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

Факультет інформаційних технологій
(факультет)

Кафедра Інформаційних технологій та комп'ютерної інженерії
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційна робота ступеня бакалавра
(бакалавра, спеціаліста, магістра)

Здобувача Дрозда Іллі Олеговича
(ПІБ)

академічної групи 126-21-2
(шифр)

спеціальності 126 «Інформаційні системи та технології»
(код і назва спеціальності)

за освітньо-професійною програмою «Інформаційні системи та технології»
(офіційна назва)

на тему Розробка full-stack месенджеру з використанням сучасних вебтехнологій
(назва за наказом ректора)

Керівник	**Прізвище, ініціали**	**Оцінка за шкалою**	
		рейтинговою	**інституційною**
кваліфікаційної роботи	доц. Соколова Н. О.		
розділів: 3			
Рецензент			
Нормоконтролер	проф. Коротенко Г.М.		

Дніпро
2025

ЗАТВЕРДЖЕНО:

завідувач кафедри

інформаційних технологій
та комп'ютерної інженерії
(повна назва)

_____ Гнатушенко В.В,
(підпис) (прізвище)

«___» _____ 2025 року

ЗАВДАННЯ
на кваліфікаційну роботу
ступеня бакалавра
(бакалавра, спеціаліста, магістра)

здобувачу Дрозд І.О._____ **академічної групи** 126-21-2
(прізвище та ініціали) (шифр)

спеціальності 126 «Інформаційні системи та технології»

за освітньо-професійною програмою _____

на тему Розробка full-stack месенджеру з використанням сучасних вебтехнологій

затверджену наказом ректора НТУ «Дніпровська політехніка» від 05.05.2025 № 336

Розділ	Зміст	Термін виконання
Розділ 1	1. Огляд існуючих рішень. 2. Аналіз галузі	17.04.2025- 03.06.2025
Розділ 2	1. Дизайн бази даних. 2. Дизайн restful API. 3. Дизайн web-socket.	17.04.2025- 03.06.2025
Розділ 3	1. E2E тестування 2. Ручне тестування застосунку	17.05.2025- 03.06.2025

Завдання видано _____ Соколова Н. О.
(підпис керівника) (прізвище, ініціали)

Дата видачі 03.02.2025

Дата подання до екзаменаційної комісії _____

Прийнято до виконання _____ Дрозд І. О.
(підпис студента) (прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 62 сторінка, 29 рисунків, 9 додатків, 10 джерел.

Тема дипломної роботи: «Розробка full-stack месенджеру з використанням сучасних веб технологій»

Предмет дослідження: архітектурні підходи , принципи розробки та інструментальні засоби, що використовуються для створення веб-додатку.

Об'єкт дослідження: процес розробки full-stack веб-додатків для обміну повідомленнями з використанням сучасних технологічних рішень та методологій.

Об’єкт розроблення: серверна частина веб-чату.

Мета кваліфікаційної роботи: проектування та розробка повнофункціонального full-stack (програмний застосунок, що охоплює як фронтенд, так і бекенд в єдиній кодовій базі) веб-додатку (месенджера), що демонструє застосування сучасних методів, технологій та принципів розробки програмного забезпечення для створення систем обміну повідомленнями в реальному часі.

Ключові слова: Backend, БАЗА ДАНИХ, RESTFUL API, WEB-SOCKET, TESTING.

ABSTRACT

Explanatory note: 62 pages, 29 figures, 9 appendix, 10 sources.

Topic of the diploma work: "Development of a full-stack messenger using modern web technologies"

Subject of research: Architectural approaches , technologies , development principles and tools used for creating the web application.

Object of research: The process of developing full-stack web applications for messaging using modern technological solutions and methodologies.

Development object: server side of web chat.

Goal of the qualification work: Designing and developing a fully functional full-stack web application (messenger), where full-stack refers to a software application that includes both front-end and back-end in a single codebase, to demonstrate the application of modern methods, technologies, and software development principles for creating real-time messaging systems.

Keywords: NESTJS, DATABASE, RESTFUL API, WEBSOCKET, TESTING.

ЗМІСТ

ВСТУП

В епоху цифрової трансформації та глобалізації комунікаційні технології відіграють ключову роль у повсякденному житті та професійній діяльності. Веб-додатки для обміну миттєвими повідомленнями (месенджери) стали невід'ємною частиною інтернет-простору, забезпечуючи швидкий, зручний та ефективний спосіб спілкування. Актуальність розробки нових та вдосконалення існуючих месенджерів зумовлена постійно зростаючими вимогами користувачів до функціональності, безпеки, швидкодії та доступності таких систем.

Метою даної кваліфікаційної роботи є розробка full-stack веб-додатку-мессенджера. Проект передбачає створення сучасної платформи для обміну повідомленнями, що включає реєстрацію та автентифікацію користувачів, створення індивідуальних та групових чатів, обмін текстовими повідомленнями в реальному часі, а також управління профілями користувачів, включаючи можливість завантаження медіафайлів. Для реалізації проекту використовуються сучасні технології та підходи, зокрема фреймворк Nest.js для серверної частини на Node.js, бібліотека React для клієнтської частини, система управління базами даних PostgreSQL для зберігання даних, а також технологія WebSocket для забезпечення комунікації в реальному часі.

У роботі детально розглядаються етапи аналізу предметної області, проектування архітектури системи, вибору технологічного стеку, розробки основного функціоналу, тестування та розгортання додатку. Особлива увага приділяється застосуванню сучасних принципів розробки програмного

забезпечення, таких як модульність, масштабованість, безпека та підтримка коду.

Результат роботи: функціональний прототип веб-додатку-месенджера з реалізованими функціями реєстрації та аутентифікації користувачів, управління профілем (включаючи завантаження фотографій на Google Drive), створення чатів та додаванням/видаленням учасників, обміну повідомленнями в реальному часі. Також результатом є розроблена серверна частина з RESTful API, WebSocket-сервером, інтеграцією з PostgreSQL та e2e тестами для ключових компонентів.

РОЗДІЛ 1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ПОСТАНОВКА ЗАВДАНЬ ДОСЛІДЖЕННЯ

У даному розділі проводиться дослідження предметної області розробки веб-месенджерів, аналіз існуючих рішень та обґрунтування вибору технологічного стеку для реалізації проекту. Визначаються ключові функціональні та нефункціональні вимоги, а також формулюються конкретні завдання бакалаврської роботи.

1.1 Огляд сучасних веб-застосунків для обміну повідомленнями

Веб-застосунки для обміну повідомленнями (месенджери) є невід'ємною частиною сучасного цифрового простору, забезпечуючи миттєвий зв'язок. Розуміння їхньої класифікації, еволюції та вимог користувачів є важливим для розробки нового продукту.

1.1.1 Визначення та класифікація месенджерів

Месенджер – це програмне забезпечення для обміну повідомленнями та файлами через Інтернет у реальному часі або асинхронно. Класифікація месенджерів здійснюється за різними критеріями:

- **Цільова аудиторія та призначення:**
    - Соціальні (особисті): Для неформального спілкування (WhatsApp, Facebook Messenger);
    - Корпоративні (командні): Для бізнес-комунікацій та співпраці

(Slack, Microsoft Teams);

- Для підтримки клієнтів: Для взаємодії бізнесу з клієнтами (Intercom, Zendesk Chat).

- **Ключовий функціонал:**
  - Текстовий чат;
  - голосові/відеодзвінки;
  - обмін файлами;
  - групові чати;
  - статуси.

- **Типи шифрування:**
  - Наскрізне шифрування (E2EE) для максимальної конфіденційності (Signal, WhatsApp);
  - Транспортне шифрування.

Навіть у соціальних месенджерах зростає потреба в контролі над приватністю даних. Спостерігається конвергенція функціоналу між різними типами месенджерів, що вказує на очікування користувачами універсальних рішень.

1.1.2 Еволюція та поточні тенденції ринку месенджерів

Месенджери еволюціонували від простих текстових чатів (ICQ) до багатофункціональних мобільних платформ (WhatsApp, Telegram). Ключові тенденції включають зростання використання мобільних каналів, посилення персоналізації та забезпечення безшовного досвіду на всіх каналах. Наявність

веб-версій у популярних месенджерів підтверджує актуальність розробки full-stack веб-застосунку.

1.1.3 Ключові очікування користувачів від сучасних месенджерів

Сучасні користувачі висувають високі вимоги до месенджерів [1]:

- **Функціональність:** Миттєвий обмін повідомленнями, групові чати, голосові/відеодзвінки, обмін файлами, інтеграції;
- **Безпека та приватність:** Наскрізне шифрування (E2EE), контроль над даними, двофакторна аутентифікація (2FA), прозорість політик. Існує сприйняття компромісу між функціоналом та приватністю, тому новий месенджер повинен прагнути до балансу;
- **Продуктивність:** Швидка доставка повідомлень, низька затримка дзвінків, ефективне використання ресурсів;
- **Надійність:** Стабільна робота, збереження історії, прозорість політик компанії.

1.2 Аналіз існуючих популярних месенджерів

Аналіз популярних месенджерів дозволяє ідентифікувати їхні сильні та слабкі сторони, а також визначити незадоволені потреби користувачів.

1.2.1 Критерії для порівняльного аналізу

Для об'єктивного аналізу використовуються такі критерії:

- Функціонал;
- Архітектурний підхід (клієнт-серверна, P2P), тип (монолітна,

мікросервісна), протоколи;

- Мови, фреймворки, СУБД (за наявності інформації);
- Модель шифрування, протоколи, 2FA, політики даних;
- Доступність на різних ОС та пристроях.

Ці критерії відображають сучасні пріоритети, де безпека та архітектура є не менш важливими, ніж функціонал.

1.2.2 Огляд WhatsApp

WhatsApp – один з найпопулярніших месенджерів рис. 1.1 .

- **Ключовий функціонал:** Текстові повідомлення, голосові/відеодзвінки (включаючи групові, демонстрацію екрану), статуси, обмін файлами, E2EE за замовчуванням, WhatsApp Business;
- **Архітектура:** Клієнт-серверна. Для E2EE – протокол Signal;
- **Обмеження:** Обмежене сховище, високе використання даних, відсутність інтеграції з CRM, обмежена підтримка, прив'язка до одного номера, вразливість до злому, поширення дезінформації, недоліки в безпеці метаданих та підтримці.

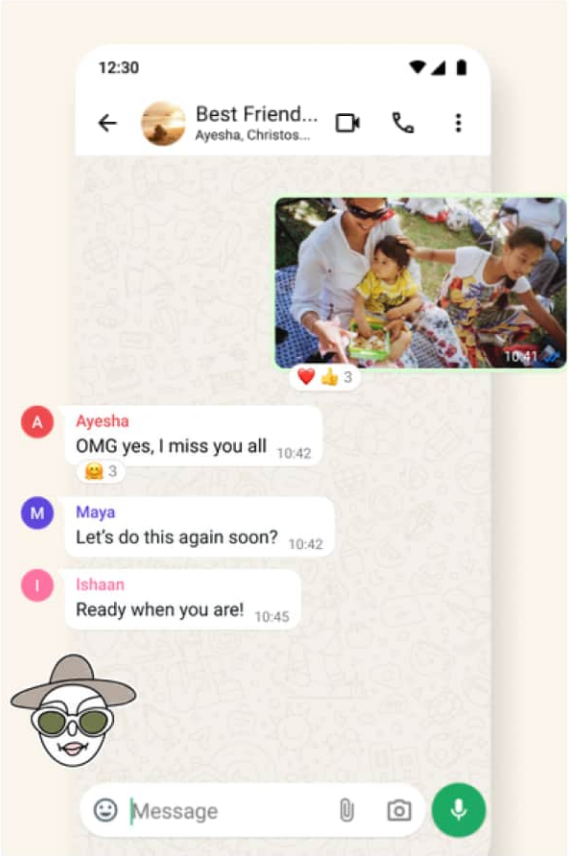


Рисунок 1.1 - огляд інтерфейсу WhatsApp

1.2.3 Огляд Telegram

Telegram відомий функціоналом та швидкістю, але має питання щодо безпеки рис. 1.2.

- **Функціонал:** Великі групи (до 200 000), канали, Instant View, боти, Live Locations, Telegram Passport, постійні конференц-дзвінки, обмін файлами, зникаючі повідомлення (в секретних чатах);
- **Архітектура:** Розподілені сервери. Власний протокол MTProto для зв'язку. Хмарні чати шифруються на сервері (Telegram має доступ до

ключів). E2EE лише для секретних чатів та дзвінків. Збирає IP-адреси, метадані;

- **Переваги:** E2EE для секретних чатів/дзвінків, відкритий код клієнтів, зникаючі повідомлення, мультиплатформність, великі групи/канали, швидке видалення даних;
- **Недоліки:** Реєстрація за номером телефону, E2EE не за замовчуванням, закритий серверний код, збір метаданих, можлива співпраця з урядами. Важлива прозорість політик безпеки.

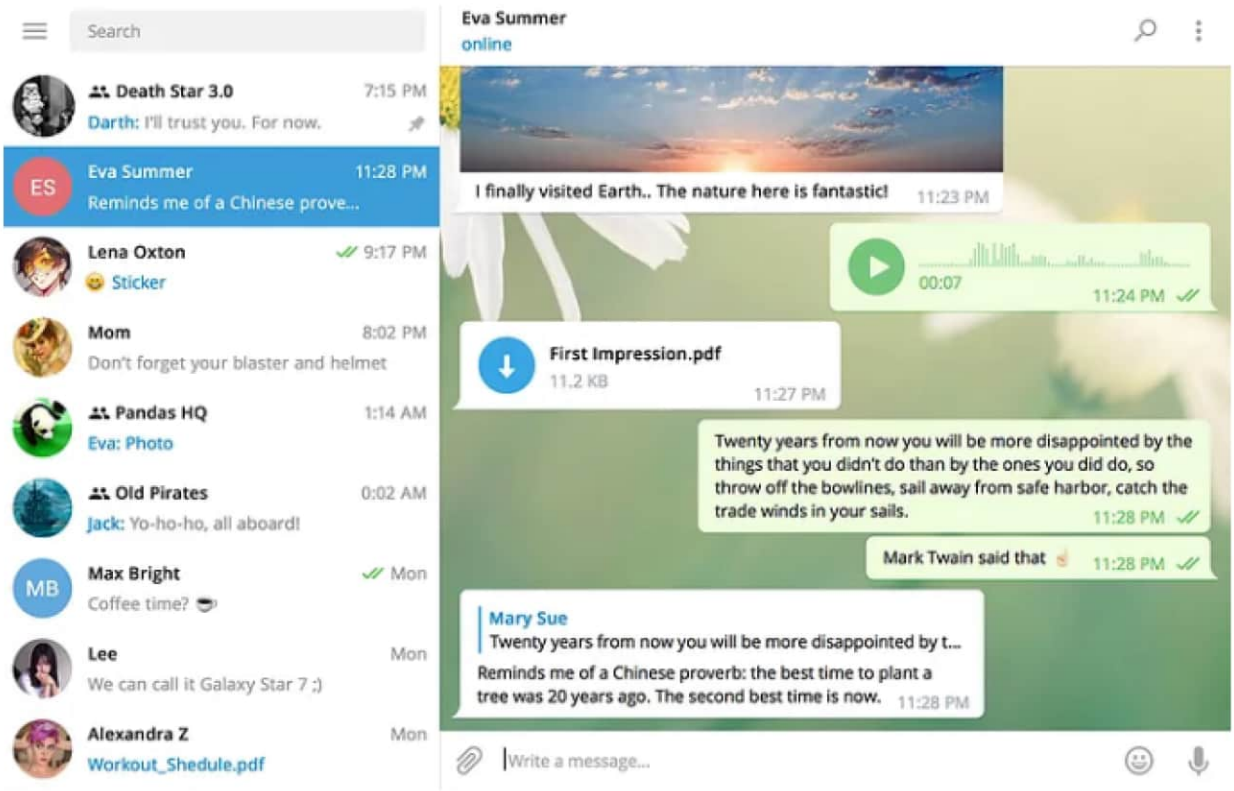


Рисунок 1.2 - Огляд дизайну telegram

1.2.4 Огляд Signal: функціональні можливості, архітектурні особливості, акцент на безпеці, сильні та слабкі сторони

Signal ставить приватність та безпеку на перше місце:

- **Функціонал:** E2EE за замовчуванням для всього, відсутність реклами/трекерів, мультиплатформність, зникаючі повідомлення, великі групи (до 1000);
- **Архітектура:** Централізована. Протокол Signal Protocol. Сервери зберігають мінімум метаданих. Дзвінки P2P (або через сервер для приховування IP). Клієнтський код відкритий, серверний – переважно;
- **Переваги:** Високий рівень безпеки/приватності, відкритий код, без реклами, безкоштовний;
- **Недоліки:** Реєстрація за номером телефону (частково вирішено іменами користувачів), вразливість до компрометації пристрою, можливе блокування, менша база користувачів. Відкритість коду є фундаментальним фактором довіри.

1.2.5 Огляд корпоративних месенджерів

Корпоративні месенджери демонструють тенденції у співпраці та управлінні спільнотами таблиця 1.1:

- **Slack:**
    - **Функціонал:** Канали, прямі повідомлення, Slack Connect, Huddles (дзвінки), Workflows, інтеграції, треди, пошук, Slack AI;
    - **Архітектура:** Клієнт-серверна, мікросервісна на AWS. Webapp сервери, real-time message сервери (WebSockets), шардування даних;
    - **Переваги:** Багатомовність, інтуїтивність, інтеграції, безпека;

- **Недоліки:** Вартість, обмеження безкоштовного плану, потенційні ризики даних.

- **Discord:**
  - **Функціонал:** Сервери/канали, голосові/відеочати, ролі, боти;
  - **Архітектура:** Клієнт-серверна. WebRTC для аудіо/відео. Бекенд на Elixir, Go. Мікросервіси, WebSockets;
  - **Переваги:** Гнучкість, якісний голосовий зв'язок, великі спільноти;
  - **Недоліки:** Крива навчання, обмеження безкоштовної версії, продуктивність, безпека, приватність. Мікросервісна архітектура та WebSockets/WebRTC є галузевими стандартами.

Таблиця 1.1 - Порівняльна таблиця популярних месенджерів

Критерій	WhatsApp	Telegram	Signal	Slack	Discord
Основний функціонал	Текст, голос/відео, групи, статуси, файли, E2EE [3]	Текст, голос/відео, великі групи/канали, боти, файли, секретні чати E2EE [4]	Текст, голос/відео, групи, файли, E2EE за замовчуванням, зникаючі повідомлення [2]	Канали, треди, інтеграції, Huddles, файли, Workflows [5]	Сервери/канали, текст, голос/відео, ролі, боти, файли [7]
Модель безпеки	E2EE (Signal Protocol) [3]	E2EE (MTProto) для секретних чатів/дзвінків; хмарні – шифр. на сервері [4]	E2EE (Signal Protocol) за замовчуванням [2]	E2EE (частково), 2FA, SSO [5]	Шифрування в транзиті; E2EE відсутнє для чатів
Збір метаданих	Значний (Meta)	Так (IP, пристрої) [4]	Мінімальний	Так (аналітика)	Так (аналітика)
Підтримка	iOS, Android,	iOS, Android,	iOS, Android,	iOS, Android,	iOS, Android,

платформ	Web, Desktop [3]	Web, Desktop, macOS, Linux [4]	Web, Desktop, macOS, Linux [2]	Web, Desktop, macOS, Linux [5]	Web, Desktop, macOS, Linux [7]
Переваги	Велика база, E2EE, безкоштовний, бізнес-функції [3]	Швидкість, великі групи/канали, боти [4]	Безпека/приватність, відкритий код, без реклами [2]	Інтеграції, автоматизація, командна робота [5]	Якісний голос, гнучкість спільнот, безкоштовний [7]
Недоліки	Приватність (Meta), підтримка, вразливості [3]	E2EE не за замовчуванням, збір метаданих, закритий сервер [4]	Менша база, реєстрація за номером (частково вирішено)	Вартість, обмеження free-плану [5]	Продуктивність, безпека, модерація, приватність

1.3 Огляд архітектурних підходів

- **Дворівнева (2-tier):** Клієнт (UI) та сервер (БД);
- **Трирівнева (3-tier):** Клієнт (Presentation), сервер застосунків (Application/Logic), сервер БД (Data). Забезпечує кращий розподіл відповідальності та масштабованість. Більшість месенджерів (WhatsApp, Discord, Signal, Slack) використовують централізовану клієнт-серверну інфраструктуру.

1.4 Монолітна архітектура vs. Мікросервісна архітектура

- **Монолітна:** Всі компоненти в єдиній кодовій базі;
    - *Переваги:* Простіша розробка на старті, швидший запуск MVP;

- *Недоліки:* Складність масштабування, повільне розгортання, низька відмовостійкість.

- **Мікросервісна:** Програма розбита на незалежні сервіси;
  - *Переваги:* Висока масштабованість, відмовостійкість, гнучкість технологій, незалежне розгортання;
  - *Недоліки:* Вища складність на старті, потреба в оркестрації, можливі мережеві затримки.

    Для месенджерів з потенціалом зростання мікросервісна архітектура є перспективнішою. [6].

1.5 Принципи модульного дизайну

Модульний дизайн – декомпозиція системи на незалежні модулі з чіткими інтерфейсами.

Основні принципи модульного дизайну:

- Декомпозиція;
- Інкапсуляція;
- Слабке зв'язування;
- Висока зв'язність.

Перевагами є покращена супроводжуваність, масштабованість, повторне використання коду, спрощене тестування, прискорення розробки. Модульність є основою для мікросервісної архітектури.

1.6 Технології для реалізації серверної частини

- **NestJS (Node.js/TypeScript):** Прогресивний фреймворк, натхненний

Angular. Переваги: масштабованість для I/O (Node.js); вбудовані WebSockets; TypeScript; модульність. [8] Недоліки: крива навчання; можливий оверхед.

- **Spring Boot (Java):** Потужний фреймворк для enterprise. Переваги: надійність; підтримка CPU-bound завдань; велика екосистема. Недоліки: вище споживання ресурсів; крута крива навчання.
- **Express.js (Node.js/JavaScript):** Мінімалістичний, гнучкий. Переваги: легкість; швидкість для I/O; гнучкість. Недоліки: відсутність структури; ризик неконсистентності.

  Для месенджера Nest.js є доцільним вибором через I/O-ефективність Node.js, TypeScript та вбудовану підтримку WebSockets. [8].
- **Переваги використання TypeScript у розробці:** Статична типізація; покращена якість коду; читабельність; супроводжуваність; продуктивність розробників; краща співпраця; підтримка сучасних можливостей JS; сильна екосистема. Виклики: Додатковий крок компіляції; крива навчання; можлива складність типів у великих проектах; інтеграція з існуючим JS. TypeScript підвищує надійність та якість, що критично для месенджерів.

1.7 Технології для реалізації клієнтської частини

- **React.js (бібліотека):** Розроблена Facebook, фокусується на view layer, компонентний підхід, віртуальний DOM. Переваги: висока продуктивність; компоненти; велика екосистема; гнучкість. Недоліки: крива навчання JSX; не є повним фреймворком.

- **Angular (фреймворк):** Комплексний фреймворк від Google, TypeScript "з коробки". Переваги: структурованість; TypeScript; повний набір інструментів. Недоліки: найкрутіша крива навчання; можливі проблеми з продуктивністю порівняно з React/Vue.
- **Vue.js (прогресивний фреймворк):** Легкий, поєднує найкраще з React та Angular. Переваги: легкість вивчення; висока продуктивність; гнучкість. Недоліки: менша екосистема.

  Для месенджера, де важливі швидке оновлення DOM та компоненти, React.js є доцільним вибором завдяки продуктивності та великій екосистемі.

1.8 Системи управління базами даних

- **Реляційні СУБД (SQL):** Структуровані таблиці, SQL, ACID-транзакції;
  - **PostgreSQL:** Потужна, об'єктно-реляційна, підтримує JSON/JSONB, повнотекстовий пошук, надійна. [9] Недоліки: ресурсоємність; крива навчання.
  - **MySQL:** Популярна, хороша продуктивність для читання. Недоліки: менша гнучкість схеми; складніше горизонтальне масштабування.
- **NoSQL СУБД (напр., MongoDB):** Гнучкі схеми, оптимізовані для масштабування;
  - **MongoDB:** Документо-орієнтована (BSON), гнучка схема, горизонтальне масштабування. Недоліки: менш строга цілісність даних; складніші JOIN'и.

Отже, для розробки мессенджера обрано PostgreSQL, що забезпечує цілісність даних завдяки (ACID) та гнучкість через JSONB для повідомлень. [9].

1.9 Забезпечення комунікації в реальному часі

- **WebSockets:** Повнодуплексний канал через одне TCP-з'єднання. [10] Переваги: низька затримка; двосторонній зв'язок; ефективність. [10] Недоліки: складність управління з'єднаннями; можливе блокування; навантаження на сервер.
- **Long Polling:** Імітація push через HTTP. Переваги: широка підтримка. Недоліки: вища затримка; більший оверхед. [10].
- **Server-Sent Events (SSE):** Односторонній потік від сервера до клієнта через HTTP. Переваги: простота; авто перепідключення. Недоліки: односторонній зв'язок (не підходить для чату). [10].

  Отже, WebSocket є стандартом для чат-додатків

1.10 Зберігання файлів

Вибір Google Drive API для фотографій профілю є компромісом між зручністю для користувача та необхідністю ретельного управління безпекою та правами доступу.

- **Хмарні сховища об'єктів (AWS S3):** Масштабоване, надійне сховище для неструктурованих даних. Переваги: масштабованість; доступність;

економічність; безпека. Недоліки: складніший API; витрати на трафік. Рекомендовано для медіафайлів.

- **Google Drive API:** Дозволяє взаємодіяти зі сховищем Google Drive користувачів. Переваги: зручність для користувачів; інтеграція з Google. Недоліки: не призначене для основного сховища програми; обмеження квот; управління OAuth. Планується використання Google Drive API для фотографій профілю, що вимагатиме ретельного налаштування OAuth з мінімально необхідними правами доступу.
- **Локальне сховище:** Зберігання файлів на тому ж сервері, що й застосунок. Переваги: простота налаштування для малих проектів. Недоліками є проблеми з масштабуванням, резервним копіюванням, безпекою, доступністю. Не рекомендується для production-систем.

1.11 Висновки до першого розділу

У першому розділі проведено всебічний аналіз предметної області розробки веб-месенджерів, що є важливим етапом для визначення основних напрямів дослідження та розробки. Було здійснено огляд сучасних месенджерів, таких як WhatsApp, Telegram, Signal, Slack та Discord, з акцентом на їх функціональні можливості, архітектурні особливості, а також сильні та слабкі сторони, що дозволило виявити незадоволені потреби користувачів, зокрема у сфері безпеки, продуктивності та зручності. Ключові очікування користувачів, такі як миттєвий обмін повідомленнями, наскрізне шифрування та стабільність роботи, стали основою для формування вимог до нового месенджера. Також було обґрунтовано вибір технологічного стеку,

який включає Nest.js для серверної частини, React для клієнтської, PostgreSQL для зберігання даних та WebSocket для комунікації в реальному часі, з урахуванням їхньої ефективності та відповідності принципам модульного дизайну і масштабованості.

Цей розділ створив теоретичну базу для подальшого проектування системи, чітко окресливши завдання дослідження та розробки, що сприяють досягненню мети створення повнофункціонального веб-додатку.

1.12 Постановка задачі

Метою даної кваліфікаційної роботи є розробка веб застосунку, що дозволяє обмінюватися повідомленнями в реальному часі, реєструватися, шукати інших користувачів, редагувати свій профіль, дивитися профіль інших користувачів, додавати їх в чати.

У ході дослідження були поставлені та виконані наступні задачі:

1. Спроєктована та розроблена база даних в Postgresql, що дозволяє зберігати дані про користувачів, чати, учасників чатів та їх повідомлення;
2. Був Розроблений дизайн Restful API, Websocket та реалізований використовуючи фреймворк Nest.js;
3. Інтегрували розроблений backend сервер з базою даних та клієнтом;
4. Розробили e2e тести для покриття ключового функціоналу системи; Проведено загальне ручне тестування програми, для перевірки працездатності застосунку.

РОЗДІЛ 2 ПРОЄКТНІ РІШЕННЯ

2.1 Архітектура Nest.js проєкту

Nest.js - це сучасний node.js фреймворк, який дуже стрімко набирає популярність створенні backend додатків, особливо серед крупних enterprise компаній.

Він має широкий набір встроєних інструментів “з коробки”, що кардинально прискорює процес розробки, тестування та відповідно випуску коду у production.

2.1.1 Ключові відмінності Nest.js

Ключовими відмінностями є:

- **Modularity** - Застосунок має єдину вхідну точку, наприклад index.ts, що задає базове налаштування серверу, підключення різних бібліотек, тощо.

  Уся логіка, окрема конфігурація модуля, слої застосунку, тощо декомпозуються в окремі модулі. Створивши окрему директорію та файл *.module.ts, де описується використані компоненти, що в контексті модуля поділяються на (imports, controllers, providers, exports). Імпортовані посилання на ці компоненти збираються в єдиний об’єкт модуля відповідно;

- **Dependency Injection** (DI) - це паттерн, де необхідні компоненти застосунку створюються не в самому компоненті, а зовні, як правило вказавши залежність в модулі компонента Nest.js

автоматично встановить необхідну залежність в коді.

Даний паттерн дозволяє легко тестувати застосунок, наприклад підмінивши слой відповідний за взаємодію з СУБД, на імітуючий (mocked), тим самим прискорює тестування та уникає проблем з втрачаням даних при помилках, особливо при великому наборі unit-тестів;

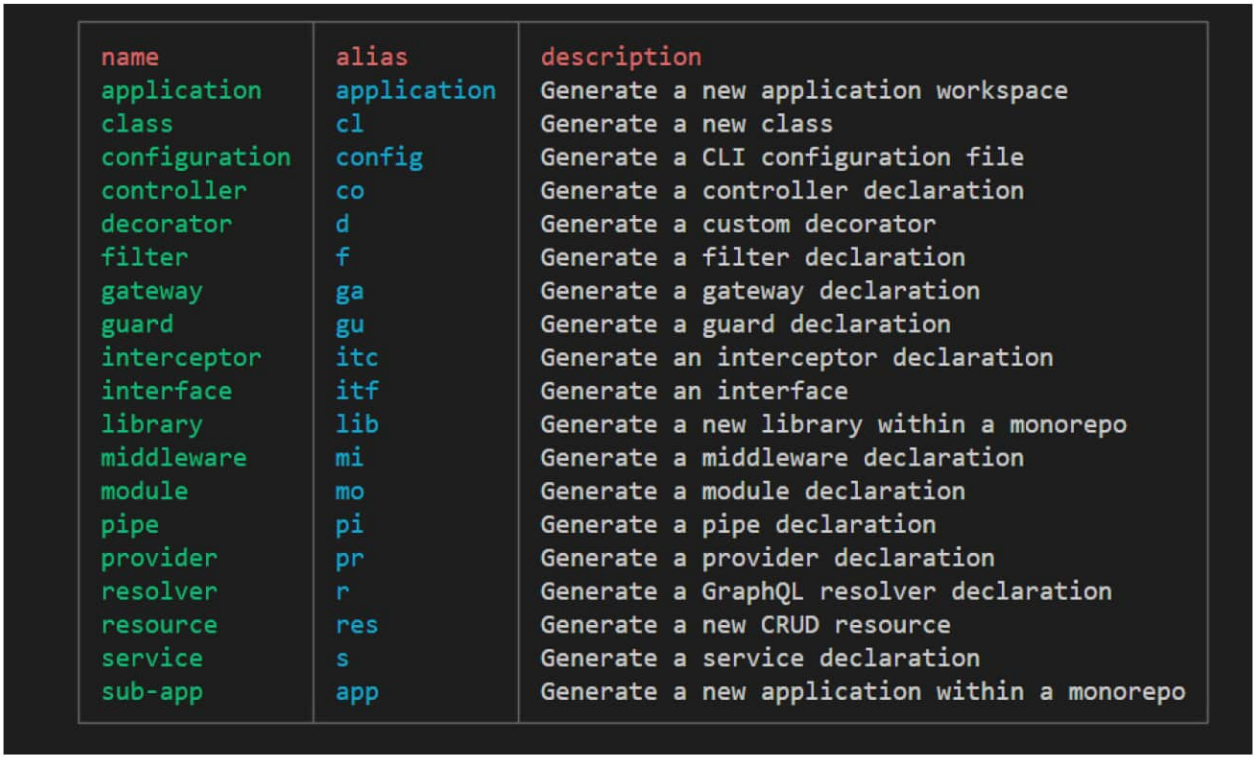
- **TypeScript -** Nest.js побудований на TypeScript, що забезпечує строго типізоване середовище. Це підвищує якість коду, полегшує його супровід і є особливо корисним для великих команд та складних застосунків. Типізація допомагає виявляти помилки на етапі розробки та підтримує консистентність між модулями;
- **Гнучкість транспортних -** Nest.js підтримує різні транспортні шари: HTTP, WebSockets та мікросервіси. Це дозволяє розробникам обирати оптимальний спосіб комунікації для різних частин застосунку, що є критично важливим для enterprise-проєктів із різноманітними точками інтеграції;
- **Декларативний підхід із декораторами -** У Nest.js широко використовуються декоратори для спрощення оголошення маршрутів, middleware та інших компонентів. Цей підхід робить код більш читабельним і легким у підтримці, що особливо цінно для великих кодових баз;
- **Інтеграція з екосистемою -** фреймворк успішно інтегрується з популярними бібліотеками, такими як TypeORM для роботи з базами даних, Passport для автентифікації та Jest для тестування та Class-validator для валідації вхідних даних. Це зменшує

потребу в кастомних рішеннях і прискорює розробку завдяки використанню перевірених інструментів;

- **CLI-інструмент -** Nest.js CLI автоматизує створення модулів, контролерів, сервісів та інших компонентів. Це не лише економить час, але й забезпечує однакову структуру коду в проєкті, що важливо для великих команд.
- **Безпека** – ключовий аспект enterprise-застосунків. Nest.js пропонує вбудовані інструменти, такі як guards (охорона), interceptors (перехоплювачі) та pipes (конвеєри), для реалізації автентифікації, авторизації та валідації вхідних даних.

2.1.2 Ключові компоненти в Nest.js

Фреймворк Nest має широкий спектр вбудованих компонентів. Прописавши у консолі команду “nest” , ми побачимо підказку від nest-cli, з можливістю авто генерації окремих компонентів, файлів, або проєкту шаблону (рис. 2.1).



name	alias	description
application	application	Generate a new application workspace
class	cl	Generate a new class
configuration	config	Generate a CLI configuration file
controller	co	Generate a controller declaration
decorator	d	Generate a custom decorator
filter	f	Generate a filter declaration
gateway	ga	Generate a gateway declaration
guard	gu	Generate a guard declaration
interceptor	itc	Generate an interceptor declaration
interface	itf	Generate an interface
library	lib	Generate a new library within a monorepo
middleware	mi	Generate a middleware declaration
module	mo	Generate a module declaration
pipe	pi	Generate a pipe declaration
provider	pr	Generate a provider declaration
resolver	r	Generate a GraphQL resolver declaration
resource	res	Generate a new CRUD resource
service	s	Generate a service declaration
sub-app	app	Generate a new application within a monorepo

Рисунок 2.1 - повний список ключових компонентів та команд

Розглянемо ключові компоненти, які будуть використані у проєкті:

- Module Модулі є основними структурними одиницями в Nest.js. Вони групують пов’язані компоненти, такі як контролери, сервіси, провайдери тощо, створюючи логічно відокремлені частини застосунку.

Модулі дозволяють організувати код за функціональністю. Наприклад, у проєкті може бути модуль для автентифікації, модуль для роботи з користувачами тощо. Це сприяє масштабованості та зрозумілості коду.

- Controller обробляють вхідні HTTP-запити від клієнтів і повертають відповіді. Вони визначають маршрути (ендпойнти) і методи (GET, POST, PUT, DELETE тощо).

Вони слугують точками входу для запитів. Зазвичай вони передають складну логіку сервісам, щоб розділити обов’язки й спростити тестування.

- Service відповідають за бізнес-логіку застосунку. Вони займаються обробкою даних, взаємодією з базами даних, зовнішніми API тощо.

Завдяки механізму Dependency Injection (DI) сервіси легко інжектуються в контролери чи інші сервіси. Це робить код модульним і зручним для тестування.

- Guard контролюють доступ до маршрутів, вирішуючи, чи може запит дістатися до контролера, залежно від умов (наприклад, автентифікація чи роль користувача). Їх застосовують для захисту ендпойнтів на рівні контролера або конкретного методу. Наприклад, Guard може перевіряти токен авторизації перед обробкою запиту.

- Інтерфейси в TypeScript задають структуру даних, визначаючи контракти для об’єктів або класів.

Вони забезпечують типізацію та консистентність даних у проєкті. Наприклад, інтерфейс для об’єкта "користувач" може визначити поля id, name, email, які використовуватимуться в сервісах і контролерах.

- Interceptors дозволяють перехоплювати запити й відповіді для виконання додаткових дій, таких як логування, трансформація даних чи кешування.

Їх можна застосовувати глобально або локально (на контролер чи

метод). Наприклад, Interceptor може додавати метадані до відповіді або вимірювати час виконання запиту. Gateway призначені для роботи з WebSockets, дозволяючи реалізувати функціональність реального часу (наприклад, чати чи сповіщення). Вони обробляють події WebSocket, такі як підключення чи отримання повідомлень, і можуть інтегруватися з сервісами для виконання логіки. Filters відповідають за обробку помилок і винятків, дозволяючи повертати клієнту кастомізовані відповіді. Їх можна налаштувати глобально або для конкретних контролерів/методів. Наприклад, Filter може перехопити помилку 404 і повернути уніфіковане повідомлення. Pipes відповідають за парсинг та валідацію вхідних даних, наприклад через контроллер. Це може бути як число, строчка або файл. При порушені певних умов, наприклад вхідний файл занадто великий, видасть помилку з кодом 400 (error bad request).

Data transfer object (DTO) - сутність клас, що представляє деякий об’єкт, що потребує відділення від бізнес логіки, а також окремої валідації завдяки декораторам та та бібліотеці class-validator. При правильному налаштуванні nest автоматично десиріалізує вхідні запити, валідує їх та при невідповідності схеми обробляє помилку з кодом 400.

Паттерн DTO дозволяє обмінюватися даними з різними слоями застосунку, поміщати логіку створення об’єкту даних в конструкторі , валідувати сам об’єкт, видаляти певні поля при відправленні к клієнту, наприклад password або іншу чутливу інформацію.

2.1.3 Алгоритм обробки http запросів в Nest.js (Data flow)

У класичному Nest.js-застосунку, коли клієнт відправляє запит на сервер, фреймворк обробляє його через чітко визначений ланцюжок викликів компонентів. Цей процес є структурованим завдяки модульній архітектурі та патернам Nest.js.

Middleware (Проміжне ПЗ):

- Запит спочатку потрапляє до **middleware**, якщо воно налаштоване (глобально чи для конкретного маршруту). Middleware може виконувати попередню обробку, наприклад, логування, перевірку заголовків чи парсинг тіла запиту;
- Приклад: Перевірка CORS або логування часу запиту.

Guard (аутентифікація):

- Якщо для маршруту налаштовано **Guard**, він перевіряє, чи має запит доступ до ресурсу. Guard повертає true або false, дозволяючи або блокуючи подальшу обробку;
- Приклад: Перевірка токена автентифікації (наприклад, JWT).

Interceptor (Перехоплювач, до виконання)**:**

- **Interceptor** може перехопити запит перед його передачею до контролера. На цьому етапі можна модифікувати запит, додати метадані або виконати логування;
- Приклад: Додавання таймера для вимірювання часу виконання запиту.

Pipe (Конвеєр)**:**

- **Pipe** обробляє та валідує вхідні дані запиту (наприклад, параметри, тіло запиту). Він може трансформувати дані або кидати виняток, якщо дані некоректні;
- Приклад: Перевірка, чи відповідає тіло запиту DTO (Data Transfer

Object).

Controller (Контролер)**:**

- Запит потрапляє до **контролера**, який визначає маршрут і викликає відповідний метод для обробки. Контролер зазвичай делегує бізнес-логіку сервісу;
- Приклад: Метод getUser у контролері викликає сервіс для отримання даних користувача.

Service (Сервіс):

- **Сервіс** виконує бізнес-логіку, наприклад, взаємодію з базою даних, зовнішніми API чи обробку даних. Контролер викликає методи сервісу через Dependency Injection;
- Приклад: Запит до бази даних через TypeORM для отримання запису.

Interceptor (Перехоплювач, після виконання):

- Після виконання логіки в сервісі та повернення результату контролером, **Interceptor** може обробити відповідь. На цьому етапі можна модифікувати відповідь або виконати додаткові дії (наприклад, логування результату);
- Приклад: Форматування відповіді у певний JSON-формат.

Filter (оброблювач помилок):

- Якщо під час обробки запиту виникає виняток, **Filter** перехоплює його та формує відповідь для клієнта. Фільтри дозволяють централізовано обробляти помилки;
- Приклад: Повернення статусу 400 з повідомленням про некоректні дані.

Після проходження всіх етапів сервер повертає відповідь клієнту, зазвичай у форматі JSON. Якщо використовуються WebSockets (через

Gateway), відповідь може бути подією реального часу.

Не всі компоненти обов’язково використовуються в кожному запиті. Наприклад, Guard або Interceptor можуть бути відсутні, якщо не налаштовані для конкретного маршруту.

Порядок викликів є фіксованим. Наприклад клієнт надсилає GET-запит /users/1 рис. 2.2 :

1. Middleware перевіряє заголовки (наприклад, CORS).
2. Guard перевіряє, чи автентифікований користувач.
3. Interceptor логирує час початку запиту.
4. Pipe валідує параметр id (переконується, що це число).
5. Controller викликає метод findUserById у сервісі.
6. Service влючає в собі бізнес логіку застосунку та запитує викликає метод findUserById репозиторію.
7. Repository відправляє автоматично сформований sql запит до бази даних.
8. Interceptor форматує відповідь, додаючи метадані.
9. Якщо сталася помилка (наприклад, користувача не знайдено), Filter повертає статус 404.
10. Клієнт отримує відповідь із даними або повідомленням про помилку.

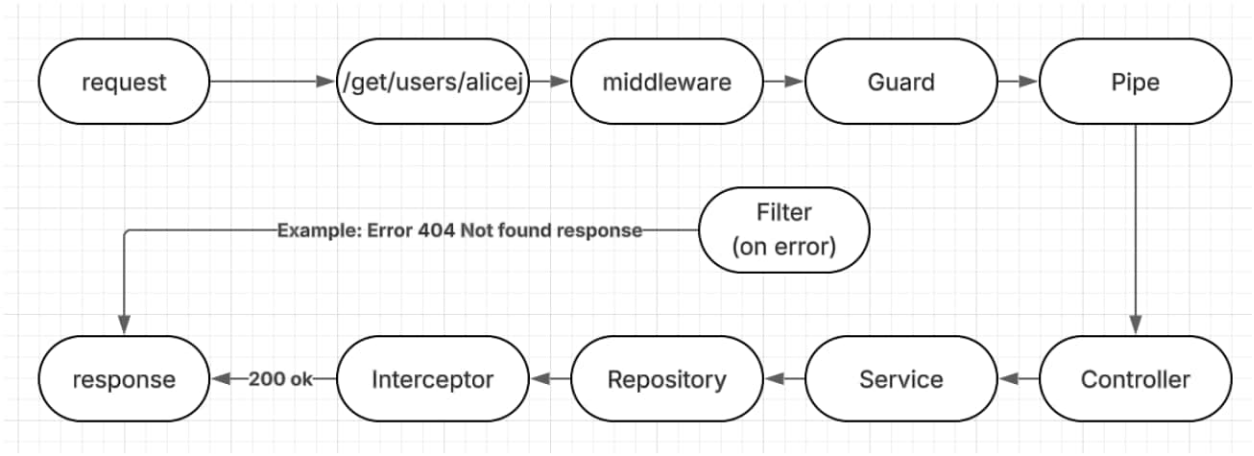


Рисунок 2.2 - Діаграма порядку виконання http запиту

2.2 Архітектура бази даних Postgresql

Для мінімальної життєздатності веб-чату необхідно зберігати в базі даних таблиці з користувачами, інформацію про чати, користувачів що знаходяться в чаті, та самі повідомлення рис 2.3 .

- Таблиця *users* містить в собі дані з користувачами застосунку. Ім’я, нікнейм, електрону пошту та пароль у (захешованому вигляді), id. Не має посилання на інші таблиці. Містить статус “online” (у мережі) та останню дату останньої ативності.
- Таблиця *chats* містить містить інформацію про чати, їхню назву, id, також може містити ключ шифрування (у майбутньому iv) . Не має посилань на інші таблиці. Також містить каунтер користувачів та повідомлень в чаті, реалізований через (Triggers в Postgresql). Цей каунтер необхідний для пагінації кількості користувачів в чаті та повідомлень.
- Таблиця *users_chats* містить посилання на id користувачів та чатів як

ключ, тим самим створюючи зв'язок користувачів до чатів. Один користувач може одночасно знаходитися в багатьох чатах. Postgresql не дозволяє мати дублікати, коли в одному чаті двох однакових користувачів, бо по перше, він створює індекс, що прискорює пошук даних, але обов'зково мати унікальні id chats та users.

- Таблиця *messages* зберігає в собі дані про повідомлення, мета-дані, а також посилання на id користувача відправника та id чату як первинні ключі. Тим самим можна легко знайти по повідомленю відправника та сам чат.

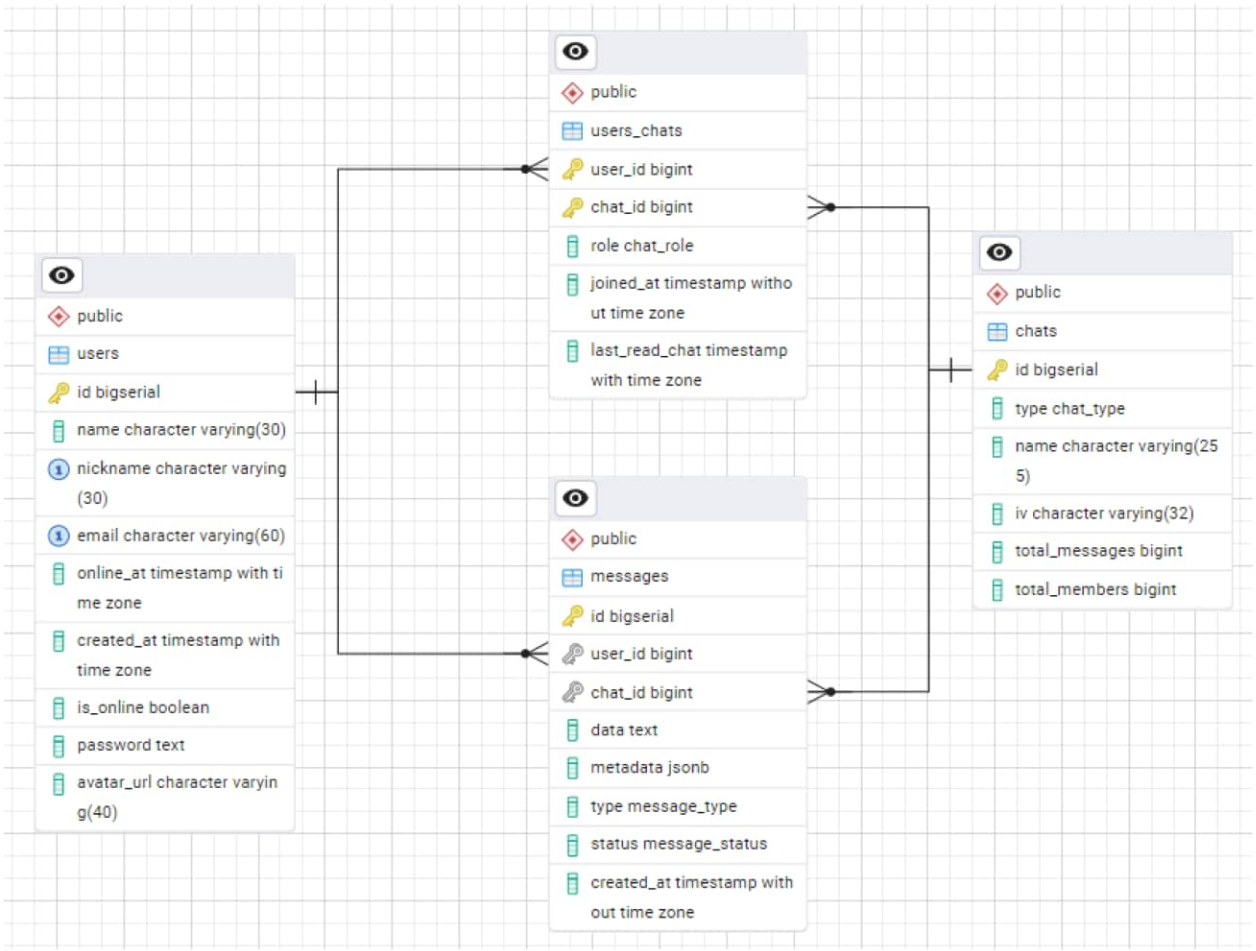


Рисунок 2.3 - ERD діаграма бази даних

Завдяки подвійному ключу в таблицях *messages* та *users_chats,* можемо ефективно управляти багатьма користувачами в одному чаті, та багатьма повідомленнями в одному чаті. Таблиці не дублюють дані друг друга, дотримуючись принципу “єдиного джерела правди”. Це дає нам змогу легко налаштовувати ролі користувачів, їхні останні дії, їхні повідомлення, та CRUD операції над ними.

Також поле *last_read_chat* було додано як часова мітка, коли користувач прочитав усі повідомлення до певного моменту. Такий підхід не потребує зв’язку з таблицею *messages* , а значить при видаленні повідомлень не прийдеться робити додаткових алгоритмів, бо усі повідомлення до часової мітки вважаються прочитаними, а усі після мітки не прочитаними. Тим самим спрощує алгоритм по підрахунку нових або прочитаних повідомлень без явних зв’язків.

2.3 Архітектура компонентів серверної частини

Модульна архітектура - це декомпозиція системи на окремі модулі, які повинні виконувати окремий функціонал, та мати лише одну зону відповідальності. Таким чином можливо розбити великий проєкт на багато окремих частин, які в ідеалі не пов’язані один з одним. Але в реальності допускається імпортування одного модуля в інший, тим самим перевикоритовуючи його функціонал, однак через це можуть виникати складні зв’язки між модулями, та певні проблеми у майбутньому.

Модуль - це набір компонентів, наприклад, Controller, Service,

Repository і тд. , що поєднані в одному каталозі, та виконують спільну задачу, але мають свої зони відповідальності.

Маємо 7 ключових модулів в системі рис. 2.4 (стрілки між модулями вказують на залежність):

- App - єдина точка запуску програми. Містить конфігурацію застосунка, імпорт залежностей, імпорт .env файлів. Не містить ніякої бізнес логіки чи складних операцій;
- Users - відповідає за взаємодію з таблицею users, та CRUD операції відносно їх. Вона містить в собі як Controller, для Rest API, так і UsersService де міститься основна логіка до роботи з користувачами;
- Auth - відповідає за авторизацію та автинфікацію користувачів та управління JWT токенами. Також працює з ключовим компонентом Guard та кастомним декоратором “isPublicstr. При успішній авторизації користувача підписує спеціальний JWT токен, що зберігається у cookies, тим самим автоматично додається в усіх http запитах. Якщо в запиті до сервера клієнт не має JWT токена , або він спробує підмінити токен , то система вияве таку поведінку та викене помилку з кодом 400. При авторизації паролі не порівнюються на пряму, а попередньо вони захешовані в базі даних, тому не можливо дізнатися через базу даних справжній пароль користувача. Тільки якщо захешований пароль та відправлений пароль збігаються в спеціальній функції де-хешування , користувач вважається авторизованим;
- Chats - відповідає за CRUD операції над самими чатами через Controller та відповідний ChatsService;
- Messages - за CRUD операціями над повідомленнями. Також має

залежність від модуля Chats ;

- Media - відповідає за підключення до Google drive сховища, та відповідно за завантаження, пошуку та виделенню медіа файлів, в заданому проєкті саме за фотографії користувачів;
- Gateway - це окремий модуль, що відповідає виключно за підключення до Web-Socket серверу, керуванням підключенням, та відправкою повідомленнями в реальному часі через NotificationService. Дуже важливим фактом є те, що GateWay компонент глобаний, та працює за паттерном singleton. Тобто маємо єдиний екземпляр, який зберігає в оперативній пам'яті підключення до серверу через Web-Socket. Цей паттерн дозволяє отримати інформацію про підключені клієнти з будь якої точки в застосунку, та відправляти повідомлення.

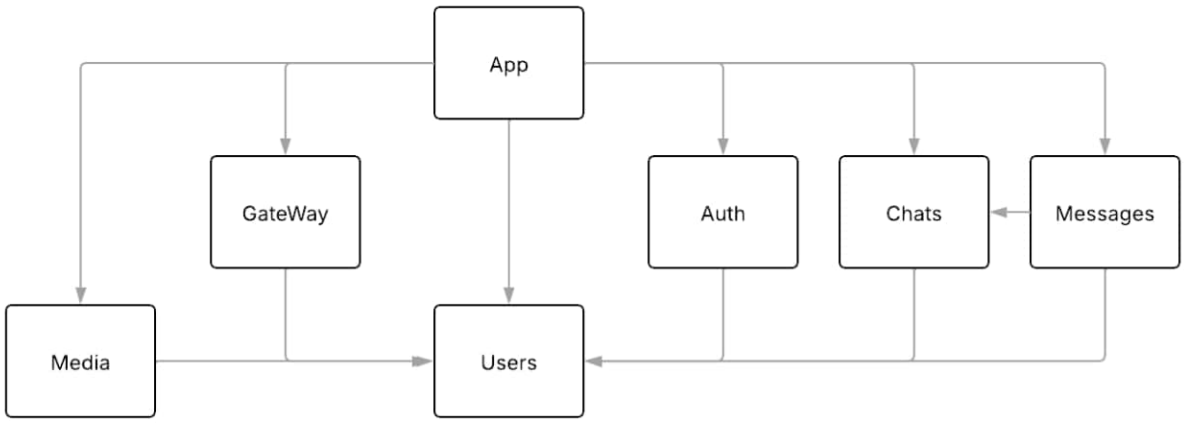


Рисунок 2.4 - Взаємозв'язок компонентів системи

2.4 Архітектура rest API

RESTful API (Representational State Transfer Application Programming

Interface) є одним із найпоширеніших архітектурних стилів для створення вебсервісів, що забезпечують взаємодію між клієнтом і сервером через мережу. Цей підхід базується на принципах REST, уперше описаних Роєм Філдінгом у 2000 році. RESTful API використовує стандартні методи HTTP для операцій над ресурсами, що робить його простим, масштабованим і незалежним від платформи.

2.4.1 Основні принципи rest API

RESTful API базується на кількох ключових принципах:

1. Клієнт і сервер є окремими компонентами, що взаємодіють через мережу, дозволяючи розвивати їх незалежно;
2. **Stateless -** кожен запит від клієнта до сервера містить усю необхідну інформацію. Сервер не зберігає стан клієнта між запитами;
3. **Кешування -** відповіді сервера можуть кешуватися для зменшення навантаження та прискорення обробки повторних запитів;
4. **Уніфікований інтерфейс -** використовуються стандартні методи HTTP (GET, POST, PUT, DELETE) для операцій над ресурсами;
5. **Ресурсно-орієнтована модель -** дані представлені як ресурси з унікальними ідентифікаторами (URI);
6. **Шарова система -** система може складатися з кількох шарів для розподілу навантаження та підвищення безпеки.

2.4.2 Найкращі практики проектування RESTful API

Для створення якісного RESTful API необхідно дотримуватися таких практик:

Використання правильних HTTP-методів:

- **GET**: отримання ресурсу або колекції;
- **POST**: створення нового ресурсу;
- **PUT**: оновлення існуючого ресурсу;
- **DELETE**: видалення ресурсу;
- **PATCH**: часткове оновлення ресурсу.

Статусні коди HTTP:

- **200 OK**: успішна операція;
- **201 Created**: ресурс створено;
- **400 Bad Request**: некоректний запит;
- **404 Not Found**: ресурс не знайдено;
- **500 Internal Server Error**: помилка сервера.

API має повертати інформативні повідомлення, наприклад:

```
{
 "error": {
  "code": 400,
  "message": "Invalid input data",
  "details": [{"field": "email", "issue": "Email is required"}]
 }
```

2.4.3 Дизайн rest API в застосунку

Було вирішено, що застосунок буде підтримувати класичний rest API та web-socket API для обміну повідумленнями в реальному часі. Розглянемо

головні роути, авторизації, роботи з користувачами, обмін повідомленнями, операції над користувачами чатів (таблиця 2.1 та таблиця 2.2).

Таблиця 2.1 - Дизайн rest API роуту /auth

Url	request	опис	response
/auth/login [POST]	{ "password": str, "nickname": str }	Авторизація в застосунку.	Повертає дані користувача User {...}
/auth/register [POST]	{ "name": str, "nickname": str, "email": str, "password": str }	Реєстрація в застосунку.	Повертає дані користувача User {...}
/auth/test [GET]	-	Перевірка JWT токену	Повертає дані користувача User {...}
/auth/password [PATCH]	{ "id: number":, "oldPassword": str, "newPassword": str	Заміна пароля	код 200 або код 400+ при помилці

	}		

Таблиця 2.2 - Дизайн rest API роуту /users

Url	request body	опис	response body
users?take=20 &where=alice &page=1 [GET]	-	Отримати список користувачів	{data:[дані користувачів], meta: {...} } (рис 2.5)
/users/:id [GET]	-	отримати користувача по id	Повертає дані користувача User {...}
/users [PATCH]	{опціональні поля користувача}	оновлює дані свого профілю	Повертає дані користувача User {...}
/users/:id [DELETE]	-	видалити користувача	Повертає дані користувача User

Маємо наступну структуру для запиту пагінації рис 2.5:

- take - кількість повернених елементів. Максимальна кількість 50 штук
- page - яка сторінка буде відображатися, залежить від кількості що потрібно повернути “take” та загальної кількості в таблиці базі даних

- order - поряд сортування або “DEST” або “ASK”.
- include[chat] - додати до відповіді інформацію про чат (де це можливо)
- include[messages] - додати інформацію про повідомлення (де це можливо)
- include[members] - додати інформацію про користувачів (де це можливо)


```
{
    "data": [
        {
            "id": 1,
            "name": "hello",
            "nickname": "alicej",
            "email": "alice@example.com",
            "is_online": false,
            "online_at": "2025-06-12T18:44:05.630Z",
            "avatar_url": "1J5GI_dvduUfl7QS_aEpRfW3F8mjetK67"
        }
    ],
    "meta": {
        "page": 1,
        "take": 20,
        "count": 1,
        "pageCount": 1,
        "hasPreviousPage": false,
        "hasNextPage": false
```


Рисунок 2.5 - Список користувачів (User), де ім’я alicej та зразок метаданих для пагінації

Для роботи з чатами використовуємо роут /chats. Він дозволяє виконувати CRUD операції над чатами. Отримувати чати з пагінацією, отримувати окремий чат по id, створювати новий чат, в який автоматично додається користувач та видалити окремий чат по id. Для будь-яких операцій над чатами клієнт має знаходитися в цьому чаті , інакше отримає помилку 403 (forbidden).

Розглянемо дизайн rest API роутів у таблицях 2.3, 2.4 для управління

чатами та учасників чатів.

Таблиця 2.3 - Дизайн rest API роуту /chats

Url	request body	опис	response body
/chats?take=10&include[members]=true&include[messages]=true [GET]	-	Отримати список з чатами. Підтримує опціональні query messages та members	Об’єкт data та meta (рис. 2.5)
/chats/40?include[members]=true&include[messages]=true&page=1 [GET]	-	отримати чат по id. Підтримує опціональні query messages та members	{chat: {id, data, status, chat_id, user_id},}
/chats [POST]	{type:"private"}	створення нового чату (без користувачів), з можливістю подальшого додавання користувачів	{ "chat": { "type": "private", "name": null, "id": 42 } }
/chats/:id [DELETE]	-	видалити чат	Повертає дані користувача User , що видалили

Таблиця 2.4 -Дизайн rest API роуту /members

Url	request body	опис	response body
/members [POST]	{ "members": [{ "id": 130 }], "chat_id": 262 }	Додати користувача до чата. Відправник повинен бути сам в цьому чаті. Повертає повну інформацію про чат, користувачів та повідомлення.	{chat: messages:[...], members: [...], }
/members [delete]	-	видалити учасника чата. Відправник видалення повинен бути в цьому чаті.	код 200 або код помилки 404
/members/:id [PATCH]	{ "last_read": date}	Оновлення даних про користувача.Наприклад , коли в останній раз заходив в чат.	повертає тіло запиту

Роут /messages (рис. 2.5) відповідно дозволяє відправляти, змінювати та видаляти повідомлення. Отримати повідомлення можливо тільки в роуті /chats з параметром запиту *?include[messages]=true* .

Таблиця 2.5 - Дизайн rest API роуту /messages

Url	request body	опис	response body

/messages
[POST]
	{
	data: str,
	chat_id: num
	}
		Відправити повідомлення до чату. Передається id чату та само повідомлення в “data” відповідно.	{
			message: {
			user_id: num,
			chat_id: num,
			data: str,
			type: “text”,
			id: num,
			status: sent,
			created_at: timestamp
			}
			}

/messages [PATCH]	{data: str}	Оновлює зміст повідомлення	повертає зміст повідомлення
/messages/:id [DELETE]	-	Видаляє повідомлення по id. Необхідно знаходитися в чаті, де видаляється повідомлення. Можливо видаляти тільки свої повідомлення	повертає зміст повідомлення

2.5 Дизайн web-socket повідомлень

Як було зазначено раніне, для застосунків типу “чат” критично важливим є миттєвий та ефективний обмін повідомленнями в реальному часі, саме тому було обрано web-socket як альтернативний спосіб комунікації між клієнтом та сервером.

Завдяки декомпозиції системи на Controller, Service, Repository та ін. (пункт 2.1) можливо легко замінити Controller(класичні http запити) на Gateway (web-socket комунікація в реальному часі), зберігаючи усі необхідні вхідні та вихідні типи даних (пункт 2.4).

Для роботи web-socket попередньо необхідно авторизуватися по роуту /auth. Після цього підключитися по спеціальному роуту */gateway* - використовуючи спеціальну бібліотеку socket.io або спеціальну програму для тестування API postman.

Замість класичних роутів в web-socket існують events (події) (рис. 2.6 та рис. 2.7) , замість класичного http запиту /chats з методом GET необхідно вказувати у події конкретну дію, що не є доброю практикою в порівнянні з класичним rest API.

EVENTS +	LISTEN	DESCRIPTION
ping		check server ping
online		send message that current user is online
chats:new		create new chat. the same as /chats[POST]
chats:delete		delete existing chat. the same as /chats[DELETE]
offline		send message that current user is offline

Рисунок 2.6 - таблиця подій сервера у програмі postman (частина 1).

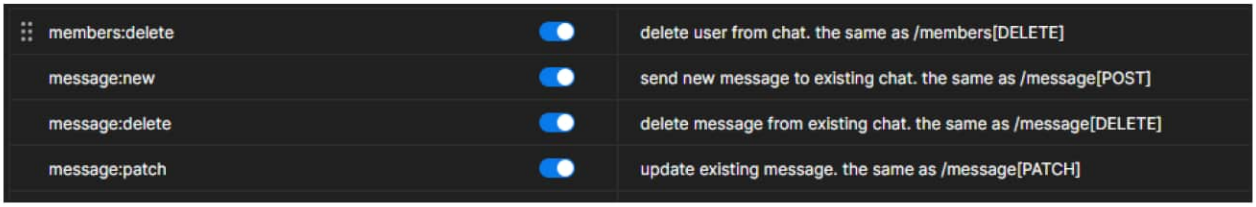


Рисунок 2.7 - таблиця подій сервера (частина 2).

2.6 Висновки до другого розділу

Другий розділ зосереджений на проектуванні архітектури та розробці технічних рішень для створення веб мессенджера, що є ключовим етапом реалізації проєкту.

Було детально описано структуру Nest.js проекту, включаючи основні компоненти, такі як контролери, сервіси та репозиторії, які забезпечують модульність і чіткий розподіл відповідальностей у системі. Особливу увагу приділено розробці бази даних у PostgreSQL, де обґрунтовано схему таблиць для ефективного управління даними користувачів, чатів і повідомлень, а також інтеграції з Google Drive для зберігання медіафайлів, що розширює функціональність додатку. Розглянуто принципи створення RESTful API та дизайн WebSocket-повідомлень, які разом забезпечують як стандартну, так і реального часу комунікацію між клієнтом і сервером, відповідаючи сучасним стандартам розробки.

Цей розділ підкреслює важливість комплексного підходу до проектування, демонструючи, як обрані рішення сприяють створенню масштабованого, безпечного та зручного у підтримці месенджера, що є основою для його практичної реалізації.

РОЗДІЛ 3 ТЕСТУВАННЯ ЗАСТОСУНКУ

Чим більше розростається проєкт, тим більш гострим стає питання написання тестів.

3.1 Тестування e2e

- Ручне тестування - вручну перевіряти сайт чи API через спеціальну програму. Має значний недолік при великому об’єму тест кейсів, чи складності проектів. Ідеально для швидкого старту;
- unit тестування - написання скрипту, що перевіряє конкретні методи чи функцію;
- E2E тестування - написання скрипту, що повністю запускає реальну програму та базу даних, та робить широкий спектр перевірок, тест кейсів на роботоспроможністьроботоспроможність усього додатку. Важкий в написанні та підтримці , повільніший (якщо не використовувати mock). Важливою перевагою є те , що цей тип тестування дає гарантію, що реальний застосунок не зламається, що дає впевненості в додаванні нового функціонала. Бажано покрити найважливіший функціонал e2e та unit-тестами;
- Інтеграційне тестування - тип тестування, де поєднуються декілька модулів програми, та перевіряється взаємодія між ними.

В даному проєкті були написані e2e тести , які повністю покривають авторизацію, автентифікацію та операції над користувачами. Попередньо

були розроблені e2e тести . Запускаємо у консолі наступну команду та отримуємо результат на рис. 3.1 та рис. 3.2:

npm run test:e2e

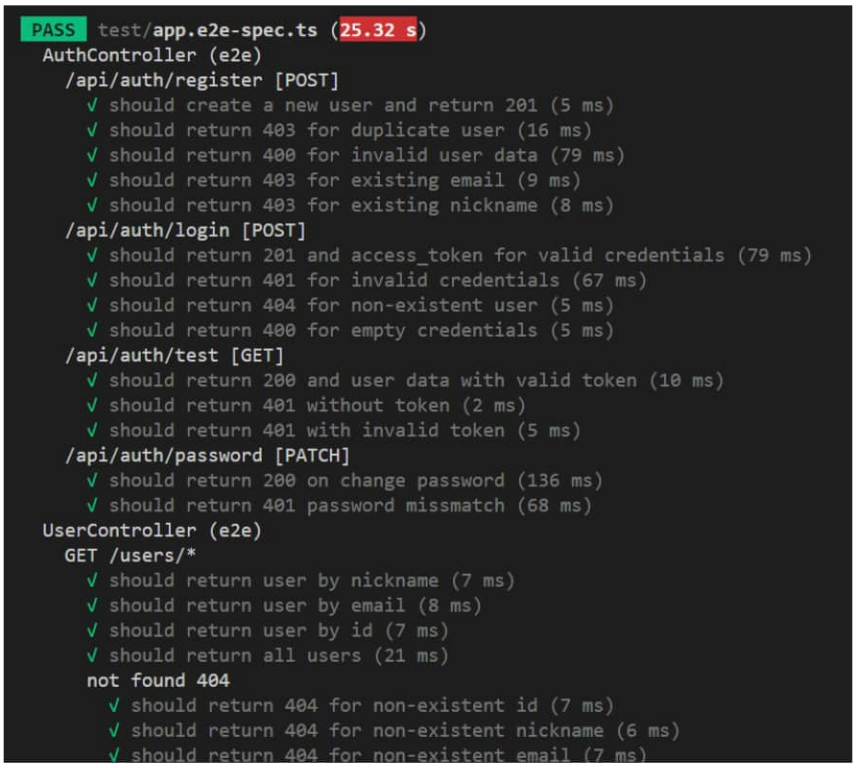


Рисунок 3.1 - вивід результатів тестування e2e тестування (частина 1).

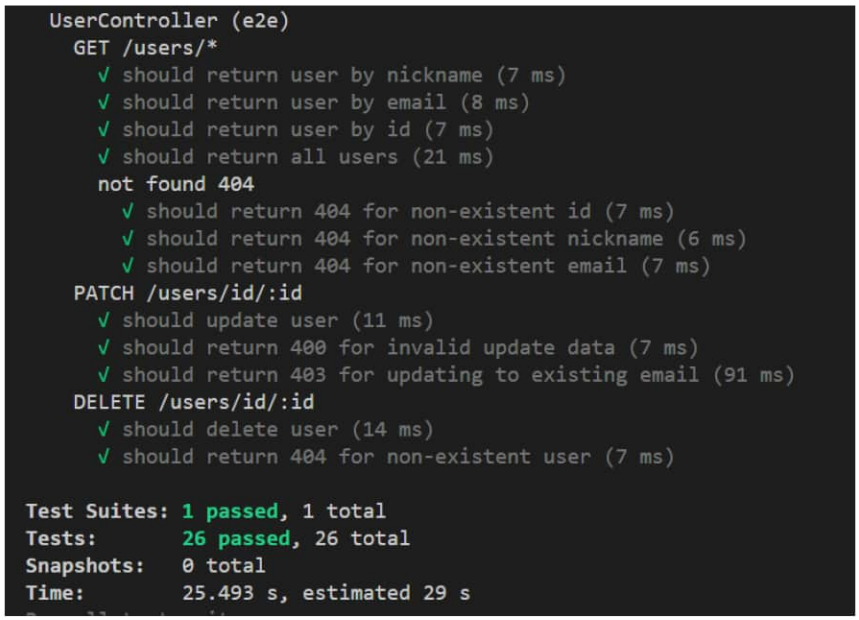


Рисунок 3.2 - вивід результатів тестування e2e (частина 2).

3.2 Тестування веб застосунку та інструкції користувача

1. Перейдіть на головний домен https://my-chat-react.vercel.app/ або на локальну версію localhost:3000 рис. 3.3;

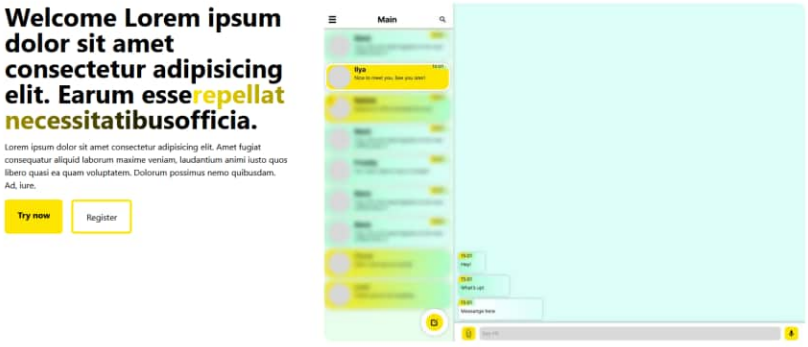


Рисунок 3.3 - Головна привітальна сторінка

2. Натисність на кнопку “Register” (/auth/login);
3. Введіть необхідні поля в формі реєстрації рис. 3.4 ;

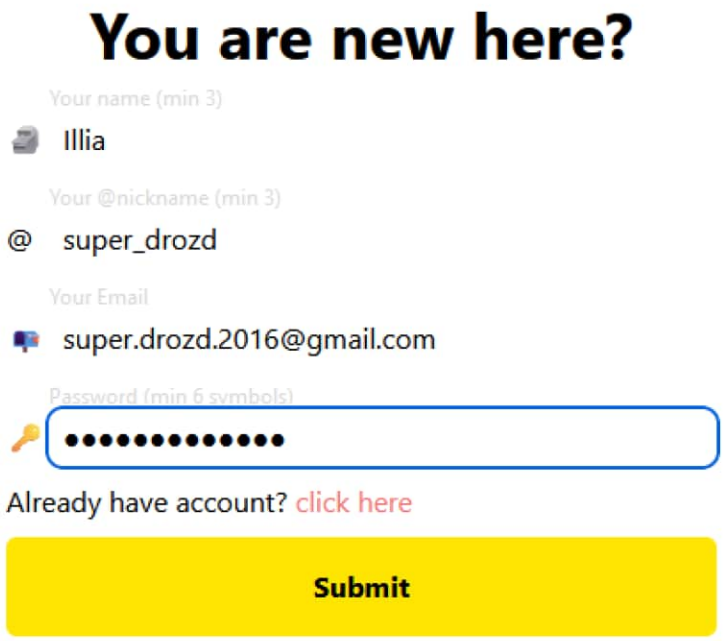


Рисунок 3.4 - форма реєстрації нового користувача

4. Натисність кнопку “Submit”;
5. Вас автоматично переведе на сторінку авторизації (/auth/login);
6. Форма авторизації автоматично буде заповнена. Натисніть на кнопку “Submit”, щоб увійти у свій профіль рис 3.5;

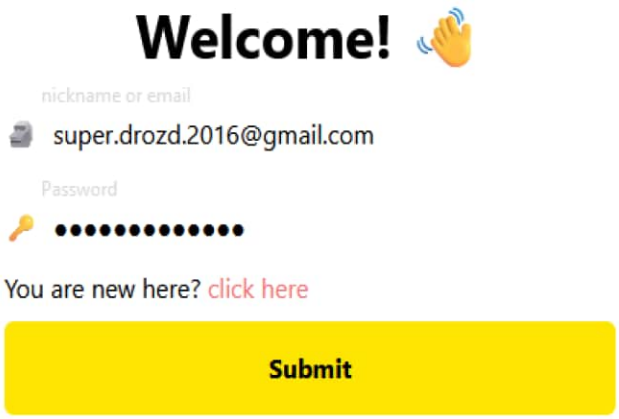


Рисунок 3.5 - сторінка авторизації

7. Після успішної авторизації вас перекине на сторінку профіля користувача /profile (рис.3.6) ;

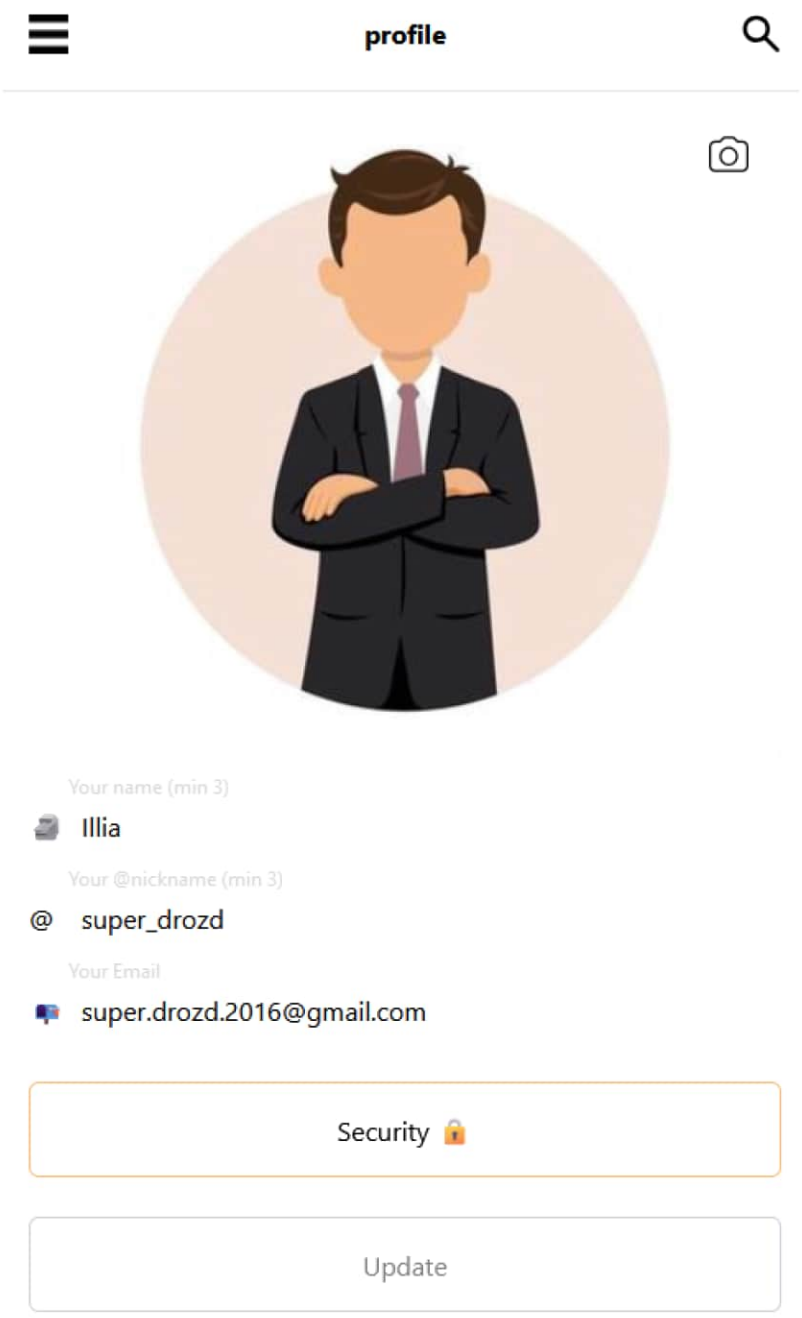


Рисунок 3.6 - успішна авторизація користувача

8. За потреби поміняйте ім’я, нікнейм, пошту, або пароль (у розділі

security). Також натиснув на фотографію оберіть фотографію для свого профіля (рис. 3.7);

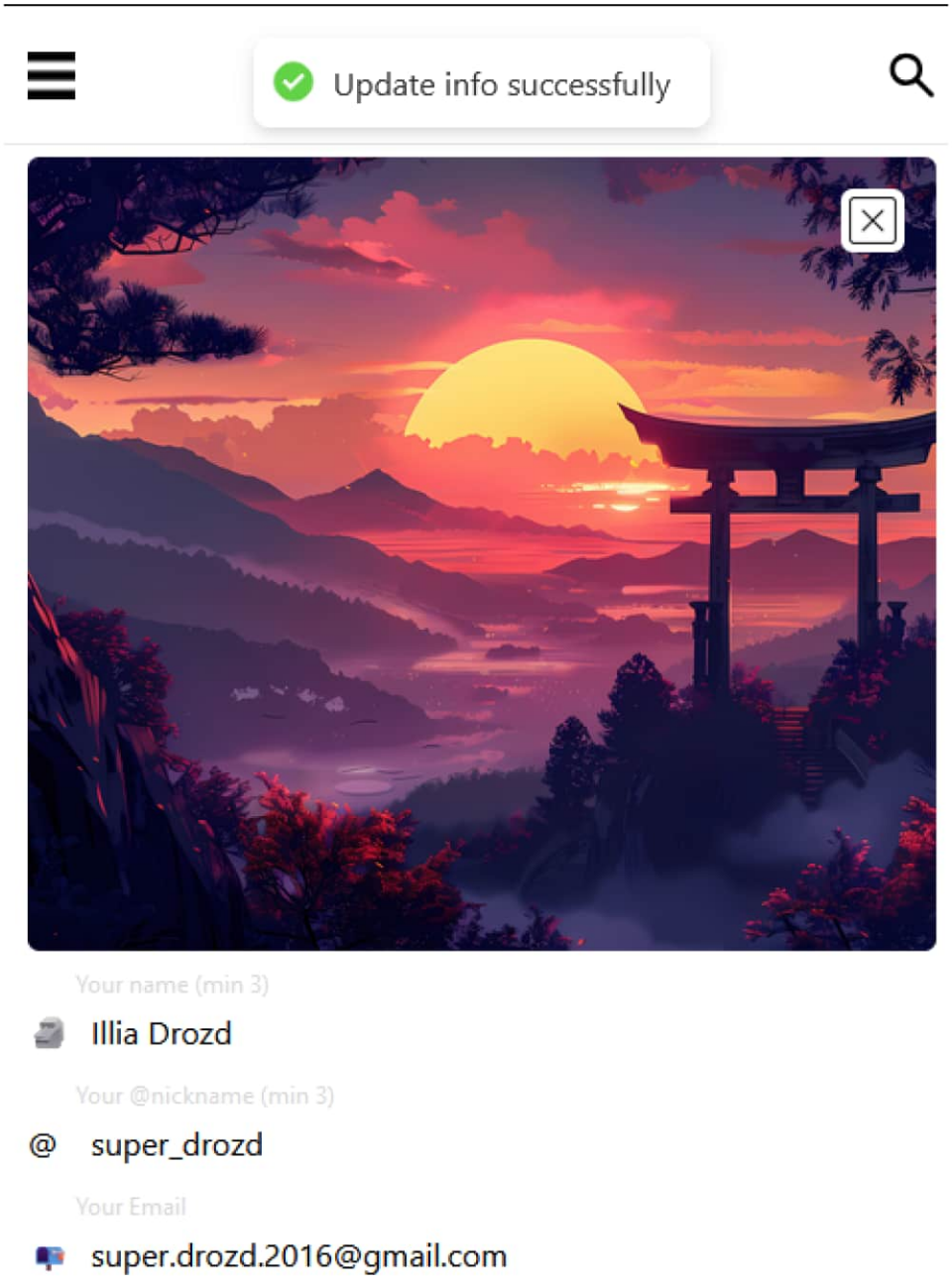


Рисунок 3.7 - успішне оновлення профілю користувача

9. Натисніть на “бургер меню” зліва в верхньому куту, щоб відкрити панель управління та натисніть на кнопку chats (рис. 3.8 та рис.3.9);

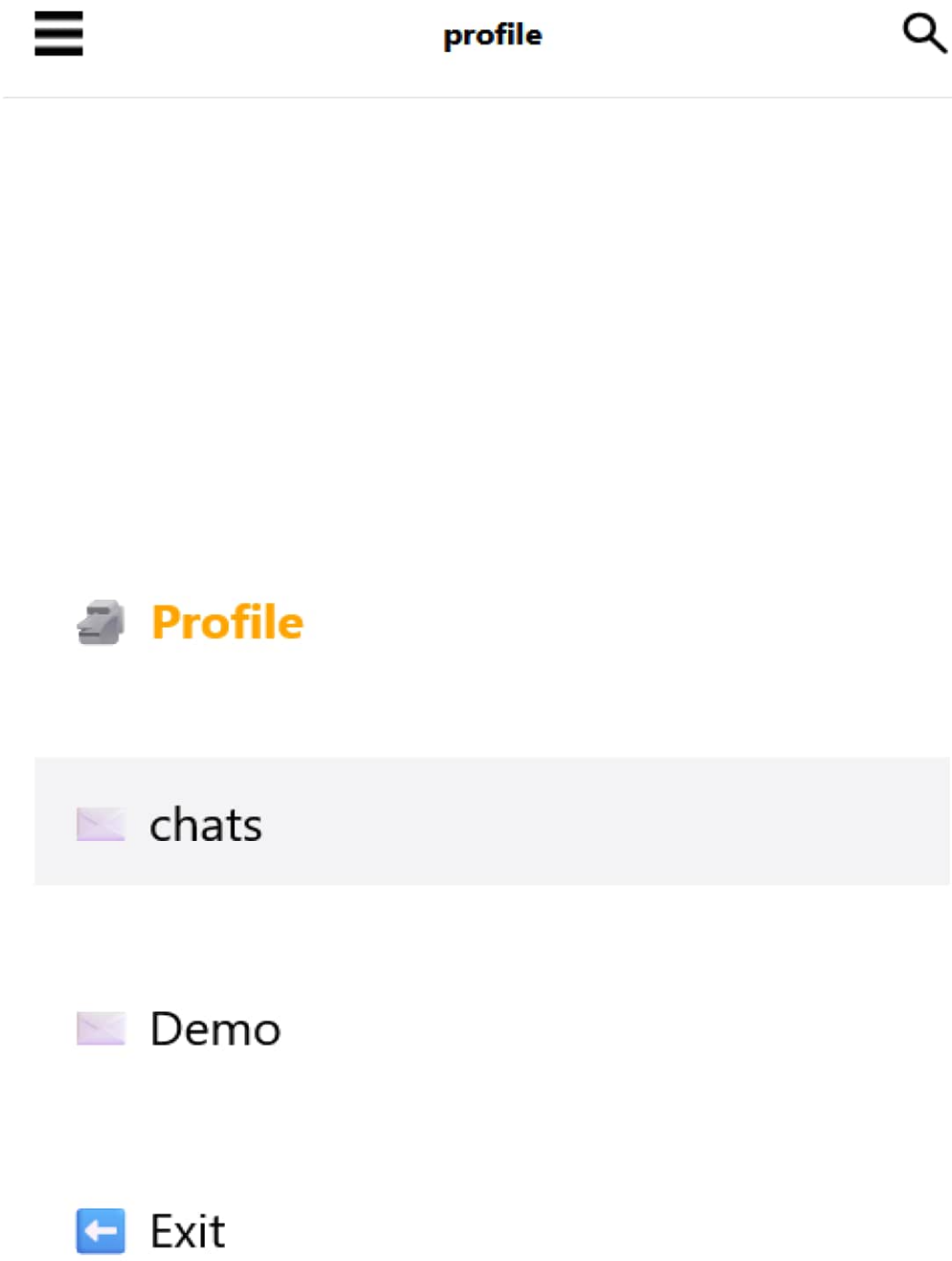
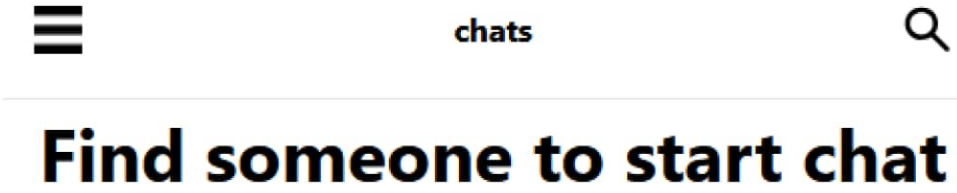


Рисунок 3.8 - відкрите меню



Find someone to start chat

Рисунок 3.9 - перехід на сторінку з пустими чатами

10. Натисніть на кнопку пошуку в правому верхньому куту, та введіть користувача з яким ви хочете почати спілкування. Користувачі з фотографіями профілю завжди будуть в горі. З права в верхньому куту написана дата, коли користувач був останній раз у мережі. Якщо користувач у мережі, то відповідно буде написано зеленим кольором текст “ONLINE” (рис. 3.10);

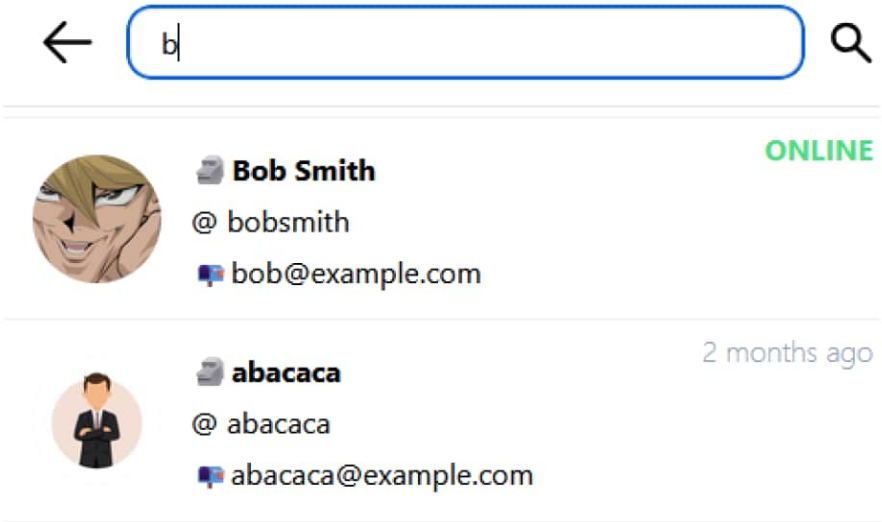


Рисунок 3.10 - список користувачів

11. Натисніть на ячейку з користувачем та вас перекине на сторінку профілю з користувачем (/profile/bobsmith) (рис.3.11);

Рисунок 3.11 - огляд профіля користувача

12. Натисніть на кнопку “chat”, щоб почати спілкування (рис. 3.12);

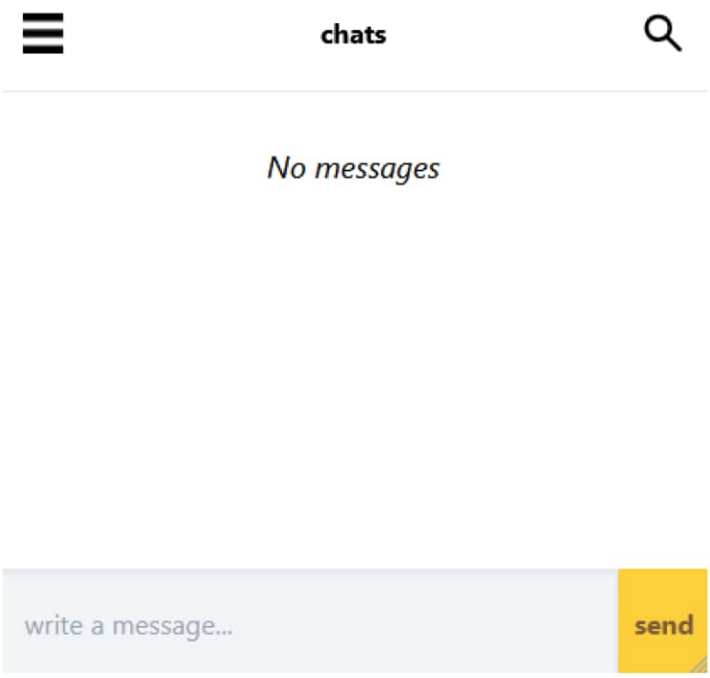


Рисунок 3.12 - новий чат з користувачем

13. Напишіть першим повідомлення та натисніть на кнопку “send” (рис. 13) ;

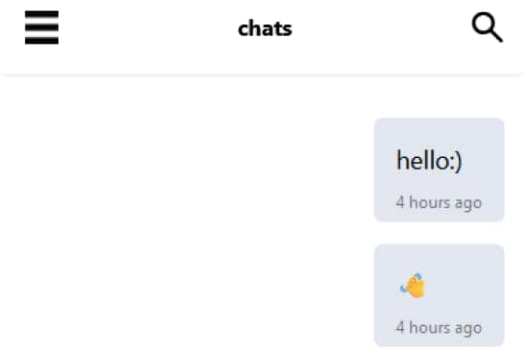


Рисунок 3.13 - відправлені привітальні повідомлення

14. Якщо ваш співрозмовник відповість, то ви отримаєте повідомлення в реальному часі без перезавантаження сторінки. Побачити його ви можете на сторінці з чатами , або в самому чаті (рис. 3.14 та рис. 3.15);

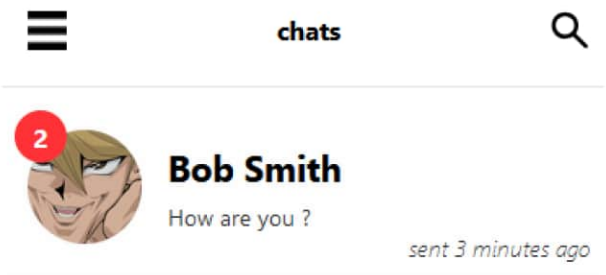


Рисунок 3.14 - відповідь користувача на сторінці з чатами

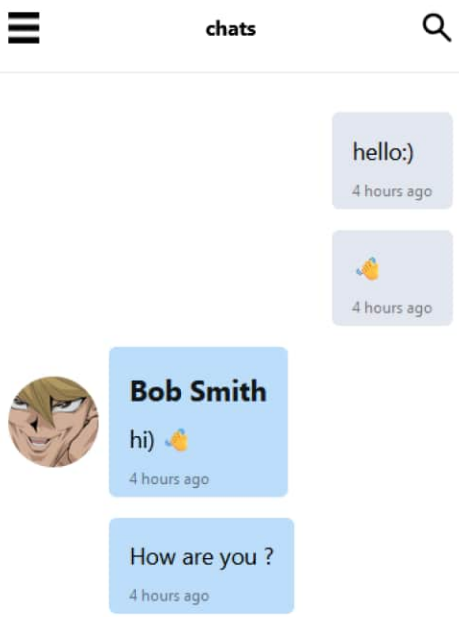


Рисунок 3.15 - повне відображення чату

15. Для видалення повідомлення натисніть правою кнопкою миші на повідомлення або натисніть пальцем (на тачскріні) (рис. 3.16 та рис.1.7);

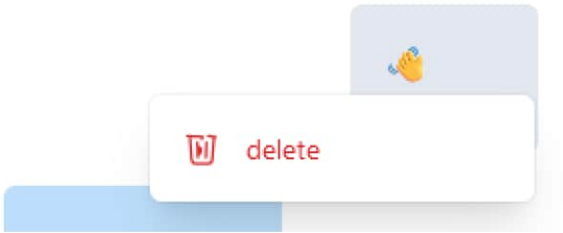


Рисунок 3.16 - меню для видалення

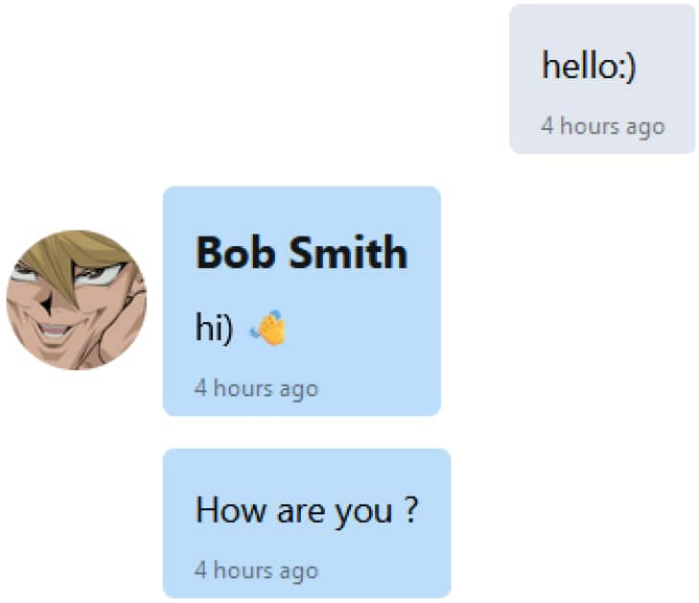


Рисунок - 3.17 - успішно вадалене повідомлення

16. Для видалення чату поверніться на сторінку /chats і так само натисніть правою кнопкою миші на чат, або затисніть користувача на тачскріні (рис. 3.18 та рис. 3.19);

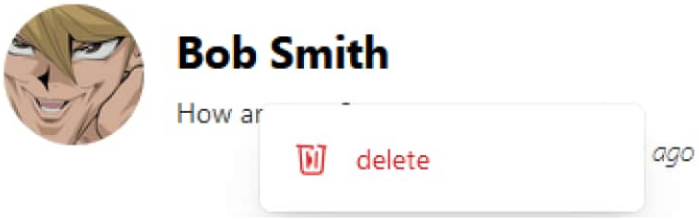


Рисунок 3.18 - меню видалення чату

Рисунок 3.19 - успішне видалення чату

3.3 Висновки до третього розділу

Третій розділ присвячений тестуванню розробленого месенджера, що є критично важливим для забезпечення його якості, стабільності та відповідності заявленим вимогам. Було розглянуто різні підходи до тестування, зокрема ручне, unit-тестування, інтеграційне та end-to-end (e2e) тестування, з акцентом на e2e тести, які охоплюють ключові функції, такі як авторизація, автентифікація та управління користувачами, підтверджуючи працездатність системи.

Наведено детальні інструкції для користувачів щодо використання веб-додатку, що ілюструють його інтуїтивність і функціональність, а також результати тестування, які свідчать про успішну реалізацію основного функціоналу. Цей етап роботи підкреслив значення тестування у процесі розробки програмного забезпечення, забезпечивши впевненість у надійності месенджера та його готовності до використання. Таким чином, можна

вважати завершеним цикл розробки, довівши, що створений прототип відповідає поставленим цілям і може слугувати основою для подальшого вдосконалення.

ВИСНОВКИ

Отже, робота була присвячена дослідженню та розробці full-stack веб-чату типу messenger, був проведений огляд існуючих рішень, а саме застосунків telegram, signal, discord, slack, та ін. Був проведений аналіз галузі та поставлене завдання на реалізацію веб-чату. Розглянуто ключові відмінності між rest API та websocket комунікацією в клієнт-серверній архітектурі.

У другому розділі було розглянуто архітектурні підходи до написання rest API, дизайн rest API та websocket роутів, детальний огляд архітектури бази даних в PostgreSQL, огляд таблиць users, users_chats, chats, messages, архітектура nest.js фреймворку та його ключових компонентів.

У третьому розділі були розглянуті основні методи тестування застосунку, а сама unit-тести, інтеграційні, e2e тести, їх ключові відмінності. В самому проекті були написані e2e тести, що покривали авторизацію, автентифікацію, управління користувачами, а також продемонстрована працездатність веб застосу ручним тестуванням.

Результатом роботи є спроектована та розроблена rest API та web-socket API застосунку, успішно інтегрована з клієнтом, написаним на React. Була спроектована та реалізована база даних на PostgreSQL. Був розроблений backend застосунок з використанням фреймворку nest.js з інтеграцією з PostgreSQL, e2e тестами та клієнтом написаним на бібліотеці React.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Instant Messaging Apps Types - PubNub [Електронний ресурс]. URL: https://www.pubnub.com/blog/instant-messaging-apps-types/ (дата звернення 11.05.2025).
2. What is Signal? 7 features that make it a go-to app for private ... [Електронний ресурс]. URL: https://www.zdnet.com/article/what-is-signal-7-features-that-make-it-a-go-to-app-for-private-secure-messaging/ (дата звернення 11.05.2025).
3. The Advantages and Disadvantages of WhatsApp | ClickUp [Електронний ресурс]. URL: https://clickup.com/blog/advantages-and-disadvantages-of-whatsapp/ (дата звернення 11.05.2025).
4. Telegram Review 2025: Pros, Cons, and Test Results - CyberInsider [Електронний ресурс]. URL: https://cyberinsider.com/secure-encrypted-messaging-apps/telegram/ (дата звернення 11.05.2025).
5. Honest Slack Review 2025: Pros, Cons, Features & Pricing [Електронний ресурс]. URL: https://connecteam.com/reviews/slack/ (дата звернення 11.05.2025).
6. Slack Architecture: The Story of AWS, Microservices & DevOps Best ... [Електронний ресурс]. URL: https://blog.pipeops.io/slack-architecture-the-story-of-aws-microservices-devops-best-practices/ (дата звернення 11.05.2025).
7. How to Build a Distributed Messaging System like Discord - AlmaBetter

[Електронний ресурс]. URL: https://www.almabetter.com/bytes/articles/build-a-distributed-messaging-system-like-discord (дата звернення 11.05.2025).

8. NestJS: A Progressive Node.js Framework - GeeksforGeeks [Електронний ресурс]. URL: https://www.geeksforgeeks.org/nestjs-a-progressive-nodejs-framework/ (дата звернення 11.05.2025).
9. PostgreSQL vs MySQL: Which Database is Right for You? - TechRepublic [Електронний ресурс]. URL: https://www.techrepublic.com/article/postgresql-vs-mysql/ (дата звернення 11.05.2025).
10. WebSockets vs. Long Polling vs. Server-Sent Events - GeeksforGeeks [Електронний ресурс]. URL: https://www.geeksforgeeks.org/websockets-vs-long-polling-vs-server-sent-events/ (дата звернення 11.05.).

ДОДАТОК А

Лістинг файлу /nest/package.json

```
{
    "name": "chat-pro",
    "version": "0.0.1",
    "description": "",
    "author": "",
    "private": true,
    "license": "UNLICENSED",
    "engines": {
        "node": "22.x"
    },
    "scripts": {
        "build": "nest build",
        "format": "prettier --write \"src/**/*.ts\" \"test/**/*.ts\"",
        "start": "cross-env NODE_ENV=production nest start",
        "start:dev": "cross-env NODE_ENV=development nest start --watch",
        "start:debug": "cross-env NODE_ENV=development nest start --debug
--watch",
        "start:prod": "cross-env NODE_ENV=production node dist/main",
        "lint": "eslint \"{src,apps,libs,test}/**/*.ts\" --fix",
        "test": "jest",
        "test:watch": "jest --watch",
        "test:cov": "jest --coverage",
        "test:debug": "node --inspect-brk -r tsconfig-paths/register -r
ts-node/register node_modules/.bin/jest --runInBand",
        "test:e2e": "cross-env NODE_ENV=development jest --config
./test/jest-e2e.json",
        "t": "npm run build && npx typeorm -d
dist/config/database.config.js",
        "migration:create": "npx typeorm migration:create
src/migrations/dev/%npm_config_name%",
        "migration:generate": "npm run t -- migration:generate
src/migrations/dev/%npm_config_name%",
        "migration:run": "npm run t -- migration:run",
        "migration:revert": "npm run t -- migration:revert",
```

```
        "t:prod": "npm run build && cross-env NODE_ENV=production npx
typeorm -d dist/config/database.config.js",
        "prod:migration:create": "npx typeorm migration:create
src/migrations/prod/%npm_config_name%",
        "prod:migration:generate": "npm run t:prod -- migration:generate
src/migrations/prod/%npm_config_name%",
        "prod:migration:run": "npm run t:prod -- migration:run",
        "prod:migration:revert": "npm run t:prod -- migration:revert"
    },
    "dependencies": {
        "@nestjs/cli": "11.0.6",
        "@nestjs/common": "^11.0.1",
        "@nestjs/config": "^4.0.1",
        "@nestjs/core": "^11.0.1",
        "@nestjs/jwt": "^11.0.0",
        "@nestjs/mapped-types": "*",
        "@nestjs/platform-express": "^11.0.1",
        "@nestjs/platform-socket.io": "^11.0.13",
        "@nestjs/typeorm": "^11.0.0",
        "@nestjs/websockets": "^11.0.13",
        "@types/multer": "^1.4.12",
        "bcrypt": "^5.1.1",
        "class-transformer": "^0.5.1",
        "class-validator": "^0.14.1",
        "cookie-parser": "^1.4.7",
        "cross-env": "^7.0.3",
        "googleapi": "^1.0.2",
        "googleapis": "^148.0.0",
        "install": "^0.13.0",
        "multer": "^1.4.5-lts.2",
        "npm": "^11.2.0",
        "pg": "^8.14.1",
        "reflect-metadata": "^0.2.2",
        "rxjs": "^7.8.1",
        "typeorm": "^0.3.21"
    },
    "devDependencies": {
```

```
        "@eslint/eslintrc": "^3.2.0",
        "@eslint/js": "^9.18.0",
        "@nestjs/schematics": "^11.0.0",
        "@nestjs/testing": "^11.0.1",
        "@swc/cli": "^0.6.0",
        "@swc/core": "^1.10.7",
        "@types/bcrypt": "^5.0.2",
        "@types/cookie-parser": "^1.4.8",
        "@types/express": "^5.0.0",
        "@types/jest": "^29.5.14",
        "@types/node": "^22.10.7",
        "@types/supertest": "^6.0.2",
        "eslint": "^9.18.0",
        "eslint-config-prettier": "^10.0.1",
        "eslint-plugin-prettier": "^5.2.2",
        "globals": "^16.0.0",
        "jest": "^29.7.0",
        "prettier": "^3.4.2",
        "source-map-support": "^0.5.21",
        "supertest": "^7.0.0",
        "ts-jest": "^29.2.5",
        "ts-loader": "^9.5.2",
        "ts-node": "^10.9.2",
        "tsconfig-paths": "^4.2.0",
        "typescript": "^5.7.3",
        "typescript-eslint": "^8.20.0"
    },
    "jest": {
        "moduleFileExtensions": [
            "js",
            "json",
            "ts"
        ],
        "rootDir": "src",
        "testRegex": ".*\\.spec\\.ts$",
        "transform": {
            "^.+\\.(t|j)s$": "ts-jest"
```

```
        },
        "collectCoverageFrom": [
            "**/*.(t|j)s"
        ],
        "coverageDirectory": "../coverage",
        "testEnvironment": "node"
    }
}
```

ДОДАТОК Б

Лістинг файлу tsconfig.json

```
{
    "compilerOptions": {
        "module": "commonjs",
        "declaration": true,
        "removeComments": true,
        "emitDecoratorMetadata": true,
        "experimentalDecorators": true,
        "allowSyntheticDefaultImports": true,
        "target": "ES2023",
        "sourceMap": true,
        "outDir": "./dist",
        "baseUrl": "./",
        "paths": {
            "@/*": ["./src/*"],
            "@package/*": ["../package/*"],
        },
        "incremental": true,
        "skipLibCheck": true,
        "strictNullChecks": true,
        "forceConsistentCasingInFileNames": true,
        "noImplicitAny": false,
        "strictBindCallApply": false,
        "noFallthroughCasesInSwitch": false
        // "esModuleInterop": true
    },
    "include": ["src/**/*", "test/app.e2e-spec.ts", "test/utils.ts"]
}
```

ДОДАТОК В

Лістинг файлу main.ts

```
import { NestFactory } from '@nestjs/core'
import { AppModule } from '@/app.module'
import { LoggingInterceptor } from
'@/shared/interceptors/logging.interceptor'
import { StrictValidationPipe } from '@/shared/pipes/validation.pipe'
import { cors } from '@/config/cors.config'
import * as cookieParser from 'cookie-parser'
import { hearBeat } from './shared/utils'
import { NestExpressApplication } from '@nestjs/platform-express'

export default async function bootstrap(port: number = 3000) {
    const appPort = process.env.PORT ?? port
    const app = await
NestFactory.create<NestExpressApplication>(AppModule)
    app.use(cookieParser())
    app.enableCors(cors)
    app.useGlobalInterceptors(new LoggingInterceptor())
    app.useGlobalPipes(StrictValidationPipe)
    app.set('query parser', 'extended')
    await app.listen(appPort).then(() => hearBeat(1000 * 60))
    console.log('App run sucessefully. \nPORT:', appPort)
    return app
}

// Start the app only if this file is run directly (not imported in tests)
if (require.main === module) {
    bootstrap()
}
```

ДОДАТОК Г

Лістинг модулів проєкту (app.module, users.module, router.module, messages.module, chats.module, auth.module)

```
import { ClassSerializerInterceptor, Module } from '@nestjs/common'
import { ConfigModule } from '@nestjs/config'
import { TypeOrmModule } from '@nestjs/typeorm'
import { UsersModule } from '@/modules/users/user.module'
import { databaseConfig, envFilePath } from '@/config/database.config'
import { AuthModule } from '@/modules/auth/auth.module'
import { JwtModule } from '@nestjs/jwt'
import { APP_GUARD, APP_INTERCEPTOR } from '@nestjs/core'
import { AuthGuard } from '@/shared/guard/auth.guard'
import { JWT_SECRET } from '@/shared/constants/jwt.secret'
import { ChatsModule } from '@/modules/chats/chats.module'
import { PingController } from '@/modules/ping.controller'
import { MediaModule } from '@/modules/media/media.module'
import { MessagesModule } from './modules/messages/messages.module'
import { RouterModule } from './modules/router/router.module'

@Module({
    imports: [
        ConfigModule.forRoot({ isGlobal: true, envFilePath }),
        TypeOrmModule.forRoot(databaseConfig()),
        JwtModule.register({
            global: true,
            secret: JWT_SECRET,
            signOptions: { expiresIn: '30d' },
        }),
        UsersModule,
        AuthModule,
        ChatsModule,
        MediaModule,
        MessagesModule,
```

```
        RouterModule,
    ],
    controllers: [PingController],
    providers: [
        { provide: APP_GUARD, useClass: AuthGuard },
          { provide: APP_INTERCEPTOR, useClass: ClassSerializerInterceptor
},
    ],
    exports: [],
})
export class AppModule {}

//users
import { TypeOrmModule } from '@nestjs/typeorm'
import { Module } from '@nestjs/common'
import { UserController } from './user.controller'
import { UserService } from './user.service'
import { UserEntity } from './user.entity'

@Module({
    imports: [TypeOrmModule.forFeature([UserEntity])],
    controllers: [UserController],
    providers: [UserService],
    exports: [UserService],
})
export class UsersModule {}
// router (gateway)
import { Global, Module } from '@nestjs/common'
import { UsersModule } from '../users/user.module'
import { ChatsModule } from '../chats/chats.module'
import { MessagesModule } from '../messages/messages.module'
import { UserService } from '../users/user.service'
import { MessageService } from '../messages/MessageService'
import { ChatsService } from '../chats/chats.service'
import { TypeOrmModule } from '@nestjs/typeorm'
import { ChatEntity } from '../chats/entities/chat.entity'
import { UserChatEntity } from '../chats/entities/user-chat.entity'
```

```
import { MessageEntity } from '../messages/message.entity'
import { UserEntity } from '../users/user.entity'
import { AppGateway } from './app.gateway'
import { UserSocketManager } from './user-socket-manager.service'
import { NotificationService } from './notification.service'

@Global()
@Module({
    imports: [
        UsersModule,
        ChatsModule,
        MessagesModule,
                    TypeOrmModule.forFeature([ChatEntity,   UserChatEntity,
MessageEntity, UserEntity]),
    ],
    controllers: [],
    providers: [
        AppGateway,
        UserSocketManager,
        NotificationService,
        MessageService,
        UserService,
        ChatsService,
    ],
    exports: [AppGateway, NotificationService, UserSocketManager],
})
export class RouterModule {}
// messages
import { Module, forwardRef } from '@nestjs/common'
import { TypeOrmModule } from '@nestjs/typeorm'
import { MessageEntity } from './message.entity'
import { MessagesController } from './messages.controller'
import { MessagesRepository } from './messages.repository'
import { MessageService } from './MessageService'
import { ChatsModule } from '../chats/chats.module'
import { ChatEntity } from '../chats/entities/chat.entity'
```

```
@Module({
    imports: [
        forwardRef(() => ChatsModule),
        TypeOrmModule.forFeature([MessageEntity, ChatEntity]),
    ],
    controllers: [MessagesController],
    providers: [MessagesRepository, MessageService, ],
    exports: [MessagesRepository],
})
export class MessagesModule {}
// chats
import { Module } from '@nestjs/common'
import { TypeOrmModule } from '@nestjs/typeorm'
import { ChatsService } from './chats.service'
import { ChatEntity } from './entities/chat.entity'
import { UserChatEntity } from './entities/user-chat.entity'
import { ChatsController, MembersController } from './chats.controller'
import { ChatsRepository } from './chats.repository'
import { MessagesModule } from '../messages/messages.module'
import { UserService } from '../users/user.service'
import { UserEntity } from '../users/user.entity'


@Module({
    imports: [
        MessagesModule,
                    TypeOrmModule.forFeature([ChatEntity,   UserChatEntity,
UserEntity]),
    ],
    controllers: [ChatsController, MembersController],
    providers: [
        ChatsService,
        ChatsRepository,
        UserService,
    ],
    exports: [ChatsService, ChatsRepository],
})
```

```
export class ChatsModule {}
//auth
import { Module } from '@nestjs/common'
import { AuthService } from './auth.service'
import { AuthController } from './auth.controller'
import { UsersModule } from '@/modules/users/user.module'

@Module({
    imports: [UsersModule],
    controllers: [AuthController],
    providers: [AuthService],
    exports: [AuthService],
})
export class AuthModule {}
```

ДОДАТОК Д

Лістинг підключених файлів до модуля auth (auth.controller, auth.dto, auth.service)

```
import {
    BadRequestException,
    Body,
    Controller,
    Get,
    Post,
    UseGuards,
    Request as Req,
    Patch,
    Res,
} from '@nestjs/common'
import { AuthService } from './auth.service'
import { ChangePasswordDTO, SignInDTO } from './auth.dto'
import { AuthGuard } from '../../shared/guard/auth.guard'
import { Request, Response } from 'express'
import { Public } from '@/shared/decorators'
import { CreateUserDto } from './auth.dto'
import { ROUTES_AUTH } from '@/shared/constants/routes'
import { UserEntity } from '../users/user.entity'

@Controller(ROUTES_AUTH.PREFIX)
export class AuthController {
    constructor(private authService: AuthService) {}

    @Public()
    @Post(ROUTES_AUTH.POST_LOGIN)
    async signIn(
```

```
        @Body() signInDTO: SignInDTO,
        @Res({ passthrough: true }) response: Response,
    ): Promise<{ access_token?: string; user: UserEntity }> {
        const { password, email, nickname } = signInDTO

          if (email) return this.authService.signIn({ email }, password,
response)
            if (nickname) return this.authService.signIn({ nickname },
password, response)

             throw new BadRequestException('email or nickname must be
provided')
    }

    @Public()
    @Post(ROUTES_AUTH.POST_REGISTER)
            async register(@Body() registrationDTO: CreateUserDto):
Promise<UserEntity> {
        return this.authService.register(registrationDTO)
    }

    @UseGuards(AuthGuard)
    @Patch(ROUTES_AUTH.PATCH_PASSWORD)
    async changePassword(
        @Body() changePasswordDTO: ChangePasswordDTO,
        @Req() request: Request,
    ): Promise<undefined> {
                return this.authService.changePassword(request.jwt.sub,
changePasswordDTO)
    }

    @UseGuards(AuthGuard)
    @Get(ROUTES_AUTH.TEST)
    async test(@Req() request: Request): Promise<UserEntity> {
                return this.authService.getProfile(request) //.cookies
//request.cookies?.access_token
    }
```

```
    @UseGuards(AuthGuard)
    @Post('logout')
      async logOut(@Res({ passthrough: true }) res: Response): Promise<{
message: string }> {
        return this.authService.logOut(res)
    }



//auth.dto
import {
    IsEmail,
    IsOptional,
    IsString,
    IsStrongPassword,
    Length,
} from 'class-validator'
import { IUser } from '@/shared/interfaces/user.interface'
import { Exclude } from 'class-transformer'

export class SignInDTO implements Partial<IUser> {
    @IsOptional()
    @IsEmail()
    email?: string

    @IsOptional()
    @Length(3, 30)
    nickname?: string

    @Exclude({ toPlainOnly: true })
    @IsString()
    password: string
}

export class CreateUserDto implements Partial<IUser> {
    @Length(3, 25)
```

```
    name: string

    @Length(3, 30)
    nickname: string

    @IsEmail()
    email: string

    @Exclude({ toPlainOnly: true })
    @IsStrongPassword({
        minLength: 6,
        minNumbers: 0,
        minUppercase: 0,
        minSymbols: 0,
    })
    password: string
}

export class ChangePasswordDTO {
    @Exclude({ toPlainOnly: true })
    @IsString()
    oldPassword: string

    @Exclude({ toPlainOnly: true })
    @IsStrongPassword({
        minLength: 6,
        minNumbers: 0,
        minUppercase: 0,
        minSymbols: 0,
    })
    newPassword: string
}
// auth.service
import {
    BadRequestException,
    Injectable,
    NotFoundException,
```

```
    UnauthorizedException,
} from '@nestjs/common'
import { UserService } from '@/modules/users/user.service'
import { FindOptionsWhere } from 'typeorm'
import { UserEntity } from '../users/user.entity'
import { JwtService } from '@nestjs/jwt'
import { ChangePasswordDTO, CreateUserDto } from './auth.dto'
import { compareHash, hash } from '@/shared/utils'
import { Response, Request, CookieOptions } from 'express'

@Injectable()
export class AuthService {

    isProduction: boolean

    constructor(
        private userService: UserService,
        private jwtService: JwtService,
    ) {
        this.isProduction = process.env.NODE_ENV === 'production'
    }

    async signIn(
        where: FindOptionsWhere<UserEntity>,
        pass: string,
        response: Response,
    ): Promise<{ access_token?: string; user: UserEntity }> {
        const user = await this.userService.findWhere(where)
        if (!(await compareHash(pass, user.password)))
            throw new UnauthorizedException('password missmatch')
        console.log({ user })
        const payload: IUserJwtPayload = { sub: user.id, email: user.email
}
        const access_token = await this.jwtService.signAsync(payload)

                            response.cookie('access_token',    access_token,
this.getCookieConfig())
```

```
        return {
            user,
            access_token: !!this.isProduction ? undefined : access_token,
        }
    }

    async register(createUserDTO: CreateUserDto): Promise<UserEntity> {
        await this.userService.checkUniqueFields(createUserDTO)
        const hashedPassword = await hash(createUserDTO.password)
        createUserDTO.password = hashedPassword
        return this.userService.usersRepository
            .save(createUserDTO)
            .catch(() => {
                throw new BadRequestException('Cannot create user')
            })
    }

    async changePassword(
        id: number,
        { newPassword, oldPassword }: ChangePasswordDTO,
    ): Promise<undefined> {
        const user = await this.userService.findWhere({ id })
        if (!(await compareHash(oldPassword, user.password)))
            throw new UnauthorizedException('password missmatch')
        const password = await hash(newPassword)

                                const    {    affected    }    =    await
this.userService.usersRepository.update(id, {
            password,
        })
        if (!affected) throw new NotFoundException('user not found')
    }

    async getProfile(request: Request) {
        return this.userService.findWhere({ id: request.jwt.sub })
    }
```

```
    async logOut(response: Response) {
        response.clearCookie('access_token', this.getCookieConfig())
        return { message: 'ok' }
    }

    private getCookieConfig() {

        return {
            httpOnly: true,
            secure: true,
            sameSite: this.isProduction ? 'none' : 'lax',
        } as CookieOptions
    }
}
```

ДОДАТОК Е

Лістинг модуля users (users.controller, users.entity, user.service, user.dto)

```
import {
    Body,
    Controller,
    Delete,
    Get,
    Param,
    ParseIntPipe,
    Patch,
    Query,
    UsePipes,
} from '@nestjs/common'
import { UserService } from './user.service'
import { UpdateUserDto } from './dto/user.dto'
import { ROUTES_USER } from '@/shared/constants/routes'
import { UserEntity } from './user.entity'
import { JWT } from '@/shared/decorators'
import { PageOptionsDto } from './dto/pageOptions.dto'

@Controller(ROUTES_USER.PREFIX)
export class UserController {
    constructor(private UserService: UserService) {}

    @Get(ROUTES_USER.GET_ALL_USERS)
    @UsePipes()
    async getAll(@Query() pageOptionsDto: PageOptionsDto) {
```

```
        return this.UserService.find(pageOptionsDto)
    }

    @Get(ROUTES_USER.GET_USER_BY_ID)
    async getByID(@Param('id', ParseIntPipe) id: number): Promise<UserEntity> {
        return this.UserService.findWhere({ id })
    }

    @Get(ROUTES_USER.GET_USER_BY_NAME)
    async getByName(@Param('name') name: string): Promise<UserEntity> {
        return this.UserService.findWhere({ name })
    }

    @Get(ROUTES_USER.GET_USER_BY_NICKNAME)
    async getByNickname(@Param('nickname') nickname: string): Promise<UserEntity> {
        return this.UserService.findWhere({ nickname })
    }

    @Get(ROUTES_USER.GET_USER_BY_EMAIL)
    async getByEmail(@Param('email') email: string): Promise<UserEntity> {
        return this.UserService.findWhere({ email })
    }

    @Patch(ROUTES_USER.PATCH_USER)
    async patchByID(
        @Body() createUserDto: UpdateUserDto,
        @JWT() jwt: IUserJwtPayload,
    ): Promise<UserEntity> {
        return this.UserService.updateForm(jwt.sub, createUserDto)
    }

    @Delete(ROUTES_USER.DELETE_USER)
    async deleteByID(@Param('id', ParseIntPipe) id: number): Promise<UserEntity> {
        return this.UserService.delete(id)
```

```
    }
}

//users.entity
import { IUser } from '@/shared/interfaces/user.interface'
import { hashSync } from '@/shared/utils'
import { Exclude } from 'class-transformer'
import { Column, Entity, OneToMany, PrimaryGeneratedColumn } from
'typeorm'
import { UserChatEntity } from '../chats/entities/user-chat.entity'
import { MessageEntity } from '../messages/message.entity'

@Entity('users')
export class UserEntity implements IUser {
    @PrimaryGeneratedColumn()
    id: number

    @OneToMany(() => MessageEntity, (MessageEntity) => MessageEntity.user)
    messages: MessageEntity[]

            @OneToMany(() => UserChatEntity, (UserChatEntity) =>
UserChatEntity.user)
    chats: UserChatEntity[]

    @Column('varchar', { length: 30 })
    name: string

    @Column('varchar', { length: 30, unique: true })
    nickname: string

    @Column('varchar', { length: 60, unique: true })
    email: string

    @Column('boolean', { default: false })
    is_online: boolean

          @Column('timestamp with time zone', { default: () =>
```

```
'CURRENT_TIMESTAMP' })
    online_at: Date

    @Exclude({ toPlainOnly: true })
    @Column('timestamp with time zone')
    created_at: Date

    @Column('varchar', { nullable: true, default: null })
    avatar_url?: string

    @Exclude({ toPlainOnly: true })
    @Column({
        type: 'text',
        default: () => {
            return `'${hashSync('Super_Secret!')}'`
        },
    })
    password: string
}
//users.dto
import { IsBoolean, IsEmail, IsOptional, IsString, Length } from
'class-validator'
import { IUser } from '@/shared/interfaces/user.interface'

export class UpdateUserDto implements Partial<IUser> {
    constructor(partial: Partial<UpdateUserDto>) {
        Object.assign(this, partial)
    }

    @IsOptional()
    @Length(3, 25)
    name?: string

    @IsOptional()
    @Length(3, 30)
    nickname?: string
```

```
    @IsOptional()
    @IsEmail()
    email?: string

    @IsOptional()
    @IsBoolean()
    is_online?: boolean

    // user avatar
    @IsOptional()
    avatar_url?: string
}
// users.service
import { ForbiddenException, Injectable, NotFoundException } from
'@nestjs/common'
import { InjectRepository } from '@nestjs/typeorm'
import { FindOptionsWhere, Repository, FindOneOptions, ILike } from
'typeorm'
import { IUser } from '@/shared/interfaces/user.interface'
import { UpdateUserDto } from './dto/user.dto'
import { UserEntity } from './user.entity'
import { PageOptionsDto } from './dto/pageOptions.dto'
import { PageDto } from './dto/page.dto'
import { PageMetaDto } from './dto/pageMeta.dto'

interface IFindWhereOptions {
    unique?: boolean
    throwOnNotFound?: boolean
    options?: FindOneOptions<UserEntity>
}

@Injectable()
export class UserService {
    constructor(
        @InjectRepository(UserEntity)
        readonly usersRepository: Repository<UserEntity>,
    ) {}
```

```
        async updateForm(id: IUser['id'], userBody: UpdateUserDto):
Promise<UserEntity> {
        // await this.findWhere({ id })
        await this.checkUniqueFields(userBody)
        const { affected } = await this.usersRepository.update(id, {
            ...userBody,
        })
        if (!affected) throw new NotFoundException('cannot update user')

        return this.findWhere({ id })
    }

    async delete(id: IUser['id']): Promise<UserEntity> {
        const user = await this.findWhere({ id })
            const { affected } = await this.usersRepository.delete({ id:
user.id })
        if (!affected) throw new NotFoundException('cannot delete user')
        return user
    }

                    async find(pageOptionsDto: PageOptionsDto):
Promise<PageDto<UserEntity>> {
        const { skip, take, order, where } = pageOptionsDto
        const iLike = ILike(`%${where}%`) // case insensitive
        const findCondition = where
            ? //find where email or nickname includes
                [{ email: iLike }, { nickname: iLike }]
            : // or find all
                undefined

        const [users, count] = await this.usersRepository.findAndCount({
            take,
            skip,
            where: findCondition,
            order: { created_at: order },
        })
```

```
        const pageMetaDto = new PageMetaDto({ count: count, pageOptionsDto
})

        return new PageDto(users, pageMetaDto)
    }

    /**
     * @param where find user by user field
      * @param IFindWhereOptions default: { throwOnNotFound: true, unique:
false}
     * @param errorMessage
     * @returns UserEntity OR Throtws error if user not found (by default)
     *- if unique: true - throws error already exists
     *- if throwOnNotFound: true - throws error not found
     */
    async findWhere(
        where: FindOptionsWhere<UserEntity>,
        { throwOnNotFound, unique, options }: IFindWhereOptions = {
            throwOnNotFound: true,
            unique: false,
            options: {},
        },
        errorMessage?: string,
    ): Promise<UserEntity> {
              const user = await this.usersRepository.findOne({ where,
...options })

                        if (throwOnNotFound && !user) throw new
NotFoundException(errorMessage)
         if (unique && user) throw new ForbiddenException(errorMessage ??
`already exists`)

        return user as UserEntity
    }

    /**
     * @param user user from dto ur entity like
```

```
     * @returns find user with provided unique keys
     * - throws an error if fields {email, nickname} already exists
     */
    async checkUniqueFields(user: Partial<IUser>): Promise<UserEntity[]> {
        const { email, nickname } = user
        const promises: Promise<UserEntity>[] = []
        if (email)
                promises.push(this.findWhere({ email }, { unique: true },
'email must be unique'))
        if (nickname)
             promises.push(this.findWhere({ nickname }, { unique: true },
'nickname must be unique'))

        return Promise.all(promises)
    }

    async toggleActivity(id: IUser['id'], isOnline: boolean) {
        const updateBody: Partial<IUser> = { is_online: isOnline }
        if (!isOnline) updateBody.online_at = new Date()
        return this.usersRepository.update({ id }, updateBody)
    }
}
```

ДОДАТОК Є

Лістинг файлів для пагінації (page.dto, pageMeta.dto, pageOptions.dto)

```
import { IsArray } from 'class-validator'
import { PageMetaDto } from './pageMeta.dto'
import { PageOptionsDto } from './pageOptions.dto'

export class PageDto<T> {
    @IsArray()
    readonly data: T[]

    readonly meta: PageMetaDto

    constructor(data: T[], meta: PageMetaDto) {
        this.data = data
        this.meta = meta
    }
}

export const getEmptyPage = (pageOptionsDto: PageOptionsDto = new
PageOptionsDto()) =>
    new PageDto([], new PageMetaDto({ count: 0, pageOptionsDto }))
```

```
import { PageMetaDtoParameters } from './pageMetaDtoParams.interface'

export class PageMetaDto {
    readonly page: number

    readonly take: number

    readonly count: number

    readonly pageCount: number

    readonly hasPreviousPage: boolean

    readonly hasNextPage: boolean

    constructor({ count, pageOptionsDto }: PageMetaDtoParameters) {
        this.page = pageOptionsDto.page
        this.take = pageOptionsDto.take
        this.count = count
        this.pageCount = Math.ceil(this.count / this.take)
        this.hasPreviousPage = this.page > 1
        this.hasNextPage = this.page < this.pageCount
    }
}


export interface PageMetaDtoParameters {
    pageOptionsDto: PageOptionsDto
    count: number
}

import { Expose, Transform, Type } from 'class-transformer'
import { IsBoolean, IsIn, IsInt, IsOptional, Length, Max, Min } from
'class-validator'
```

```
const order = ['DESC', 'ASC'] as const
export type OrderOptionsType = (typeof order)[number]

export class IncludeDTO {
    @IsOptional()
    @IsBoolean()
    @Expose({ name: 'members' })
    @Transform(({ value }) => value === 'true')
    user_chats?: boolean = false

    @IsOptional()
    @IsBoolean()
    @Transform(({ value }) => value === 'true')
    messages?: boolean = false


    @IsOptional()
    @IsBoolean()
    @Transform(({ value }) => value === 'true')
    user?: boolean = false
}

export class PageOptionsDto {
    @IsOptional()
    @IsIn(order)
    order?: OrderOptionsType = 'DESC'

    @Type(() => Number)
    @IsInt()
    @Min(1)
    @IsOptional()
    page: number = 1

    @Type(() => Number)
    @IsInt()
    @Min(1)
```

```
    @Max(50)
    @IsOptional()
    take: number = 10

    @Length(1, 30)
    @IsOptional()
    where?: string

    @IsOptional()
    @Type(() => IncludeDTO)
    include?: IncludeDTO

    get skip(): number {
        return (this.page - 1) * this.take
    }
    constructor(include?: IncludeDTO) {
        this.include = include
    }
}
```

ДОДАТОК Ж

Лістинг модуля chats (chats.controller, chats.service, chats.repository)

```
import { JWT } from '@/shared/decorators'
import {
    BadRequestException,
    Body,
    Controller,
    Delete,
    Param,
    ParseIntPipe,
    Patch,
    Post,
    Query,
} from '@nestjs/common'
import { Get } from '@nestjs/common'
import { ChatsService } from './chats.service'
import { CreateChatDTO } from './dto/create-chat.dto'
import { MembersDTO } from './dto/members.dto'
import { PageOptionsDto } from '../users/dto/pageOptions.dto'
import { UpdateMemberDTO } from './dto/update-member.dto'
```

```
@Controller('chats')
export class ChatsController {
    constructor(private chatsService: ChatsService) {}

    @Get()
            getChats(@Query()   options:   PageOptionsDto,   @JWT()   jwt:
IUserJwtPayload) {
        return this.chatsService.getChats(options, jwt)
    }

    @Get(':id')
    getChatByID(
        @Param('id', ParseIntPipe) chatID: number,
        @JWT() jwt: IUserJwtPayload,
        @Query() options: PageOptionsDto,
    ) {
        return this.chatsService.getChatById(chatID, options, jwt)
    }

    @Delete(':id')
        deleteChat(@Param('id',  ParseIntPipe)  id:  number,  @JWT()  jwt:
IUserJwtPayload) {
        if (!id) throw new BadRequestException('param id not provided')
        return this.chatsService.deleteChat(id, jwt)
    }

    @Post()
         createChat(@Body()  createChatDTO:  CreateChatDTO,  @JWT()  jwt:
IUserJwtPayload) {
        return this.chatsService.createChat(createChatDTO, jwt)
    }
}

@Controller('members')
export class MembersController {
    constructor(private chatsService: ChatsService) {}
```

```
    @Patch(':id')
    updateUser(
        @Param('id', ParseIntPipe) chatId: number,
        @Body() updateDTO: UpdateMemberDTO,
        @JWT() jwt: IUserJwtPayload,
    ) {
        return this.chatsService.updateChat(chatId, jwt, updateDTO)
    }

    @Post()
    inviteUser(@Body() inviteDTO: MembersDTO, @JWT() jwt: IUserJwtPayload)
{
        return this.chatsService.inviteUser(inviteDTO, jwt)
    }

    @Delete(':id')
    deleteUsers(
        @Param('id', ParseIntPipe) userID: number,
        @Query('chat', ParseIntPipe) chatID: number,
        @JWT() jwt: IUserJwtPayload,
    ) {
        console.log({ userID, chatID })
        return this.chatsService.deleteUsers([userID], chatID, jwt)
    }
}

// chats.service
import { UpdateMemberDTO } from './dto/update-member.dto';
import {
    BadRequestException,
    ForbiddenException,
    Injectable,
    NotFoundException,
} from '@nestjs/common'
import { CreateChatDTO } from './dto/create-chat.dto'
import { MembersDTO, MemberDTO } from './dto/members.dto'
```

```
import { ChatsRepository } from './chats.repository'
import { PageOptionsDto } from '../users/dto/pageOptions.dto'

type CreateInviteList = (creatorID: number, members: MemberDTO[]) =>
MemberDTO[]

type MapInviteList = Record<ChatPrivacyType, CreateInviteList>

@Injectable()
export class ChatsService {
    constructor(private chatsRepository: ChatsRepository) {}

    async createChat(chatDTO: CreateChatDTO, jwt: IUserJwtPayload) {
        const role: UserRoleType = chatDTO.type === 'private' ? 'member' :
'admin'
        const dto = { id: jwt.sub, role }

        const [chat, members] = await this.chatsRepository.transaction([
            async (repo) => repo.createChat(chatDTO),
                async (repo, [newChat]) => repo.inviteUsers(newChat.id,
[dto]),
        ])

        return { chat }
    }

    /*** get all chats where user exists */
    async getChats(options: PageOptionsDto, jwt: IUserJwtPayload) {
        const chatIds = await this.chatsRepository.getUserChatsIds(jwt)
        if (!options.include) return { chats: chatIds }
            const chats = await this.chatsRepository.getChats(chatIds,
options)
        return { chats }
    }

        async getChatById(id: number, options: PageOptionsDto, jwt:
IUserJwtPayload) {
```

```
        const [chat] = await this.chatsRepository.getChats([id], options)
        if (!chat?.id) throw new NotFoundException('chat not found')

            const userInChat = chat.members?.data.find(({ user_id }) =>
user_id === jwt.sub)
        if (!userInChat) throw new ForbiddenException('acess denied')
        return { chat }
    }

     async updateChat(chatId: number, jwt: IUserJwtPayload, memberBody:
UpdateMemberDTO) {
         const res = await this.chatsRepository.updateMemberInChat(chatId,
jwt.sub, memberBody)

        return res
    }

    async inviteUser(inviteDTO: MembersDTO, jwt: IUserJwtPayload) {
                await this.chatsRepository.getUserInChatOrFail(jwt.sub,
inviteDTO.chat_id)

               await this.chatsRepository.inviteUsers(inviteDTO.chat_id,
inviteDTO.members)

        const chat = await this.chatsRepository.getChats(
            [inviteDTO.chat_id],
            new PageOptionsDto({ user_chats: true, messages: true }),
        )

        return { chat }
    }

          async deleteUsers(users: number[], chatID: number, jwt:
IUserJwtPayload) {
                           const { chat, ...user } = await
this.chatsRepository.getUserInChatOrFail(jwt.sub, chatID)
          const affected = await this.chatsRepository.deleteUsers(users,
```

```
chatID)
        return { affected }
    }

    async deleteChat(chatID: number, jwt: IUserJwtPayload) {
                                        const     user     =     await
this.chatsRepository.getUserInChatOrFail(jwt.sub, chatID)

        if (user.chat.type === 'public' && user.role !== 'admin')
            throw new ForbiddenException('public: user must be admin')

        const { affected } = await this.chatsRepository.deleteChat(chatID)

          if (affected !== 1) throw new NotFoundException('Cannot delete
chat')

        return { affected }
    }
}
//chats.repository
import { UpdateMemberDTO } from './dto/update-member.dto'
import { PageOptionsDto } from './../users/dto/pageOptions.dto'
import { FindOptionsRelations, In, Repository } from 'typeorm'
import { ChatEntity } from './entities/chat.entity'
import {
    BadRequestException,
    ConflictException,
    Injectable,
    NotFoundException,
} from '@nestjs/common'
import { InjectRepository } from '@nestjs/typeorm'
import { UserChatEntity } from './entities/user-chat.entity'
import { CreateChatDTO } from './dto/create-chat.dto'
import { MemberDTO } from './dto/members.dto'
import { PageMetaDto } from '../users/dto/pageMeta.dto'
import { PageDto } from '../users/dto/page.dto'
import  {  convertToUserChatEntity,  GetChatsResponse,  selectChats  }  from
```

```
'./utils'
import { NotificationService } from '../router/notification.service'

@Injectable()
export class ChatsRepository {
    constructor(
        @InjectRepository(ChatEntity)
        private readonly chats: Repository<ChatEntity>,

        @InjectRepository(UserChatEntity)
        private readonly members: Repository<UserChatEntity>,

        private notification: NotificationService,
        // private socket: UserSocketManager,
    ) {}

    async createChat({ ...chat }: CreateChatDTO): Promise<ChatEntity> {
        return this.chats.save(chat)
    }

    async getUserInChatOrFail(
        userID: number,
        chatID: number,
        include?: FindOptionsRelations<UserChatEntity>,
        role?: UserRoleType[],
    ) {
        const relations = include ?? { chat: true }

        const user = await this.members.findOne({
            where: {
                user_id: userID,
                chat_id: chatID,
                ...(role && { role: In(role) }),
            },
            relations,
        })
```

```
        if (!user) throw new NotFoundException('user or chat not found')

        return user as Required<Omit<UserChatEntity, 'user'>>
    }

    async getUserChatsIds(jwt: IUserJwtPayload): Promise<number[]> {
        return this.members
            .find({
                select: { chat_id: true },
                where: { user_id: jwt.sub },
            })
            .then((res) => res.map(({ chat_id }) => chat_id))
    }

    async inviteUsers(chatID: number, members: MemberDTO[]) {
        const usersToInvite = convertToUserChatEntity(members, chatID)
        console.log({ usersToInvite })

          const insert = await this.members.insert(usersToInvite).catch(()
=> {
             throw new ConflictException('fail: user exists in chat or not
found')
        })

            const idsUsersToInvite = usersToInvite.map(({ user_id }) =>
user_id)

        this.notification.notifyNewChat(chatID, idsUsersToInvite)

        return insert.identifiers
    }

    async deleteChat(chatID: number) {
        const deleteRes = await this.chats.delete({ id: chatID }).catch(()
=> {
            throw new NotFoundException('fail to delete chat')
        })
```

```
        this.notification.notifyDeleteChat(chatID)

        return deleteRes
    }

    async deleteUsers(ids: number[], chatId: number) {
        const { affected } = await this.members
            .delete({ user_id: In(ids), chat_id: chatId })
            .catch(() => {
                    throw new NotFoundException('fail to remove user from
chat')
            })

        if (!affected) throw new NotFoundException('user not found')

        this.notification.notifyDeleteToMembers(chatId, ids)

        return affected
    }

     /** utility method for find all chats with pagination of members and
chats*/
        async getChats(ids: number[], pageOptionsDto: PageOptionsDto):
Promise<GetChatsResponse[]> {
        if (!ids.length) return []

        const raw = selectChats(ids, pageOptionsDto)
                            const query: Array<ChatEntity> = await
this.chats.query(raw).catch((err) => {
            console.log(err)
                  throw new BadRequestException('fail to get chats with
messages')
        })

        const chats = query.map(({ messages, users_chats, ...chat }) => {
            const res: any = { ...chat }
```

```
            messages = messages ?? []
            users_chats = users_chats ?? []

            if (pageOptionsDto.include?.messages) {
                const count = chat.total_messages
                const metaProps = { count, pageOptionsDto }
                const messagesMeta = new PageMetaDto(metaProps)
                res.messages = new PageDto(messages, messagesMeta)
            }

            if (pageOptionsDto.include?.user_chats) {
                const count = chat.total_members
                const metaProps = { count, pageOptionsDto }
                const membersMeta = new PageMetaDto(metaProps)
                res.members = new PageDto(users_chats, membersMeta)
            }

            return res
        }) as GetChatsResponse[]

        return chats
    }

    async getInChatMemberIds(chatID: number) {
        return this.members.find({
            // select: { user_id: true },
            where: { chat_id: chatID },
        })
    }

     async updateMemberInChat(chatId: number, userId: number, updateBody:
UpdateMemberDTO) {
        if (updateBody.last_read) updateBody.last_read_chat = new Date()

        const updateRes = await this.members
            .update(
                { chat_id: chatId, user_id: userId },
```

```
                {
                    role: 'member',
                    last_read_chat: new Date(),
                },
            )
            .catch((err) => {
                console.log('error in update member', err)
                throw new NotFoundException('not found')
            })

        if (updateBody.last_read_chat) {
        }

        return updateRes
    }

    /*** if one of call backs fails it roll back all queries
     ** also provide array with awaited results */
     async transaction<T extends any[]>(operations: OperationsTuple<T>):
Promise<T> {
                                        const     queryRunner     =
this.chats.manager.connection.createQueryRunner()
        await queryRunner.connect()
        await queryRunner.startTransaction()
        try {
            const chats = queryRunner.manager.getRepository(ChatEntity)
                                                const    members    =
queryRunner.manager.getRepository(UserChatEntity)
            const transactionalRepository = new ChatsRepository(
                chats,
                members,
                this.notification,
                // this.socket,
            )

            const results: any[] = []
            for (const operation of operations) {
```

```
                const result = await operation(transactionalRepository,
results)
                results.push(result)
            }
            await queryRunner.commitTransaction()
             return results as T // Safe cast, as TypeScript infers T as a
tuple
        } catch (error) {
            await queryRunner.rollbackTransaction()
            throw error
        } finally {
            await queryRunner.release()
        }
    }
}

// Type for a single operation
type Operation<T> = (repo: ChatsRepository, results: any[]) => Promise<T>

// Type for an array of operations, mapping to a tuple of return types
type OperationsTuple<T extends any[]> = {
    [K in keyof T]: Operation<T[K]>
}
```

Лістинг файлів модуля messages (messages.controller, messages.service, messages.service)

```
import { Body, Controller, Delete, Param, ParseIntPipe, Patch, Post } from
'@nestjs/common'
import { MessageService } from './MessageService'
import { JWT } from '@/shared/decorators'
import { CreateMessageDTO } from './dto/create-message.dto'
import { UpdateMessageDTO } from './dto/update-message.dto'

@Controller('messages')
export class MessagesController {
```

```
    constructor(private messageService: MessageService) {}

    @Post()
      sendMessage(@Body() createMessageDTO: CreateMessageDTO, @JWT() jwt:
IUserJwtPayload) {
        return this.messageService.sendMessage(createMessageDTO, jwt)
    }

    @Patch(':id')
    updateMessage(
        @Param('id', ParseIntPipe) messageID: number,
        @Body() updateMessageDTO: UpdateMessageDTO,
        @JWT() jwt: IUserJwtPayload,
    ) {
                    return   this.messageService.updateMessage(messageID,
updateMessageDTO, jwt)
    }

    @Delete(':id')
     deleteMessage(@Param('id', ParseIntPipe) chatID: number, @JWT() jwt:
IUserJwtPayload) {
        return this.messageService.deleteMessage(chatID, jwt)
    }
}

//messages.service
import { NotificationService } from './../router/notification.service';
import { ForbiddenException, Injectable } from '@nestjs/common'
import { MessagesRepository } from './messages.repository'
import { ChatsRepository } from '../chats/chats.repository'
import { CreateMessageDTO } from './dto/create-message.dto'
import { UpdateMessageDTO } from './dto/update-message.dto'

@Injectable()
export class MessageService {
    constructor(
        private messageRepository: MessagesRepository,
```

```
        private chatRepository: ChatsRepository,
        private notification: NotificationService
    ) {}

          async   sendMessage(createMessageDTO:   CreateMessageDTO,   jwt:
IUserJwtPayload) {
                  await  this.chatRepository.getUserInChatOrFail(jwt.sub,
createMessageDTO.chat_id, {})

        createMessageDTO.user_id = jwt.sub
                                        const     message     =     await
this.messageRepository.sendMessage(createMessageDTO)

        this.notification.notifyNewMessage(message.chat_id, message)

        return { message }
    }

    async updateMessage(
        messageID: number,
        updateMessageDTO: UpdateMessageDTO,
        jwt: IUserJwtPayload,
    ) {
                                        const     message     =     await
this.messageRepository.getMessageByIdOrFail(messageID)
        if (message.user_id !== jwt.sub)
            throw new ForbiddenException('cannot patch not own messages')

                                    const     patchResponse     =     await
this.messageRepository.patchMessage(messageID, updateMessageDTO)

                    this.notification.notifyPatchMessage(message.chat_id,
patchResponse)

        return { message: patchResponse }
    }
```

```
    async deleteMessage(messageID: number, jwt: IUserJwtPayload) {
                                        const     message     =     await
this.messageRepository.getMessageByIdOrFail(messageID)

                        if   (jwt.sub   !==   message.user_id)   throw   new
ForbiddenException('access denied')

                                    const    {    affected    }    =    await
this.messageRepository.deleteMessage(message.id)

        this.notification.notifyDeleteMessage(message.chat_id, message)

        return { affected }
    }
}


//messages.repository
import { PageDto } from './../users/dto/page.dto'
import { PageOptionsDto } from '../users/dto/pageOptions.dto'
import   {   FindOptionsRelations,   FindOptionsWhere,   In,   QueryBuilder,
Repository } from 'typeorm'
import {
    ConflictException,
    Injectable,
    NotFoundException,
} from '@nestjs/common'
import { InjectRepository } from '@nestjs/typeorm'
import { PageMetaDto } from '../users/dto/pageMeta.dto'
import { MessageEntity } from './message.entity'
import { CreateMessageDTO } from './dto/create-message.dto'
import { UpdateMessageDTO } from './dto/update-message.dto'
import { ChatEntity } from '../chats/entities/chat.entity'


@Injectable()
```

```
export class MessagesRepository {
    constructor(
        @InjectRepository(MessageEntity)
        private readonly messages: Repository<MessageEntity>,
    ) {}

            async   getMessagesByChatID(chatID:   number,   pageOptionsDto:
PageOptionsDto) {
        const { skip, take, order } = pageOptionsDto
        const [chats, count] = await this.messages.findAndCount({
            skip,
            take,
            order: { created_at: order },
            where: { chat_id: chatID },
        })
               const  pageMetaDataDTO  =  new  PageMetaDto({  count:  count,
pageOptionsDto })
        return new PageDto(chats, pageMetaDataDTO)
    }

            async   getMessageByIdOrFail(messageID:   number,   relations?:
FindOptionsRelations<MessageEntity>) {
             const  message  =  await  this.messages.findOne({  where:  {  id:
messageID }, relations })
        if (!message) throw new NotFoundException('Message not found')
        return message
    }

    async sendMessage(message: CreateMessageDTO | CreateMessageDTO[]) {
                                const    {    generatedMaps    }    =    await
this.messages.insert(message).catch(() => {
            throw new ConflictException('chat or user not found')
        })

                           const   maps   =   {   ...generatedMaps[0],   id:
Number(generatedMaps[0].id) }
```

```
        return { ...message, ...maps } as MessageEntity
    }
    async patchMessage(id: number, body: UpdateMessageDTO) {
        const res = await this.messages.update(id, body).catch(() => {
            throw new NotFoundException('fail to update message')
        })

        const response = { id, ...body, ...res.raw[0] }
        return response as Partial<MessageEntity>
    }

    async deleteMessage(messageID: number | number[]) {
        return await this.messages.delete(messageID).catch(() => {
            throw new NotFoundException('chat not found')
        })
    }
}
```

ДОДАТОК З

Лістинг файлів модуля router (gateway), що відповідає за підключення до websocket, управління кімнатами та отримання/відправлення повідомлень в реальному часі.

Лістинг файлів app.gateway, user-socket-manager-service, notification.service.

```
import {
    WebSocketGateway,
    WebSocketServer,
    SubscribeMessage,
    MessageBody,
    ConnectedSocket,
} from '@nestjs/websockets'
import { Server, Socket } from 'socket.io'
import { Logger, Injectable } from '@nestjs/common'
import { BaseGateway } from './base.gateway'
import { UserSocketManager } from './user-socket-manager.service'
import { NotificationService } from './notification.service'
import { cors } from '@/config/cors.config' // Update path
import { ChatsRepository } from '../chats/chats.repository'
import { UserService } from '../users/user.service'

@WebSocketGateway({
    namespace: 'users',
    cors: { ...cors },
})
@Injectable() // Gateways are NestJS providers
export class AppGateway extends BaseGateway {
    @WebSocketServer()
    private server!: Server // NestJS injects the server instance

    private readonly logger = new Logger(AppGateway.name)

    constructor(
        private readonly userSocketManager: UserSocketManager,
        private readonly notificationService: NotificationService,
        private readonly chatsRepository: ChatsRepository,
        private readonly usersRepository: UserService,
    ) {
        super('APP_GATEWAY') // Name for BaseGateway's logging
```

```
    }

    afterInit(server: Server) {
        super.afterInit(server) // Call base class method
        this.notificationService.initialize(this.server) // Provide server
instance to NotificationService
          this.logger.log('AppGateway initialized and NotificationService
linked.')
    }

          async   handleConnection(@ConnectedSocket()   client:   Socket):
Promise<void> {
        this.logger.debug(`Socket ${client.id} attempting to connect.`)
        const jwt = await super.handleConnection(client)

        if (!jwt?.sub) {
                  const  log  =  `Socket  ${client.id}  connection  refused:
Authentication failed or no user ID.`
            this.logger.warn(log)
            client.disconnect(true)
            return
        }

        const userId = jwt.sub
        this.userSocketManager.registerUser(userId, client)
          this.logger.log(`User ${userId} (Socket ${client.id}) connected
successfully.`)
        const chatIds = await this.chatsRepository.getUserChatsIds(jwt)

         // Example: Join user to their predefined rooms (e.g., all their
chats)
        const roomIdsToJoin = chatIds.map((id) => `room_${id}`)

        // if (roomIdsToJoin.length > 0) {
        await this.userSocketManager.joinRooms(userId, roomIdsToJoin)
                this.logger.log(`User  ${userId}  joined  initial  rooms:
${roomIdsToJoin.join(', ')}.`)
```

```
        // Emit 'online' status to these rooms
        // }

        this.broadcastOnlineStatusToRooms(roomIdsToJoin, userId, true)

        this.userSocketManager.logState()
    }

          async   handleDisconnect(@ConnectedSocket()   client:   Socket):
Promise<any> {
        const userId = (await super.handleDisconnect(client)).sub
        if (!userId) {
             this.logger.warn(`Socket ${client.id} disconnected without a
clear User ID.`)
            return
        }

                this.logger.log(`User  ${userId}  (Socket  ${client.id})
disconnecting.`)
                                          const     roomsUserWasIn     =
this.userSocketManager.getRoomsForUser(userId)
         this.userSocketManager.unregisterUser(userId) // This also cleans
internal room subscriptions

        // Socket.IO automatically removes client from all its server-side
rooms on disconnect.
        // No need to call client.leave() for all rooms here explicitly.

        const log = `User ${userId} was in rooms: ${roomsUserWasIn.join(',
')}. Broadcasting offline status.`
        this.logger.log(log)

        this.broadcastOnlineStatusToRooms(roomsUserWasIn, userId, false)
        this.userSocketManager.logState()
    }

    /**
```

```
     * Broadcasts the online status to a list of specified rooms.
     * @param roomIds The IDs of the rooms to notify.
      * @param actingUserId The user ID whose status change triggered this
broadcast.
       * @param isOnline Indicates if the acting user is now online or
offline.
     */
    private broadcastOnlineStatusToRooms(
        roomIds: string[],
        actingUserId: number,
        isOnline: boolean,
    ): void {
        this.usersRepository.toggleActivity(actingUserId, isOnline)

          this.notificationService.notifyUserStatus(roomIds, actingUserId,
isOnline)
    }

    // Example of a client subscribing to a specific chat room dynamically
    @SubscribeMessage('chats:join')
    async handleJoinChat(
        @ConnectedSocket() client: Socket,
        @MessageBody() data: { chatId: number | string },
    ): Promise<{ status: string; message: string }> {
        const userId = client.handshake.auth.sub
        const roomId = `room_${data.chatId}`

        await this.userSocketManager.joinRoom(userId, roomId)
        this.logger.log(`User ${userId} subscribed to ${roomId}`)
        // Notify everyone in that room about the updated online list
        this.broadcastOnlineStatusToRooms([roomId], userId, true)
        return { status: 'success', message: `Joined ${roomId}` }
    }

}
```

```
// user-socket-manager.service
import { Injectable, Logger } from '@nestjs/common'
import { Socket } from 'socket.io'

@Injectable()
export class UserSocketManager {
    private readonly logger = new Logger(UserSocketManager.name)
    // Map<userId (number), Socket instance>
    private connectedUsers: Map<number, Socket> = new Map()
    // Map<roomId (string), Set<userId (number)>>
    private roomSubscriptions: Map<string, Set<number>> = new Map()

    registerUser(userId: number, client: Socket): void {
         // If user is already connected with an old socket, disconnect it
(optional, depends on desired behavior)
        const existingSocket = this.connectedUsers.get(userId)
        if (existingSocket && existingSocket.id !== client.id) {
            this.logger.warn(
                 `User ${userId} reconnected with new socket ${client.id},
old socket ${existingSocket.id} will be implicitly disconnected or managed
by socket.io.`,
            )
            // existingSocket.disconnect(true); // Or handle as needed
        }
        this.connectedUsers.set(userId, client)
               this.logger.log(`User  registered:  ${userId},  Socket  ID:
${client.id}`)
    }

    unregisterUser(userId: number): Socket | undefined {
        const client = this.connectedUsers.get(userId)
        if (client) {
            this.connectedUsers.delete(userId)
                this.logger.log(`User unregistered: ${userId}, Socket ID:
${client.id}`)

            // Clean up user from all internal room subscriptions
```

```
            this.roomSubscriptions.forEach((userIdsInRoom, roomId) => {
                if (userIdsInRoom.has(userId)) {
                    userIdsInRoom.delete(userId)
                    this.logger.log(
                            `User ${userId} removed from internal room
subscription: ${roomId}`,
                    )
                    if (userIdsInRoom.size === 0) {
                        this.roomSubscriptions.delete(roomId)
                          this.logger.log(`Internal room ${roomId} deleted
(empty).`)
                    }
                }
            })
        } else {
            this.logger.warn(
                        `Attempted to unregister non-existent or already
unregistered user: ${userId}`,
            )
        }
        return client
    }


    async joinRoom(userId: number, roomId: string): Promise<boolean> {
        const client = this.connectedUsers.get(userId)
        if (!client) {
             this.logger.error(`Cannot join room ${roomId}: User ${userId}
not connected.`)
            return false
        }

        if (!this.roomSubscriptions.has(roomId)) {
            this.roomSubscriptions.set(roomId, new Set())
        }
        this.roomSubscriptions.get(roomId)!.add(userId)
        await client.join(roomId) // Actual Socket.IO join
```

```
        this.logger.log(
              `User ${userId} (Socket ${client.id}) joined Socket.IO room
${roomId}. Client rooms: ${Array.from(client.rooms)}`,
        )
        return true
    }

    async joinRooms(userId: number, roomIds: string[]): Promise<void> {
        for (const roomId of roomIds) {
            await this.joinRoom(userId, roomId)
        }
    }

    async leaveRoom(userId: number, roomId: string): Promise<boolean> {
        const client = this.connectedUsers.get(userId)
         // Client might be null if unregisterUser was called first during
a disconnect
        if (client && client.connected && client.rooms.has(roomId)) {
            // Check if client is still connected and in room
            await client.leave(roomId)
            this.logger.log(
                 `User ${userId} (Socket ${client.id}) left Socket.IO room
${roomId}. Client rooms: ${Array.from(client.rooms)}`,
            )
        } else if (client) {
            this.logger.log(
                `User ${userId} (Socket ${client.id}) was not in Socket.IO
room ${roomId} to leave, or already disconnected.`,
            )
        }

        const room = this.roomSubscriptions.get(roomId)
        if (room) {
            room.delete(userId)
               this.logger.log(`User ${userId} removed from internal room
subscription: ${roomId}`)
            if (room.size === 0) {
```

```
                this.roomSubscriptions.delete(roomId)
                        this.logger.log(`Internal room ${roomId} deleted
(empty).`)
            }
        }
        return true
    }

    async leaveRooms(userId: number, roomIds: string[]): Promise<void> {
        for (const roomId of roomIds) {
            await this.leaveRoom(userId, roomId)
        }
    }

    getSocketByUserId(userId: number): Socket | undefined {
        return this.connectedUsers.get(userId)
    }

    getUserIdsInRoom(roomId: string): number[] {
        return Array.from(this.roomSubscriptions.get(roomId) || [])
    }

    // Utility to get all rooms a user is internally subscribed to
    getRoomsForUser(userId: number): string[] {
        const rooms: string[] = []
        this.roomSubscriptions.forEach((users, roomId) => {
            if (users.has(userId)) {
                rooms.push(roomId)
            }
        })
        return rooms
    }

    // get all unique users across a list of rooms
    getAllUserIdsInMultipleRooms(roomIds: string[]): number[] {
        const uniqueUserIds = new Set<number>()
        for (const roomId of roomIds) {
```

```
            const usersInRoom = this.roomSubscriptions.get(roomId)
            if (usersInRoom) {
                usersInRoom.forEach((userId) => uniqueUserIds.add(userId))
            }
        }
        return Array.from(uniqueUserIds)
    }

    logState(): void {
        // For debugging
                                    this.logger.debug(`Connected    Users:
${Array.from(this.connectedUsers.keys())}`)
        const roomSubs: Record<string, number[]> = {}
        this.roomSubscriptions.forEach((value, key) => {
            roomSubs[key] = Array.from(value)
        })
                                this.logger.debug(`Room    Subscriptions:
${JSON.stringify(roomSubs)}`)
    }
}

// notification.service
import   {   Injectable,   InternalServerErrorException,   Logger   }   from
'@nestjs/common'
import { Server } from 'socket.io'
import { UserSocketManager } from './user-socket-manager.service'
import { MessageEntity } from '../messages/message.entity'

@Injectable()
export class NotificationService {
    private readonly logger = new Logger(NotificationService.name)
    private server!: Server // Initialized by AppGateway

    constructor(private readonly socket: UserSocketManager) {}

    initialize(server: Server): void {
        this.server = server
```

```
        this.logger.log('NotificationService initialized with Socket.IO Server instance.')
    }
    //** notify and add sockets to this room */
    async notifyNewChat(chatId: number, usersToInvite: number[]) {
        const roomId = `room_${chatId}`

        //join all invited members into room
            const promisesToJoin = usersToInvite.map(async (id) => this.socket.joinRoom(id, roomId))
        // wait until all joined
        await Promise.all(promisesToJoin)

        this.emitToRoom(roomId, 'chats:new', { chat: chatId })
    }

    //** notify and leave all members of this chat */
    notifyDeleteChat(chatId: number) {
        const roomId = `room_${chatId}`
        this.emitToRoom(roomId, 'chats:delete', chatId)
        const connected = this.socket.getUserIdsInRoom(roomId)
        connected.forEach((id) => this.socket.leaveRoom(id, roomId))
    }

    //** notify and delete given sockets from chat */
    notifyDeleteToMembers(chatId: number, usersToDelete: number[]) {
        const roomId = `room_${chatId}`

        usersToDelete.forEach((id) => {
            const messageBody = { removedUser: id, chatId }
            this.emitToRoom(roomId, 'members:delete', messageBody)
            this.socket.leaveRoom(id, roomId)
        })
    }

     notifyUserStatus(roomIds: string[], actingUserId: number, isOnline: boolean) {
```

```
        this.logger.debug(
            `Broadcasting online status for user ${actingUserId} (online:
${isOnline}) to rooms: ${roomIds.join(', ')}`,
        )

        const event = isOnline ? 'online' : 'offline'
        this.broadcastToListOfRooms(roomIds, event, (roomId) => {
            const usersInThisRoom = this.socket.getUserIdsInRoom(roomId)
            return isOnline ? usersInThisRoom : actingUserId
        })
    }

    notifyNewMessage(chatId: number, message: MessageEntity) {
        this.emitToRoom(`room_${chatId}`, 'message:new', { message })
    }

    notifyDeleteMessage(chatId: number, message: MessageEntity) {
        this.emitToRoom(`room_${chatId}`, 'message:delete', { message })
    }

    notifyPatchMessage(chatId: number, message: Partial<MessageEntity>) {
        this.emitToRoom(`room_${chatId}`, 'message:patch', { message })
    }

    private emitToUser(userId: number, event: string, data: any): boolean
{
        this.isInitialized()

        const clientSocket = this.socket.getSocketByUserId(userId)
        if (clientSocket && clientSocket.connected) {
            clientSocket.emit(event, data)
            this.logger.log(
                    `Emitted event '${event}' to user ${userId} (Socket
${clientSocket.id})`,
            )
            return true
        }
```

```
        this.logger.warn(
            `Failed to emit event '${event}': User ${userId} not connected
or socket not found.`,
        )
        return false
    }

    private emitToUsers(userIds: number[], event: string, data: any): void
{
        this.isInitialized()

        userIds.forEach((userId) => this.emitToUser(userId, event, data))
    }

        private  emitToRoom(roomId:  string,  event:  string,  data:  any,
exceptSocketId?: string): void {
        this.isInitialized()

        let emitter = this.server.to(roomId)
        if (exceptSocketId) {
            emitter = emitter.except(exceptSocketId)
        }
        emitter.emit(event, data)
        this.logger.log(
            `Emitted event '${event}' to room ${roomId}` +
                (exceptSocketId ? ` (except ${exceptSocketId})` : ''),
        )
    }

    private broadcastToListOfRooms(
        roomIds: string[],
        event: string,
        dataFactory: (roomId: string) => any,
    ): void {
        this.isInitialized()
        for (const roomId of roomIds) {
            const eventData = dataFactory(roomId)
```

```
            this.server.to(roomId).emit(event, eventData)
                this.logger.log(`Broadcasted event '${event}' to room
${roomId}`)
        }
    }

    private isInitialized(): boolean {
        if (!this.server) {
                        this.logger.error('Server not initialized in
NotificationService.')
            throw new InternalServerErrorException('Server not initialized
in NotificationService.')
        }
        return true
    }
}
```