Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Запорізька політехніка»

Комп’ютерних наук і технологій
(повне найменування факультету)

Комп’ютерні системи та мережі
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

бакалаврський
(ступінь вищої освіти)

на тему: РОЗРОБКА ВЕБСИСТЕМИ МЕСЕНДЖЕРА НА ОСНОВІ БІБЛІОТЕКИ REACTJS

Виконав(ла): студент(ка) 4 курсу,
групи КНТ-521

Спеціальності
123 Комп’ютерна інженерія
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Комп’ютерна інженерія
(назва освітньої програми (спеціалізації)

КРАВЧЕНКО І.В.
(ПРІЗВИЩЕ та ініціали)

Керівник СКРУПСЬКИЙ С.Ю.
(ПРІЗВИЩЕ та ініціали)

Рецензент СТЕПАНЕНКО О.О.
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет Комп'ютерних наук і технологій
Кафедра комп'ютерних систем та мереж
Ступінь вищої освіти бакалаврський
Спеціальність 123 Комп'ютерна інженерія
(код і найменування)
Освітня програма (спеціалізація) Комп'ютерна інженерія
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ
Завідувач кафедри КУДЕРМЕТОВ Р.К.

«14» квітня 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

КРАВЧЕНКА Івана Віталійовича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка вебсистеми месенджера на основі бібліотеки ReactJS

керівник проєкту (роботи) к.т.н., доцент, СКРУПСЬКИЙ С.Ю.
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом закладу вищої освіти від « 08 » квітня 2025 року № 151
2. Строк подання студентом проєкту (роботи) 01.06.2025 р.
3. Вихідні дані до проєкту (роботи) опис предметної області, технології розробки клієнтської та серверної частини, середовища розробки Visual Studio Code, Postman, MongoDB Compass. Персональний комп'ютер із процесором Apple M4 на базі операційної системи MacOS Sequoia version 15.5.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1) Визначення технологій розробки;
2) Проєктування та моделювання ПЗ;
3) Реалізація сервісу;
4) Випробування системи.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	СКРУПСЬКИЙ С.Ю.		
Нормоконтроль	ПОЛЬСЬКА О.В.		

7. Дата видачі завдання «14» квітня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Визначення технологій розробки	до 18.04.2025	
2	Проєктування та моделювання ПЗ	до 22.04.2025	
3	Реалізація сервісу	до 05.05.2025	
4	Адаптивний дизайн	до 12.05.2025	
5	Випробування системи	до 22.05.2025	
6	Оформлення пояснювальної записки	до 25.05.2025	
7	Проходження нормоконтролю	до 01.06.2025	
8	Перевірка на наявність академічного плагіату	до 03.06.2025	
9	Проходження рецензування	до 05.06.2025	

Студент(ка) ____________ (підпис) Іван КРАВЧЕНКО (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи) ____________ (підпис) Степан СКРУПСЬКИЙ (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 92 с., 6 табл., 23 рис., 1 дод., 31 джерел.

ВЕБІНТЕРФЕЙС, ВЕБМЕСЕНДЖЕР, РЕАЛЬНИЙ ЧАС, СОЦІАЛЬНА ІНТЕРАКЦІЯ, ХЕШУВАННЯ ПАРОЛІВ, JWT-АУТЕНТИФІКАЦІЯ, MONGODB, NODE.JS, REACT, SOCKET.IO, TAILWIND CSS

Об'єкт розробки: вебзастосунок FlexiChat — система миттєвого обміну повідомленнями між користувачами через вебінтерфейс.

Мета роботи: створення безпечного, адаптивного та функціонального вебмесенджера, який забезпечує обмін повідомленнями в реальному часі, збереження історії чатів, автентифікацію користувачів та персоналізацію інтерфейсу з підтримкою мобільних пристроїв.

У ході розробки виконано аналіз сучасних інструментів комунікації та архітектурних підходів до створення систем реального часу. Вимоги до власного рішення: простота, масштабованість, захист даних і мінімалістичний інтерфейс.

Для реалізації клієнтської частини використано React, Tailwind CSS і Zustand. Серверна частина побудована на Node.js з використанням Express для REST API та Socket.IO для реального часу. Збереження даних реалізовано на базі MongoDB, автентифікація — через JWT, а паролі хешуються за допомогою bcrypt. Система підтримує збереження теми інтерфейсу в localStorage, а також забезпечує адаптивний вигляд для мобільних пристроїв. Розроблений застосунок повністю працює у браузері без потреби в окремих клієнтських програмах.

Реалізована система є повноцінним прототипом вебмесенджера, що відповідає вимогам безпеки, продуктивності та зручності. Використані технології забезпечують гнучкість, масштабованість і придатність до подальшого розвитку.

ЗМІСТ

Скорочення та умовні познаки 7
Вступ 8
1 Визначення технологій розробки 9
1.1 Вибір архітектури системи 9
1.2 Вибір технологій для фронтенду та бекенду 16
1.3 Вибір бази даних 26
1.4 Висновки до розділу 1 30
2 Проєктування та моделювання ПЗ 31
2.1 Загальна структура сервісу 31
2.2 Моделювання сутностей БД 37
2.3 Структура клієнтської частини 40
2.4 Структура серверної частини 43
2.5 Структура інфраструктури 44
2.6 Загальний алгоритм роботи системи 46
2.7 Висновки до розділу 2 47
3 Реалізація сервісу 49
3.1 Реалізація системи реального часу 49
3.2 Система автентифікації користувачів 51
3.3 Компоненти користувацького інтерфейсу 52
3.4 Управління станом застосунку 58
3.5 Адаптивний дизайн 60
3.6 Висновки до розділу 3 61
4 Випробування системи 62
4.1 Сторінки застосунку 62
4.2 Функції застосунку 70
4.3 Безпека та захист даних 71
4.4 Висновки до розділу 4 73

Висновки 75
Перелік джерел посилання 77
Додаток А Лістинги програм 80

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

БД – База даних

СУБД – Система управління базами даних

AES – Advanced Encryption Standard, стандарт симетричного шифрування

API – Application Programming Interface, програмний інтерфейс прикладного рівня

BSON – Binary JSON, бінарний формат, який використовується MongoDB для зберігання даних

CDN – Content Delivery Network, мережа доставки контенту

CORS – Cross-Origin Resource Sharing, механізм спільного використання ресурсів між доменами

CRUD – Create, Read, Update, Delete, базові операції з даними

CSS – Cascading Style Sheets, каскадні таблиці стилів

DOM – Document Object Model, об'єктна модель документа

HTML – HyperText Markup Language, мова розмітки гіпертексту

HTTP – HyperText Transfer Protocol, протокол передавання гіпертексту

HTTPS – HTTP Secure, захищений протокол передавання гіпертексту

JWT – JSON Web Token, формат токена для безпечної передачі інформації

JS – JavaScript, мова програмування для веброзробки

JSON – JavaScript Object Notation, формат обміну даними

OAuth – Open Authorization, протокол авторизації

REST – Representational State Transfer, архітектурний стиль вебсервісів

UI – User Interface, користувацький інтерфейс

UX – User Experience, користувацький досвід

UUID – Universally Unique Identifier, універсальний унікальний ідентифікатор

ВСТУП

У сучасному цифровому суспільстві, де швидкість обміну інформацією має вирішальне значення, виникає потреба в надійних і зручних засобах комунікації. Інтернет-технології вже давно стали невіддільною частиною нашого повсякденного життя, а разом із ними зросла важливість інструментів, які забезпечують миттєвий обмін повідомленнями між людьми в будь-якій точці світу.

Особливої актуальності набувають вебмесенджери — системи, що дозволяють користувачам в режимі реального часу вести приватні та групові діалоги, ділитися медіафайлами та координувати дії. Такі системи не лише сприяють особистому спілкуванню, а й активно впроваджуються у бізнес, освіту, соціальні ініціативи та сферу обслуговування.

Ця дипломна робота присвячена розробці сучасного вебзастосунку для обміну повідомленнями, який реалізовано з використанням стеку MERN (MongoDB, Express.js, React.js, Node.js). Основна мета проєкту — створення швидкої, зручної та інтуїтивно зрозумілої системи обміну повідомленнями, яка задовольнить потреби як звичайних користувачів, так і організацій, що потребують ефективної внутрішньої комунікації.

У розробці основна увага приділяється:

- створенню зручного клієнтського інтерфейсу для миттєвого листування;
- забезпеченню надійного обміну даними через WebSocket-з’єднання;
- безпеці та конфіденційності користувачів;
- гнучкій архітектурі, здатній масштабуватись зі зростанням навантаження.

Проєкт включає глибокий аналіз існуючих месенджерів, формування функціональних вимог до системи, розробку програмної архітектури, реалізацію основного функціоналу та тестування. Особлива увага приділяється інтерактивному дизайну, адаптивності інтерфейсу, а також відповідності сучасним стандартам веброзробки.

1 ВИЗНАЧЕННЯ ТЕХНОЛОГІЙ РОЗРОБКИ

1.1 Вибір архітектури системи

Архітектура програмного забезпечення є одним із ключових етапів планування будь-якої інформаційної системи. Вона визначає основні структурні компоненти, способи їхньої взаємодії, а також впливає на масштабованість, ефективність, безпеку, надійність та простоту підтримки майбутнього програмного продукту. У контексті вебзастосунків, а особливо таких, які працюють у режимі реального часу — як, наприклад, месенджери, — архітектурне планування дозволяє врахувати як функціональні, так і нефункціональні вимоги ще до початку реалізації.

Правильно підібрана архітектура дозволяє забезпечити стійкість системи до навантажень, гнучкість у розширенні, ефективну обробку запитів та повідомлень, а також підвищити загальну продуктивність. Архітектурний підхід визначає, яким чином компоненти системи обмінюватимуться даними, як буде реалізована логіка взаємодії клієнтів і сервера, чи можлива паралельна обробка подій та наскільки легкою буде інтеграція з іншими системами чи сторонніми сервісами.

У сучасній практиці існує кілька основних типів архітектури, які найчастіше застосовуються при створенні вебдодатків:

- монолітна архітектура — уся система функціонує як єдине ціле, де всі компоненти об'єднані в одному програмному блоці [1];

- клієнт-серверна архітектура — класична модель розділення відповідальності між клієнтською частиною (інтерфейс) і сервером (логіка, база даних) [2];

- мікросервісна архітектура — система розділяється на незалежні сервіси, кожен з яких виконує окрему бізнес-функцію [3];

- Serverless-архітектура — використання хмарних обчислень, де окремі функції запускаються за запитом без управління інфраструктурою [4];

- архітектура на основі API (API-first) — це підхід до розробки, де API проєктується та документується ще до створення інтерфейсів чи бізнес-логіки [5];
- односторінкова архітектура (SPA) — це вебзастосунок, який динамічно переписує вміст однієї HTML-сторінки без повного перезавантаження сторінки [6].

Кожен із вищенаведених підходів має своє призначення та використовується в залежності від вимог конкретного проєкту. У випадку створення месенджера, який потребує роботи з повідомленнями в реальному часі, обробки великої кількості одночасних підключень та швидкого обміну даними, вибір архітектури повинен ґрунтуватися на цих критичних аспектах.

1.1.1 Монолітна архітектура

Монолітна архітектура є найбільш традиційним підходом до розробки програмних систем. Всі компоненти системи, включаючи користувацький інтерфейс, бізнес-логіку, серверну частину та базу даних, розташовані в єдиному кодовому репозиторії й взаємодіють безпосередньо між собою. Це означає, що всі функціональні елементи взаємопов’язані в межах одного процесу, що спрощує розробку, тестування та деплоймент на початкових етапах проекту. Монолітні системи можуть бути ефективними при малих і середніх проектах, де важлива швидкість розробки, оскільки усі компоненти є частинами одного цілого, і не потрібно вирішувати питання складної інтеграції між різними частинами. Це також означає, що монолітні системи зазвичай є більш дешевими на початкових етапах, оскільки не потрібно створювати і підтримувати інфраструктуру для взаємодії між численними сервісами [1].

Однак із часом, коли проект починає рости, моноліт може створювати певні труднощі. Оскільки всі компоненти тісно інтегровані, масштабування стає проблемою, і для того, щоб обробити більше запитів чи обсягів даних, доводиться масштабувати всю систему, що є не завжди ефективним. Також, коли система розростається, важко підтримувати та оновлювати її без ризику виникнення збоїв через зміни в одному з компонентів. У випадку помилки в одному з елементів, це може спричинити відмову всієї системи, що знижує надійність і стабільність роботи. Ось чому монолітну архітектуру зазвичай використовують для невеликих

проектів або на перших етапах розвитку продукту, коли потрібно швидко реалізувати базовий функціонал без великих витрат на інфраструктуру.

Монолітна архітектура є найбільш підходящою для невеликих або короткострокових проектів, які не потребують складних інтеграцій між компонентами або масштабування в майбутньому. Наприклад, для простих вебсайтів або MVP, де основна мета — швидко доставити продукт користувачам без додаткових витрат часу на створення складної інфраструктури або взаємодії між окремими частинами системи. З цією архітектурою легко почати роботу, і вона дозволяє зосередитись на розвитку основного функціоналу, що робить її хорошим вибором для ранніх етапів розробки.

1.1.2 Клієнт-серверна архітектура

Клієнт-серверна архітектура є однією з найбільш поширених моделей для вебзастосунків. Вона полягає в чіткому розподілі завдань між двома основними компонентами: клієнтом (фронтенд) та сервером (бекенд). Клієнт відповідає за відображення інформації та взаємодію з користувачем, а сервер обробляє запити клієнта та надає необхідні дані. Цей підхід є дуже зручним для веб-застосунків, оскільки дозволяє розділити функціональність між двома частинами і ефективно управляти кожною з них. Система взаємодіє через протоколи передачі даних, як-от HTTP чи WebSocket, що дозволяє реалізувати високопродуктивні і масштабовані рішення [2].

Завдяки чіткому розподілу між клієнтом і сервером, така архітектура дозволяє розробникам працювати над різними частинами системи окремо. Клієнт може бути створений з використанням різних технологій (JavaScript, HTML, CSS), тоді як серверна частина може бути розроблена з використанням серверних технологій, таких як Node.js, Django або Flask. Це дає змогу використовувати найкращі інструменти для кожної частини системи, що підвищує гнучкість і ефективність у розробці.

Однією з основних переваг клієнт-серверної архітектури є те, що вона дозволяє масштабувати систему, розділяючи навантаження між клієнтською та серверною частинами. Сервер може обробляти велику кількість запитів і

працювати з великими обсягами даних, а клієнт відповідно здійснює тільки необхідні запити, мінімізуючи навантаження на його систему. Також, якщо потрібно змінити або оновити серверну частину, це можна зробити без впливу на клієнтську частину, що підвищує гнучкість у розробці та тестуванні.

Проте, така архітектура може бути чутливою до проблем з мережею, оскільки всі запити від клієнта до сервера залежать від стабільного з'єднання. Крім того, у великих системах виникають складнощі із забезпеченням високої доступності та ефективної обробки великої кількості одночасних запитів. Тому клієнт-серверна архітектура є ідеальним рішенням для проектів, де важлива швидка взаємодія з користувачем, але також передбачаються високі вимоги до масштабованості та стабільності. Вона широко застосовується для месенджерів, чатів, електронної комерції та багатьох інших веб-застосунків, де необхідна чітка і розділена обробка даних та взаємодія з користувачем.

1.1.3 Мікросервісна архітектура

Мікросервісна архітектура є більш сучасним і складним підходом, який полягає у розбитті великої системи на численні невеликі сервіси, кожен з яких виконує окрему функцію. Кожен мікросервіс є автономною одиницею, що може бути розроблена, розгорнута, масштабована та оновлена незалежно від інших частин системи. Сервери для кожного мікросервісу можуть бути окремими, і кожен з них має свою базу даних, що дозволяє краще керувати даними та оптимізувати їх використання [3].

Мікросервісна архітектура відрізняється від монолітної тим, що дозволяє більше гнучкості при розвитку та масштабуванні системи. Кожен мікросервіс можна написати на різних мовах програмування та використовувати різні технології, що дає можливість більш ефективно вибирати технології, оптимізовані для конкретної задачі. Крім того, завдяки незалежності мікросервісів, кожен з них може бути масштабований окремо, що робить систему більш ефективною у разі високих навантажень.

Проте мікросервіси вимагають складної інфраструктури для їх взаємодії та управління. Комунікація між сервісами має відбуватися через API або інші

механізми, що може створювати затримки та додаткові навантаження на мережу. Окрім того, мікросервісна архітектура потребує значних зусиль для моніторингу та забезпечення злагодженої роботи всіх компонентів, оскільки відмови в одному мікросервісі можуть вплинути на загальну стабільність системи. Мікросервіси найбільш підходять для великих та складних проектів, де важливі масштабованість, незалежність частин системи та їх гнучкість. Вони часто використовуються в таких системах, як великі електронні магазини, месенджери, онлайн-платформи з великою кількістю функцій та користувачів.

1.1.4 Serverless архітектура

Serverless архітектура — це модель, де програмне забезпечення розгортається і працює без необхідності керувати фізичними або віртуальними серверами. Замість цього, ресурсами займаються хмарні провайдери (наприклад, AWS Lambda, Google Cloud Functions, Azure Functions), які автоматично масштабують інфраструктуру залежно від навантаження. У serverless архітектурі розробник фокусується на написанні коду, а не на управлінні серверами чи інфраструктурою. Система самостійно обирає кількість необхідних ресурсів для виконання завдань, що дозволяє ефективно обробляти запити з великою кількістю користувачів, не вимагаючи від розробника витрат на підтримку серверів або їх масштабування [4].

Ця архітектура особливо підходить для проектів з нестабільним навантаженням або для тих, хто хоче знизити витрати на інфраструктуру. Завдяки автоматичному масштабуванню можна обробляти багато запитів без зайвих витрат на ресурси в періоди низького навантаження. Система оплачуватиметься лише за фактичне використання ресурсів, що значно знижує витрати для малих та середніх проектів. Вона також дозволяє швидше реалізувати певні функціональності, оскільки зосереджує увагу на створенні окремих сервісів і мікрофункцій, що можна легко налаштувати через хмару.

Однак serverless архітектура має й обмеження. Вона не підходить для тривалих або складних операцій, оскільки кожен "функціональний виклик" обмежений по часу. Крім того, важко передбачити, як система поведеться в умовах високого або специфічного навантаження, оскільки система повністю залежить від

сторонніх постачальників хмарних послуг. Serverless підходить для систем, які мають мале або середнє навантаження, або для специфічних, швидких задач, таких як обробка подій, аналітика в реальному часі або API. Тому, для проектів, де потрібно високе масштабування, швидка реакція на зміну трафіку та економія на ресурсах, serverless архітектура може бути ідеальним рішенням.

Serverless архітектура буде особливо корисною для веб-додатків, які мають спорадичне навантаження, наприклад, для додатків з функціями обробки запитів користувачів, збору та обробки даних або інтеграцій із сторонніми сервісами.

1.1.5 Архітектура на основі API (API-first)

API-first архітектура є підходом, в якому розробка веб-застосунку починається з проектування інтерфейсів для взаємодії між різними компонентами системи (наприклад, між фронтендом і бекендом). Всі сервіси та компоненти збудовані таким чином, щоб вони могли взаємодіяти через чітко визначені API. Це дозволяє різним частинам системи бути незалежними одна від одної, та забезпечує гнучкість при масштабуванні та розвитку. API-first підхід дозволяє використовувати однакові API для різних клієнтів: мобільних додатків, веб-додатків, а також для інтеграцій з іншими сервісами [5].

Такий підхід підходить для складних систем, де необхідно забезпечити високу інтеграцію з іншими сервісами або підтримку декількох платформ одночасно. API-first дає можливість розробникам створювати інтерфейси для користувачів без необхідності турбуватися про технічні деталі обробки запитів. Це дозволяє зосередитись на тому, як дані будуть передаватися між різними частинами системи, а не на тому, як вони будуть оброблятися в кожному окремому компоненті. Крім того, це дозволяє організувати просте оновлення або заміну окремих компонентів системи, оскільки API не змінюється.

Основною перевагою API-first є те, що він дозволяє створювати масштабовані та модульні системи, що можуть бути легко інтегровані з іншими сервісами. Це забезпечує гнучкість при розробці та дозволяє підтримувати високий рівень взаємодії з іншими сервісами. Водночас, такі системи можуть потребувати додаткових зусиль для управління версіями API та координації між різними

компонентами, оскільки кожен елемент має чітко визначену точку входу для взаємодії.

API-first підхід стане оптимальним для проектів, де важлива інтеграція з іншими веб-сервісами чи платформами, а також для тих, де потрібно підтримувати взаємодію з різними клієнтами, такими як мобільні додатки, веб-сайти чи сторонні сервіси. Веб-застосунки, орієнтовані на швидке розширення функціоналу та інтеграцію, можуть скористатися API-first архітектурою для досягнення високої гнучкості та масштабованості.

1.1.6 Односторінкова архітектура (SPA)

SPA (Single Page Application) є архітектурою, що передбачає побудову веб-застосунків, де вся логіка виконання та взаємодії з користувачем розгортається на одному HTML-документі. Усі необхідні ресурси завантажуються за один раз, а для взаємодії з сервером використовуються асинхронні запити через AJAX або WebSockets, що дозволяє додатку змінювати контент без перезавантаження сторінки. Цей підхід дозволяє створювати плавні та швидкі інтерфейси для користувачів, оскільки всі дані підвантажуються в фоновому режимі і обробляються на клієнтській стороні [6].

Такий підхід особливо підходить для веб-додатків з високими вимогами до інтерфейсу користувача, таких як месенджери, чати, соціальні мережі, панелі управління або інші інтерактивні сервіси. SPA дозволяє створити користувацький інтерфейс, який працює швидко та ефективно, забезпечуючи безперервний досвід взаємодії з додатком. Всі зміни на сторінці виконуються миттєво, без потреби перезавантаження сторінки, що робить користування додатком більш інтуїтивно зрозумілим та комфортним.

Водночас, є і недоліки такої архітектури: вона потребує потужних фронтенд-технологій, таких як React, Angular або Vue.js, а також значного часу на оптимізацію додатка для мобільних пристроїв і різних браузерів. Крім того, для великої кількості одночасних користувачів можуть виникнути труднощі із збереженням ефективної роботи та низьким рівнем затримки при обміні даними з сервером.

1.2 Вибір технологій для фронтенду та бекенду

На етапі проєктування вебзастосунку, особливо такого типу, як месенджер, критично важливим є грамотний вибір інструментів для реалізації як клієнтської, так і серверної частини. Сучасний стек веброзробки включає в себе десятки фреймворків, бібліотек та СУБД, і розуміння їх призначення, переваг і сфери використання є обов'язковим кроком у побудові ефективного програмного продукту.Клієнтська частина відповідає за взаємодію користувача з системою, тому має бути не лише функціональною, а й зручною, швидкою та адаптивною. Для цього найчастіше використовуються JavaScript-фреймворки й бібліотеки. Серед найпопулярніших: React, Vue.js, Angular [7].

Серверна частина відповідає за логіку, авторизацію, роботу з базами даних та обробку запитів. Серед основних технологій для бекенду: Node.js, Django, Laravel.

Для тих випадків, коли один і той самий стек застосовується для клієнтської та серверної частин, говоримо про fullstack-інструменти. Серед них: Next.js та Meteor.js.

1.2.1 React.js

React.js — це одна з найпопулярніших бібліотек JavaScript для розробки інтерфейсів користувача, створена компанією Meta. Вона є ефективним інструментом для побудови динамічних та масштабованих односторінкових додатків (SPA), де важливо забезпечити інтерактивність та реактивну зміну контенту без перезавантаження сторінки. В основі React лежить концепція компонентного підходу, що дозволяє розділяти інтерфейс на логічно ізольовані, повторно використовувані частини. Це не лише покращує читабельність і підтримку коду, а й дозволяє масштабувати проєкт без порушення вже існуючого функціоналу [8].

Однією з ключових особливостей React є використання віртуального DOM, який значно пришвидшує оновлення користувацького інтерфейсу, мінімізуючи кількість безпосередніх маніпуляцій з реальним DOM браузера. Це критично

важливо для застосунків, що мають велику кількість взаємодій з інтерфейсом, таких як месенджери, де кожне нове повідомлення, статус користувача або сповіщення має миттєво з'являтися на екрані. React також має розвинену екосистему — наприклад, бібліотеки react-router для маршрутизації, redux або zustand для керування станом, що значно спрощує реалізацію складної логіки клієнтської частини.

Ще однією перевагою є можливість використання JSX — розширення синтаксису JavaScript, що дозволяє писати розмітку безпосередньо в коді, роблячи структуру компонентів наочнішою. React також добре інтегрується з TypeScript, що надає підтримку типізації для складних проєктів і допомагає уникати частих помилок під час розробки. Завдяки цим властивостям, React обирається як основна технологія для розробки інтерфейсу сучасних вебзастосунків, де ключову роль відіграє динамічність, стабільність, швидкість рендерингу та адаптивність інтерфейсу до різних пристроїв.

1.2.2 Vue.js

Vue.js — це прогресивний JavaScript-фреймворк для створення інтерфейсів користувача, який вирізняється легкістю у вивченні, гнучкістю та високою продуктивністю. Його було створено Еваном Ю, колишнім працівником Google, з метою забезпечити просту, але потужну альтернативу таким громіздким фреймворкам, як Angular. На відміну від багатьох інших рішень, Vue дозволяє поступово впроваджувати свій функціонал, що робить його привабливим як для новачків, так і для досвідчених розробників [9].

Архітектура Vue побудована на основі шаблонів, реактивності даних і компонентного підходу. Компоненти — це окремі логічні блоки інтерфейсу, які можна повторно використовувати в межах проєкту. Такий підхід дозволяє підтримувати чисту, модульну структуру коду, що особливо цінно в проєктах із великою кількістю візуальних елементів і взаємодії, як-от вебмесенджери. Vue забезпечує автоматичну реактивність даних, тобто будь-які зміни у стані інтерфейсу миттєво відображаються на екрані без потреби вручну оновлювати DOM.

Окремою перевагою Vue є його декларативна природа: описуючи, що має відображатися в інтерфейсі залежно від стану даних, розробник звільняється від необхідності прописувати як саме це відбувається. Це суттєво спрощує логіку взаємодії з DOM і дає змогу швидше створювати динамічні інтерфейси, особливо в режимі реального часу — що критично важливо для месенджерів. Vue також добре інтегрується з WebSocket-рішеннями, даючи змогу організувати двосторонню комунікацію між клієнтами та сервером.

Однією з визначальних особливостей Vue є його гнучкість — він не нав’язує єдину парадигму. Розробник може писати компоненти як у стилі класичних HTML-шаблонів із вбудованим JavaScript, так і використовувати повноцінний JSX або TypeScript. Це дозволяє адаптувати проєкт під команду з різним рівнем досвіду, обираючи зручний стиль розробки. Крім того, Vue має багатий екосистемний простір — бібліотеки для маршрутизації (Vue Router), керування станом (Pinia, раніше Vuex), інтеграції з API та побудови SPA-додатків.

У багатьох випадках Vue застосовують у проєктах, де важлива швидкість розробки, адаптивність інтерфейсу та чіткий поділ відповідальностей. Він підходить як для невеликих інтерактивних модулів, вбудованих у вже наявні сайти, так і для повноцінних односторінкових застосунків. Для вебмесенджера це може бути особливо корисно, коли інтерфейс має бути максимально швидким, чуйним і здатним відображати зміни у повідомленнях, списках контактів чи стані користувачів без перезавантаження сторінки.

Завдяки своїй відкритості, великій спільноті, прозорій документації та високій продуктивності Vue.js залишається одним із провідних фреймворків для фронтенду, який активно використовується в компаніях по всьому світу. Хоча в цьому проєкті він не був обраний основною технологією, його популярність і потенціал роблять його вагомим кандидатом для будь-якого сучасного вебзастосунку.

1.2.3 Angular

Angular — це фронтенд-фреймворк із відкритим кодом, розроблений і підтримуваний компанією Google. Його основне призначення — створення

масштабованих, продуктивних і структурованих вебзастосунків. Angular є повноцінним фреймворком, який надає розробникам усе необхідне для побудови сучасного клієнтського застосунку без потреби у сторонніх бібліотеках: це включає засоби маршрутизації, керування станом, валідації форм, модульності, HTTP-запитів, DI-контейнер тощо [10].

Один із фундаментальних принципів Angular — використання TypeScript як основної мови розробки. Це дає змогу створювати статично типізований код, що покращує читаємість, передбачуваність та полегшує рефакторинг великих проєктів. У контексті командної розробки Angular ідеально підходить для корпоративного програмного забезпечення, де важливо підтримувати високу структурованість коду, сувору архітектуру та контроль якості.

Завдяки суворій модульності, Angular дозволяє будувати застосунки, які легко масштабуються. Кожен функціональний блок можна виділити в окремий модуль, що робить застосунок більш організованим. Angular активно використовує систему декораторів і метаданих, що дозволяє зрозуміло описувати компоненти, сервіси, маршрути та інші сутності.

Особливістю Angular є двостороннє зв’язування даних (two-way data binding), що дозволяє автоматично синхронізувати модель і вигляд. Це зручно для інтерактивних застосунків, де стан інтерфейсу змінюється в залежності від дій користувача — наприклад, у чатах або месенджерах, де важлива миттєва реакція на вхідні повідомлення. Однак ця особливість може також впливати на продуктивність у складних інтерфейсах, тому Angular має механізми оптимізації — наприклад, зонування та зміни детекції.

У випадках, коли проєкт вимагає суворої архітектури, високої надійності, підтримки масштабних інтерфейсів і чіткої структури, Angular може бути оптимальним вибором. Він особливо корисний у великих організаціях, де процес розробки розділений між багатьма командами і необхідна узгодженість між різними частинами застосунку. Його набір інструментів дозволяє забезпечити уніфікованість, тестованість і передбачуваність навіть у складних вебзастосунках.

1.2.4 Node.js

Node.js — це середовище виконання JavaScript на сервері, побудоване на рушії V8 від Google Chrome, яке дозволяє розробникам писати серверну логіку використовуючи ту саму мову програмування, що й для клієнтської частини. Це відкриває можливість створення повноцінних застосунків із єдиним технологічним стеком, що значно знижує поріг входу та полегшує координацію між фронтендом і бекендом. Node.js не є фреймворком у класичному розумінні, а радше середовищем, яке надає доступ до файлової системи, мережі та іншим низькорівневим API, що є необхідними для створення серверної логіки вебдодатків [11].

Головною технічною особливістю Node.js є його неблокуюча, подієво-орієнтована модель вводу/виводу, яка дозволяє обробляти тисячі одночасних підключень без створення окремих потоків для кожного запиту. Завдяки цьому Node.js особливо добре підходить для розробки застосунків реального часу, таких як чат-додатки або месенджери, де критично важливо забезпечити миттєву доставку повідомлень між клієнтами та обробку великої кількості з’єднань одночасно. Асинхронна природа Node.js також дозволяє ефективно взаємодіяти з базами даних, API та іншими зовнішніми сервісами без блокування процесу виконання програми.

Зазвичай розробка на Node.js базується на використанні додаткових фреймворків, таких як Express.js, які спрощують створення RESTful API, маршрутизацію та обробку запитів. Node.js підтримує велику кількість npm-пакетів, що дозволяє швидко розширювати функціональність додатку — від автентифікації до обробки сокет-з’єднань через Socket.IO. Такий підхід дозволяє зосередитися на логіці застосунку, а не на низькорівневій реалізації протоколів і структури запитів.

Node.js також відзначається своєю продуктивністю та масштабованістю, що робить його ідеальним вибором для стартапів, MVP-проєктів, а також для вже існуючих великих систем із великою кількістю активних користувачів. Саме тому ця технологія стала невід’ємною частиною стеку MERN і широко

використовується в проєктах, де потрібна висока швидкість розробки, масштабованість та єдина мова на всіх рівнях вебдодатку.

Попри свою складність у порівнянні з фреймворками на кшталт Vue або React, Angular залишається надійним і стабільним вибором для реалізації повнофункціональних вебдодатків, де важливі чітка структура, контроль і довготривала підтримка.

1.2.5 Django

Django — це високорівневий вебфреймворк для розробки серверної частини застосунків, написаний мовою програмування Python. Його головна мета — забезпечити швидку, безпечну та масштабовану розробку вебдодатків із мінімальними зусиллями. Django був створений з акцентом на принципі DRY (Don't Repeat Yourself) і філософією «батарейки в комплекті», що означає наявність великого набору вбудованих інструментів, які значно полегшують процес розробки [12].

Фреймворк надає повноцінну структуру для реалізації CRUD-операцій, маршрутизації, аутентифікації, адміністрування, форм, ORM (об'єктно-реляційного відображення), кешування, захисту від CSRF і XSS, обробки помилок, роботи з шаблонами тощо. Завдяки цьому Django дозволяє сконцентруватися на бізнес-логіці застосунку, зменшуючи потребу у використанні сторонніх бібліотек або самописних рішень.

Однією з найсильніших сторін Django є його ORM-система, яка забезпечує взаємодію з базами даних через Python-класи замість написання SQL-запитів вручну. Це дозволяє ефективно працювати з базами даних і підтримувати їх незалежність від конкретного типу СУБД. Django також має вбудовану панель адміністратора, яка генерується автоматично на основі моделей, що суттєво прискорює розробку внутрішніх інтерфейсів керування вмістом чи користувачами.

У контексті архітектури, Django дотримується принципу MTV (Model–Template–View), який концептуально близький до MVC, але зі своїми особливостями: Model відповідає за структуру даних, Template — за відображення,

а View — за логіку обробки запитів. Така структура добре масштабована, зрозуміла та ефективна для підтримки середніх і великих застосунків.

Django активно застосовується в розробці новинних порталів, соціальних мереж, панелей керування, інтернет-магазинів, REST API для SPA-застосунків і мобільних клієнтів. Він особливо добре підходить для проєктів, які потребують швидкого запуску MVP, надійної структури безпеки (все вбудовано з коробки), стабільного розширення і добре документованої бази знань.

Цей фреймворк варто обирати тоді, коли ключовим є швидкий запуск, потреба в розвиненій системі авторизації та управління користувачами, гнучка робота з базами даних, інтеграція з адміністративною панеллю або просто бажання скористатися перевагами Python. Django також часто використовують у поєднанні з фронтенд-фреймворками як бекенд API-сервер, особливо в архітектурах на основі REST або GraphQL.

Таким чином, Django — це універсальний інструмент, який забезпечує швидкість розробки, високу безпеку та масштабованість для серверної частини вебзастосунків будь-якого масштабу.

1.2.6 Laravel

Laravel — це потужний фреймворк для розробки вебзастосунків на мові програмування PHP, що став стандартом де-факто для побудови складних, масштабованих серверних систем у цьому середовищі. Його філософія базується на простоті, елегантності та читабельності коду, а також дотриманні сучасних архітектурних принципів, таких як MVC (Model-View-Controller), що дозволяє розділяти логіку бізнесу, інтерфейс та управління даними в межах однієї структури [13].

Laravel забезпечує розробника широким спектром вбудованих можливостей: маршрутизація, міграції бази даних, шаблонізація з Blade, система черг, підтримка WebSocket через Laravel Echo, робота з кешуванням, робота з файлами, реалізація авторизації та аутентифікації, підтримка RESTful API та багато іншого. Одним із найсильніших аспектів Laravel є його власний ORM — Eloquent, який дозволяє

працювати з базами даних за допомогою об'єктно-орієнтованого підходу, значно спрощуючи роботу з SQL-запитами.

Завдяки чіткій структурі, Laravel дозволяє ефективно масштабувати застосунок — як у межах монолітної архітектури, так і з поступовим розбиттям на окремі сервіси або API. Також він ідеально підходить для створення серверної частини сучасних односторінкових додатків, мобільних застосунків або гібридних рішень, забезпечуючи високу продуктивність завдяки інструментам на кшталт Laravel Sanctum або Laravel Passport, що відповідають за токен-базовану автентифікацію.

Laravel має винятково активну спільноту та екосистему з безліччю додаткових пакетів, які спрощують інтеграцію популярних функцій — від підключення платіжних систем до реалізації багатомовності чи побудови адміністративних панелей. Крім того, Laravel пропонує інструменти для CI/CD (наприклад, Laravel Envoyer), для розгортання в хмарі (Laravel Vapor), а також фреймворк для побудови інтерфейсів у реальному часі — Laravel Livewire.

Laravel чудово підходить для проєктів, де ключовими є стабільність, надійність, гнучкість і потреба в довгостроковій підтримці. Його часто обирають у корпоративному середовищі, у розробці SaaS-продуктів, панелей керування, CRM-систем, e-commerce рішень і порталів, де важливо підтримувати чітку логіку, багатоетапну авторизацію користувачів та модульність системи.

Цей фреймворк доцільно обирати тоді, коли застосунок потребує високого рівня кастомізації, підтримки складної бізнес-логіки, а також коли команда володіє досвідом розробки на PHP або проект вже частково реалізований у цьому стеку. Laravel також підходить для проєктів з довгим життєвим циклом, оскільки його архітектура добре витримує ріст складності, а оновлення фреймворку ретельно документовані й підтримуються стабільно.

1.2.7 Next.js

Next.js — це сучасний фреймворк для розробки React-застосунків, створений командою Vercel, який дозволяє будувати високопродуктивні, SEO-оптимізовані вебдодатки з підтримкою серверного рендерингу, генерації статичних сторінок та

гібридних підходів. На відміну від звичайного React, який є лише бібліотекою для побудови інтерфейсів користувача, Next.js забезпечує повноцінну інфраструктуру для створення повноцінного вебзастосунку — з маршрутизацією, API-ендпоінтами, системою компонування сторінок та обробкою SSR/SSG (Server-Side Rendering / Static Site Generation) [14].

Однією з найважливіших переваг Next.js є автоматичне рендерування сторінок на сервері або попередня генерація, що критично важливо для швидкого завантаження сторінок і високого рівня оптимізації для пошукових систем. Завдяки цьому Next.js часто використовують для створення маркетингових сайтів, блогів, документацій, динамічних порталів і навіть складних внутрішніх систем, де потрібно поєднати продуктивність, інтерактивність та доступність.

Next.js забезпечує систему маршрутизації «на основі файлів», де структура URL відповідає розміщенню файлів у директорії /pages, що значно спрощує навігацію й розробку. Крім того, є підтримка динамічних маршрутів, catch-all роутів, мідлварів, налаштування кешування і навіть розгортання API-ендпоінтів прямо всередині проєкту. Це дозволяє будувати повноцінні fullstack-застосунки без потреби створювати окремий сервер на Node.js чи інший бекенд.

Особливо важливою є можливість гнучко обирати підхід до рендерингу сторінки: можна рендерити її на сервері при кожному запиті, попередньо генерувати під час збірки або навіть комбінувати ці варіанти в одному застосунку. Це дозволяє точно налаштувати баланс між продуктивністю і свіжістю даних. Також підтримується ISR (Incremental Static Regeneration), що дозволяє оновлювати статично згенеровані сторінки вже після деплою.

У поєднанні з сучасним інструментарієм, таким як Tailwind CSS, TypeScript, SWR (для клієнтської роботи з даними), React Query або Zustand (для стану), Next.js дозволяє побудувати дійсно масштабовану та надійну клієнтську частину. Крім того, інтеграція з хмарним сервісом Vercel забезпечує миттєве розгортання, preview-середовища й автоматичне масштабування, що дуже зручно в командній розробці.

Next.js особливо доцільний для проєктів, де потрібна висока швидкість рендерингу, підтримка SEO, комплексні UI-компоненти та швидке масштабування. Його часто використовують у стартапах, SaaS-платформах, маркетингових сайтах, dashboard-інтерфейсах, e-commerce проєктах і в усіх випадках, коли потрібно об'єднати клієнтський та серверний функціонал під одним дахом.

Цей фреймворк — не просто оболонка навколо React, а повноцінна інженерна платформа, яка підходить як для малих MVP-прототипів, так і для великих продакшн-проєктів з високими вимогами до продуктивності, UX та масштабованості.

1.2.8 Meteor.js

Meteor.js — це повноцінна платформа для розробки сучасних JavaScript-застосунків, яка дозволяє будувати як клієнтську, так і серверну частину з використанням однієї мови — JavaScript. Вперше представлений у 2012 році, Meteor став популярним завдяки своїй простоті, автоматичній синхронізації даних між клієнтом і сервером (через систему pub/sub), а також модульній архітектурі, яка дозволяє швидко створювати інтерактивні вебдодатки без потреби у великій кількості конфігурацій [15].

Meteor вирізняється з-поміж інших фреймворків тим, що пропонує повний «стек із коробки». У типовому проєкті на Meteor є вже все необхідне для старту: сервер на Node.js, інтеграція з MongoDB, система маршрутизації, реактивний клієнт, автоматична побудова та деплой. Такий підхід знижує поріг входу та дозволяє розробнику зосередитися саме на функціональності, а не на інфраструктурі. Особливо це корисно для невеликих команд або індивідуальних проєктів.

Однією з ключових концепцій Meteor є реактивність — здатність інтерфейсу миттєво відображати будь-які зміни в даних без необхідності ручного оновлення сторінки або використання запитів до API. Це реалізовано через механізм публікацій та підписок (pub/sub), який дозволяє передавати лише ті дані, які дійсно потрібні клієнту. Завдяки цьому Meteor підходить для створення чатів, соціальних

мереж, реального часу систем (наприклад, трекінг), а також інструментів спільної роботи.

І хоча Meteor довгий час асоціювався з використанням Blaze як вбудованого шаблонізатора, сучасні версії повністю підтримують React, Vue, Svelte та інші сучасні бібліотеки інтерфейсу. Це дозволяє використовувати Meteor як бекенд-платформу зі зручним реактивним API у поєднанні з будь-якою сучасною клієнтською технологією.

Meteor також має власний пакетний менеджер — Atmosphere, а також можливість інтеграції з NPM. Це дає доступ як до офіційних специфічних рішень під Meteor, так і до величезної екосистеми загальних JavaScript-бібліотек. Водночас система пакунків у Meteor дозволяє ділити застосунок на модулі, що покращує масштабованість проєкту.

З точки зору розгортання, Meteor пропонує як власне PaaS-рішення (Galaxy), так і можливість деплою на будь-який сервер, що підтримує Node.js.

Meteor буде доцільним вибором у випадках, коли потрібен швидкий результат, тісна інтеграція між клієнтом і сервером, а також коли важлива реактивність і робота в реальному часі. Проте варто враховувати, що Meteor не завжди найкраще підходить для високонавантажених мікросервісних архітектур чи розподілених систем, де потрібна чітка межа між фронтендом і бекендом. Він чудово себе зарекомендував у створенні MVP, чатів, панелей керування, live-додатків, але менш придатний для систем, де потрібна масштабована логіка на бекенді з різними мовами програмування чи складною архітектурою.

1.3 Вибір бази даних

База даних (БД) — це організований набір даних, який зберігається та керується системою керування базами даних (СУБД). База даних призначена для зберігання, обробки, пошуку та маніпуляції даними. Використання БД забезпечує

ефективне зберігання великих обсягів інформації та доступ до неї з мінімальними затратами часу.

Система керування базами даних (СУБД) — це програмне забезпечення, яке дозволяє створювати, управляти та взаємодіяти з базами даних. СУБД займається організацією, зберіганням і захистом даних, а також надає інтерфейс для доступу до цих даних через запити. СУБД можуть бути різних типів в залежності від архітектури бази даних, яку вони підтримують.

Існує кілька основних типів баз даних, серед яких найбільш поширені реляційні та нереляційні бази даних. Кожен з цих типів має свої особливості та застосування в різних ситуаціях.

Реляційні бази даних (RDBMS, від англ. Relational Database Management System) — це бази даних, у яких інформація зберігається у вигляді таблиць, що містять рядки і стовпці. Кожна таблиця має певний набір стовпців, де кожен стовпець має визначений тип даних (число, текст, дата тощо), а кожен рядок містить конкретну інформацію.

Реляційні СУБД забезпечують доступ до даних за допомогою мови запитів SQL (Structured Query Language). SQL дозволяє виконувати різноманітні операції з даними: вибірку, оновлення, вставку, видалення та інші маніпуляції з даними в таблицях. Реляційні бази даних підтримують принцип нормалізації, що дозволяє уникати дублювання даних і забезпечує консистентність бази.

Реляційні бази даних застосовуються в ситуаціях, де необхідна сувора структура даних і зв’язки між ними. Вони ідеально підходять для транзакційних систем, де важлива цілісність та безпека даних, а також для складних запитів, які потребують багатошарового аналізу. Приклади популярних реляційних баз даних: MySQL, PostgreSQL [16].

Нереляційні бази даних (NoSQL, від англ. Not Only SQL) — це бази даних, які не використовують стандартну реляційну модель таблиць для зберігання даних. Замість цього вони можуть зберігати дані у вигляді документів, графів, пар "ключ-значення" або стовпців. Бази даних NoSQL розроблені для роботи з великими

обсягами даних, які не можна чітко структурувати в таблицях або коли структура даних постійно змінюється [17]. Основні типи NoSQL баз даних:

- документно-орієнтовані — дані зберігаються у вигляді документів (наприклад, JSON або BSON). Кожен документ може містити різні поля, що дозволяє легко зберігати структуровані або напівструктуровані дані;
- ключ-значення — зберігаються пари ключ-значення, де кожен ключ є унікальним і асоційований з певним значенням. Цей тип БД підходить для простих додатків, де необхідно швидко отримувати значення за ключем.

Нереляційні бази даних зазвичай використовуються в ситуаціях, де потрібно працювати з великими даними, швидко обробляти запити та мати гнучкість у структурі даних. Вони також є гарним вибором для проектів, де дані динамічно змінюються, і їх структура не може бути чітко визначена заздалегідь. Прикладами популярних нереляційних баз даних є MongoDB та Redis [18].

Для вибору бази даних для конкретного проєкту важливо враховувати кілька факторів, таких як тип даних, масштабованість, швидкість доступу до даних, необхідність у складних запитах та транзакціях. Якщо ваш проєкт потребує високої консистентності даних і підтримки складних запитів з багатьма зв’язками між даними, тоді найкращим вибором буде реляційна база даних. Якщо ж ваш проект орієнтований на високу швидкість роботи з великими обсягами даних або гнучкість структури, то нереляційні бази даних можуть бути більш підходящими.

1.3.1 MySQL

MySQL — це система управління базами даних, яка належить до реляційного типу. Її основна особливість полягає в тому, що дані організовуються у вигляді таблиць з чітко визначеними зв’язками між ними, що дає змогу ефективно працювати з великою кількістю взаємопов’язаних сутностей. MySQL забезпечує чітку структуру збереження інформації, що дозволяє запобігати дублюванню даних, підтримувати їхню цілісність та реалізовувати складну бізнес-логіку вже на рівні бази. Вона підтримує транзакції, що важливо при розробці систем, де критичною є коректність кожної дії (наприклад, зміна балансу чи збереження повідомлення в месенджері). MySQL добре підходить для проєктів, де очікується

робота з чітко структурованими, однаковими за типом записами, наприклад: користувачі, повідомлення, налаштування профілю. Її інструментарій дозволяє здійснювати складні запити, об'єднувати декілька таблиць, фільтрувати дані, будувати статистичні вибірки тощо. Такий підхід особливо ефективний для внутрішньої аналітики або адміністративного інтерфейсу, де важливо мати гнучкий доступ до взаємозалежної інформації [19].

1.3.2 PostgreSQL

PostgreSQL — це розвинена реляційна система управління базами даних, яка вирізняється гнучкістю, високою відповідністю стандарту SQL та підтримкою розширених можливостей для роботи з даними. Основною особливістю PostgreSQL є орієнтація не лише на класичну табличну модель, а й на розширення типів даних, підтримку об'єктно-реляційної моделі та можливість створення власних функцій, агрегатів, типів і навіть операторів. У практичному використанні це означає, що система дозволяє розробникам адаптувати базу даних під специфічні задачі додатка без необхідності змін у логіці застосунку. Наприклад, в месенджерах можна реалізувати збереження геоданих (місцезнаходження під час відправки повідомлення) або багатовимірних масивів, якщо такі потрібні — без обмежень типових СУБД [20].

1.3.3 MongoDB

MongoDB — це документно-орієнтована нереляційна база даних, яка зберігає дані у форматі BSON (розширення JSON), що дозволяє працювати з гнучкими, неструктурованими даними. Цей підхід особливо зручний для вебзастосунків, де структура даних може змінюватися залежно від потреб користувача або функціоналу. MongoDB забезпечує масштабованість як вертикальну, так і горизонтальну, що дозволяє ефективно обробляти великі обсяги інформації та динамічні зміни структури. Вона підтримує потужні механізми запитів, індексації, агрегації та реплікації. У контексті створення месенджерів MongoDB є ефективним рішенням для збереження повідомлень, даних користувачів, історії чатів, а також налаштувань, завдяки своїй гнучкості, простоті інтеграції з Node.js/Express та

високій продуктивності при роботі з великими обсягами нестандартизованих даних [21].

1.3.4 Redis

Redis — це високошвидкісна система збереження даних типу ключ-значення, що працює в оперативній пам’яті. Вона належить до класу NoSQL баз даних і найчастіше використовується як кеш або брокер повідомлень. На відміну від класичних реляційних СУБД, Redis забезпечує майже миттєвий доступ до даних завдяки збереженню всієї інформації у RAM, що робить її ідеальною для систем, де критичною є швидкодія — зокрема для месенджерів у реальному часі. Redis підтримує різні структури даних: рядки, списки, множини, хеші, потоки та впорядковані множини, що дозволяє ефективно реалізовувати черги повідомлень, статуси користувачів, кеш сесій тощо. Крім того, Redis має можливість зберігати дані на диск для забезпечення надійності, а також підтримує реплікацію, кластери та механізми високої доступності (Sentinel). У вебмесенджерах Redis використовується для тимчасового зберігання повідомлень, реалізації онлайн-статусів, систем pub/sub та зменшення навантаження на основну базу даних [22].

1.4 Висновки до розділу 1

У результаті проведеного аналізу сучасних архітектурних підходів і технологій, найбільш доцільним рішенням для реалізації вебзастосунку месенджера стало впровадження клієнт-серверної архітектури, що чітко розмежовує відповідальність між фронтендом і бекендом, забезпечуючи гнучкість, масштабованість і прозорість у взаємодії між частинами системи. Такий підхід дозволяє ефективно організувати комунікацію через API, оптимізувати навантаження та підтримувати окремий розвиток клієнтської й серверної логіки без взаємних залежностей, що особливо важливо для проєктів, орієнтованих на зростання та інтеграцію додаткового функціоналу у майбутньому.

Для реалізації клієнтської частини було обрано React.js, як сучасний компонентний фреймворк, що дозволяє створювати динамічні та реактивні інтерфейси. React забезпечує гнучку архітектуру компонування, високу продуктивність завдяки віртуальному DOM, та активну екосистему бібліотек, що полегшує інтеграцію функцій чатів, роутінгу, авторизації, а також оптимізацію рендерингу в реальному часі.

На серверній стороні проєкт реалізовано з використанням Node.js у зв'язці з Express.js. Цей вибір зумовлений необхідністю забезпечення асинхронної обробки запитів, низької латентності та зручності у створенні REST API або сокет-з'єднань. Завдяки Node.js можливе одночасне обслуговування великої кількості підключень, що є критично важливим для чат-застосунків, які працюють у режимі реального часу.

Як база даних використовується MongoDB, що належить до класу документно-орієнтованих нереляційних баз. Її структура на основі JSON-документів ідеально підходить для збереження гнучких і варіативних об'єктів, таких як повідомлення, інформація про користувачів, історії чатів тощо. MongoDB добре масштабується горизонтально та підтримує механізми реплікації, що забезпечує високу доступність даних та відмовостійкість.

2 ПРОЄКТУВАННЯ ТА МОДЕЛЮВАННЯ ПЗ

2.1 Загальна структура сервісу

2.1.1 Архітектурний огляд

Розроблюваний вебмесенджер побудований за клієнт-серверною архітектурною моделлю, яка складається з таких основних компонентів: frontend, backend, база даних, а також вебсокет-сервер для обміну повідомленнями в реальному часі. Архітектура реалізована відповідно до принципів модульності,

масштабованості, розділення відповідальностей та зручності супроводу. Загальну схему сервісу подано на рисунку 2.1 [23].

Frontend відповідає за взаємодію з користувачем через вебінтерфейс. Клієнтська частина реалізована за допомогою:

- React — бібліотека для створення інтерфейсу користувача;
- TypeScript — забезпечує статичну типізацію та полегшує розробку великих додатків;
- Tailwind CSS — використовує для побудови адаптивного дизайну;
- React Router — забезпечує маршрутизацію;
- Axios — для HTTP-запитів до REST API;
- Socket.IO Client — для встановлення постійного з’єднання з сервером WebSocket.

Клієнт взаємодіє з сервером через:

- REST API — для авторизації, реєстрації, отримання списку чатів, історії повідомлень;
- WebSocket (через Socket.IO) — для миттєвого надсилання й отримання повідомлень у реальному часі.

Backend — реалізований на базі Node.js з використанням Express.js як основного фреймворку для побудови API [24]. Основні функції:

- реєстрація та автентифікація користувачів із використанням JWT;
- обробка REST API-запитів (реєстрація, вхід, отримання повідомлень, створення чатів);
- підтримка WebSocket-з'єднання через Socket.IO — забезпечення надсилання повідомлень у реальному часі;
- зберігання даних у базі даних;
- хешування паролів за допомогою bcrypt;
- міжкористувацька комунікація.

База даних — використовується нереляційна СУБД MongoDB. Структура даних включає наступні колекції:

- користувачів (Users);

- повідомлень (Messages).

WebSocket-сервер на базі Socket.IO інтегровано з бекендом і відповідає за:

- обробку підключень користувачів;
- трансляцію нових повідомлень учасникам чату;
- відображення статусу "в мережі".

Інфраструктура розгортання — на етапі розробки застосунок розгортається локально через npm з можливістю подальшого розгортання на хмарну платформу. Використання .env файлів дозволяє зручно керувати конфігурацією токенів, URI БД та секретами. Особливості архітектури:

- Вебсокет-підключення постійне, базується на подіях;
- REST API реалізує всі допоміжні функції, не пов’язані з реальним часом;
- JWT використовується як у HTTP-запитах, так і передається через Socket.IO-підключення для авторизації.

Загальна структурна схема системи наведена на рисунку 2.1.

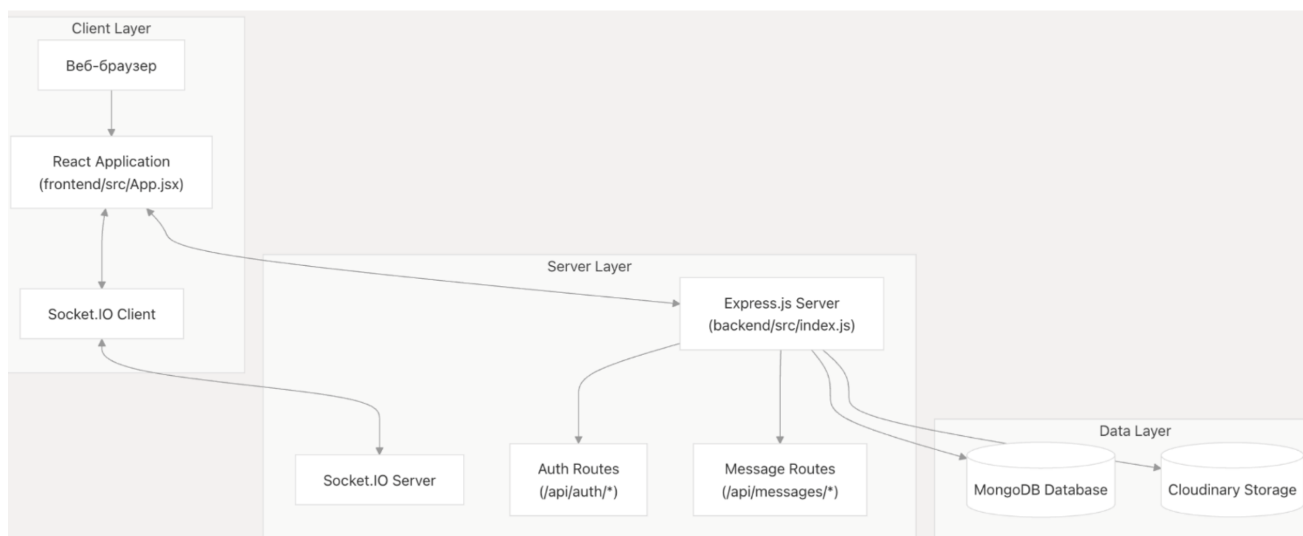


Рисунок 2.1 - Загальна структурна схема системи

2.1.2 Вимоги до клієнтського середовища

Вимоги, наведені на таблиці 2.1, допомагають забезпечити стабільну роботу застосунку на всіх етапах: від розробки до продакшен-розгортання.

Таблиця 2.1 – Вимоги до клієнтського середовища

Компонент	Мінімальні вимоги	Рекомендовані вимоги
Операційна система	Windows 10, macOS 10.14, Linux (Ubuntu 20.04)	Windows 11, macOS 12+, остання версія Ubuntu
Браузер	Chrome 90, Firefox 88, Edge 90, Safari 13	Остання стабільна версія Chrome або Firefox
Процесор	2 ядра, 1.5 ГГц	4 ядра, 2.5 ГГц і вище
Оперативна пам’ять	2 ГБ	4 ГБ і більше
Розширення екрану	375×667 px	1920×1080 px або більше
Підключення до інтернету	1 Мбіт/с	10 Мбіт/с або вище

2.1.3 Вимоги до хостингу та мережі

Для забезпечення стабільної роботи вебзастосунку FlexiChat необхідно дотримуватись певних вимог до інфраструктури розміщення та мережевого середовища. Враховуючи архітектуру проєкту, яка передбачає клієнт-серверну модель із використанням WebSocket-з’єднання, серверної логіки на Node.js і бази даних MongoDB, на таблицях 2.2 та 2.3 наведено основні технічні вимоги.

Таблиця 2.2 – Вимоги до хостингу

Параметр	Мінімальні вимоги	Рекомендовані вимоги
Операційна система	Linux (Ubuntu 20.04)	Linux (Ubuntu Server 22.04 або вище)
Дисковий простір	2 ГБ	5+ ГБ
Процесор	1 ядро	2 ядра і більше
Оперативна пам’ять	512 МБ	1–2 ГБ
Підтримка Node.js	v18+	v20 LTS
Підтримка MongoDB	v6.0+	v7.0+

Таблиця 2.3 – Вимоги до мережі

Параметр	Вимога
Протокол передачі даних	HTTPS (порт 443)
Порт WebSocket	TCP 443 (або 3000 за проксі)
Пропускна здатність	мінімум 5 Мбіт/с
IP-адреса	статична або динамічна з DNS
DNS-підтримка	доменне ім’я із записами A/CNAME
Захист	брандмауер, захист від DDoS

2.1.4 Ізоляція даних користувачів та архітектура комунікацій

У вебмесенджері ізоляція даних користувачів та безпечна передача повідомлень у реальному часі є критично важливими аспектами. Основна мета — гарантувати, що жоден користувач не має доступу до чужих чатів або повідомлень, навіть при прямому запиті до API чи через злам клієнтського коду.

Архітектурно система передбачає централізовану базу даних, у якій кожен запис чітко асоційований з авторизованим користувачем через userId. Ізоляція даних досягається за допомогою кількох рівнів:

- JWT-автентифікація: кожен користувач після входу в систему отримує JWT-токен, який вбудовується в усі запити до API та WebSocket-сесій. Токен містить у собі унікальний userId, що використовується для фільтрації даних на сервері;

- Серверна перевірка доступу: усі запити до API супроводжуються перевіркою авторизованого userId з токена. Повідомлення доступні лише тим користувачам, які є учасниками відповідного чату;

- WebSocket-сесії з автентифікацією: під час встановлення з'єднання через Socket.IO клієнт надсилає JWT-токен у параметрах запиту. Сервер перевіряє токен і асоціює з'єднання з конкретним користувачем. Надсилання повідомлень між клієнтами відбувається лише після перевірки їхньої участі в одному чаті.

Сервер реалізує наступні механізми безпечної роботи з даними:

- після авторизації кожне з'єднання пов’язане з userId, що дозволяє серверу обмежити доступ до чужих чатів або повідомлень;

- коли користувач запитує список доступних чатів, сервер повертає лише ті, в яких userId є учасником;

- доступ до повідомлень дозволяється лише якщо користувач є учасником відповідного чату;

- облікові записи зберігаються з використанням хешування паролів через bcrypt, що виключає можливість компрометації навіть у разі витоку бази даних.

Переваги обраної архітектури:

- кожен користувач має доступ виключно до власних чатів та повідомлень;

- централізована база даних полегшує адміністрування, резервне копіювання та моніторинг;

- використання WebSocket забезпечує передачу повідомлень у реальному часі.

Обмеження та виклики:

- на відміну від систем з окремими БД, усі дані зберігаються в одній інстанції, що підвищує вимоги до валідації доступу на рівні коду;

- при збільшенні кількості користувачів можуть виникати вузькі місця у продуктивності, тому необхідне горизонтальне масштабування або кешування;

- повноцінне покриття сценаріїв реального часу вимагає комплексних інструментів для емуляції декількох клієнтів одночасно.

Потенційні напрями розвитку:

- перехід до розподіленої обробки WebSocket через Redis Pub/Sub або кластеризацію Socket.IO;

- кешування активних чатів і останніх повідомлень через Redis;

- додавання наскрізного шифрування повідомлень на клієнтському рівні для підвищення конфіденційності.

2.1.3 Діаграма варіантів використання

Для кращого розуміння взаємодії користувача з вебзастосунком FlexiChat побудовано діаграму варіантів використання (use case diagram), яка наведена на рисунку 2.2.

Діаграма демонструє ключові дії, які може виконати користувач системи.

Основні варіанти використання включають:

- перегляд списку контактів;

- вибір користувача для чату;

- перегляд статусу онлайн/офлайн;

- фільтрацію онлайн користувачів;

- перегляд історії повідомлень;

- відправку текстових повідомлень;

- відправку зображень;

- автоматичне прокручування до нових повідомлень;
- отримання повідомлень у реальному часі;
- закриття чату.

Рисунок 2.2 – Діаграма варіантів використання FlexiChat

2.2 Моделювання сутностей БД

Моделювання бази даних є ключовим етапом проєктування вебмесенджера, оскільки воно визначає, як саме зберігатимуться користувацькі дані, повідомлення та зв’язки між ними. У розроблюваній системі використовується документоорієнтована база даних MongoDB, що забезпечує гнучкість, масштабованість та високу швидкість доступу до даних [25].

MongoDB дозволяє зберігати дані у вигляді документів BSON, які логічно відповідають об’єктам JavaScript. У поточній реалізації система має дві основні колекції: users (користувачі) та messages (повідомлення). Кожен документ у колекції має унікальний ідентифікатор _id, який використовується для встановлення зв’язків між об’єктами [26].

2.2.1 Колекція users

Колекція users зберігає інформацію про зареєстрованих користувачів, наведена на таблиці 2.4.

Приклад документа user наведено на лістингу 2.1.

Таблиця 2.4 – Колекція users

Поле	Тип	Опис
_id	ObjectId	Унікальний ідентифікатор користувача
email	String	Електронна адреса користувача
fullName	String	Повне ім’я користувача
password	String (bcrypt)	Хешований пароль користувача за допомогою bcrypt
profilePic	String	Шлях до зображення профілю (або порожній рядок, якщо не вказано)
createdAt	Date	Дата створення облікового запису
updatedAt	Date	Дата останнього оновлення

Лістинг 2.1 – Приклад документа user

```
{
  "_id": "6833975d98edbc0d61150663",
  "email": "ivan@email.com",
  "fullName": "Ivan Kravchenko",
  "password":
"$2b$10$.4BLkClRBEcUBcBXfopHiuTAiO.gbjM.yk46PwiYv9MTezie3F2nu",
  "profilePic": "",
  "createdAt": "2025-04-12T12:56:55.705Z",
  "updatedAt": "2025-04-12T12:56:55.705Z"
}
```

2.2.2 Колекція messages

Колекція messages зберігає повідомлення, що обмінюються між користувачами. Кожне повідомлення належить конкретному відправнику та адресоване одному отримувачу (табл 2.5).

Таблиця 2.5 – Колекція users

Поле	Тип	Опис
_id	ObjectId	Унікальний ідентифікатор повідомлення
senderId	ObjectId	Ідентифікатор користувача, який надіслав повідомлення
receiverId	ObjectId	Ідентифікатор користувача, який отримав повідомлення
text	String	Текст повідомлення
createdAt	Date	Дата та час створення повідомлення
updatedAt	Date	Дата та час останнього оновлення (для редагованих повідомлень)

Приклад документа messages наведено на лістингу 2.2.

Лістинг 2.2 – Приклад документа messages

```
{
  "_id": "683396f498edbc0d61150662",
  "senderId": "67fe9360e8c22de5d144d848",
  "receiverId": "67e7cf01aba6989824279e8a",
  "text": "qwe",
  "createdAt": "2025-05-17T15:47:15.219Z",
  "updatedAt": "2025-05-17T15:47:15.219Z"
}
```

2.2.3 ER-модель бази даних

Для кращого розуміння структури збереження даних у застосунку FlexiChat було побудовано ER-модель, яка візуалізує зв’язки між основними сутностями системи. У застосунку використовується документно-орієнтована база даних MongoDB, тому модель відображає сутності у вигляді колекцій, а зв’язки — через посилання (reference) на _id інших документів.

Основні сутності:

- User — представляє користувача системи, включає ім’я, електронну пошту, пароль (у хешованому вигляді) та аватар;

- Message — зберігає текстові повідомлення, посилання на відправника та отримувача, дату створення, а також, за потреби, прикріплене зображення.

На рисунку 2.3 зображено ER-діаграму, яка демонструє зв’язки між цими сутностями.

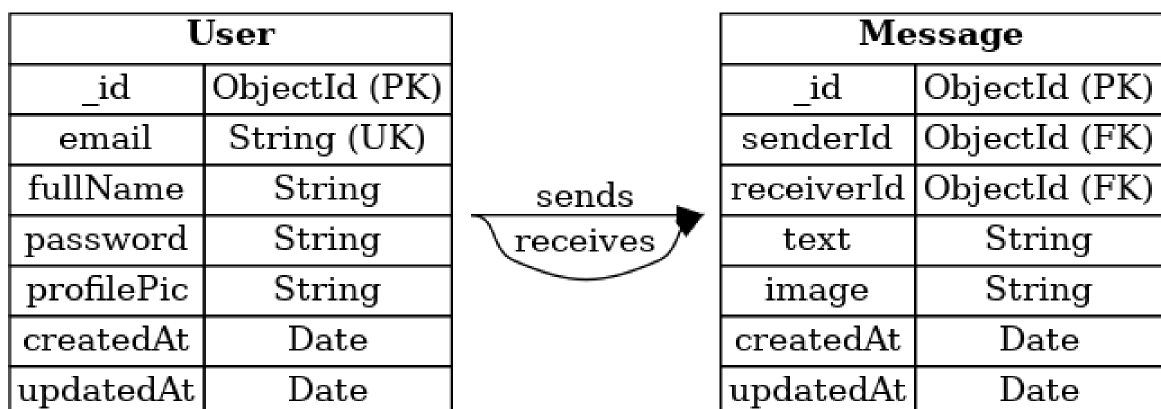


Рисунок 2.3 – ER-модель бази даних FlexiChat

2.3 Структура клієнтської частини

Клієнтська частина вебмесенджера відповідає за взаємодію з користувачем через вебінтерфейс, забезпечуючи обмін повідомленнями в реальному часі, перегляд історії діалогів, реєстрацію та авторизацію. Frontend реалізовано за допомогою бібліотеки React, використовується TypeScript для типобезпеки, Tailwind CSS — для адаптивної верстки, а Socket.IO — для обміну повідомленнями через WebSocket-з'єднання [27].

Для HTTP-запитів до REST API використовується Axios, маршрутизація реалізована через React Router DOM. Розробка ведеться з використанням Vite, що забезпечує швидке збирання та гаряче оновлення під час розробки. Клієнтська частина взаємодіє з сервером через два канали — REST API для реєстрації/авторизації та Socket.IO для обміну повідомленнями [28].

Діаграма структури клієнтської частини наведена на рисунку 2.4.

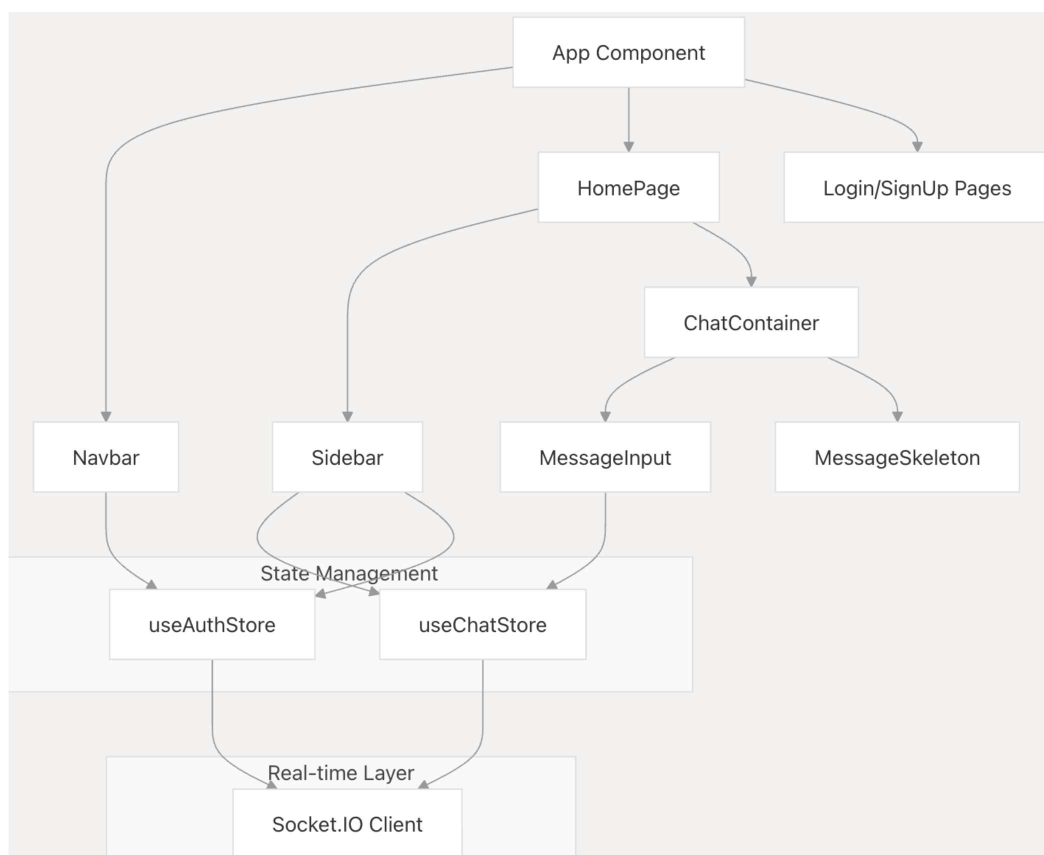


Рисунок 2.4 - Діаграма структури клієнтської частини

2.3.1 Компонентна архітектура та маршрутизація

Архітектуру компонентів клієнтської частини показано на рисунку 2.5.

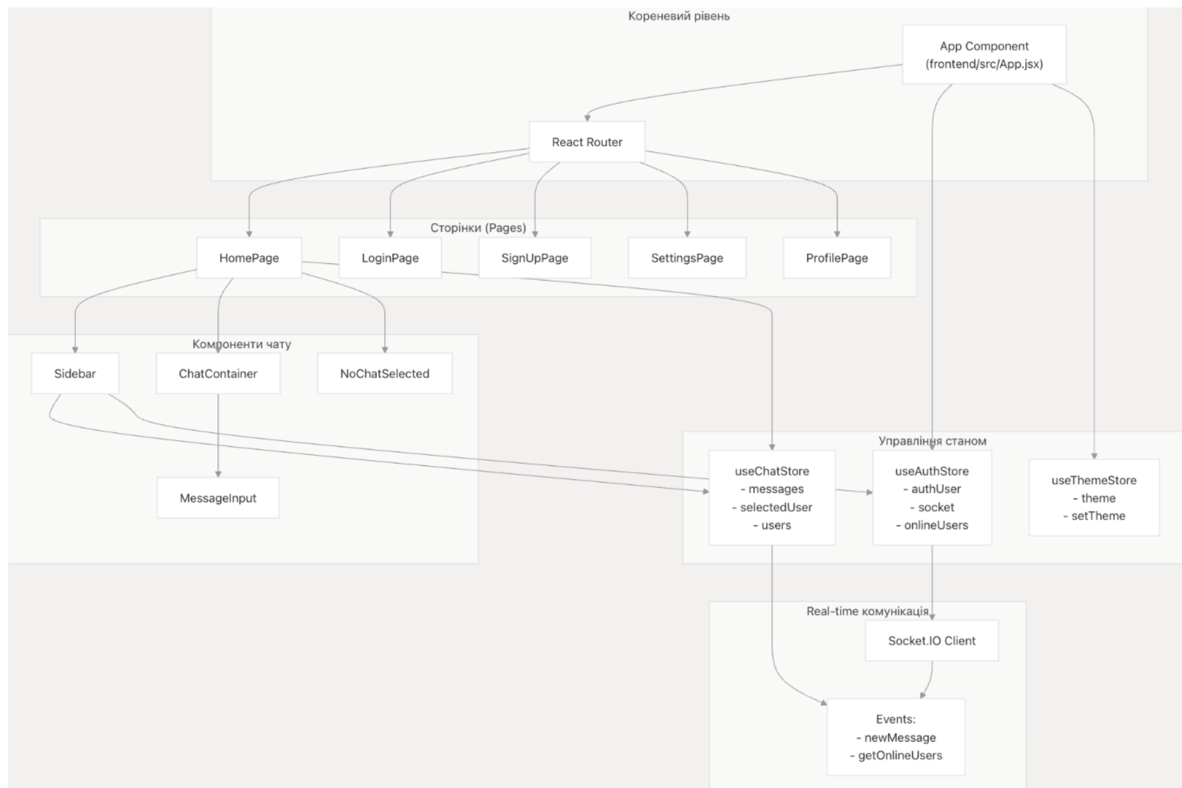


Рисунок 2.5 – Компонентна структура

Клієнтська частина має компонентну архітектуру, поділену на функціональні блоки:

а) сторінки:

1) LoginPage – сторінка авторизації користувача;

2) RegisterPage – сторінка створення нового облікового запису;

3) HomePage – головна сторінка після входу в систему, де розміщується інтерфейс чату;

б) компоненти інтерфейсу:

1) ChatList – список доступних діалогів користувача;

2) ChatWindow – область перегляду повідомлень вибраного чату;

3) MessageItem – окреме повідомлення (вхідне або вихідне);

4) MessageInput – поле введення повідомлення з кнопкою надсилання;

5) Navbar – верхня панель з кнопкою виходу та іменем користувача;

6) UserList – список користувачів для початку нового діалогу;

в) компоненти маршрутизації:

1) AppRouter – конфігурація маршрутів через React Router;

2) PrivateRoute – захист доступу до внутрішніх сторінок (авторизація);

3) NotFoundPage – сторінка 404;

г) утиліти та хуки:

1) useAuth – хук для роботи з токенами, вхід/вихід, отримання користувача;

2) useSocket – кастомний хук для роботи з WebSocket-підключенням;

3) formatDate – утиліта для форматування дати повідомлення.

2.3.2 Управління станом

Схема упраління станами наведена на рисунку 2.6.

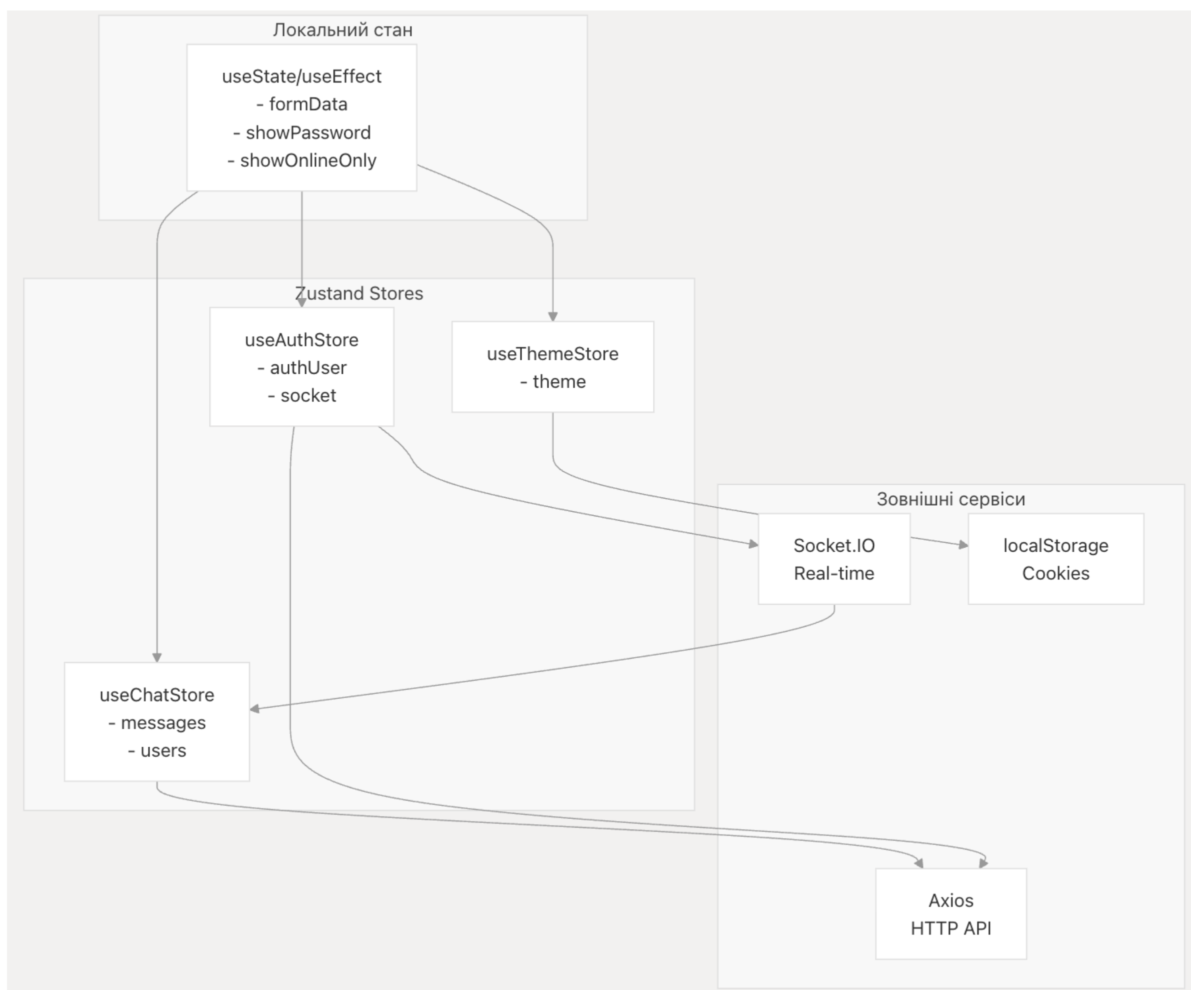


Рисунок 2.6 – Схема управління станами

Стан додатку управляється через поєднання локального стану, глобального стану та контекстів:

- локальний стан (useState, useEffect, useReducer) — використовується у компонентах для керування видимістю, введенням тексту, локальними подіями;

- глобальний стан — реалізований через React Context, який містить інформацію про авторизованого користувача та токен автентифікації;

- WebSocket-стан — підключення через Socket.IO, керується через кастомний хук useSocket, який автоматично обробляє вхідні повідомлення та оновлює список чатів/повідомлень;

- API-дані — отримуються через Axios з обробкою статусів завантаження, помилок та збереженням історії.

2.3.3 API-взаємодія

Клієнт взаємодіє з backend через два канали:

а) REST API (через Axios):

1) /api/register – створення облікового запису;
2) /api/login – авторизація та отримання JWT-токена;
3) /api/users – отримання списку користувачів;
4) /api/messages/:userId – отримання історії повідомлень;

б) WebSocket (через Socket.IO):

1) connect – встановлення з’єднання з сервером;
2) send-message – подія надсилання повідомлення;
3) receive-message – обробка вхідного повідомлення у реальному часі;
4) user-connected / user-disconnected – оновлення статусу контактів.

2.4 Структура серверної частини

Серверна частина побудована за принципом модульності, з розділенням на логічні частини (табл 2.6).

Таблиця 2.6 – Структура серверної частини

Модуль	Відповідальність
server.js / index.js	Точка входу: підключення до бази даних, запуск Express і Socket.IO
routes/	Маршрути REST API (автентифікація, повідомлення, користувачі)
controllers/	Обробка HTTP-запитів та взаємодія з моделями
models/	Mongoose-схеми: User, Message
middleware/	Middleware для перевірки JWT, обробки помилок
socket/	Ініціалізація Socket.IO, обробка подій надсилання/отримання повідомлень
utils/	Утиліти для роботи з токенами, хешування паролів, форматування часу

Серверна частина месенджера відповідає за обробку бізнес-логіки, управління автентифікацією, зберігання повідомлень, а також забезпечення обміну повідомленнями в реальному часі. Backend побудовано з використанням Node.js та Express.js, що забезпечує легкість, швидкість та масштабованість розробки. Зберігання даних реалізовано через MongoDB, а для забезпечення обміну повідомленнями в реальному часі використовується Socket.IO [29].

2.5 Структура інфраструктури

Інфраструктура вебмесенджера забезпечує зберігання даних, обробку повідомлень у реальному часі, розгортання серверної та клієнтської частин, а також підтримку середовища розробки.

Складові інфраструктури:

а) MongoDB:

1) централізована база даних MongoDB використовується для збереження всіх даних застосунку:

- колекція users зберігає інформацію про зареєстрованих користувачів;

- колекція messages — історію текстових повідомлень;

2) MongoDB підтримує динамічні схеми документів, що дозволяє легко адаптувати базу до нових функцій;

3) при зростанні навантаження база може бути масштабована за допомогою шардінгу або переходу на хмарну інфраструктуру;

4) зв’язок із сервером реалізовано через офіційний Node.js-драйвер або через бібліотеку Mongoose, що дозволяє зручно описувати схеми та валідацію;

б) Socket.IO (WebSocket-сервер):

1) Socket.IO створює канал зв’язку між клієнтом і сервером, що дозволяє обмінюватися повідомленнями в режимі реального часу;

2) за потреби може бути реалізовано горизонтальне масштабування через Redis Pub/Sub або Sticky Sessions;

3) сервер WebSocket інтегрується в Express-додаток та автентифікує користувачів через JWT;

в) Docker

1) можливість створення Docker-контейнерів для frontend, backend та MongoDB, що спрощує налаштування середовища розробки й розгортання;

2) Docker Compose дозволяє одночасно запускати всі частини проєкту у локальному середовищі або на сервері;

3) всі учасники проєкту можуть запускати систему в ідентичному оточенні, незалежно від ОС чи конфігурації;

г) інструменти розробки:

1) Vite — високошвидкісний збирач клієнтської частини з підтримкою HMR (hot module reload);

2) Postman / Thunder Client — для тестування REST API на етапі розробки [30];

3) nodemon — автоматичне перезавантаження backend-сервера під час змін;

4) dotenv — зберігання секретів і конфігурацій у .env файлі.

2.6 Загальний алгоритм роботи системи

Додаток запускається через основний компонент, який перевіряє, чи користувач автентифікований. У разі відсутності автентифікації відображається екран завантаження, що забезпечує плавний перехід до подальшої взаємодії. Для навігації між сторінками використовується система маршрутизації, яка базується на React Router. Сторінка основного інтерфейсу доступна лише для автентифікованих користувачів, тоді як сторінки для входу та реєстрації відкриті виключно для неавтентифікованих. Сторінки налаштувань і профілю доступні за умовного доступу, залежно від статусу автентифікації. Основний інтерфейс чату спроєктовано як двопанельну структуру, що забезпечує зручну взаємодію між списком контактів і активним діалогом (рис. 2.7).

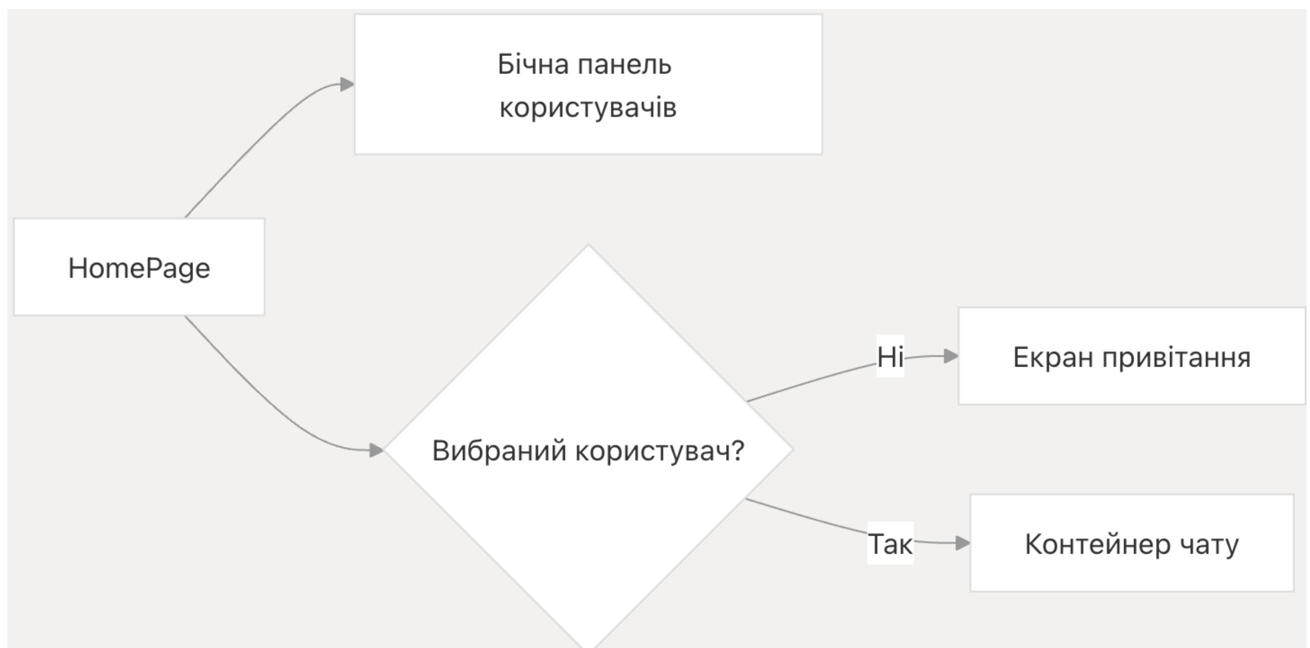


Рисунок 2.7 - Архітектура головної сторінки

Управління станом чату здійснюється через спеціалізований механізм, який відповідає за обробку даних про повідомлення та користувачів. Цей механізм забезпечує завантаження списку контактів, отримання історії діалогів і відправлення нових повідомлень, створюючи централізовану систему для роботи з

чатом. Для комунікації в реальному часі використовується технологія Socket.IO, яка дозволяє отримувати повідомлення миттєво через підписку на відповідні події. При зміні активного співрозмовника система автоматично оновлює підписку на нові повідомлення, забезпечуючи безперервне оновлення діалогу.

Відображення повідомлень реалізується через спеціальну область інтерфейсу, яка обробляє діалоги з автоматичним прокручуванням до останнього повідомлення для зручності користувача. Повідомлення можуть містити текст або зображення, причому відправлені та отримані повідомлення візуально відрізняються за стилем для легкого розрізнення. Управління списком контактів здійснюється через бічну панель, яка відображає перелік доступних співрозмовників із можливістю фільтрувати тих, хто перебуває в мережі, та показує їхній статус присутності, що полегшує швидку орієнтацію в активності користувачів.

2.7 Висновки до розділу 2

У процесі проєктування було змодельовано вебмесенджер із класичною клієнт-серверною архітектурою, що включає окремі компоненти: frontend, backend та інфраструктуру. Архітектура орієнтована на модульність, масштабованість, простоту супроводу та швидку обробку подій у реальному часі. Основними технологічними принципами стали поділ відповідальностей, використання сучасних бібліотек, а також підтримка гібридної комунікації через REST API і WebSocket [31].

Клієнтська частина побудована на React із використанням TypeScript, що забезпечує типобезпеку й інтуїтивну розробку. Tailwind CSS застосовується для побудови адаптивного та мінімалістичного інтерфейсу. Компонентна архітектура дозволяє ефективно розділяти функціональність: від відображення списку чатів до окремих повідомлень і введення тексту. Маршрутизація реалізована через React

Router, а управління станом — через локальні хуки та контексти. Передбачено взаємодію з backend через Axios (REST) та Socket.IO (обмін повідомленнями в реальному часі).

Серверна частина реалізована за допомогою Node.js та Express, із зберіганням даних у MongoDB. Архітектура backend поділена на маршрути, контролери, моделі та WebSocket-хендлери. Система підтримує автентифікацію через JWT, а паролі зберігаються у зашифрованому вигляді за допомогою bcrypt. Socket.IO забезпечує підтримку live-чату — нові повідомлення доставляються моментально. REST API відповідає за реєстрацію, логін, завантаження історії повідомлень і профілі користувачів.

База даних змодельована з урахуванням логічної ізоляції користувачів: кожне повідомлення має посилання на відправника та отримувача через ObjectId. Структура передбачає можливість масштабування, розширення (реакції, вкладення, групові чати) та індексацію для пришвидшення запитів. Для кожного користувача забезпечується доступ лише до власних чатів.

Інфраструктура включає:

- MongoDB як основне сховище даних;
- Vite — для збирання клієнтської частини;
- можливість контейнеризації через Docker;
- підтримку простого деплою на хмарні платформи.

Усі компоненти проєкту узгоджені з функціональними вимогами: автентифікація, обмін повідомленнями, історія чатів, безпечне зберігання даних. Також враховано нефункціональні вимоги — масштабованість, продуктивність, безпека, адаптивність інтерфейсу та зручність користування.

Результати цього етапу створюють міцну технічну основу для подальшої реалізації програмного забезпечення та впровадження додаткового функціоналу (групові чати, вкладення, нотифікації). Розроблена архітектура спрямована на конкурентоспроможність, простоту підтримки та подальший розвиток месенджера.

3 РЕАЛІЗАЦІЯ СЕРВІСУ

3.1 Реалізація системи реального часу

3.1.1 Ініціалізація WebSocket-сервера

На сервері створено окремий файл socket.js, у якому конфігурується логіка підключення користувачів до WebSocket та обробки подій надсилання/отримання повідомлень. Компонент призначенний для встановлення з'єднання з клієнтом, прослуховування події send-message та ретранслювання їх іншим користувачам (ліст 3.1).

Лістинг 3.1 – Реалізація WebSocket-серверу (файл socket.js)

```
const socketIO = require("socket.io");
let io;

const initSocket = (server) => {
  io = socketIO(server, {
    cors: {
      origin: process.env.CLIENT_URL,
      methods: ["GET", "POST"],
      credentials: true,
    },
  });

  io.on("connection", (socket) => {
    const userId = socket.handshake.query.userId;
    socket.join(userId);

    socket.on("send-message", (data) => {
      const { receiverId, message } = data;
      io.to(receiverId).emit("receive-message", {
        senderId: userId,
        message,
      });
    });

    socket.on("disconnect", () => {
      socket.leave(userId);
    });
  });
};

module.exports = { initSocket };
```

3.1.2 Ініціалізація WebSocket на клієнті

На стороні клієнта ініціалізація WebSocket з'єднання відбувається за допомогою бібліотеки socket.io-client. Компонент призначенний для підключення клієнта до WebSocket-сервера та оброблення вхідних повідомлень (ліст 3.2).

Лістинг 3.2 – Хук useSocket.js для ініціалізації підключення

```
import { useEffect, useRef } from "react";
import { io } from "socket.io-client";
import useAuthStore from "../store/auth";

const useSocket = () => {
  const socketRef = useRef(null);
  const user = useAuthStore((state) => state.user);

  useEffect(() => {
    if (user?._id) {
      socketRef.current = io(import.meta.env.VITE_SERVER_URL, {
        query: { userId: user._id },
        withCredentials: true,
      });
    }

    return () => {
      socketRef.current?.disconnect();
    };
  }, [user]);

  return socketRef;
};

export default useSocket;
```

3.1.3 Відправлення повідомлень

У компоненті ChatContainer.jsx реалізовано надсилання повідомлення через WebSocket і локальне оновлення стану чату (ліст 3.3).

Лістинг 3.3 – Компонент ChatContainer.jsx

```
socket.current.emit("send-message", {
  receiverId: selectedUser._id,
  message: newMessage,
});
```

3.2 Система автентифікації користувачів

Система автентифікації є критично важливим компонентом будь-якого вебзастосунку, що працює з персональними даними користувачів. У вебмесенджері реалізовано JWT-автентифікацію з хешуванням паролів через bcrypt, зберіганням токена у httpOnly cookie, захистом маршрутів та можливістю перевірки активної сесії.

3.2.1 Реєстрація користувача

Функція signup виконує перевірку унікальності email, хешування пароля та збереження користувача в базу MongoDB. Компонент призначенний для створення нового облікового запису з безпечним збереженням пароля (ліст. 3.4).

Лістинг 3.4 – Реєстрація користувача (signup)

```
exports.signup = async (req, res) => {
  const { email, fullName, password } = req.body;
  const existingUser = await User.findOne({ email });
  if (existingUser) return res.status(400).json({ error: "Email already exists" });
  const hashedPassword = await bcrypt.hash(password, 10);
  const user = await User.create({ email, fullName, password: hashedPassword });
  const token = createToken(user._id);
  res.cookie("token", token, { httpOnly: true, secure: true });
  res.status(201).json({ user });
};
```

3.2.2 Вхід у систему

При вході у систему функція login перевіряє правильність email і пароля. Якщо дані валідні — створюється JWT-токен, який передається клієнту через cookie (ліст 3.5).

Лістинг 3.5 – Авторизація користувача (login)

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

```
  if (!user) return res.status(400).json({ error: "Invalid credentials" });
  const isMatch = await bcrypt.compare(password, user.password);
  if (!isMatch) return res.status(400).json({ error: "Invalid credentials" });
  const token = createToken(user._id);
  res.cookie("token", token, { httpOnly: true, secure: true });
  res.status(200).json({ user });
};
```

3.2.3 Захист маршрутів

Для захисту приватних API-ендпоінтів реалізовано middleware protectRoute, який перевіряє JWT у cookie. Компонент призначенний для дозволення доступу лише авторизованим користувачам до чутливих маршрутів (ліст 3.6).

Лістинг 3.6 – Middleware захисту (protectRoute)

```
exports.protectRoute = async (req, res, next) => {
  const token = req.cookies.token;
  if (!token) return res.status(401).json({ error: "Not authorized" });

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = await User.findById(decoded.userId).select("-password");
    next();
  } catch {
    res.status(401).json({ error: "Token expired or invalid" });
  }
};
```

3.3 Компоненти користувацького інтерфейсу

Інтерфейс месенджера реалізований у вигляді SPA (Single Page Application) на базі React. Компонентна структура розділяє відповідальність між окремими функціональними частинами застосунку: область повідомлень, список користувачів, поле введення, тощо. Для стилізації використовується Tailwind CSS + DaisyUI, що забезпечує адаптивність і сучасний вигляд.

3.3.1 Компонент ChatContainer

Компонент ChatContainer є центральною частиною інтерфейсу чату, відповідаючи за відображення історії повідомлень та обробку нових вхідних і вихідних повідомлень. Він встановлює з'єднання з WebSocket для забезпечення комунікації в реальному часі, відображає список повідомлень у зрозумілій і візуально приємній формі, а також автоматично прокручує чат до останнього повідомлення, що робить процес спілкування зручним і природним (ліст. 3.7).

Лістинг 3.7 – Компонент ChatContainer

```
<div className={`chat ${isCurrentUser ? "chat-end" : "chat-
start"}`}>
  <div className="chat-bubble">{message.text}</div>
  <div className="text-
xs">{formatMessageTime(message.createdAt)}</div>
</div>
```

3.3.2 Компонент UserList

Компонент UserList призначений для показу списку всіх доступних для чату користувачів, дозволяючи відкрити діалог із вибраним контактом одним кліком. Він отримує дані про користувачів через API, підсвічує активний контакт для кращої орієнтації в інтерфейсі та підтримує skeleton loading під час завантаження, що забезпечує плавний і приємний користувацький досвід навіть у процесі отримання даних (ліст. 3.8).

Лістинг 3.8 – Компонент UserList

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';

const UserList = ({ selectedUser, onSelectUser }) => {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    axios.get('/api/users')
      .then(response => setUsers(response.data))
      .finally(() => setLoading(false));
  }, []);
  if (loading) {
```

```
    return <div>Loading users...</div>;
  }
  return (
    <ul>
      {users.map(user => (
        <li
          key={user.id}
          onClick={() => onSelectUser(user)}
          style={{ fontWeight: user.id === selectedUser?.id ? 'bold'
: 'normal' }}
        >
          {user.name}
        </li>
      ))}
    </ul>
  );
};
export default UserList;
```

3.3.3 Компонент MessageInput

Компонент MessageInput являє собою поле введення тексту разом із кнопкою "Надіслати", яке дозволяє користувачам створювати та відправляти повідомлення. Він підтримує відправлення повідомлень клавішею Enter, автоматично очищає поле введення після надсилання для зручності та викликає події через WebSocket для забезпечення миттєвої комунікації, роблячи процес обміну повідомленнями швидким і ефективним (ліст. 3.9).

Лістинг 3.9 – Компонент MessageInput

```
<form onSubmit={handleSubmit}>
  <input type="text" value={newMessage} onChange={handleChange} />
  <button type="submit">Send</button>
</form>
```

3.3.4 Компонент Sidebar

Компонент Sidebar є бічною панеллю, яка містить профіль користувача, кнопку "Вийти" та налаштування, такі як перемикач тем. Він відображає ім’я та аватар користувача для персоналізації, підтримує функцію виходу з системи з очищенням файлів cookie, а також дозволяє змінювати тему інтерфейсу, що робить його зручним і адаптованим до вподобань користувача (ліст. 3.10).

Лістинг 3.10 – Компонент MessageInput

```
import React from 'react';

const Sidebar = ({ user, onLogout, theme, toggleTheme }) => {
  return (
    <div className="sidebar">
      <div className="profile">
        <img src={user.avatar} alt="Avatar" />
        <p>{user.name}</p>
      </div>
      <button onClick={toggleTheme}>
        Switch to {theme === 'light' ? 'dark' : 'light'} theme
      </button>
      <button onClick={onLogout}>Logout</button>
    </div>
  );
};

export default Sidebar;
```

3.3.5 Компонент NoChatSelected

Компонент NoChatSelected відображається, коли не вибрано жодного співрозмовника, покращуючи користувацький досвід за допомогою чіткого візуального підказування під час першого входу або в разі відсутності активного чату. Цей компонент забезпечує інтуїтивно зрозумілий і відшліфований інтерфейс, допомагаючи користувачам орієнтуватися в програмі, коли діалог ще не розпочато (літ. 3.11).

Лістинг 3.11 – Компонент MessageInput

```
import React from 'react';

const NoChatSelected = () => {
  return (
    <div className="no-chat-selected">
      <p>Please select a chat to start messaging.</p>
    </div>
  );
};

export default NoChatSelected;
```

3.3.6 Компонент App

Компонент App є основною точкою входу у клієнтську частину застосунку FlexiChat. Він відповідає за ініціалізацію перевірки автентифікації користувача, підключення глобальних тем інтерфейсу, відображення маршрутизованих сторінок, а також загальних елементів, таких як Navbar і Toaster для сповіщень.

Після виклику checkAuth() через хук useEffect, застосунок очікує завершення перевірки. Якщо автентифікація ще триває — відображається анімований лоадер. Залежно від статусу автентифікації, користувача автоматично перенаправляє або на домашню сторінку, або на сторінки входу/реєстрації.

Компонент App є центральним у визначенні логіки маршрутизації всього застосунку (ліст. А.1).

3.3.7 Компонент ChatContainer

Компонент ChatContainer відповідає за основну частину інтерфейсу спілкування — тобто, безпосереднє відображення історії повідомлень, аватарів, текстів і зображень, а також введення нових повідомлень через MessageInput.

Компонент підключає повідомлення обраного співрозмовника, отримуючи їх через getMessages, і підписується на нові повідомлення в реальному часі через subscribeToMessages. Також реалізовано автоматичне прокручування вниз після отримання кожного нового повідомлення. У разі, якщо повідомлення ще не завантажено, виводиться скелетон-інтерфейс MessageSkeleton.

Компонент демонструє адаптивний та реактивний інтерфейс спілкування користувача з іншими, підкреслюючи, з якого облікового запису надійшло кожне повідомлення (ліст. А.2).

3.3.8 Компонент MessageInput

Компонент MessageInput відповідає за введення повідомлень у чаті, включаючи текст і зображення. Він забезпечує функціональність попереднього перегляду зображення, валідацію типу та розміру файлу (не більше 50 KB), а також очищення форми після надсилання.

Усяя логіка побудована на хуках useState, useRef та функції sendMessage з контексту чату. Зображення завантажуються через елемент <input type="file" />,

прихований у візуальному інтерфейсі, та активуються через окрему кнопку. При виборі зображення генерується попередній перегляд за допомогою FileReader.

Цей компонент є ключовим для взаємодії користувача з системою, забезпечуючи швидке введення повідомлень і візуальну гнучкість при додаванні медіа (літ. А.3).

3.3.9 Компонент Navbar

Компонент Navbar реалізує верхню панель навігації, яка доступна на всіх сторінках застосунку. Вона містить логотип і назву застосунку, кнопку переходу до сторінки налаштувань, а також елементи керування обліковим записом — посилання на профіль та кнопку виходу. Якщо користувач автентифікований, він бачить повний набір навігаційних можливостей; у протилежному випадку — лише базові.

Інтерфейс адаптований для різних розмірів екранів: деякі текстові елементи приховані на малих екранах, що підвищує зручність використання на мобільних пристроях. Вихід з облікового запису реалізовано через виклик методу logout з глобального сховища (літ. А.5).

3.3.10 Компонент SignUpPage

Компонент SignUpPage реалізує інтерфейс реєстрації нового користувача. Він містить форму з полями для введення повного імені, електронної пошти та пароля. Реалізована валідація на клієнтському рівні: перевірка на заповненість, формат email, мінімальну довжину пароля. У разі помилок користувач отримує повідомлення через toast.

Також реалізована можливість перегляду введеного пароля за допомогою перемикача showPassword. Після успішної перевірки форми викликається метод signup зі сховища useAuthStore, що відправляє дані на backend.

Візуальне оформлення реалізовано через Tailwind CSS, із розбиттям інтерфейсу на дві частини: форму та ілюстративний блок з текстом (літ. А.6).

3.3.11 Контролер auth.controller.jsx

Файл auth.controller.jsx реалізує основну бізнес-логіку автентифікації на стороні сервера. У ньому містяться такі функції:

- signup: перевіряє вхідні дані, хешує пароль через bcrypt, створює нового користувача та встановлює JWT-токен;

- login: виконує перевірку автентичності, порівнює хешовані паролі, генерує токен і повертає користувача;

- logout: видаляє JWT з cookie, завершуючи сесію;

- updateProfile: оновлює аватар користувача через сервіс Cloudinary;

- checkAuth: повертає інформацію про поточного автентифікованого користувача, якщо токен валідний.

Контролер дотримується принципів REST і має обробку помилок, що підвищує стабільність API (ліст. А.7).

3.4 Управління станом застосунку

У вебмесенджері стан керується через сучасну легку бібліотеку Zustand, яка дозволяє створювати глобальні сховища для різних частин застосунку. Крім того, локальний стан контролюється стандартними хуками React (useState, useEffect), а взаємодія між сокетами й інтерфейсом координується через централізовані store.

3.4.1 Глобальний стан автентифікації

Глобальний стан автентифікації, визначений у файлі store/auth.js, призначений для зберігання інформації про поточного користувача та його статус авторизації. Він містить основні поля, такі як user для збереження даних користувача, метод setUser(user) для встановлення цих даних після успішного входу чи реєстрації, а також метод logout(), який очищає інформацію про користувача та викликає API для завершення сеансу, забезпечуючи надійне керування автентифікацією (ліст. 3.12).

Лістинг 3.12 – Компонент store/auth.js

```
import { create } from "zustand";
const useAuthStore = create((set) => ({
```

```
  user: null,
  setUser: (user) => set({ user }),
  logout: async () => {
    await fetch("/api/auth/logout");
    set({ user: null });
  },
}));
export default useAuthStore;
```

3.4.2 Стан повідомлень і чату

Стан повідомлень і чату, реалізований у файлі store/chat.js, відповідає за управління активним співрозмовником, списком повідомлень та вибором чату. Він включає поле selectedUser для зберігання інформації про вибраного користувача для діалогу, messages для збереження поточної історії повідомлень, а також методи setSelectedUser(user) для вибору користувача, addMessage(msg) для додавання нового повідомлення до списку та clearChat() для очищення чату при переході до іншого діалогу, забезпечуючи ефективне управління станом чату (ліст. 3.13).

Лістинг 3.13 – Компонент store/chat.js

```
import { createSlice } from '@reduxjs/toolkit';
const chatSlice = createSlice({
  name: 'chat',
  initialState: {
    selectedUser: null,
    messages: [],
  },
  reducers: {
    setSelectedUser(state, action) {
      state.selectedUser = action.payload;
      state.messages = []; // Clear messages when a new user is
selected
    },
    addMessage(state, action) {
      state.messages.push(action.payload);
    },
    clearChat(state) {
      state.selectedUser = null;
      state.messages = [];
    },
  },
});
export const { setSelectedUser, addMessage, clearChat } =
chatSlice.actions;
export default chatSlice.reducer;
```

3.4.3 Стан теми інтерфейсу

Стан теми інтерфейсу, описаний у файлі store/theme.js, призначений для керування світлою або темною темою застосунку з можливістю збереження налаштувань у localStorage. Цей стан дозволяє користувачам перемикатися між темами, зберігаючи їхні вподобання для наступних сеансів, що забезпечує гнучкість і зручність у налаштуванні зовнішнього вигляду інтерфейсу (ліст. 3.14).

Лістинг 3.14 – Компонент store/theme.js

```
const useThemeStore = create((set) => ({
  theme: localStorage.getItem("theme") || "light",
  toggleTheme: () => set((state) => {
    const newTheme = state.theme === "light" ? "dark" : "light";
    localStorage.setItem("theme", newTheme);
    return { theme: newTheme };
  }),
}));
```

3.5 Адаптивний дизайн

Адаптивність інтерфейсу — одна з важливих характеристик сучасних вебзастосунків, що забезпечує коректне відображення контенту на різних типах пристроїв: комп’ютерах, планшетах, смартфонах. У розробці месенджера застосовується Tailwind CSS разом з DaisyUI, що забезпечує адаптивну верстку через мобільні брейкпоінти та утилітні класи.

Tailwind CSS — це утилітна CSS-бібліотека, яка дозволяє безпосередньо в класах HTML/JSX-елементів описувати поведінку елементів при різних розмірах екранів (ліст. 3.15).

Лістинг 3.15 – Приклад адаптивності

```
<div className="flex flex-col md:flex-row gap-4">
  <Sidebar />
  <ChatContainer />
</div>
```

В деяких компонентах, таких як Sidebar, використовується умовна видимість залежно від розміру екрану (ліст 3.16).

Лістинг 3.16 – Адаптивність залежно від розміру екрану

```
<div className="hidden md:block w-full md:w-[250px]">
  {/* Sidebar content */}
</div>
```

Компонент ChatContainer змінює відступи, шрифти, розміри залежно від брейкпоінтів (ліст 3.17).

Лістинг 3.17 – Адаптивність залежно від брейкпоінтів

```
<div className="p-2 sm:p-4 text-sm sm:text-base h-[calc(100vh-64px)]">
```

DaisyUI автоматично підлаштовується під тему, яку користувач може перемикати вручну. Tailwind підтримує ці теми за класом data-theme (ліст 3.18).

Лістинг 3.18 – Темна/світла тема

```
<html data-theme="light"> або <html data-theme="dark">
```

3.6 Висновки до розділу 3

У процесі реалізації було створено повноцінний вебмесенджер із підтримкою обміну повідомленнями в реальному часі, автентифікацією, адаптивним інтерфейсом і хмарною інтеграцією. Система відповідає всім поставленим функціональним та нефункціональним вимогам, включаючи швидкодію, безпеку, масштабованість і зручність користування.

Серверна частина, побудована на Node.js з Express, забезпечує обробку REST-запитів, автентифікацію через JWT, збереження користувачів і повідомлень

у базі даних MongoDB, а також підтримку обміну повідомленнями в реальному часі через Socket.IO. Аутентифікація захищена через httpOnly cookie, а паролі шифруються з використанням bcrypt, що гарантує безпеку персональних даних.

Клієнтська частина, реалізована на React, має модульну архітектуру з використанням Zustand для керування станом, Tailwind CSS і DaisyUI для стилізації та адаптивності, а також підтримує перемикання теми інтерфейсу. Застосунок підтримує динамічну маршрутизацію, зручну навігацію, миттєвий обмін повідомленнями та автоматичне оновлення чатів без перезавантаження.

Для зберігання зображень профілю інтегровано Cloudinary, що дозволяє зменшити навантаження на сервер і забезпечити швидке завантаження медіафайлів через CDN. Ця інтеграція також дозволяє обмежити формат і розмір зображень, підвищуючи UX.

Особливу увагу приділено адаптивному дизайну — верстка реалізована за принципом mobile-first з використанням брейкпоінтів Tailwind CSS. Інтерфейс однаково зручний як на десктопах, так і на мобільних пристроях.

Загалом, розроблений вебмесенджер демонструє ефективне поєднання сучасних технологій фронтенду та бекенду, забезпечуючи стабільну роботу, безпечну авторизацію, просту підтримку та можливість подальшого масштабування. Реалізована архітектура та компоненти створюють міцну основу для подальшого розширення функціоналу — зокрема, впровадження групових чатів, нотифікацій, вкладень або push-повідомлень.

4 ВИПРОБУВАННЯ СИСТЕМИ

4.1 Сторінки застосунку

4.1.1 Сторінки реєстрації та входу

Сторінки реєстрації та входу в застосунок FlexiChat забезпечують просту та інтуїтивно зрозумілу взаємодію з користувачем. Вони слугують початковою

точкою взаємодії з месенджером та реалізовані у відповідності до принципів UX-дизайну.

На сторінці входу користувач може ввести свої облікові дані і натиснути кнопку Sign in для доступу до особистого кабінету. У випадку, якщо користувач ще не зареєстрований, доступне посилання Create account (рис. 4.2).

Сторінка реєстрації дозволяє створити новий обліковий запис, ввівши повне ім'я, email та пароль. Після натискання кнопки Create Account система виконує перевірку валідності даних та створює нового користувача (рис. 4.1).

На обох сторінках реалізовано базову валідацію форм: перевірку формату електронної пошти, довжини пароля, а також відображення помилок при неправильному введенні даних. При виникненні помилок або при успішній авторизації система відображає відповідне повідомлення.

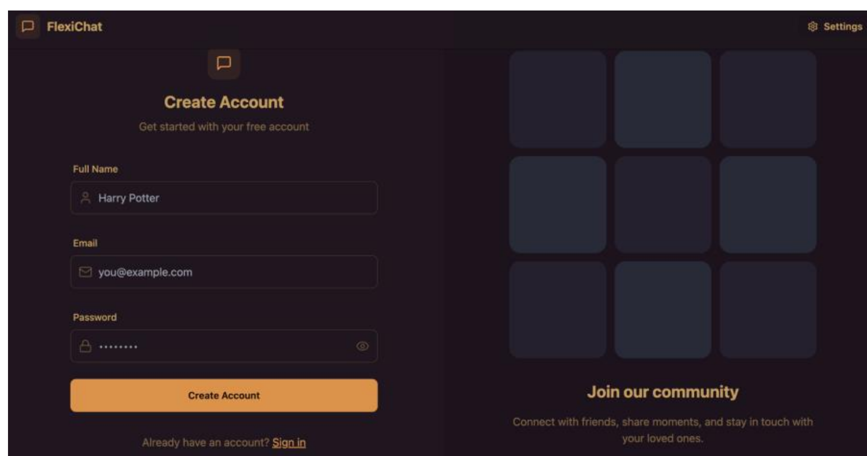


Рисунок 4.1 – Сторінка реєстрації

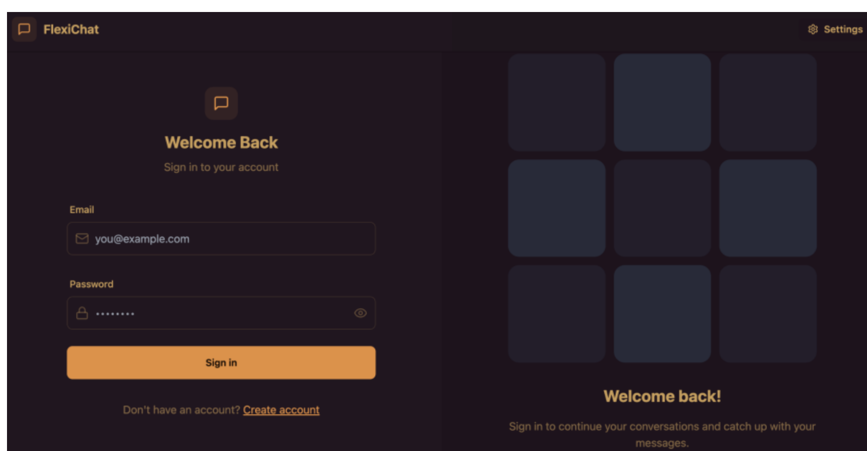


Рисунок 4.2 – Сторінка входу в обліковий запис

4.1.2 Хедер

Хедер є постійним елементом інтерфейсу застосунку FlexiChat і розміщується у верхній частині всіх сторінок. Він забезпечує базову навігацію та швидкий доступ до головних функцій. На рисунку 4.3 продемонстровано вигляд хедера на різних розмірах екранів, що підтверджує його адаптивність.

До складу хедера входять:

- логотип і назва застосунку, розміщені ліворуч, які одночасно є посиланням на головну сторінку;

- кнопка “Settings”, розташована праворуч, що відкриває сторінку з налаштуваннями інтерфейсу.

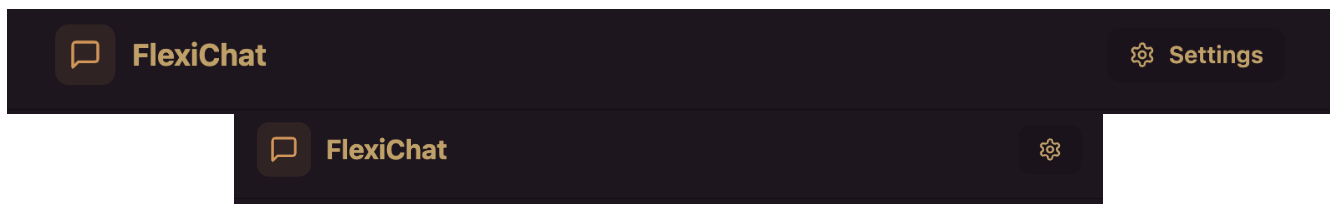


Рисунок 4.3 – Адаптивність хедера на різних пристроях

4.1.3 Головна сторінка

Головна сторінка застосунку FlexiChat надає користувачу доступ до основного функціоналу системи — обміну повідомленнями з іншими зареєстрованими користувачами. Інтерфейс реалізовано з використанням компонентного підходу React та стилізовано за допомогою Tailwind CSS, що забезпечує адаптивність, легкість сприйняття й естетичну узгодженість.

Інтерфейс головної сторінки структуровано на кілька функціональних блоків (зліва направо на рисунку 4.4):

- секція контактів — відображає всіх користувачів, доступних для чату. Вибір одного з них відкриває відповідну переписку. Також присутня кнопка-перемикач, яка дозволяє виводити тільки тих користувачів, які на даний момент в мережі;

- секція повідомлень — центральна область, яка показує історію листування з вибраним користувачем у хронологічному порядку. Внизу розміщується поле для введення та надсилання нових повідомлень.

Завдяки адаптивному дизайну інтерфейс автоматично підлаштовується під розмір екрана: на мобільних пристроях акценти зміщуються — спочатку відображається список чатів, після вибору — сам чат, що покращує користувацький досвід на малих екранах (рис. 4.4 – 4.7).

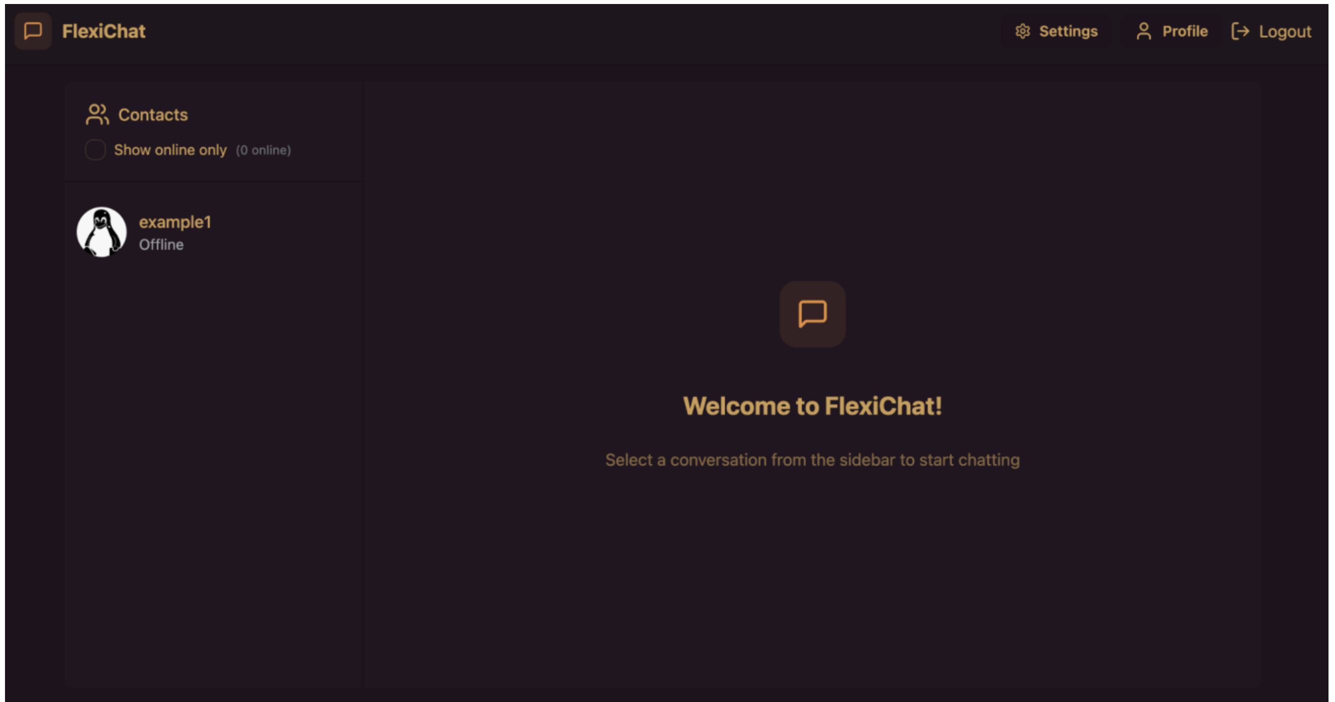


Рисунок 4.4 – Головна сторінка FlexiChat (десктопна версія)

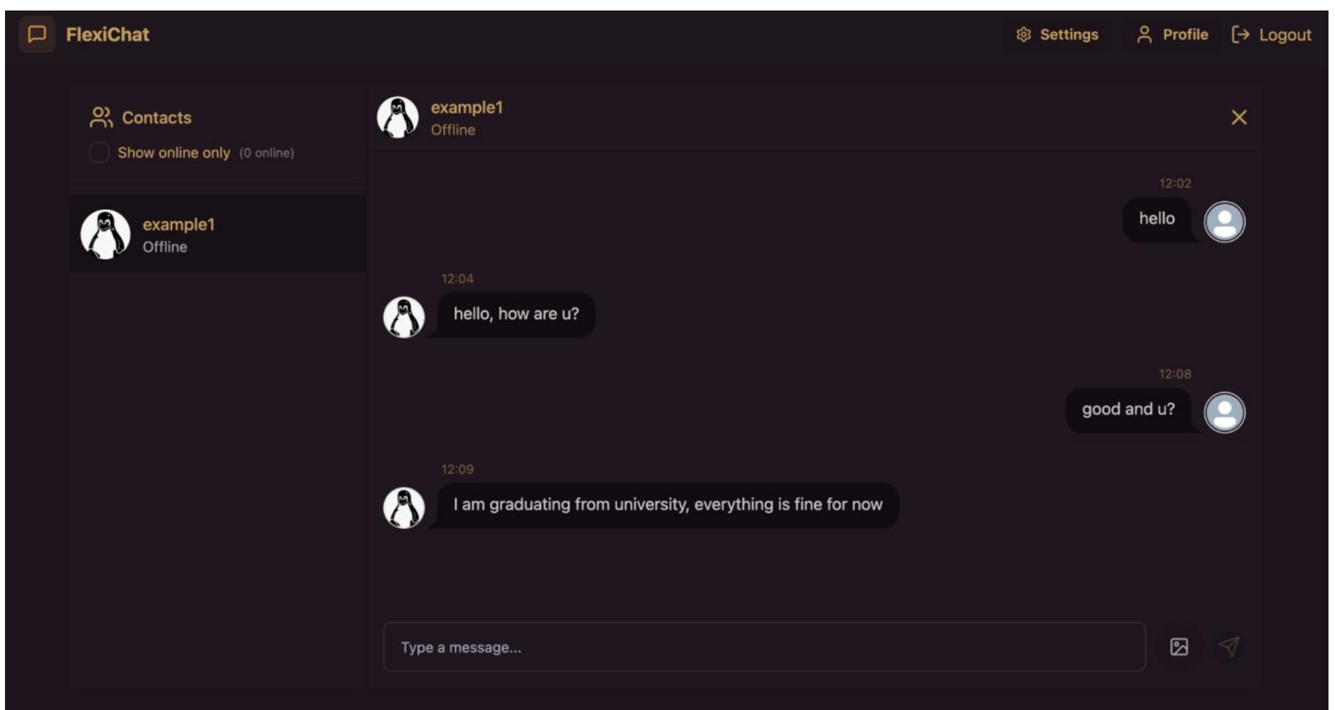


Рисунок 4.5 – Головна сторінка FlexiChat при активному чаті (десктопна версія)

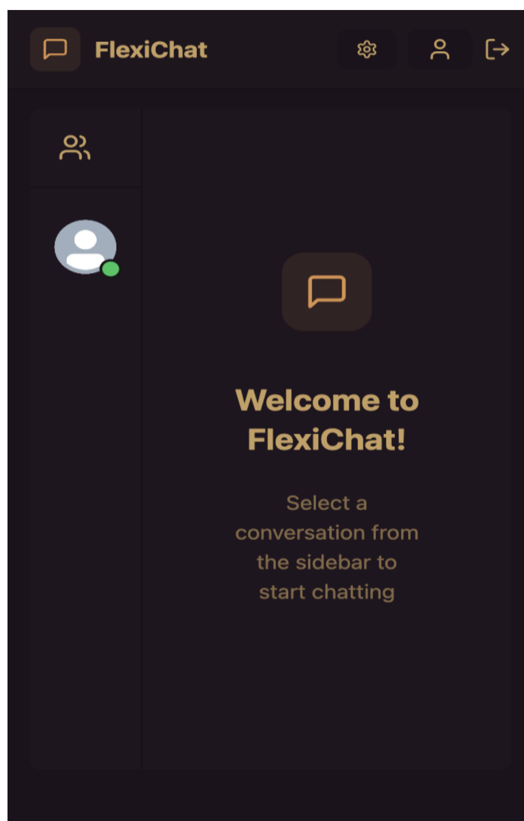


Рисунок 4.6 – Головна сторінка FlexiChat (iPhone SE, 375×667)

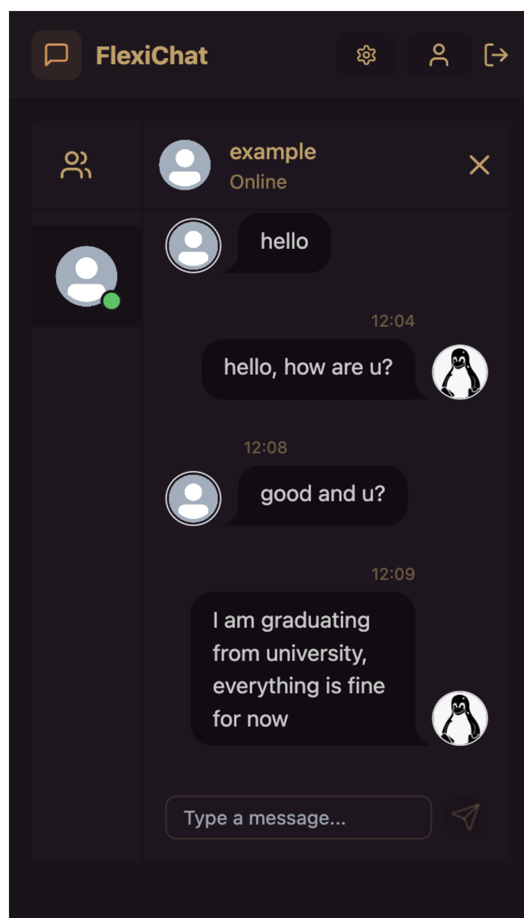


Рисунок 4.7 – Головна сторінка FlexiChat при активному чаті (iPhone SE, 375×667)

4.1.4 Сторінка налаштувань

Сторінка налаштувань у застосунку FlexiChat дозволяє користувачу персоналізувати інтерфейс, змінюючи візуальну тему оформлення. У доступному переліку тем користувач може обрати світлу, темну або інші варіанти. Для зручності під кожним варіантом теми наведено попередній перегляд, який демонструє, як виглядатиме вікно чату після застосування відповідного стилю.

Інтерфейс сторінки залишається адаптивним: на мобільних пристроях елементи автоматично підлаштовуються під розмір екрана (рис. 4.8 – 4.9).

Рисунок 4.8 – Сторінка налаштувань FlexiChat (iPhone SE, 375×667)

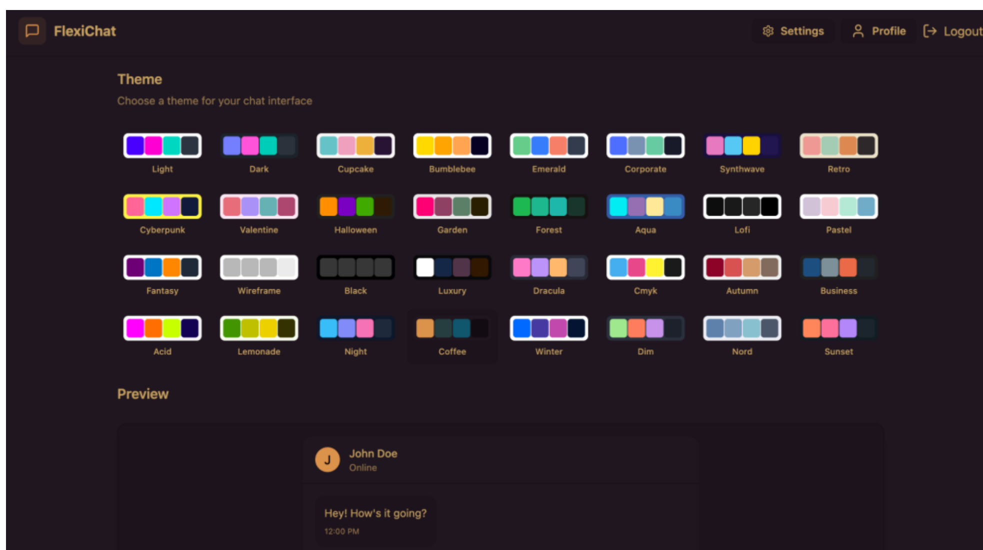


Рисунок 4.9 – Сторінка налаштувань FlexiChat (десктопна версія)

4.1.5 Сторінка профілю

Сторінка профілю в застосунку FlexiChat відображає основну інформацію про поточного користувача: ім’я, електронну адресу, а також дату створення облікового запису. Ця інформація дозволяє користувачу перевірити, під яким акаунтом він авторизований, та коли було здійснено реєстрацію. Також сторінка профілю дозволяє встановити фото обліково запису, яке можна обрати з файлів на комп’ютері або телефоні (рис 4.12).

Інтерфейс сторінки мінімалістичний і адаптований до всіх типів пристроїв, що забезпечує зручність перегляду як на десктопах, так і на смартфонах (рис 4.10 - 4.11).

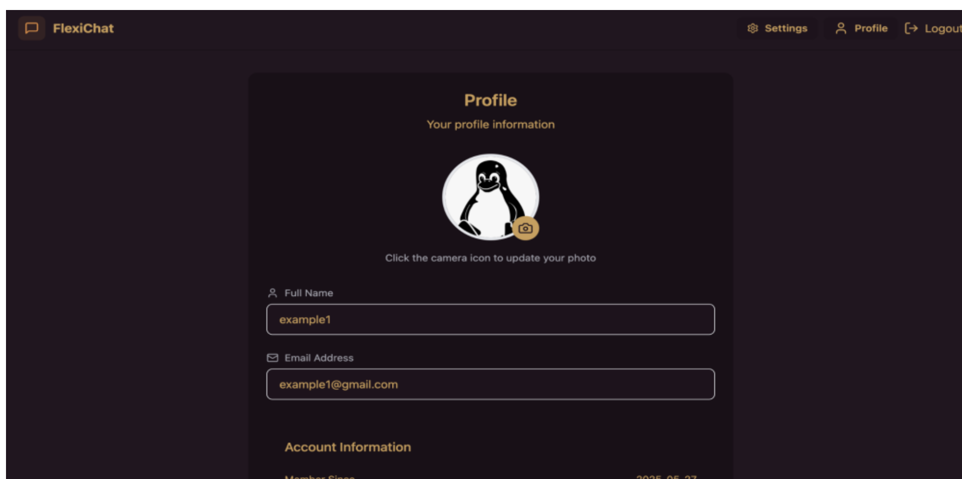


Рисунок 4.10 – Сторінка профілю FlexiChat (десктопна версія)

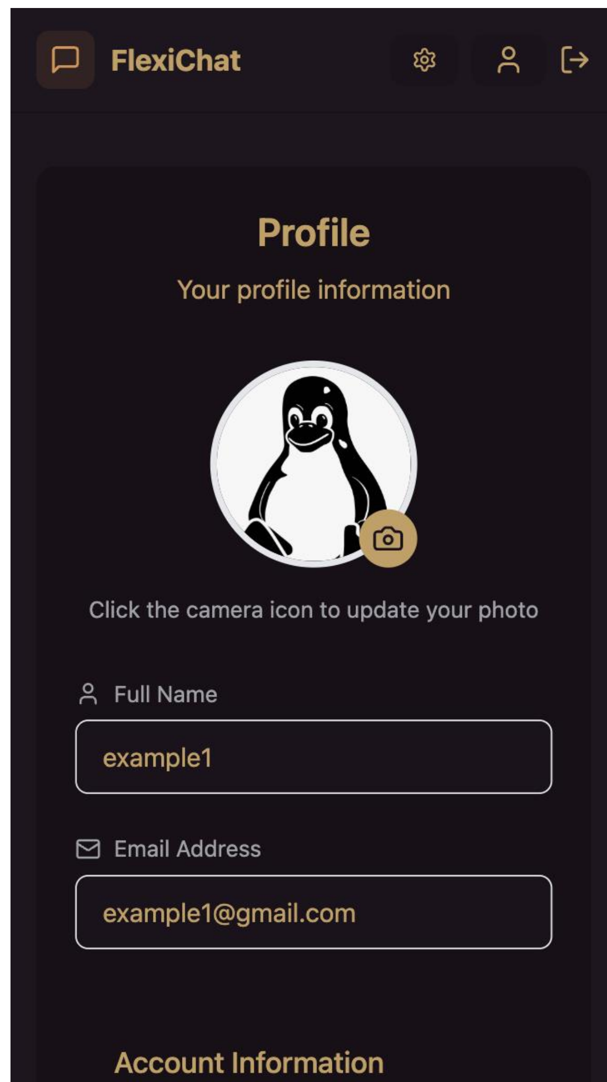


Рисунок 4.11 – Сторінка профілю FlexiChat (iPhone SE, 375×667)

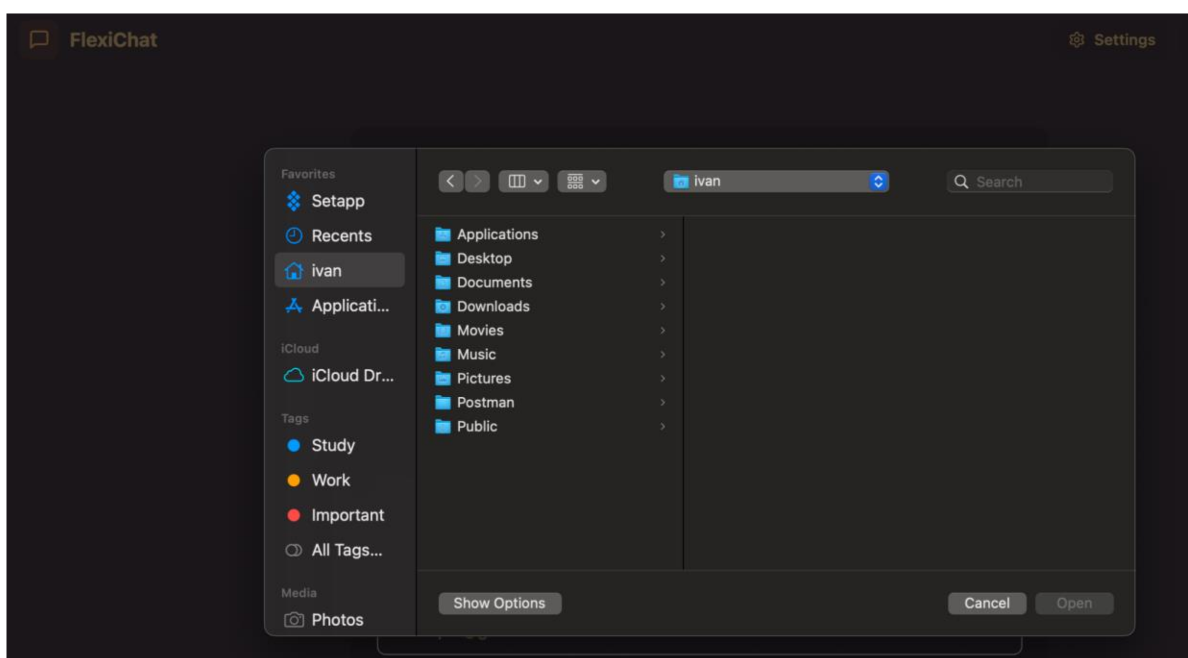


Рисунок 4.12 – Встановлення фото облікового запису

4.2 Функції застосунку

4.2.1 Обмін повідомленнями в реальному часі

Однією з головних функціональних можливостей FlexiChat є миттєвий обмін повідомленнями між користувачами в режимі реального часу. Технологія WebSocket, реалізована через бібліотеку Socket.IO, забезпечує постійне з'єднання між клієнтом і сервером, що дозволяє передавати нові повідомлення без затримок.

Коли користувач надсилає повідомлення у відкритому чаті, воно одразу відображається у власному інтерфейсі, а також миттєво з'являється у вікні співрозмовника, якщо той знаходиться онлайн. Усі повідомлення паралельно зберігаються в базі даних MongoDB (рис. 4.13).

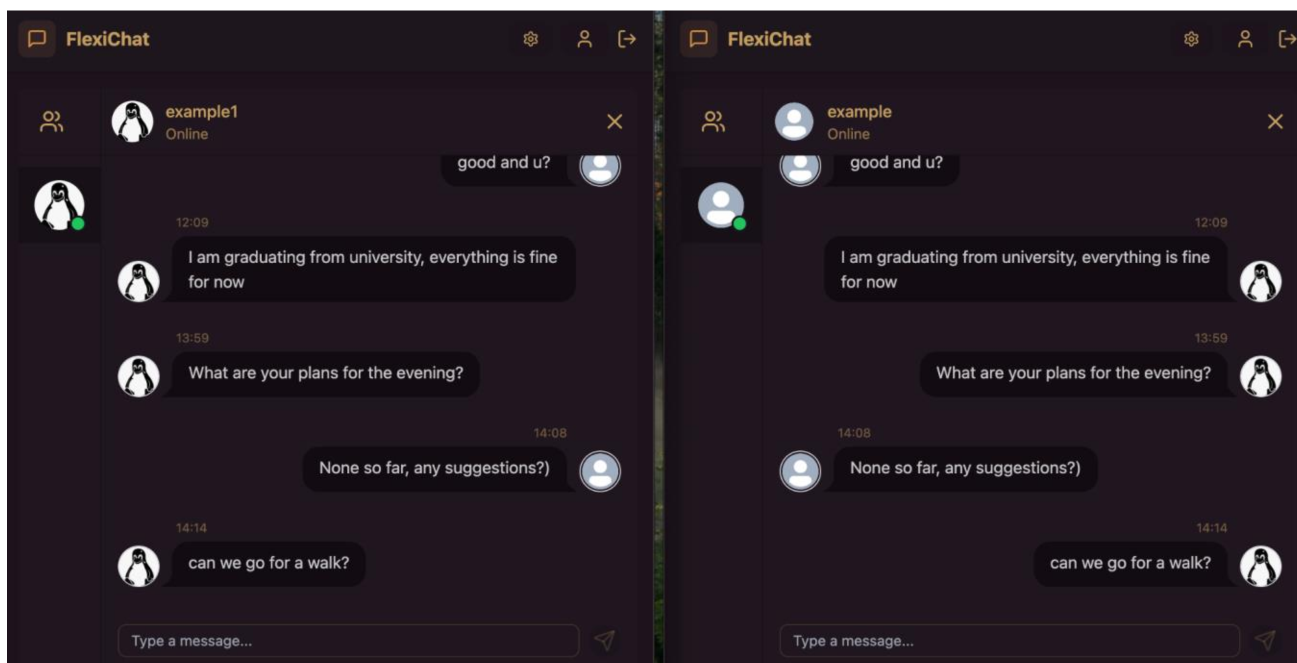


Рисунок 4.13 – Миттєве отримання повідомлення іншим користувачем

4.2.2 Зміна теми інтерфейсу в налаштуваннях

Застосунок FlexiChat дозволяє користувачу персоналізувати вигляд інтерфейсу шляхом вибору теми з-поміж доступних варіантів. Це реалізовано на окремій сторінці налаштувань, де для кожної теми надається попередній перегляд вигляду чату (рис. 4.14).

Після вибору теми інтерфейс автоматично оновлюється без перезавантаження сторінки. Обрана тема зберігається в localStorage, що дозволяє зберігати вибір навіть після оновлення сторінки або повторного входу в систему.

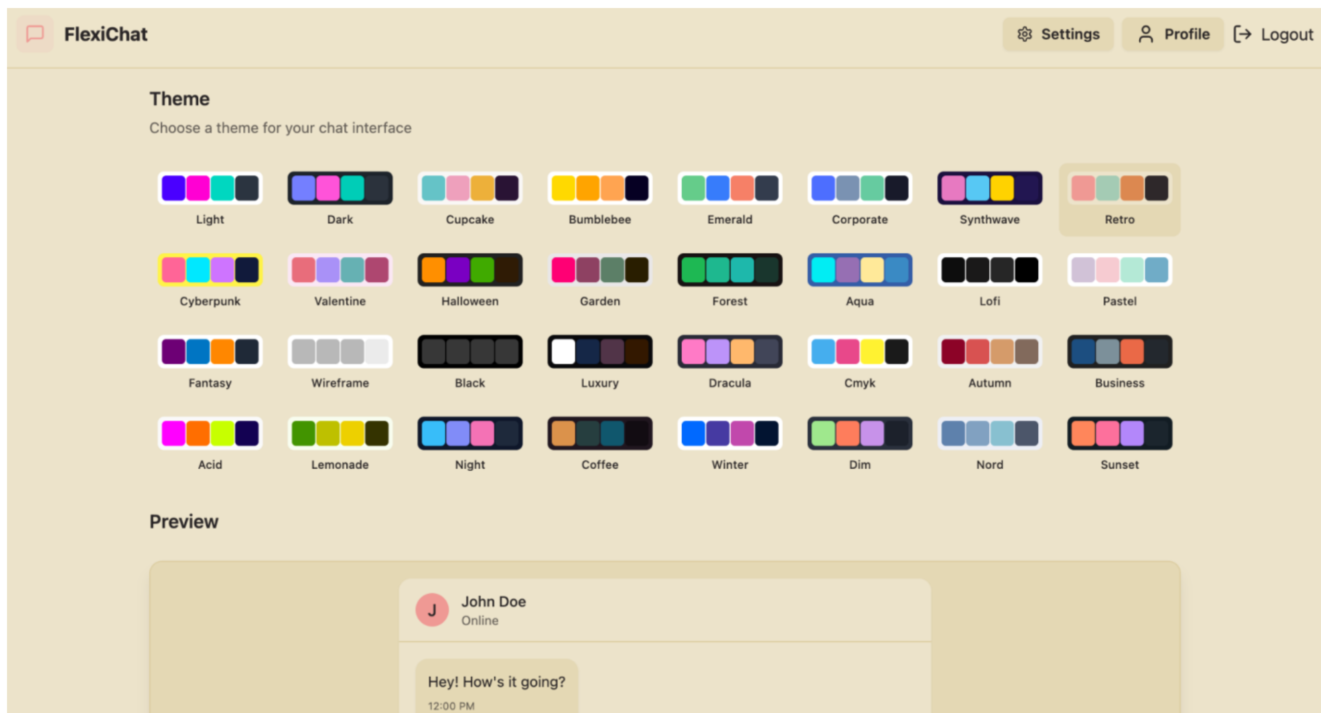


Рисунок 4.14 – Зміна теми інтерфейсу в налаштуваннях

4.3 Безпека та захист даних

У вебзастосунку FlexiChat особливу увагу приділено безпеці даних користувачів. Реалізовано стандартні механізми автентифікації, хешування паролів та захисту доступу до персональних даних. Завдяки використанню сучасних технологій, система відповідає базовим вимогам до конфіденційності, цілісності та автентичності інформації.

4.3.1 Аутентифікація користувачів

Для автентифікації користувачів застосовується механізм JWT, який дозволяє безпечно ідентифікувати користувача після входу в систему. Після

успішної авторизації сервер створює токен, який шифрується за допомогою секретного ключа та зберігається у клієнта у вигляді httpOnly cookie (рис. 4.15).

Такий підхід дозволяє:

- уникнути доступу до токена з боку JavaScript;

- автоматично передавати токен при кожному запиті до серверу;

- реалізувати middleware для перевірки автентифікації користувача при доступі до захищених маршрутів.

Кожен захищений API-запит перевіряє наявність та дійсність токена. Якщо токен відсутній або недійсний — доступ блокується.

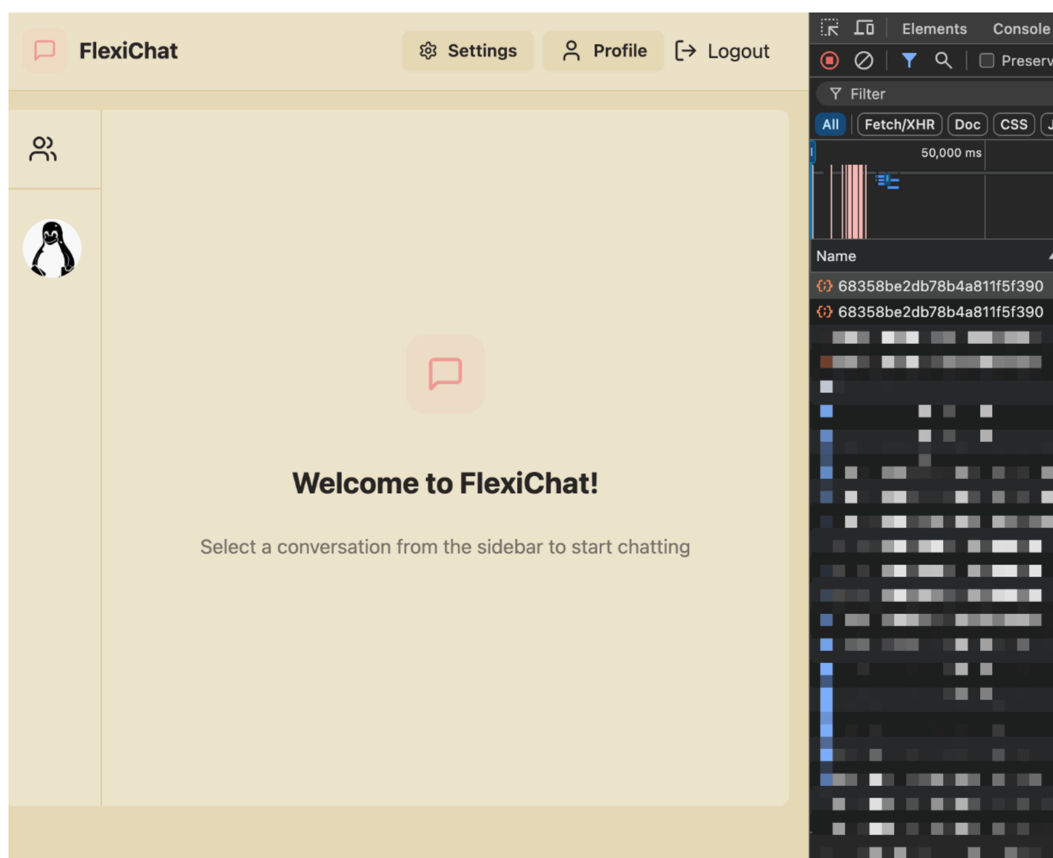


Рисунок 4.15 - Призначення JWT при вході в обліковий запис

4.3.2 Хешування паролів облікових записів

Для зберігання паролів у базі даних використовується алгоритм bcrypt, який дозволяє захистити облікові записи навіть у випадку витоку бази даних. Перед збереженням у MongoDB пароль хешується з використанням сільованого хешування з 10 ітераціями (рис. 4.16).

Переваги використання bcrypt:

- унеможливлення відновлення початкового пароля з хешу;
- захист від атак методом перебору (brute force);
- кожен хеш унікальний, навіть для однакових паролів.

Такий підхід відповідає галузевим стандартам безпеки та дозволяє впевнено зберігати конфіденційні дані користувачів.


```
_id: ObjectId('6835885bdb78b4a811f5f389')
email : "example@gmail.com"
fullName : "example"
password : "$2b$10$AXbRrAYudv6p2tv28Fszeeued8GOYqPXCAtpMMlsjh.SFFLpmnTuG"
profilePic : ""
createdAt : 2025-05-27T09:39:39.356+00:00
updatedAt : 2025-05-27T09:39:39.356+00:00
__v : 0
```

```
_id: ObjectId('68358be2db78b4a811f5f390')
email : "example1@gmail.com"
fullName : "example1"
password : "$2b$10$A.gK3yOWPBwSySZS2jgIN.6HKyQUSoX2J206KSinGA0wEHXbAozuy"
profilePic : "https://res.cloudinary.com/dtcybdnmi/image/upload/v1748339783/vn3rqlzi…"
createdAt : 2025-05-27T09:54:42.312+00:00
updatedAt : 2025-05-27T09:56:24.088+00:00
__v : 0
```


Рисунок 4.16 - Хешовані паролі в базі даних

4.4 Висновки до розділу 4

У даному розділі продемонстровано результати реалізації ключових функціональних можливостей вебзастосунку FlexiChat. Сервіс підтримує повний цикл користувацької взаємодії — від реєстрації та автентифікації до обміну повідомленнями в реальному часі, зміни теми інтерфейсу та перегляду профілю.

Інтерфейс системи реалізовано з урахуванням адаптивності, що дозволяє комфортно користуватися месенджером як на десктопах, так і на мобільних

пристроях. Передбачено просту навігацію, миттєве оновлення чату через WebSocket-з’єднання, а також збереження історії повідомлень у базі MongoDB.

Окрему увагу приділено безпеці: реалізовано захищену аутентифікацію з використанням JWT та збереження паролів у зашифрованому вигляді з використанням bcrypt. Це дозволяє гарантувати конфіденційність та цілісність даних користувачів.

Таким чином, створений вебмесенджер демонструє ефективне поєднання сучасних технологій, безпечної архітектури та зручного користувацького досвіду.

ВИСНОВКИ

Дипломну роботу присвячено створенню інформаційної системи FlexiChat — сучасного вебмесенджера, що забезпечує обмін повідомленнями між користувачами в режимі реального часу. У процесі розробки виконано повний цикл створення програмного забезпечення: від аналізу предметної області до проєктування архітектури, реалізації, тестування та демонстрації результатів.

Під час аналізу виявлено потребу у швидкій, безпечній та зручній комунікації в браузері без встановлення додаткового програмного забезпечення. Проаналізовано наявні рішення на ринку, виявлено їх обмеження щодо адаптивності, відсутності режиму реального часу або складності у розгортанні, що стало основою для обґрунтування потреби в новому застосунку.

Для реалізації системи обрано технології, що відповідають сучасним вимогам до масштабованості та безпеки: React для frontend, Node.js і Express для backend, Socket.IO для реалізації двосторонньої передачі повідомлень у реальному часі, MongoDB для зберігання даних. Інтерфейс застосунку є адаптивним та дозволяє змінювати тему, що підвищує зручність користування на різних пристроях.

Особливу увагу приділено безпеці: реалізовано автентифікацію на основі JWT, збереження токенів у httpOnly cookie, а також хешування паролів через bcrypt. Це дозволяє захистити персональні дані користувачів та контролювати доступ до захищених ресурсів.

Система забезпечує стабільну роботу як у десктопному, так і мобільному середовищі. Реалізовані основні функції: реєстрація, авторизація, обмін повідомленнями, відправлення зображень, зміна теми інтерфейсу, перегляд профілю та історії чатів. Передбачено автоматичне прокручування до нових повідомлень, відображення статусу онлайн і підтримку відфільтрованого перегляду контактів.

Окремо сформульовано системні вимоги до серверної інфраструктури. Для роботи FlexiChat потрібен сервер із Node.js і MongoDB, мінімум 4 ГБ оперативної

пам’яті, підключення до Інтернету з пропускною здатністю не менше 10 Мбіт/с, підтримка захищеного протоколу HTTPS та можливість використання сокет-з'єднань.

Інформаційна система FlexiChat відповідає усім поставленим вимогам щодо функціональності, безпеки та продуктивності. Результати розробки демонструють ефективне використання сучасних вебтехнологій для створення комунікаційної системи, що може бути адаптована як для приватного використання, так і для потреб організацій. Система готова до подальшого розширення — додавання групових чатів, push-нотифікацій, системи ролей та шифрування повідомлень.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Monolithic Architecture. URL: https://www.ibm.com/think/topics/monolithic-architecture (дата звернення: 05.05.2025).

2. Client-Server Architecture. URL: https://www.geeksforgeeks.org/client-server-architecture-system-design/ (дата звернення: 05.05.2025).

3. Microservices Architecture URL: https://www.atlassian.com/microservices-/microservices-architecture (дата звернення: 05.05.2025).

4. AWS. URL: https://aws.amazon.com/lambda/serverless-architectures-learnmore/ (дата звернення: 05.05.2025).

5. API-First. URL: https://swagger.io/resources/articles/adopting-an-api-first-approach/ (дата звернення: 05.05.2025).

6. SPA. URL: https://developer.mozilla.org/en-US/docs/Glossary/SPA (дата звернення: 05.05.2025).

7. JavaScript Frameworks and Libraries MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Frameworks_ libraries (дата звернення: 05.05.2025).

8. React.js Introduction. URL: https://www.w3schools.com/react/react_intro.asp (дата звернення: 05.05.2025).

9. Vue.js Guide. URL: https://vuejs.org/guide/introduction.html (дата звернення: 04.05.2025).

10. Angular Official Documentation. URL: https://angular.io/guide/what-isangular (дата звернення: 05.05.2025).

11. About Node.js. URL: https://nodejs.org/en/about (дата звернення: 05.05.2025).

12. Django Documentation. URL: https://docs.djangoproject.com/en/stable/ (дата звернення: 30.04.2025).

13. Laravel Documentation. URL: https://laravel.com/docs (дата звернення: 1.05.2025).

14. Next.js Documentation. URL: https://nextjs.org/docs (дата звернення: 10.04.2025).

15. Meteor.js Documentation. URL: https://docs.meteor.com/ (дата звернення: 03.05.2025).

16. MySQL Overview Oracle. URL: https://www.oracle.com/mysql/what-ismysql/ (дата звернення: 05.05.2025).

17. SQL vs. NoSQL Databases: What's the Difference? URL: https://www.ibm.com/think/topics/sql-vs-nosql (дата звернення: 03.05.2025).

18. Comparison of relational and non-relational databases. URL: https://www.geeksforgeeks.org/difference-between-relational-and-non-relational-database/ (дата звернення: 04.05.2025).

19. MySQL Oracle. URL: https://dev.mysql.com/doc/refman/8.2/en/what-is.html (дата звернення: 05.05.2025).

20. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 21.04.2025).

21. MongoDB Documentation. URL: https://www.mongodb.com/docs/ (дата звернення: 02.05.2025).

22. Redis Documentation. URL: https://redis.io/docs/ (дата звернення: 20.04.2025).

23. React documentation. URL: https://react.dev/learn (дата звернення: 25.05.2025).

24. Node.js documentation. URL: https://nodejs.org/en/docs (дата звернення: 25.05.2025).

25. Mongoose ODM Guide. URL: https://mongoosejs.com/docs/models.html (дата звернення: 25.05.2025).

26. MongoDB Schema Design. URL: https://www.mongodb.com/basics/schema (дата звернення: 25.05.2025).

27. React documentation – Components. URL: https://react.dev/learn/your-first-component (дата звернення: 25.05.2025).

28. React Router. URL: https://reactrouter.com/en/main (дата звернення: 25.05.2025).

29. Express.js Guide. URL: https://expressjs.com/en/starter/hello-world.html (дата звернення: 25.05.2025).

30. RESTful API Design. URL: https://restfulapi.net/ (дата звернення: 25.05.2025).

31. Socket.IO Documentation. URL: https://socket.io/docs/v4/ (дата звернення: 25.05.2025).

ДОДАТОК А
ЛІСТИНГИ ПРОГРАМ

Лістинг А.1 – Код реалізації компоненту App.jsx

```
const App = () => {
  const { authUser, checkAuth, isCheckingAuth, onlineUsers } =
useAuthStore();
  const { theme } = useThemeStore();

  console.log({ onlineUsers });

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

  console.log({ authUser });

  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;
```

Лістинг А.2 – Код реалізації компоненту ChatContainer.jsx

```
const ChatContainer = () => {
  const {
```

```
    messages,
    getMessages,
    isMessagesLoading,
    selectedUser,
    subscribeToMessages,
    unsubscribeFromMessages,
  } = useChatStore();
  const { authUser } = useAuthStore();
  const messageEndRef = useRef(null);

  useEffect(() => {
    getMessages(selectedUser._id);

    subscribeToMessages();

    return () => unsubscribeFromMessages();
  },     [selectedUser._id,     getMessages,     subscribeToMessages,
unsubscribeFromMessages]);

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

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

  return (
    <div className="flex-1 flex flex-col overflow-auto">
      <ChatHeader />

      <div className="flex-1 overflow-y-auto p-4 space-y-4">
        {messages.map((message) => (
          <div
            key={message._id}
            className={`chat  ${message.senderId  ===  authUser._id  ?
"chat-end" : "chat-start"}`}
            ref={messageEndRef}
          >
            <div className=" chat-image avatar">
              <div className="size-10 rounded-full border">
                <img
                  src={
                    message.senderId === authUser._id
                      ? authUser.profilePic || "/avatar.png"
```

```
                    : selectedUser.profilePic || "/avatar.png"
                }
                alt="profile pic"
              />
            </div>
          </div>
          <div className="chat-header mb-1">
            <time className="text-xs opacity-50 ml-1">
              {formatMessageTime(message.createdAt)}
            </time>
          </div>
          <div className="chat-bubble flex flex-col">
            {message.image && (
              <img
                src={message.image}
                alt="Attachment"
                className="sm:max-w-[200px] rounded-md mb-2"
              />
            )}
            {message.text && <p>{message.text}</p>}
          </div>
        </div>
      ))}
    </div>

    <MessageInput />
  </div>
  );
};
export default ChatContainer;
```

Лістинг А.3 – Код реалізації компоненту MessageInput.jsx

```
const MessageInput = () => {
  const [text, setText] = useState("");
  const [imagePreview, setImagePreview] = useState(null);
  const fileInputRef = useRef(null);
  const { sendMessage } = useChatStore();

  const handleImageChange = (e) => {
    const file = e.target.files[0];
    if (!file.type.startsWith("image/")) {
      toast.error("Please select an image file");
      return;
    }

    if (file.size > 50 * 1024) {
      toast.error("Image size must be less than 50 KB");
      return;
    }

    const reader = new FileReader();
```

```
      reader.onloadend = () => {
        setImagePreview(reader.result);
      };
      reader.readAsDataURL(file);
    };

    const removeImage = () => {
      setImagePreview(null);
      if (fileInputRef.current) fileInputRef.current.value = "";
    };

    const handleSendMessage = async (e) => {
      e.preventDefault();
      if (!text.trim() && !imagePreview) return;

      try {
        await sendMessage({
          text: text.trim(),
          image: imagePreview,
        });

        // Clear form
        setText("");
        setImagePreview(null);
        if (fileInputRef.current) fileInputRef.current.value = "";
      } catch (error) {
        console.error("Failed to send message:", error);
      }
    };

    return (
      <div className="p-4 w-full">
        {imagePreview && (
          <div className="mb-3 flex items-center gap-2">
            <div className="relative">
              <img
                src={imagePreview}
                alt="Preview"
                className="w-20  h-20  object-cover  rounded-lg  border
border-zinc-700"
              />
              <button
                onClick={removeImage}
                className="absolute -top-1.5 -right-1.5 w-5 h-5 rounded-
full bg-base-300
                flex items-center justify-center"
                type="button"
              >
                <X className="size-3" />
              </button>
            </div>
          </div>
        )}
```

```
      <form onSubmit={handleSendMessage} className="flex items-center 
gap-2">
        <div className="flex-1 flex gap-2">
          <input
            type="text"
            className="w-full input input-bordered rounded-lg input-
sm sm:input-md"
            placeholder="Type a message..."
            value={text}
            onChange={(e) => setText(e.target.value)}
          />
          <input
            type="file"
            accept="image/*"
            className="hidden"
            ref={fileInputRef}
            onChange={handleImageChange}
          />

          <button
            type="button"
            className={`hidden sm:flex btn btn-circle
                     ${imagePreview ? "text-emerald-500" : "text-
zinc-400"}`}
            onClick={() => fileInputRef.current?.click()}
          >
            <Image size={20} />
          </button>
        </div>
        <button
          type="submit"
          className="btn btn-sm btn-circle"
          disabled={!text.trim() && !imagePreview}
        >
          <Send size={22} />
        </button>
      </form>
    </div>
  );
};
export default MessageInput;
```

Лістинг А.4 – Код реалізації компоненту Sidebar.jsx

```
const Sidebar = () => {
  const { getUsers, users, selectedUser, setSelectedUser, 
isUsersLoading } = 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 />;
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">
          <Users className="size-6" />
          <span className="font-medium hidden
lg:block">Contacts</span>
        </div>
        {/* TODO: Online filter toggle */}
        <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">({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.name}
                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>

            {/* User info - only visible on larger screens */}
            <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>
        ))}

        {filteredUsers.length === 0 && (
          <div  className="text-center  text-zinc-500  py-4">No  online
users</div>
        )}
      </div>
    </aside>
  );
};
export default Sidebar;
```

Лістинг А.5 – Код реалізації компоненту Navbar.jsx

```
const Navbar = () => {
  const { logout, authUser } = useAuthStore();

  return (
    <header className="bg-base-100 border-b border-base-300 fixed w-
full top-0 z-40 backdrop-blur-lg bg-base-100/80">
       <div className="container mx-auto px-4 h-16">
        <div className="flex items-center justify-between h-full">
          <div className="flex items-center gap-8">
            <Link   to="/"   className="flex   items-center   gap-2.5
hover:opacity-80 transition-all">
              <div  className="size-9  rounded-lg  bg-primary/10  flex
items-center justify-center">
                <MessageSquare className="w-5 h-5 text-primary" />
              </div>
              <h1 className="text-lg font-bold">FlexiChat</h1>
            </Link>
          </div>

          <div className="flex items-center gap-2">
```

```
            <Link
              to={"/settings"}
              className={`btn btn-sm gap-2 transition-colors`}>
              <Settings className="w-4 h-4" />
              <span className="hidden sm:inline">Settings</span>
            </Link>

            {authUser && (
              <>
                <Link to={"/profile"} className={`btn btn-sm gap-2`}>
                  <User className="size-5" />
                  <span className="hidden sm:inline">Profile</span>
                </Link>

                <button    className="flex    gap-2    items-center"
onClick={logout}>
                  <LogOut className="size-5" />
                  <span className="hidden sm:inline">Logout</span>
                </button>
              </>
            )}
          </div>
        </div>
      </div>
    </header>
  )
}

export default Navbar
```

Лістинг А.6 – Код реалізації компоненту SignUpPage.jsx

```
const SignUpPage = () => {
  const [showPassword, setShowPassword] = useState(false);
  const [formData, setFormData] = useState({
    fullName: "",
    email: "",
    password: "",
  });

  const {signup, isSigningUp} = useAuthStore();

  const validateForm = () => {
    if (!formData.fullName.trim()) return toast.error("Full name is
required");
    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;
  };

  const handleSubmit = (e) => {
    e.preventDefault()

    const success = validateForm();

    if(success === true) signup(formData)
  };

  return (
    <div className='min-h-screen grid lg:grid-cols-2'>
      {/* left side */}
      <div className="flex flex-col justify-center items-center p-6 sm:p-12">
        <div className="w-full max-w-md space-y-8">
          {/* LOGO */}
          <div className="text-center mb-8">
            <div className="flex flex-col items-center gap-2 group">
              <div className="size-12 rounded-xl bg-primary/10 flex items-center justify-center group-hover:bg-primary/20 transition-colors">
                <MessageSquare className="size-6 text-primary" />
              </div>
              <h1 className='text-2xl font-bold mt-2'>Create Account</h1>
              <p className='text-base-content/60'>Get started with your free account</p>
            </div>
          </div>

          <form onSubmit={handleSubmit} className='space-y-6'>
            <div className="form-control">
              <label className="label">
                <span className="label-text font-medium">Full Name</span>
              </label>
              <div className="relative">
                <div className="absolute inset-y-0 left-0 pl-3 flex items-center pointer-events-none z-10">
                  <User className="size-5 text-base-content/40" />
                </div>
                <input
                  type="text"
                  className={`input input-bordered w-full pl-10`}
                  placeholder="Harry Potter"
                  value={formData.fullName}
                  onChange={(e) => setFormData({ ...formData, fullName: e.target.value })}
```

```
                />
              </div>
            </div>

            <div className="form-control">
              <label className="label">
                <span className="label-text font-medium">Email</span>
              </label>
              <div className="relative">
                <div className="absolute inset-y-0 left-0 pl-3 flex
items-center pointer-events-none z-10">
                  <Mail className="size-5 text-base-content/40" />
                </div>
                <input
                  type="email"
                  className={`input input-bordered w-full pl-10`}
                  placeholder="you@example.com"
                  value={formData.email}
                  onChange={(e) => setFormData({ ...formData, email:
e.target.value })}
                />
              </div>
            </div>
<div className="form-control">
              <label className="label">
                <span          className="label-text           font-
medium">Password</span>
              </label>
              <div className="relative">
                <div className="absolute inset-y-0 left-0 pl-3 flex
items-center pointer-events-none z-10">
                  <Lock className="size-5 text-base-content/40" />
                </div>
                <input
                  type={showPassword ? "text" : "password"}
                  className={`input input-bordered w-full pl-10`}
                  placeholder="••••••••"
                  value={formData.password}
                  onChange={(e)    =>    setFormData({    ...formData,
password: e.target.value })}
                />
                <button
                  type="button"
                  className="absolute  inset-y-0  right-0  pr-3  flex
items-center"
                  onClick={() => setShowPassword(!showPassword)}
                >
                  {showPassword ? (
                    <EyeOff  className="size-5  text-base-content/40"
/>
                  ) : (
                    <Eye className="size-5 text-base-content/40" />
                  )}
```

```
              </button>
            </div>
          </div>

          <button type="submit" className="btn btn-primary w-full"
disabled={isSigningUp}>
            {isSigningUp ? (
              <>
                <Loader2 className="size-5 animate-spin" />
                Loading...
              </>
            ) : (
              "Create Account"
            )}
          </button>

        </form>

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

    {/* right side */}

    <AuthImagePattern
      title="Join our community"
      subtitle="Connect with friends, share moments, and stay in
touch with your loved ones."
    />

  </div>
 )
}

export default SignUpPage
```

Лістинг А.7 – Код реалізації компоненту auth.controller.jsx

```
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 credantials"})
    }

    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 {
    console.log("Error in logout controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
};

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

    if(!profilePic){
      return res.status(400).json({ message: "Profile pic is 
required"});
    }
    const uploadResponse = await 
cloudinary.uploader.upload(profilePic)
    const updatedUser = await User.findByIdAndUpdate(userId, 
{profilePic:uploadResponse.secure_url}, {new:true})

    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" })
  }
};
```

ДИПЛОМНА РОБОТА
РОЗРОБКА ВЕБСИСТЕМИ МЕСЕНДЖЕРА НА ОСНОВІ БІБЛІОТЕКИ REACTJS

Розробив: ст. гр. КНТ-521 І.В. Кравченко
Керівник: доцент, к.т.н. С.Ю. Скрупський

Національний Університет
«Запорізька Політехніка»

Актуальність

У сучасному цифровому світі зростає потреба в швидкому, зручному та безпечному обміні повідомленнями.

Вебмесенджери стали важливими інструментами для навчання, бізнесу та повсякденного спілкування.

Користувачі очікують миттєвої доставки повідомлень, підтримки на різних пристроях і захисту персональних даних.

Національний Університет
«Запорізька Політехніка»

Мета

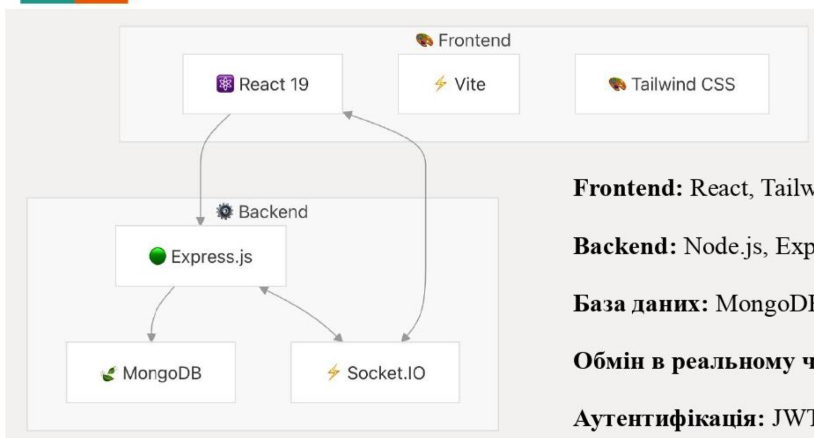
Метою є створення повноцінної вебсистеми обміну повідомленнями в реальному часі, що включає:

- фронтенд-застосунок із простим та адаптивним інтерфейсом,
- бекенд-сервер з підтримкою безпечної автентифікації користувачів,
- інтеграцію для збереження історії чатів,
- масштабовану архітектуру для підтримки зростання кількості користувачів.

Система реалізується під умовною назвою FlexiChat і призначена для демонстрації сучасних підходів до побудови інтерактивних вебзастосунків.

Національний Університет
«Запорізька Політехніка»

Технології, що використовувались



Frontend: React, Tailwind CSS, DaisyUI;

Backend: Node.js, Express;

База даних: MongoDB;

Обмін в реальному часі: Socket.IO;

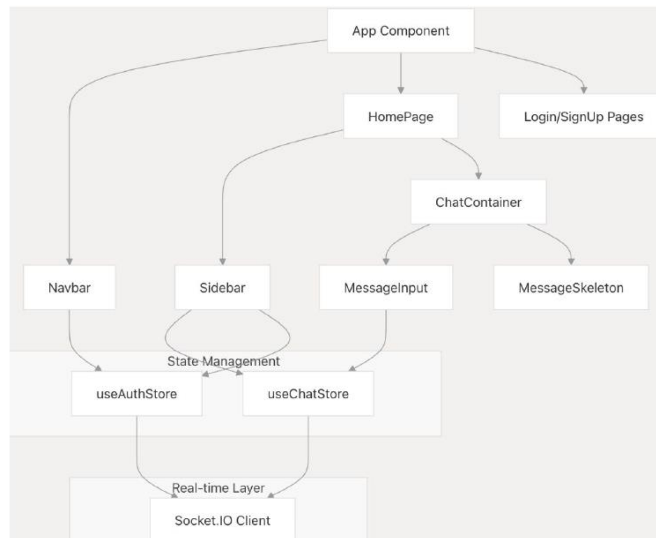
Аутентифікація: JWT, bcrypt, httpOnly cookie.

Національний Університет
«Запорізька Політехніка»

Діаграма варіантів використання системи

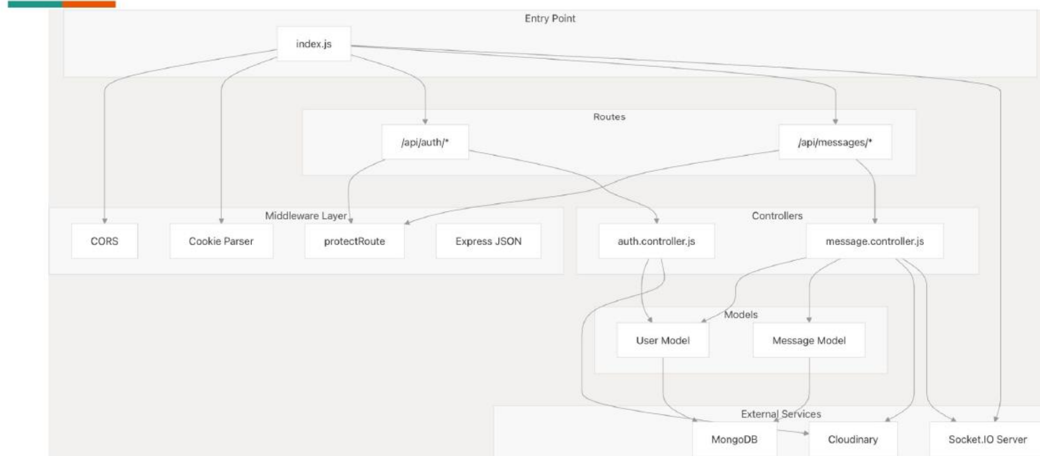
Національний Університет
«Запорізька Політехніка»

Діаграма структури клієнтської частини

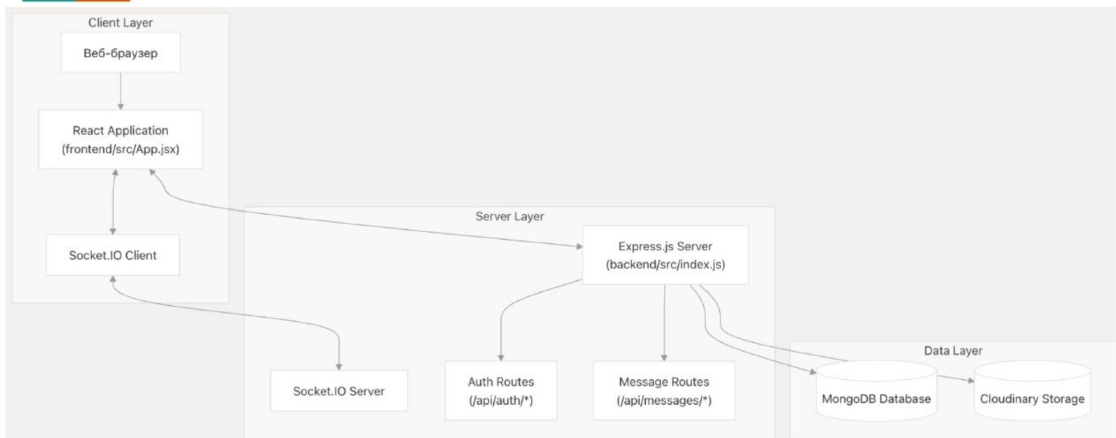


Національний Університет
«Запорізька Політехніка»

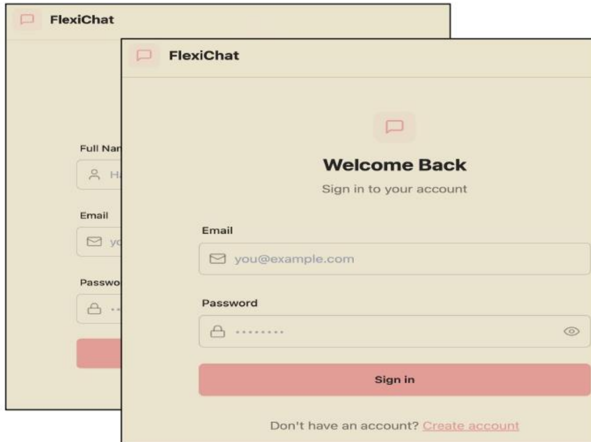
Діаграма структури серверної частини

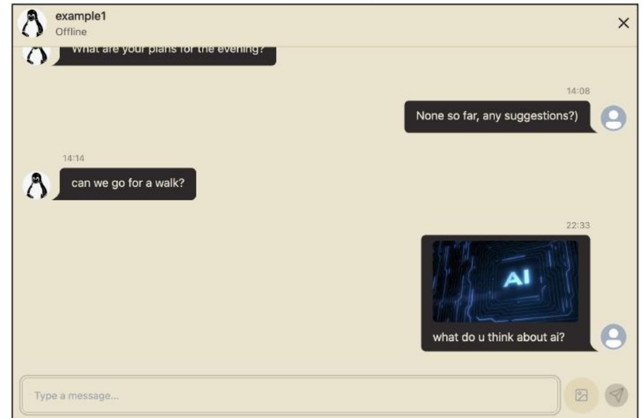


Архітектура системи



Основні функції месенджера



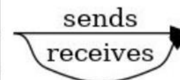


Національний Університет
«Запорізька Політехніка»

База даних та модель даних (ER-діаграма)

Було обрано базу
даних MongoDB із :
причини документі
у форматі JSON, як
добре синергують з
великим потоком
непреривних даних
месенджері..

User	
_id	ObjectId (PK)
email	String (UK)
fullName	String
password	String
profilePic	String
createdAt	Date
updatedAt	Date



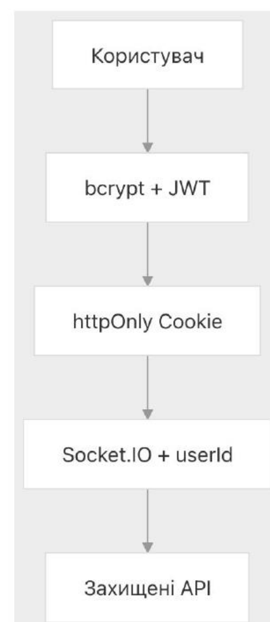
Message	
_id	ObjectId (PK)
senderId	ObjectId (FK)
receiverId	ObjectId (FK)
text	String
image	String
createdAt	Date
updatedAt	Date

- Оптимізована для швидкого читання й запису, що критично для чатів у реальному часі
- Підтримка горизонтального масштабування при збільшенні навантаження
- Чудово працює з Node.js та екосистемою JavaScript

Національний Університет
«Запорізька Політехніка»

Безпека та захист даних

- **JWT** – токени для авторизації користувача. Використовуються для підтвердження особи користувача при зверненнях до серверу.
- **httpOnly cookie** – захист токена від XSS-атак. Зберігає JWT у недоступному для JavaScript cookie, що унеможливлює крадіжку токена зі сторінки.
- **bcrypt** – хешування паролів, стійке до зломів. Забезпечує зберігання паролів у вигляді хешів, що неможливо розшифрувати навіть при доступі до бази.
- **Socket.IO авторизація** – перевірка токена при з'єднанні в реальному часі. Дозволяє приймати повідомлення тільки від авторизованих користувачів у WebSocket-з'єднанні.



Національний Університет
«Запорізька Політехніка»

Висновки

- FlexiChat реалізовано як повноцінну систему обміну повідомленнями в реальному часі;
- Сервіс підтримує реєстрацію, авторизацію, обмін текстом і зображеннями, а також адаптивний інтерфейс;
- Використано сучасний стек технологій та забезпечено високий рівень безпеки;
- Система протестована, стабільно працює й готова до використання.

Національний Університет
«Запорізька Політехніка»