швидко адаптивний дизайн було використано TailwindCSS – CSS-фреймворк із відкритим вихідним кодом, створеним Адамом Уетеном і підтримуваним Tailwind Labs. Особливість

цієї бібліотеки в тому, що вона не передбачає CSS-класи окремих елементів. Замість цього вона надає службові класи, які можна об’єднати для стилізації кожного елемента. Ми використовуємо daisyUI – це плагін Tailwind CSS, який надає корисні імена класів та компонентів, щоб допомогти писати менше коду і швидше кодити. З цим плагіном цей вебзастосунок виглядає красиво, естетично та зручно у користувані, як для ПК так і для мобільних пристроїв, планшетів або телефонів. Інтерфейс продуманий до дрібниць і ви ще не раз переконаєтесь у цьому. Користувачі будуть легко переглядати усі свої чати, керувати історією повідомлень, редагувати свій профіль, налаштовувати свої персональні дані та налаштовувати вигляд сторінки, для цього у додатку є 32 різних тем, які повинні задовільняти найвибагливіших клієнтів.

Для управління станами у застосунку використовується бібліотека Zustand - сучасний інструмент для керування станом React-додатків. У проєкті він потрібен, щоб зберігати дані про сеанти повідомлення і авторизацію. Щоб працювати з REST API, для управління користувачами, чатами, історією чатами, повідомленнями та авторизацією, для всього цього використовується бібліотека Axios, щоб гарантувати клієнтам надійну та безпечну взаємодію між клієнтом та сервером.

Backend частина PurrChat використовує Express.js або просто Express –веб-фреймворк для Node.js, реалізований як вільне та відкрите програмне забезпечення під ліцензією MIT. Він спроєктований для створення веб-прикладень і API. Де-факто є стандартним каркасом для Node.js. За допомогою якого оброблюються запити до REST API - архітектурний підхід до організації взаємодії компонентів розподілених додатків у мережі, відомий як REST, можна описати як набір принципів, які допомагають розробнику структурувати код серверної частини так, щоб різні системи могли без проблем обмінюватися інформацією. Аутентифікація для юзерів реалізовується за допомогою JWT, щоб забезпечити безпечне зберігання даних. JWT це відкритий стандарт для створення токенів доступу, що базується на форматі JSON. Як правило, використовується для передачі даних після автентифікації в клієнт-серверних програмах. Конфіденційні параметри, такі як

ключі для токенів, захищені за допомогою dotenv. Паролі будуть шифруватися за допомогою бібліотеки bcryptjs, що робить PurrChat надійним навіть у разі витоку. Щоб забезпечити наших клієнтів від XSS чи SQL-ін’єкції, був застосований механізм валідації даних, бібліотека cookie-parser - це інструмент, який допомагає серверу розбирати та обробляти кукі, що надходять із запитів клієнта, щоб зручно витягувати з них дані й керувати сеансами користувачів у вебдодатку. А також cors для керування заголовками безпеки та cookie. Це повинно забезпечувати надійність та безпеку від потенційних загроз.

Для зберігання даних використовується база даних MongoDB – це база даних, яка дозволяє гнучко зберігати інформацію у вигляді документів, схожих на JSON, і швидко з ними працювати, що робить її зручною для додатків, де потрібно масштабуватися та обробляти великі обсяги даних без жорсткої структури. Щоб взаємодіяти з MongoDB треба застосовувати бібліотеку Mongoose. Під час перебору файлів для кожного користувача файли будуть зберігатись на хмарному сервісі через Cloudinary, що дасть зручність для зберігання фотографій профілю користувача або повідомлень із зображеннями.

Висновки до розділу 2

Під час розробки вебзастосунку для корпоративної комунікації особливо приділялась увага, щоб інтегрувати сучасні техногогії, які будуть здатні забезпечити маштабованість та доступність. У PurrChat була застосована спрощена архітектура DHT-мережі на основі централізованої бази даних. Коли у теоретичній частині було розглянуто класичну модель (зокрема протоколи на кшталт Chord або Kademlia), який можна використати як орієнтир для подальшого вдосконалення PurrChat.

Таким чином PurrChat є надійним та гнучким застосунком з сучасними технологічними рішеннями, що дозволять досягти безпечної та маштабованої взаємодії у реальному часі. Це створює ґрунт для подальшого розвитку PurrChat, щоб той став ще надійнішим та безпечнішим для майбутніх клієнтів.

3 АНАЛІЗ ОСОБЛИВОСТЕЙ СТВОРЕННЯ ВЕБЗАСТОСУНКУ ДЛЯ ОБМІНУ ПОВІДОМЛЕННЯМИ В РЕАЛЬНОМУ ЧАСІ

3.1 Особливості предметної області комунікаційних вебзастосунків

Нижче наведена таблиця, що описує схему User, яка зберігає інформацію про користувачів у базі даних.

Таблиця 3.1 – Таблиця класу User

Атрибут	Тип	Опис
_id	ObjectId	Унікальний ID користувача
fullName	String	Унікальне ім’я користувача
password	String	Хешований пароль
email	String	Електронна адреса
profilePic	String	Посилання на зображення профілю
createdAt	Date	Дата створення облікового запису
updatedAt	Date	Дата останнього оновлення запису

Один користувач може відправити багато повідомлень — зв’язок User ||--o{ Message : "senderId".

Таблиця 3.2 – Таблиця класу Message

Атрибут	Тип	Опис
_id	ObjectId	Унікальний ID для повідомлення
senderId	ObjectId	Посилання на відправника (User._id)
receiverId	ObjectId	Посилання на одержувача
text	String	Текст повідомлення
image	String	URL зображення
createdAt	Date	Дата створення облікового запису
updatedAt	Date	Дата останнього оновлення запису

Один користувач може отримати багато повідомлень — зв’язок User ||--o{ Message : "receiverId".

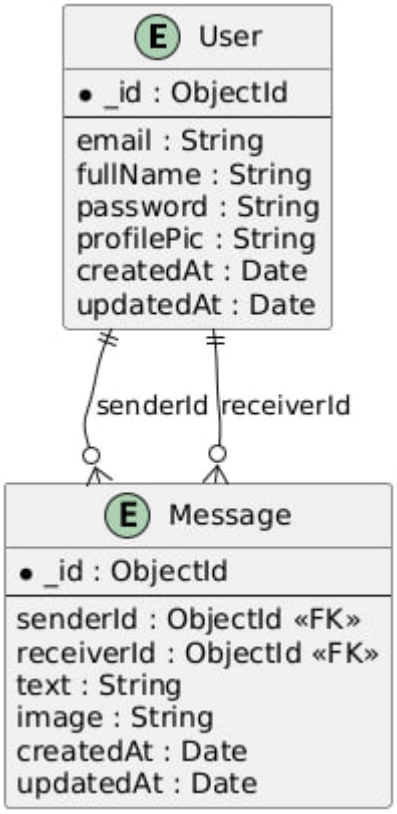


Рисунок 3.1 – Діаграма структури бази даних із сутностями User та Messages

Таблиця 3.3 – Методи класу User

Атрибут	Параметри	Повертає	Опис
_findById(id)	id: ObjectId	Promise	Отримання документа користувача за його ObjectId
find(query)	query: Object	Promise<User[]>	Пошук користувачів за довільним критерієм
select(fields)	fields: String[]	User	Вибір необхідних полів для мінімізації обсягу даних

Ця таблиця демонструє методи, що реалізовані для роботи з колекцією юзерів

Таблиця 3.4 – Методи класу Message

Атрибут	Параметри	Повертає	Опис
find(query)	query: Object	Promise<Message[]>	Отримання списку повідомлень за критеріями
findByIdAndUpdate(id, data, options)	id: ObjectId, data: Object	Promise	Редагування вже наявного повідомлення
findByIdAndDelete(id)	id: ObjectId	Promise	Видалення повідомлення
save()	—	Promise	Створення нового документа повідомлення

Ця таблиця демонструє методи, що реалізовані для роботи з колекцією меседжів

3.1.1 Компоненти проміжної обробки Компоненти проміжної обробки

Проміжна перевірка автентифікації protectRoute(req, res, next): void:

- отримує JWT із заголовку Authorization;
- верифікує токен через jsonwebtoken;
- додає інформацію про користувача (req.user) або повертає помилку 401.

Менеджер сокетів SocketManager:

- socketUserMap: Map<socketId, userId> – зв’язок між ідентифікатором з’єднання та користувачем;
- userSocketMap: Map<userId, socketId> – зворотній зв’язок для швидкої доставки.

Методи:

- onConnection(socket): void – обробка події підключення клієнта;
- getReceiverSocketId(userId): string – пошук активного сокету для відправки повідомлень у реальному часі.

Завантажувач CloudinaryUploader:

- upload(image): Promise<{ secure_url: string }> – завантаження файлу на сервіс Cloudinary та повернення захищеного URL.

3.1.2 Компоненти проміжної обробки Компоненти проміжної обробки

а) клієнт (React) формує об’єкт { senderId, receiverId, text, imageData } і через Socket.IO емулює подію sendMessage;

б) SocketManager.onConnection перехоплює подію, викликає MessageController.sendMessage(req, res) на сервері;

в) AuthMiddleware.protectRoute перевіряє JWT у перехопленому HTTP-запиті/Socket-евенті.

г) MessageController.sendMessage:

1) якщо є imageData, викликає CloudinaryUploader.upload → отримує URL → додає в поле image;

2) викликає new Message({ senderId, receiverId, text, image }) → save() → зберігає в базу даних;

3) викликає SocketManager.getReceiverSocketId(receiverId) → відправляє подію newMessage у браузер одержувача.

д) клієнт через Socket.IO отримує newMessage і викликається рендер інтерфейсу.

Таблиця 3.5 – Характеристика вхідних даних

Поле	Джерело	Тип	Обмеження	Опис
senderId	Клієнт	ObjectId	Обов’язкове	ID відправника
receiverId	Клієнт	ObjectId	Обов’язкове	ID одержувача
text	Клієнт	String	–	Тіло повідомлення
imageData	Клієнт	Base64	optional	imageData
JWT	Заголовок	String	–	Токен для захисту маршруту
secure_url	Cloudinary	String	–	URL на медіа

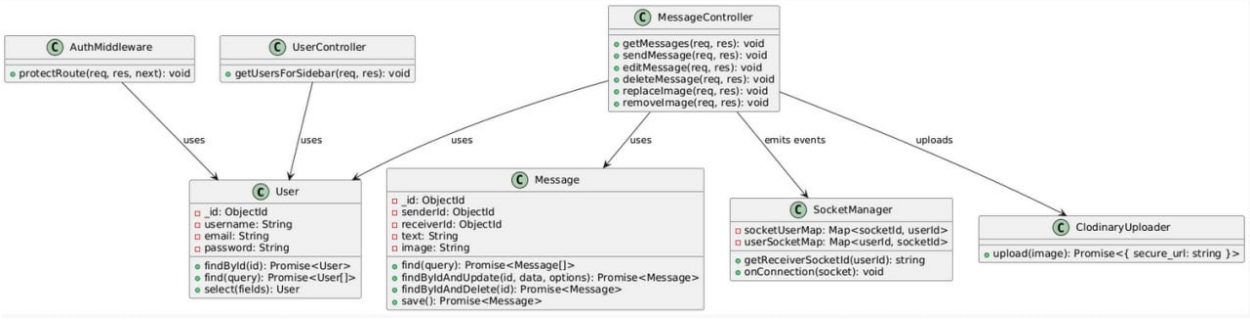


Рисунок 3.2 – Зв’язки між компонентами

3.1.3 Послідовність обробки запитів автентифікації, отримання користувачів та надсилання повідомлень

На початку відбувається запит до AuthMiddleware та викликається метод protectRoute(req, res, next), щоб перевірити авторизацію. Middleware декодує токен за допомогою jwt модуля з допомогою методу verify(token, secret), після цього отримуємо ID нового користувача, щоб звернутися до моделі User, викликається метод findById(decoded.userId), щоб отримати інформацію про користувача. Якщо перевірка була успішною викликається метод next(), і запит буде перенаправлено до UserController.

Цей UserController обробляє запит getUsersForSidebar(req, res), у якому виконується пошук користувачів окрім поточного (find({_id: {$ne:loggedInUserId}})), після цього список користувачів повертається користувачу у вигляді JSON формату.

Клієнт подає запит sendMessage(req, res) до MessageController. Контролер перенаправляє зображення до clodinary через upload(image), потім отримуємо відповідь за посиланням secure_url, після цього зберігаємо повідомлення до бази даних, викликаємо метод save({senderId, receiverId, text, imageUrl}) у моделі Message. Збережене повідомлення newMessage назад відправляємо клієгту і паралельно надсилаємо подію за допомогою socket.io методом emit("newMessage", newMessage).

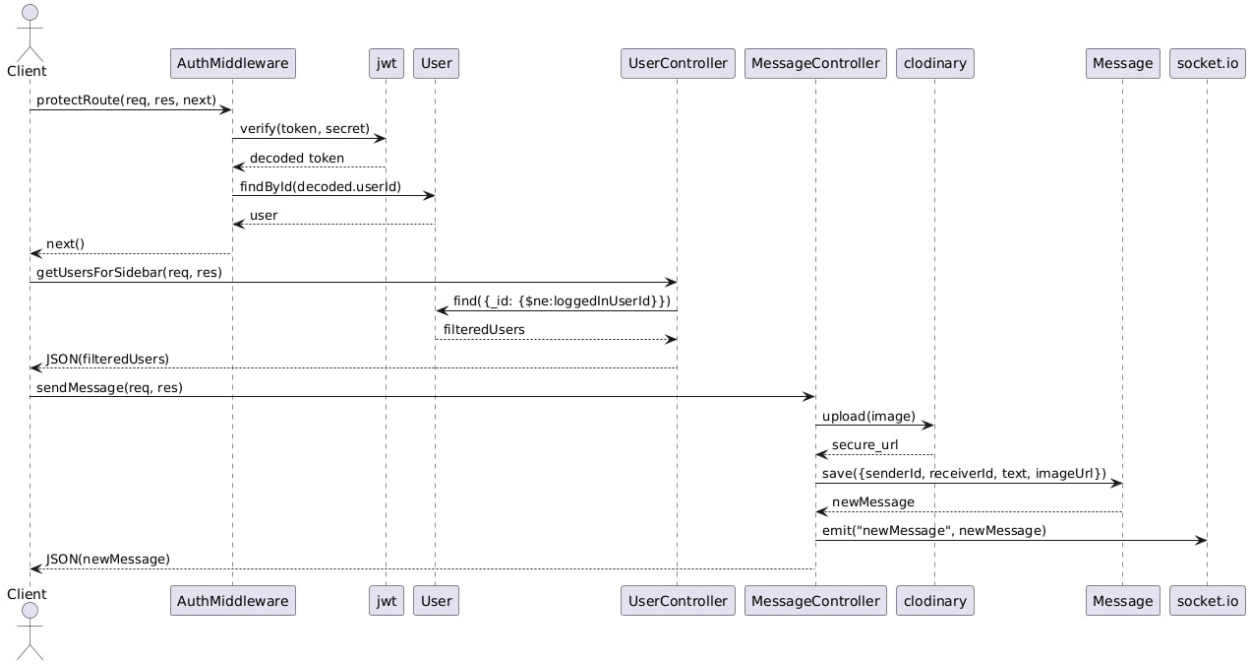


Рисунок 3.3 – Послідовність взаємодії клієнта із сервером під час автентифікації, отримання списку користувачів і надсилання повідомлення

3.2 Аналіз отриманих результатів

Застосунок, що був впроваджений для подальшого комунікування всіх показав свою функціональність та виконує поставлені задачі при плануванні. Зазначена модель відносно ключової суті на етапі користувача та повідомлення дозволяє забезпечити дуже чітку та зрозумілу всю структуру даних, а це в свою чергу спрощує в подальшому логістичний ланцюжок. В момент коли база даних може працювати в різні моменти, які реалізовані через Promise, це насамперед підвищує саму продуктивність системи, коли йде навантаження.

Значним важелем цієї системи стало об’єднання механізму підтвердження ідентичності на основі JWT (JSON Web Token), а це дає гарантовану безпеку в подальшому для користувачів та захищає зазначені джерела від шахрайських дій. А от використання проміжного програмного забезпечення по типу (middleware) AuthMiddleware має змогу на обробку вже загальної ідентичності, що дає можливість знизити дублювання коду та підвищує різноманітність будови. А це, в

свою чергу дає змогу легко використовувати розширюванні системи, щоб додавати нових користувачів, але при цьому не змінювати код, який вже існує.

Важливою деталлю також є WebSocket-технології (Socket.IO), він організовує двонаправленність щодо обміну повідомленнями між сервером та замовником і все це в реальному часі. А це приводить до швидкості вирішення питань, що має велике значення в комунікації усіляких платформ. З'єднання та відповідність у користувачів здійснено через структури Map у класі SocketManager, що знову ж таки призводить до оптимізації декількох сесій і не приводить до затримок, коли доставляються повідомлення.

Щоб відбувалася якісна робота мультимедійного контекту у вигляді зображень була використана система сервісу Cloudinary, а це пришвидшує завантаження, зберігання вищезазначених зображень та знімає навантаження з серверу, що знову ж такі впливає на якість, швидкість користувацького попиту та дає надійність в роботі.

Зроблений аналіз між взаємодією різних напрямків зробленою системою показав, що унікальність застосована навіть дуже правильно та відповідає принципам роботи як окремої частини, вона безпечна і багатозадачна.

Дивлячись на результати, які отримані по факту: система показала високу незмінність та стабільність при впровадженні функцій комунікації, але має розширений потенціал для подальшого коригування та впровадження ідей.

Всі напрямки, що можна буде вдосконалити це розширити функціональність за рахунок впровадження групових чатів,вдосконалення щодо фільтрації спаму та звісно покращення досвіду в користуванні, а ще розширена можливість у поліпшенні поглядів інтерфейсу, щоб була зручність, швидкість та адаптативність.

Тобто, сама система, що впроваджена все ж таки є доволі надійною відносно як побудови сучасної комунікаційної платформи та може спокійно бути використана для подальшого створення більших складних по функціоналу та наповненості платформ, які задовольнять більш широку аудиторію користувачів.

Висновки до розділу 3

В цьому розділі було здійснено проєктування, щодо додатку PurrChat. В основному у цьому розділи були задіяні діаграми сутностей (ER-діаграми) та діаграми послідовності, які впевнено показують логічний ланцюжок взаємодії між компонентами.

Будова ER-діаграми дає змогу визначити ключову суть на прикладі: User (користувач) та Message (повідомлення),дає чіткий опис між їх особливістю та взаємозв’язку.

Окремо було встановлено двосторонній зв'язок між користувачами завдяки тому, що повідомлення має підтвердження відправника та отримувача у зв’язку з наявністю двох зовнішніх ключів.

Авторизація була зроблена як токен та проходить перевірку за рахунок middleware, тобто з бази даних ми отримуємо користувача, а вже потім здійснюється перевірка користувацького списку

В процесі, коли відбувається відправка повідомлення показано, що зображення має процес завантаження у хмарне облако Cloudinary, з’являється нове повідомлення, а вже потім відтворюється через socket.io до безпосереднього користувача.

Зазначена будова має всі реальні можливості для подальшого використання на сучасних вебпорталах бо вона чітка та зорієнтована саме для цього.

При створенні застосунку була можливість отримати відбудоване уявлення про всі запроваджені системні компоненти, іх функціонал та зв'язок і це дає змогу забезпечити всю базу, щоб в подальшому реалізувати та й ще облегшує супровід та розкриває подальші можливості в розвиненні.

Зазначена модель показала себе надійною сходинкою для подальшої будови розширеного за стосунку з більш вагомими функціями, щоб мати змогу обмінюватися повідомленнями в режимі онлайн.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ ВЕБЗАСТОСУНКУ ДЛЯ КОРПОРАТИВНОЇ КОМУНІКАЦІЇ

4.1 Клієнтська частина вебзастосунку

При створені вебзастосунку була використана бібліотека React. Для початку у головному файлі App.jsx на (рис 4.1) ми зустрінемо:

```
useEffect(() => {
  checkAuth();
}, [checkAuth]);
```

Рисунок 4.1 – Функціональний компонент з хуком useEffect, для перевірки авторизації користувача

На (рис 4.2) для маршрутизації використовується react-router-dom. У App.jsx визначено 5 маршрутів для сторінок за допомогою компонентів <Routes> та <Route>. Якщо користувач не зареєстрований його перенаправить у LoginPage:

```
<Routes>
  <Route path="/" element={authUser ? <HomePage /> : <Navigate to="/login" />} />
  <Route path="/signup" element={!authUser ?  <SignUpPage /> : <Navigate to="/" />} />
  <Route path="/login" element={!authUser ? <LoginPage /> : <Navigate to="/" />} />
  <Route path="/settings" element={<SettingsPage />} />
  <Route path="/profile" element={authUser ? <ProfilePage /> : <Navigate to="/login" />} />
</Routes>
```

Рисунок 4.2 – Налаштування маршрутів в залежності від авторизації користувача

Інтерфейс використуває фреймворк Tailwind CSS, який налаштований у файлі tailwind.config.js також використовується плагін daisyUI, для забезпечення тем у застосунку. Ця тема зберігається у localStorage та керується за допомогою хука create із використанням бібліотеки zustand у файлі useThemeStore.js на (рис. 4.3) продемонстрировано використання хука та створенням кастомного store для доступу до теми та її подальшої зміни.

```
import { create } from "zustand";

export const useThemeStore = create((set) => ({
    theme: localStorage.getItem("chat-theme") || "dark",
    setTheme: (theme) => {
        localStorage.setItem("chat-theme", theme);
        set({ theme });
    },
}));
```

Рисунок 4.3 – Створення стану для теми за допомогою хука create з бібліотеки zustand та збереженням її у localStorage

Ця тема динамічно використовується у App.jsx за допомогою хука const {theme} = useThemeStore() та атрибута який імпортується з useThemeStore.js.

Логіка авторизації була реалізована через глобальний стан useAuthStore.js, який був створений за допомогою Zustand. До прикладу, метод login відправляє дані на серверну частину, зберігає користувача в стані та встановлює WebSocket-з'єднання, демонстрація на (рис 4.4):

```
login: async (data) => {
    set({ isLoggingIn: true });
    try {
        const res = await axiosInstance.post("/auth/login", data);
        if (res && res.data) {
            set({ authUser: res.data });
            toast.success("Logged in successfully");

            get().connectSocket()
        } else {
            throw new Error("Invalid response format");
        }
    } catch (error) {
        console.error("Login error:", error);
        const errorMessage = error.response ? error.response.data.message : error.message;
        toast.error(errorMessage || "An error occurred during login");
    } finally {
        set({ isLoggingIn: false });
    }
},
```

Рисунок 4.4 – Метод авторизації користувача за відповідним route та підключенням до сокета

Також глобальний стан чату через useChatStore.js який керує повідомленнями, користувачами та отримує повідомлення в реальному часі через WebSocket, демонстрація на (рис 4.5):

```
socket.on("newMessage", (newMessage) => {
  const currentMessages = get().messages;
  const isDuplicate = currentMessages.some(msg => msg._id === newMessage._id);
  const isRelevantMessage =
    (newMessage.senderId === authUser._id && newMessage.receiverId === selectedUser._id) ||
    (newMessage.senderId === selectedUser._id && newMessage.receiverId === authUser._id);

  if (!isDuplicate && isRelevantMessage) {
    set({ messages: [...currentMessages, newMessage] });
  }
});
```

Рисунок 4.5 – Обробка отримання повідомлення

Отже, PurrChat це вебзастосунок, який був побудований на сучасному стеку технологій, таких як React, Tailwind CSS та WebSockets та різних бібліотек, хуків, що забезпечують високу реактивність інтерфейсу.

4.1.1 Опис основних сторінок інтерфейсу

У PurrChat є 5 основних сторінок. LoginPage, SingUpPage, HomePage, ProfilePage, SettingsPage. У цьому підрозділі ми опишимо ці сторінки та за що вони відповідають.

LoginPage – сторінка авторизації для користувача. При введені електроної пошти та паролю додаток валідує дані і якщо немає ніяких помилок чи проблем з даними відправляє запит на вхід до системи. На сторінці є індикатор завантаження та обробник помилок. Якщо неправильний email чи пароль, а також є перемикач для відображення пароля, демонстрація на (рис 4.6):

```
const validateForm = () => {
    if (!formData.email.trim()) return toast.error("Email is required");
    if (!/\S+@\S+\.\S+/.test(formData.email)) return toast.error("Invalid email format");
    if (!formData.password) return toast.error("Password is required");
    if (formData.password.length < 6) return toast.error("Password must be at least 6 characters");

    return true;
};
```

а)

```
const handleSubmit = async (e) => {
    e.preventDefault();
    if (!validateForm()) return;

    try {
        await login(formData);
    } catch (error) {
        console.error("Login failed:", error);

        if (error?.response?.data?.message) {
            toast.error(error.response.data.message);
        } else {
            toast.error("Failed to log in");
        }
    }
};
```

б)

```
{error && (
    <div className="text-center mt-4 text-red-500">
        <p>{error}</p>
    </div>
)}
```

в)

Рисунок 4.6 – Валідація форми для перевірки полів email та password (а), обробка відправки форми (б) та відображення помилки для користувача (в)

На (рис 4.7), можна побачити загальний вигляд сторінки LoginPage

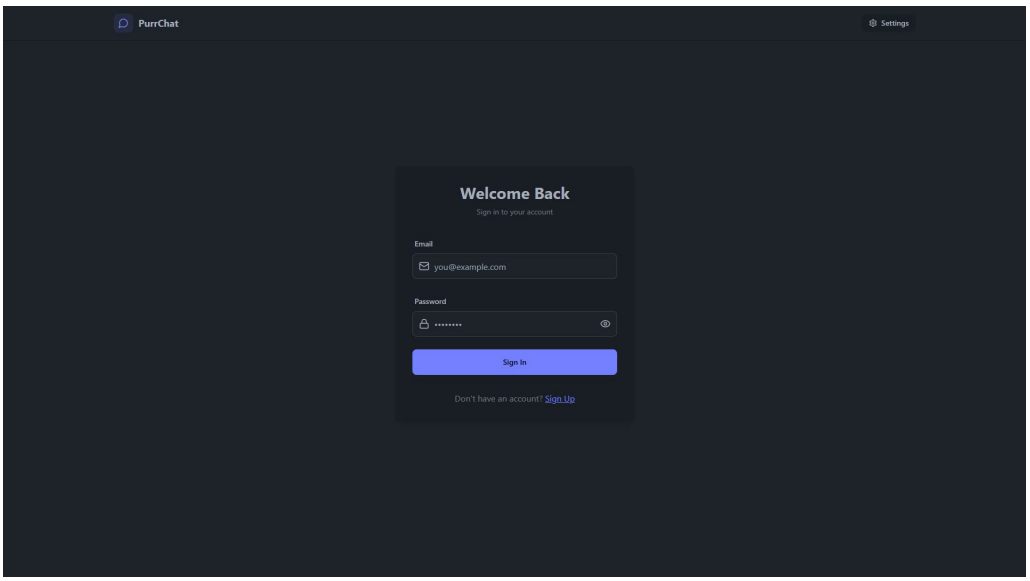


Рисунок 4.7 – Загальний вигляд сторінки LoginPage

SignUpPage – це сторінка для реєстрації нового користувача. Користувач повинен ввести ім’я, електронну пошту та пароль. Потім це валідується на коректність і якщо нема помилок, це все відправляється запит на backend частину. Як і у LoginPage є індекатор завантаження, оброка помилок, перемикач для паролю. Якщо реєстрація не була успішною, користувач отримає повідомлення про помилку. Якщо все ж реєстрація вдалася, то користувача перекидує на сторінку для входу, демонстпація на (рис 4.8-4.9).

```
const handleSubmit = (e) => {
    e.preventDefault();
    if (validateForm()) {
        signup(formData).catch(error => {
            toast.error(error.message || "Registration failed");
        });
    }
};
```

Рисунок 4.8 – Обровка відправки форми

```
<div className="text-center pt-4">
    <p className="text-base-content/60">
        Already have an account?{" "}
        <Link to="/login" className="link link-primary">
            Sign in
        </Link>
    </p>
</div>
```

Рисунок 4.9 – Посилання для переходу на сторінку для входу

HomePage – це головна сторінка куди потрапляє користувач після входу. На ній користувач може побачити бокову панель (Sidebar) на якій є список зареєстрованих користувачів і зони для чату. Якщо користувач не перебуває з кимось у чаті відображається компонент NoChatSelected, інакше відкривається компонент ChatContainer, щоб відобразити чат з іншим користувачем, лістинг коду та загальний вигляд сторінки HomePage на (рис 4.10 - 4.11).

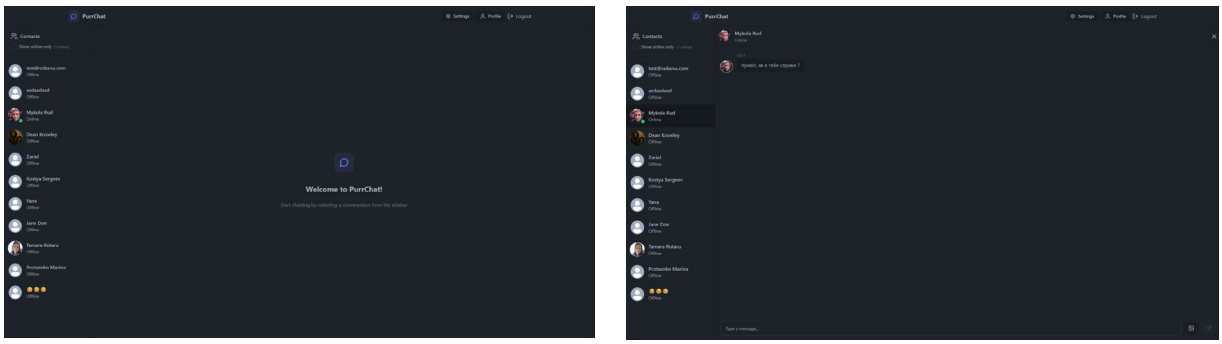
```
import ChatContainer from "../components/ChatContainer";
import NoChatSelected from "../components/NoChatSelected";
import Sidebar from "../components/Sidebar";
import { useChatStore } from "../store/useChatStore";

const HomePage = () => {
    const { selectedUser } = useChatStore();

    return (
        <div className="h-screen bg-base-200 flex flex-col">
            <div className="h-[65px]"></div>
            <div className="flex-1 w-full overflow-hidden">
                <div className="h-full w-full bg-base-100">
                    <div className="flex h-full">
                        <Sidebar />
                        {!selectedUser ? <NoChatSelected /> : <ChatContainer />}
                    </div>
                </div>
            </div>
        </div>
    );
};

export default HomePage;
```

Рисунок 4.10 – Розмітка компоненту HomePage зі стилями TailwindCSS та логікою відображання компонентів NoChatSelected та ChatContainer



а) б)

Рисунок 4.11 – HomePage з активним компонентом NoChatSelected (а) та HomePage з активним компонентом ChatContainer

ProfilePage – це сторінка, де користувач може відредагувати свій профіль, а саме може оновити ім’я, email, змінити аватарку або оновити пароль. Також присутні правила валідації пошти, щоб користувач, не зміг ввести некоректну пошту, обмеження на довжину імені, від трьох символів, а також, це все супроводжується повідомленнями про успіх зміни у профілі або невдачу, якщо є якась помилка чи користувач ввів щось не коректне, система про це попередить, демонстрація на (рис 4.12-4.17).

```
const validateEmail = (email) => {
  const emailRegex = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;

  if (!emailRegex.test(email)) return false;

  if (/[^\w.%+-@]/.test(email)) return false;

  if (email.startsWith('.') || email.endsWith('.') || email.includes('..')) {
    return false;
  }

  const atCount = (email.match(/@/g) || []).length;
  if (atCount !== 1) return false;

  const parts = email.split('@');
  if (parts.length !== 2) return false;

  const [localPart, domain] = parts;

  if (localPart.length > 64) return false;

  if (domain.length > 253) return false;

  const domainParts = domain.split('.');
  if (domainParts.some(part => part.length > 63)) return false;

  return true;
};
```

Рисунок 4.12 – Функція, яка відповідає за валідацію пошти

```
const handleImageUpload = async (e) => {
  if (isUpdatingProfile) return;

  const file = e.target.files?.[0];
  if (!file) return;

  if (!file.type.startsWith("image/")) {
    toast.error("Please upload an image file");
    return;
  }

  const reader = new FileReader();
  reader.onload = async () => {
    const base64Image = reader.result;
    setSelectedImg(base64Image);
    try {
      await updateProfile({ profilePic: base64Image });
      toast.success("Profile picture updated", { id: "avatar-success" });
    } catch (err) {
      toast.error(err.message || "Failed to update profile picture");
      setSelectedImg(authUser?.profilePic || null);
    }
  };
  reader.readAsDataURL(file);
};
```

Рисунок 4.13 – Функція, яка відповідає за валідацію пошти

```
const handleSavePassword = async () => {
  if (isUpdatingProfile) return;

  toast.dismiss();

  try {
    if (!password && !confirmPassword) {
      toast.success("No changes detected");
      setPassword("");
      setConfirmPassword("");
      setIsEditingPassword(false);
      return;
    }

    if (!password || !confirmPassword) {
      toast.error("Please fill both password fields");
      return;
    }

    if (password !== confirmPassword) {
      toast.error("Passwords do not match");
      return;
    }

    if (password.length < 6) {
      toast.error("Password must be at least 6 characters");
      return;
    }

    await updateProfile({ password });
    toast.success("Password updated successfully");
    setPassword("");
    setConfirmPassword("");
    setIsEditingPassword(false);
  } catch (err) {
    if (err.message === "NEW_PASSWORD_SAME_AS_CURRENT") {
      toast.error("New password must be different from current one");
    } else {
      toast.error(err.message || "Failed to update password");
    }
  }
};
```

Рисунок 4.14 – Функція, яка відповідає за збереження та валідацію паролю

```
const handleSaveProfile = async () => {
  if (isUpdatingProfile || saveLock || !isEditingProfile) return;
  setSaveLock(true);
  toast.dismiss();

  try {
    if (fullName.trim().length < 3) {
      toast.error("Name must be at least 3 characters long", { id: "profile-error" });
      setFullName(prevFullName);
      setEmail(prevEmail);
      return;
    }

    if (!validateEmail(email)) {
      toast.error("Please enter a valid email address (e.g., example@domain.com)", { id: "profile-error" });
      setFullName(prevFullName);
      setEmail(prevEmail);
      return;
    }

    const updates = {};
    if (email !== prevEmail) updates.email = email;
    if (fullName !== prevFullName) updates.fullName = fullName;

    if (Object.keys(updates).length === 0) {
      toast.success("No changes detected");
      setIsEditingProfile(false);
      return;
    }

    await updateProfile(updates);

    toast.success("Profile updated successfully");
    setPrevFullName(fullName);
    setPrevEmail(email);
    setIsEditingProfile(false);
  } catch (err) {
    console.error("Update error:", err);
    toast.error(err.message || "Failed to update profile");
    setFullName(prevFullName);
    setEmail(prevEmail);
  } finally {
    setSaveLock(false);
  }
};
```

Рисунок 4.15 –Функція, яка відповідає за перевірку змінених полів по всім прописаним правилам

```
updateProfile: async (data) => {
    set({ isUpdatingProfile: true });
    try {
      const res = await axiosInstance.put("/auth/update-profile", data);
      const updatedUser = { ...get().authUser, ...res.data };
      set({ authUser: updatedUser });
      return true;
    } catch (error) {
      if (error.response?.data?.message === "NEW_PASSWORD_SAME_AS_CURRENT") {
        throw new Error("NEW_PASSWORD_SAME_AS_CURRENT");
      }
      throw new Error(error.response?.data?.message || "Update failed");
    } finally {
      set({ isUpdatingProfile: false });
    }
},
```

Рисунок 4.16 – Метод який подає запит на змінення профілю

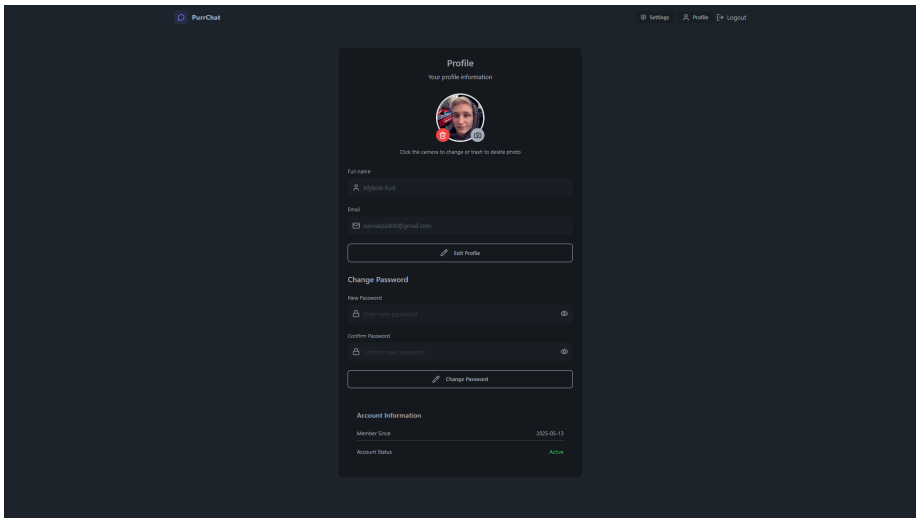


Рисунок 4.17 – Загальний вигляд сторінки ProfilePage

SettingsPage – це сторінка, де користувач може вибрати одну тему на вибір з 32 можливих. А також на сторінці присутнє прев’ю яке відповідає вибраній темі, як на (рис 4.18-4.19).

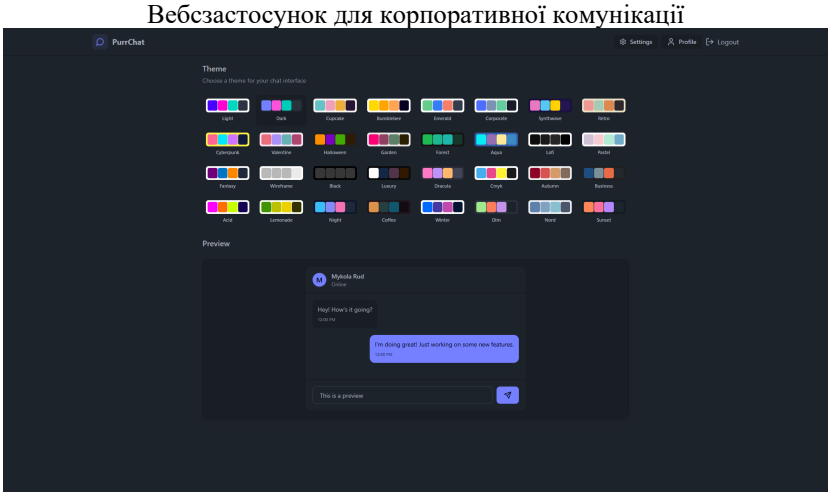
```jsx
<div className="grid grid-cols-4 sm:grid-cols-6 md:grid-cols-8 gap-2">
  {THEMES.map((t) => (
    <button
      key={t}
      className={`
        group flex flex-col items-center gap-1.5 p-2 rounded-lg transition-colors
        ${theme === t ? "bg-base-200" : "hover:bg-base-200/50"}
      `}
      onClick={() => setTheme(t)}
    >
      <div className="relative h-8 w-full rounded-md overflow-hidden" data-theme={t}>
        <div className="absolute inset-0 grid grid-cols-4 gap-px p-1">
          <div className="rounded bg-primary"></div>
          <div className="rounded bg-secondary"></div>
          <div className="rounded bg-accent"></div>
          <div className="rounded bg-neutral"></div>
        </div>
      </div>
      <span className="text-[11px] font-medium truncate w-full text-center">
        {t.charAt(0).toUpperCase() + t.slice(1)}
      </span>
    </button>
  ))}
</div>
```

а)

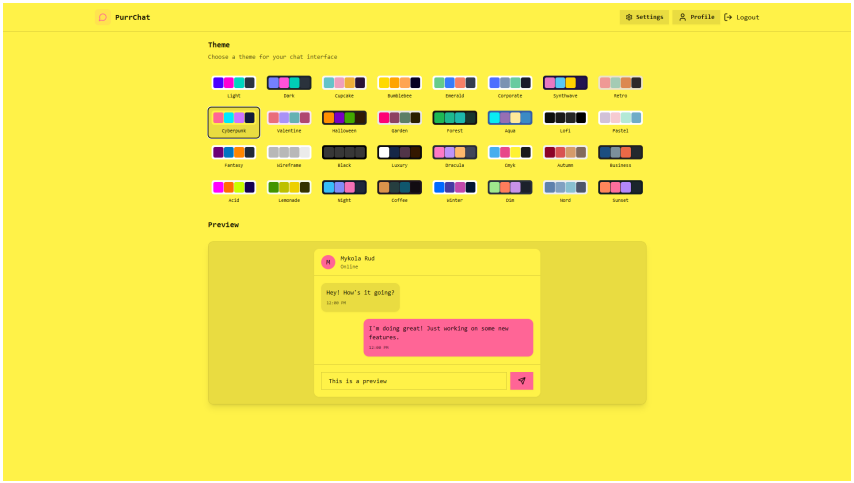
```jsx
export const THEMES = [
    "light",
    "dark",
    "cupcake",
    "bumblebee",
    "emerald",
    "corporate",
    "synthwave",
    "retro",
    "cyberpunk",
    "valentine",
    "halloween",
    "garden",
    "forest",
    "aqua",
    "lofi",
    "pastel",
    "fantasy",
    "wireframe",
    "black",
    "luxury",
    "dracula",
    "cmyk",
    "autumn",
    "business",
    "acid",
    "lemonade",
    "night",
    "coffee",
    "winter",
    "dim",
    "nord",
    "sunset",
];
```

б)

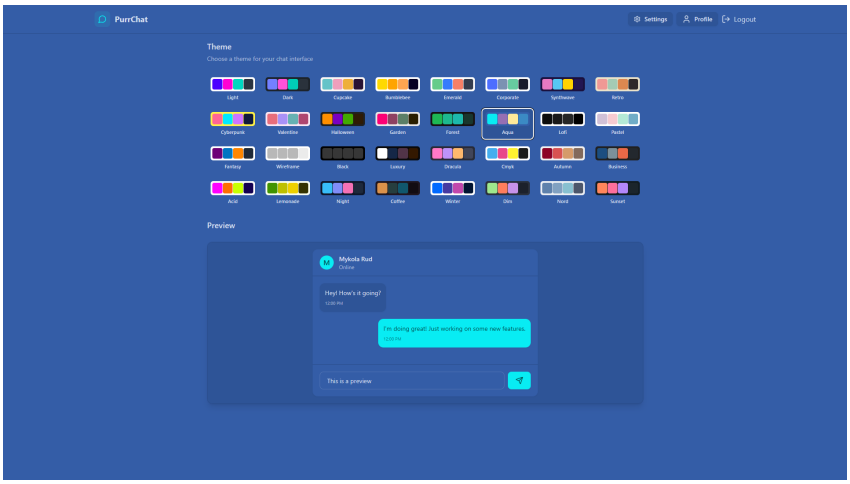
Рисунок 4.18 – Структура, яка відповідає, щоб користувач міг обрати тему (а) та масив збереженій у змінній THEMES з усіма темами застосунку з плагіну DaisyUI



а)



б)



в)

Рисунок 4.19 – Загальний вигляд сторінки Settings Page та темами Dark (а), Cyberpunk (б), Auqa (в)

4.1.2 Ключові компоненти вебзастосунку

ChatContainer – це одна з головних частин інтерфейсу цього вебзастосунку. Він відображає повідомлення, відповідає за їх редагування та видалення тексту або зображення. А також відповідає за обробку нових повідомлень. Компонент працює з useChatStore, може скорлити до останнього повідомлення.

На (рис 4.20-4.35), для інтеграції з useChatStore, компонент ChatContainer імпортує та деструктурує з нього методи та стани.

```
const ChatContainer = () => {
  const {
    messages,
    getMessages,
    isMessagesLoading,
    selectedUser,
    deleteMessage,
    editMessage,
    subscribeToMessages,
    unsubscribeToMessages,
  } = useChatStore();
```

Рисунок 4.20 – Деструктиризація методів з useChatStore

```
useEffect(() => {
  getMessages(selectedUser._id);
  subscribeToMessages();
  return () => unsubscribeToMessages();
}, [selectedUser._id, getMessages, subscribeToMessages, unsubscribeToMessages]);
```

Рисунок 4.21 – Хук який відповідає підписання та відписання від повідомлень

```
useEffect(() => {
  if (messageEndRef.current && messages) {
    messageEndRef.current.scrollIntoView({ behavior: "smooth" });
  }
}, [messages]);
```

Рисунок 4.22 – Хук який відповідає за автоматичне гортання до останього повідомлення

```
const handleEdit = (id, text) => {
  setEditingId(id);
  setEditedText(text || "");
};
```

Рисунок 4.23 – Обробник редагування

```
const handleSave = async (id) => {
  const originalMessage = messages.find((msg) => msg._id === id);
  const trimmedText = editedText.trim();

  if (trimmedText === originalMessage?.text) {
    toast("No changes detected", { icon: <Info className="w-5 h-5" /> });
    setEditingId(null);
    setEditedText("");
    return;
  }

  if (trimmedText === "" && !originalMessage?.image) {
    await deleteMessage(id);
    toast.success("Message deleted!");
  } else {
    await editMessage(id, trimmedText, originalMessage?.image);
    toast.success("Message updated!");
  }
  setEditingId(null);
  setEditedText("");
};
```

Рисунок 4.24 – Обробник збереження змін повідомлення

```
const handleRemoveImage = async (message) => {
  if (confirmRemoveImageId === message._id) {
    try {
      if (!message.text?.trim()) {
        await deleteMessage(message._id);
        toast.success("Message deleted!");
      } else {
        await editMessage(message._id, message.text, null);
        toast.success("Image removed!");
      }
      setConfirmRemoveImageId(null);
    } catch {
      toast.error("Failed to remove image");
    }
  } else {
    setConfirmRemoveImageId(message._id);
    setTimeout(() => setConfirmRemoveImageId(null), 3000);
  }
};
```

Рисунок 4.25 – Обробник для видалення зображень

```
const handleRemoveText = async (message) => {
  if (confirmRemoveTextId === message._id) {
    try {
      if (!message.image) {
        await deleteMessage(message._id);
        toast.success("Message deleted!");
      } else {
        await editMessage(message._id, "", message.image);
        toast.success("Text removed!");
      }
      setConfirmRemoveTextId(null);
    } catch {
      toast.error("Failed to remove text");
    }
  } else {
    setConfirmRemoveTextId(message._id);
    setTimeout(() => setConfirmRemoveTextId(null), 3000);
  }
};
```

Рисунок 4.26 – Обробник для видалення тексту

```
const handleReplaceImage = async (message) => {
  const fileInput = document.createElement("input");
  fileInput.type = "file";
  fileInput.accept = "image/*";
  fileInput.onchange = async (e) => {
    const file = e.target.files?.[0];
    if (file) {
      try {
        if (file.name === message.imageFileName) {
          toast("Image is the same as the previous one.", { icon: <Info className="w-5 h-5" /> });
          return;
        }
        const reader = new FileReader();
        reader.onloadend = async () => {
          const base64 = reader.result;
          await editMessage(message._id, message.text || "", base64);
          toast.success("Image updated!");
        };
        reader.readAsDataURL(file);
      } catch {
        toast.error("Failed to update image");
      }
    }
  };
  fileInput.click();
};
```

Рисунок 4.27 – Обробник заміни зображення

```
getMessages: async (userId) => {
  set({ isMessagesLoading: true });
  try {
    const res = await axiosInstance.get(`/messages/${userId}`);
    set({ messages: res.data });
  } catch (error) {
    toast.error(error.response.data.message);
  } finally {
    set({ isMessagesLoading: false });
  }
},
```

Рисунок 4.28 – Метод, який відправляє запит на серверну частину для отримання повідомлень

```
subscribeToMessages: () => {
  const { selectedUser } = get();
  const socket = useAuthStore.getState().socket;
  const authUser = useAuthStore.getState().authUser;

  if (!selectedUser || !socket || !socket.connected) return;

  socket.off("newMessage");
  socket.off("messageUpdated");
  socket.off("messageDeleted");

  socket.on("newMessage", (newMessage) => {
    const currentMessages = get().messages;
    const isDuplicate = currentMessages.some(msg => msg._id === newMessage._id);
    const isRelevantMessage =
      (newMessage.senderId === authUser._id && newMessage.receiverId === selectedUser._id) ||
      (newMessage.senderId === selectedUser._id && newMessage.receiverId === authUser._id);

    if (!isDuplicate && isRelevantMessage) {
      set({ messages: [...currentMessages, newMessage] });
    }
  });

  socket.on("messageUpdated", (updatedMessage) => {
    const updatedMessages = get().messages.map(msg =>
      msg._id === updatedMessage._id ? updatedMessage : msg
    );
    set({ messages: updatedMessages });
  });

  socket.on("messageDeleted", (deletedMessageId) => {
    const filteredMessages = get().messages.filter(msg => msg._id !== deletedMessageId);
    set({ messages: filteredMessages });
  });
},
```

Рисунок 4.29 – Метод, який відповідає за повідомлення, які відправляються, редагуються або видаляються у реальному часі за допомогою Socket.IO

```
unsubscribeToMessages: () => {
  const { socket } = get();
  if (socket) {
    socket.off("newMessage");
  }
},
```

Рисунок 4.30 – Метод, який відповідає за відписання від сокета «newMessage»

Метод unsubscribeToMessages потрыбен, щоб перестати отримувати повідомлення в реальному часі, наприклад, якщо користувал змінить чат або вийде зовсім.

```
editMessage: async (messageId, text, image) => {
  try {
    const { messages } = get();
    const res = await axiosInstance.put(`/messages/${messageId}`, { text, image });

    const updatedMessages = messages.map(msg =>
      msg._id === messageId ? res.data : msg
    );

    set({ messages: updatedMessages });
    return res.data;
  } catch (error) {
    toast.error(error.response.data.message);
    throw error;
  }
},
```

Рисунок 4.31 – Метод, який відправляє запит на серверну частину для редагування повідомлень

```
deleteMessage: async (messageId) => {
  try {
    const { messages } = get();
    await axiosInstance.delete(`/messages/${messageId}`);

    const updatedMessages = messages.filter(msg => msg._id !== messageId);
    set({ messages: updatedMessages });
  } catch (error) {
    toast.error(error.response.data.message);
    throw error;
  }
},
```

Рисунок 4.32 – Метод, який відповляє запит на серверну частину для видалення повідомлень

```
export function formatMessageTime(date) {
  return new Date(date).toLocaleTimeString("en-US", {
    hour: "2-digit",
    minute: "2-digit",
    hour12: false,
  });
}
```

Рисунок 4.33 – Функція, для відображання часу у додатку

```
<time className="text-xs opacity-50">
  {formatMessageTime(message.createdAt)}
</time>
```

Рисунок 4.34 – застосування функції у UI

Рисунок 4.35 – Загальний вигляд компоненту ChatContainer

Компонент ChatHeader (рис 4.36) – він відповідає за відображання шапки чату, а саме за показ аватарки ім’я та статусу online/offline. Є кнопка для закриття чату при натискані на яку можна вийти з поточного чату. Загалом цей компонент слугує за відображання заголовка чату та керуванням виходу з нього.

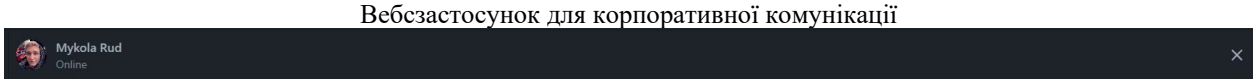


Рисунок 4.36 – Загальний вигляд компоненту ChatHeader

Компонент MessageInput – він відповідає за написання та відправку текстових повідомлень або відправка картинки з текстом чи без нього, код та UI вигляд на(рис 4.37 - 4.38):

```
sendMessage: async (messageData) => {
  const { selectedUser, messages } = get();
  try {
    const res = await axiosInstance.post(`/messages/send/${selectedUser._id}`, messageData);
    set({ messages: [...messages, res.data] });
  } catch (error) {
    toast.error(error.response.data.message);
  }
},
```

Рисунок 4.37 – Метод, який відповідає за відправку повідомлень

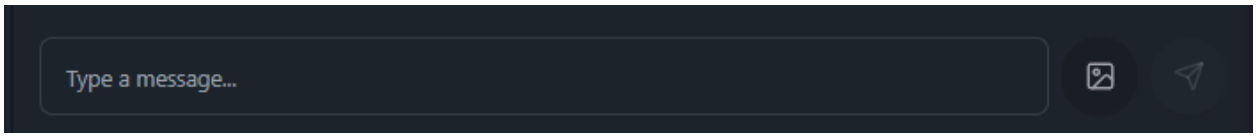


Рисунок 4.38 – Загальний вигляд компоненту MessageInput

NoChatSelected –цей компонент відображається коли користувач не знаходиться з кимось у чаті. Він слугує як вітальне вікно з користувачем, він продемострований на (рис 4.39):

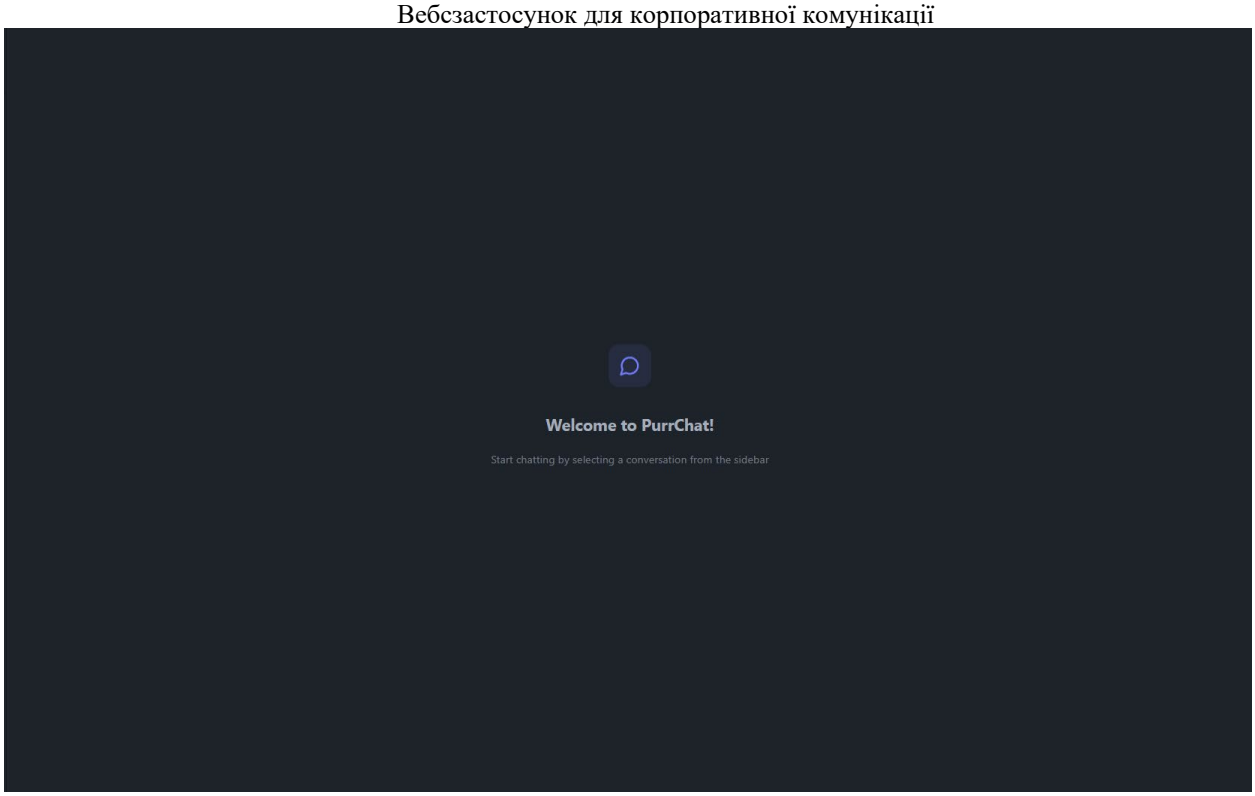


Рисунок 4.39 – Загальний вигляд компоненту NoChatSelected

4.1.3 Навігація та побічні елементи

NavBar – (рис 4.40), це компонент для навігації у PurrChat на якому відображається логотип застосунку та кнопки для переходу до сторінок налаштування та профілю, а також є кнопка для виходу з акаунту. При натискані на логотип він повертає на головну сторінку або змінює компонент ChatContainer на NoChatSelected.

Рисунок 4.40 – Загальний вигляд компоненту NavBar

SideBar – це компонент бічної навігації, яка показує список користувачів у чаті з можливістю фільтрації лише за тими хто онлайн. Також вона слугує, для вибору співрозмовника, а також автоматично підвантажує користувачів при запуску вебзастосунку, демострація коду та UI дизайну на (рис 4.41-4.42):

```
const filteredUsers = showOnlineOnly
  ? users?.filter(user => onlineUsers.includes(user._id)) || []
  : users || [];
```

а)

```
<div className="overflow-y-auto w-full py-3">
  {filteredUsers.map((user) => (
    <button
      key={user._id}
      onClick={() => setSelectedUser(user)}
      className={`
        w-full p-3 flex items-center gap-3
        hover:bg-base-300 transition-colors
        ${selectedUser?._id === user._id ? "bg-base-300 ring-1 ring-base-300" : ""}
      `}
    >
```

б)

Рисунок 4.41 – Фультрація користувачів, щоб показувати лише тих хто в онлайні (а) та використання цієї логіки у UI

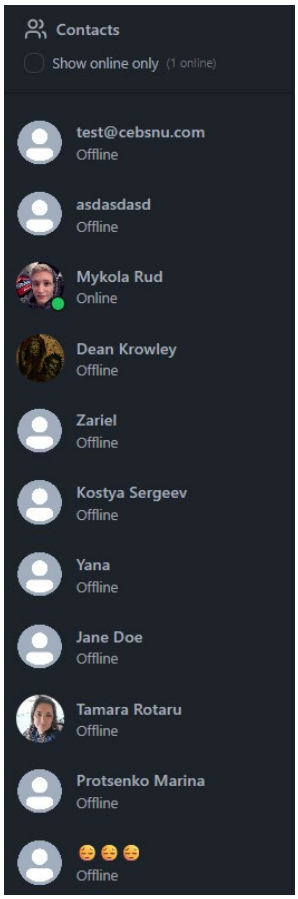


Рисунок 4.42 – Загальний вигляд компоненту SideBar

На сайті використовуються Skeleton компоненти, такі як MessageSkeleton та SidebarSkeleton, які використовуються під час завантаження контенту. SidebarSkeleton викликається під час завантаження Sidebar коли той завантажує список користувачів та MessageSkeleton завантажується під час того коли завантажується повідомлення чату у компоненті ChatContainer. Це продемонстровано на рис (4.43-4.49):

```jsx
import { Users } from "lucide-react";

const SidebarSkeleton = () => {
  // Create 8 skeleton items
  const skeletonContacts = Array(8).fill(null);

  return (
    <aside
      className="h-full w-20 lg:w-72 border-r border-base-300
    flex flex-col transition-all duration-200"
    >
      {/* Header */}
      <div className="border-b border-base-300 w-full p-5">
        <div className="flex items-center gap-2">
          <Users className="w-6 h-6" />
          <span className="font-medium hidden lg:block">Contacts</span>
        </div>
      </div>

      {/* Skeleton Contacts */}
      <div className="overflow-y-auto w-full py-3">
        {skeletonContacts.map((_, idx) => (
          <div key={idx} className="w-full p-3 flex items-center gap-3">
            {/* Avatar skeleton */}
            <div className="relative mx-auto lg:mx-0">
              <div className="skeleton size-12 rounded-full" />
            </div>

            {/* User info skeleton - only visible on larger screens */}
            <div className="hidden lg:block text-left min-w-0 flex-1">
              <div className="skeleton h-4 w-32 mb-2" />
              <div className="skeleton h-3 w-16" />
            </div>
          </div>
        ))}
      </div>
    </aside>
  );
};

export default SidebarSkeleton;
```

Рисунок 4.43 – розмітка <SidebarSkeleton> компонента

```jsx
if (isUsersLoading) return <SidebarSkeleton />;
```

Рисунок 4.44 – виклик <SidebarSkeleton> компонента

Якщо користувачі ще не завантажились викликається компонент <SidebarSkeleton>

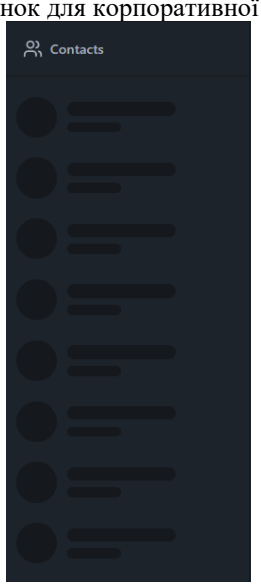


Рисунок 4.45 – Загальний вигляд SidebarSkeleton

```jsx
const MessageSkeleton = () => {
    // Create an array of 6 items for skeleton messages
    const skeletonMessages = Array(6).fill(null);

    return (
      <div className="flex-1 overflow-y-auto p-4 space-y-4">
        {skeletonMessages.map((_, idx) => (
          <div key={idx} className={`chat ${idx % 2 === 0 ? "chat-start" : "chat-end"}`}>
            <div className="chat-image avatar">
              <div className="size-10 rounded-full">
                <div className="skeleton w-full h-full rounded-full" />
              </div>
            </div>

            <div className="chat-header mb-1">
              <div className="skeleton h-4 w-16" />
            </div>

            <div className="chat-bubble bg-transparent p-0">
              <div className="skeleton h-16 w-[200px]" />
            </div>
          </div>
        ))}
      </div>
    );
  };

  export default MessageSkeleton;
```

Рисунок 4.46 – розмітка <MessageSkeleton> компонента

```
if (isMessagesLoading) {
  return (
    <div className="flex-1 flex flex-col overflow-auto">
      <ChatHeader />
      <MessageSkeleton />
      <MessagesInput />
    </div>
  );
}
```

Рисунок 4.47 – виклик <MessageSkeleton> компонента

Якщо повідомлення ще не завантажились викликається компонент <MessageSkeleton >

Рисунок 4.48 – Загальний вигляд MessageSkeleton

Також на сайті використовується Loader при вході чи реєстрації або при завантаженні застосунку з відповідною анімацією <Loader className = "size-10 animate-spin" />

Рисунок 4.49 – Загальний вигляд Loader

4.1.4 Управління станом та стилізація

У цьому застосунку, через Zustand (useAuthStore.js) реалізовано особливості авторизації, де в загальному стані authUser (користувач, що авторизований) та методи для login, signup, logout, updateProfile, а також checkAuth, що дає змогу перевірити активність сесії. Наприклад, checkAuth викликає axiosInstance.get("/auth/check"), що дає змогу підняти дані користувача, якщо він звісно зареєстрований вже, і зберігає їх у стані через set({ authUser: res.data }).Це свого роду дає змогу на автоматичну авторизацію, без необхідності постійно вводити логін. Демонстрація методів на (рис. 4.50-4.52):

```
checkAuth: async () => {
    try {
        const res = await axiosInstance.get("/auth/check");
        set({ authUser: res.data });
        get().connectSocket()

    } catch (error) {
        console.log("Error in checkAuth:", error);
        set({ authUser: null });
    } finally {
        set({ isCheckingAuth: false });
    }
},
```

Рисунок 4.50 – Метод який подає запит на backend для перевірки авторизації

```
signup: async (data) => {
    set({ isSigningUp: true });
    try {
       const res = await axiosInstance.post("/auth/signup", data);
       set({ authUser: res.data });
       toast.success("Account created successfully");
       get().connectSocket()

    } catch (error) {
        toast.error(error.response.data.message);
    } finally {
        set({ isSigningUp: false });
    }
},
```

Рисунок 4.51 – Метод який подає запит на backend для реєстрації акаунт

```
login: async (data) => {
    set({ isLoggingIn: true });
    try {
        const res = await axiosInstance.post("/auth/login", data);
        if (res && res.data) {
            set({ authUser: res.data });
            toast.success("Logged in successfully");

            get().connectSocket()
        } else {
            throw new Error("Invalid response format");
        }
    } catch (error) {
        console.error("Login error:", error);
        const errorMessage = error.response ? error.response.data.message : error.message;
        toast.error(errorMessage || "An error occurred during login");
    } finally {
        set({ isLoggingIn: false });
    }
},
```

Рисунок 4.52 – Метод який подає запит на backend для входу у акаунта

```
logout: async () => {
    try {
      await axiosInstance.post("/auth/logout");
      set({ authUser: null });
      toast.success("Logged out successfully");
      get().disconnectSocket()
    } catch (error) {
      toast.error(error.response.data.message);
    }
},
```

Рисунок 4.53 – Метод який подає запит на backend для входу виходу з акаунта

Через useAuthStore також відбувається підключення socket.io де спосіб connectSocket() відчиняє сокет-з’єднання зразу після того, як користувач авторизувався: query: { userId: authUser._id }.

Тобто, це говорить про те, що усі сокет-події прив’язані до сеансу, який активний і при logout викликається disconnectSocket, який фактично розриває з’єднання метод продемонстрований на (рис 4.54):.

```
connectSocket: () => {
    const { authUser } = get()
    if(!authUser || get().socket?.connected) return;

    const socket = io(BASE_URL, {
        query: {
            userId: authUser._id,
        },
    })
    socket.connect()
    set({ socket: socket});

    socket.on("getOnlineUsers", (userIds) => {
        set({onlineUsers: userIds})
    });
},

disconnectSocket: () => {
    if(get().socket?.connected) get().socket.disconnect();
},
```

Рисунок 4.54 – Методи для підключення та відключення Socket.IO, щоб додаток працював у реальному часі

Через withCredentials відбувається збереження токенів на сервері: true у axios.js, що дає змогу забезпечити передачу cookie з токеном між фронтендом і сервером. Це говорить про те, що користувач не користується токенами напряму у LocalStorage чи SessionStorage, а може опиратися на безпечне cookie-збереження, що виключає можливість XSS-доступ, як на (рис 4.55):

```
import axios from "axios";

export const axiosInstance = axios.create ({
    baseURL: import.meta.env.MODE === "development" ? "http://localhost:5001/api" : "/api",
    withCredentials: true,
});
```

Рисунок 4.55 – Налаштування Axios для клієнт-серверної взаємодії залежно від середовища

Файл tailwind.config.js дає змогу згенерувати лише ті стилі, що фактично використовуються в реальному часі в ./src/**/*.{js,ts,jsx,tsx}, завдяки чому бандл буде оптимальним. Частина цього коду продемонстровано на (рис 4.56):

```
import daisyui from "daisyui";

/** @type {import('tailwindcss').Config} */
export default {
  content: ["./index.html", "./src/**/*.{js,ts,jsx,tsx}"],
  theme: {
    extend: {},
  },
  plugins: [daisyui],
  daisyui: {
    themes: [
      "light",
      "dark",
      "cupcake",
      "bumblebee",
      "emerald",
```

Рисунок 4.56 – Лістинг файлу tailwind.config.js

А у index.css (рис 4.57) через @tailwind base, components, utilities включаються базові стилі та утиліти Tailwind для усього зазначеного проєкту.

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Рисунок 4.57 – Лістинг файлу index.css

```
export default {
  plugins: {
    tailwindcss: {},
    autoprefixer: {},
  },
}
```

Рисунок 4.58 – Лістинг файлу postcss.config.js

На (рис 4.58) продемонсровано конфігураційний для роботи з PostCSS, підключає плагін TailwindCSS та додає префікси по типу -webkit-, -moz- до CSS-властивостей

Здатність інтерфейсу до змін можна досягнути завдяки вбудованим утилітам Tailwind, що використовуються в компонентах через класові модифікатори, наприклад sm:, md:, lg:. Наприклад, можна побачити свого роду універсальність

поведінки компонентів через класи типу w-full sm, які можуть змінювати ширину блоку в залежності від самого розміру екрана. Tailwind дає змогу уникати ручного написання медіа-запитів та зберігає якість коду, Те як записується Tailwind продемонстровано на (рис 4.59):

```
<p className="text-sm text-base-content/70">
```

а)

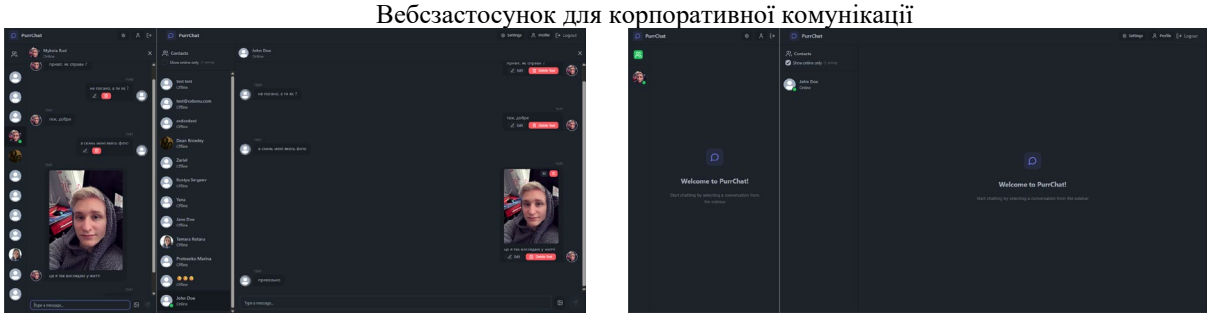
```
className="btn btn-circle btn-sm sm:btn-md ■btn-primary hover:bg-primary/90 transition-colors"
```

б)

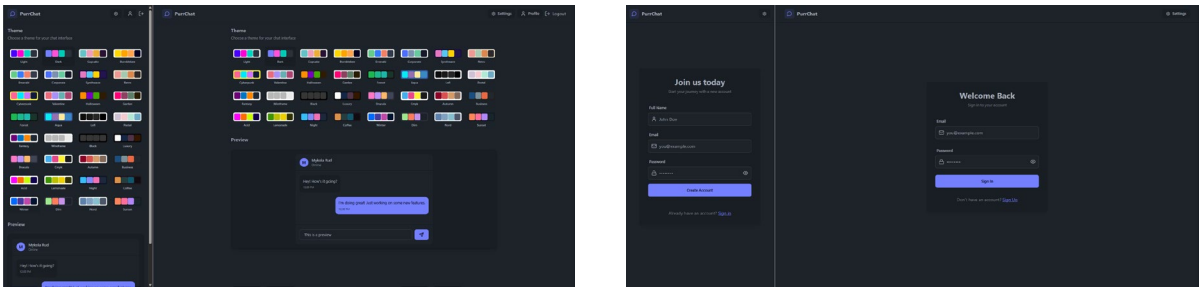
Рисунок 4.59 – модифітор sm (а) та md

модифітор sm відповідає за ширину екрану ≥ 768px, а модифікатор md відповідає за ширину екрану ≥ 1024px.

Все вищезазначене створює єдину, відповідну будову, де авторизація реально взаємодіє, всі налаштування всього вигляду та гнучкість користування повністю об’єднані в інфраструктуру через добре вбудовані модулі коду, приклад адаптації для додатку продемонстровано на (рис 4.60):



а) б)



в) г)

Рисунок 4.60 – Приклад адаптації при різних розмірах екрана у HomePage з активним ChatContainer (а), HomePage з активним NoChatSelected (б), SettingsPage (в) та в SignUpPage та LoginUpPage (г).

4.2 Серверна частина вебзастосунку. Архітектура та структура проєкту

При створені серверної частини вебзастосунку було використано Express. І на самому початку. Головного файлу index.js на (рис 4.61) ми зустрінемо:

```
import express from "express";
import dotenv from "dotenv";
import cookieParser from "cookie-parser";
import cors from "cors";
import path from "path";
import { fileURLToPath } from "url";
import { dirname } from "path";

import { connectDB } from "./lib/db.js";
import authRoutes from "./routes/auth.route.js";
import messageRoutes from "./routes/message.route.js";
import { app, server } from "./lib/socket.js";
```

Рисунок 4.61 – Імпорти для налаштування Express-сервера

Для початку на необхідно імпортувати усі необхідні модулі для створення серверної частини серверу. Dotenv для роботи з конфігураційним файлом .env, cookie-parser для роботи з cookie, Cors для запитів з іншим доменів. Утиліти по типу path, fileURLToPath та dirname потрібні для роботи з файлами з модулями.

Також ми на (рис 4.62), бачимо як імпортуються такі модулі, як функція connectDB, щоб працювати з базою даних MongoDB

```
dotenv.config();
```

Рисунок 4.62 – Ініціалізація dotenv

Завантаження змінних з .env (рис 4.63).

```
app.use(express.json({ limit: "10mb" }));
app.use(cookieParser());
app.use(
  cors({
    origin: "http://localhost:5173",
    credentials: true
  })
);
```

Рисунок 4.63 – Налаштування серверної частини додатку

Тут ми налаштовуємо JSON-запити з обмеженнями у 10MB, для того щоб коректно працювати з cloudinary. І фото які користувач може змінювати у своєму профілі та відправляти у чатах, не повинно привищувати цей ліміт.

Також тут на (рис 4.64) присутнє налаштування cookies і дозвіл CORS-запитів з frontend частини, що працює на localhost:5173

```
app.use("/api/auth", authRoutes);
app.use("/api/messages", messageRoutes);
```

Рисунок 4.64 – Налаштування маршрутів длясерверної частини додатку

Ці маршрути, які на (рис 4.65) потрібні, щоб відповідати за автентифікацію та обмін повідомленнями

```
server.listen(PORT, () => {
  console.log("Server is running on PORT: " + PORT);
});
```

Рисунок 4.65 – Запуск серверу

4.2.1 Обробка запитів (контролери)

У цьому пункті буде розглянутися функції, які будуть реалізовувати обробку запитів сервера. Основними серед таких є файли auth.controller.js – він потрібен для реєстрації та роботи з профілем клієнта. message.controller.js потрібен для того, щоб користувач міг відправляти повідомлення та зображення, змінювати прфіль і це все бачили інші користувачі.

Нижче (на рис. 4.66-4.70) навелено функції які відповідають за повну реєстрацію клієнта, а саме за вхід, вихід та оновлення профілю

```
export const signup = async (req, res) => {
  const { fullName, email, password } = req.body;
  try {
    if (!fullName || !email || !password) {
      return res.status(400).json({ message: "All fields are required" });
    }

    if (password.length < 6) {
      return res.status(400).json({ message: "Password must be at least 6 characters" });
    }

    const user = await User.findOne({ email });

    if (user) return res.status(400).json({ message: "Email already exists" });

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

    const newUser = new User({
      fullName,
      email,
      password: hashedPassword,
    });

    if (newUser) {
      generateToken(newUser._id, res);
      await newUser.save();

      res.status(201).json({
        _id: newUser._id,
        fullName: newUser.fullName,
        email: newUser.email,
        profilePic: newUser.profilePic,
      });
    } else {
      res.status(400).json({ message: "Invalid user data" });
    }
  } catch (error) {
    console.log("Error in signup controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
};
```

Рисунок 4.66 – Функція, яка відповідає за реєстрацію

На цьому рисунку ми бачимо функцію, яка відповідає за реєстрацію клієнта, вона перевіряє дані з форми, яку отримує з frontend частини, хешує пароль за допомогою бібліотеки bcryptjs, створює нового клієнта та генерує токен.

```
export const login = async (req, res) => {
  const { email, password } = req.body;
  try {
    const user = await User.findOne({ email });

    if (!user) {
      return res.status(400).json({ message: "Invalid credentials" });
    }

    const isPasswordCorrect = await bcrypt.compare(password, user.password);
    if (!isPasswordCorrect) {
      return res.status(400).json({ message: "Invalid credentials" });
    }

    generateToken(user._id, res);

    res.status(200).json({
      _id: user._id,
      fullName: user.fullName,
      email: user.email,
      profilePic: user.profilePic,
    });
  } catch (error) {
    console.log("Error in login controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
};
```

Рисунок 4.67 – Функція, яка відповідає за вхід

Ця функція перевіряє чи існує такий користувач, якщо існує генерує для нього токен.

```
export const logout = (req, res) => {
  try {
    res.cookie("jwt", "", { maxAge: 0 });
    res.status(200).json({ message: "Logged out successfully" });
  } catch (error) {
    console.log("Error in logout controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
};
```

Рисунок 4.68 – Функція, яка відповідає за вихід

Ця функція очищає токен з cookies.

```
export const updateProfile = async (req, res) => {
  try {
    const { fullName, email, password, profilePic, removeProfilePic } = req.body;
    const userId = req.user._id;

    const updateData = {};

    // Update name
    if (fullName) {
      if (fullName.trim().length < 3) {
        return res.status(400).json({ message: "Name must be at least 3 characters" });
      }
      updateData.fullName = fullName;
    }

    // Update email
    if (email) {
      if (!/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email)) {
        return res.status(400).json({ message: "Invalid email format" });
      }

      const existingUser = await User.findOne({ email });
      if (existingUser && existingUser._id.toString() !== userId.toString()) {
        return res.status(400).json({ message: "Email already in use" });
      }
      updateData.email = email;
    }

    // Update password
    if (password) {
      if (password.length < 6) {
        return res.status(400).json({ message: "Password must be at least 6 characters" });
      }
```

Рисунок 4.69 – Частина функції, яка відповідає за оновдення профілю

Ця функція дозволяє змінити ім’я, пошту, пароль та аватарку (за допомогою сервісу Cloudinary)

```
export const checkAuth = (req, res) => {
  try {
    res.status(200).json(req.user);
  } catch (error) {
    console.log("Error in checkAuth controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
};
```

Рисунок 4.70 – Функція, яка відповідає за оновдення профілю

Ця функція слугує для перевірки чи авторизований користувач.

Далі перейдемо до файлу message.controller.js цей контролер відповідає за логіку обміну повідомленнями між користувачами.

Нижче (на рис. 4.71-4.74) навелено функції які відповідають за обмін повідомленнями між клієнтами, а саме за відправку повідомлень, отримання користувачів для SideBar, отримання повідомлень у чатах, для змінення, видачення текстових повідомлень та зображень.

```
export const sendMessage = async (req, res) => {
    try {
        const { text, image } = req.body;
        const { id: receiverId } = req.params;
        const senderId = req.user._id;

        let imageUrl = null;

        if (image) {
            // Upload base64 image to cloudinary
            const uploadResponse = await clodinary.uploader.upload(image);
            imageUrl = uploadResponse.secure_url;
        }

        const newMessage = new Message({
            senderId,
            receiverId,
            text,
            image: imageUrl,
        });

        await newMessage.save();

        const receiverSocketId = getReceiverSocketId(receiverId);
        if(receiverSocketId) {
          io.to(receiverSocketId).emit("newMessage", newMessage);
        }

        res.status(201).json(newMessage);
    } catch (error) {
        console.log("Error in sendMessage controller: ", error.message);
        res.status(500).json({ error: "Internal server error" });
    }
};
```

Рисунок 4.71 – Функція, яка відповідає за відправку повідомлень

Ця функція створює нове повідомлення, дає можливість відправки картинок через платформу Cloudinary. Зберігає це все у базу даних та передає через WebSoket на frontend частину, де це побачить користувач.

```
export const getUsersForSidebar = async(req, res) => {
    try {
        const loggedInUserId = req.user._id;
        const filteredUsers = await User.find({_id: {$ne:loggedInUserId}}).select("-password");

        res.status(200).json(filteredUsers);
    } catch(error) {
        console.error("Error in getUsersForSidebar: ", error.message);
        res.status(500).json({ error: "Internal server error" });
    }
};
```

Рисунок 4.72 – Функція, яка відповідає за отримання користувачів

Ця функція слугує за отримання списку користувачів для SideBar. Повертаючи список усіх користувачів окрім поточного.

```
export const getMessages = async(req, res) => {
    try {
        const { id:userToChatId } = req.params
        const myId = req.user._id;

        const messages = await Message.find({
            $or: [
                {senderId:myId, receiverId:userToChatId},
                {senderId:userToChatId, receiverId:myId}
            ]
        })

        res.status(200).json(messages)
    } catch(error) {
        console.log("Error in getMessages controller: ", error.message);
        res.status(500).json({ error: "Internal server error" });
    }
};
```

Рисунок 4.73 – Функція, яка відповідає за отримання повідомлень

Ця функція слугує за отримання повідомлень за певним id між двома користувачами.

```
export const editMessage = async (req, res) => {
  try {
    const { messageId } = req.params;
    const { text, image } = req.body;

    let imageUrl = image || null;

    if (image && image.startsWith("data:image/")) {
      const uploadResponse = await clodinary.uploader.upload(image);
      imageUrl = uploadResponse.secure_url;
    }

    const updatedMessage = await Message.findByIdAndUpdate(
      messageId,
      { text, image: imageUrl },
      { new: true }
    );

    if (!updatedMessage) {
      return res.status(404).json({ error: "Message not found" });
    }

    res.status(200).json(updatedMessage);
    io.emit("messageUpdated", updatedMessage);
  } catch (error) {
    console.log("Error in editMessage controller: ", error.message);
    res.status(500).json({ error: "Internal server error" });
  }
};
```

а)

```
export const deleteMessage = async (req, res) => {
  try {
    const { messageId } = req.params;

    const deletedMessage = await Message.findByIdAndDelete(messageId);

    if (!deletedMessage) {
      return res.status(404).json({ error: "Message not found" });
    }

    res.status(200).json({ message: "Message deleted successfully" });
    io.emit("messageDeleted", messageId);
  } catch (error) {
    console.log("Error in deleteMessage controller: ", error.message);
    res.status(500).json({ error: "Internal server error" });
  }
};
```

б)

```
export const replaceImage = async (req, res) => {
  try {
    const { messageId } = req.params;
    const { image } = req.body;

    const uploadResponse = await clodinary.uploader.upload(image);
    const imageUrl = uploadResponse.secure_url;

    const updatedMessage = await Message.findByIdAndUpdate(
      messageId,
      { image: imageUrl },
      { new: true }
    );

    if (!updatedMessage) {
      return res.status(404).json({ error: "Message not found" });
    }

    res.status(200).json(updatedMessage);
    io.emit("messageUpdated", updatedMessage);
  } catch (error) {
    console.log("Error in replaceImage controller: ", error.message);
    res.status(500).json({ error: "Internal server error" });
  }
};
```

в)

```
export const removeImage = async (req, res) => {
  try {
    const { messageId } = req.params;

    const updatedMessage = await Message.findByIdAndUpdate(
      messageId,
      { image: "" },
      { new: true }
    );

    if (!updatedMessage) {
      return res.status(404).json({ error: "Message not found" });
    }

    res.status(200).json(updatedMessage);
    io.emit("messageUpdated", updatedMessage);
  } catch (error) {
    console.log("Error in removeImage controller: ", error.message);
    res.status(500).json({ error: "Internal server error" });
  }
};
```

г)

Рисунок 4.74 – Функції для редагування повідомлень (а), видалення повідомлень (б), редагування картинок (б) та видалення картинок (г)

(На рис. 4.74) функції які відповідають за редагування та видалення текстових повідомлень та зображень.

4.2.2 Моделі та маршрути

Далі перейдемо до опису схеми користувача для бази даних, (на рис. 4.75) наданий лістинг коду файлу user.model.js.

```
import mongoose from "mongoose";

const userSchema = new mongoose.Schema(
    {
        email: {
            type: String,
            required: true,
            unique: true,
        },
        fullName: {
            type: String,
            required: true,
        },
        password: {
            type: String,
            required: true,
            minlength: 6,
        },
        profilePic: {
            type: String,
            default: "",
        },
    },
    { timestamps: true }
);

const User = mongoose.model("User", userSchema);
export default User;
```

Рисунок 4.75 – Схема користувача для бази даних

На цьому рисунку наданий код схеми користувача для бази даних MongoBD. Тут використовується бібліотека mongogoose. На схемі можна побачити, як визначені поля, а саме email, fullName, password та profilePic.

Також timestamps: true каже нам, що буде показуватись дата та час створення цієї схеми. І дату ми використуємо на клієнтській частині у ProfilePage, щоб показувати користувачу дату його реєстрації.

Тепер перейдемо до опису схеми для збереження повідомлень у базі даних. (На рис. 4.76) наданий лістинг коду файлу message.model.js.

```
import mongoose from "mongoose";

const messageSchema = new mongoose.Schema(
  {
    senderId: {
      type: mongoose.Schema.Types.ObjectId,
      ref: "User",
      required: true,
    },
    receiverId: {
      type: mongoose.Schema.Types.ObjectId,
      ref: "User",
      required: true,
    },
    text: {
      type: String,
    },
    image: {
      type: String,
    },
  },
  { timestamps: true }
);

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

export default Message;
```

Рисунок 4.76 – Схема для збереження повідомлень у базі даних

На цьому рисунку наданий код схеми для збереження повідомлень у базі даних. На схемі можна побачити, що кожне повідомлення має як senderId – id для відправника та receiverId для отримувача текстових повідомлень або забражень. Також timestamps: true каже нам, що буде показуватись дата та час створення.

Далі (на рис. 4.77) наданий лістинг коду файлу auth.route.js.

```
import express from "express";
import { checkAuth, login, logout, signup, updateProfile } from "../controllers/auth.controller.js";
import { protectRoute } from "../middleware/auth.middleware.js";

const router = express.Router();

router.post("/signup", signup);
router.post("/login", login);
router.post("/logout", logout);

router.put("/update-profile", protectRoute, updateProfile);

router.get("/check", protectRoute, checkAuth);

export default router;
```

Рисунок 4.77 – Routes пов’язані з автентифікацією користувача

На цьому рисунку ми побачиму усі routes які пов’язані з автентифікацією користувача. Тут є усі маршрути для входу, виходу, реєстрації, оновлення профілю та перевірки автенфікації користувача. Такоє маршрути для перевірки автенфікації користувача та оновлення профілю мають захист protectedRoute, який здійснюється у middleware.

Далі (на рис. 4.78) наданий лістинг коду файлу message.route.js.

```
import express from "express";
import { protectRoute } from "../middleware/auth.middleware.js";
import { getMessages, getUsersForSidebar, sendMessage, editMessage, deleteMessage, replaceImage, removeImage } from "../controllers/message.controller.js";
const router = express.Router();

router.get("/users", protectRoute, getUsersForSidebar);
router.get("/:id", protectRoute, getMessages);
router.post("/send/:id", protectRoute, sendMessage);
router.put("/:messageId", protectRoute, editMessage);
router.delete("/:messageId", protectRoute, deleteMessage);
router.put("/image/:messageId", protectRoute, replaceImage);
router.delete("/image/:messageId", protectRoute, removeImage);

export default router;
```

Рисунок 4.78 – Routes пов’язані з повідомленнями

На цьому рисунку ми побачимо усі routes які пов’язані з повідомленнями. Тут у нас присутні routes отримання користувачів для бічної панелі, а також routes для надсилання, редагування, видалення текстових повідомлень та зображень. Усі маршрути мають захист protectedRoute, який здійснюється у middleware.

4.2.3 Підключення до хмарного сервісу Cloudinary

На протязі всієї дипломної роботи так чи інакше було згадано за платформу «cloudinary». У цьому підрозділі буде опис налаштування сервісу cloudinary для роботи з фото відправкою зображень та можливістю зміни аватару профілю.

Після того, як ми зареєстрували акаунт на платформі, ми можемо побачити ім’я нашої хмари (рис 4.79):

Рисунок 4.79 – Ім’я хмари на платформі Cloudinary

Потім ми можемо натиснути на кнопку «Go to API Keys». Щоб отримати API Key та API Secret, який ми використаємо у нашому застосунку для під’єднання платформи (рис 4.80):

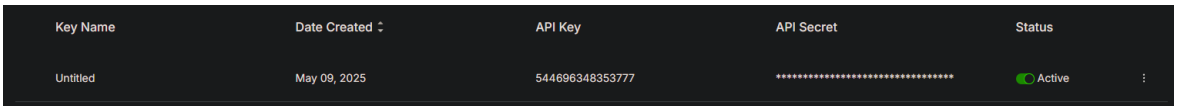


Рисунок 4.80 – API Key та API Secret

Це все, що нам знадобиться для підключення платформи. Ми це все записуємо у конфігураційний файл .env. Потім у файлі cloudinary.js під’єднуємо та налаштовуємо сервіс, щоб він працював у нашому додатку:

```
import {v2 as clodinary} from "cloudinary";

import {config} from 'dotenv';

config();

clodinary.config({
    cloud_name: process.env.CLOUDINARY_CLOUD_NAME,
    api_key: process.env.CLOUDINARY_API_KEY,
    api_secret: process.env.CLOUDINARY_API_SECRET,
});

export default clodinary;
```

Рисунок 4.81 – Лістинг файлу cloudinary.js

(На рис. 4.81) лістинг коду файлу cloudinary.js, який відповідає за налаштування платформи для отримання зображень та захищених URL. У файлі .env прописані ключові параметри, які ми дістали зареєструвавшись на cloudinary. А саме: назва хмари, API Key та API Secret.

Після того, як ми налаштували файл cloudinary.js він експортується для того, щоб його використовувати в інших частинах програми для оновлення аватару профілю та відправки картинок у чатах.

4.2.4 Робота з WebSocket через Socket.IO

Робота з Socket.IO реалізується за допомогоюWebsoket та відповідної бібліотеки. Тут ми бачимо, як створюється HTTP-сервер на Express та ініціалізацією Socket.IO-серверу з CORS та можливістю взаємодії з клієнтським застосунком (рис 4.82):

```
import { Server } from "socket.io";
import http from "http";
import express from "express";

const app = express();
const server = http.createServer(app);

const io = new Server (server, {
    cors: {
        origin: ["http://localhost:5173"]
    },
});
```

Рисунок 4.82 – Створення HTTP-сервера та підключення Socket.IO з налаштуванням CORS.

uesrSocketMap та socketUserMap – це мапи, які призвані, щоб зберігати сумісність між користувачами та їхніми сокетами. Це потрібно для того, щоб знати, хто зараз онлайн і на який сокет кому відправляти повідомлення (рис 4.83):

```
const socketUserMap = {};
const userSocketMap = {};
```

Рисунок 4.83 – uesrSocketMap та socketUserMap

Після того, як підключився новий сокет за ним ідентифікується користувач за відповідним userId, який збереться у query і всі під’єднані користувачі отримують оновлений список користувачів у онлайні (рис 4.84):

```
io.on("connection", (socket) => {
    const userId = socket.handshake.query.userId;
    if (userId) {
        userSocketMap[userId] = socket.id;
        socketUserMap[socket.id] = userId;
    }

    io.emit("getOnlineUsers", Object.keys(userSocketMap));
```

Рисунок 4.84 – Подія connection

Подія disconnect потрібна, щоб видалити сокет ідентифікованого користувача зі списку онлайнових і обновити онлайнових юзерів (рис 4.85):

```
socket.on("disconnect", () => {
    const disconnectedUserId = socketUserMap[socket.id];
    delete userSocketMap[disconnectedUserId];
    delete socketUserMap[socket.id];
    io.emit("getOnlineUsers", Object.keys(userSocketMap));
});
```

Рисунок 4.85 – Подія disconnect

І ще в нас є функція getReceiverSocketId(userId) потрібен, щоб отримати ідентифікатор сокета юзера, щоб надсилати йому повідомлення (рис 4.86):

```
export function getReceiverSocketId(userId) {
    return userSocketMap[userId];
}
```

Рисунок 4.86 – Функція getReceiverSocketId(userId)

Потім файл експортує io, app та server, для їх подальшого використання в інших частинах програми. Наприклад для підключення юзерів і запуску сервера у реальному часі(рис 4.87):

```
export { io, app, server };
```

Рисунок 4.87 – Експорт io, app та server

4.2.5 Middleware та безпека

authMiddleware.js потрібен для того, щоб перевіряти юзера через JWT та захищати routes від несанкціонованого доступу. Метою protectRoute аналізувати токен, який він отримав з cookie запиту, декодувати його, перевірити валідність та перевірити існування цього юзера у базі даних (рис 4.88):

```
import jwt from "jsonwebtoken";
import User from "../models/user.model.js";

export const protectRoute = async (req, res, next) => {
    try {
        const token = req.cookies.jwt

        if(!token) {
            return res.status(401).json({message: "Unnathorized - No Token Provided"});
        }

        const decoded = jwt.verify(token, process.env.JWT_SECRET);

        if(!decoded) {
            return res.status(401).json({message: "Unauthorized - Invalid Token"});
        }

        const user = await User.findById(decoded.userId).select("-password");

        if(!user) {
            return res.status(404).json({message: "User not found"});
        }

        req.user = user
        next()

    } catch(error) {
        console.log("Error in protectRoute middleware:", error.message);
        res.status(500).json({message: "Internal server error"});
    }
}
```

Рисунок 4.88 – Лістинг коду authMiddleware.js

Також, якщо JWT-токен відсутній або недійсний по якійсь причині, запит закінчується з HTTP-статусом 401 (Unauthorized) або 404 (User not found). Але якщо запит був вдалий, то юзер попадає в об’єкт req і перенаправляється до наступного middleware або через конролер next().

units.js це файл який створює JWT-токен орієнтуючись на userId, підписує його за допомогою JWT_SECRET та зберігає його в cookie.

```
import jwt from "jsonwebtoken";

export const generateToken = (userId, res) => {
  const token = jwt.sign({ userId }, process.env.JWT_SECRET, {
    expiresIn: "7d",
  });

  res.cookie("jwt", token, {
    maxAge: 7 * 24 * 60 * 60 * 1000, // MS
    httpOnly: true, // prevent XSS attacks cross-site scripting attacks
    sameSite: "strict", // CSRF attacks cross-site request forgery attacks
    secure: process.env.NODE_ENV !== "development",
  });

  return token;
};
```

Рисунок 4.89 – Лістинг коду units.js

Функція яка (на рис 4.89) дає гарантію не те, що токен зберігається безпечно на клієнтській частині.

Тому, можна зробити деякий висновок, що middleware та логіка по якій генерується JWT-токен, буде забезпечувати надійний захист маршрутів і автентифікації користувача.

4.3 Налаштування та підключення до MongoDB

Для початку, щоб під’єднатися до бази даних MongoDB, необхідно зайти на MongoDB Atlas – це база даних яка знаходиться у хмарі. Для того, щоб почати з нею працювати, необхідно створити проєкт (рис 4.90):.

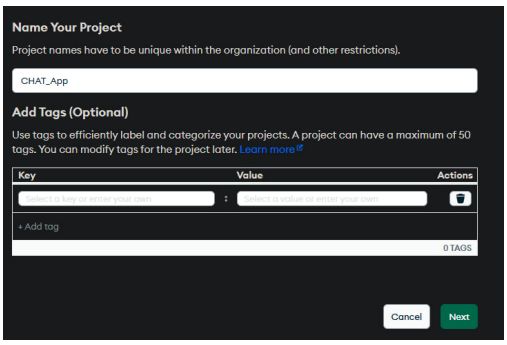
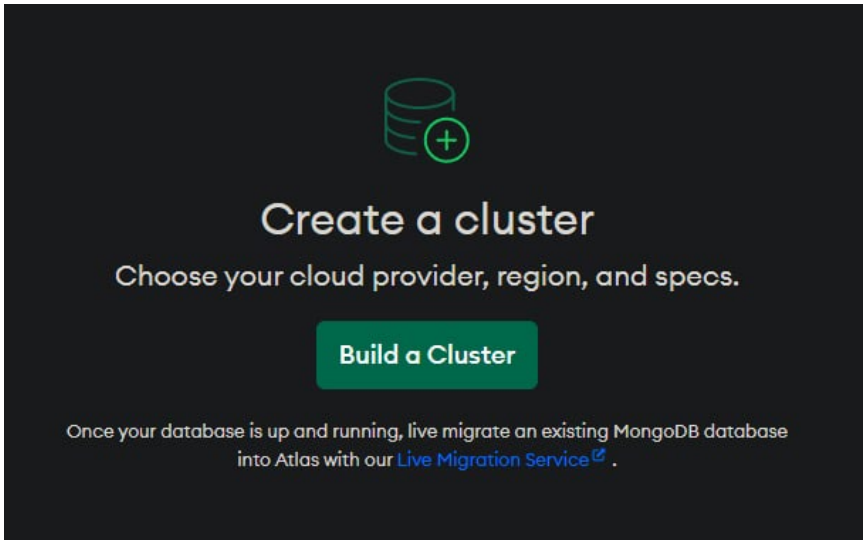
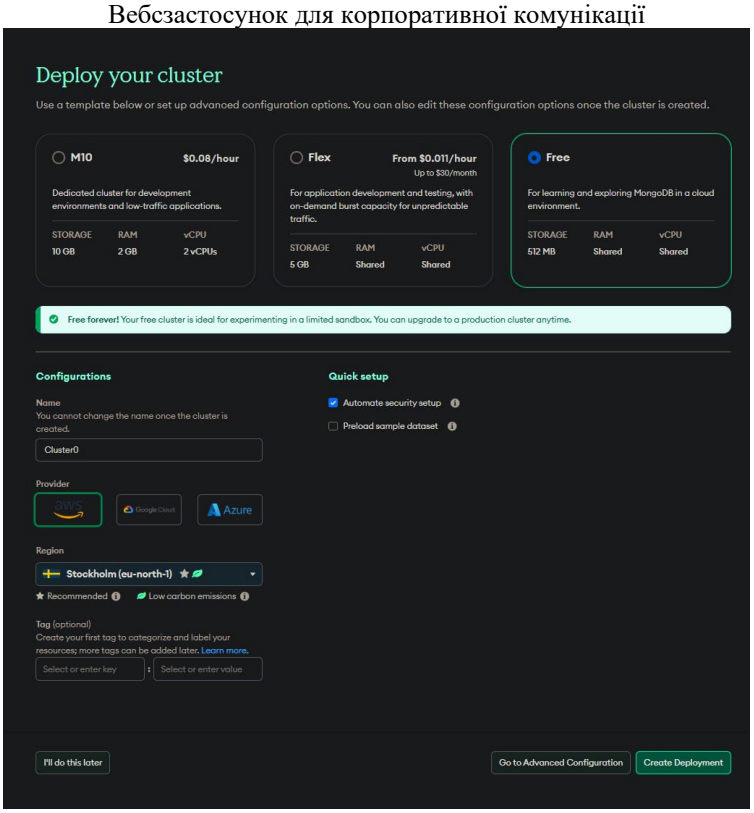


Рисунок 4.90 – Даємо назву для проєкту на MongoDB Atlas

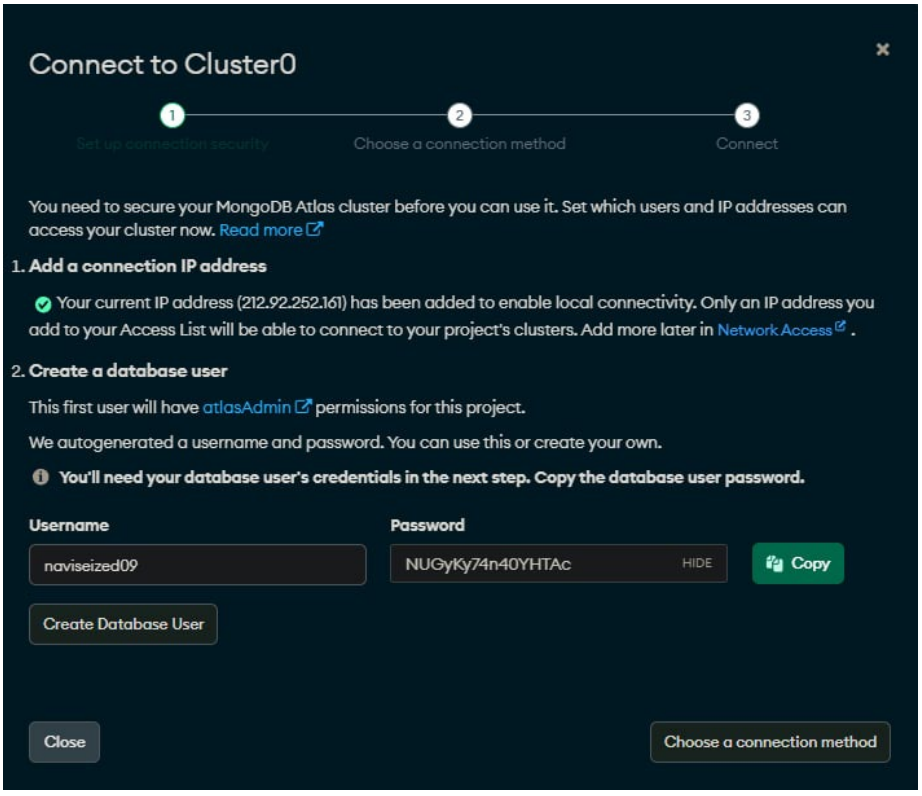
Після створення проєкту, необхідно створити кластер до якого ми будемо під’єднувати наш проєкт, обрати план, обрати метод пыдключення та отримати лінк для підключення до бази даних. Це все продемонстровано (на рис 4.91).



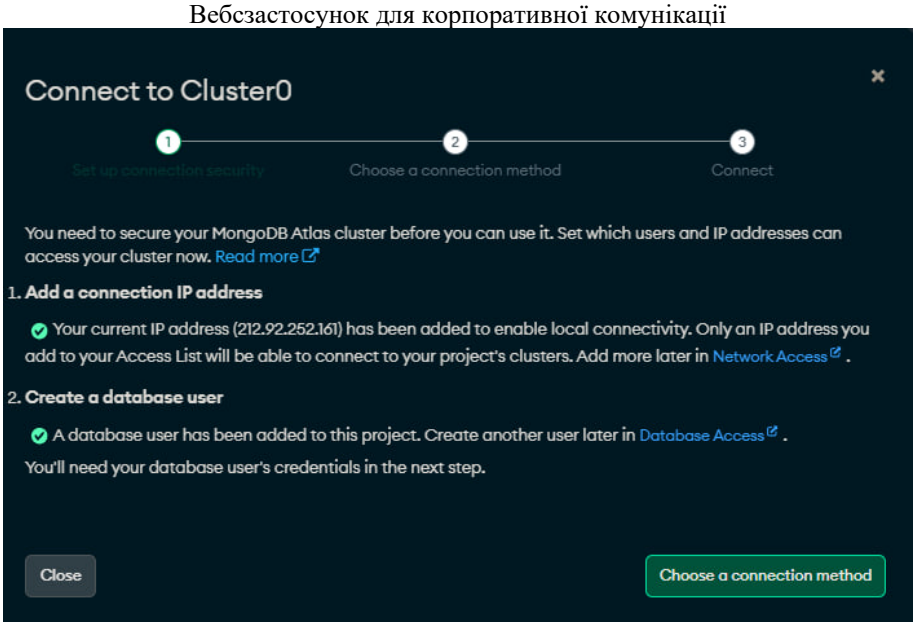
а)


Deploy your cluster
Use a template below or set up advanced configuration options. You can also edit these configuration options once the cluster is created.
M10
$0.08/hour
Dedicated cluster for development environments and low-traffic applications.
STORAGE 10 GB
RAM 2 GB
vCPU 2 vCPUs
Flex
From $0.011/hour
Up to $30/month
For application development and testing, with on-demand burst capacity for unpredictable traffic.
STORAGE 5 GB
RAM Shared
vCPU Shared
Free
For learning and exploring MongoDB in a cloud environment.
STORAGE 512 MB
RAM Shared
vCPU Shared
Free forever! Your free cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.
Configurations
Name
You cannot change the name once the cluster is created.
Cluster0
Provider
Azure
Region
Stockholm (eu-north-1)
Recommended
Low carbon emissions
Tag (optional)
Create your first tag to categorize and label your resources; more tags can be added later. Learn more.
Select or enter key
Select or enter value
Quick setup
Automate security setup
Preload sample dataset
I'll do this later
Go to Advanced Configuration
Create Deployment

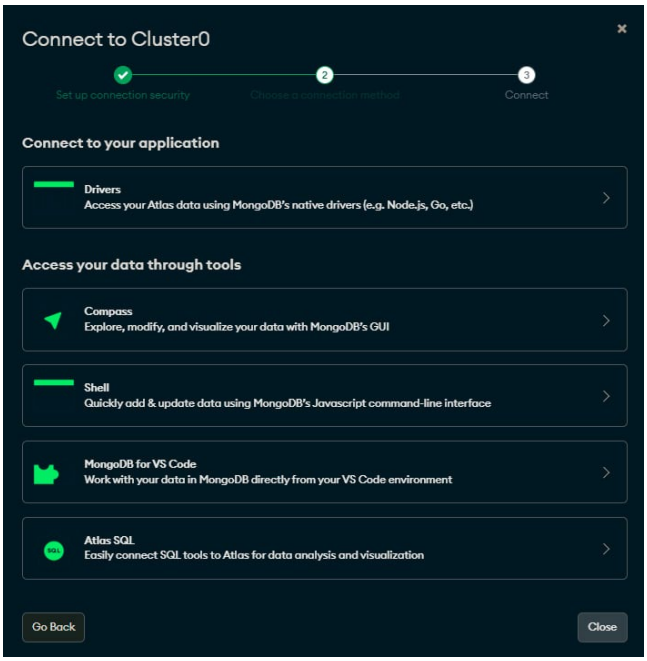
б)


Connect to Cluster0
1
Set up connection security
2
Choose a connection method
3
Connect
You need to secure your MongoDB Atlas cluster before you can use it. Set which users and IP addresses can access your cluster now. Read more
1. Add a connection IP address
Your current IP address (212.92.252.161) has been added to enable local connectivity. Only an IP address you add to your Access List will be able to connect to your project's clusters. Add more later in Network Access.
2. Create a database user
This first user will have atlasAdmin permissions for this project.
We autogenerated a username and password. You can use this or create your own.
You'll need your database user's credentials in the next step. Copy the database user password.
Username
naviseized09
Password
NUGyKy74n40YHTAc
HIDE
Copy
Create Database User
Close
Choose a connection method

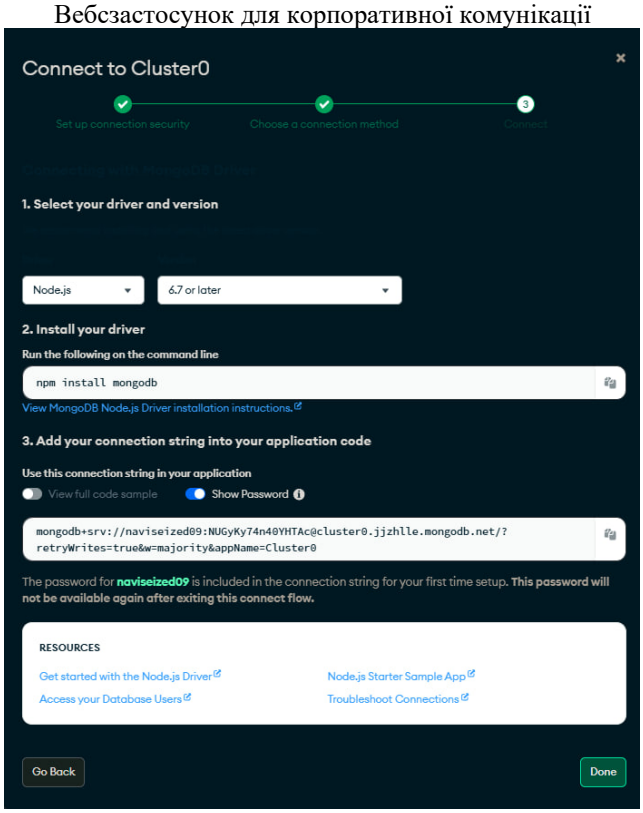
в)


Connect to Cluster0
1
Set up connection security
2
Choose a connection method
3
Connect
You need to secure your MongoDB Atlas cluster before you can use it. Set which users and IP addresses can access your cluster now. Read more
1. Add a connection IP address
Your current IP address (212.92.252.161) has been added to enable local connectivity. Only an IP address you add to your Access List will be able to connect to your project's clusters. Add more later in Network Access .
2. Create a database user
A database user has been added to this project. Create another user later in Database Access .
You'll need your database user's credentials in the next step.
Close
Choose a connection method

г)


Connect to Cluster0
Set up connection security
Choose a connection method
Connect
Connect to your application
Drivers
Access your Atlas data using MongoDB's native drivers (e.g. Node.js, Go, etc.)
Access your data through tools
Compass
Explore, modify, and visualize your data with MongoDB's GUI
Shell
Quickly add & update data using MongoDB's Javascript command-line interface
MongoDB for VS Code
Work with your data in MongoDB directly from your VS Code environment
Atlas SQL
Easily connect SQL tools to Atlas for data analysis and visualization
Go Back
Close

ґ)



д)

Рисунок 4.91 – Налаштування бази даних MongoDB, створення кластеру (а), обираємо план (б), створюємо базу даних (в), натискаємо кнопку Choose a connection method (г), обираємо метод підключення «Drivers» (ґ), і копіюємо MONGODB_URI (д)

Після налаштування необхідно скопіювати MONGODB_URI та вставити його у конфігураційний файл .env. Тепер необхідно створити функцію, яка буде під’єднувати нас до бази даних. Для цього у дереві проєкту створено файл db.js. І в ньому описано спосіб підключення.

```
import mongoose from "mongoose";

export const connectDB = async () => {
  try {
    mongoose.set('strictQuery', true);
    const conn = await mongoose.connect(process.env.MONGODB_URI);
    console.log(`MongoDB connected: ${conn.connection.host}`);
  } catch (error) {
    console.log("MongoDB connection error:", error);
  }
};
```

Рисунок 4.92 – Асинхронна функція connectDB

(На рис. 4.92) наданий лістинг файлу db.js, який описує підключення до бази даних. Ми можемо побачити mongoose.set('strictQuery', true); – він вмикає суровий режим обробки запитів та асинхроно під’єднується до бази даних за URI, який зберігається в .env. Якщо підключення успішне виводиться відповідний console.log(MongoDB connected: ${conn.connection.host}); якщо є якась помилка, то ловиться catch та виводиться console.log("MongoDB connection error:", error);

Далі треба підняти базу даних, щоб вона запускалась разом з нашим сервером. Для цього трішки змінемо index.js як (на рис 4.93)

```
connectDB().then(() => {
  server.listen(PORT, () => {
    console.log("Server is running on PORT: " + PORT);
  });
});
```

Рисунок 4.93 – Запуск бази даних

Після того, як ми підключились до бази данних за допомогою connectDB, ми запускаємо сервер.

```
PS E:\chat-app\backend> npm run dev

> backend@1.0.0 dev
> nodemon src/index.js

[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node src/index.js`
MongoDB connected: ac-aw35aiz-shard-00-00.d8ofhbp.mongodb.net
Server is running on PORT: 5001
```

Рисунок 4.94 – npm run dev

У логах на (рис 4.94) бачимо, що сервер та база даних успішно під'єднані

4.4 Розгортання вебзастосунку на Render Dashboard

Для того, щоб розгорнути наш вебзастосунок на платформі Render Dashboard. Необхідно, для початку необхідно зібрати frontend та передати її на backend частину, щоб створити server-side static hosting. Створимо файл package.json у на рівні проєкту, де знаходяться папки frontend та backend за допомогою команди, npm init –y. як (на рис 4.95).

```
PS E:\chat-app> npm init -y
```

Рисунок 4.95 – npm init –y

Потім у створеному файлі, необхідно ввести ті команди, які буде прописувати Render Dashboad. Тому нам необхідно написати у розділі scripts пару команд, які продемонстровані (на рис 4.96).

```
{
  "name": "chat-app",
  "version": "1.0.0",
  "main": "index.js",
  ▷ Debug
  "scripts": {
    "build": "npm install --prefix backend && npm install --prefix frontend && npm run build --prefix frontend",
    "start": "npm run start --prefix backend && npm run dev --prefix frontend"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/poravoz/chatApp.git"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/poravoz/chatApp/issues"
  },
  "homepage": "https://github.com/poravoz/chatApp#readme",
  "description": "",
  "devDependencies": {
    "concurrently": "^9.1.2"
  }
}
```

Рисунок 4.96 – Лістинг файлу pakage.json

Тепер, нам необхідно, щоб після того, як ми скомпілюємо весь проєкт він передався на backend частину. Для цього напишемо у index.js два наступні рядки (рис 4.97):

```
const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);

console.log("__filename:", __filename);
console.log("__dirname:", __dirname);
```

Рисунок 4.97 – отримання шляху шляху до поточного файлу

Ці два рядки, необхідні, щоб ми мали змогу отримати абсолютний шлях до поточного файлу у форматі ES-модулів, тому що __dirname прямого доступу (рис 4.98-4.99):

```
const distPath = path.join(__dirname, "..", "..", "frontend", "dist");
console.log("Dist path:", distPath);
```

Рисунок 4.98 – Налаштування сервінг фронтенду створеного за допомогою Vite з папки dist

```
app.use("/", (req, res) => {
  console.log("Sending index.html from:", distPath);
  res.sendFile(path.join(distPath, "index.html"));
});
```

Рисунок 4.99 – Перенаправлення запиту на index.html які не змогли обробитися

Тепер необхідно збілдити проєкт, та перевірити чи передасться директорія dist на серверну частину (рис 4.100):

```
PS E:\chat-app> npm run build

> chat-app@1.0.0 build
> npm install --prefix backend && npm install --prefix frontend && npm run build --prefix frontend


up to date, audited 245 packages in 1s

26 packages are looking for funding
  run `npm fund` for details

2 vulnerabilities (1 moderate, 1 critical)

To address all issues, run:
  npm audit fix

Run `npm audit` for details.

up to date, audited 371 packages in 2s

129 packages are looking for funding
  run `npm fund` for details

3 moderate severity vulnerabilities

To address all issues, run:
  npm audit fix

Run `npm audit` for details.

> frontend@0.0.0 build
> vite build

vite v5.4.11 building for production...

   daisyUI 4.10.2
├─ ✓ 32 themes added          https://daisyui.com/docs/themes
└─ ★ Star daisyUI on GitHub    https://github.com/saadeghi/daisyui

✓ 1681 modules transformed.
dist/index.html                  0.47 kB │ gzip:  0.31 kB
dist/assets/index-B76u2zGW.css  79.55 kB │ gzip: 15.85 kB
dist/assets/index-BB-6HPVO.js  299.30 kB │ gzip: 95.12 kB
✓ built in 4.13s
```

Рисунок 4.100 – команда npm run build

Команда npm run build в корені проєкту успішно спрацювала. І ми можемо побачити, уся frontend частина збілдилась у два файли, як на (рис 4.101):

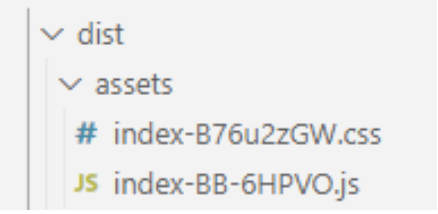


Рисунок 4.101 – папка dist успішно створилася

```
PS E:\chat-app\backend> npm run dev

> backend@1.0.0 dev
> nodemon src/index.js

[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node src/index.js`
__filename: E:\chat-app\backend\src\index.js
__dirname: E:\chat-app\backend\src
Dist path: E:\chat-app\frontend\dist
MongoDB connected: ac-aw35aiz-shard-00-00.d8ofhbp.mongodb.net
Server is running on PORT: 5001
[nodemon] restarting due to changes...
[nodemon] starting `node src/index.js`
__filename: E:\chat-app\backend\src\index.js
__dirname: E:\chat-app\backend\src
Dist path: E:\chat-app\frontend\dist
MongoDB connected: ac-aw35aiz-shard-00-00.d8ofhbp.mongodb.net
Server is running on PORT: 5001
```

Рисунок 4.102 – Запуск сервера

Ми успішно запустили сервер на (рис 4.102) та він знайшов папку dist і передав її на серверну частину. Оскільки ми не бачимо помилок. Після цього, нам треба спробувати запустити вже не localhost:5173. На якому працює фронтенд, а запустити localhost:5001. І якщо все спрацювало, то наш інтерфейс повинен запуститися на цьому порті.

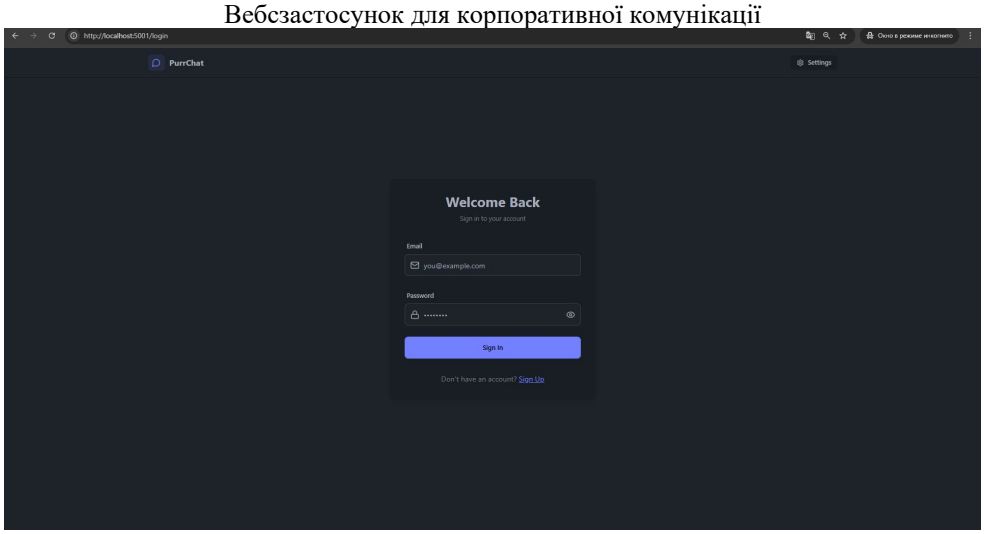
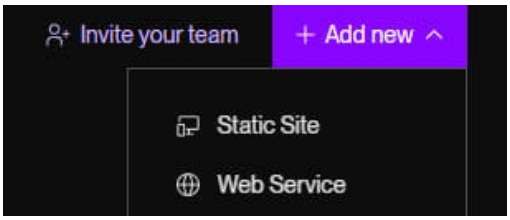
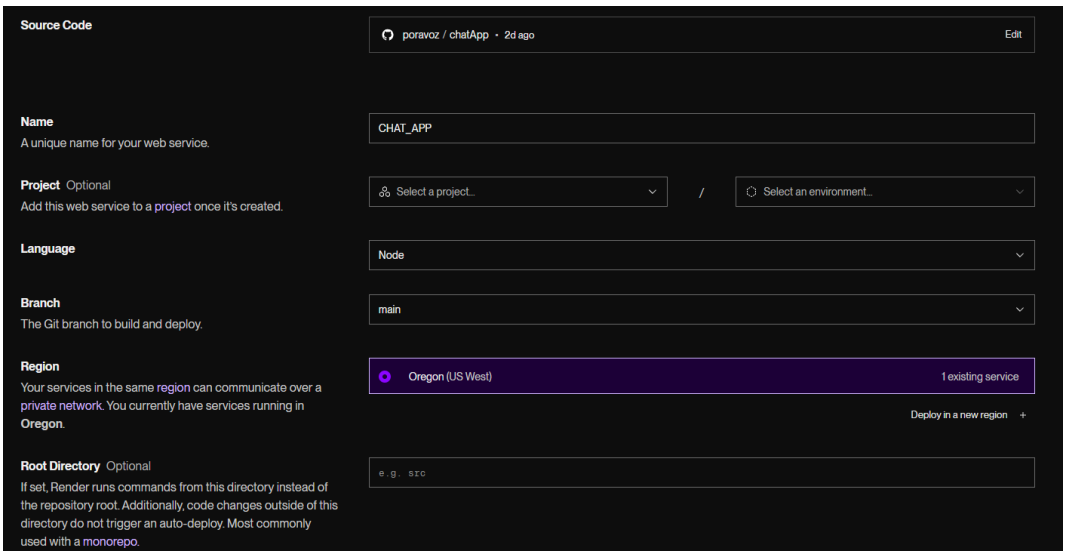


Рисунок 4.103 – Успішний запуск проєкту на localhost:5001

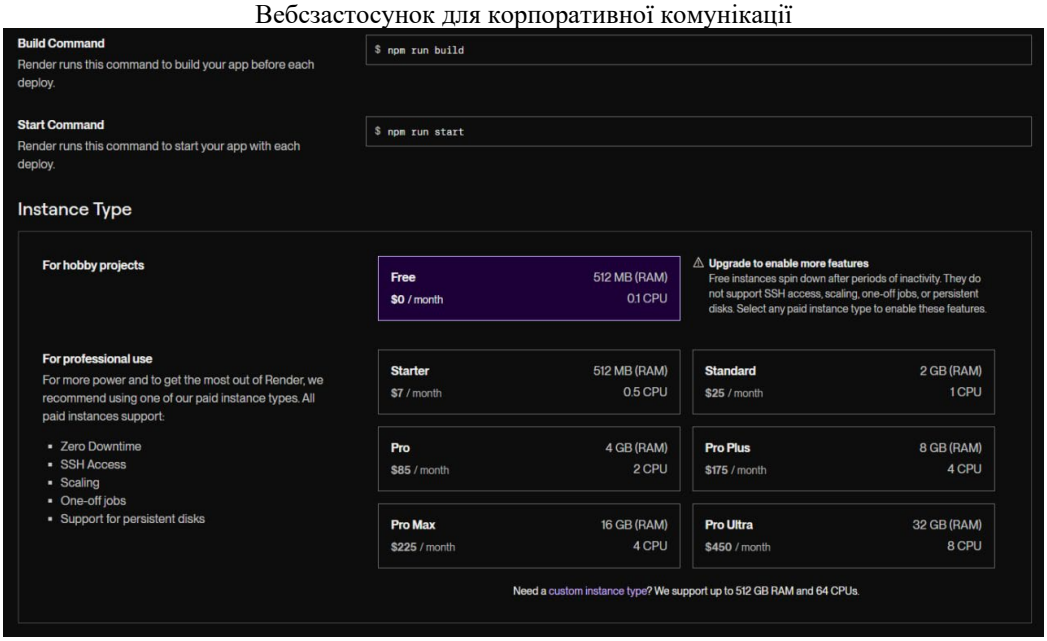
Тепер, коли на (рис 4.103) ми підготували проєкт до розгортання його на Render Dashboard далі нам слід зареєструватись на даній платформі, після чого створити новий web-service, та розгорнути там наш проєкт, як (на рис 4.104).



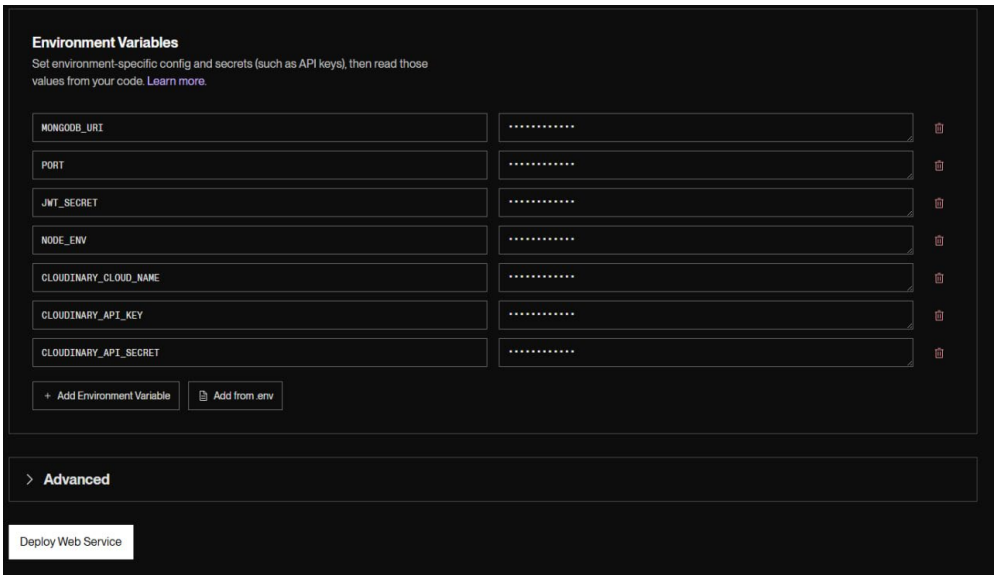
а)



б)



в)



г)

Рисунок 4.104 – Створення web service (а), обираємо вебзастосунок з github (б), пишемо команди для build та запуску (в). Та додаємо конфігурацію .env (г)

Після очікування в п’ять хвилин, поки проєкт розгорнеться на Render Dashboard бачимо, що все пройшло успішно і проєкт розгорнувся (рис 4.105):

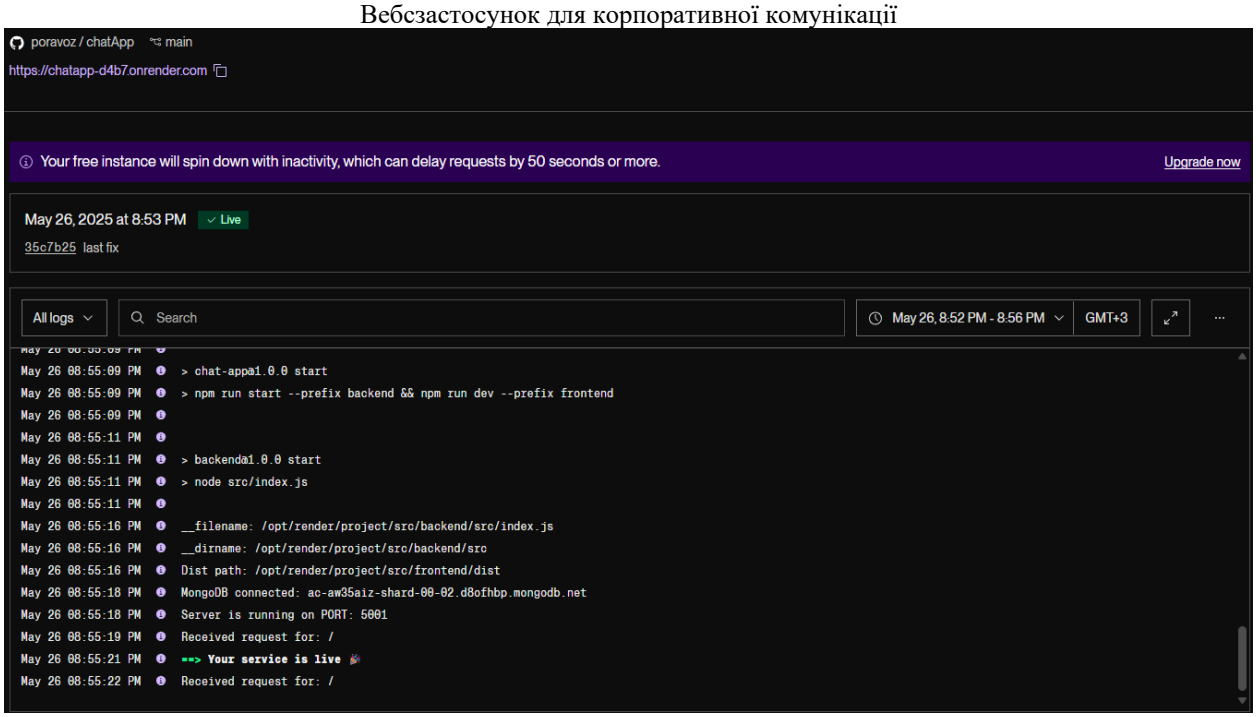


Рисунок 4.105 – Логи сереверу

І наостанок переходимо по лінку, який нам дала платформа і побачимо, що все працює. І вебсервіс успішно розгорнувся і тепер чат існує у загальному доступі, демонстрація роботи на (рис 4.106):

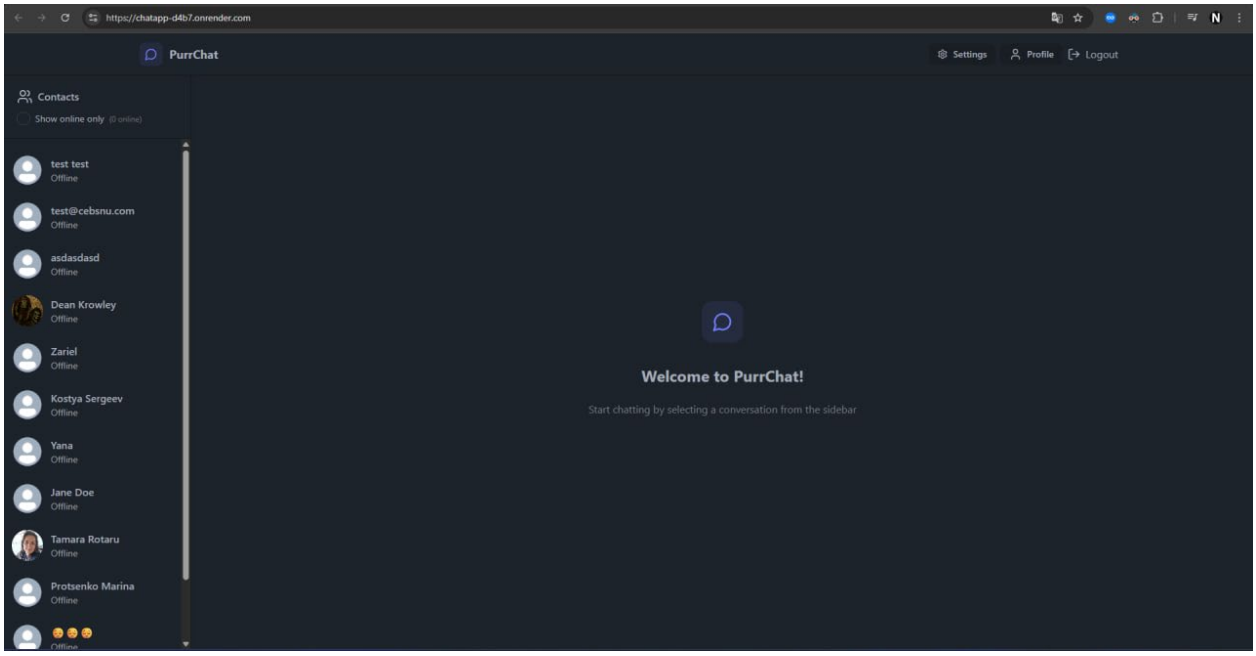


Рисунок 4.106 – Результат роботи

Висновки до розділу 4

В зазначеному розділі було використано повну серверну частину веб додатку, щоб була змога щодо обміну повідомленнями це включає будову з використанням Node.js, Express і MongoDB.

Розроблено дві основні Mongoose-моделі — User та Message, між ними встановлено зв’язок через поля ObjectId, що дало змогу зберігати будову відправника та отримувача для кожного повідомлення. Нарізними маршрутами (auth.route.js, message.route.js) було використовувано функцію реєстрації, підтвердження ідентичності, обробки повідомлень та усіляких дій з ними, а також можливості оновлення профілю.

Для безпечності системи було використано middleware authMiddleware.js, це в свою чергу перевіряє JWT-токен і захищає особисті маршрути та функцію generateToken, яка дає можливість генерувати токен доступу з відповідними атрибутами захисту (httpOnly, sameSite, secure). Завдяки підключенню Cloudinary у файлі cloudinary.js реалізовано безпечне завантаження зображень із збереженням їх secure_url. А от щоб обробка відбувалася в реальному часі реалізована за допомогою бібліотеки socket.io, що забезпечує поєднання користувачів, ведення мапи онлайн-ідентифікаторів та можливість передавати повідомлення у режимі реального часу.

Також, було успішно розгорнуто вебзастосунок на платформі Render Dashboard і викладено його на загальний доступ.

Було використана можливість структуровано розроблену, безпечну і доволі масштабовану серверну частину, яка забезпечила ідентифікацію щодо зберігання та обміну різними повідомленнями, а також можливість взаємодії з медіа даними в хмарному облаку та розгорнули платформу у загальний доступ на платформі Render Dashboard.

ВИСНОВКИ

У даній кваліфікаційній роботі було зроблено повне коло розробки веб додатку PurrChat для корпоративної комунікації в режимі реального часу, саме він відповідає осучасненим вимогам щодо швидкості, безпечності , для обсягу та зручності у користуванні.

По-перше, проведений аналіз сучасних аналогів (Microsoft Teams, Discord, Telegram, WhatsApp) та акцентоване зростання ролі месенджерів як універсальних інструментів для цифрової взаємодії. Проєкт PurrChat виявився як конкурентоспроможна альтернатива замість використанню SPA-побудови, підтримці об’ємності та фокусі на реальному часі. Було виявлені ключові технічні виклики — мінімізація усіляких затримок, обробка завеликої кількості з'єднань та забезпечення безпеки, які можуть вирішуватись за допомогою JWT-автентифікації, bcryptjs-шифрування та інтеграції з хмарними облаками на кшталт Cloudinary.

По-друге, підлягало розгляду теорія основи створення розподілених архітектур і обґрунтовано сам вибір спрощеної DHT-моделі як дуже перспективного направлення для подальшого вдосконалення. У розробці було використовувано сучасні технології, які дали змогу реалізації окремих архітектурних компонентів — обробку повідомлень, автентифікацію, профілі усіх користувачів та роботу з медіаконтентом — як незалежні та узагальнені блоки. Що насамперед дозволяє системі залишатись адаптованою до навантаження і можливості бути відкритою в подальшому для розширення.

По-третє, також було представлено логічний ланцюжок системи через ER-діаграми та діаграми послідовності, що демонструють зв'язки між головними компонентами (а саме: користувачами та повідомленнями), також підтверджена ефективність всіх механізмів, які реалізовані обміну повідомленнями в онлайн режимі через socket.io, а також надійність системи реєстрації з використанням middleware. Завдяки вищезазначеному опису досягнуто повної функціональної

інтеграції серверу та клієнтської частин із дуже навіть високим рівнем взаємодій в подальшому.

І в останню чергу слід зауважити, що розкриті деталі реалізації серверного ланцюга з використанням Node.js, Express, MongoDB, Mongoose та хмарного сервісу Cloudinary, що дало змогу забезпечити безпечну, надійну, а найголовніше: ефективну інфраструктуру. Сама система була використана на платформі Render Dashboard та доволі успішно протестована. В результаті створений вебзастосунок з повною можливістю авторизації, обміном повідомленнями, можливістю завантаження зображень, підтримкою реального часу та об’ємною будовою.

Тобто, PurrChat — виявився практичним втіленням сучасного рішення у сфері вебкомунікацій, що в свою чергу довело свою якість, працездатність, ефективність і потенціал в подальшого для розвитку як на технічному, так і на практичному рівнях. Робота поєднала теоретичний аналіз з практичним застосуванням, що показало вміння інтегрувати складні технологічні рішення у доволі цілісну сучасну вебсистему.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Install daisyUI as a Tailwind plugin. URL: https://daisyui.com/docs/install/ (date of access: 14.04.2025).

2. Cornelissen, J. (2014). Corporate Communication: A Guide to Theory and Practice (4th ed.). SAGE Publications.

3. Cornelissen, J. (2014). Corporate Communication: A Guide to Theory and Practice (4th ed.). SAGE Publications.

4. Що таке Axios?. URL: https://axios-http.com/uk/docs/intro (дата звернення: 08.04.2025).

5. Cowan, D. (2014). Strategic Internal Communication: How to Build Employee Engagement and Performance. Kogan Page.

6. Gartner, U. (2021). Corporate Communications in Restructuring Phases: Successfully Shaping Change with Strategic Communication. Springer.

7. Lucide React URL: https://lucide.dev/guide/packages/lucide-react (date of access: 16.04.2025).

8. Marinchak, C. L. M., & Deluliis, S. M. (2023). Corporate Communication and Integrated Marketing Communication: Audience Beyond Stakeholders in a Technological Age. Lexington Books.

9. Dahlman, S., & Heide, M. (2020). Strategic Internal Communication: A Practitioner's Guide to Implementing Cutting-Edge Methods for Improved Workplace Culture. Routledge.

10. Schiefelbein, J. (2024). Effective Business Communication for Dummies. Wiley.

11. React Reference Overview. URL: https://react.dev/reference/react (date of access: 04.04.2025).

12. Bennetch, R., Owen, C., & Keesey, Z. (2021). Effective Professional Communication: A Rhetorical Approach. Open Textbook Library.

13. Greene, J. O. (2021). Essentials of Communication Skill and Skill Enhancement: A Primer for Students and Professionals. Routledge.

14. ReactDOM. URL: https://ru.legacy.reactjs.org/docs/react-dom.html (date of access: 09.04.2025).

15. Strawser, M. G., Smith, S. A., & Rubenking, B. (2021). Multigenerational Communication in Organizations: Insights from the Workplace. Routledge.

16. Casteleyn, S., Daniel, F., Dolog, P., & Matera, M. (2009). Engineering Web Applications. Springer.

17. Mendes, E. (2005). Cost Estimation Techniques for Web Projects. IGI Publishing.

18. What Socket.IO is. URL: https://socket.io/docs/v4/ (date of access: 25.06.2025).

19. Kappel, G., Pröll, B., Reich, S., & Retschitzegger, W. (2006). Web Engineering - The Discipline of Systematic Development of Web Applications. John Wiley & Sons.

20. Rossi, G., Pastor, O., Schwabe, D., & Olsina, L. (2007). Web Engineering: Modelling and Implementing Web Applications. Springer.

21. Suh, W. (2005). Web Engineering: Principles and Techniques. Idea Group Publishing.

22. Draheim, D., & Weber, G. (2005). Form-Oriented Analysis -- A New Methodology to Model Form-Based Applications. Springer.

23. How to use Zustand. URL: https://zustand.docs.pmnd.rs/getting-started/introduction (date of access: 20.05.2025).

24. Conallen, J. (2003). Building Web Applications with UML (2nd ed.). Pearson Education.

25. Morville, P., & Rosenfeld, L. (2002). Information Architecture for the World Wide Web (2nd ed.). O'Reilly.

26. Cloudinary. URL: https://cloudinary.com/ (date of access: 12.05.2025).

27. Powell, T. A., Jones, D. L., & Cutts, D. C. (1998). Web Site Engineering: Beyond Web Page Design. Prentice Hall.

28. Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., & Matera, M. (2002). Designing Data-Intensive Web Applications. Morgan Kaufmann.

29. Cookie-parser. URL: https://expressjs.com/en/resources/middleware/cookie-parser.html (date of access: 01.05.2025).

30. Tanenbaum, A. S., & van Steen, M. (2007). Distributed Systems: Principles and Paradigms (2nd ed.). Prentice Hall.

31. Alshahrani, S., Al-Rakhami, M., Alfarraj, O., & Alalwan, N. (2021). Blockchain-based applications in education: A systematic review. Egyptian Informatics Journal, 22(3), 331–339. https://doi.org/10.1016/j.eij.2021.03.003

32. Bass, L., Clements, P., & Kazman, R. (2012). Software Architecture in Practice (3rd ed.). Addison-Wesley.

33. Dotenv-vault. URL: https://www.dotenv.org/docs/quickstart (date of access: 28.04.2025).

34. Express. URL: https://devdocs.io/express/ (date of access: 04.04.2025).

35. Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media.

36. Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley.

37. Get Started with Atlas. URL: https://www.mongodb.com/docs/atlas/getting-started/ (date of access: 04.04.2025).

38. Hohpe, G., & Woolf, B. (2003). Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley.

39. Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley.

40. Why Vite. URL: https://vite.dev/guide/why.html (date of access: 02.04.2025).

41. Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly Media.

42. Lunt, B. M., Ekstrom, J. J., Gorka, S., & Miller, J. (2010). Principles of Web Development. Cengage Learning.

ДОДАТОК А

Головний файл для вебзастосунку App.jsx

```
import Navbar from "./components/Navbar";
import HomePage from "./pages/HomePage";
import SignUpPage from "./pages/SignUpPage";
import LoginPage from "./pages/LoginPage";
import SettingsPage from "./pages/SettingsPage";
import ProfilePage from "./pages/ProfilePage";
import { Routes, Route, Navigate } from "react-router-dom";
import { useAuthStore  } from "./store/useAuthStore";
import { useEffect } from "react;
import { Loader } from "lucide-react";
import { Toaster } from "react-hot-toast";
import { useThemeStore } from "./store/useThemeStore";
const App = () => {
  const {authUser, checkAuth, isCheckingAuth} = useAuthStore()
  const {theme} = useThemeStore()

  useEffect(() => {
    checkAuth();
  }, [checkAuth]);

  if(isCheckingAuth && !authUser) return (
    <div className="flex items-center justify-center h-screen">
      <Loader className = "size-10 animate-spin" />
    </div>
  )
  return (
    <div data-theme={theme}>
      <Navbar />
      <Routes>
        <Route path="/" element={authUser ? <HomePage /> : <Navigate to="/login" />}
/>
        <Route path="/signup" element={!authUser ?  <SignUpPage /> : <Navigate to="/"
/>} />
        <Route path="/login" element={!authUser ? <LoginPage /> : <Navigate to="/"
/>} />
        <Route path="/settings" element={<SettingsPage />} />
        <Route path="/profile" element={authUser ? <ProfilePage /> : <Navigate
to="/login" />} />
      </Routes>
      <Toaster />
    </div>
  );
}
export default App;
```

Керування станом автентифікації за допомогою useAuthStore

```
import { create } from "zustand";
import { axiosInstance } from "../lib/axios.js";
import toast from "react-hot-toast";
import { io } from "socket.io-client";
export const useAuthStore = create((set, get) => ({ authUser: null, isSigningUp:
false, isLoggingIn: false, isUpdatingProfile: false, isCheckingAuth: true,
onlineUsers: [], socket: null,
    checkAuth: async () => {
        try {
            const res = await axiosInstance.get("/auth/check");
            set({ authUser: res.data });
            get().connectSocket()
        } catch (error) {
            console.log("Error in checkAuth:", error);
            set({ authUser: null });
        } finally {
            set({ isCheckingAuth: false });
        }},
    signup: async (data) => {
        set({ isSigningUp: true });
        try {
           const res = await axiosInstance.post("/auth/signup", data);
           set({ authUser: res.data });
           toast.success("Account created successfully");
           get().connectSocket()
        } catch (error) {
            toast.error(error.response.data.message);
        } finally {
            set({ isSigningUp: false });
        }},
    login: async (data) => {
        set({ isLoggingIn: true });
        try {
            const res = await axiosInstance.post("/auth/login", data);
            if (res && res.data) {
                set({ authUser: res.data });
                toast.success("Logged in successfully");
                get().connectSocket()
            } else {
                throw new Error("Invalid response format");
            }
        } catch (error) {
            console.error("Login error:", error);
            const errorMessage = error.response ? error.response.data.message :
error.message;
            toast.error(errorMessage || "An error occurred during login");
```

```
        } finally {
            set({ isLoggingIn: false });
        }},
    logout: async () => {
        try {
          await axiosInstance.post("/auth/logout");
          set({ authUser: null });
          toast.success("Logged out successfully");
          get().disconnectSocket()
        } catch (error) {
            toast.error(error.response.data.message);
        }},
    updateProfile: async (data) => {
        set({ isUpdatingProfile: true });
        try {
          const res = await axiosInstance.put("/auth/update-profile", data);
          const updatedUser = { ...get().authUser, ...res.data };
          set({ authUser: updatedUser });
          return true;
        } catch (error) {
          if (error.response?.data?.message === "NEW_PASSWORD_SAME_AS_CURRENT") {
            throw new Error("NEW_PASSWORD_SAME_AS_CURRENT");
          }
          throw new Error(error.response?.data?.message || "Update failed");
        } finally {
          set({ isUpdatingProfile: false });
        }},
    connectSocket: () => {
        const { authUser } = get()
        if(!authUser || get().socket?.connected) return;
        const socket = io(BASE_URL, {
            query: {
                userId: authUser._id,
            },
        })
        socket.connect()
        set({ socket: socket});
        socket.on("getOnlineUsers", (userIds) => {
            set({onlineUsers: userIds})
        });
},
    disconnectSocket: () => {
        if(get().socket?.connected) get().socket.disconnect();
    },
}));
```

Керування станом чату за допомогою useChatStore

```
import { create } from "zustand";
import toast from "react-hot-toast";
import { axiosInstance } from "../lib/axios";
import { useAuthStore } from "./useAuthStore";
export const useChatStore = create((set, get) => ({ messages: [], users: [],
selectedUser: null, isUsersLoading: false, isMessagesLoading: false, socket: null,
  getUsers: async () => {
    set({ isUsersLoading: true });
    try {
      const res = await axiosInstance.get("/messages/users");
      set({ users: res.data });
    } catch (error) {
      toast.error(error.response.data.message);
    } finally {
      set({ isUsersLoading: false });
    }},
  getMessages: async (userId) => {
    set({ isMessagesLoading: true });
    try {
      const res = await axiosInstance.get(`/messages/${userId}`);
      set({ messages: res.data });
    } catch (error) {
      toast.error(error.response.data.message);
    } finally {
      set({ isMessagesLoading: false });
    }},
  sendMessage: async (messageData) => {
    const { selectedUser, messages } = get();
    try {
      const res = await axiosInstance.post(`/messages/send/${selectedUser._id}`,
messageData);
      set({ messages: [...messages, res.data] });
    } catch (error) {
      toast.error(error.response.data.message);
    }},
  subscribeToMessages: () => {
    const { selectedUser } = get();
    const socket = useAuthStore.getState().socket;
    const authUser = useAuthStore.getState().authUser;
    if (!selectedUser || !socket || !socket.connected) return;
    socket.off("newMessage");
    socket.off("messageUpdated");
    socket.off("messageDeleted");
    socket.on("newMessage", (newMessage) => {
      const currentMessages = get().messages;
      const isDuplicate = currentMessages.some(msg => msg._id === newMessage._id);
```

```
      const isRelevantMessage =
        (newMessage.senderId === authUser._id && newMessage.receiverId ===
selectedUser._id) ||
        (newMessage.senderId === selectedUser._id && newMessage.receiverId ===
authUser._id);
      if (!isDuplicate && isRelevantMessage) {
        set({ messages: [...currentMessages, newMessage] });
      }});
    socket.on("messageUpdated", (updatedMessage) => {
      const updatedMessages = get().messages.map(msg =>
        msg._id === updatedMessage._id ? updatedMessage : msg
      );
      set({ messages: updatedMessages });
    });
    socket.on("messageDeleted", (deletedMessageId) => {
      const filteredMessages = get().messages.filter(msg => msg._id !==
deletedMessageId);
      set({ messages: filteredMessages });
    });},
  unsubscribeToMessages: () => {
    const { socket } = get();
    if (socket) {
      socket.off("newMessage"); }},
  editMessage: async (messageId, text, image) => {
    try {
      const { messages } = get();
      const res = await axiosInstance.put(`/messages/${messageId}`, { text, image });
      const updatedMessages = messages.map(msg => msg._id === messageId ? res.data :
msg);
      set({ messages: updatedMessages });
      return res.data;
    } catch (error) {
      toast.error(error.response.data.message);
      throw error;}},
  deleteMessage: async (messageId) => {
    try {
      const { messages } = get();
      await axiosInstance.delete(`/messages/${messageId}`);
      const updatedMessages = messages.filter(msg => msg._id !== messageId);
      set({ messages: updatedMessages });
    } catch (error) {
      toast.error(error.response.data.message);
      throw error;
    }},
  setSelectedUser: (selectedUser) => set({ selectedUser }), }));
```

Компонент HomePage: Головна сторінка

```
import ChatContainer from "../components/ChatContainer";
import NoChatSelected from "../components/NoChatSelected";
import Sidebar from "../components/Sidebar";
import { useChatStore } from "../store/useChatStore";

const HomePage = () => {
    const { selectedUser } = useChatStore();

    return (
        <div className="h-screen bg-base-200 flex flex-col">
            <div className="h-[65px]"></div>
            <div className="flex-1 w-full overflow-hidden">
                <div className="h-full w-full bg-base-100">
                    <div className="flex h-full">
                        <Sidebar />
                        {!selectedUser ? <NoChatSelected /> : <ChatContainer />}
                    </div>
                </div>
            </div>
        </div>
    );
};

export default HomePage;
```

Список тем, які виеористовуються у додатку

```
export const THEMES = [
    "light",
    "dark",
    "cupcake",
    "bumblebee",
    "emerald",
    "corporate",
    "synthwave",
    "retro",
    "cyberpunk",
    "valentine",
    "halloween",
    "garden",
    "forest",
    "aqua",
    "lofi",
    "pastel",
    "fantasy",
    "wireframe",
    "black",
    "luxury",
    "dracula",
    "cmyk",
    "autumn",
    "business",
    "acid",
    "lemonade",
    "night",
    "coffee",
    "winter",
    "dim",
    "nord",
    "sunset",
  ];
```

Компонент Sidebar зі списком користувачів і фільтром онлайн

```
import { useEffect, useState } from "react";
import { useChatStore } from "../store/useChatStore";
import { useAuthStore } from "../store/useAuthStore";
import { Users } from "lucide-react";
import SidebarSkeleton from "./skeleton/SidebarSkeleton";
const Sidebar = () => {
  const { getUsers, selectedUser, setSelectedUser, isUsersLoading, users } =
useChatStore();
  const { onlineUsers = [] } = useAuthStore();
  const [showOnlineOnly, setShowOnlineOnly] = useState(false);
  useEffect(() => {
    getUsers();
  }, [getUsers]);
  const filteredUsers = showOnlineOnly
    ? users?.filter(user => onlineUsers.includes(user._id)) || []
    : users || [];
  if (isUsersLoading) return <SidebarSkeleton />;
  const toggleOnlineOnly = () => setShowOnlineOnly(prev => !prev);
  return (
    <aside className="h-full w-20 lg:w-72 border-r border-base-300 flex flex-col
transition-all duration-200">
      <div className="border-b border-base-300 w-full p-5">
        <div className="flex items-center gap-2">
          <button
            onClick={toggleOnlineOnly}
            className={`
              lg:hidden p-1 rounded-md transition-colors
              ${showOnlineOnly ? "bg-green-500 text-white" : "hover:bg-base-200"}
            `}
          >
            <Users className="size-6" />
          </button>

          <div className="hidden lg:flex items-center gap-2">
            <Users className="size-6" />
            <span className="font-medium">Contacts</span>
          </div>
        </div>
        <div className="mt-3 hidden lg:flex items-center gap-2">
          <label className="cursor-pointer flex items-center gap-2">
            <input
              type="checkbox"
              checked={showOnlineOnly}
              onChange={(e) => setShowOnlineOnly(e.target.checked)}
              className="checkbox checkbox-sm"
            />
```

```
            <span className="text-sm">Show online only</span>
          </label>
          <span className="text-xs text-zinc-500">
            ({Math.max(0, onlineUsers.length - 1)} online)
          </span>
        </div>
      </div>
      <div className="overflow-y-auto w-full py-3">
        {filteredUsers.map((user) => (
          <button
            key={user._id}
            onClick={() => setSelectedUser(user)}
            className={`
              w-full p-3 flex items-center gap-3
              hover:bg-base-300 transition-colors
              ${selectedUser?._id === user._id ? "bg-base-300 ring-1 ring-base-300" :
""}
            `}
          >
            <div className="relative mx-auto lg:mx-0">
              <img
                src={user.profilePic || "/avatar.png"}
                alt={user.fullName}
                className="size-12 object-cover rounded-full"
              />
              {onlineUsers.includes(user._id) && (
                <span
                  className="absolute bottom-0 right-0 size-3 bg-green-500
                  rounded-full ring-2 ring-zinc-900"
                />
              )}
            </div>
            <div className="hidden lg:block text-left min-w-0">
              <div className="font-medium truncate">{user.fullName}</div>
              <div className="text-sm text-zinc-400">
                {onlineUsers.includes(user._id) ? "Online" : "Offline"}
              </div>
            </div>
          </button>
        ))}
      </div>
    </aside>
  );
};
export default Sidebar
```

Компонент SidebarSkeleton для анімації під час отримання юзерів

```
import { Users } from "lucide-react";

const SidebarSkeleton = () => {
  // Create 8 skeleton items
  const skeletonContacts = Array(8).fill(null);

  return (
    <aside
      className="h-full w-20 lg:w-72 border-r border-base-300
    flex flex-col transition-all duration-200"
    >
      {/* Header */}
      <div className="border-b border-base-300 w-full p-5">
        <div className="flex items-center gap-2">
          <Users className="w-6 h-6" />
          <span className="font-medium hidden lg:block">Contacts</span>
        </div>
      </div>

      {/* Skeleton Contacts */}
      <div className="overflow-y-auto w-full py-3">
        {skeletonContacts.map((_, idx) => (
          <div key={idx} className="w-full p-3 flex items-center gap-3">
            {/* Avatar skeleton */}
            <div className="relative mx-auto lg:mx-0">
              <div className="skeleton size-12 rounded-full" />
            </div>

            {/* User info skeleton - only visible on larger screens */}
            <div className="hidden lg:block text-left min-w-0 flex-1">
              <div className="skeleton h-4 w-32 mb-2" />
              <div className="skeleton h-3 w-16" />
            </div>
          </div>
        ))}
      </div>
    </aside>
  );
};

export default SidebarSkeleton;
```

Компонент MessageSkeleton для анімації під час отримання повідомлень

```
const MessageSkeleton = () => {
    // Create an array of 6 items for skeleton messages
    const skeletonMessages = Array(6).fill(null);

    return (
      <div className="flex-1 overflow-y-auto p-4 space-y-4">
        {skeletonMessages.map((_, idx) => (
          <div key={idx} className={`chat ${idx % 2 === 0 ? "chat-start" : "chat-
end"}`}>
            <div className="chat-image avatar">
              <div className="size-10 rounded-full">
                <div className="skeleton w-full h-full rounded-full" />
              </div>
            </div>

            <div className="chat-header mb-1">
              <div className="skeleton h-4 w-16" />
            </div>

            <div className="chat-bubble bg-transparent p-0">
              <div className="skeleton h-16 w-[200px]" />
            </div>
          </div>
        ))}
      </div>
    );
  };

  export default MessageSkeleton;
```

Головний файл для запуску серверної частини index.js

```
import express from "express";
import dotenv from "dotenv";
import cookieParser from "cookie-parser";
import cors from "cors";
import path from "path";
import { fileURLToPath } from "url";
import { dirname } from "path";
import { connectDB } from "./lib/db.js";
import authRoutes from "./routes/auth.route.js";
import messageRoutes from "./routes/message.route.js";
import { app, server } from "./lib/socket.js";
dotenv.config();
const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);
console.log("__filename:", __filename);
console.log("__dirname:", __dirname);
const PORT = process.env.PORT;
app.use((req, res, next) => {
  console.log('Received request for:', req.originalUrl);
  next();
});
app.use(express.json({ limit: "10mb" }));
app.use(cookieParser());
app.use(
  cors({
    origin: "http://localhost:5173",
    credentials: true
  })
);
app.use("/api/auth", authRoutes);
app.use("/api/messages", messageRoutes);
const distPath = path.join(__dirname, "..", "..", "frontend", "dist");
console.log("Dist path:", distPath);
app.use(express.static(distPath));
app.use("/", (req, res) => {
  console.log("Sending index.html from:", distPath);
  res.sendFile(path.join(distPath, "index.html"));
});
connectDB().then(() => {
  server.listen(PORT, () => {
    console.log("Server is running on PORT: " + PORT);
  });
});
```

Контролер, який відповідає за логіку автентифікації

```
import { generateToken } from "../lib/units.js";
import User from "../models/user.model.js";
import bcrypt from "bcryptjs";
import cloudinary from "../lib/cloudinary.js";
export const signup = async (req, res) => {
  const { fullName, email, password } = req.body;
  try {
    if (!fullName || !email || !password) {
      return res.status(400).json({ message: "All fields are required" });
    }
    if (password.length < 6) {
      return res.status(400).json({ message: "Password must be at least 6 characters"
});
    }
    const user = await User.findOne({ email });
    if (user) return res.status(400).json({ message: "Email already exists" });
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);
    const newUser = new User({
      fullName,
      email,
      password: hashedPassword,
    });
    if (newUser) {
      generateToken(newUser._id, res);
      await newUser.save();
      res.status(201).json({
        _id: newUser._id,
        fullName: newUser.fullName,
        email: newUser.email,
        profilePic: newUser.profilePic,
      });
    } else {
      res.status(400).json({ message: "Invalid user data" });
    }
  } catch (error) {
    console.log("Error in signup controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
};
export const login = async (req, res) => {
  const { email, password } = req.body;
  try {
    const user = await User.findOne({ email });
    if (!user) {
      return res.status(400).json({ message: "Invalid credentials" });
```

```
    }
    const isPasswordCorrect = await bcrypt.compare(password, user.password);
    if (!isPasswordCorrect) {
      return res.status(400).json({ message: "Invalid credentials" });
    }
    generateToken(user._id, res);
    res.status(200).json({
      _id: user._id,
      fullName: user.fullName,
      email: user.email,
      profilePic: user.profilePic,
    });
  } catch (error) {
    console.log("Error in login controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
};
export const logout = (req, res) => {
  try {
    res.cookie("jwt", "", { maxAge: 0 });
    res.status(200).json({ message: "Logged out successfully" });
  } catch (error) {
    console.log("Error in logout controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
};
export const updateProfile = async (req, res) => {
  try {
    const { fullName, email, password, profilePic, removeProfilePic } = req.body;
    const userId = req.user._id;
    const updateData = {};
    if (fullName) {
      if (fullName.trim().length < 3) {
        return res.status(400).json({ message: "Name must be at least 3 characters"
});
      }
      updateData.fullName = fullName;
    }
    if (email) {
      if (!/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email)) {
        return res.status(400).json({ message: "Invalid email format" });
      }

      const existingUser = await User.findOne({ email });
      if (existingUser && existingUser._id.toString() !== userId.toString()) {
        return res.status(400).json({ message: "Email already in use" });
      }
      updateData.email = email;
```

```
    }
    if (password) {
      if (password.length < 6) {
        return res.status(400).json({ message: "Password must be at least 6
characters" });
      }
      const user = await User.findById(userId);
      const isSamePassword = await bcrypt.compare(password, user.password);
      if (isSamePassword) {
        return res.status(400).json({ message: "NEW_PASSWORD_SAME_AS_CURRENT" });
      }
      const salt = await bcrypt.genSalt(10);
      updateData.password = await bcrypt.hash(password, salt);
    }
    if (removeProfilePic) {
      updateData.profilePic = null;
    } else if (profilePic && profilePic.startsWith('data:image')) {
      const uploadResponse = await cloudinary.uploader.upload(profilePic);
      updateData.profilePic = uploadResponse.secure_url;
    }
    const updatedUser = await User.findByIdAndUpdate(
      userId,
      updateData,
      { new: true }
    ).select("-password");
    res.status(200).json(updatedUser);
  } catch (error) {
    console.log("Error in update profile:", error);
    res.status(500).json({ message: "Internal server error" });
  }
};
export const checkAuth = (req, res) => {
  try {
    res.status(200).json(req.user);
  } catch (error) {
    console.log("Error in checkAuth controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
};
```

Контролер, який відповідає за логіку повідомлень

```
import User from "../models/user.model.js";
import Message from "../models/message.model.js";
import clodinary from "../lib/cloudinary.js";
import { getReceiverSocketId, io } from "../lib/socket.js";
export const getUsersForSidebar = async(req, res) => {
    try {
        const loggedInUserId = req.user._id;
        const filteredUsers = await User.find({_id: {$ne:loggedInUserId}}).select("-
password");
        res.status(200).json(filteredUsers);
    } catch(error) {
        console.error("Error in getUsersForSidebar: ", error.message);
        res.status(500).json({ error: "Internal server error" });
    }
};
export const getMessages = async(req, res) => {
    try {
        const { id:userToChatId } = req.params
        const myId = req.user._id;
        const messages = await Message.find({
            $or: [
                {senderId:myId, receiverId:userToChatId},
                {senderId:userToChatId, receiverId:myId}
            ]
        })
        res.status(200).json(messages)
    } catch(error) {
        console.log("Error in getMessages controller: ", error.message);
        res.status(500).json({ error: "Internal server error" });
    }
};
export const sendMessage = async (req, res) => {
    try {
        const { text, image } = req.body;
        const { id: receiverId } = req.params;
        const senderId = req.user._id;
        let imageUrl = null;
        if (image) {
            const uploadResponse = await clodinary.uploader.upload(image);
            imageUrl = uploadResponse.secure_url;
        }
        const newMessage = new Message({
            senderId,
            receiverId,
            text,
            image: imageUrl,
```

```
        });
        await newMessage.save();
        const receiverSocketId = getReceiverSocketId(receiverId);
        if(receiverSocketId) {
          io.to(receiverSocketId).emit("newMessage", newMessage);
        }
        res.status(201).json(newMessage);
    } catch (error) {
        console.log("Error in sendMessage controller: ", error.message);
        res.status(500).json({ error: "Internal server error" });
    }
};
export const editMessage = async (req, res) => {
  try {
    const { messageId } = req.params;
    const { text, image } = req.body;

    let imageUrl = image || null;

    if (image && image.startsWith("data:image/")) {
      const uploadResponse = await clodinary.uploader.upload(image);
      imageUrl = uploadResponse.secure_url;
    }

    const updatedMessage = await Message.findByIdAndUpdate(
      messageId,
      { text, image: imageUrl },
      { new: true }
    );
    if (!updatedMessage) {
      return res.status(404).json({ error: "Message not found" });
    }
    res.status(200).json(updatedMessage);
    io.emit("messageUpdated", updatedMessage);
  } catch (error) {
    console.log("Error in editMessage controller: ", error.message);
    res.status(500).json({ error: "Internal server error" });
  }
};
export const deleteMessage = async (req, res) => {
    try {
      const { messageId } = req.params;
      const deletedMessage = await Message.findByIdAndDelete(messageId);
      if (!deletedMessage) {
        return res.status(404).json({ error: "Message not found" });
      }
      res.status(200).json({ message: "Message deleted successfully" });
      io.emit("messageDeleted", messageId);
```

```
    } catch (error) {
      console.log("Error in deleteMessage controller: ", error.message);
      res.status(500).json({ error: "Internal server error" });
    }
};
export const replaceImage = async (req, res) => {
    try {
      const { messageId } = req.params;
      const { image } = req.body;
      const uploadResponse = await clodinary.uploader.upload(image);
      const imageUrl = uploadResponse.secure_url;
      const updatedMessage = await Message.findByIdAndUpdate(
        messageId,
        { image: imageUrl },
        { new: true }
      );
      if (!updatedMessage) {
        return res.status(404).json({ error: "Message not found" });
      }
      res.status(200).json(updatedMessage);
      io.emit("messageUpdated", updatedMessage);
    } catch (error) {
      console.log("Error in replaceImage controller: ", error.message);
      res.status(500).json({ error: "Internal server error" });
    }
};
export const removeImage = async (req, res) => {
    try {
      const { messageId } = req.params;
      const updatedMessage = await Message.findByIdAndUpdate(
        messageId,
        { image: "" },
        { new: true }
      );
      if (!updatedMessage) {
        return res.status(404).json({ error: "Message not found" });
      }
      res.status(200).json(updatedMessage);
      io.emit("messageUpdated", updatedMessage);
    } catch (error) {
      console.log("Error in removeImage controller: ", error.message);
      res.status(500).json({ error: "Internal server error" });
    }
};
```

JWT-middleware для перевірки авторизації

```
import jwt from "jsonwebtoken";
import User from "../models/user.model.js";

export const protectRoute = async (req, res, next) => {
    try {
        const token = req.cookies.jwt

        if(!token) {
            return res.status(401).json({message: "Unnathorized - No Token
Provided"});
        }

        const decoded = jwt.verify(token, process.env.JWT_SECRET);

        if(!decoded) {
            return res.status(401).json({message: "Unauthorized - Invalid Token"});
        }

        const user = await User.findById(decoded.userId).select("-password");

        if(!user) {
            return res.status(404).json({message: "User not found"});
        }

        req.user = user
        next()

    } catch(error) {
        console.log("Error in protectRoute middleware:", error.message);
        res.status(500).json({message: "Internal server error"});
    }
}
```

Mongoose-схема Message: модель повідомлень між користувачами

```
import mongoose from "mongoose";

const messageSchema = new mongoose.Schema(
  {
    senderId: {
      type: mongoose.Schema.Types.ObjectId,
      ref: "User",
      required: true,
    },
    receiverId: {
      type: mongoose.Schema.Types.ObjectId,
      ref: "User",
      required: true,
    },
    text: {
      type: String,
    },
    image: {
      type: String,
    },
  },
  { timestamps: true }
);

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

export default Message;
```

Сервер Socket.IO

```
import { Server } from "socket.io";
import http from "http";
import express from "express";

const app = express();
const server = http.createServer(app);

const io = new Server (server, {
    cors: {
        origin: ["http://localhost:5173"]
    },
});

export function getReceiverSocketId(userId) {
    return userSocketMap[userId];
}

const socketUserMap = {};
const userSocketMap = {};

io.on("connection", (socket) => {
    const userId = socket.handshake.query.userId;
    if (userId) {
        userSocketMap[userId] = socket.id;
        socketUserMap[socket.id] = userId;
    }

    io.emit("getOnlineUsers", Object.keys(userSocketMap));

    socket.on("disconnect", () => {
        const disconnectedUserId = socketUserMap[socket.id];
        delete userSocketMap[disconnectedUserId];
        delete socketUserMap[socket.id];
        io.emit("getOnlineUsers", Object.keys(userSocketMap));
    });
});

export { io, app, server };
```

Конфігурація підключення до Cloudinary

```
import {v2 as clodinary} from "cloudinary";

import {config} from 'dotenv';

config();

clodinary.config({
    cloud_name: process.env.CLOUDINARY_CLOUD_NAME,
    api_key: process.env.CLOUDINARY_API_KEY,
    api_secret: process.env.CLOUDINARY_API_SECRET,
});

export default clodinary;
```

Маршрути обробки повідомлень із захистом доступу

```
import express from "express";
import { protectRoute } from "../middleware/auth.middleware.js";
import { getMessages, getUsersForSidebar, sendMessage, editMessage, deleteMessage,
replaceImage, removeImage } from "../controllers/message.controller.js";
const router = express.Router();

router.get("/users", protectRoute, getUsersForSidebar);
router.get("/:id", protectRoute, getMessages);
router.post("/send/:id", protectRoute, sendMessage);
router.put("/:messageId", protectRoute, editMessage);
router.delete("/:messageId", protectRoute, deleteMessage);
router.put("/image/:messageId", protectRoute, replaceImage);
router.delete("/image/:messageId", protectRoute, removeImage);

export default router;
```

Функція підключення до MongoDB через Mongoose з обробкою помилок

```
import mongoose from "mongoose";

export const connectDB = async () => {
  try {
    mongoose.set('strictQuery', true);
    const conn = await mongoose.connect(process.env.MONGODB_URI);
    console.log(`MongoDB connected: ${conn.connection.host}`);
  } catch (error) {
    console.log("MongoDB connection error:", error);
  }
};
```

Кафедра інтелектуальних інформаційних систем
Вебсзастосунок для корпоративної комунікації

Конфігураційний файл .env

```
MONGODB_URI=mongodb+srv://Mykola:qztZhdc444V6hin3@cluster0.d8ofhbp.mongodb.net/chat_db?retryWrites=true&w=majority
PORT=5001
JWT_SECRET= Q29vZFNlY3JldEtleUluQmFzZTY0Rm9ybWF0IQo=
NODE_ENV=development
CLOUDINARY_CLOUD_NAME=dmy88rasg
CLOUDINARY_API_KEY=544696348353777
CLOUDINARY_API_SECRET=ADDUcBahCzoqfl6hk-phL6gkqak
```