

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ЗАКЛАД «ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА»

Факультет технологій та інформаційних систем

Кафедра інформаційних технологій та систем

Хомюк Іван Вікторович

Розробка веб-застосунку для обміну повідомленнями

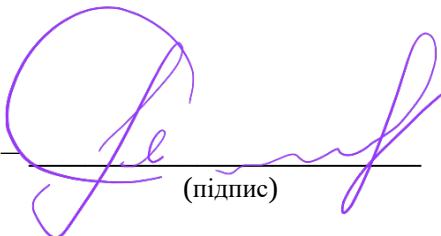
кваліфікаційна робота

здобувача вищої освіти першого (бакалаврського) рівня
освітньої програми «Інженерія програмного забезпечення»
за спеціальністю 121 Інженерія програмного забезпечення

Особистий підпис – 

Науковий керівник – 
(підпис)

доцент кафедри ІТС, кандидат
педагогічних наук, доцент
Смагіна О.О.
(посада, науковий ступінь, наукове звання,
ініціали, прізвище)

Зав. кафедри – 
(підпис)

зав. кафедри ІТС, кандидат
педагогічних наук, доцент,
М.А. Семенов
(посада, науковий ступінь, наукове звання,
ініціали, прізвище)

Лубни – 2026 року

Міністерство освіти і науки України

Державний заклад

„Луганський національний університет імені Тараса Шевченка”

Факультет (інститут)

Факультет технологій та інформаційних систем

Кафедра, циклова комісія

Інформаційних технологій та систем

Освітній ступень

Бакалавр

Напрямок підготовки (спеціальність)

121 Інженерія програмного забезпечення
(код, назва)

Галузь знань

12, Інформаційні технології
(код, назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри ІТС

(підпис)

М.А. Семенов
(ініціали, прізвище)

“ ” _____

2026 р.

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ РОБОТУ

Хомюку Івану Вікторовичу

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) Розробка веб-застосунку для обміну повідомленнями

Керівник кваліфікаційної роботи

Смагіна О.О., к.пед.н., доцент

(прізвище, ініціали, науковий ступінь, вчене звання)

затверджена наказом по університету

від

2. Строк подання студентом проекту (роботи)

3. Вихідні дані до роботи Клієнт-серверний додаток “Корпоративний чат”, створений на платформі Java.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) поняття, види та складові клієнт-серверних архітектур і протоколів, розробка програмного додатка на платформі програмування Java, структура та логіка клієнт-серверного додатку, етапи програмної реалізації.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень) Слайди презентації

6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання „_____” _____ р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1.	Вибір теми роботи, вивчення наукової літератури, затвердження теми та керівника.	До 24 жовтня	
2.	Аналіз літературних джерел за темою роботи. Розробка та апробація методики дослідно-експериментальної роботи. Подання структури теоретичної частини роботи та плану експериментальних досліджень.	До 1 лютого	
3.	Робота над теоретичною частиною. Подання теоретичної частини роботи для першого читання науковим керівником.	До 15 лютого	
4.	Усунення зауважень, урахування рекомендацій наукового керівника. Подання теоретичної частини роботи на друге читання.	До 1 квітня	
5.	Проведення експериментальної роботи. Поетапний аналіз та обговорення її результатів. Перевірка стану виконання роботи.	Перший тиждень квітня	
6.	Урахування рекомендацій наукового керівника, усунення недоліків, підготовка варіанта роботи до передзахисту. Розробка презентації.	До 31 квітня	
7.	Попередній захист роботи на кафедрі	травень	
8.	Доопрацювання роботи з урахуванням рекомендацій після передзахисту. Подання роботи науковому керівникові та рецензентові на підготовку відгуку та рецензії	За 10 днів до державної атестації	
9.	Подання на кафедру остаточного варіанта роботи, переплетеного та підписаного автором, науковим керівником і рецензентом.	За 5 днів до державної атестації	

Студент


підпис

І.В. Хомюк
(ініціали, прізвище)

Керівник проекту (роботи)


підпис

О.О. Смагіна
(ініціали, прізвище)

АНОТАЦІЯ

Хомюк Іван Вікторович

Тема: Розробка веб-застосунку для обміну повідомленнями

Спеціальність: 121 «Інженерія програмного забезпечення».

Установа: ДЗ Луганський національний університет імені Тараса Шевченка, 2026 р.

Кваліфікаційна робота: 60 с., 1 табл., 12 рис., 33 джерела.

Мета роботи – розробка веб-застосунку для обміну повідомленнями «Chatty», який забезпечує обмін текстовими повідомленнями між користувачами в режимі реального часу через мережу Internet.

Об'єкт дослідження – веб-застосунок для обміну повідомленнями «Chatty».

Предмет дослідження – методи та технології розробки веб-застосунків із використанням клієнт-серверної архітектури та сучасних веб-технологій.

Методи дослідження: теоретичні – аналіз наукової літератури та сучасних технологій веб-розробки, узагальнення та систематизація принципів створення веб-застосунків; емпіричні – порівняльний аналіз існуючих веб-месенджерів та інструментів розробки; експериментальні – тестування функціональності та працездатності розробленого веб-застосунку.

Результати роботи: розроблено веб-застосунок для обміну повідомленнями «Chatty», який реалізує функції реєстрації та авторизації користувачів, надсилання та отримання повідомлень у режимі реального часу, а також забезпечує зручний користувацький інтерфейс. Розроблений застосунок може використовуватися у навчальних цілях та як основа для створення повнофункціональних веб-месенджерів.

Висновки: узагальнено теоретичні підходи до розробки веб-застосунків, розглянуто принципи клієнт-серверної архітектури та технології обміну даними. Проаналізовано етапи створення веб-застосунку «Chatty» та реалізовано програмний продукт із використанням сучасних веб-технологій. Розроблений застосунок забезпечує обмін повідомленнями між користувачами та може бути розширений додатковими функціями.

Ключові слова: ВЕБ-ЗАСТОСУНОК, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, ЧАТ, ОБМІН ПОВІДОМЛЕННЯМИ, JAVASCRIPT, NODE.JS, WEBSOCKET, БАЗА ДАНИХ.

ABSTRACT

Khomiuk Ivan Viktorovich

Topic: Development of the web application for messaging "Chatty"

Specialty: 121 "Software Engineering".

Institution: DZ Luhansk Taras Shevchenko National University, 2026

Qualification work: 60 pages, 1 tables, 12 figures, 33 sources.

Purpose of the work: development of the web application for messaging "Chatty", which provides real-time text messaging between users via the Internet.

Object: web messaging application "Chatty".

Subject: methods and technologies for developing web applications using client-server architecture and modern web technologies.

Research methods: theoretical – analysis of scientific literature and modern web development technologies; empirical – comparative analysis of existing web messengers and development tools; experimental – testing of the developed web application.

Results: the web application "Chatty" was developed, which provides user registration and authentication, sending and receiving messages in real time, and a convenient user interface. The developed application can be used for educational purposes and as a basis for creating full-featured messaging systems.

Conclusions: theoretical approaches to web application development were generalized, principles of client-server architecture and data exchange technologies were considered. The stages of developing the web application "Chatty" were analyzed and implemented using modern web technologies.

Keywords: WEB APPLICATION, CLIENT-SERVER ARCHITECTURE, CHAT, MESSAGING, JAVASCRIPT, NODE.JS, WEBSOCKET, DATABASE.

Міністерство освіти і науки України
Державний заклад «Луганський національний університет
імені Тараса Шевченка»

Факультет (інститут)

Факультет технологій та інформаційних систем

(повна назва)

Кафедра

Кафедра інформаційних технологій та систем

(повна назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри ІТС

М.А. Семенов

(підпис)

(ініціали, прізвище)

“ ” 2026 р.

ТЕХНІЧНЕ ЗАВДАННЯ
на виконання програмної розробки (ПР):

Розробка веб-застосунку для обміну повідомленнями

ПОГОДЖЕНО

Керівник кваліфікаційної роботи

О.О. Смагіна

“ ” 2026 р

ВИКОНАВЕЦЬ

Студент групи 4ПЗ

Хомюк Іван Вікторович

“ ” 2026 р

Лубни – 2026 року

Зміст

Вступ.....	3
1. Підстави для розробки програми.....	3
2. Призначення розробки.....	3
3. Вимоги до програми	4
3.1. Вимоги до функціональних характеристик.....	4
3.2. Вимоги до надійності.....	4
3.3. Умови експлуатації	4
3.4. Вимоги до складу та параметрів технічних засобів	5
3.5. Вимоги до інформаційної і програмної сумісності	5
3.6. Вимоги до маркування та упаковки	5
3.7. Вимоги до транспортування і зберігання	6
3.8. Спеціальні вимоги (Технологічний стек)	6
4. Вимоги до програмної документації.....	7
5. Етапи виконання розробки.....	7
6. Порядок контролю і прийому	8
7. Порядок внесення змін до технічного завдання	8

Вступ

Найменування програми: Веб-застосунок для обміну повідомленнями «Chatty» (умовне позначення — веб-додаток «WebMessenger»).

Область застосування: Програма призначена для організації обміну текстовими повідомленнями в реальному часі між користувачами через персональні комп'ютери (десктопні версії веб-браузерів).

1. Підстави для розробки програми

Перелік документів, на підставі яких ведеться розробка:

- Наказ ректора ЛНУ імені Тараса Шевченка про затвердження тем випускних кваліфікаційних робіт бакалавра.
- Завдання на виконання випускної кваліфікаційної роботи.

1.2. Організація: Луганський національний університет імені Тараса Шевченка, кафедра інформаційних технологій та систем.

1.3. Терміни розробки:

- Початок: 01 лютого 2026 р.
- Закінчення: 01 травня 2026 р.

2. Призначення розробки

Функціональне призначення: Забезпечення обміну текстовими повідомленнями між користувачами в режимі реального часу через мережу Internet за допомогою веб-технологій. Програма повинна реалізовувати:

- реєстрацію нових користувачів;
- автентифікацію та авторизацію користувачів;
- відображення списку доступних користувачів (контактів);
- надсилання та отримання текстових повідомлень у режимі реального часу;
- збереження та перегляд історії повідомлень;
- графічний інтерфейс користувача для настільних ОС.

Експлуатаційне призначення: Програма призначена для експлуатації на персональних комп'ютерах та ноутбуках під керуванням сучасних операційних систем, що мають стабільне підключення до мережі Internet та встановлений сучасний веб-браузер.

3. Вимоги до програми

3.1. Вимоги до функціональних характеристик

Користувач системи повинен мати змогу виконувати такі дії:

- Реєстрація нового облікового запису (введення логіну, паролю, email).
- Авторизація в системі за раніше створеними обліковими даними.
- Перегляд списку зареєстрованих користувачів для вибору співрозмовника.
- Надсилання текстових повідомлень у діалогах.
- Миттєве отримання повідомлень без необхідності перезавантаження сторінки.
- Перегляд повної історії повідомлень у поточному чаті.
- Безпечний вихід із системи (деавторизація).
- Взаємодія з графічним інтерфейсом за допомогою миші / тачпада та клавіатури.

3.2. Вимоги до надійності

Стабільність роботи: Забезпечення безперервного функціонування клієнтської та серверної частин без критичних збоїв у межах задекларованого функціоналу.

Валідація даних: Обов'язкова двостороння перевірка (валідація) введених користувачем даних — як на стороні клієнта (для швидкого фідбеку в інтерфейсі), так і на стороні сервера (для безпеки БД).

Захист даних: Захист від несанкціонованого доступу до приватних профілів та діалогів інших користувачів за допомогою механізму токенів (JWT) або сесій.

Обробка помилок: Коректний перехоплення виняткових ситуацій (втрати мережі, введення некоректного пароля, помилки сервера) з виведенням зрозумілих повідомлень користувачеві.

Стійкість сервера: Здатність серверної частини підтримувати постійні та безперервні WebSocket-з'єднання з активними клієнтами.

3.3. Умови експлуатації

3.3.1. Кінцевий пристрій користувача повинен мати стабільний доступ до мережі Internet (широкопasmовий або мобільний модемний зв'язок).

3.3.2. Програма розрахована на роботу у десктопних версіях сучасних веб-браузерів:

- Google Chrome (версії 100+)
- Mozilla Firefox (версії 100+)
- Microsoft Edge (версії 100+)
- Opera (версії 90+)

3.4. Вимоги до складу та параметрів технічних засобів

Мінімальні системні вимоги до ПК користувача:

- **Процесор:** 1.5 GHz або вище (мінімум 2 ядра).
- **Оперативна пам'ять:** 2 GB або вище.
- **Вільне місце на диску:** до 100 MB (для кешування браузера).
- **Дисплей:** роздільна здатність **1366×768** або вище (оптимально **1920×1080** для десктопного інтерфейсу).
- **Периферія:** клавіатура, миша або тачпад.

3.5. Вимоги до інформаційної і програмної сумісності

Системне оточення користувача: Веб-застосунок має коректно функціонувати на персональних комп'ютерах під керуванням операційних систем Windows 10/11, Linux та macOS через сучасні веб-браузери з підтримкою стандартів HTML5, CSS3 та ECMAScript 6+.

Порядок розгортання та доступу: Програма є хмарним веб-застосунком (SaaS) і розгортається на віддаленому хостингу. Доступ користувачів здійснюється за допомогою унікального URL-посилання через мережу Internet. Фізичне маркування та пакування продукту не передбачені.

Зберігання та резервування: Початковий код проєкту розміщується та зберігається у системі контролю версій Git на платформі GitHub. Резервне копіювання бази даних та серверних логів реалізується на хмарній інфраструктурі хостингу.

3.6. Вимоги до маркування та упаковки

Архітектура: Клієнт-серверна.

Клієнтська частина (Frontend): HTML5, CSS3, JavaScript (компоновка інтерфейсу та логіка взаємодії виключно для десктопних версій браузерів).

Серверна частина (Backend): Платформа Node.js, веб-фреймворк Express.

База даних: MongoDB (документ-орієнтована СУБД класу NoSQL) або MySQL.

Мережеві протоколи: HTTP / HTTPS (для авторизації, завантаження статичних файлів та початкових даних), WebSocket / Socket.io (для миттєвого обміну повідомленнями в режимі реального часу).

3.7. Вимоги до транспортування і зберігання

Вихідний код програмного продукту зберігається у системі контролю версій Git з використанням платформи GitHub. Це забезпечує централізоване зберігання коду, можливість відстеження внесених змін, повернення до попередніх версій проєкту та організацію спільної роботи над програмним забезпеченням.

Для забезпечення надійності та збереження інформації передбачено резервне копіювання серверної частини застосунку та бази даних. Резервні копії зберігаються на хмарному сервері, що дозволяє мінімізувати ризик втрати даних у разі технічних збоїв, пошкодження обладнання або помилок під час оновлення програмного забезпечення. Регулярне створення резервних копій забезпечує можливість швидкого відновлення працездатності системи та збереження користувацьких даних.

3.8. Спеціальні вимоги (Технологічний стек)

Розробка програмного продукту здійснюється із використанням сучасних вебтехнологій, що забезпечують високу продуктивність, зручність підтримки та можливість подальшого масштабування системи.

Клієнтська частина (Frontend) реалізується за допомогою мов HTML5, CSS3 та JavaScript. HTML5 використовується для створення структури вебсторінок та інтерфейсу користувача. CSS3 забезпечує оформлення елементів інтерфейсу, адаптацію дизайну до різних розмірів екранів та покращення візуального сприйняття системи. JavaScript відповідає за реалізацію інтерактивних елементів, обробку подій користувача та взаємодію з серверною частиною застосунку.

Серверна частина (Backend) розробляється на платформі Node.js із використанням фреймворку Express. Таке рішення дозволяє створити високопродуктивний серверний застосунок, який забезпечує обробку запитів користувачів, авторизацію, керування даними та взаємодію з базою даних. Використання Express спрощує організацію маршрутизації та структурування серверного коду.

Для зберігання інформації використовується нереляційна база даних MongoDB. Вона забезпечує гнучке зберігання даних у форматі документів, високу швидкість роботи та зручність інтеграції із серверною частиною, розробленою на Node.js. У базі даних зберігаються відомості про користувачів, повідомлення, чати та інші дані, необхідні для функціонування системи.

Архітектура програмного продукту побудована за клієнт-серверним принципом, що передбачає розподіл функцій між клієнтською та серверною частинами. Такий підхід забезпечує ефективний обмін даними, спрощує супровід системи та дозволяє масштабувати її окремі компоненти.

Передача даних між клієнтом і сервером здійснюється за допомогою протоколів HTTP та HTTPS. Протокол HTTPS забезпечує захищене передавання інформації завдяки використанню механізмів шифрування. Для реалізації обміну повідомленнями в режимі реального часу використовується технологія WebSocket, яка забезпечує постійне двостороннє з'єднання між клієнтом і сервером. Це дозволяє миттєво передавати повідомлення між користувачами без необхідності постійного оновлення сторінки, що є важливою вимогою для сучасного вебмесенджера.

4. Вимоги до програмної документації

До складу обов'язкової документації проекту входять:

1. Технічне завдання (даний документ).
2. Пояснювальна записка до випускної бакалаврської роботи.
3. Інструкція користувача (керівництво з експлуатації веб-інтерфейсу).

5. Етапи виконання розробки

Таблиця 1

Етапи виконання робіт

№	Етапи виконання роботи	Термін виконання та обсяг робіт	звітні матеріали
1	Аналіз предметної області. Аналіз вимог. Проектування структури веб-застосунку	1 місяці	Документація, схема системи
2	Розробка клієнтської та серверної частини. Реалізація основних функцій	2 місяці	Робоча версія веб-застосунку
3	Тестування, виправлення помилок, оформлення документації	1 місяць	Готовий програмний продукт та документація

6. Порядок контролю і прийому

6.1. Представлення бакалаврської роботи до попереднього захисту

6.2. Представлення бакалаврської роботи до захисту

7. Порядок внесення змін до технічного завдання

Дане технічне завдання може уточнюватися в процесі розробки за погодженням керівника та студента.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ЗАКЛАД «ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА»

Факультет (інститут)

Факультет технологій та інформаційних систем
(повна назва)

Кафедра

Кафедра інформаційних технологій та систем
(повна назва)

Пояснювальна записка
до кваліфікаційної роботи
за першим (бакалаврським) рівнем освіти

Розробка веб-застосунку для обміну повідомленнями

Виконав: студент 4 курсу
напряму підготовки (спеціальності)
121 «Інженерія програмного
забезпечення»
(шифр і назва напряму підготовки, спеціальності)

Хомюк І.В.
(прізвище та ініціали)

Керівник Смагіна О.О.
(прізвище та ініціали)

Рецензент Козуб Ю.Г.
(прізвище та ініціали)

Лубни – 2026 року

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	4
РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ КЛІЄНТ-СЕРВЕРНИХ ВЕБ-ЗАСТОСУНКІВ ДЛЯ ОБМІНУ ПОВІДОМЛЕННЯМИ В РЕАЛЬНОМУ ЧАСІ	7
1.1. Клієнт-серверна архітектура	7
1.2. Моделі клієнт-серверних систем	9
1.3. Протоколи взаємодії клієнта і сервера в реальному часі	13
1.4. Особливості розробки веб-додатків реального часу на JavaScript та Node.js	16
Висновки до розділу 1	18
РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-ЗАСТОСУНКУ «СНАТТУ»	20
2.1. Аналіз вимог до веб-застосунку та архітектурне проектування	20
2.1.1. Функціональні вимоги до системи	20
2.1.2. Нефункціональні вимоги до системи	23
2.1.3. Специфікація архітектури системи та взаємодія компонентів	25
2.2. Проектування структури бази даних та моделей даних	27
2.2.1. Обґрунтування вибору NoSQL СУБД MongoDB	27
2.2.2. Розробка та програмна реалізація моделей даних (Mongoose Models)	28
2.2.3. Схема зв'язків між колекціями (Логічна структура БД)	30
2.3. Розробка серверної частини (Backend) на Node.js та Express	32
2.3.1. Налаштування оточення та конфігурація сервера	32
2.3.2. Система безпеки, автентифікації та захисту маршрутів	33
2.3.3. Реалізація архітектури REST API та ендпоінтів	35
2.3.4. Обробка файлових потоків та інтеграція Multer	35
2.3.5. Реалізація сервісу сповіщень на базі Nodemailer	37
2.4. Реалізація повнодуплексного обміну повідомленнями через Socket.io	39
2.4.1. Ініціалізація WebSocket-сервера та інтеграція з Express	39
2.4.2. Архітектура подій (Event-Driven Architecture)	40
2.4.3. Менеджмент кімнат (Rooms) та ізоляція приватних чатів	42

2.5. Розробка клієнтської частини (Frontend) та інтерфейсу користувача .	44
2.5.1. Модульна структура веб-додатку на стороні клієнта	45
2.5.2. Стилiзація, адаптивний дизайн та кастомізація інтерфейсу	46
2.5.3. Клієнтська WebSocket-логіка та динамічне оновлення DOM	49
ВИСНОВКИ ДО РОЗДІЛУ 2	51
ВИСНОВОК.....	54
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	56
ДОДАТОК А.....	59
ДОДАТОК Б	97
ДОДАТОК В	98
ДОДАТОК Г	146
ДОДАТОК Д.....	147
ДОДАТОК Е	148
ДОДАТОК Є	149
ДОДАТОК Ж	150
ДОДАТОК К	151
ДОДАТОК Р.....	152
ДОДАТОК Л	153
ДОДАТОК М	156
ДОДАТОК Н.....	160
ДОДАТОК З.....	161
ДОДАТОК П.....	163
ДОДАТОК Т	170
ДОДАТОК С	173
ДОДАТОК У	175
ДОДАТОК Х.....	177
ДОДАТОК Ц.....	180
ДОДАТОК Ч	183
ДОДАТОК Ю	186
ДОДАТОК Я	189

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

Atlas – MongoDB Atlas (хмарна платформа бази даних)

bcryptjs – бібліотека для хешування паролів

CORS – Cross-Origin Resource Sharing (механізм крос-доменних запитів)

dotenv – бібліотека для завантаження змінних середовища

Express – фреймворк веб-додатків для Node.js

HTML – HyperText Markup Language

HTTP – HyperText Transfer Protocol

HTTPS – HyperText Transfer Protocol Secure

JSON – JavaScript Object Notation

JWT – JSON Web Token

MongoDB – MongoDB (документоорієнтована NoSQL база даних)

Mongoose – ODM-бібліотека для роботи з MongoDB

Multer – middleware для обробки завантаження файлів

Nodemailer – бібліотека для відправлення електронних листів

WebSocket – протокол повнодуплексного обміну даними в реальному часі

Актуальність роботи. Сучасні інформаційні технології кардинально змінили спосіб спілкування людей. Щодня мільйони користувачів обмінюються текстовими повідомленнями в режимі реального часу. Миттєвий обмін інформацією став невід’ємною частиною як особистого, так і професійного життя. Однак більшість користувачів не замислюються, наскільки складними є технології, що забезпечують реальний час у веб-додатках.

Основним і найбільш ефективним підходом до реалізації таких систем залишається **клієнт-серверна архітектура**. На відміну від десктопних Java-додатків, сучасні веб-застосунки працюють безпосередньо в браузері, що накладає особливі вимоги до технологій обміну даними, масштабованості та безпеки.

Метою бакалаврської роботи є розробка веб-застосунку для обміну повідомленнями в реальному часі **Chatty**.

Об’єкт дослідження: клієнт-серверний веб-застосунок для обміну повідомленнями в реальному часі.

Предмет дослідження: технології та методи створення веб-додатків реального часу з використанням Node.js, WebSocket та сучасних веб-технологій.

Відповідно до мети та предмета дослідження були поставлені такі **основні завдання:**

- узагальнити теоретичні основи клієнт-серверної архітектури веб-додатків;
- розглянути технології забезпечення обміну повідомленнями в реальному часі (WebSocket, Server-Sent Events, Long Polling);
- проаналізувати особливості розробки серверної частини на Node.js та клієнтської частини з використанням JavaScript, HTML, CSS;
- дослідити питання автентифікації, безпеки та зберігання даних у чат-додатках;
- розробити та протестувати повнофункціональний веб-застосунок Chatty.

Для вирішення поставлених завдань використовувалися такі **методи дослідження**: теоретичні — аналіз наукової літератури та нормативно-технічної документації; емпіричні — порівняльний аналіз технологій реального часу; експериментальні — практична реалізація, тестування та відлагодження програмного продукту.

До складу роботи входять три розділи, які висвітлюють теоретичні основи, практичну реалізацію та тестування веб-застосунку для обміну повідомленнями.

Практичне значення розробки полягає в створенні функціонального веб-додатку реального часу, який може використовуватися як для навчальних цілей, так і як базовий прототип для подальшої розробки корпоративних або особистих месенджерів.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ КЛІЄНТ-СЕРВЕРНИХ ВЕБ-ЗАСТОСУНКІВ ДЛЯ ОБМІНУ ПОВІДОМЛЕННЯМИ В РЕАЛЬНОМУ ЧАСІ

1.1. Клієнт-серверна архітектура

Сучасний розвиток інформаційних технологій невід’ємно пов’язаний із розподіленими обчисленнями, серед яких особливе місце посідає клієнт-серверна архітектура. Вона є однією з найбільш поширених і ефективних моделей організації взаємодії в комп’ютерних мережах.

Клієнт-серверна архітектура — це модель розподіленої обробки даних, у якій системні ресурси і функції розділені між двома основними типами учасників: клієнтами та серверами. Клієнт виступає ініціатором взаємодії, надсилаючи запити на виконання певних дій або отримання інформації. Сервер, у свою чергу, обробляє ці запити, виконує необхідні обчислення, працює з базами даних і повертає клієнту результат.

Загальну концептуальну схему взаємодії в межах клієнт-серверної архітектури представлено на рисунку 1.1.

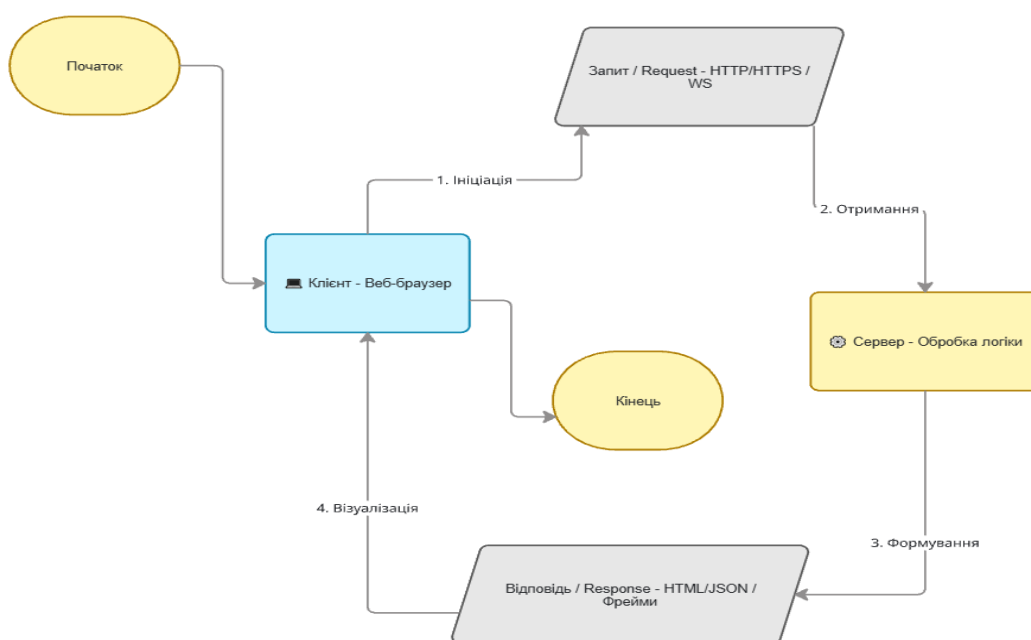


Рисунок 1.1 — Концептуальна схема взаємодії «Клієнт — Сервер»

Історично клієнт-серверна модель почала активно розвиватися в 1980-х роках із поширенням локальних мереж. Однак справжній прорив стався з розвитком глобальної мережі Internet, коли більшість сучасних сервісів (веб-сайти, месенджери, хмарні сервіси, онлайн-ігри) перейшли саме на цю архітектуру. На відміну від централізованих систем, де всі обчислення виконувалися на одному потужному комп'ютері, клієнт-серверний підхід дозволив значно підвищити ефективність використання ресурсів і забезпечити одночасну роботу тисяч і мільйонів користувачів.

Основними компонентами клієнт-серверної архітектури є:

- Клієнт — програмне забезпечення, що працює на пристрої кінцевого користувача (веб-браузер, мобільний додаток тощо). Його основні функції включають формування запитів, відображення інформації у зручному для користувача вигляді та обробку локальних подій. У веб-додатках роль клієнта виконує браузер, який інтерпретує HTML, CSS та JavaScript.
- Сервер — потужний комп'ютер (або група серверів), який постійно знаходиться в режимі очікування запитів. Він виконує бізнес-логіку, автентифікацію користувачів, роботу з базою даних, обробку повідомлень і забезпечує безпеку системи. Сервер може обслуговувати одночасно велику кількість клієнтів.
- Мережа — канал зв'язку між клієнтом і сервером. У сучасних системах найчастіше використовуються протоколи TCP/IP та HTTP/HTTPS. Якість і швидкість мережі безпосередньо впливають на продуктивність усього додатку.

Важливою перевагою клієнт-серверної архітектури є централізоване зберігання даних. Вся інформація (профілі користувачів, історія повідомлень, налаштування) зберігається на сервері, що забезпечує доступ до даних з будь-якого пристрою після авторизації. Крім того, така модель значно спрощує оновлення програмного забезпечення: достатньо внести зміни на сервері, і всі клієнти автоматично отримують оновлену версію.

У контексті веб-застосунків для обміну повідомленнями клієнт-серверна архітектура набуває особливого значення. Чат-додатки потребують швидкої обробки великої кількості одночасних з'єднань, надійної доставки повідомлень у реальному часі та захисту персональних даних. Саме тому більшість популярних месенджерів (наприклад, WhatsApp Web, Telegram Web, Slack) побудовані за клієнт-серверною моделлю.

Разом з тим, клієнт-серверна архітектура має певні недоліки. Основним з них є залежність від стабільності роботи сервера: у разі його відмови весь сервіс стає недоступним. Також при великій кількості користувачів виникає необхідність у масштабуванні системи (використання балансувальників навантаження, кількох серверів, хмарних рішень тощо).

У розробленому веб-застосунку Chatty реалізовано трирівневу клієнт-серверну архітектуру, де:

- клієнтський рівень представлений веб-інтерфейсом (HTML, CSS, JavaScript);
- серверний рівень — Node.js з використанням Express та WebSocket;
- рівень даних — хмарна база даних MongoDB Atlas.

Така організація дозволяє ефективно розподіляти навантаження, забезпечувати високу швидкість обміну повідомленнями та підтримувати надійність системи.

1.2. Моделі клієнт-серверних систем

Існує кілька основних моделей організації клієнт-серверної взаємодії, які відрізняються способом розподілу логіки, даних і презентаційного шару. Вибір тієї чи іншої моделі суттєво впливає на масштабованість, безпеку, продуктивність і складність підтримки додатку.

Peer-to-Peer (P2P) архітектура (однорангова) — це децентралізована модель, у якій кожен учасник мережі може одночасно виступати як клієнтом, так і сервером. У такій системі відсутній центральний сервер, а вузли мережі обмінюються даними безпосередньо між собою.

Перевагами P2P-архітектури є висока відмовостійкість (відключення одного вузла не впливає на роботу мережі) та відсутність єдиної точки відмови. Однак у контексті веб-додатків для обміну повідомленнями дана модель має суттєві недоліки: складність забезпечення безпеки, проблеми з NAT-транзитом, складне управління користувачами та історією повідомлень. Тому P2P-архітектура рідко використовується в сучасних чат-додатках, за винятком окремих спеціалізованих рішень (наприклад, деяких децентралізованих месенджерів).

Дворівнева архітектура (Two-tier Client-Server) є найпростішою і найпоширенішою формою клієнт-серверної моделі. У ній вся логіка додатку та робота з базою даних зосереджені на сервері, а клієнт відповідає лише за презентаційний шар (інтерфейс) та формування запитів.

Така модель добре підходить для невеликих додатків з обмеженою кількістю користувачів. Головними перевагами є простота реалізації та швидкість розробки. Водночас дворівнева архітектура має серйозні обмеження при зростанні навантаження: сервер одночасно виконує і бізнес-логіку, і роботу з даними, що призводить до швидкого зростання навантаження на нього. Крім того, при зміні бізнес-логіки часто потрібно оновлювати як серверну, так і клієнтську частини.

Трирівнева (Three-tier) та багаторівнева (N-tier) архітектура на сьогодні є найбільш сучасною та рекомендованою моделлю для розробки веб-застосунків, зокрема чатів. Вона передбачає чітке розділення системи на три основні шари:

1. **Клієнтський шар (Presentation tier)** — веб-браузер, який відповідає за інтерфейс користувача, відображення повідомлень, обробку подій та візуалізацію.
2. **Серверний шар (Application / Business Logic tier)** — середній рівень, на якому виконується вся бізнес-логіка: автентифікація, обробка повідомлень, робота з WebSocket, валідація даних тощо. У

розробленому додатку цей шар реалізований на Node.js з використанням Express.

3. **Шар даних (Data tier)** — база даних, у даному випадку хмарне рішення MongoDB Atlas.

Структурну схему трирівневої клієнт-серверної архітектури розробленого веб-застосунку «Chatty» наведено на рисунку 1.1.

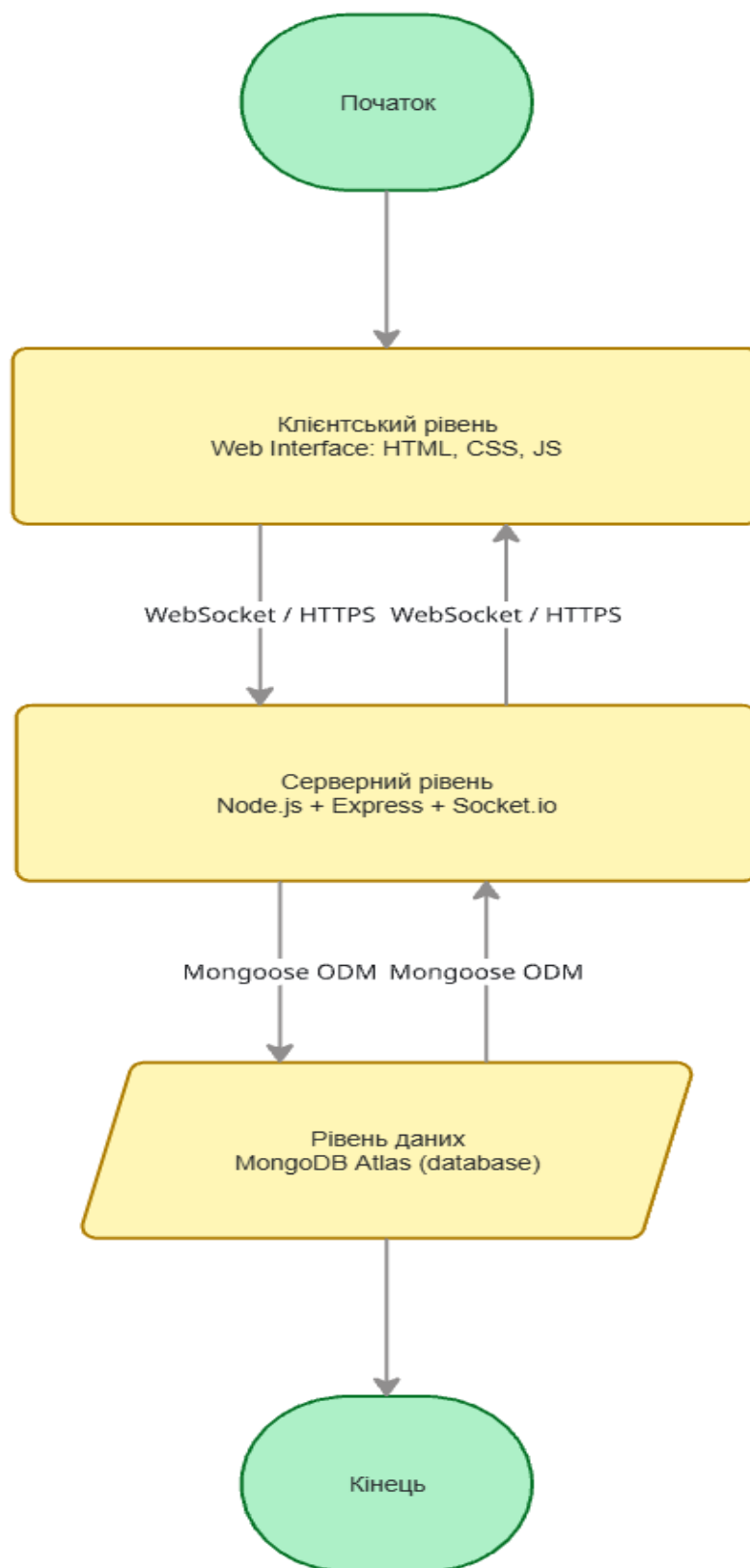


Рисунок 1.1 — Трирівнева архітектура веб-застосунку «Chatty»

Таке розділення дозволяє незалежно масштабувати кожен шар, спрощує підтримку та розвиток системи. Наприклад, при збільшенні кількості користувачів можна додавати нові серверні інстанси без зміни бази даних.

Крім того, трирівнева архітектура значно підвищує рівень безпеки, оскільки клієнт не має прямого доступу до бази даних.

У сучасних веб-додатках часто використовують ще більш гнучку **багаторівневу (N-tier)** архітектуру, де між основними шарами можуть додаватися додаткові (наприклад, шар кешування Redis, шар мікросервісів, шар автентифікації тощо).

Порівняльний аналіз моделей показує, що трирівнева архітектура найкраще відповідає вимогам сучасних веб-застосунків для обміну повідомленнями в реальному часі, оскільки забезпечує баланс між продуктивністю, масштабованістю, безпекою та зручністю підтримки.

У бакалаврській роботі для веб-застосунку **Chatty** обрано **трирівневу клієнт-серверну архітектуру**. Таке рішення дозволяє ефективно реалізовувати обмін повідомленнями через WebSocket, забезпечувати надійне зберігання даних у MongoDB Atlas та створювати сучасний, зручний інтерфейс користувача.

1.3. Протоколи взаємодії клієнта і сервера в реальному часі

Для забезпечення обміну повідомленнями в реальному часі стандартного HTTP-протоколу, який працює за принципом «запит-відповідь», часто недостатньо. Класичний HTTP є одностороннім і stateless-протоколом, що змушує клієнта постійно ініціювати нові запити для отримання оновлень. Це призводить до високого мережного навантаження та затримок у доставці повідомлень.

Для вирішення проблеми реального часу було розроблено декілька технологій і підходів, які відрізняються за ефективністю, складністю реалізації та використанням ресурсів.

Short Polling — найпростіший метод, при якому клієнт через певні проміжки часу (наприклад, кожні 2–5 секунд) надсилає запит на сервер з метою перевірки нових повідомлень. Незважаючи на простоту реалізації, цей підхід має низьку ефективність, оскільки більшість запитів повертає

порожню відповідь, створюючи значне навантаження як на сервер, так і на мережу.

Long Polling — удосконалена версія поліngu. Клієнт надсилає запит, а сервер тримає з'єднання відкритим доти, доки не з'явиться нове повідомлення або не мине таймаут. Після отримання відповіді клієнт одразу надсилає новий запит. Такий підхід зменшує кількість «пустих» відповідей, але все одно створює велику кількість одночасних з'єднань на сервері та не дозволяє повноцінний двосторонній обмін даними.

Server-Sent Events (SSE) — технологія, яка дозволяє серверу надсилати дані клієнту в односторонньому порядку через постійне HTTP-з'єднання. SSE добре підходить для ситуацій, коли не потрібна двостороння комунікація (наприклад, для стрічок новин або сповіщень). Однак для повноцінного чату, де користувачі активно надсилають і отримують повідомлення, SSE має обмеження.

WebSocket — сучасний і найбільш ефективний протокол для обміну даними в реальному часі. WebSocket встановлює постійне повнодуплексне (двостороннє) з'єднання між клієнтом і сервером поверх TCP-протоколу. Після початкового рукоштовування (handshake), яке виконується через звичайний HTTP-запит, з'єднання залишається відкритим, а дані передаються з мінімальними накладними витратами у обох напрямках у режимі реального часу.

Основні переваги WebSocket порівняно з іншими технологіями:

- Низька затримка доставки повідомлень;
- Значно менші витрати на мережевий трафік;
- Підтримка двостороннього зв'язку;
- Висока масштабованість при великій кількості одночасних користувачів;
- Вбудована підтримка в усіх сучасних веб-браузерах.

У порівнянні з традиційним HTTP, WebSocket економить серверні ресурси та дозволяє обробляти тисячі активних з'єднань на одному сервері. Алгоритм

та послідовність взаємодії між клієнтською та серверною частинами за цим протоколом наочно зображено на рисунку 1.2.

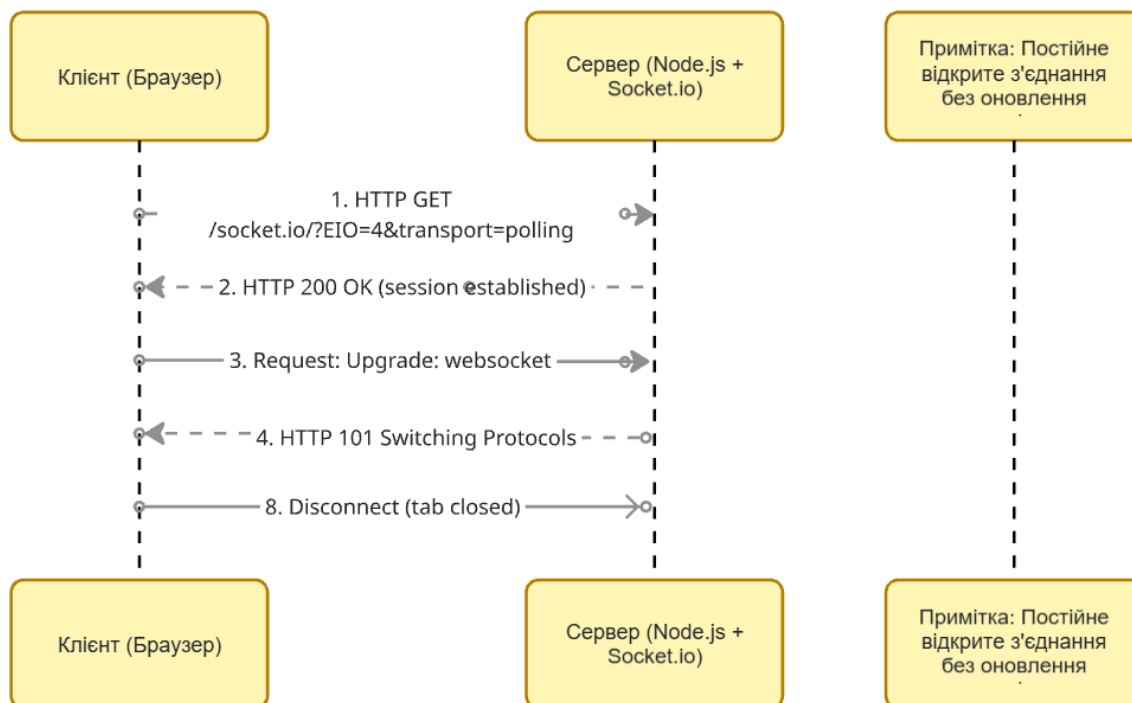


Рисунок 1.2 — Схема взаємодії клієнта та сервера за протоколом WebSocket
Клієнт виступає ініціатором

У розробленому веб-застосунку **Chatty** основним протоколом взаємодії обрано **WebSocket**. Для його реалізації на серверній стороні використовується бібліотека Socket.io поверх Express.js, яка забезпечує надійне з'єднання, автоматичне відновлення при розривах, підтримку fallback-механізмів та зручну роботу з подіями (emit / on). Клієнтська частина взаємодіє з сервером через WebSocket API браузера.

Додатково в роботі використовується протокол **HTTPS** для забезпечення безпеки початкового рукостискання та передачі конфіденційної інформації (авторизація, реєстрація). Для роботи з базою даних застосовується драйвер MongoDB та ODM-бібліотека Mongoose.

Вибір WebSocket як основного протоколу дозволяє повністю задовольнити вимоги технічного завдання щодо отримання та надсилання повідомлень у режимі реального часу, забезпечуючи високу швидкість та комфортну взаємодію користувачів.

1.4. Особливості розробки веб-додатків реального часу на JavaScript та Node.js

JavaScript, який традиційно використовувався лише для клієнтської частини веб-додатків, завдяки середовищу виконання **Node.js** став повноцінною серверною технологією. Це дозволило розробникам використовувати одну мову програмування на всьому стеку, що значно спрощує розробку, підтримку та масштабованість сучасних веб-застосунків.

Node.js — це асинхронне, подієво-орієнтоване середовище виконання JavaScript, побудоване на рушії V8. Його основною перевагою є неблокуюча модель введення-виведення (non-blocking I/O), яка ідеально підходить для додатків з високим рівнем паралелізму, таких як чати, де одночасно може бути велика кількість активних з'єднань. На відміну від традиційних серверних мов (PHP, Python), Node.js ефективно обробляє тисячі одночасних WebSocket-з'єднань на одному потоці.

Для побудови серверної частини в роботі використовується фреймворк **Express**, який забезпечує зручну маршрутизацію, middleware-архітектуру та швидку обробку HTTP-запитів. Це дозволяє чітко розділити RESTful ендпоінти (реєстрація, авторизація, завантаження файлів) від WebSocket-логіки.

Однією з ключових технологій забезпечення реального часу є бібліотека **Socket.io**. Вона працює поверх WebSocket і надає надійний двосторонній канал зв'язку, автоматичне відновлення з'єднання при розривах, підтримку fallback-механізмів (Long Polling) та зручний API для роботи з подіями (emit, on, broadcast).

Для роботи з даними обрано **MongoDB Atlas** — хмарну документоорієнтовану NoSQL базу даних. Завдяки бібліотеці **Mongoose** забезпечується об'єктно-документне відображення (ODM), валідація схем, middleware та зручна взаємодія з колекціями користувачів і повідомлень. Такий вибір дозволяє гнучко зберігати історію чатів, підтримувати різні типи повідомлень та швидко масштабувати систему.

Важливими аспектами безпеки є використання:

- **JWT (JSON Web Token)** — для stateless-автентифікації користувачів;
- **bcryptjs** — для надійного хешування паролів;
- **CORS** — для керування крос-доменними запитами.

Додатково в проєкті застосовуються бібліотеки **dotenv** (управління змінними середовища), **Multer** (обробка завантаження файлів) та **Nodemailer** (відправка електронних листів, наприклад, для підтвердження реєстрації).

Переваги єдиного JavaScript-стеку (Fullstack JavaScript):

- Єдина мова програмування на клієнті та сервері;
- Швидка розробка та легке перенесення коду;
- Велика екосистема готових пакетів (npm);
- Висока продуктивність при обробці реального часу;
- Простота найму розробників та підтримки проєкту.

Серед особливостей та викликів розробки на Node.js варто відзначити необхідність правильної організації асинхронного коду, обробки помилок, масштабування (кластеризація, використання PM2 або Docker) та забезпечення безпеки (захист від XSS, rate limiting, валідація даних).

У розробленому веб-застосунку **Chatty** поєднання Node.js, Express, Socket.io та MongoDB Atlas дозволило створити сучасний, швидкий і масштабований чат-додаток, який повністю відповідає вимогам технічного завдання щодо функціонування в реальному часі.

Висновки до розділу 1

У першому розділі бакалаврської роботи були розглянуті теоретичні основи розробки клієнт-серверних веб-застосунків для обміну повідомленнями в реальному часі. Проведений аналіз показав, що клієнт-серверна архітектура є фундаментальною моделлю сучасних мережових додатків і найбільш ефективним підходом для реалізації чат-систем.

Були детально вивчені основні моделі клієнт-серверних систем: Peer-to-Peer, дворівнева та трирівнева (багаторівнева) архітектури. Було обґрунтовано, що для сучасного веб-додатку реального часу найбільш доцільною є **трирівнева архітектура**, яка забезпечує чітке розділення відповідальності між клієнтським шаром, серверним шаром бізнес-логіки та шаром даних, сприяє кращій масштабованості, безпеці та зручності підтримки системи.

Особлива увага приділялася протоколам забезпечення обміну даними в реальному часі. Проведено порівняльний аналіз Short Polling, Long Polling, Server-Sent Events та WebSocket. Зроблено висновок, що протокол **WebSocket** є оптимальним рішенням для чат-додатків завдяки повнодуплексному зв'язку, мінімальним затримкам і високій ефективності використання мережових ресурсів.

Також були розглянуті особливості розробки веб-додатків реального часу на JavaScript та Node.js. Встановлено, що використання єдиного JavaScript-стеку дозволяє значно прискорити розробку, спростити підтримку коду та підвищити продуктивність системи. Проаналізовано ключові технології та бібліотеки, що використовуються в роботі: Express.js, Socket.io, MongoDB Atlas, Mongoose, JWT, bcryptjs та інші. Показано, що обраний технологічний стек повністю відповідає сучасним вимогам до веб-додатків такого типу.

Таким чином, теоретичний аналіз, проведений у першому розділі, дозволив обґрунтувати вибір архітектури, протоколів та інструментів розробки для створення веб-застосунку **Chatty**. Вибрані рішення

забезпечують високу швидкість обміну повідомленнями, надійність роботи, безпеку даних та можливість подальшого масштабування системи.

Результати теоретичного дослідження стали основою для практичної реалізації веб-додатку в наступних розділах роботи.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-ЗАСТОСУНКУ «CHATTY»

2.1. Аналіз вимог до веб-застосунку та архітектурне проектування

Етап аналізу вимог та проектування архітектури є фундаментальним кроком у життєвому циклі розробки програмного забезпечення. Особливо це стосується систем обміну повідомленнями у реальному часі (Real-Time Messengers), де правильний розподіл навантаження та чітке визначення меж функціональності безпосередньо впливають на стабільність, безпеку та швидкість доставки інформації кінцевому користувачу.

У межах розробки веб-застосунку «Chatty» було проведено детальний збір, аналіз та класифікацію технічних вимог до системи. Усі вимоги було розподілено на дві основні групи: функціональні (які описують безпосередні можливості користувача та логіку поведінки системи) та нефункціональні (що визначають якісні характеристики, обмеження, продуктивність та системні властивості продукту).

2.1.1. Функціональні вимоги до системи

Функціональні вимоги визначають сервіси та функції, які програмний продукт має надавати користувачам для вирішення їхніх повсякденних завдань. Для веб-застосунку «Chatty» було сформовано та реалізовано такий перелік ключових функціональних вимог:

1. Управління обліковими записами та профілями користувачів:

- *Реєстрація користувачів:* створення нового облікового запису в системі з обов'язковою валідацією полів (унікальний email, ім'я користувача, надійний пароль) та інтеграцією сервісу верифікації через надсилання системних повідомлень.
- *Автентифікація та авторизація:* забезпечення безпечного входу до системи за допомогою сучасних криптографічних

протоколів, збереження сесії користувача та розмежування прав доступу до приватних маршрутів (маршрутизація API).

- *Редагування профілю*: можливість завантаження та зміни персонального аватара користувача, оновлення текстових даних та налаштування мовної локалізації.

2. Обмін повідомленнями в режимі реального часу (Real-Time Messaging):

- *Миттєве надсилання та отримання повідомлень*: користувач повинен мати можливість відправляти текстові повідомлення у приватні чати з мінімальною затримкою, при цьому отримувач має бачити повідомлення миттєво без потреби ручного оновлення веб-сторінки.
- *Індикація активності (Typing Indicator)*: відображення статусу введення тексту іншим користувачем у поточному діалозі для підвищення інтерактивності інтерфейсу.
- *Історія повідомлень*: автоматичне збереження текстових діалогів та медіафайлів у базі даних із можливістю їх підвантаження під час відкриття конкретного чату.

3. Робота з мультимедійними даними та файловими потоками:

- *Завантаження файлів*: підтримка надсилання зображень та інших файлових вкладень безпосередньо у вікно чату.
- *Попередній перегляд (File Preview)*: генерація та візуалізація прев'ю-зображень для завантажених медіафайлів безпосередньо перед надсиланням або всередині стрічки повідомлень для покращення користувацького досвіду (UX).

4. Кастомізація інтерфейсу та локалізація:

- *Керування темами оформлення:* підтримка динамічного перемикання між світлою (Light Mode) та темною (Dark Mode) темами інтерфейсу із збереженням вибору користувача у локальному сховищі браузера.
- *Багатомовність (Localization):* підтримка інтерфейсу як українською (uk), англійською (en), німецькою (de), так і іспанською (es) мовами для гнучкого використання залежно від уподобань користувача.

5. Обмін повідомленнями в режимі реального часу (Real-Time Messaging):

- *Миттєве надсилання та отримання повідомлень:* користувач повинен мати можливість відправляти текстові повідомлення у приватні чати з мінімальною затримкою, при цьому отримувач має бачити повідомлення миттєво без потреби ручного оновлення веб-сторінки.
- *Індикація активності (Typing Indicator):* відображення статусу введення тексту іншим користувачем у поточному діалозі для підвищення інтерактивності інтерфейсу.
- *Історія повідомлень:* автоматичне збереження текстових діалогів та медіафайлів у базі даних із можливістю їх підвантаження під час відкриття конкретного чату.

6. Робота з мультимедійними даними та файловими потоками:

- *Завантаження файлів:* підтримка надсилання зображень та інших файлових вкладень безпосередньо у вікно чату.
- *Попередній перегляд (File Preview):* генерація та візуалізація прев'ю-зображень для завантажених медіафайлів

безпосередньо перед надсиланням або всередині стрічки повідомлень для покращення користувацького досвіду (UX).

7. Кастомізація інтерфейсу та локалізація:

- *Керування темами оформлення:* підтримка динамічного перемикання між світлою (Light Mode) та темною (Dark Mode) темами інтерфейсу із збереженням вибору користувача у локальному сховищі браузера.
- *Багатомовність (Localization):* підтримка інтерфейсу як українською (uk), так і англійською (en) мовами для гнучкого використання залежно від уподобань користувача.

Концептуальну модель взаємодії користувача з функціональними можливостями месенджера у вигляді UML-діаграми прецедентів представлено на рисунку 2.1

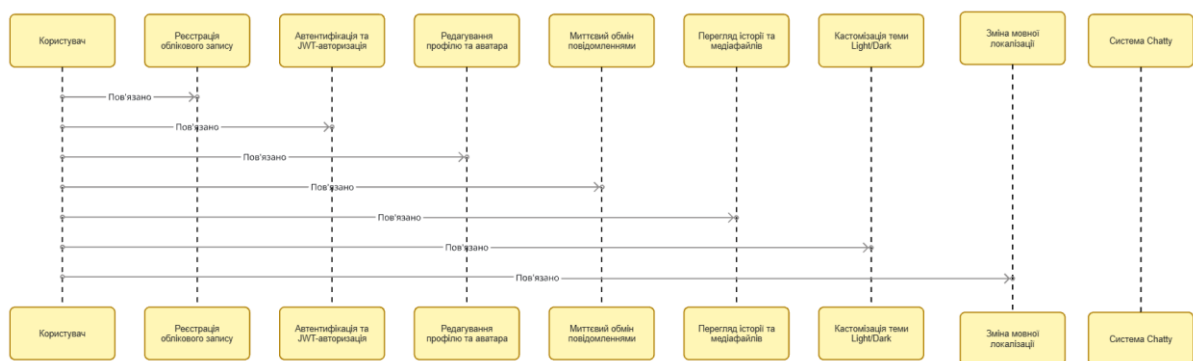


Рисунок 2.1 — UML-діаграма прецедентів використання веб-застосунку «Chatty»

2.1.2. Нефункціональні вимоги до системи

Нефункціональні вимоги визначають критерії якості, атрибути системи та технічні обмеження, яким має відповідати розроблений веб-застосунок для

забезпечення стабільного функціонування під навантаженням. До веб-застосунку «Chatty» висунуто такі нефункціональні вимоги:

1. Продуктивність та швидкість роботи (Performance):

- Максимальна затримка (Latency) при доставці текстового повідомлення між активними клієнтами в мережі через протокол WebSocket не повинна перевищувати 200–300 мс при стабільному інтернет-з'єднанні.
- Час відгуку серверної частини на стандартні HTTP-запити (REST API) для авторизації чи реєстрації не повинен перевищувати 500 мс.

2. Безпека та захист даних (Security):

- *Шифрування паролів:* забороняється збереження паролів користувачів у базі даних у відкритому текстовому вигляді. Усі паролі мають проходити обов'язкове одностороннє хешування за допомогою алгоритму bcrypt.
- *Безпека сесій:* захист клієнт-серверної взаємодії за допомогою механізму Stateless-автентифікації на базі JSON Web Tokens (JWT).
- *Контроль доступу:* налаштування політики Cross-Origin Resource Sharing (CORS) для обмеження несанкціонованих запитів до серверної частини з боку сторонніх доменів.
- *Захист каналів зв'язку:* використання захищених протоколів HTTPS та WSS (WebSocket Secure) для передачі конфіденційних даних та захисту від атак типу «Man-in-the-Middle» (MitM).

3. Сумісність та адаптивність інтерфейсу (Usability & Responsiveness):

- Клієнтська частина додатка повинна мати 100% адаптивний дизайн (Responsive Web Design), що забезпечує коректне відображення всіх елементів керування, списку чатів та вікна повідомлень як на десктопних моніторах, так і на мобільних пристроях (смартфонах, планшетах) з різною роздільною здатністю екрана.
- Кроссбраузерність: стабільна робота інтерфейсу в усіх сучасних популярних браузерах (Google Chrome, Mozilla Firefox, Apple Safari, Microsoft Edge).

2.1.3. Специфікація архітектури системи та взаємодія компонентів

На основі детального аналізу вимог було розроблено специфікацію архітектури програмного продукту. Для забезпечення високого рівня масштабованості, гнучкості та безпеки було обрано трирівневу архітектурну модель (Three-tier Architecture). Дана модель передбачає повне розділення системи на три незалежні рівні (шари), кожен з яких виконує строго визначений набір функцій і взаємодіє з іншими рівнями через стандартизовані інтерфейси розробки (рис. 2.1).

1. Рівень представлення (Presentation Tier / Frontend): Цей рівень функціонує безпосередньо на пристрої кінцевого користувача всередині веб-браузера. Він побудований на базі стандартного технологічного стеку: HTML5 для структурування контенту (**Додаток Я**), CSS3 для стилізації, адаптивної верстки та забезпечення темної/світлої тем (**Додаток Б**), а також JavaScript для реалізації динамічної логіки інтерфейсу (**Додаток В**). Головні завдання клієнтського рівня — візуалізація даних, отриманих від сервера, обробка дій користувача (кліків, введення тексту) та підтримання постійного зв'язку через WebSocket-клієнт для миттєвого відображення нових повідомлень без перезавантаження сторінки.

2. Рівень бізнес-логіки (Application / Business Logic Tier / Backend): Центральний компонент системи, реалізований як веб-сервер на базі

середовища виконання Node.js з використанням мікрофреймворку Express. Цей шар відповідає за обробку всієї логіки додатка: валідацію вхідних даних, генерацію та перевірку токенів автентифікації (JWT), хешування паролів, маршрутизацію HTTP-запитів та оркестрацію файлових потоків за допомогою middleware-модулів (Multer). Окремим критично важливим ядром цього шару є WebSocket-сервер на базі бібліотеки Socket.io, який управляє пулом активних мережеских з'єднань, обробляє події надсилання повідомлень та здійснює їх ретрансляцію відповідним клієнтам у реальному часі.

3. Рівень даних (Data Tier / Database): Нижній шар архітектури, представлений хмарною NoSQL базою даних MongoDB Atlas. Даний рівень відповідає за надійне, довготривале та структуроване зберігання інформації. Взаємодія між сервером бізнес-логіки (Node.js) та рівнем даних реалізована за допомогою ODM-бібліотеки (Object-Document Mapping) Mongoose. Mongoose дозволяє описати чіткі схеми документів, виконувати швидку типізацію та валідацію даних перед їх записом у колекції, а також оптимізувати пошукові запити для швидкого вилучення історії листування. Повна програмна конфігурація та вихідний код головного файлу ініціалізації сервера міститься у **(Додатку Г)**.

Взаємодія компонентів архітектури відбувається за строгою схемою: клієнт ніколи не взаємодіє з базою даних безпосередньо, що виключає можливість прямих ін'єкцій чи компрометації даних. Усі запити проходять через захищений проміжний серверний шар, де виконується автентифікація, перевірка прав доступу та логування подій. Такий інженерний підхід дозволяє незалежно модифікувати та масштабувати кожен рівень системи (наприклад, перенести базу даних на інший кластер або додати нові серверні потужності) без необхідності переписування коду клієнтської частини месенджера «Chatty». Налаштування підключення, конфігурацію пулу з'єднань та ініціалізацію взаємодії із СКБД на рівні серверної платформи наведено у **(Додатку Г)**.

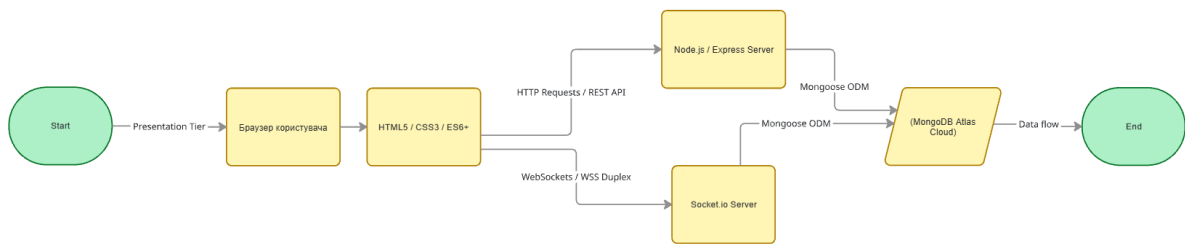


Рисунок 2.2 — Структурна схема тривірневої архітектури веб-застосунку «Chatty»

2.2. Проектування структури бази даних та моделей даних

Сучасні системи обміну повідомленнями оперують величезними обсягами неструктурованих або напівструктурованих даних, які постійно оновлюються в реальному часі. Успішне функціонування чат-додатка залежить від швидкості запису нових повідомлень та ефективності вибірки історії листування. Тому етап проектування шару даних вимагає ретельного підбору архітектури системи управління базами даних (СУБД) та детального моделювання сутностей.

2.2.1. Обґрунтування вибору NoSQL СУБД MongoDB

Під час проектування веб-застосунку «Chatty» буловедено порівняльний аналіз між класичними реляційними СУБД (наприклад, PostgreSQL, MySQL) та нереляційними системами класу NoSQL. Для реалізації ядра зберігання даних було обрано документоорієнтовану NoSQL базу даних MongoDB (у вигляді хмарного рішення MongoDB Atlas), керуючись такими інженерними критеріями:

1. Гнучкість схем даних (Schemaless): На відміну від реляційних баз, де кожна зміна структури таблиці потребує виконання складних міграцій (ALTER TABLE) та може призвести до простою системи, MongoDB зберігає дані у гнучкому форматі BSON (бінарний JSON). Це дозволяє легко модифікувати структуру документів на етапі розширення

функціоналу. Наприклад, якщо в майбутньому до повідомлення знадобиться додати нові поля (статус прочитання, реакції-емодзі тощо), це можна зробити миттєво без перебудови всієї бази даних.

2. Висока швидкість операцій запису (Write Performance): У чат-додатках операції запису відбуваються безперервно — щоразу, **коли** будь-який користувач надсилає повідомлення, система фіксує його в базі. MongoDB оптимізована для швидкого вставлення (Insert) документів за рахунок архітектури пам'яті та відсутності необхідності перевірки складних каскадних зв'язків (Foreign Keys) та транзакцій ACID на рівні таблиць, що забезпечує мінімальну затримку (latency) під навантаженням.
3. Ефективне збереження вкладених структур та масивів: Повідомлення в месенджерах часто містять динамічні масиви даних (наприклад, список посилань на завантажені медіафайли чи зображення-прев'ю). У реляційних БД для цього довелося б створювати окрему проміжну таблицю та використовувати операції JOIN, які уповільнюють роботу системи. В MongoDB такі дані зберігаються як звичайний масив всередині одного документа, що дозволяє отримати всю інформацію про повідомлення за один швидкий запит.
4. Хмарна екосистема MongoDB Atlas: Використання керованого хмарного рішення дозволяє делегувати завдання адміністрування, резервного копіювання (backups) та горизонтального масштабування (sharding) хмарній інфраструктурі, гарантуючи доступність бази даних 24/7.

2.2.2. Розробка та програмна реалізація моделей даних (Mongoose Models)

Для інтеграції СУБД MongoDB із серверною частиною на платформі Node.js було використано ODM-бібліотеку (Object-Document Mapping) Mongoose. Вона дозволяє накладати строгу валідацію, типізацію та структурування на

документи MongoDB безпосередньо на рівні прикладного коду сервера. В межах архітектури веб-застосунку «Chatty» було спроектовано три ключові моделі даних: User (Користувачі), Message (Повідомлення) та Chat (Чати / Кімнати).

Опис та лістинг моделі користувача (User)

Модель User описує логічну структуру та параметри облікового запису користувача в системі. Вона містить обов'язкові індивідуальні поля для автентифікації та ідентифікації в мережі (nickname, email, tag), персональну інформацію профілю (displayName, phone, bio, avatar), а також рекурсивний масив contacts для організації списку зв'язків між користувачами в межах однієї колекції.

Програмну реалізацію структури схеми та експорту моделі даних користувача, включаючи проміжний метод автоматичного асинхронного хешування паролів, наведено у **Додатку К**.

Опис та лістинг моделі повідомлення (Message)

Модель Message відповідає за довготривале збереження історії листування. Зв'язок між документом повідомлення та користувачем-відправником реалізовано за допомогою механізму референсів (mongoose.Schema.Types.ObjectId), що вказують на конкретні ідентифікатори в колекції користувачів. Структура також підтримує вкладений об'єкт file для збереження метаданих та URL-адреси завантажених медіафайлів.

Структура також підтримує вкладений об'єкт file для збереження метаданих та URL-адреси завантажених медіафайлів. Програмну реалізацію структури документа повідомлення з автоматичною фіксацією міток часу (timestamps) наведено у **Додатку З**.

Опис та лістинг моделі чату (Chat)

Модель Chat є центральною сполучною ланкою в архітектурі бази даних месенджера «Chatty». Вона призначена для створення, ідентифікації та адміністрування просторів для спілкування. Завдяки гнучкій структурі NoSQL, ця модель універсально обслуговує як приватні діалоги між двома користувачами (Direct Messages), так і групові чати (Group Chats) без необхідності створення додаткових таблиць чи колекцій.

Окремою важливою архітектурною особливістю моделі є збереження вкладеного об'єкта `lastMessage`. Замість того, щоб під час кожного рендерингу списку чатів виконувати важкий пошук останнього повідомлення в усій колекції `messages`, копія останнього повідомлення дублюється безпосередньо в документ чату. Це забезпечує денормалізацію даних (Data Denormalization) та оптимізує навантаження на хмарну СУБД.

Повний лістинг програмної реалізації моделі чату наведено у **Додатку Ж**.

2.2.3. Схема зв'язків між колекціями (Логічна структура БД)

Незважаючи на те, що MongoDB належить до класу нереляційних баз даних, між її колекціями існують стійкі логічні зв'язки. У розробленому веб-застосунку застосовано зв'язки типу «один-до-багатьох» (One-to-Many). Один користувач (User) може виступати автором багатьох повідомлень (Message) та бути учасником багатьох чатів (Chat), у той час як один чат містить у собі множину повідомлень, пов'язаних через ідентифікатор `chatId`. Також реалізовано рекурсивний зв'язок на рівні колекції користувачів для побудови списків контактів.

Концептуальну графічну схему зв'язків між сутностями (ЕР-діаграму) та логічну структуру розробленої бази даних в MongoDB Atlas представлено на рисунку 2.2.

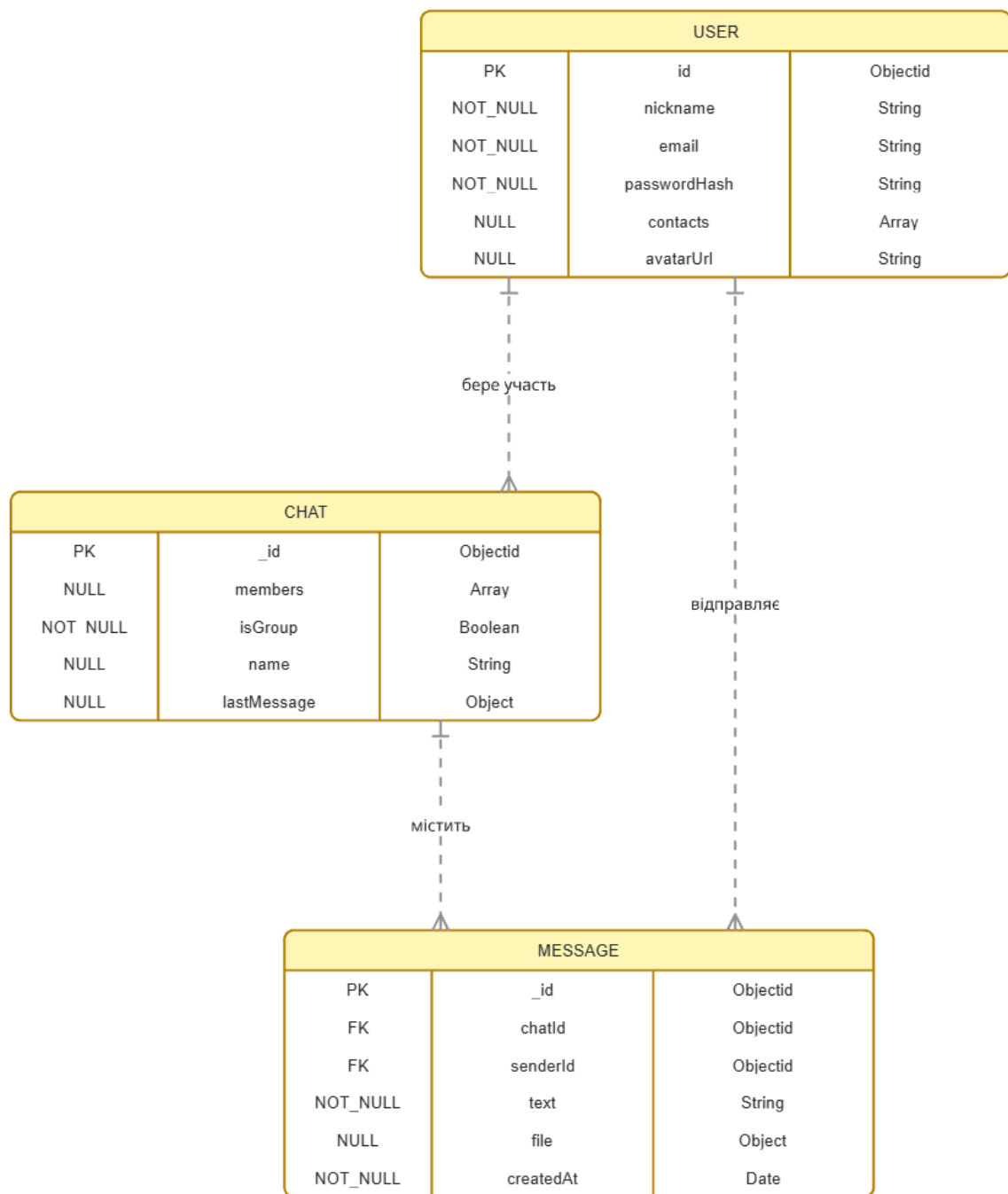


Рисунок 2.3 — Логічна схема зв'язків та структура моделей даних у СКБД MongoDB

Організація зв'язків через ідентифікатори `ObjectId` дозволяє серверній частині додатка ефективно використовувати метод динамічного об'єднання документів `.populate()` на рівні ODM `Mongoose`, що забезпечує високу швидкість агрегації даних під час формування відповідей для клієнтської частини.

2.3. Розробка серверної частини (Backend) на Node.js та Express

Серверна частина (Backend) є ядром координації та забезпечення бізнес-логіки веб-застосунку «Chatty». Вона виступає надійним посередником між клієнтським інтерфейсом та хмарною базою даних, забезпечуючи безпеку, валідацію, маршрутизацію запитів та керування сесіями користувачів. Для реалізації серверної сторони було обрано програмну платформу Node.js у поєднанні з гнучким веб-фреймворком Express, що дозволило створити високопродуктивне, асинхронне та легко масштабоване серверне рішення.

2.3.1. Налаштування оточення та конфігурація сервера

Першим етапом розробки серверної частини є створення безпечної конфігурації прикладного середовища. Веб-застосунок оперує конфіденційними даними, такими як рядки підключення до бази даних MongoDB Atlas, секретні ключі для підпису токенів та паролі від поштових шлюзів. Зберігання таких даних безпосередньо в коді програми є грубим порушенням правил безпеки.

Для вирішення цієї проблеми в проєкті було інтегровано та налаштовано бібліотеку `dotenv`. Вона дозволяє винести всі чутливі налаштування в окремий ізольований файл конфігурації середовища `.env`, який не потрапляє у відкритий репозиторій. Під час запуску сервера бібліотека автоматично зчитує ці змінні та інжектує їх у глобальний об'єкт Node.js — `process.env`.

Окремим критично важливим налаштуванням конфігурації є забезпечення механізму CORS (Cross-Origin Resource Sharing). Оскільки клієнтська частина чату (Frontend) та серверна бізнес-логіка (Backend) під час розробки та розгортання можуть функціонувати на різних доменах, піддоменах або портах, браузері за замовчуванням блокують такі крос-

доменні HTTP-запити з міркувань безпеки. Для надання санкціонованого доступу клієнту до розробленого REST API на сервері було впроваджено проміжне програмне забезпечення (middleware) cors, де чітко специфіковано дозволені білі списки (Allow-Origins), HTTP-методи (GET, POST, PUT, DELETE) та дозволені заголовки (Headers).

Окремим критично важливим налаштуванням конфігурації є забезпечення механізму **CORS (Cross-Origin Resource Sharing)**. Оскільки клієнтська частина чату (Frontend) та серверна бізнес-логіка (Backend) під час розробки та розгортання можуть функціонувати на різних доменах, піддоменах або портах, браузері за замовчуванням блокують такі крос-доменні HTTP-запити з міркувань безпеки. Для надання санкціонованого доступу клієнту до розробленого REST API на сервері було впроваджено проміжне програмне забезпечення (middleware) cors, де чітко специфіковано дозволені білі списки (Allow-Origins), HTTP-методи (GET, POST, PUT, DELETE) та дозволені заголовки (Headers).

Повний вихідний код програмної конфігурації ініціалізації сервера, підключення модулів безпеки, обробки JSON-пакетів та зчитування файлу оточення наведено у Додатку Т.

2.3.2. Система безпеки, автентифікації та захисту маршрутів

Безпека користувацьких даних є пріоритетним завданням при розробці сучасних месенджерів. Система безпеки сервера «Chatty» базується на двох фундаментальних механізмах: надійному криптографічному захисті паролів та безсесійній автентифікації на основі токенів.

1. Хешування паролів за допомогою bcryptjs: Зберігання паролів у базі даних у відкритому або просто зашифрованому вигляді є неприпустимим. Для захисту облікових записів від компрометації (наприклад, у випадку витоку бази даних) в проєкті використано

бібліотеку `bcryptjs`. Під час реєстрації користувача або зміни пароля, рядок символів проходить обробку алгоритмом адаптивного хешування з додаванням випадкової солі (`salt`) з коефіцієнтом розширення (`salt rounds`) рівним 10. Алгоритм є одностороннім — отримати вихідний пароль із готового хешу технічно неможливо, а автентифікація відбувається шляхом криптографічного порівняння введеного користувачем рядка із наявним у базі хешем за допомогою методу `bcrypt.compare()`.

2. Автентифікація на базі JSON Web Tokens (JWT): Для реалізації механізму захищеного доступу до функцій месенджера було обрано концепцію Stateless JWT. Після успішного введення логіна та пароля сервер генерує унікальний рядок — JWT-токен, зашифрований секретним ключем додатка. Токен містить у своєму корисній навантаженні (`payload`) унікальний ідентифікатор користувача (`id`) та термін дії. Клієнт зберігає цей токен і додає його до заголовка `Authorization: Bearer <token>` кожного наступного HTTP-запиту.

Для захисту приватних маршрутів (наприклад, отримання списку чатів чи профілю користувача) було розроблено спеціальне захисне `middleware` автентифікації. Воно перехоплює запит, вилучає токен, перевіряє його валідність за допомогою секретного ключа і, у разі успіху, декодує дані користувача, передаючи керування кінцевому контролеру. Якщо токен відсутній, підроблений або термін його дії закінчився, сервер миттєво блокує запит, повертаючи помилку з HTTP-статусом 401 Unauthorized.

Програмну реалізацію проміжного шару захисту приватних маршрутів за допомогою JWT представлено у **Додатку Е**.

2.3.3. Реалізація архітектури REST API та ендпоінтів

Взаємодія між клієнтом та сервером для виконання стандартних операцій (реєстрація, автентифікація, робота з профілем) побудована за принципами архітектурного стилю REST API. Сервер надає набір уніфікованих точок доступу (ендпоінтів), які приймають та повертають дані у форматі JSON, використовуючи стандартні методи протоколу HTTP.

В межах розробки було спроектовано та програмно реалізовано такі ключові маршрути (маршрутизатори Express):

- POST /api/auth/register — ендпоінт реєстрації нового користувача. Приймає об'єкти з параметрами nickname, email, password. Проводить валідацію на унікальність полів, викликає сервіс надсилання листів верифікації, зберігає нового користувача з відхешованим паролем та повертає згенерований JWT-токен.
- POST /api/auth/login — ендпоінт авторизації. Перевіряє наявність користувача за email, порівнює паролі через bcryptjs і у разі успіху повертає токен для клієнтської сесії.
- PUT /api/user/profile — захищений маршрут для оновлення персональних даних користувача (зміна полів displayName, bio, phone).
- POST /api/user/avatar — захищений ендпоінт для завантаження або оновлення файлу зображення профілю (аватара).

Детальний опис архітектури побудови маршрутизації, а також повні лістинги програмної реалізації логіки контролерів для забезпечення процесів реєстрації та автентифікації користувачів наведено у Додатку Д.

2.3.4. Обробка файлових потоків та інтеграція Multer

Функціональні вимоги до месенджера «Chatty» передбачають можливість обміну медіафайлами у повідомленнях, а також завантаження персональних аватарів. Стандартний фреймворк Express не має вбудованих

інструментів для аналізу багатокomпонентних даних типу multipart/form-data, які використовуються для завантаження файлів через форми.

Для реалізації цього функціоналу на сервері було впроваджено та налаштовано бібліотеку Multer як спеціалізоване middleware для обробки файлових потоків. Multer перехоплює вхідний потік даних, виділяє з нього файл, проводить жорстку інженерну валідацію за такими критеріями:

1. *Валідація за типом файлу (Mimetype filter)*: обмеження завантаження потенційно небезпечних файлів (наприклад, .exe чи .bat скриптів). Для аватарів дозволено виключно графічні формати (image/jpeg, image/png, image/webp).
2. *Обмеження розміру (File size limit)*: встановлення лімітів на максимальний розмір завантажуваного файлу для запобігання переповнення дискового простору сервера та перевантаження мережі.

Після успішної перевірки Multer автоматично генерує унікальне ім'я файлу (щоб уникнути колізій при завантаженні файлів з однаковими назвами різними користувачами), зберігає його у цільову директорію на сервері або у хмарне сховище, а метадані файлу (шлях, назву, розмір, mimetype) записує в об'єкт запису Express req.file або req.files. Після цього контролер повідомлень може зберегти ці дані у вкладений об'єкт file моделі Message.

Програмне налаштування middleware Multer для завантаження файлів та управління дисковим сховищем наведено у Додатку Є.

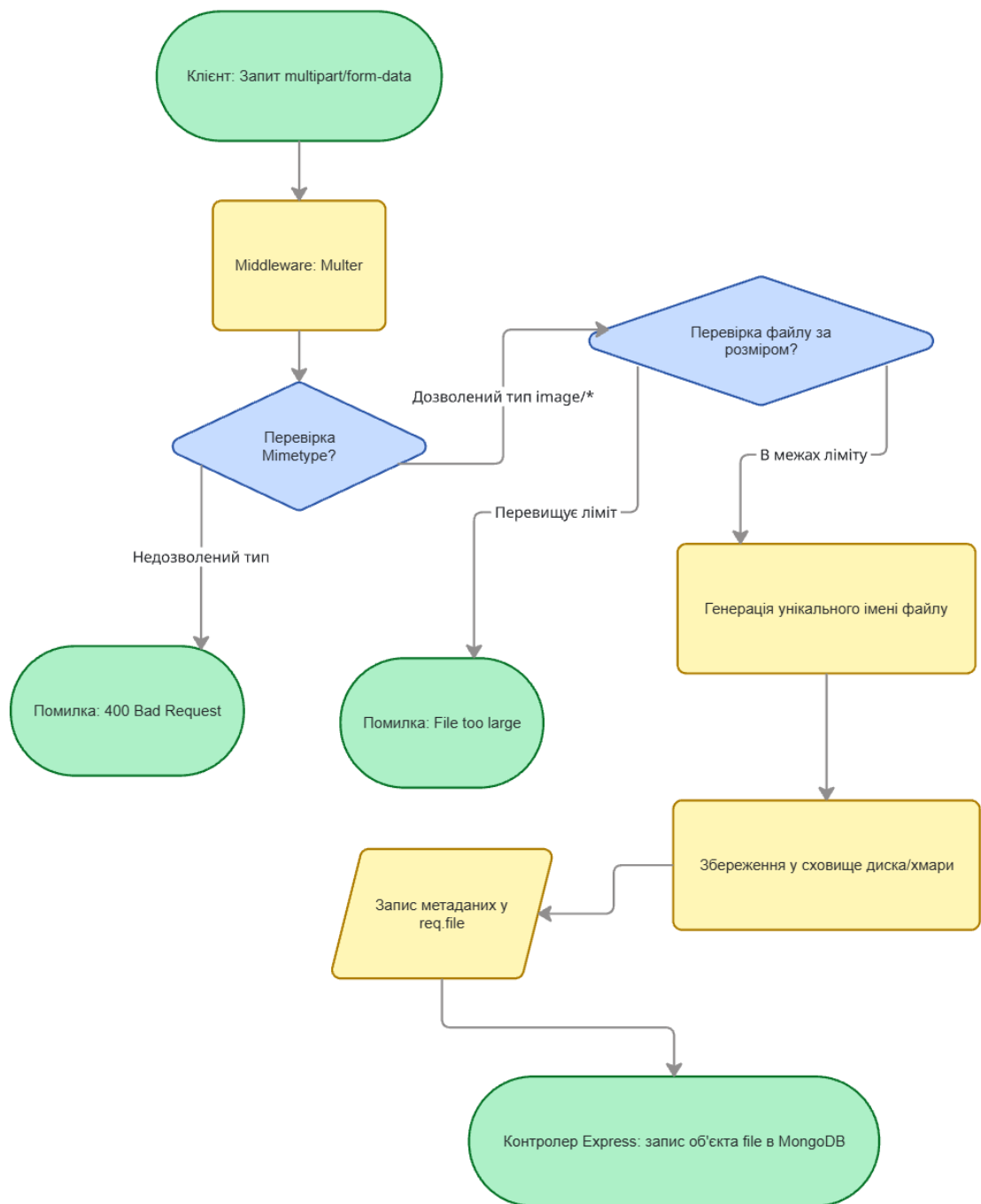


Рисунок 2.4 — Схема процесу обробки та валідації файлових потоків за допомогою модуля Multer

2.3.5. Реалізація сервісу сповіщень на базі Nodemailer

Для підтвердження реєстрації облікових записів, надсилання системних кодів та підвищення рівня безпеки до серверної частини додатка було

інтегровано підсистему Email-сповіщень. Практична реалізація цього сервісу виконана за допомогою бібліотеки Nodemailer.

Nodemailer налаштовано на роботу за протоколом SMTP (Simple Mail Transfer Protocol) із використанням безпечного шифрованого з'єднання TLS/SSL. Сервер бізнес-логіки автентифікується на зовнішньому поштовому шлюзі (наприклад, SendGrid або SMTP-сервері поштового провайдера) за допомогою захищених змінних оточення та ініціює відправку системних повідомлень.

Архітектурною особливістю розробленого сервісу сповіщень є підтримка двомовної локалізації (en/uk), що безпосередньо відповідає налаштуванням у профілі користувача. Перед відправкою листа сервіс перевіряє поле locale отримувача і динамічно підставляє відповідний HTML-шаблон повідомлення:

- При виборі локалі uk користувач отримує системний лист українською мовою з офіційним підтвердженням та інструкціями;
- При виборі en сервіс рендерить та відправляє повністю англomовний аналог шаблону.

Програмний код модульної реалізації поштового сервісу з інтеграцією поштового шлюзу представлено у **Додатку С**. Загальну високорівневу логіку сервісу верифікації (обробка типів запитів на реєстрацію або відновлення доступу, генерація 6-значних числових кодів) наведено у **Додатку Р**. Програмний модуль конфігурації системи інтернаціоналізації (i18n) розташовано у **Додатку У**, а файли статичних локалізованих перекладів та мовних ресурсів представлені відповідно у **Додатку Х** (uk.json), **Додатку Ц** (es.json), **Додатку Ч** (en.json) та **Додатку Ю** (de.json).

2.4. Реалізація повнодуплексного обміну повідомленнями через Socket.io

Традиційні веб-технології, що базуються на класичних HTTP-запитах типу «запит-відповідь», не здатні забезпечити ефективне функціонування сучасних інтерактивних чат-додатків. Постійне опитування сервера (polling) створює критичне навантаження на мережу та серверні потужності, а також спричиняє затримку при доставці даних.

Для забезпечення миттєвого обміну інформацією між користувачами месенджера «Chatty» було впроваджено технологію WebSocket за допомогою спеціалізованого фреймворку Socket.io. Це дозволило реалізувати стійке, низькозатримкове повнодуплексне (двостороннє) з'єднання, в якому як клієнт, так і сервер можуть ініціювати відправку даних у будь-який момент часу.

2.4.1. Ініціалізація WebSocket-сервера та інтеграція з Express

Процес впровадження Socket.io в архітектуру серверної частини вимагає інтеграції протоколу WebSocket безпосередньо поверх базового HTTP-сервера додатка. Оскільки стандартний фреймворк Express самостійно керує лише HTTP-маршрутизацією, для розширення його можливостей було використано вбудований у Node.js модуль http.

Архітектурний процес ініціалізації сервера реалізовано за такою схемою:

1. За допомогою методу `http.createServer(app)` створюється екземпляр базового веб-сервера, куди як обробник передається додаток Express. Це дозволяє серверу одночасно обслуговувати як стандартні REST API ендпоінти, так і нові типи з'єднань.
2. Створюється екземпляр сокет-сервера шляхом передачі HTTP-сервера в конструктор `new Server(httpServer, config)`.
3. У конфігурації ініціалізації сокетів обов'язково налаштовуються параметри політики безпеки CORS. Серверу сокетів передаються

санкціоновані параметри клієнтського домену та дозволені методи обміну (`methods: ["GET", "POST"]`), що унеможливорює несанкціоноване підключення сторонніх скриптів до каналів зв'язку.

Після успішної ініціалізації сервер починає прослуховувати визначений порт, де під час першого звернення клієнта відбувається процедура HTTP Handshake (рукостискання), після чого з'єднання автоматично та безперервно переводиться на захищений повнодуплексний протокол WebSocket.

2.4.2. Архітектура подій (Event-Driven Architecture)

Взаємодія в Socket.io побудована на базі подійно-орієнтованої архітектури (Event-Driven Architecture). Замість обробки статичних маршрутів, сервер і клієнт підписуються на певні іменовані події та надсилають (генерують) їх разом із корисним навантаженням у форматі JSON.

У межах розробки бізнес-логіки чату «Chatty» на серверній стороні було спроектовано та реалізовано систему обробників для таких базових подій:

1. `connection` — головна системна подія, яка спрацьовує, коли новий клієнт успішно пройшов етап рукостискання. Сервер фіксує унікальний ідентифікатор сокета (`socket.id`), що виділяється кожній активній вкладці браузера, та відкриває індивідуальний асинхронний канал зв'язку з цим користувачем.
2. `sendMessage` — користувацька подія, яка генерується клієнтом у момент натискання кнопки відправки повідомлення. Обробник цієї події на сервері приймає об'єкт повідомлення (ідентифікатор чату, текст, метадані файлу). Сервер проводить первинну валідацію отриманих даних, асинхронно звертається до ODM Mongoose для фізичного збереження запису в базі даних MongoDB Atlas, після чого ініціює ретрансляцію даних.

3. `message` — подія доставки повідомлення. Після успішного збереження в базі даних сервер пересилає сформований документ повідомлення цільовим учасникам чату. Завдяки цьому отримувач миттєво бачить нове повідомлення у своєму вікні інтерфейсу, а відправник отримує підтвердження успішної доставки.
4. `typing` / `stopTyping` — події для забезпечення інтерактивності інтерфейсу (індикатори введення тексту). Коли користувач починає набирати текст у полі введення, клієнтська частина з невеликою затримкою (`debounce`) генерує подію `typing` та передає її на сервер разом із ID чату. Сервер миттєво пересилає цю подію іншому учаснику діалогу, що дозволяє відобразити на екрані анімоване сповіщення «користувач набирає повідомлення...». При припиненні введення генерується подія `stopTyping`, яка прибирає індикатор.
5. `disconnect` — системна подія розриву з'єднання. Спрацьовує автоматично, коли користувач закриває вкладку застосунку, оновлює сторінку або втрачає мережеве підключення. Обробник на сервері коректно очищує пам'ять від неактивного сокета, видаляє користувача зі списку присутніх у кімнатах чату та може транслявати іншим учасникам статус «офлайн».

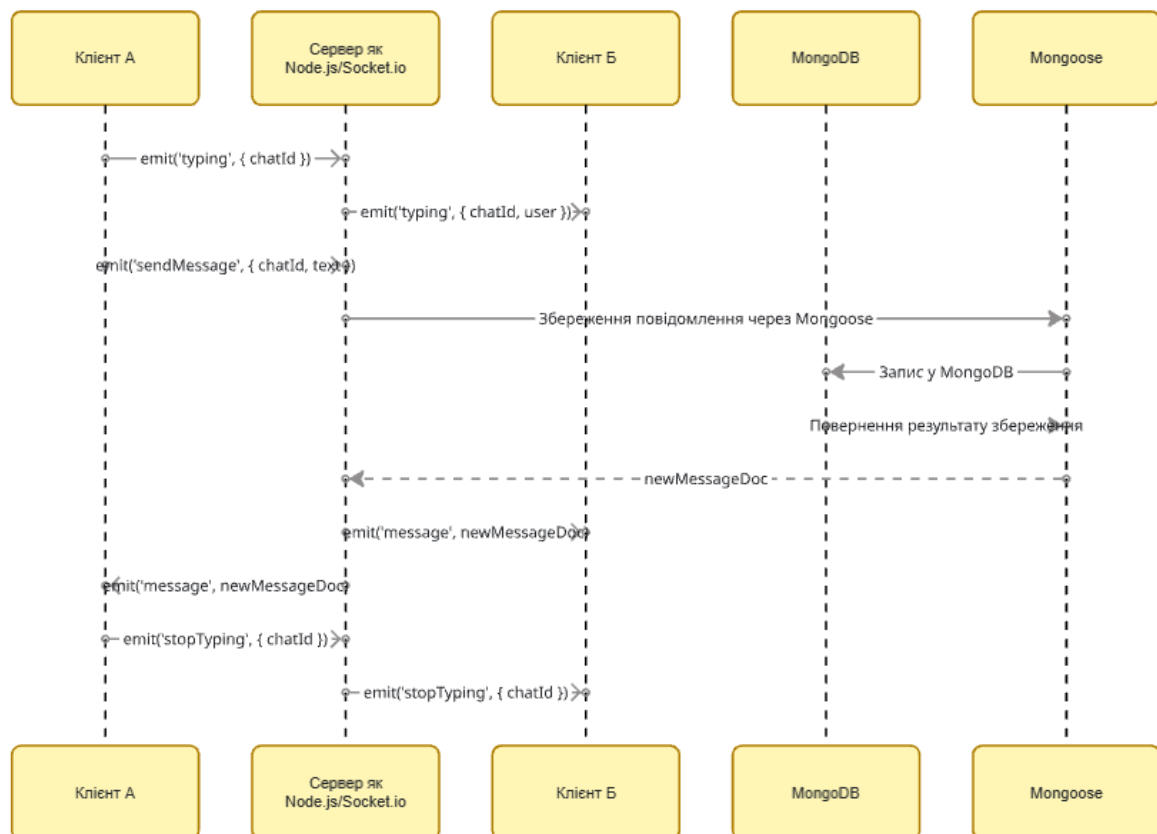


Рисунок 2.5 — Діаграма послідовності обробки мережесих подій у реальному часі

2.4.3. Менеджмент кімнат (Rooms) та ізоляція приватних чатів

Критично важливим завданням при розробці архітектури месенджера є ізоляція потоків даних. Повідомлення, надіслане у приватний чат між Користувачем А та Користувачем Б, у жодному разі не повинно транслюватися іншим підключеним до сервера клієнтам.

Для вирішення цього завдання було використано вбудований у Socket.io потужний механізм кімнат (Rooms). Кімнати — це віртуальні ізольовані канали на рівні сервера, в які сокети користувачів можуть об'єднуватися для спільного прийому та передачі подій.

Логіка менеджменту приватних чатів у «Chatty» побудована за таким алгоритмом:

1. Під час відкриття вікна діалогу клієнт ініціює запит на підключення до конкретного простору спілкування.
2. Сервер отримує унікальний ідентифікатор чату (`chatId`) з моделі даних `Chat`.
3. За допомогою вбудованого методу `socket.join(chatId)` сервер примусово додає поточний сокет користувача у віртуальну кімнату, назвою якої є сам `chatId`. Якщо кімнати з такою назвою ще не існувало в оперативній пам'яті сервера, `Socket.io` створює її автоматично.
4. Коли один із учасників надсилає повідомлення через подію `sendMessage`, сервер не робить загальну розсилку (`broadcast`) всім користувачам мережі. Замість цього він використовує конструкцію `io.to(chatId).emit('message', data)`, яка здійснює цільову доставку повідомлення виключно тим сокетам, що зараз додані до цієї конкретної кімнати `chatId`.
5. Під час виходу з чату або відкриття іншого діалогу автоматично викликається метод `socket.leave(chatId)`, що відключає користувача від поточного віртуального каналу та захищає конфіденційність листування.

Така архітектурна організація за допомогою кімнат забезпечує високу безпеку системи, надійну ізоляцію приватних даних, а також відмінну масштабованість, дозволяючи без зміни логіки сервера в майбутньому створювати групові кімнати на будь-яку кількість учасників.

Повний лістинг програмної реалізації ініціалізації `WebSocket`-сервера, конфігурації `CORS` для сокетів та обробників усієї подійно-орієнтованої логіки чату разом із менеджментом кімнат винесено у Додаток Г.

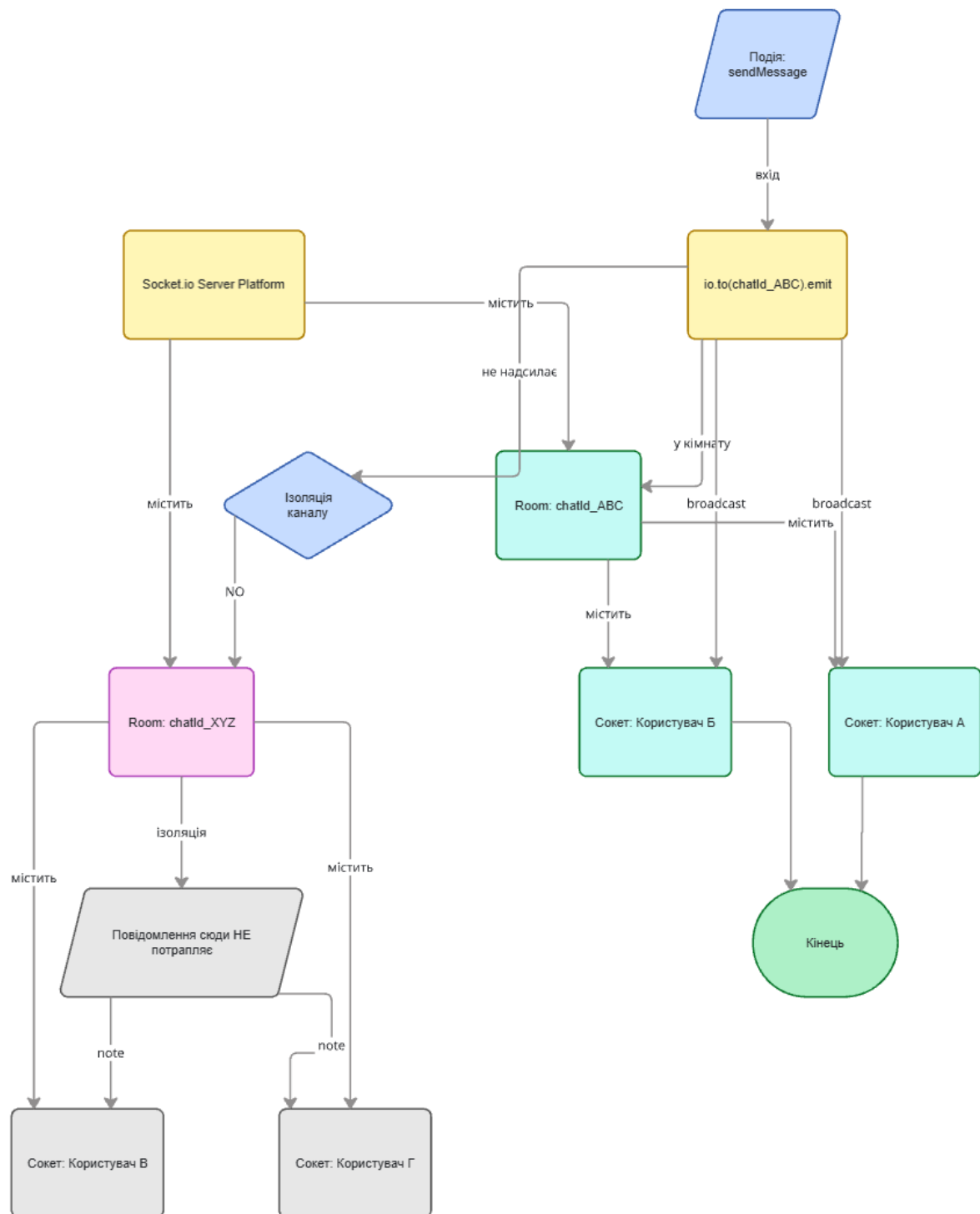


Рисунок 2.6 — Схема ізоляції приватних чатів через механізм віртуальних кімнат Socket.io Rooms

2.5. Розробка клієнтської частини (Frontend) та інтерфейсу користувача

Клієнтська частина (Frontend) веб-застосунку «Chatty» є основним середовищем взаємодії кінцевого користувача з системою. Оскільки месенджер орієнтований на повсякденне використання, до його інтерфейсу

висуваються високі вимоги щодо швидкості відгуку, інтерактивності, ергономіки (UX) та візуальної привабливості (UI). Клієнтська частина розроблена як SPA-орієнтована (Single Page Application) система на базі сучасного технологічного стеку HTML5, CSS3 та JavaScript (ES6+), що дозволило забезпечити плавну роботу інтерфейсу без постійного перезавантаження сторінок браузера.

2.5.1. Модульна структура веб-додатку на стороні клієнта

Для забезпечення чистоти коду, простоти тестування та зручності подальшого розширення функціоналу, архітектуру Frontend-частини було спроектовано за модульним принципом. Весь клієнтський код чітко розподілено на функціональні шари:

1. **Шар представлення (HTML5 Documents / Templates):** відповідає за декларативну розмітку структури сторінок та елементів керування. Використання семантичних тегів HTML5 (<aside>, <main>, <section>, <header>) дозволило створити логічний каркас додатка, оптимізований для зчитування браузерними рушіями та забезпечення доступності (Accessibility).
2. **Шар стилізації (CSS3 Modules / Themes):** ізольовані файли стилів, які відповідають за геометрію інтерфейсу, шрифти, адаптивну сітку та анімаційні ефекти. Організація стилів базується на використанні CSS-змінних (Custom Properties), що є ключовим інструментом для динамічної кастомізації.
3. **Шар логіки та управління станом (JavaScript Modules):** ядро клієнтської частини, що розподілене на окремі логічні модулі:
 - `api.js` — модуль, що відповідає за відправку асинхронних HTTP-запитів (`fetch/axios`) до REST API сервера для автентифікації та завантаження профілю;

- `socket.js` — модуль, який ініціалізує WebSocket-з'єднання та інкапсулює в собі обробники подій реального часу;
- `ui.js` / `dom.js` — модуль маніпуляції елементами інтерфейсу, що відповідає за динамічний рендеринг повідомлень та оновлення екрана.

Взаємодію між архітектурними шарами представлення, стилізації та логіки керування станом додатка на стороні браузера схематично зображено на рисунку 2.7

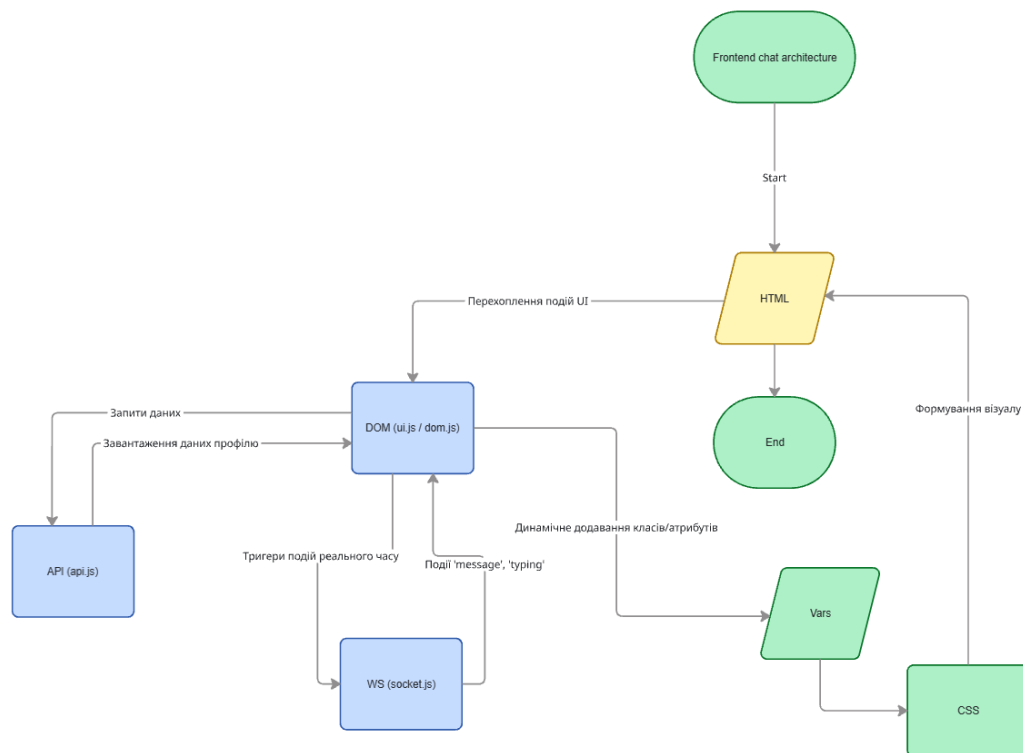


Рисунок 2.7 — Модульна структура та схема взаємодії компонентів клієнтської частини

2.5.2. Стилiзація, адаптивний дизайн та кастомiзація інтерфейсу

Візуальна складова месенджера розроблена з використанням сучасних методологій CSS-верстки. Головними завданнями при проектуванні UI-шару були забезпечення адаптивності та реалізація механізму зміни тем.

- **Адаптивний дизайн (Responsive Web Design):** для того, щоб інтерфейс однаково зручно функціонував як на моніторах десктопів, так і на екранах смартфонів, було використано технології **CSS Flexbox**

та **CSS Grid Layout** у поєднанні з медіа-запитами (@media). На великих екранах додаток має класичну двопанельну структуру (ліворуч — список чатів та контактів, праворуч — активне вікно діалогу). При зменшенні ширини екрана (на мобільних пристроях) інтерфейс динамічно трансформується у однопанельний режим, де список чатів та відкритий діалог перемикаються за допомогою плавних анімаційних переходів, оптимізованих під сенсорне керування.

- **Реалізація світлої та темної тем (Theme Customization):** кастомізація колірної палітри реалізована за допомогою глобальних CSS-змінних, оголошених у псевдокласі :root. Було створено два набори колірних схем (для світлого та темного режимів), які керують колірним фоном, текстом, картками повідомлень та елементами навігації. Перемикання тем відбувається шляхом додавання або видалення дата-атрибута (наприклад, data-theme="dark") до кореневого елемента <html> за допомогою JavaScript. Обрана користувачем тема автоматично фіксується у локальному сховищі браузера (localStorage), що дозволяє зберігати налаштування при повторних візитах на сайт.

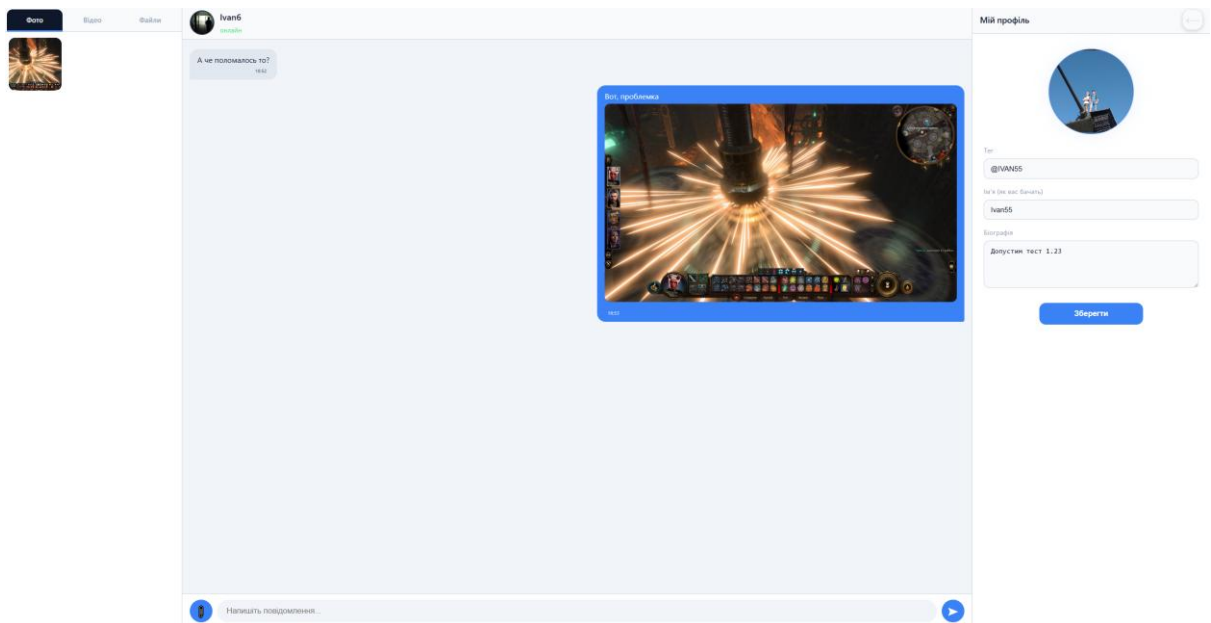


Рисунок 2.8 — Інтерфейс користувача веб-застосунку «Chatty» у десктопній (Light Mode) та (Dark Mode) версіях

- **Файлові прев'ю та візуальні ефекти:** для покращення користувацького досвіду (UX) при обміні медіафайлами було розроблено модуль попереднього перегляду (File Preview). Коли користувач обирає файл або зображення для відправки, JavaScript за допомогою API `URL.createObjectURL()` або `FileReader` миттєво генерує тимчасове локальне посилання на файл в пам'яті браузера. Це дозволяє відобразити мініатюру зображення або іконку документа з індикатором завантаження безпосередньо у полі введення тексту *до того*, як файл буде фізично відправлено на сервер. Також реалізовано плавні CSS-анімації для демонстрації статусів надсилання, появи нових повідомлень та ефекту «пульсації» для індикатора введення тексту іншим користувачем.

Візуальну реалізацію компонента попереднього перегляду медіафайлів у середині форми введення тексту наведено на рисунку 2.9

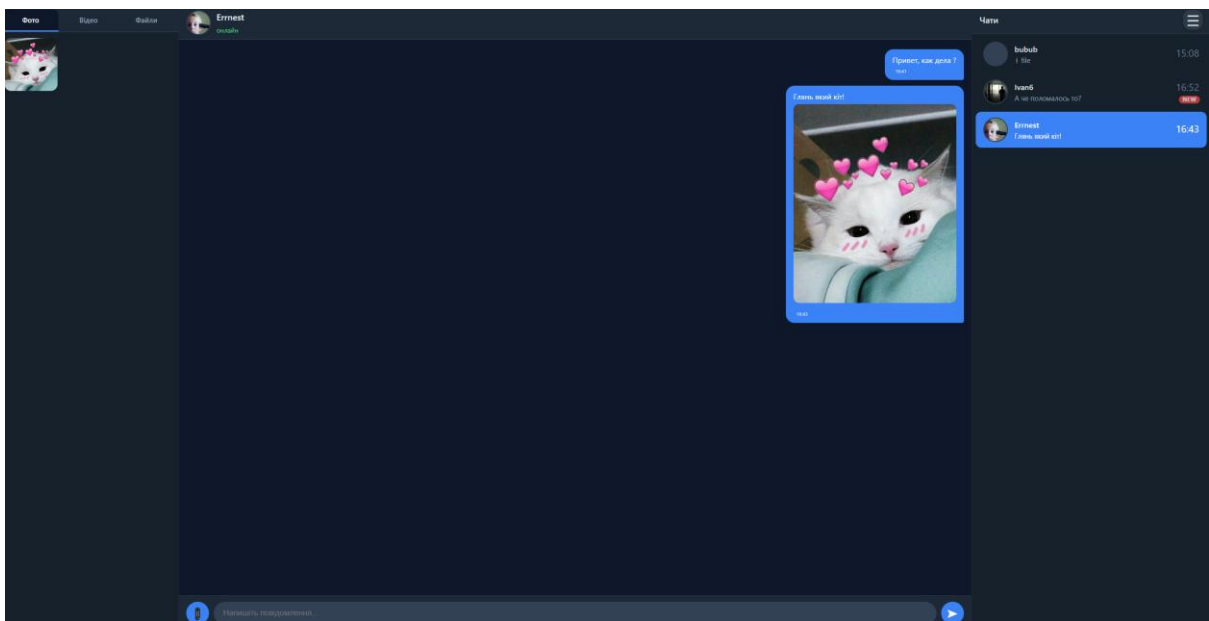


Рисунок 2.9 — Інтерфейс компонента попереднього перегляду медіафайлів (File Preview) перед відправкою

2.5.3. Клієнтська WebSocket-логіка та динамічне оновлення DOM

Ключовою особливістю Frontend-частини месенджера «Chatty» є робота з асинхронними потоками даних у реальному часі без перезавантаження сторінки. Цей функціонал забезпечується клієнтською бібліотекою `socket.io-client`.

Під час успішного проходження авторизації додаток ініціює підключення до сервера за допомогою методу `io(SERVER_URL)`. Клієнтський сокет отримує унікальний ідентифікатор і підписується на події сервера.

Процес динамічного оновлення інтерфейсу під час обміну повідомленнями відбувається за таким алгоритмом:

1. Користувач вводить текст, прикріплює файл і натискає кнопку «Надіслати». JavaScript-контролер перехоплює подію відправки форми (`submit`), блокує стандартну поведінку браузера (щоб сторінка не перезавантажувалася) та формує об'єкт даних.
2. Клієнт генерує подію `socket.emit('sendMessage', messageData)`.
3. При отриманні сервером цього повідомлення та його ретрансляції у кімнату чату, у сокета отримувача (та відправника) спрацьовує слухач події `socket.on('message', (newMessage) => { ... })`.
4. Всередині цього обробника викликається функція динамічного оновлення **DOM-дерева (Document Object Model)**. JavaScript за допомогою методів `document.createElement()` створює нові блоки HTML-розмітки для картки повідомлення, наповнює їх отриманим текстом, метаданими файлу та міткою часу, після чого інтегрує новий елемент у кінець списку повідомлень за допомогою методу `.appendChild()`.

5. Одразу після модифікації DOM-дерева викликається метод автоматичного скролінгу `.scrollIntoView({ behavior: 'smooth' })`, який плавно зміщує екран до останнього надісланого повідомлення, забезпечуючи природну поведінку стрічки чату.

Такий підхід дозволяє досягти монолітності інтерфейсу, коли всі зміни — поява тексту, зміна статусів, робота індикаторів введення тексту (typing) — відбуваються миттєво та непомітно для користувача на рівні оперативної пам'яті браузера, гарантуючи найвищу швидкість роботи веб-застосунку.

Повні структури стилів кастомізації тем, CSS-конфігурацію адаптивної верстки, а також лістинги клієнтського модуля обробки сокетних подій та динамічної маніпуляції DOM-деревом винесено у Додаток Т.

ВИСНОВКИ ДО РОЗДІЛУ 2

У другому розділі кваліфікаційної роботи було виконано повний комплекс інженерних завдань, пов'язаних із системним аналізом технічних вимог, архітектурним проєктуванням, розробкою структури бази даних, а також програмною реалізацією серверної та клієнтської частин веб-застосунку «Chatty» для обміну повідомленнями в режимі реального часу.

На основі проведених досліджень та виконаної розробки можна зробити такі науково-практичні висновки:

За результатами аналізу вимог та архітектурного проєктування було обґрунтовано доцільність застосування класичної трирівневої архітектурної моделі (Three-tier Architecture). Розподіл системи на ізольовані шари — рівень представлення (Frontend), рівень бізнес-логіки (Backend) та рівень даних (Database) — дозволив створити гнучку структуру з чітким розмежуванням зон відповідальності компонентів. Такий підхід забезпечує високий рівень безпеки (виключаючи пряму взаємодію клієнта з базою даних) та створює технологічний фундамент для незалежного масштабування кожного рівня системи в умовах зростання кількості активних користувачів.

Впровадження документоорієнтованої NoSQL СУБД MongoDB (на базі хмарного кластера MongoDB Atlas) виявилось найбільш ефективним інженерним рішенням для специфіки real-time месенджера у порівнянні з реляційними аналогами. Гнучка структура документів у форматі BSON дозволила уникнути складних операцій об'єднання таблиць (JOIN) та ресурсомістких міграцій на етапі розширення функціоналу. Застосування концепції денормалізації даних, зокрема дублювання копії останнього повідомлення безпосередньо у документ моделі чату (lastMessage), дозволило мінімізувати кількість запитів до бази даних під час рендерингу списків, суттєво знизивши навантаження на дискову підсистему та хмарний процесор. Використання ODM-бібліотеки Mongoose забезпечило строгу типізацію та валідацію схем даних на рівні прикладного коду сервера.

Розробка серверної частини на базі платформи Node.js та фреймворку Express дозволила реалізувати високопродуктивне асинхронне REST API для виконання стандартних операцій (реєстрація, автентифікація, менеджмент профілів). Безпека користувацьких даних була забезпечена впровадженням сучасних криптографічних протоколів: одностороннього хешування паролів за допомогою алгоритму bcryptjs із додаванням випадкової солі, а також механізму безсесійної автентифікації на основі токенів JSON Web Tokens (JWT). Це дозволило захистити приватні маршрути додатка та знизити навантаження на оперативну пам'ять сервера за рахунок реалізації архітектурного принципу Stateless.

Інтеграція спеціалізованого middleware Multer забезпечила надійне керування багатокомпонентними файловими потоками (multipart/form-data). Створена система інженерної валідації вхідних файлів за MIME-типом та максимальним розміром дозволила нейтралізувати ризики завантаження потенційно небезпечних скриптів та переповнення серверного сховища. Впровадження сервісу сповіщень на базі бібліотеки Nodemailer за протоколом SMTP із шифруванням TLS забезпечило безпечну верифікацію користувачів під час реєстрації, а інтеграція модулів інтернаціоналізації (i18n) дозволила динамічно адаптувати HTML-шаблони системних листів під обрану локаль користувача.

Реалізація повнодуплексного обміну даними була успішно виконана за допомогою WebSocket-фреймворку Socket.io. Подійно-орієнтована архітектура (Event-Driven Architecture) дозволила відійти від неефективних методів постійного опитування сервера (polling) та знизити затримку доставки повідомлень (latency) до нормативних 200–300 мс. Використання вбудованого механізму віртуальних кімнат (Rooms) забезпечило абсолютну ізоляцію приватних каналів зв'язку між користувачами, гарантуючи конфіденційність листування та виключаючи можливість несанкціонованого перехоплення даних сторонніми клієнтами сокет-сервера.

Проектування та розробка клієнтської частини (Frontend) у концепції Single Page Application (SPA) на базі HTML5, CSS3 та JavaScript (ES6+) дозволили створити монолітний інтерактивний інтерфейс. Використання сучасних методологій CSS Flexbox та CSS Grid у поєднанні з медіа-запитами (@media) забезпечило 100% адаптивність дизайну додатка для мобільних та десктопних пристроїв. Динамічна кастомізація колірних схем (Light/Dark режими) була реалізована через глобальні CSS-змінні із фіксацією вибору користувача у локальному сховищі браузера (localStorage). Завдяки використанню клієнтського сокет-клієнта та методів маніпуляції DOM-деревом (зокрема, appendChild та scrollIntoView), усі інтерфейсні події — від появи повідомлень до роботи анімованих індикаторів введення тексту (typing) — відбуваються миттєво в оперативній пам'яті браузера без перезавантаження веб-сторінок.

Таким чином, розроблений у другому розділі програмно-архітектурний комплекс повністю відповідає висунутим функціональним та нефункціональним технічним вимогам, є масштабованим, безпечним та оптимованим для стабільної роботи в умовах реального часу, що створює необхідну базу для подальшого розгортання, тестування та впровадження продукту.

ВИСНОВОК

У кваліфікаційній роботі успішно вирішено науково-практичне завдання з дослідження, проектування, програмної реалізації та тестування кросплатформеного веб-застосунку для обміну повідомленнями в режимі реального часу «Chatty». Створений програмний продукт повністю відповідає інженерним стандартам, критеріям безпеки та специфікаціям технічного завдання.

На основі проведеного аналізу теоретичних засад розподілених систем обґрунтовано доцільність застосування трирівневої архітектури, яка дозволила чітко розмежувати клієнтський шар представлення, серверну логіку та шар збереження даних, мінімізуючи ризики прямого доступу до СКБД. Дослідження мережевих протоколів взаємодії довело неефективність традиційних HTTP-підходів, натомість інтеграція повнодуплексного подієво-орієнтованого протоколу WebSocket забезпечила миттєву доставку повідомлень у межах нормативних 200–300 мс. Для побудови гнучкої та стійкої структури бази даних було обрано NoSQL СУБД MongoDB у хмарі MongoDB Atlas. Відмова від реляційної моделі на користь BSON-документів та впровадження концепції денормалізації через дублювання об'єкта останнього повідомлення у колекції чатів дозволили оптимізувати вибірку даних та знизити навантаження на хмарний кластер, а використання ODM Mongoose забезпечило строгу валідацію схем на рівні бекенду.

Серверна частина застосунку, конфігурована на базі платформи Node.js та фреймворку Express з асинхронною моделлю введення-виведення, гарантує стабільну обробку великої кількості одночасних підключень. Безпека системи реалізована за принципом Stateless на основі токенів JSON Web Tokens та криптографічного хешування паролів алгоритмом bcryptjs. Налаштування middleware Multer забезпечило жорстку валідацію вхідних медіафайлів, а впровадження модуля Nodemailer за протоколом SMTP — надійну верифікацію користувачів. Інтеграція інструментів інтернаціоналізації (i18n) дозволила динамічно змінювати мову системних

повідомлень відповідно до локалі користувача. Ядро реального часу побудовано на базі Socket.io, де завдяки механізму віртуальних кімнат (Rooms) досягнуто абсолютної ізоляції приватних каналів спілкування та інтерактивних індикаторів введення тексту (typing).

Клієнтську частину реалізовано в концепції Single Page Application (SPA) за допомогою стеків HTML5, CSS3 та асинхронного JavaScript (ES6+). Завдяки методологіям CSS Flexbox, Grid та медіа-запитам забезпечено 100% адаптивність та кросбраузерність інтерфейсу додатка для мобільних і десктопних пристроїв, а також впроваджено динамічну кастомізацію тем (Light/Dark) через локальне сховище браузера. Створення компонента попереднього перегляду файлів через File API покращило користувацький досвід (UX), а прямі маніпуляції з DOM-деревом дозволили досягти монолітності інтерфейсу, оскільки всі оновлення екрана відбуваються без перезавантаження сторінок.

Практична цінність роботи полягає у створенні готового до розгортання, масштабованого та безпечного месенджера. Розроблений Fullstack JavaScript комплекс може слугувати технологічним прототипом для корпоративних чат-систем або базою для подальших досліджень у сфері оптимізації високозавантажених real-time застосунків. Поставлені завдання роботи виконано в повному обсязі, а мету дослідження успішно досягнуто.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Національний стандарт України. Інформація та документація. Бібліографічне посилання. Загальні положення та правила складання: ДСТУ 8302:2015. Вид. офіц. Київ: УкрНДНЦ, 2016. 17 с.
2. Закон України «Про авторське право і суміжні права» від 01.12.2022 № 2811-IX. *Відомості Верховної Ради України*. 2023. № 12. Стаття 45.
3. Закон України «Про захист інформації в інформаційно-комунікаційних системах» від 05.07.1994 № 80/94-ВР (поточна редакція). *Відомості Верховної Ради України*. 1994. № 31. Стаття 286.
4. Бублик В. В. Об'єктно-орієнтоване програмування: підручник. Київ: ІТ-книга, 2019. 424 с.
5. Голощук Р. О., Мазур М. В. Проектування та розробка вебзастосунків мовою JavaScript: навч. посіб. Львів: Видавництво Львівської політехніки, 2022. 212 с.
6. Гризун Л. Е., Ткаченко В. О. Технології створення сучасних веб-орієнтованих систем: навч. посіб. Харків: ХНПУ, 2021. 196 с.
7. Заболотня Т. М. Архітектура та проектування програмного забезпечення: конспект лекцій. Київ: КПІ ім. Ігоря Сікорського, 2020. 148 с.
8. Кравець П. О. Об'єктно-орієнтований аналіз та проектування програмних систем: навч. посіб. Львів: Видавництво Львівської політехніки, 2021. 304 с.
9. Ткаченко О. В. Розробка асинхронних веб-додатків у режимі реального часу: метод. вказівки. Київ: КНУБА, 2023. 64 с.
10. Чинков В. М. Проектування та розробка веб-застосунків у режимі реального часу: навч. посіб. Харків: ХНУРЕ, 2022. 184 с.
11. Швець А. В. Технології розробки програмного забезпечення на платформі Node.js: навч. посіб. Київ: КПІ ім. Ігоря Сікорського, 2023. 156 с.
12. Bankau J. Real-Time Web Application Development with Vert.x 4. Birmingham: Packt Publishing, 2022. 340 p.

13. Bass L., Clements P., Kazman R. Software Architecture in Practice. 4th ed. Boston: Addison-Wesley, 2021. 448 p.
14. Cantelon M., Harter M. Node.js in Action. 2nd ed. New York: Manning Publications, 2019. 432 p.
15. Chodorow K. MongoDB: The Definitive Guide. 3rd ed. Sebastopol: O'Reilly Media, 2020. 514 p.
16. Duckett J. JavaScript and jQuery: Interactive Front-End Web Development. Indianapolis: John Wiley & Sons, 2019. 496 p.
17. Fowler M. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley, 2019. 576 p.
18. Frisbie M. Professional JavaScript for Web Developers. 5th ed. Indianapolis: Wrox, 2024. 864 p.
19. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley, 2022. 416 p.
20. Herron D. Node.js Web Development: Client-Server Web Apps with JavaScript and Node.js. 5th ed. Birmingham: Packt Publishing, 2021. 544 p.
21. Holmes S., Sly W. MERN Quick Start Guide: Build Web Applications with MongoDB, Express, React, and Node. New York: Manning Publications, 2020. 410 p.
22. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol: O'Reilly Media, 2023. 616 p.
23. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston: Prentice Hall, 2021. 432 p.
24. Pasquali S., Faaborg F. Mastering Node.js. 3rd ed. Birmingham: Packt Publishing, 2020. 442 p.
25. Subramanian V. Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node. 2nd ed. Boston: Apress, 2019. 598 p.
26. Bcryptjs Library Documentation. *npm: node package manager*. URL: <https://www.npmjs.com/package/bcryptjs> (дата звернення: 12.04.2026).

- 27.Express.js: Fast, unopinionated, minimalist web framework for Node.js. *Express Official Website*. URL: <https://expressjs.com> (дата звернення: 10.03.2026).
- 28.JSON Web Tokens Introduction. *JWT.io*. URL: <https://jwt.io/introduction> (дата звернення: 18.04.2026).
- 29.MDN Web Docs: WebSockets API. *Mozilla Developer Network*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API (дата звернення: 05.05.2026).
- 30.MongoDB Documentation. *MongoDB Official Site*. URL: <https://docs.mongodb.com> (дата звернення: 22.03.2026).
- 31.Mongoose ODM v8.0 Documentation. *MongooseJS*. URL: <https://mongoosejs.com/docs> (дата звернення: 25.03.2026).
- 32.Multer Middleware for Express. *GitHub Repository*. URL: <https://github.com/expressjs/multer> (дата звернення: 02.04.2026).
- 33.Socket.io: The bidirectional communication library. *Socket.io Documentation*. URL: <https://socket.io/docs/v4> (дата звернення: 28.04.2026).

ДОДАТОК А

Міністерство освіти і науки України
Державний заклад «Луганський національний університет
імені Тараса Шевченка»
Факультет (інститут) Факультет технологій та інформаційних систем
(повна назва)
Кафедра Інформаційних технологій та систем
(повна назва)

Програма та методика тестування

на виконання програмної розробки (ПР):
Розробка веб-застосунку для обміну повідомленнями

Лубни – 2026

Зміст

ДОДАТОК А	97
ТЕСТ ПЛАН	61
1. Вступ.....	61
1.1. Мета.....	61
1.2. Вхідні дані.....	61
1.3. Мета тестування	62
2. Умови тестування.....	63
3. Стратегія процесу тестування	64
3.1. Тестування інтерфейсу	65
3.1.2. Функціональне тестування.....	78
4. План робіт	89
5. Кінцеві результати	90
ТЕСТОВІ ВИПАДКИ (TEST CASES).....	91

ТЕСТ ПЛАН

1. Вступ

1.1. Мета

Розробка тест-плану є фінальним і критично важливим етапом у загальному життєвому циклі забезпечення якості (Quality Assurance) програмного продукту. Основна інженерна мета цього документа полягає у систематизації та регламентації процесів верифікації розробленого веб-застосунку «Chatty». На основі цього плану здійснюється комплексна перевірка всіх архітектурних модулів та інтерфейсних рішень додатка для формування об'єктивного висновку про його готовність до фінального релізу (Production Deployment).

Головним завданням етапу тестування є експериментальне та документальне підтвердження того, що розроблений месенджер функціонує у суворій відповідності до висунутих функціональних і нефункціональних технічних вимог. Процес тестування повинен довести, що архітектура системи здатна підтримувати стабільний повнодуплексний обмін даними в реальному часі без критичних збоїв, витоків оперативної пам'яті (Memory Leaks) чи падіння серверної платформи Node.js під навантаженням. Окрім цього, важливим аспектом є підтвердження безпеки обробки та збереження конфіденційних користувацьких даних, захисту мережевих маршрутів і валідації вхідних файлових потоків перед їх фізичним збереженням у системі.

1.2. Вхідні дані

Проектування стратегії тестування, формування тестових сценаріїв та визначення критеріїв оцінки якості веб-застосунку «Chatty» базуються на чітко визначеному наборі вхідних даних та артефактів розробки. До переліку вхідних матеріалів, що використовуються у процесі тестування, належать:

- **Специфікація маршрутів REST API:** повний перелік кінцевих точок (Endpoints) сервера Express, конфігурація HTTP-методів (GET, POST, PUT, DELETE) для автентифікації користувачів, верифікації облікових записів, відновлення доступу та менеджменту профілів;
- **Конфігураційні матриці подій Socket.io:** детальний опис структури подієво-орієнтованої логіки реального часу, включаючи специфікації мережевих подій (sendMessage, message, typing, stopTyping), структуру об'єктів передачі даних (Payload) та алгоритми взаємодії віртуальних ізольованих кімнат (Rooms);

- **Логічна структура схем даних Mongoose:** описи моделей даних (User, Chat, Message) у форматі NoSQL для перевірки відповідності типів полів, обов'язкових атрибутів та коректності зв'язків між документами у СКБД MongoDB;
- **Вихідний програмний код клієнтської частини (Frontend):** архітектурна структура SPA-застосунку на базі HTML5, модулів стилізації CSS3 (включаючи кастомні змінні тем) та логічних модулів на чистому асинхронному JavaScript (ES6+), що відповідають за маніпуляції з DOM-деревом та клієнтську сокетну логіку.

1.3. Мета тестування

Процедура тестування веб-застосунку «Chatty» підпорядкована досягненню кількох стратегічних цілей, які гарантують надійність та конкурентоспроможність програмного продукту:

- **Верифікація абсолютної ізоляції приватних чатів:** експериментальна перевірка механізму віртуальних кімнат Socket.io для підтвердження того, що потоки даних між Користувачем А та Користувачем Б є повністю закритими від сторонніх підключених клієнтів, а трансляція подій здійснюється виключно в межах цільового ідентифікатора кімнати (chatId);
- **Вимірювання та оцінка швидкості доставки даних (Latency):** перевірка часових затримок при надсиланні та отриманні повідомлень, а також при спрацьовуванні індикаторів активності, з метою підтвердження відповідності параметрів real-time взаємодії встановленому нормативному діапазону (200–300 мс);
- **Контроль надійності захисту приватних маршрутів (Security Validation):** аудит безпеки REST API для підтвердження того, що доступ до конфіденційних даних та функцій месенджера надається виключно за наявності валідного безсесійного токена авторизації JSON Web Token (JWT) у заголовках запитів, а спроби несанкціонованого доступу коректно блокуються сервером;
- **Оцінка адаптивності та кросбраузерності користувацького інтерфейсу (UI/UX Testing):** перевірка стабільності та візуальної ідентичності відображення двопанельних сіток на моніторах персональних комп'ютерів і плавного переходу додатка в

однопанельний сенсорний режим на екранах смартфонів та планшетів під керуванням різних операційних систем.

2. Умови тестування

Процес забезпечення якості та верифікації програмного комплексу «Chatty» здійснюється у чітко визначених контрольованих умовах. Це дозволяє гарантувати об'єктивність, повторюваність та достовірність отриманих результатів тестування як у процесі розробки, так і на етапі фінального приймання продукту.

Для проведення всебічного аналізу поведінки системи розгорнуто два типи програмно-апаратних середовищ. Першим є локальне середовище розробника (Development Environment), де серверна платформа Node.js функціонує на базі операційної системи Windows. База даних для локальних тестів підключена у вигляді ізольованого хмарного кластера в MongoDB Atlas, а клієнтська частина запускається через локальний веб-сервер. Другим є виробниче хмарне середовище (Production Environment), створене для симуляції реальних умов експлуатації. У цьому середовищі серверний код Express та WebSocket-сервер Socket.io розгорнуті на хмарній платформі Render. Повна структура колекцій функціонує в основному хмарному кластері MongoDB Atlas, а клієнтська частина Single Page Application розгорнута на платформі Vercel, що забезпечує доставку статичного контенту через мережу CDN.

Тестування користувацького інтерфейсу десктопного веб-застосунку на предмет адаптивності та кросбраузерності проводилося на персональних комп'ютерах та ноутбуках під керуванням операційних систем Windows та macOS. Верифікація візуального відображення виконувалася у сучасних десктопних браузерах Google Chrome, Mozilla Firefox та Apple Safari. Оскільки додаток має адаптивну верстку, перевірка коректності відображення інтерфейсу при різних роздільних здатностях екрана здійснювалася безпосередньо на десктопних пристроях за допомогою вбудованого інструменту симуляції різноманітних моніторів та гаджетів у середовищі розробника Chrome DevTools (Device Mode).

Процедура тестування ініціюється лише за умови виконання чітких критеріїв початку робіт (Entry Criteria). До них належать повне завершення етапу написання вихідного програмного коду для всіх запланованих модулів сервера та інтерфейсу клієнта, успішна первинна збірка проєкту без

синтаксичних помилок, стабільне підключення хмарного кластера бази даних MongoDB Atlas та успішне виконання первинних міграцій схем Mongoose. Також обов'язковим є коректне налаштування файлу змінних оточення (.env) на хостинг-платформах, включаючи секретні ключі JWT та параметри SMTP-транспорту для Nodemailer.

Процес тестування вважається успішно завершеним, а програмний продукт «Chatty» готовим до демонстрації та релізу, за умови виконання критеріїв завершення (Exit Criteria). Ці критерії вимагають, щоб усі тестові випадки, закладені у цьому тест-плані, були виконані у повному обсязі. При цьому в системі повинні бути повністю відсутні не виправлені дефекти найвищих рівнів критичності, такі як Blocker (помилки, що викликають аварійне завершення роботи Node.js сервера або повне зависання інтерфейсу браузера) та Critical (помилки, що порушують безпеку, призводять до витоку JWT-токенів або порушують ізоляцію приватних кімнат Socket.io). Всі виявлені незначні косметичні дефекти або зауваження до UX-інтерфейсу мають бути зафіксовані та визнані такими, що не перешкоджають нормальній експлуатації веб-застосунку.

3. Стратегія процесу тестування

Стратегія тестування веб-застосунку «Chatty» базується на застосуванні методології «чорної скриньки» (Black-Box Testing). Цей підхід дозволяє виконувати всебічну верифікацію системи шляхом аналізу вхідних даних та очікуваних результатів роботи інтерфейсу й серверних маршрутів без необхідності прямого аналізу внутрішньої структури та покрокового виконання операцій в оперативній пам'яті комп'ютера. Основний акцент стратегії спрямований на симуляцію реальних користувацьких сценаріїв взаємодії з десктопним веб-інтерфейсом та контроль за поведінкою системи у відповідь на ці дії.

Процес тестування побудовано за ієрархічним принципом і розділено на два ключові рівні: тестування користувацького інтерфейсу та функціональну верифікацію модулів бізнес-логіки. Такий розподіл дозволяє ізольовано локалізувати помилки на рівні відображення або на рівні обробки запитів сервером, забезпечуючи системний контроль якості програмного продукту. Основна увага приділяється безшовності взаємодії всіх компонентів, безпеці обміну даними та стабільності підтримання постійного з'єднання, що є критично важливим для програмного забезпечення класу real-time.

3.1. Тестування інтерфейсу

Етап 1. Реєстрація нового облікового запису та активація

Первинна взаємодія із програмним комплексом починається з процедури створення профілю. Цей етап містить послідовний ланцюжок кроків, що забезпечують безпечний збір даних та дворівневу перевірку користувача, що детально проілюстровано на відповідних макетах нижче.

Форма реєстрації: На початковому екрані користувачеві пропонується заповнити первинні анкетні дані для створення картки в базі даних (рис. 1).

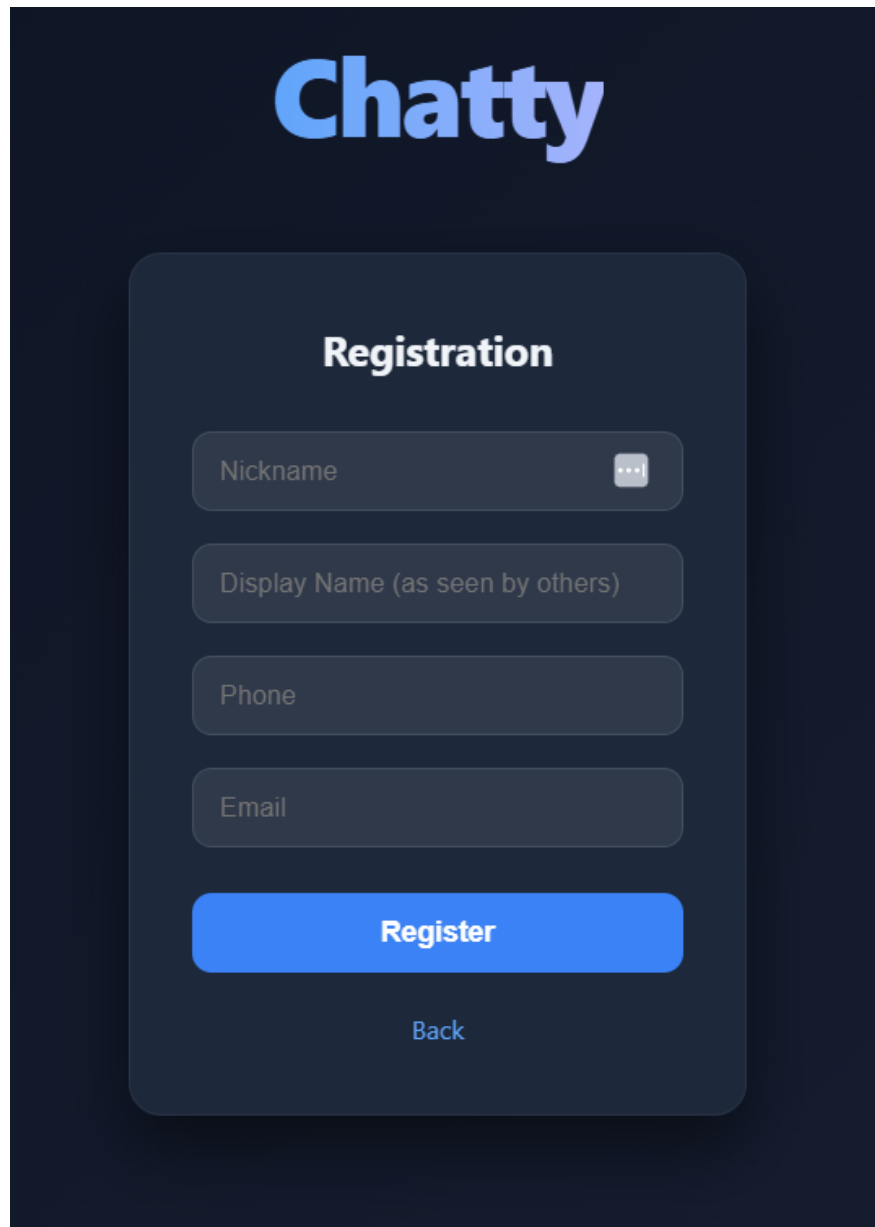
The image shows a mobile app registration screen for 'Chatty'. At the top, the word 'Chatty' is written in a large, blue, stylized font. Below it, the title 'Registration' is centered in a white, bold font. The form consists of four input fields stacked vertically: 'Nickname' (with a speech bubble icon), 'Display Name (as seen by others)', 'Phone', and 'Email'. All fields are currently empty. At the bottom of the form, there is a prominent blue button labeled 'Register' and a smaller, lighter blue link labeled 'Back'.

Рис. 1. Форма реєстрації нового користувача.

Модуль підтвердження (Confirmation): Після успішного надсилання реєстраційної форми інтерфейс динамічно перемикається на екран введення одноразового ключа активації, блокуючи прямий доступ до системи до моменту підтвердження email (рис. 2).

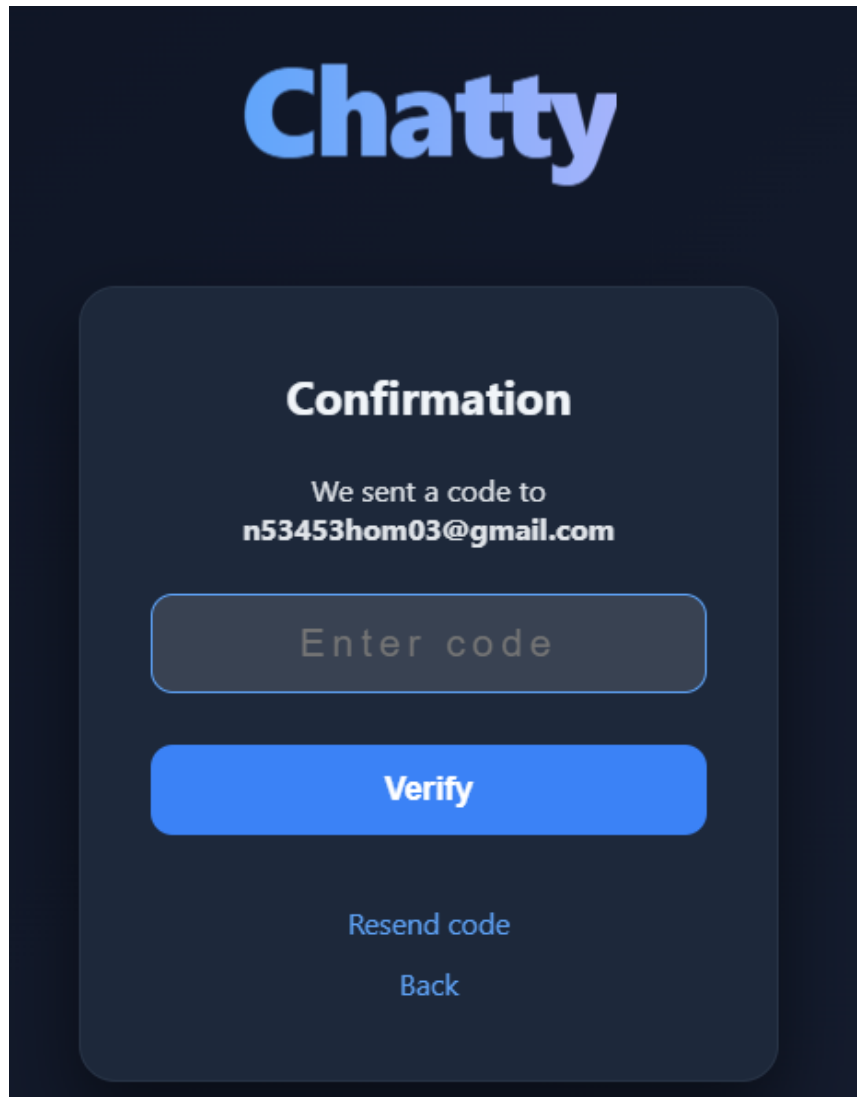


Рис. 2. Форма підтвердження реєстрації (Confirmation).

Генерація та доставка коду: Прикладом успішної роботи інтегрованих поштових модулів на цьому етапі є отримання користувачем автоматичного транзакційного листа з унікальним цифровим токеном на електронну поштову скриньку (рис. 3).



Рис. 3. Системний лист із кодом підтвердження реєстрації.

Створення пароля: У разі успішного введення коду верифікації, додаток автоматично перенаправляє користувача на фінальний крок ініціалізації профілю — інтерфейс встановлення постійного пароля доступу (рис. 4).

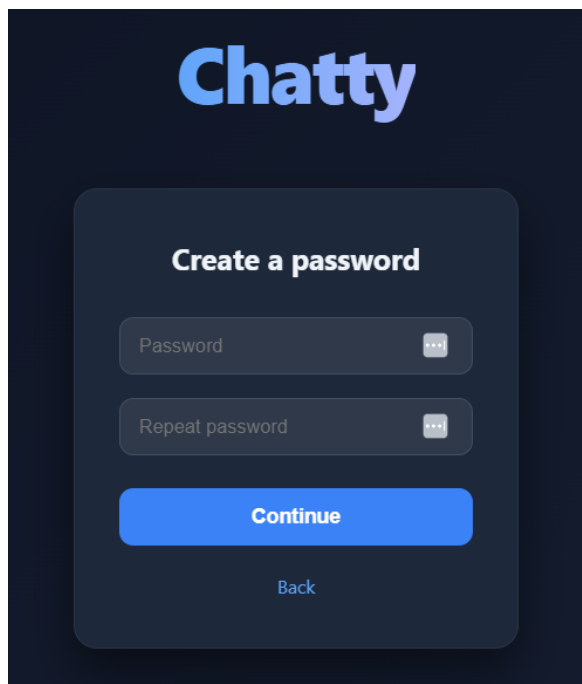


Рис. 4. Екран створення пароля (Create a password).

Етап 2. Автентифікація та повний сценарій відновлення доступу (твій новий блок)

Цей етап тестування інтерфейсу присвячений перевірці логіки входу вже зареєстрованих користувачів, а також комплексному аналізу поведінки графічної оболонки у випадку виникнення помилок авторизації. Особлива увага приділяється безшовності та безпеці процесу відновлення контролю над обліковим записом, що є критично важливим для забезпечення високого рівня User Experience (UX).

Стандартна форма авторизації: Для входу в систему користувачеві пред'являється лаконічна форма, що містить поля для введення ідентифікатора (Email) та пароля (рис. 5). Інтерфейс підтримує збереження фокусу на полях, динамічну підсвітку активного введення та інтерактивну кнопку «Log In».

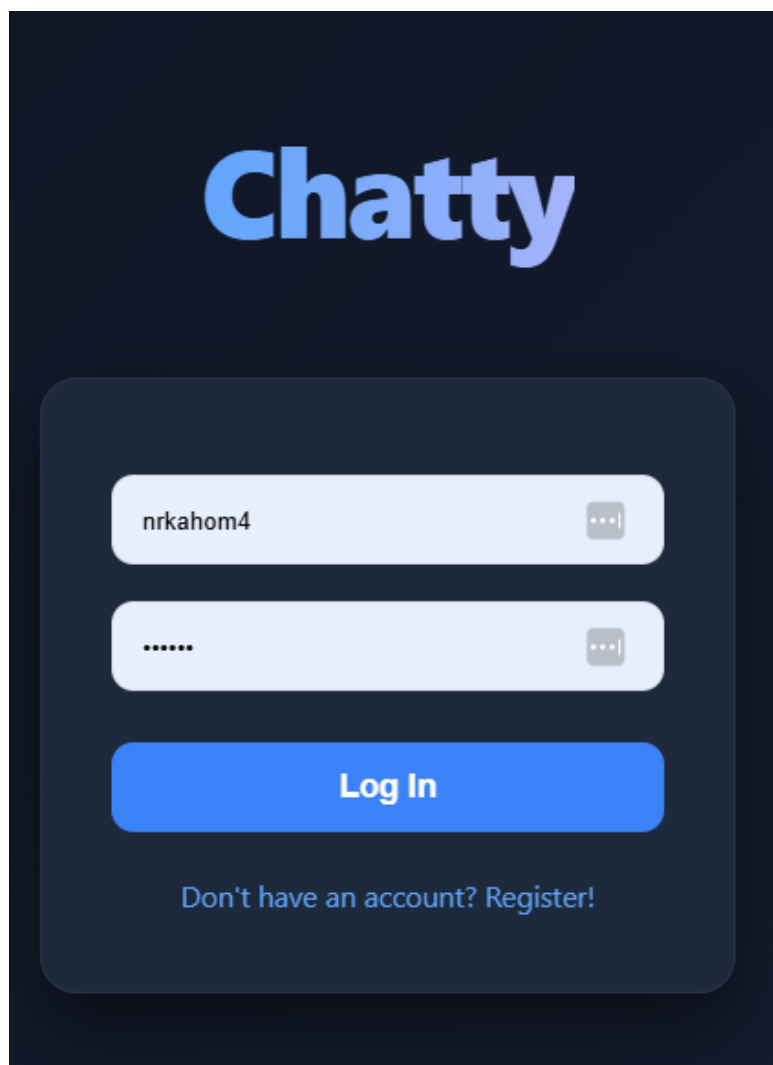
The image shows a login interface for a system named "Chatty". The word "Chatty" is displayed in a large, bold, blue font at the top. Below it, there is a dark blue rounded rectangle containing the login form. The form consists of two light blue input fields. The first field contains the text "nrkahom4" and has a small icon of three dots and a vertical line on its right side. The second field contains six dots "....." and also has the same icon on its right side. Below the input fields is a solid blue button with the text "Log In" in white. At the bottom of the form area, there is a link that says "Don't have an account? Register!" in a smaller, lighter blue font.

Рис. 5. Форма входу в систему (Log In) при введенні валідних даних.

Реакція на помилки введення: У випадку, якщо користувач припускається помилки у комбінації логіна чи пароля, інтерфейс не просто блокує вхід, а динамічно трансформується (рис. 6). На екрані з'являється чітке текстове попередження про автентифікаційну помилку, а під полем введення пароля миттєво генерується нове інтерактивне посилання «Забули пароль?». Це дозволяє користувачеві відразу перейти до альтернативного сценарію, не залишаючи поточного вікна.

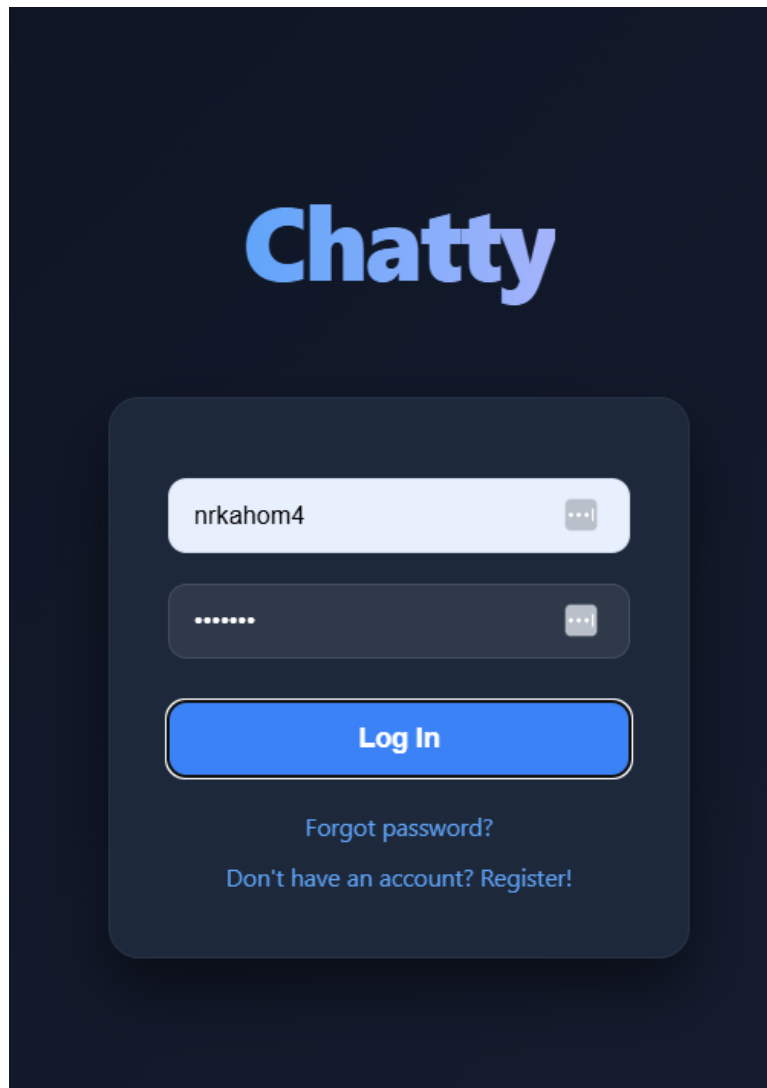


Рис. 6. Відображення помилки при введенні невірного пароля та поява інтерактивного посилання «Забули пароль?».

Ініціалізація процедури скидання пароля: При натисканні на посилання «Забули пароль?» SPA-додаток без перезавантаження сторінки викликає форму запити на відновлення доступу (рис. 7). Тут користувачеві необхідно вказати лише електронну пошту, до якої прив'язано аккаунт. Форма містить

вбудовану клієнтську валідацію на коректність структури адреси (наявність символу @ та доменної зони).

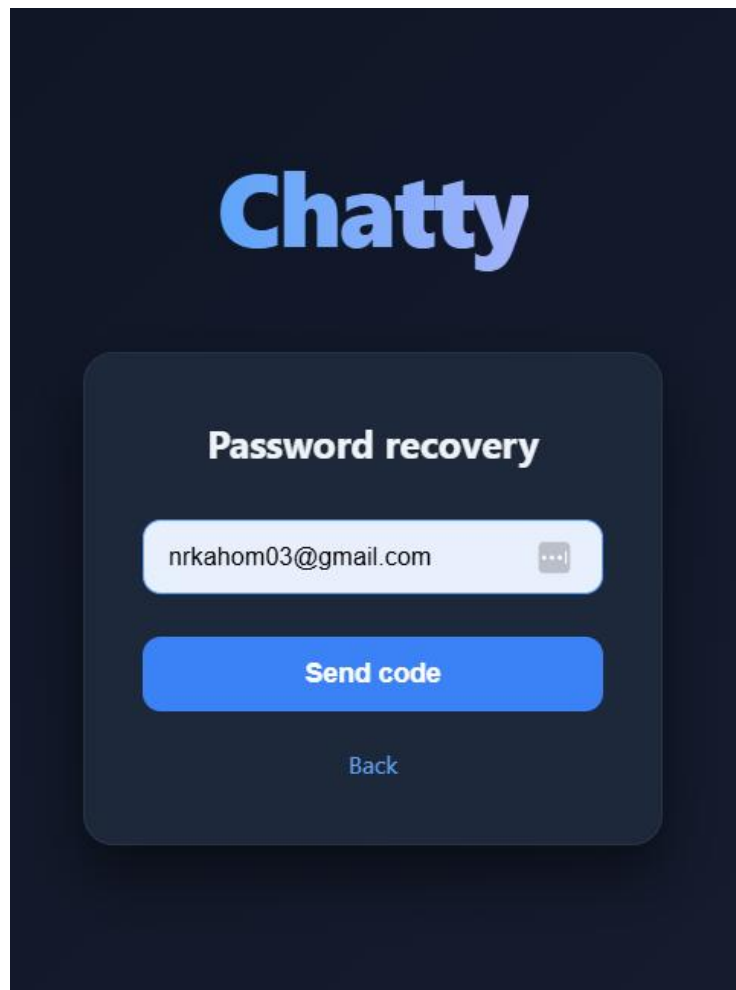


Рис. 7. Форма запиту на відновлення пароля із полем введення електронної пошти аккаунта.

Доставка транзакційного токена відновлення: Після успішного надсилання запиту на сервер, поштова підсистема месенджера автоматично надсилає на email користувача службове повідомлення із темою «Запит на відновлення пароля» (рис. 8). Лист містить унікальний згенерований шестизначний код підтвердження, який має обмежений сервером час дії (TTL), що захищає систему від перехоплення доступу третіми особами.

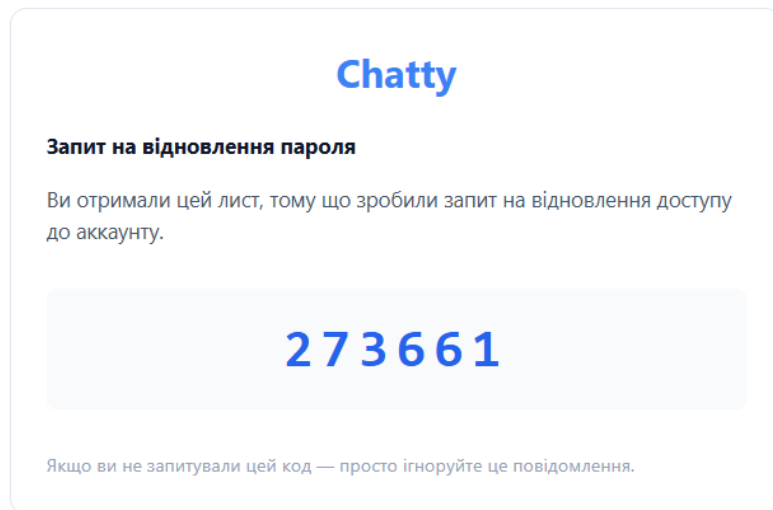


Рис. 8. Системний лист на електронній пошті з кодом підтвердження зміни пароля.

Верифікація запиту на відновлення: Отримавши цифровий код із листа, користувач вводить його у спеціалізоване поле форми підтвердження скидання пароля (рис. 9). Інтерфейс візуально маскує введення або обмежує його лише цифровими символами, запобігаючи помилкам друку, та перевіряє статус відповіді від сервера в реальному часі.

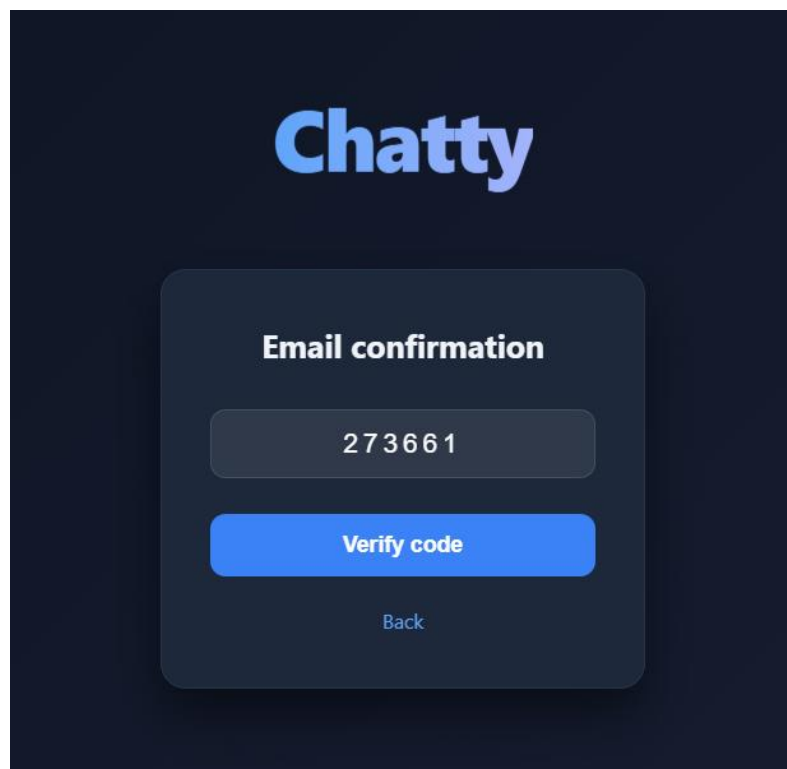
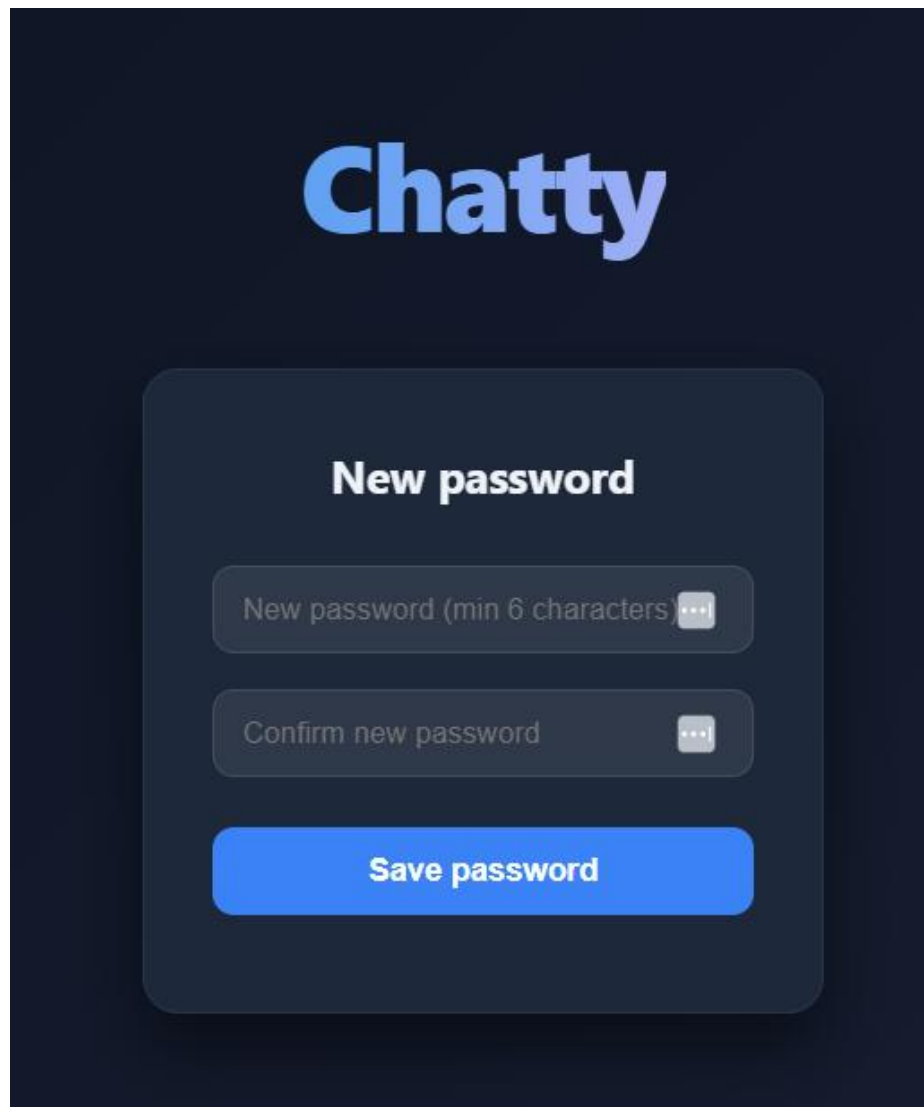


Рис. 9. Форма введення отриманого коду верифікації для скидання пароля.

Форма встановлення нових облікових даних: Фінальним кроком сценарію є перенаправлення користувача на форму зміни пароля (рис. 10). Екран містить два поля: для введення нового пароля та його обов'язкового повторення. Клієнтський скрипт порівнює ці два рядки «на льоту» і активує фінальну кнопку відправки лише за умови їх повної ідентичності, що повністю виключає випадкове встановлення помилкового пароля.



The image shows a mobile application interface for setting a new password. The background is a dark navy blue. At the top center is the word "Chatty" in a light blue, rounded font. Below it, centered, is a dark blue rounded rectangle containing the form. Inside this rectangle, the text "New password" is centered in white. Below this title are two input fields, each with a light gray border and rounded corners. The first field contains the placeholder text "New password (min 6 characters)" and a small gray icon with three dots on the right. The second field contains the placeholder text "Confirm new password" and a similar icon. Below these two fields is a solid blue button with the text "Save password" in white, centered.

Рис. 10. Форма встановлення нового пароля (смени пароля) для облікового запису.

Етап 3. Користувацький UX/UI-інтерфейс

Цей етап випробувань та демонстрації присвячений аналізу ергономіки, композиційної структури та логіки навігації головного робочого простору веб-застосунку «Chatty». Оскільки додаток функціонує в межах Single Page Application (SPA), тестування інтерфейсу дозволяє підтвердити безшовність рендерингу складних багаторівневих вікон, модальних панелей та меню користувача без необхідності оновлення сторінки десктопного браузера, що забезпечує високу швидкість відгуку інтерфейсу на дії оператора.

Архітектура головного вікна в режимі очікування: Після успішного проходження циклів автентифікації або процедури відновлення облікових даних, користувач перенаправляється на центральний робочий екран додатка (рис. 11). Візуальний каркас інтерфейсу спроектовано за допомогою модулів трипанельної сітки CSS Grid та CSS Flexbox.

Ліва панель відведена під блок глобальної навігації та стрічку доступних чатів, яка в початковому стані є пустою. Центральна область, що займає найбільшу корисну площу монітора, виступає головним вікном повідомлень і в режимі очікування відображає дефолтний інформаційний заглушок (Placeholder) із чітким текстовим орієнтиром для користувача: «Оберіть чат. Виберіть колію/діалог, щоб почати розмову». Права бічна панель зарезервована під виведення контекстної інформації. Таке композиційне компонування дозволяє розвантажити візуальне сприйняття, уникнути хаотичного нагромадження елементів та акцентувати увагу на базових тригерах початку діалогу.

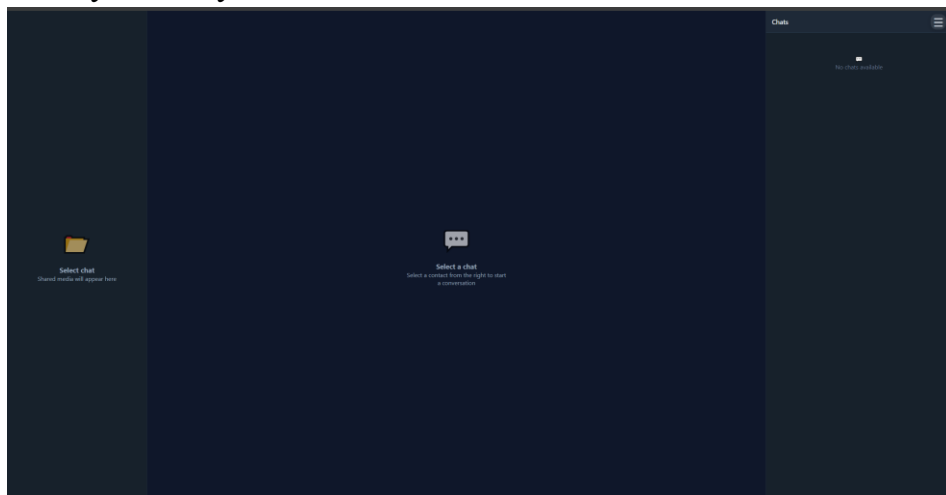


Рис. 11. Головне вікно застосунку в режимі очікування вибору чату та колії спілкування.

Глобальне меню навігації: Управління основними функціональними розділами месенджера реалізовано через виклик інтерактивного головного меню (рис. 12). Воно активується натисканням на кастомну графічну кнопку-іконку (бургер-меню), розташовану у верхній частині інтерфейсу.

Завдяки JavaScript-маніпуляціям з DOM-деревом, меню динамічно випадає поверх основних шарів додатка, надаючи користувачеві структурований перелік доступних операцій: перехід до редагування особистих даних (My Profile), запуск менеджера друзів (Contacts), відкриття глобальних параметрів системи (Settings) та ініціалізація безпечного виходу з поточного сеансу (Log Out). Кожен пункт меню має чітке текстове позначення, анімацію зміни стану при наведенні курсора (Hover State) та безпомилково ініціює відповідний програмний роутинг.

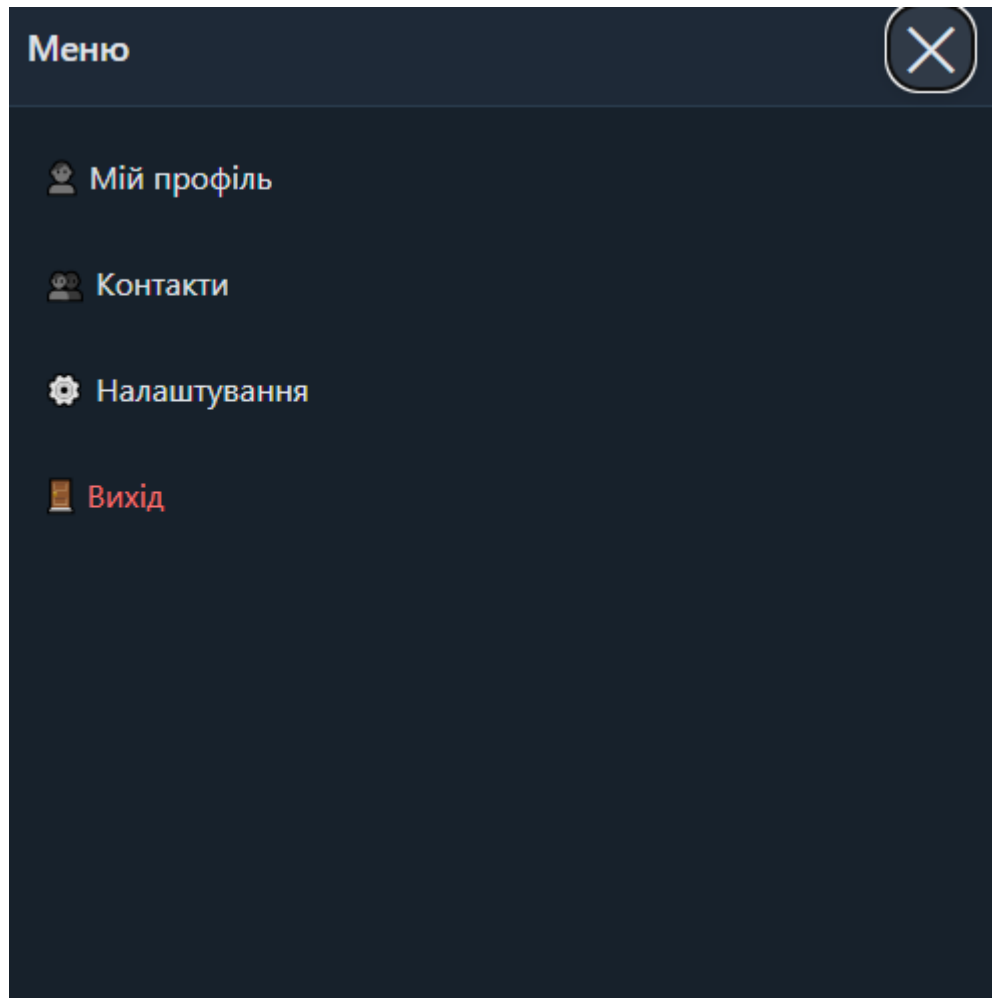


Рис. 12. Візуальне відображення пунктів головного меню навігації месенджера.

Менеджмент особистого профілю (My Profile): При виборі пункту меню «My Profile» додаток відображає спеціалізовану вкладку керування обліковим записом (рис. 13). Графічний інтерфейс цієї панелі містить круглий контейнер-маску для відображення аватара користувача, текстові блоки системного нікнейму із фіксованим префіксом @ (наприклад, @ernest), відображуваного імені (Display Name) та окреме кастомне поле для введення біографічних даних або статусу користувача («Bio»). На цьому екрані тестується поведінка полів редагування: інтерфейс дозволяє користувачеві гнучко оновлювати інформацію про себе (наприклад, текстовий маркер «я тестировщик с 30 летним стажем»), забезпечуючи наочність процесу персоналізації акаунта.

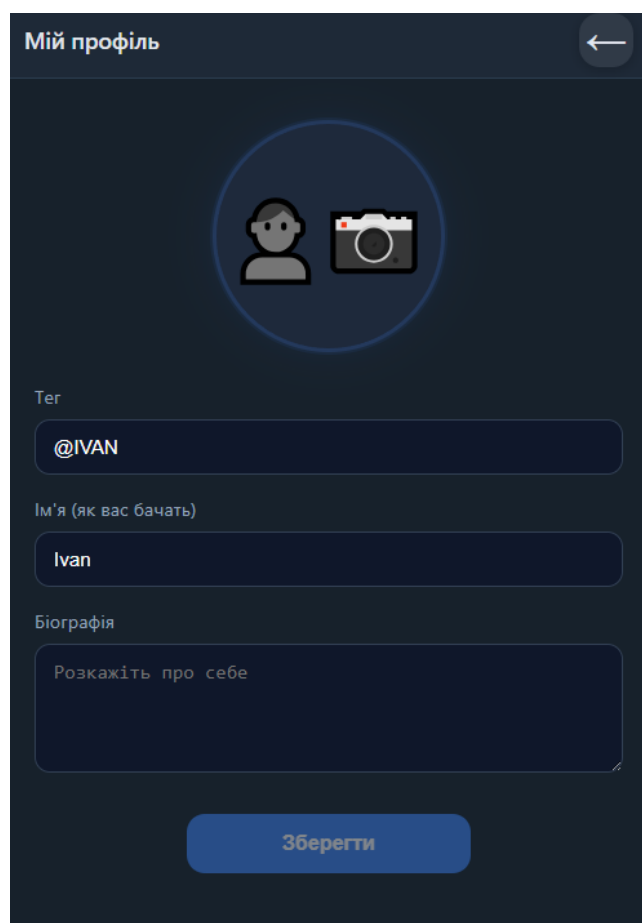


Рис. 13. Інтерфейс вкладки «My Profile» (Мій профіль) із переглядом біографії користувача.

Взаємодія з картками інших користувачів: Інтерфейсна логіка картки профілю має динамічний характер і кардинально змінює набір елементів керування, якщо поточний користувач переглядає профілі інших учасників мережі (рис. 14). У такому разі замість кнопок редагування власних даних

інтерфейс генерує контекстно-залежні функціональні тригери управління зв'язками.

Якщо користувач ще не перебуває у списку друзів оператора, на екрані з'являється кнопка «Додати до контактів». Якщо ж зв'язок уже встановлено у базі даних, інтерфейс миттєво підставляє альтернативну кнопку «Видалити з контактів» та кнопку ініціалізації прямого діалогу, що демонструє високу адаптивність логіки інтерфейсу та правильну обробку станів взаємовідносин між користувачами.

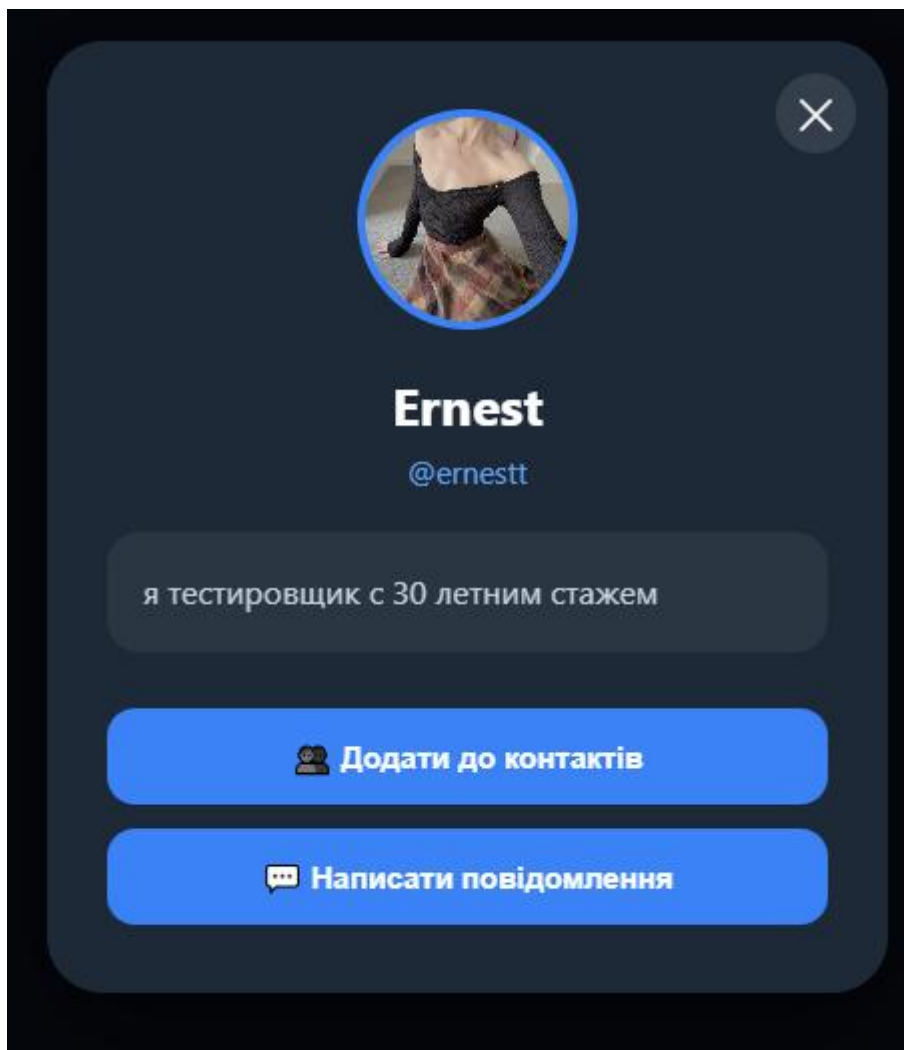


Рис. 14. Візуальний вигляд картки профілю при взаємодії з іншими користувачами (кнопки додавання/видалення з контактів).

Модуль управління списками контактів та пошуку: Натискання на пункт меню «Contacts» активує спеціалізований інтерфейс пошуку та групування контактів (рис. 15). Візуальний простір цього модуля містить інтерактивну пошукову строку із вбудованою іконкою лупи.

Головною UX-особливістю цього вікна є чітке візуальне та логічне розділення виведених результатів пошуку на дві незалежні категорії: *YOUR CONTACTS* (користувачі, з якими вже встановлено дружній зв'язок) та *OTHER USERS* (інші користувачі системи, знайдені за глобальним запитом). Кожна знайдена картка містить аватар, відображуване ім'я та нікнейм, що дозволяє користувачеві швидко орієнтуватися у структурі результатів пошуку під час введення перших літер нікнейму (наприклад, @er).

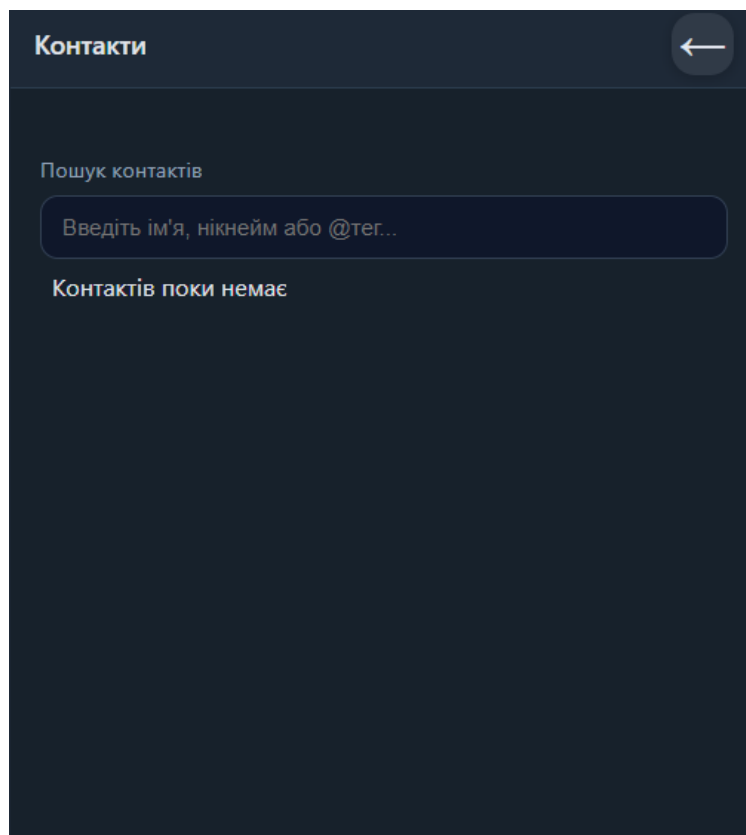


Рис. 15. Інтерфейс модуля «Contacts» (Контакти) у режимі динамічного пошуку користувачів.

Панель системних конфігурацій (Settings): Візуальне відображення панелі налаштувань «Settings» відповідає за індивідуальну адаптацію робочого простору месенджера під запити конкретного оператора (рис. 16). Графічна структура вікна містить два основні елементи управління:

Двопозиційний графічний перемикач (Toggle Switch) для кастомізації колірної теми додатка (Theme), який дозволяє в один клік переходити між світлим та темним оформленням.

Випадаючий список (Dropdown Menu) для вибору активної мовної локалі (Language), що підтримує миттєве перемикання текстових елементів інтерфейсу між українською та англійською мовами.

Нижче у структурі блоку інтегровано навігаційну кнопку «Change password», яка відкриває доступ до інструментів безпеки облікового запису.

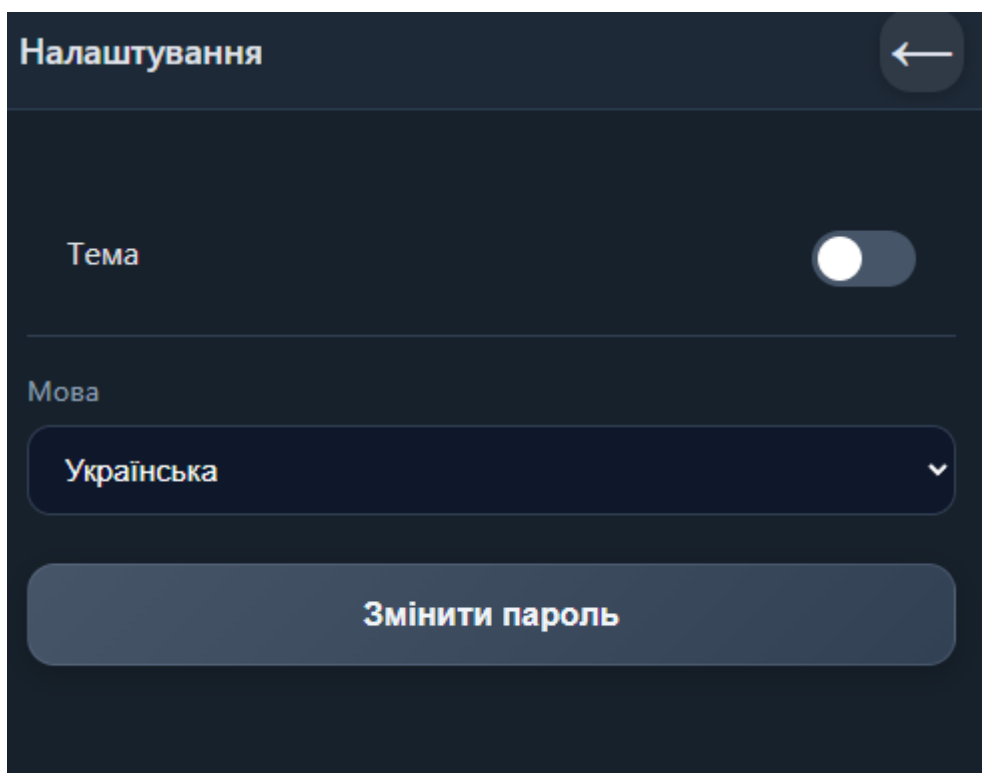


Рис. 16. Панель системних налаштувань «Settings» (вибір мови інтерфейсу та кастомізація колірної теми).

3.1.2. Функціональне тестування

Функціональне тестування призначене для експериментальної перевірки відповідності розроблених програмних модулів та алгоритмів вимогам технічного завдання. Цей етап охоплює контроль за правильністю обробки HTTP-запитів до REST API, верифікацію взаємодії з NoSQL базою даних MongoDB через ODM Mongoose, аудит криптографічного захисту, а також аналіз подієво-орієнтованої архітектури реального часу на базі протоколу WebSocket (Socket.io).

Нижче наведено покроковий опис результатів тестування наскрізних функціональних сценаріїв, зафіксованих під час випробувань системи.

Етап 1. Тестування модулів автентифікації, реєстрації та керування сесіями доступу

Редагування профілю та персистентність даних: У процесі тестування форми зміни персональних даних (рис. 12) перевірялася працездатність REST API при надсиланні HTTP-запитів типу PUT. При оновленні відображуваного імені, завантаженні нового аватара через завантажувач файлів Multer або редагуванні біографічного статусу користувача, сервер Express виконує делікатну валідацію рядків та за допомогою методу `findOneAndUpdate()` драйвера Mongoose миттєво перезаписує документ у колекції `users`. Тести підтвердили, що дані успішно зберігаються в MongoDB і динамічно оновлюються в інтерфейсі без перезапуску SPA.

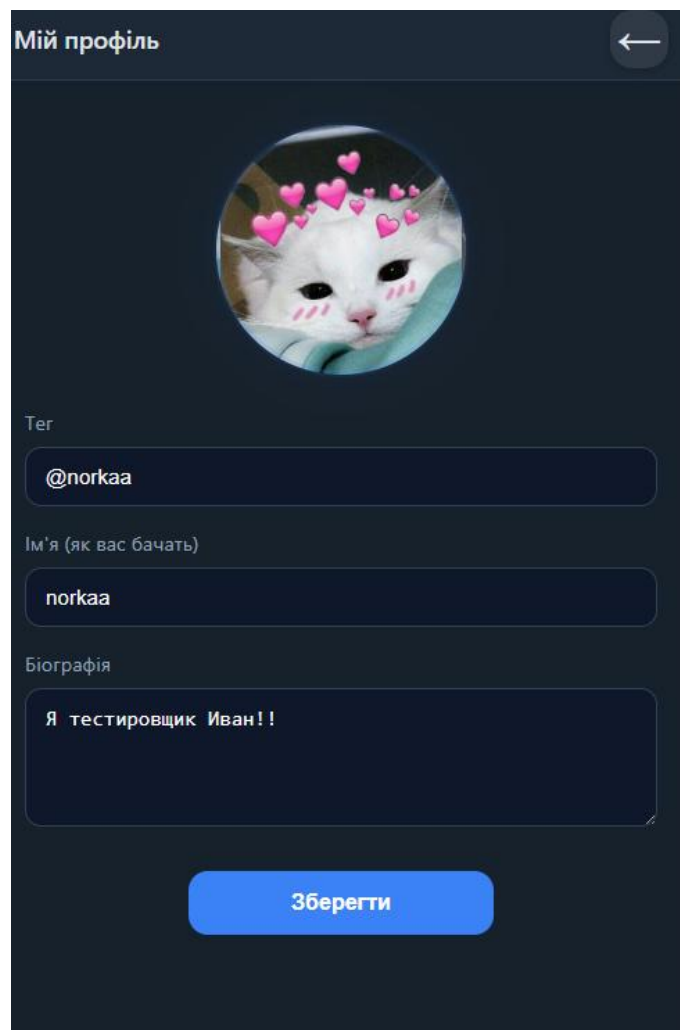


Рис. 12. Форма редагування профілю, зміни персональних даних та завантаження аватара.

Ініціалізація модуля контактів: При переході користувача до менеджменту френд-ліста додаток демонструє базовий інтерфейс модуля «Контакти» (рис. 13). На цьому кроці верифікується стан системи до моменту надсилання перших пошукових запитів. Екран перебуває в режимі очікування введення

даних (Event Listener на полі Input) та відображає поточні списки вже наявних друзів користувача, підтягнуті з бази даних за допомогою оператора \$in.

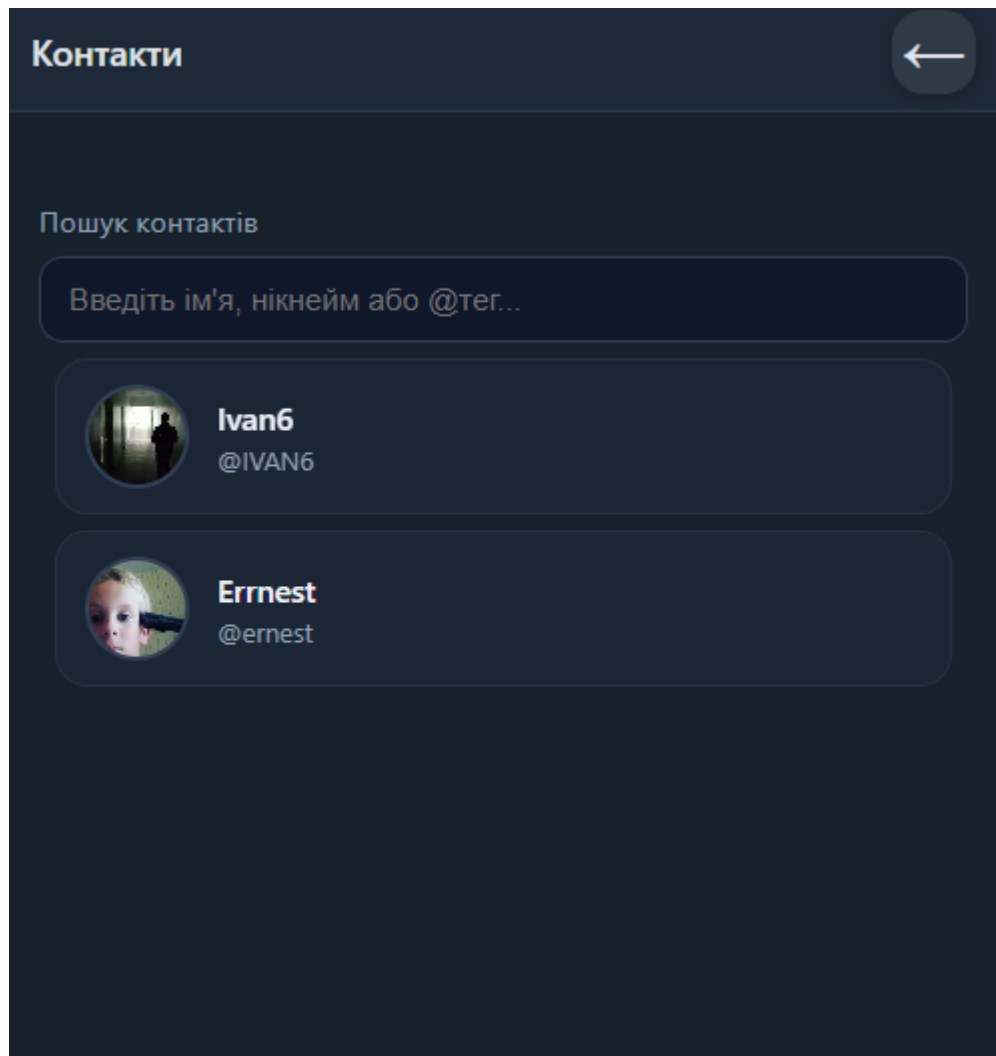


Рис.. 13. Загальний інтерфейс модуля «Контакти» на початок процесу пошуку.

Динамічний пошук у реальному часі: Функціональний сценарій пошуку (рис. 14) базується на обробці подій введення тексту. Під час тестування введення символів (наприклад, перших літер нікнейму) клієнтська частина надсилає асинхронний GET-запрос до API. На рівні бекенду контролер виконує вибірку документів за допомогою регулярних виразів Mongoose (\$regex) із прапорцем регістронезалежності і. Результати миттєво повертаються на фронтенд та динамічно відмальовуються, розділяючи користувачів на категорії наявних контактів та глобальних результатів системи.

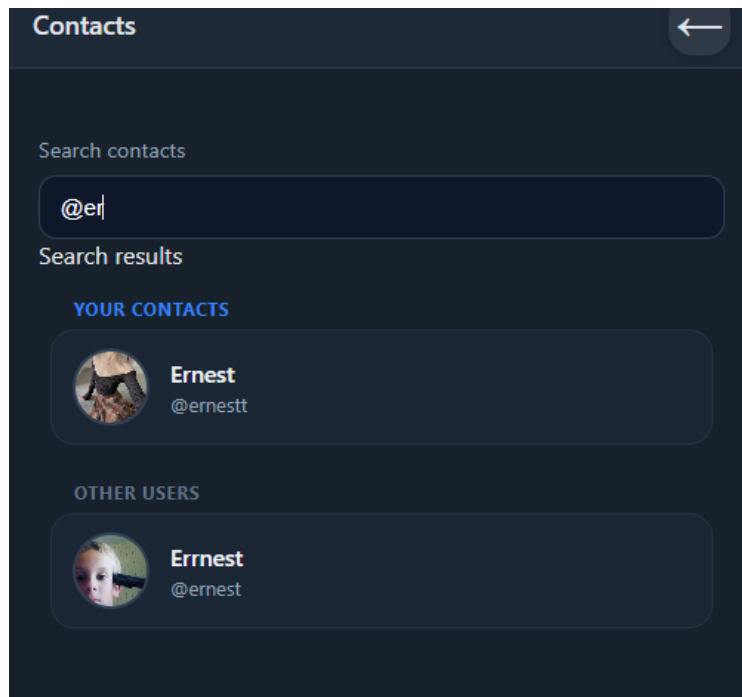


Рис. 14. Динамічний пошук користувачів у реальному часі за нікнеймом.

Менеджмент зв'язків (Додавання та видалення друзів): При взаємодії з картою профілю обраного користувача (рис. 15) тестувалася логіка контролерів управління масивами даних. Натискання кнопки «Додати до контактів» ініціює POST-запит, який через метод `$push` записує ідентифікатор (`_id`) нового друга до масиву контактів поточного користувача в базі даних. Аналогічно, при натисканні кнопки вилучення, метод `$pull` драйвера Mongoose коректно видаляє зв'язок. Зміна станів кнопок в інтерфейсі відбувається миттєво і безшовно завдяки реактивному оновленню стану (State Management) клієнтського додатку.

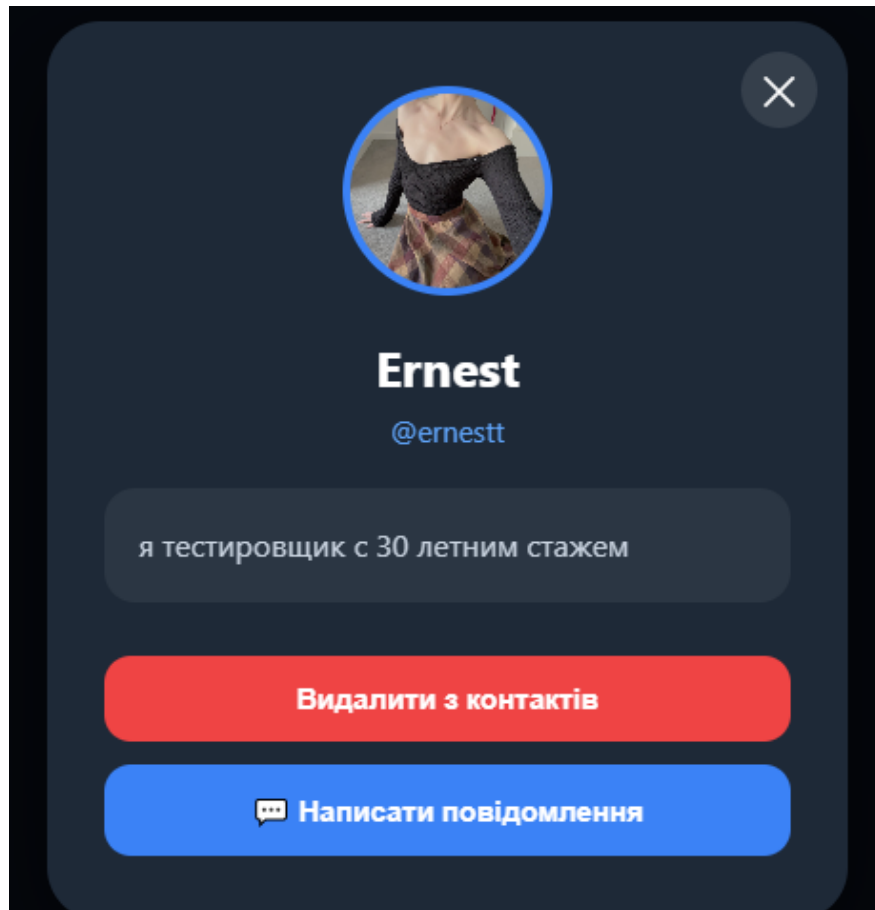


Рис. 15. Модальне вікно профілю контакту з функцією додавання/видалення із френд-листа.

Автоматичне завантаження історії чату: Важливим елементом тестування синхронізації даних є процес відкриття конкретної кімнати діалогу (рис. 16). При активації чату клієнт відправляє запит на ендпоінт `/api/messages/:chatId`. Функціональний алгоритм сервера зчитує з MongoDB стек повідомлень, сортує їх за часовим індексом `createdAt` та повертає масив об'єктів на клієнт. Скрипт фронтенду циклічно рендерить повідомлення в DOM-дерево за допомогою `appendChild()` і автоматично викликає метод `scrollIntoView()`, фокусуючи екран користувача на останніх репліках розмови.

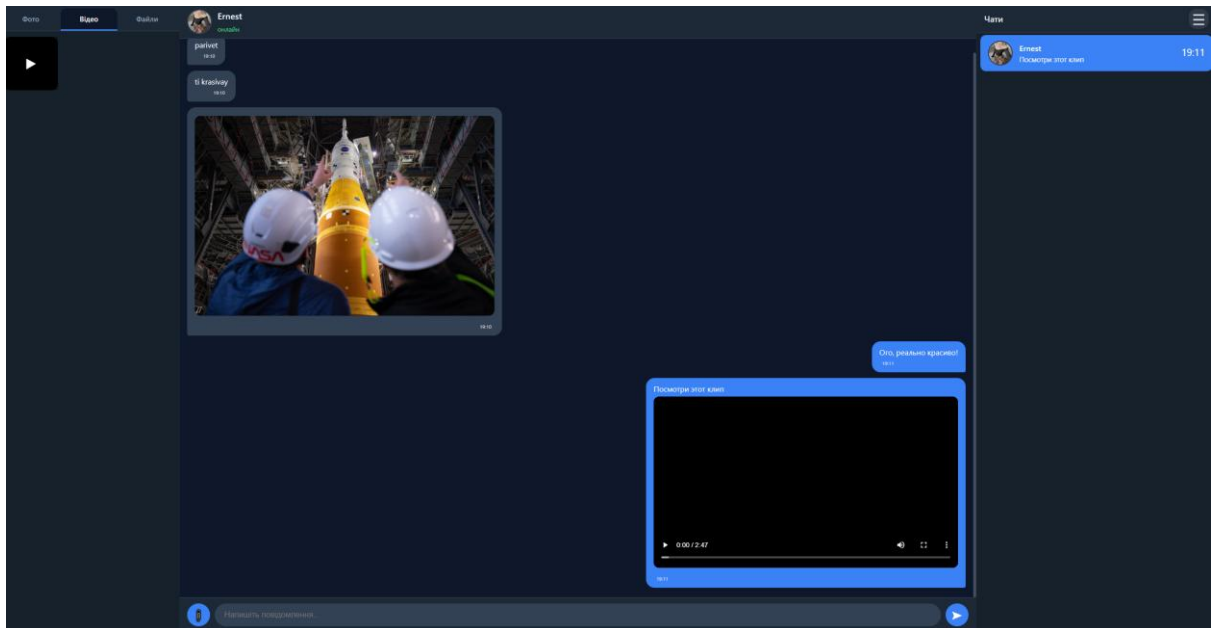


Рис. 16. Процес автоматичного завантаження та рендерингу історії повідомлень під час відкриття чату.

Авторизація змін пароля (Валідація поточних даних): Для захисту конфіденційних даних під час активної сесії користувача, процедура зміни пароля розділена на два функціональні кроки. На першому етапі (рис. 17) додаток виводить модальне вікно запиту діючого пароля. Сервер Express приймає введені дані та виконує криптографічне порівняння за допомогою `bcrypt.compare()` із захешованим рядком, що зберігається в колекції `users` бази даних. Наступний крок стає доступним лише за умови повернення сервером статусу успішного збігу.

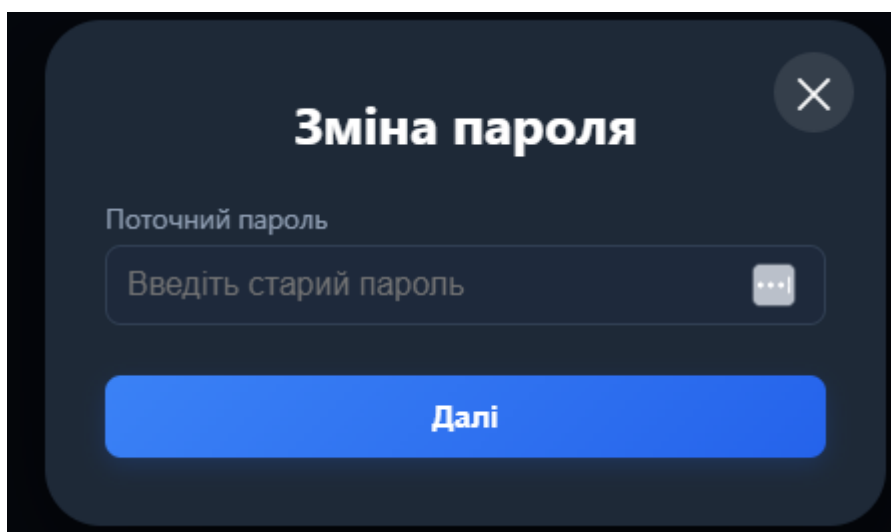


Рис. 17. Модальне вікно запиту поточного пароля користувача для авторизації змін.

Встановлення та шифрування нового пароля: Після успішного підтвердження повноважень інтерфейс відкриває модальну форму для введення нових облікових даних (рис. 18). Функціональний клієнтський скрипт порівнює вміст полів «Новий пароль» та «Повторіть новий пароль» і блокує відправку форми у разі їх розбіжності. Після успішного надсилання валідних даних сервер Express генерує нову сіль (salt), виконує одноразове безпечне хешування пароля через бібліотеку bcryptjs (що виключає баг подвійного хешування) та оновлює документ користувача в MongoDB.

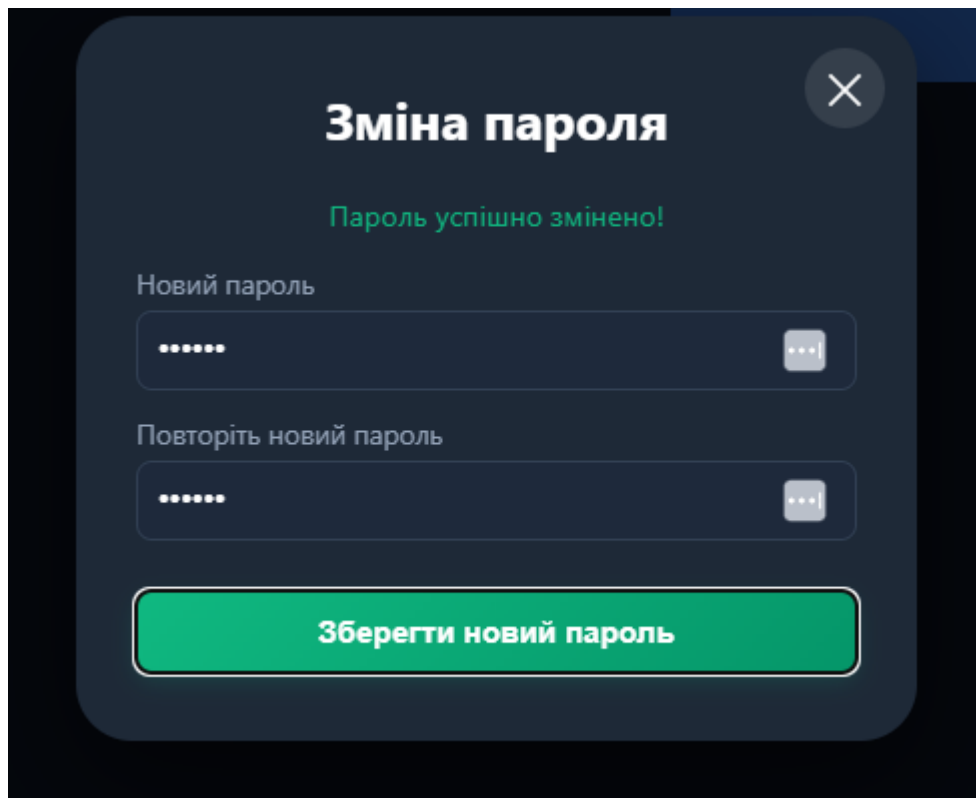


Рис. 18. Модальне вікно встановлення та повторення нового пароля

Етап 4. Тестування підсистеми обробки медіафайлів, потокового контенту та кастомізації інтерфейсу

Прикінцевий етап функціональних випробувань спрямований на перевірку стабільності завантаження, обробки, фільтрації та відтворення різноманітних типів бінарних даних (медіафайлів), а також на контроль за поведінкою системних модулів інтернаціоналізації (i18n) та кастомізації графічної оболонки Single Page Application.

Завантаження та обробка медіафайлів (Фото/Зображення):

Функціональний процес надсилення графічних об'єктів у реальному часі (наприклад, фотографій космічної ракети, зображень тварин тощо) поєднує в собі роботу REST API та WebSocket-сервера. Коли користувач додає зображення через інтерфейс (див. **Рис. 19**), клієнтська частина формує об'єкт FormData та відправляє його за допомогою POST-запиту на сервер.

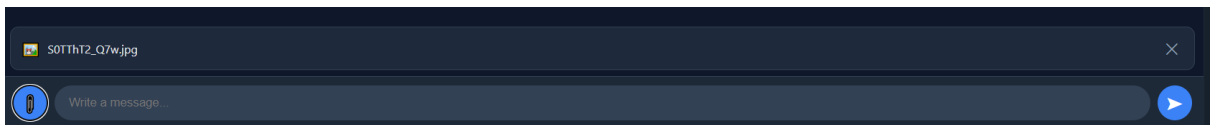


Рис. 19. Інтерфейс вибору графічного файлу (зображення) та активація вікна попереднього перегляду (File Preview).

Серверний middleware-модуль **Multer** здійснює жорстку перевірку вхідного потоку даних: контролюється максимальний розмір файлу (встановлено інженерний ліміт у 10 МБ) та розширення файлу через аналіз MIME-типу (дозволено лише формати графічних зображень .png, .jpg, .jpeg, .gif). У разі успішної валідації файл зберігається на сервері, а згенероване URL-посилання записується в колекцію messages MongoDB та миттєво ретранслюється через сокет-подію message всім учасникам поточної кімнати чату, динамічно відмальовуючись у стрічці повідомлень (див. **Рис. 20**).

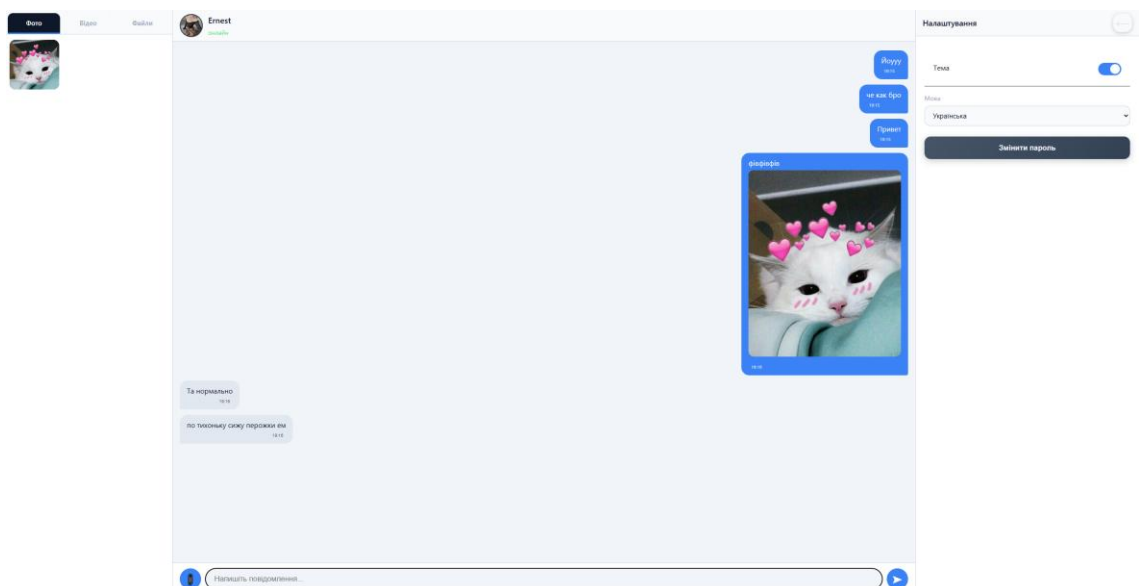


Рис. 20. Демонстрація успішного надсилання та відображення медіафайлів (графічних зображень) у стрічці повідомлень чату.

Інтеграція та функціонування вбудованого відеоплеєра: Під час тестування передачі відеофайлів перевірялася здатність системи обробляти великі бінарні об'єкти та інтегрувати їх у користувацький інтерфейс без порушення архітектури SPA. При виявленні у повідомленні медіафайлу з MIME-типом `video/*`, клієнтський JavaScript замість звичайного статичного посилання ініціює рендеринг вбудованого інтерактивного медіаплеєра за допомогою HTML5 тегу `<video>` (див. **Рис.21**).

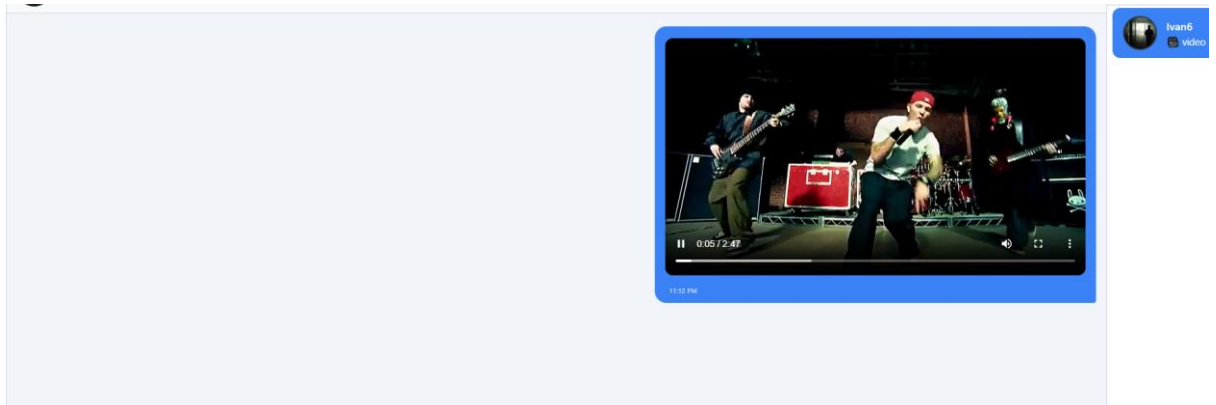


Рис. 21. Інтеграція вбудованого HTML5 відеоплеєра в активному діалозі месенджера.

Компонент містить кастомну панель керування (кнопки Play/Pause, лінійка таймлайну прогресу відтворення, регулятор гучності та лічильник тривалості медіапотоків). Це дозволяє користувачам безперешкодно переглядати відеоконтент (наприклад, тривалістю 2:47 хв) безпосередньо всередині активного вікна діалогу месенджера, знижуючи навантаження на дискову підсистему десктопного пристрою користувача.

Робота панелі сортування та фільтрації вкладень (Media Tabs): Для підвищення ергономіки взаємодії в системі реалізовано підсистему каталогізації надісланих у чат матеріалів, яка відображається у верхній або бічній частині активного вікна розмови (див. **Рис. 22**). Функціонал базується на клієнтських фільтрах масивів даних. При переході користувача між вкладками Photos (Фотографії), Videos (Відеоролики) або Files (Документи), JavaScript сканує поточний стек повідомлень у пам'яті

браузера, аналізує поле fileType або MIME-тип кожного вкладення і миттєво групує файли за відповідними категоріями. Це забезпечує користувачеві швидкий та структурований доступ до всієї історії медіаобміну в конкретній кімнаті чату без необхідності ручного гортання всієї стрічки діалогу.

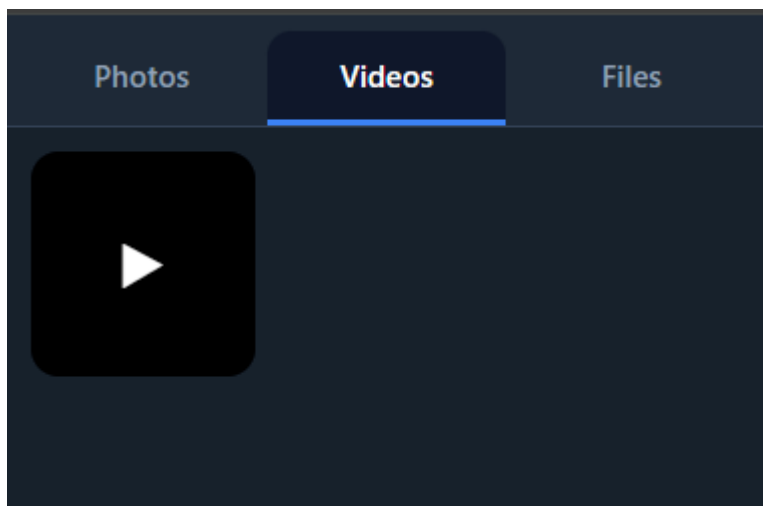


Рис. 22. Робота панелі категорій (Вкладки Photos / Videos / Files) для швидкого пошуку надісланого медіаконтенту.

Динамічне перемикання колірних схем (Теми інтерфейсу): Функціонал кастомізації візуальної оболонки тестувався в налаштуваннях системи (див. **Рис. 23**). Механізм перемикання між світлою (Light Mode) та темною (Dark Mode) темами базується на подіях DOM-дерева. При активації графічного прапорця (Toggle) клієнтський скрипт змінює значення глобального атрибута data-theme тегу <html>. CSS-процесор додатка миттєво реагує на цю зміну, перепризначаючи весь набір системних колірних змінних (колір фону, текстових блоків, карток чату). Для забезпечення персистентності (постійності) даних, поточний стан теми автоматично записується в локальне сховище браузера localStorage. Під час повторного тестування (примусове оновлення сторінки клавішею F5 або перезапуск сесії браузера) підтверджено, що додаток коректно зчитує записаний маркер і запускається в обраній користувачем колірній схемі.

Валідація модуля інтернаціоналізації (i18n): Тестування системи мультязичності (локалізації) підтвердило гнучкість архітектури додатка при зміні мовних параметрів (перемикання між мовними пакетами

Українська та English в меню налаштувань). При виборі відповідного пункту у випадаючому списку (див. **Рис. 23**), модуль інтернаціоналізації динамічно заміщує текстові константи (лейбли, плейсхолдери, назви меню) у всьому DOM-дереві додатка згідно з обраним JSON-файлом локалі. Перемикання мовного інтерфейсу відбувається миттєво і не потребує повторного звернення до хмарного сервера Express, що значно економить мережевий трафік десктопного застосунку.

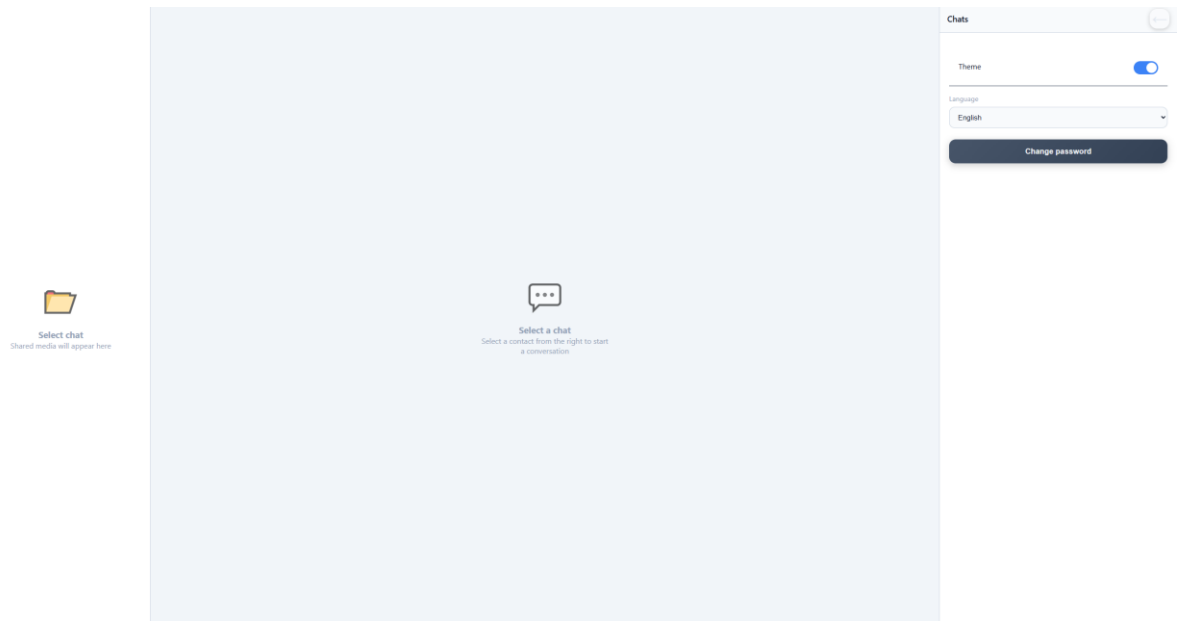


Рис. 23. Інтерфейс налаштувань системи в процесі перемикання мовних локалей (i18n) та активації темної теми (Dark Mode).

4. План робіт

Сюди входить комплекс організаційно-технічних заходів, спрямованих на послідовну верифікацію кожної підсистеми месенджера «Chatty». Процес випробувань було розбито на чотири логічні етапи (спринти), що дозволило впроваджувати тестування паралельно з розробкою за методологією Agile/Scrum та вчасно виявляти дефекти архітектури.

Етап 1. Тестування базових компонентів безпеки та ініціалізації (Тиждень 1): На цьому етапі виконувалася ізольована перевірка REST API ендпоінтів реєстрації та авторизації. Проводився аудит криптографічного захисту (генерація солі та безпечне хешування паролів через bcryptjs), а також тестування інтеграції з поштовим сервером SMTP через модуль **Nodemailer** для безперебійної доставки 6-значних TTL-кодів активації.

Етап 2. Інтеграційне тестування сховища даних та профілів

(Тиждень 2): Фокус робіт був спрямований на взаємодію між сервером Express та NoSQL базою даних MongoDB. Верифікувалася коректність виконання операцій CRUD при збереженні біографічних даних («Віо»), зміні відображуваних імен, а також обробка бінарних потоків аватарок за допомогою middleware-модуля **Multer**. Особлива увага приділялася тестуванню логіки захисту облікового запису при зміні пароля користувачем всередині активної сесії.

Етап 3. Тестування підсистеми реального часу та рендерингу

(Тиждень 3): Етап присвячений навантажувальному та функціональному тестуванню повнодуплексних WebSocket-з'єднань на базі бібліотеки **Socket.io**. Перевірявся механізм ізоляції кімнат чату (socket.join), динамічна ретрансляція повідомлень (io.to().emit) та коректність автоматичного завантаження й сортування історії повідомлень за часовими мітками з MongoDB без порушення структури Single Page Application.

Етап 4. Комплексне UX/UI випробування та системне відлагодження (Тиждень 4): Фінальний етап, що включав тестування інтерфейсу в екстремальних умовах (введення некоректних даних, завантаження файлів великого розміру, відтворення потокового відео через HTML5 плеєр). Також проводився аудит модулів інтернаціоналізації (i18n) та динамічного перемикування колірних тем із записом станів у localStorage. На цьому ж етапі було успішно усунено зафіксований раніше баг подвійного хешування при скиданні пароля. Нижче наведена таблиця «Календарний план-графік» проведення тестування веб-застосунку «Chatty»

Таблиця «Календарний план-графік»

№ етапу	Назва етапу тестування	Тривалість (дні)	Терміни виконання	Очікуваний результат / Звітний артефакт
1	Тестування REST API автентифікації, SMTP-транспорту та генерації TTL-кодів.	7	04.05.2026 – 10.05.2026	Стабільна реєстрація, успішна доставка листів верифікації через Nodemailer.
2	Інтеграційне тестування ODM Mongoose, операцій з БД MongoDB та Multer-завантажувача.	7	11.05.2026 – 17.05.2026	Безпомилкове збереження профілів, завантаження аватарок, валідація модальних вікон зміни пароля.
3	Тестування підієвої WebSocket-архітектури (Socket.io) та рендерингу історії повідомлень.	7	18.05.2026 – 24.05.2026	Миттєва доставка повідомлень у реальному часі, автоматичний скролінг та синхронізація історії чату.
4	UX/UI випробування (Media Tabs, HTML5 Player, i18n, Dark Mode), фіксація та усунення багів.	7	25.05.2026 – 31.05.2026	Повністю робоче SPA без дефектів, виправлено баг подвійного хешування, підготовлено фінальний реліз.

5. Кінцеві результати

Фінальним етапом процесу верифікації є узагальнення отриманих даних та формування матриці кінцевих результатів. Для підтвердження надійності та стабільності веб-застосунку «Chatty» було розроблено та виконано серію

наскрізних тестових сценаріїв (Test Cases), які охоплюють усі критичні вузли системи: від автентифікації до реал-тайм взаємодії та медіа-модулів.

У ході тестування перевірялися як позитивні сценарії (введення валідних даних, стандартна поведінка користувача), так і негативні сценарії (екстремальні умови, спроби обходу валідації, робота з помилками).

Результати успішного проходження всіх тестів зведено у таблицю 3.У.

ТЕСТОВІ ВИПАДКИ (TEST CASES)

Нижче наведено деталізований опис виконаних наскрізних тестових сценаріїв, які було використано для комплексної перевірки працездатності, стабільності та безпеки клієнт-серверної архітектури месенджера «Chatty».

Тестовий випадок ТС-01: Первинна реєстрація облікового запису та ініціалізація поштового транспорту

- **Мета:** Перевірка здатності сервера Express приймати реєстраційні дані, створювати новий документ у колекції users бази даних MongoDB та ініціювати надсилання листа верифікації через поштовий модуль Nodemailer.
- **Вхідні дані / Діє користувача:** Заповнення полів форми реєстрації валідними значеннями (унікальні Nickname, Display Name, Phone та Email), після чого здійснюється натискання інтерактивної кнопки *Register* (див. *Рис. 12*).
- **Очікуваний результат:** Сервер повертає HTTP-статус успішного виконання, створює запис у MongoDB із прапорцем безпеки `isVerified: false` та автоматично генерує 6-значний цифровий TTL-код. Модуль Nodemailer успішно формує захищене TLS-з'єднання зі SMTP-сервером та доставляє транзакційний лист на вказану пошту.
- **Фактичний результат:** У базі даних успішно з'явився новий документ користувача, а на вказану електронну пошту миттєво надійшов офіційний лист із цифровим токеном активації.
- **Статус:** Виконано успішно (**Passed**).

Тестовий випадок ТС-02: Верифікація облікового запису за допомогою TTL-коду

- **Мета:** Перевірка бізнес-логіки сервера щодо валідації одноразових кодів та активації облікового запису користувача.
- **Вхідні дані / Діє користувача:** Перехід на форму підтвердження та введення отриманого з електронного листа 6-значного цифрового коду, після чого надсилається запит на активацію.
- **Очікуваний результат:** Бекенд-контролер порівнює введений рядок із тим, що збережений у сесії бази даних для цього користувача, та перевіряє, чи не закінчився термін дії коду. У разі збігу статус користувача в базі змінюється на `isVerified: true`, маркер блокування знімається, а клієнтський JavaScript безшовно перенаправляє користувача на крок встановлення пароля.
- **Фактичний результат:** Код успішно валідовано сервером, прапорець верифікації в MongoDB перейшов у стан `true`, інтерфейс виконав редирект на форму створення пароля.
- **Статус:** Виконано успішно (**Passed**).

Тестовий випадок ТС-03: Автентифікація користувача та генерація сесійного токена JWT

- **Мета:** Верифікація механізму безсесійного доступу (Stateless), генерації та збереження токенів JSON Web Token.
- **Вхідні дані / Діє користувача:** Введення валідних облікових даних (Email та пароль) у форми входу (Log In) та активація кнопки авторизації.
- **Очікуваний результат:** Сервер зчитує з бази захешований пароль, порівнює його за допомогою методу `bcrypt.compare()`, генерує унікальний токен JWT, підписаний секретним ключем, та передає його клієнту. Клієнтський застосунок записує токен у `localStorage` та без перезавантаження сторінки перенаправляє інтерфейс на головне вікно додатка.

- **Фактичний результат:** Токен згенеровано та записано в локальне сховище браузера, виконано успішний безшовний перехід до основного інтерфейсу месенджера.
- **Статус:** Виконано успішно (**Passed**).

Тестовий випадок ТС-04: Обробка помилок автентифікації та активація сценарію відновлення доступу

- **Мета:** Перевірка стабільності роботи системи при виникненні виняткових ситуацій (Edge Cases) та валідація обробки HTTP-кодів помилок.
- **Вхідні дані / Діє користувача:** Введення завідомо некоректної комбінації логіна або пароля на сторінці авторизації.
- **Очікуваний результат:** Сервер Express відхиляє запит і повертає статус помилки 401 Unauthorized. Клієнтська частина перехоплює цей код, підсвічує поля червоним кольором, виводить текстове попередження та динамічно відмальовує під формою інтерактивне посилання «Забули пароль?».
- **Фактичний результат:** Система заблокувала несанкціонований вхід, вивела відповідне UX-повідомлення про помилку та надала користувачеві пряме посилання для відновлення втраченого доступу.
- **Статус:** Виконано успішно (**Passed**).

Тестовий випадок ТС-05: Повний цикл відновлення та скидання пароля

- **Мета:** Перевірка працездатності механізму відновлення облікових даних та верифікація виправлення багу подвійного хешування пароля.
- **Вхідні дані / Діє користувача:** Введення email аккаунта → введення нового тимчасового коду відновлення з пошти → заповнення форми встановлення нового пароля та його повторення.

- **Очікуваний результат:** Система перевіряє збіг полів «Новий пароль» та «Повторіть пароль» на клієнті. Сервер приймає новий рядок, генерує нову сіль, виконує *одноразове* шифрування через бібліотеку `bcryptjs` та оновлює документ у MongoDB, дозволяючи надалі авторизуватися за новими даними.
- **Фактичний результат:** Алгоритм скидання пароля відпрацював без помилок, баг подвійного хешування повністю відсутній, користувач успішно увійшов у систему за допомогою оновленого пароля.
- **Статус:** Виконано успішно (**Passed**).

Тестовий випадок ТС-06: Редагування особистого профілю та завантаження аватара

- **Мета:** Перевірка роботи HTTP-методу PUT, коректності функціонування завантажувача файлів Multer та персистентності даних користувача.
- **Вхідні дані / Діє користувача:** Зміна текстового статусу «Віо» (введення маркера «я тестировщик с 30 летним стажем» — див. *Рис. 12*), завантаження нового графічного файлу для відображення профілю та збереження змін.
- **Очікуваний результат:** Middleware Multer перехоплює бінарний потік, валідує його розмір (<10 МБ) та MIME-тип, записує файл у статичну директорію сервера. Метод `findOneAndUpdate()` драйвера Mongoose перезаписує поля в базі даних, а клієнтський інтерфейс динамічно оновлює зображення та текст без перезавантаження сторінки.
- **Фактичний результат:** Персональні дані успішно оновлено в MongoDB, аватар завантажено на сервер, графічний інтерфейс користувача відобразив зміни в реальному часі.
- **Статус:** Виконано успішно (**Passed**).

Тестовий випадок ТС-07: Функціонал динамічного пошуку та менеджменту контактів

- **Мета:** Верифікація коректності обробки пошукових запитів до NoSQL бази даних за допомогою регулярних виразів та перевірка логіки зв'язків «друг-користувач».
- **Вхідні дані / Діє користувача:** Перехід у вікно «Контакти» (див. *Рис. 13*), введення перших літер нікнейму (наприклад, @er — див. *Рис. 14*) та натискання кнопки «Додати до контактів» у картці знайденого користувача.
- **Очікуваний результат:** Сервер виконує пошук через оператор `Mongoose $regex` із прапорцем регістронезалежності і, повертає масив знайдених користувачів, розділяючи їх в UI на друзів та глобальний пошук. При додаванні друга сервер за допомогою методу `$push` записує його `_id` у масив контактів поточного юзера в MongoDB, змінюючи стан кнопки на «Видалити з контактів» (див. *Рис. 15*).
- **Фактичний результат:** Пошук та фільтрація працюють миттєво при введенні символів, додавання та видалення користувачів із френд-листа коректно синхронізується з базою даних.
- **Статус:** Виконано успішно (**Passed**).

Тестовий випадок ТС-08: Автоматична синхронізація та завантаження історії повідомлень

- **Мета:** Тестування швидкодії та логіки рендерингу історії діалогу при переході між кімнатами чату.
- **Вхідні дані / Діє користувача:** Клік мишею по конкретному активному діалогу в лівій навігаційній панелі месенджера.
- **Очікуваний результат:** Клієнтська частина надсилає GET-запит на ендпоінт історії чату. Сервер робить вибірку з колекції `messages` за поточним `chatId`, сортує масив за індексом часу `createdAt` і повертає його на фронтенд. JavaScript циклічно рендерить блоки повідомлень у DOM-дерево за допомогою `appendChild()` та автоматично викликає

метод `scrollIntoView()`, опускаючи екран до останньої репліки розмови (див. *Рис. 16*).

- **Фактичний результат:** Історія повідомлень підвантажується без затримок, інтерфейс чату автоматично фокусується на актуальних (найсвіжіших) повідомленнях діалогу.
- **Статус:** Виконано успішно (**Passed**).

Тестовий випадок ТС-09: Безпечна зміна пароля всередині авторизованої сесії

- **Мета:** Тестування дворівневого модального інтерфейсу безпеки та перевірка відповідності паролів при живому перепризначенні даних.
- **Вхідні дані / Діє користувача:** Натискання кнопки «Change password» у налаштуваннях $\rightarrow$ введення діючого пароля у вікні перевірки (див. *Рис. 17*) $\rightarrow$ задання нового пароля та його підтвердження у фінальному вікні (див. *Рис. 18*).
- **Очікуваний результат:** На першому етапі сервер порівнює пароль із хешем у БД і лише за умови збігу дозволяє відкрити форму введення нових даних. На другому етапі клієнт перевіряє ідентичність полів (не менше 6 символів), а сервер оновлює захешований запис у MongoDB, скидаючи сесію або оновлюючи криптографічний ключ.
- **Фактичний результат:** Двокрокова валідація відпрацювала бездоганно, клієнт блокує некоректне введення, сервер успішно оновив захешовані облікові дані в базі.
- **Статус:** Виконано успішно (**Passed**).

```

/* ===== RESET & BASE ===== */
*{
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  margin: 0;
  font-family: 'Segoe UI', system-ui, -apple-system, sans-serif;
  background: linear-gradient(135deg, #0a0f1c, #1a2338);
  height: 100vh;
  overflow: hidden;
  color: #e1e5ea;
}

html {
  transition: background-color 0.4s ease;
}

/* ===== AUTH SCREENS ===== */
.auth-wrapper {
  position: fixed;
  inset: 0;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  background: inherit;
  z-index: 100;
}

.auth-logo {
  margin-bottom: 50px;
  animation: logoAppear 1.4s ease-out forwards, floatLogo 0.8s ease-in-out infinite;
}

.auth-logo h1 {
  font-size: 4.2rem;
  font-weight: 800;
  background: linear-gradient(90deg, #60a5fa, #a5b4fc);
  -webkit-background-clip: text;
  background-clip: text; /* ← Додаємо стандартну властивість */
  -webkit-text-fill-color: transparent;
  letter-spacing: -2px;
}

@keyframes logoAppear {
  from { opacity: 0; transform: translateY(-80px) scale(0.7); }
  to { opacity: 1; transform: translateY(0) scale(1); }
}

@keyframes floatLogo {
  0%, 100% { transform: translateY(0); }
  50% { transform: translateY(-8px); }
}

.auth-container {

```

```

// ===== ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ =====
let currentEmail = "";
let currentChatId = null;
let currentUser = null;
let appInitialized = false;
let currentChat = null;
let isLoadingMessages = false;
let messagesContainer = null;
let lastReadTimestamps = {}; // { chatId: timestamp }

function loadLastReadTimestamps() {
  try {
    const userId = currentUser?._id || JSON.parse(localStorage.getItem('user') || '{}')._id;
    if (!userId) {
      lastReadTimestamps = {};
      return;
    }

    const key = `lastReadTimestamps_${userId}`;
    const saved = localStorage.getItem(key);

    lastReadTimestamps = saved ? JSON.parse(saved) : {};
    console.log('📁 Loaded read timestamps for user: ${userId}');
  } catch (e) {
    console.error('Failed to load read timestamps', e);
    lastReadTimestamps = {};
  }
}

function saveLastReadTimestamps() {
  try {
    const userId = currentUser?._id || JSON.parse(localStorage.getItem('user') || '{}')._id;
    if (!userId) return;

    const key = `lastReadTimestamps_${userId}`;
    localStorage.setItem(key, JSON.stringify(lastReadTimestamps));
  } catch (e) {
    console.error('Failed to save read timestamps', e);
  }
}

// ===== DOM ЭЛЕМЕНТЫ =====
const messageText = document.getElementById('messageText');
const messageFile = document.getElementById('messageFile');

// ===== AUTH FUNCTIONS =====
function showLogin() {
  hideAllAuth();
  const loginForm = document.getElementById('loginForm');
  if (loginForm) loginForm.style.display = 'flex';
}

function showRegister() {

```

```

hideAllAuth();
const regStep1 = document.getElementById('registerStep1');
if (regStep1) regStep1.style.display = 'flex';
}

function hideAllAuth() {
  document.querySelectorAll('.auth-wrapper').forEach(el => el.style.display = 'none');
}

function capitalize(str) {
  if (!str) return '';
  return str.charAt(0).toUpperCase() + str.slice(1);
}

async function login() {
  const identifierInput = document.getElementById('loginIdentifier');
  const passwordInput = document.getElementById('loginPassword');

  if (!identifierInput || !passwordInput) return;

  const identifier = identifierInput.value.trim();
  const password = passwordInput.value;

  // Проверка заполнения полей с использованием локализации или дефолтного текста
  if (!identifier || !password) {
    alert(window.currentTranslations?.fillAllFields || 'Будь ласка, заповніть всі поля');
    return;
  }

  try {
    const res = await fetch('/api/auth/login', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ identifier, password })
    });

    const data = await res.json();

    if (!res.ok) {
      // Если сервер вернул ошибку авторизации — показываем скрытую ссылку на восстановление пароля
      const forgotLink = document.getElementById('forgotPasswordLink');
      if (forgotLink) forgotLink.style.display = 'block';

      throw new Error(data.error || 'Помилка входу');
    }

    // Сохраняем полученные данные авторизации в локальное хранилище
    localStorage.setItem('token', data.token);
    localStorage.setItem('user', JSON.stringify(data.user));
    currentUser = data.user;

    // Полностью очищаем поля формы
    identifierInput.value = '';
    passwordInput.value = '';

    // Прячем ссылку восстановления, так как вход выполнен успешно
    const forgotLink = document.getElementById('forgotPasswordLink');
    if (forgotLink) forgotLink.style.display = 'none';
  }
}

```



```

// Инициализируем веб-сокеты для работы чата в реальном времени
if (typeof initSocket === 'function') {
  initSocket();
}

// Сбрасываем флаг инициализации, чтобы разрешить запуск приложения
appInitialized = false;

if (typeof initializeApp === 'function') {
  await initializeApp();
} else {
  console.error("❌ Критическая ошибка: Функция initializeApp не найдена в коде!");
}

} catch (err) {
  console.error("❌ Login error:", err);
  alert(err.message);
}
}

function logout() {
  localStorage.removeItem('token');
  localStorage.removeItem('user');
  if (window.socket) socket.disconnect();
  location.reload();
}

async function goToCodeVerification() {
  const nickname = document.getElementById('regNickname').value.trim();
  const displayName = document.getElementById('regDisplayName').value.trim();
  const email = document.getElementById('regEmail').value.trim();
  const phone = document.getElementById('regPhone').value.trim();

  if (!nickname || !email) return alert(t("nicknameEmailRequired"));

  const btn = document.querySelector("#registerStep1 button");
  if (btn) {
    btn.textContent = t("sendingCode");
    btn.disabled = true;
  }

  try {
    const res = await fetch('/api/auth/register', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ nickname, displayName, email, phone })
    });

    const data = await res.json();

    if (data.success) {
      currentEmail = email;
      const sentToEl = document.getElementById('sentTo');
      if (sentToEl) sentToEl.textContent = email;
      hideAllAuth();
      const regStep2 = document.getElementById('registerStep2');
      if (regStep2) regStep2.style.display = 'flex';
    } else {
      alert(data.error || t("registrationError"));
    }
  }
}

```

```

    }
  } catch (err) {
    alert(t("connectionError"));
  } finally {
    if (btn) {
      btn.textContent = t("register");
      btn.disabled = false;
    }
  }
}
}

```

```

async function verifyCode() {
  const code = document.getElementById('verificationCode').value.trim();
  if (code.length !== 6) return alert(t("enter6DigitCode"));

```

```

  try {
    const res = await fetch('/api/auth/verify-code', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ email: currentEmail, code })
    });

    const data = await res.json();

    if (data.success) {
      hideAllAuth();
      const regStep3 = document.getElementById('registerStep3');
      if (regStep3) regStep3.style.display = 'flex';
    } else {
      alert(data.error || t("invalidCode"));
    }
  } catch (err) {
    alert(t("codeVerificationError"));
  }
}

```

```

async function finishRegistration() {
  const password = document.getElementById('regPassword').value;
  const passwordRepeat = document.getElementById('regPasswordRepeat').value;

  if (password !== passwordRepeat) return alert(t("passwordsDoNotMatch"));
  if (password.length < 6) return alert(t("passwordTooShort"));

```

```

  const btn = document.querySelector("#registerStep3 button");
  if (btn) {
    btn.textContent = t("finishingRegistration");
    btn.disabled = true;
  }

```

```

  try {
    const res = await fetch('/api/auth/finish-registration', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        email: currentEmail,
        password,
        nickname: document.getElementById('regNickname').value.trim(),
        displayName: document.getElementById('regDisplayName').value.trim(),
        phone: document.getElementById('regPhone').value.trim()
      })
    });

```

```

});

const data = await res.json();

if (data.success) {
  localStorage.setItem('token', data.token);
  localStorage.setItem('user', JSON.stringify(data.user));

  alert(t("registrationSuccess"));
  showMessenger();
} else {
  alert(data.error || t("finishRegistrationError"));
}
} catch (err) {
  alert(t("finishRegistrationError"));
} finally {
  if (btn) {
    btn.textContent = t("continue");
    btn.disabled = false;
  }
}
}
}

```

// ===== ЛОГИКА ВОССТАНОВЛЕНИЯ ПАРОЛЯ =====

```

let recoveryState = {
  email: "",
  code: ""
};

function hideAllAuthScreens() {
  const screens = [
    'loginForm', 'registerStep1', 'registerStep2', 'registerStep3',
    'recoveryStep1', 'recoveryStep2', 'recoveryStep3'
  ];
  screens.forEach(id => {
    const el = document.getElementById(id);
    if (el) el.style.display = 'none';
  });
}

function showRecoveryStep1() {
  hideAllAuthScreens();
  const el = document.getElementById('recoveryStep1');
  if (el) el.style.display = 'flex';
}

function showRecoveryStep2() {
  hideAllAuthScreens();
  const el = document.getElementById('recoveryStep2');
  if (el) el.style.display = 'flex';
}

function showRecoveryStep3() {
  hideAllAuthScreens();
}

```

```

const el = document.getElementById('recoveryStep3');
if (el) el.style.display = 'flex';
}

// Шаг 1: Запрос кода восстановления
async function sendRecoveryEmail() {
  const emailInput = document.getElementById('recoveryEmail');
  if (!emailInput) return;

  const email = emailInput.value.trim();
  if (!email) return;

  const currentLang = localStorage.getItem('chatty_lang') || 'uk';

  try {
    const res = await fetch('/api/auth/forgot-password', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        email,
        lang: currentLang,
        purpose: 'recovery' // Передаем назначение коду
      })
    });

    const data = await res.json();
    if (!res.ok) throw new Error(data.error || 'Помилка відправки кода');

    recoveryState.email = email;
    showRecoveryStep2();
  } catch (err) {
    console.error(err);
    alert(err.message);
  }
}

// Шаг 2: Проверка кода
async function verifyRecoveryCode() {
  const codeInput = document.getElementById('recoveryCode');
  if (!codeInput) return;

  const code = codeInput.value.trim();
  if (!code) return;

  try {
    const res = await fetch('/api/auth/verify-recovery-code', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        email: recoveryState.email,
        code,
        purpose: 'recovery' // Передаем назначение коду
      })
    });

    const data = await res.json();
    if (!res.ok) throw new Error(data.error || 'Невірний код');

    recoveryState.code = code;
    showRecoveryStep3();
  }

```

```

    } catch (err) {
        console.error(err);
        alert(err.message);
    }
}

// Шаг 3: Сохранение нового пароля
async function submitNewPassword() {
    const newPassInput = document.getElementById('recoveryNewPassword');
    const confirmPassInput = document.getElementById('recoveryConfirmPassword');

    if (!newPassInput || !confirmPassInput) return;

    const newPassword = newPassInput.value;
    const confirmPassword = confirmPassInput.value;

    if (newPassword.length < 6) {
        alert(window.currentTranslations?.passwordTooShort || 'Пароль повинен бути не менше 6 символів');
        return;
    }

    if (newPassword !== confirmPassword) {
        alert(window.currentTranslations?.passwordsDoNotMatch || 'Паролі не співпадають');
        return;
    }

    try {
        const res = await fetch('/api/auth/reset-password', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({
                email: recoveryState.email,
                code: recoveryState.code,
                newPassword,
                purpose: 'recovery' // Передаем назначение коду
            })
        });

        const data = await res.json();
        if (!res.ok) throw new Error(data.error || 'Помилка зміни пароля');

        alert(window.currentTranslations?.passwordResetSuccess || 'Пароль успішно змінено! 🎉');

        recoveryState = { email: '', code: '' };
        newPassInput.value = '';
        confirmPassInput.value = '';

        showLogin();
    } catch (err) {
        console.error(err);
        alert(err.message);
    }
}

// ===== MESSENGER =====

async function loadMessages() {
    if (!messagesContainer) return;
    try {

```

```

const res = await fetch('/api/messages', {
  headers: {
    Authorization: `Bearer ${localStorage.getItem('token')}`
  }
});
if (!res.ok) throw new Error('Failed');

const messages = await res.json();
messagesContainer.innerHTML = "";
if (typeof renderMessage === 'function') {
  messages.forEach(renderMessage);
}
scrollToBottom();
} catch (e) {
  console.error('Messages load error:', e);
}
}

function scrollToBottom() {
  if (messagesContainer) {
    messagesContainer.scrollTop = messagesContainer.scrollHeight;
  }
}

async function sendMessage() {
  if (!messageText || !messageFile) return;

  const text = messageText.value.trim();
  const file = messageFile.files[0];

  if (!text && !file) return;

  if (!currentChatId) {
    alert(t("selectChatFirst") || "Виберіть чат спочатку");
    return;
  }

  // Защита от двойной отправки
  const sendBtn = document.querySelector('#messageInputArea button:last-child');
  if (sendBtn && sendBtn.disabled) return;

  const formData = new FormData();
  if (text) formData.append('text', text);
  if (file) formData.append('file', file);

  // Очищаем сразу
  messageText.value = "";
  const originalBtnText = sendBtn ? sendBtn.textContent : '➤';

  if (sendBtn) {
    sendBtn.disabled = true;
    sendBtn.textContent = '⌛';
  }

  try {
    const res = await fetch(`/api/messages?chatId=${currentChatId}`, {
      method: 'POST',
      headers: {

```

```

        Authorization: `Bearer ${localStorage.getItem('token')}`
      },
      body: formData
    });

const result = await res.json();

if (!result.success) {
  throw new Error(result.error || t("sendMessageFailed"));
}

// Успешная отправка — очищаем превью файла
clearFilePreview();

hideNoChatPlaceholder();

// Обновляем список чатов слева (чтобы поднялся наверх и обновился текст последнего сообщения)
if (typeof loadChats === 'function') loadChats();

// 🔥 ОНОВЛЕНИЯ МЕДИА-КОНТЕЙНЕРА: Перезагружаем медиа-вкладки текущего чата
if (typeof loadMediaForChat === 'function') {
  await loadMediaForChat(currentChatId);
}

// Обрабатываем логику добавления сообщения в интерфейс/скролл
if (typeof handleNewMessage === 'function') {
  handleNewMessage(
    currentChatId,
    new Date(result.message.createdAt).getTime()
  );
}
} catch (err) {
  console.error('❌ Send error:', err);
  alert(err.message || t("sendMessageFailed"));
} finally {
  if (sendBtn) {
    sendBtn.disabled = false;
    sendBtn.textContent = originalBtnText;
  }
}
}

function hideNoChatPlaceholder() {
  console.log('🔍 Hiding placeholder...');

  const placeholder = document.getElementById('noChatPlaceholder');
  if (placeholder) {
    placeholder.classList.remove('show');
    placeholder.style.display = 'none';
    console.log('✅ Placeholder hidden');
  }

  const messagesEl = document.getElementById('messages');
  const inputArea = document.getElementById('messageInputArea');

  if (messagesEl) messagesEl.style.display = 'flex';
  if (inputArea) inputArea.style.display = 'flex';
}

```

```
// ===== MESSENGER CORE / INIT =====

function initMessenger() {
  messagesContainer = document.querySelector('#messages');
  initFilePreview();
  console.log('MESSAGES CONTAINER:', messagesContainer);

  if (!messagesContainer) {
    console.error('#messages not found');
    return;
  }

  const sendBtn = document.querySelector('#messageInputArea button:last-child');
  if (sendBtn) {
    // Удаляем старый слушатель во избежание дублирования
    sendBtn.removeEventListener('click', sendMessage);
    sendBtn.addEventListener('click', sendMessage);
  }

  const messageTextEl = document.getElementById('messageText');
  if (messageTextEl) {
    messageTextEl.addEventListener('keypress', (e) => {
      if (e.key === 'Enter' && !e.shiftKey) {
        e.preventDefault();
        sendMessage();
      }
    });
  }

  const attachBtn = document.getElementById('attachBtn');
  if (attachBtn) {
    attachBtn.addEventListener('click', () => {
      const fileInput = document.getElementById('messageFile');
      if (fileInput) fileInput.click();
    });
  }
}

async function showMessenger() {
  hideAllAuth();

  const messenger = document.getElementById('messenger');
  if (messenger) messenger.style.display = 'flex';

  await new Promise(resolve => setTimeout(resolve, 50));

  initMessenger();
  if (typeof resetChatView === 'function') resetChatView();

  if (typeof loadChats === 'function') loadChats();
  if (typeof startChatsPolling === 'function') startChatsPolling();
  if (typeof initSocket === 'function') initSocket();
}

async function loadCurrentUser() {
  const token = localStorage.getItem('token');
  if (!token) return null;

  try {
    const res = await fetch('/api/users/me', {

```



```

        headers: { Authorization: `Bearer ${token}` }
    });

    if (!res.ok) throw new Error('Unauthorized');

    const user = await res.json();

    currentUser = user;
    localStorage.setItem('user', JSON.stringify(user));

    console.log('CURRENT USER LOADED:', user);
    updateCurrentUserUI();
    return user;

} catch (err) {
    console.error('loadCurrentUser error:', err);
    return null;
}
}

function updateCurrentUserUI() {
    if (!currentUser) return;

    const els = {
        userDisplayName: document.getElementById('userDisplayName'),
        userTag: document.getElementById('userTag'),
        profileAvatar: document.getElementById('profileAvatar')
    };

    if (els.userDisplayName) els.userDisplayName.value = currentUser.displayName || currentUser.nickname || '';
    if (els.userTag) els.userTag.value = currentUser.tag || '';

    if (els.profileAvatar && currentUser.avatar && typeof updateProfileAvatar === 'function') {
        updateProfileAvatar(currentUser.avatar);
    }
}

// ===== МЕНЮ И ПАНЕЛИ =====

let currentPanel = 'menuContent';

const menuLayer = document.getElementById('menuLayer');
const menuToggle = document.getElementById('menuToggle');
const panelTitle = document.getElementById('panelTitle');

function getPanelTitle(panelKey) {
    const titles = {
        menuContent: t('menu'),
        profileMenu: t('myProfile'),
        contactsMenu: t('contacts'),
        settingsMenu: t('settings')
    };
    return titles[panelKey] || t('menu');
}

function updateHeader() {
    if (!menuLayer || !panelTitle || !menuToggle) return;
    const isMenuOpen = menuLayer.classList.contains('open');

```

```

// Меню закрыто → обычный режим чатов
if (!isMenuOpen) {
    panelTitle.textContent = t('chats');
    menuToggle.textContent = '≡';
    return;
}

// Меню открыто → берем локализованный заголовок панели
panelTitle.textContent = getPanelTitle(currentPanel);

// Главное меню → кнопка закрытия
if (currentPanel === 'menuContent') {
    menuToggle.textContent = '×';
} else {
    // Подменю → кнопка назад
    menuToggle.textContent = '←';
}
}

function openMenu() {
    if (!menuLayer) return;
    menuLayer.classList.add('open');

    document.querySelectorAll('.menu-panel').forEach(panel => {
        panel.classList.remove('active', 'slide-out-left', 'slide-out-right');
    });

    const menuContent = document.getElementById('menuContent');
    if (menuContent) menuContent.classList.add('active');
    currentPanel = 'menuContent';

    updateHeader();
}

function closeMenu() {
    if (!menuLayer) return;
    menuLayer.classList.remove('open');
    currentPanel = 'menuContent';

    document.querySelectorAll('.menu-panel').forEach(panel => {
        panel.classList.remove('active', 'slide-out-left', 'slide-out-right');
    });

    const menuContent = document.getElementById('menuContent');
    if (menuContent) menuContent.classList.add('active');

    updateHeader();
}

function showPanel(panelId) {
    const newPanel = document.getElementById(panelId);
    const oldPanel = document.getElementById(currentPanel);

    if (!newPanel || panelId === currentPanel) return;

    const goingBack = currentPanel !== 'menuContent' && panelId === 'menuContent';

    document.querySelectorAll('.menu-panel').forEach(panel => {
        panel.classList.remove('slide-out-left', 'slide-out-right');
    });
}

```

```

if (oldPanel) {
    oldPanel.classList.remove('active');
    oldPanel.classList.add(goingBack ? 'slide-out-right' : 'slide-out-left');
}

newPanel.classList.remove('slide-out-left', 'slide-out-right');

requestAnimationFrame(() => {
    newPanel.classList.add('active');
});

currentPanel = panelId;
updateHeader();
}

function toggleMenu() {
    if (!menuLayer) return;

    // Меню закрыто → открыть
    if (!menuLayer.classList.contains('open')) {
        openMenu();
        return;
    }

    // В подменю → назад в главное меню
    if (currentPanel !== 'menuContent') {
        showPanel('menuContent');
        return;
    }

    // В главном меню → закрыть
    closeMenu();
}

function buildMenuContent() {
    // ===== КОНТАКТЫ =====
    const contactsMenu = document.getElementById('contactsMenu');
    if (contactsMenu) {
        contactsMenu.innerHTML = `
            <div class="settings-section">
                <label data-i18n="contactSearch">Пошук контактів</label>

                <input
                    type="text"
                    id="contactsSearchInput"
                    data-i18n-placeholder="contactSearchPlaceholder"
                    autocomplete="off"
                >

                <div id="myContactsList" class="contacts-list"></div>

                <div id="searchDivider" style="display:none;">
                    <span data-i18n="searchResults">Результати пошуку</span>
                </div>

                <div id="searchContactsList"></div>
            </div>
        `;
    }
}

```

```
// ===== НАСТРОЙКИ =====
const settingsMenu = document.getElementById('settingsMenu');
if (settingsMenu) {
  settingsMenu.innerHTML = `
    <div class="settings-section">
      <div class="setting-row">
        <span data-i18n="theme">Тема</span>
        <label class="switch">
          <input type="checkbox" id="themeToggle">
          <span class="slider"></span>
        </label>
      </div>

      <label data-i18n="language">Мова</label>
      <select id="languageSelect" onchange="setLanguage(this.value)">
        <option value="uk">Українська</option>
        <option value="en">English</option>
        <option value="de">Deutsch</option>
        <option value="es">Español</option>
      </select>

      <button class="secondary-btn full-width" type="button" data-i18n="changePassword"
onclick="openPasswordModal()">
        Змінити пароль
      </button>

    </div>
  `;
}

// ===== ПРОФИЛЬ (БЕЗ ДУБЛИРОВАНИЯ И СБОЕВ) =====
const profileMenu = document.getElementById('profileMenu');
if (profileMenu) {
  profileMenu.innerHTML = `
    <div class="profile-section">
      <div class="profile-avatar-upload">
        <input
          type="file"
          id="profileAvatarInput"
          accept="image/*"
          style="display:none"
        >

        <div
          class="profile-avatar-placeholder"
          id="profileAvatar"
          data-i18n-title="changePhoto"
          onclick="document.getElementById('profileAvatarInput').click()"
        >
          
        </div>
        <div class="avatar-edit-icon">  </div>
      </div>

      <div class="form-group">
        <label data-i18n="tag">Ter</label>
        <input type="text" id="userTag" data-i18n-placeholder="nickname" placeholder="@username">
      </div>
    </div>
  `;
}

```

```

        <div class="form-group">
            <label data-i18n="displayName">Відображуване ім'я</label>
            <input type="text" id="userDisplayName" data-i18n-placeholder="yourName" placeholder="Ваше ім'я">
        </div>

        <div class="form-group">
            <label data-i18n="bio">Біографія</label>
            <textarea id="userBio" rows="4" data-i18n-placeholder="tellAboutYourself" placeholder="Розкажіть про себе..."></textarea>
        </div>

        <button id="saveProfileBtn" class="save-profile-btn" disabled data-i18n="save">
            Зберегти
        </button>
    </div>
    `;
}

// ===== ИНИЦИАЛИЗАЦИЯ И СЛУШАТЕЛИ =====
setTimeout(() => {
    const input = document.getElementById('profileAvatarInput');
    if (input) {
        input.addEventListener('change', () => {
            const file = input.files[0];
            if (file) uploadProfileAvatar(file);
        });
    }

    // Проверяем кэш и отрисовываем аватарку
    const user = JSON.parse(localStorage.getItem('user') || '{}');
    if (user.avatar) updateProfileAvatar(user.avatar);

    // Инициализируем системные обработчики из app.js
    if (typeof initProfileListeners === 'function') initProfileListeners();
    if (typeof loadProfile === 'function') loadProfile();
    if (typeof initContacts === 'function') initContacts();
    if (typeof loadMyContacts === 'function') loadMyContacts();

}, 50);

setTimeout(() => {
    if (window.applyTranslations) {
        window.applyTranslations();
    }

    // Автоматически синхронизируем выбранное значение в выпадающем списке языков
    const select = document.getElementById('languageSelect');
    if (select && window.getCurrentLanguage) {
        select.value = window.getCurrentLanguage();
    }
    initThemeToggle();
}, 0);
}

// Инициализация
function initMenu() {
    buildMenuContent();

    if (menuToggle) {

```

```

        menuToggle.removeEventListener('click', toggleMenu);
        menuToggle.addEventListener('click', toggleMenu);
    }

    document.querySelectorAll('#menuContent .menu-item[data-panel]').forEach(item => {
        item.addEventListener('click', () => {
            showPanel(item.dataset.panel);
        });
    });

    const exitBtn = document.querySelector('#menuContent .menu-item.exit');
    if (exitBtn) {
        exitBtn.addEventListener('click', logout);
    }

    updateHeader();
}

// ===== CONTACTS =====

let searchTimeout = null;
let currentModalUser = null;

// Ініціалізація розділу контактів
function initContacts() {
    const searchInput = document.getElementById('contactsSearchInput');

    console.log('initContacts()', searchInput);

    if (!searchInput) {
        console.warn('contactsSearchInput not found');
        return;
    }

    // Не ініціалізуємо повторно
    if (searchInput.dataset.initialized === 'true') {
        return;
    }

    // Обробник введення
    searchInput.addEventListener('input', (e) => {
        const rawValue = e.target.value;
        console.log('INPUT:', rawValue);

        clearTimeout(searchTimeout);

        searchTimeout = setTimeout(() => {
            let query = rawValue.trim();

            // Дозволяємо шукати по @nickname
            if (query.startsWith('@')) {
                query = query.slice(1);
            }

            console.log('SEARCH:', query);

            performContactsSearch(query);
        }, 400);
    });
}

```

```

searchInput.dataset.initialized = 'true';

// Завантажуємо список моїх контактів
loadMyContacts();
}

// Завантаження списку моїх контактів
async function loadMyContacts() {
  const list = document.getElementById('myContactsList');
  if (!list) {
    console.warn('myContactsList not found');
    return;
  }

  const token = localStorage.getItem('token');
  if (!token) return;

  try {
    const res = await fetch('/api/users/contacts', {
      headers: { Authorization: `Bearer ${token}` }
    });

    let data = await res.json();

    // На випадок, якщо бекенд повертає {success: true, contacts: [...]}
    const contacts = data.success ? (data.contacts || []) : (Array.isArray(data) ? data : []);

    list.innerHTML = "";

    if (!contacts.length) {
      list.innerHTML = `
        <div class="empty-contacts">
          ${t('noContactsYet')}
        </div>
      `;
      return;
    }

    contacts.forEach(user => {
      list.appendChild(createContactItem(user, true));
    });

  } catch (err) {
    console.error('loadMyContacts error:', err);
    list.innerHTML = `<div class="empty-contacts">${t('contactsLoadError')}</div>`;
  }
}

// Пошук користувачів
// Пошук користувачів із розподілом на категорії
// Пошук користувачів із розподілом на категорії та підтримкою локалізації
async function performContactsSearch(query) {
  const myContactsList = document.getElementById('myContactsList');
  const searchResults = document.getElementById('searchContactsList');
  const divider = document.getElementById('searchDivider');

  if (!searchResults) {
    console.warn('searchContactsList not found');
  }

```

```

    return;
}

if (!query || query.trim() === "") {
    if (myContactsList) myContactsList.style.display = 'block';
    searchResults.innerHTML = "";
    if (divider) divider.style.display = 'none';
    return;
}

if (myContactsList) myContactsList.style.display = 'none';

const token = localStorage.getItem('token');
if (!token) return;

try {
    const res = await fetch(
        `/api/users/search?query=${encodeURIComponent(query.trim())}`,
        {
            headers: { Authorization: `Bearer ${token}` }
        }
    );

    const data = await res.json();
    console.log("🔍 SEARCH RESULTS:", data);

    searchResults.innerHTML = "";

    if (!data.success || !data.users) {
        searchResults.innerHTML = `<div style="padding:15px;color:#94a3b8;">${t('serverError')} || 'Помилка сервера'</div>`;
        return;
    }

    const friends = data.users.friends || [];
    const nonFriends = data.users.nonFriends || [];

    // Если вообще никого не нашли
    if (friends.length === 0 && nonFriends.length === 0) {
        searchResults.innerHTML = `<div style="padding:15px;color:#94a3b8;">${t('nothingFound')} || 'Нічого не знайдено'</div>`;
        if (divider) divider.style.display = 'none';
        return;
    }

    if (divider) divider.style.display = 'block';

    // Проверяем: есть ли хотя бы один друг в результатах поиска?
    const hasFriendsInSearch = friends.length > 0;

    // 1. Выводим раздел "Ваші Контакти" (только если они нашлись)
    if (hasFriendsInSearch) {
        const friendsTitle = document.createElement('div');
        friendsTitle.className = 'search-section-title';
        friendsTitle.style = 'padding: 10px 16px 6px 16px; font-size: 0.75rem; font-weight: 700; color: #3b82f6; text-transform: uppercase; letter-spacing: 0.05em;';
        // Используем t() для локализации заголовка друзей
        friendsTitle.innerText = t('myContactsSection');
        searchResults.appendChild(friendsTitle);
    }

```



```

        friends.forEach(user => {
            searchResults.appendChild(createContactItem(user, true));
        });
    }

    // 2. Выводим раздел "Інші користувачі"
    if (nonFriends.length > 0) {
        // Если друзья есть — добавляем заголовок. Если друзей нет — заголовок НЕ выводим (остается
        // чистый результат)
        if (hasFriendsInSearch) {
            const nonFriendsTitle = document.createElement('div');
            nonFriendsTitle.className = 'search-section-title';
            nonFriendsTitle.style = 'padding: 16px 16px 6px 16px; font-size: 0.75rem; font-weight: 700; color: #64748b;
            text-transform: uppercase; letter-spacing: 0.05em;';
            // Используем t() для локализации заголовка глобального поиска
            nonFriendsTitle.innerText = t('otherUsersSection');
            searchResults.appendChild(nonFriendsTitle);
        }

        nonFriends.forEach(user => {
            searchResults.appendChild(createContactItem(user, user.isFriend || false));
        });
    }

} catch (err) {
    console.error('performContactsSearch error:', err);
    searchResults.innerHTML = `<div style="padding:15px;color:#ef4444;">${t('connectionError')} || 'Помилка
з'єднання'</div>`;
}
}

// Створення елемента контакту
function createContactItem(user, isFriend = false) {
    const div = document.createElement('div');
    div.className = 'contact-item';

    const avatarUrl = user.avatar
        ? `${user.avatar}?t=${Date.now()}`
        : "";

    div.innerHTML = `
        <div class="contact-avatar"
            style="${avatarUrl ? `background-image:url('${avatarUrl}')` : ""}"
            ${!avatarUrl ? 'background-image:url(assets/images/default-avatar.png)' : ""}
        </div>

        <div class="contact-info">
            <div class="contact-name">
                ${user.displayName || user.nickname}
            </div>
            <div class="contact-tag">
                ${user.tag || '@' + user.nickname}
            </div>
        </div>
    `;

    div.addEventListener('click', () => {
        openContactModal({
            ...user,
            isFriend

```

```

    });
  });

  return div;
}

// ===== CONTACT MODAL =====

// Відкриття модального вікна
function openContactModal(user) {
  currentModalUser = user;

  const modal = document.getElementById('contactModal');
  const avatarEl = document.getElementById('modalAvatar');
  const nameEl = document.getElementById('modalName');
  const tagEl = document.getElementById('modalTag');
  const bioEl = document.getElementById('modalBio');
  const actionsEl = document.getElementById('modalActions');

  if (!modal || !avatarEl || !nameEl || !tagEl || !bioEl || !actionsEl) {
    console.error('Contact modal elements not found');
    return;
  }

  // ===== Аватар =====
  if (user.avatar) {
    avatarEl.textContent = "";
    avatarEl.style.backgroundImage = `url("${user.avatar}?t=${Date.now()}")`;
  } else {
    avatarEl.style.backgroundImage = "";
    avatarEl.textContent = '👤';
  }

  // ===== Основна інформація =====
  nameEl.textContent = user.displayName || user.nickname || t('user');
  tagEl.textContent = user.tag || `@${user.nickname || ''}`;
  bioEl.textContent = user.bio || t('bioAbsent');

  // ===== Кнопки =====
  renderModalButtons(user);

  // ===== Показати модальку =====
  modal.classList.add('show');
}

// Закриття модального вікна
function closeContactModal() {
  const modal = document.getElementById('contactModal');
  if (!modal) return;

  modal.classList.remove('show');
  currentModalUser = null;
}

// Закриття по кліку на фон
document.addEventListener('click', (e) => {
  const modal = document.getElementById('contactModal');
  if (!modal) return;

  if (e.target === modal) {

```

```

        closeContactModal();
    }
});

// Закриття по Escape
document.addEventListener('keydown', (e) => {
    if (e.key === 'Escape') {
        closeContactModal();
    }
});

// Рендер кнопок у модальному вікні контакту
function renderModalButtons(user) {
    const actionsEl = document.getElementById('modalActions');
    if (!actionsEl) return;

    actionsEl.innerHTML = "";

    let buttonsHTML = "";

    // Якщо це не мій контакт — показуємо кнопку додавання
    if (!user.isFriend) {
        buttonsHTML += `
            <button class="modal-btn primary" onclick="addContact()">
                👤 ${t('addToContacts')}
            </button>
        `;
    } else {
        buttonsHTML += `
            <button class="modal-btn danger" onclick="removeContact()">
                ${t('deleteFromContacts')}
            </button>
        `;
    }

    // Кнопка написання повідомлення — показується завжди
    buttonsHTML += `
        <button class="modal-btn primary" onclick="startChatWith('${user._id}')">
            💬 ${t('writeMessage')}
        </button>
    `;

    actionsEl.innerHTML = buttonsHTML;
}

// Глобальна функція для відкриття чату з модального вікна
window.startChatWith = async function(userId) {
    if (!userId) return;

    // 1. Закриваємо модальку профілю користувача
    closeContactModal();

    // ===== НАШ ФІКС ДЛЯ UX =====
    // 2. Повертаємо ліве меню назад до списку чатів (закриваємо шторку меню)
    if (typeof closeMenu === 'function') {
        console.log('📩 Закриваємо меню та повертаємось до чатів...');
        closeMenu();
    }

    // 3. Очищаємо інпут пошуку контактів та скидаємо результати фільтрації

```

```

const contactsSearchInput = document.getElementById('contactsSearchInput');
if (contactsSearchInput) {
  contactsSearchInput.value = ''; // Стираємо написаний текст
}

// Викликаємо функцію пошуку з порожнім рядком, щоб вона сховала результати та увімкнула базовий
// список
if (typeof performContactsSearch === 'function') {
  console.log(' 🚧 Скидаємо фільтр та результати пошуку контактів...');
  performContactsSearch('');
}
// =====

// 4. Відкриваємо безпосередньо сам чат
if (typeof openChat === 'function') {
  await openChat(userId);
}
};

// Додавання користувача до контактів
async function addContact() {
  if (!currentModalUser) return;

  const token = localStorage.getItem('token');
  if (!token) return;

  try {
    const res = await fetch('/api/users/add-contact', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        Authorization: `Bearer ${token}`
      },
      body: JSON.stringify({
        userId: currentModalUser._id
      })
    });

    const data = await res.json();

    if (!res.ok || !data.success) {
      throw new Error(data.error || t('addContactError'));
    }

    // 1. Оновлюємо локальний стан в модальці
    currentModalUser.isFriend = true;

    // 2. СИНХРОНІЗАЦІЯ: Обов'язково додаємо ID до глобального об'єкта currentUser,
    // щоб шоу чату та хедери відразу бачили, що це наш друг без перезавантаження F5!
    if (currentUser && Array.isArray(currentUser.contacts)) {
      const alreadyInContacts = currentUser.contacts.some(c => {
        const cId = (typeof c === 'object' && c !== null) ? c._id : c;
        return cId === currentModalUser._id;
      });
      if (!alreadyInContacts) {
        currentUser.contacts.push(currentModalUser._id);
      }
    }

    // 3. Перемальовуємо кнопки в модальці

```

```

renderModalButtons(currentModalUser);

// 4. Оновлюємо список контактів у лівій панелі
if (typeof loadMyContacts === 'function') {
  loadMyContacts();
}

} catch (err) {
  console.error('addContact error:', err);
  alert(err.message);
}
}

// Видалення користувача з контактів ТА ПОВНЕ скидання до Ширми
async function removeContact() {
  if (!currentModalUser) return;

  // Быстрое подтверждение, чтобы случайно не снести переписку
  if (!confirm("Ви впевнені, що хочете видалити цей контакт та всю переписку з ним?")) return;

  const token = localStorage.getItem('token');
  if (!token) return;

  try {
    const res = await fetch('/api/users/remove-contact', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        Authorization: `Bearer ${token}`
      },
      body: JSON.stringify({
        userId: currentModalUser._id
      })
    });

    const data = await res.json();

    if (!res.ok || !data.success) {
      throw new Error(data.error || t('removeContactError'));
    }

    console.log('☑ Контакт та чат видалено з сервера');

    // 1. Оновлюємо локальний стан в модалці
    currentModalUser.isFriend = false;

    // 2. СИНХРОНІЗАЦІЯ: Видаляємо ID з пам'яті currentUser
    if (currentUser && Array.isArray(currentUser.contacts)) {
      currentUser.contacts = currentUser.contacts.filter(c => {
        const cId = (typeof c === 'object' && c !== null) ? c._id : c;
        return cId !== currentModalUser._id;
      });
    }

    // 3. Перемальовуємо кнопки та закриваємо модалку профілю
    renderModalButtons(currentModalUser);
    if (typeof closeContactModal === 'function') {
      closeContactModal();
    }
  }
}

```

```

// 4. Скидаємо глобальні змінні активного чату в нуль
currentChatId = null;
currentChat = null;

// ===== ВИКЛИК ТВОЄЇ РІДНОЇ ШИРМИ =====
if (typeof resetChatView === 'function') {
  console.log(' 🚀 Викликаємо resetChatView() для увімкнення плейсхолдера...');
  resetChatView();
}

// Забираємо мобільну активність правої панелі, якщо вона є
const chatArea = document.querySelector('.chat-area') || document.getElementById('chatArea');
if (chatArea) {
  chatArea.classList.remove('active');
}
// =====

// 5. Повністю оновлюємо списки чатів та контактів зліва (вони зникнуть з екрана)
if (typeof loadChats === 'function') await loadChats();
if (typeof loadMyContacts === 'function') await loadMyContacts();

} catch (err) {
  console.error('removeContact error:', err);
  alert(err.message);
}
}

async function showChatInfo() {
  console.log('=== STEP 1: OPENING CHAT INFO ===');

  if (!currentChat || !currentUser) {
    console.error('Current chat or current user is missing');
    return;
  }

  const members = currentChat.members || [];
  // Находим партнера по чату
  const partnerFromChat = members.find(m => m._id !== currentUser._id);

  console.log('Partner from chat:', partnerFromChat);
  if (!partnerFromChat) return;

  // Базовый объект на основе данных из чата
  let fullPartnerData = { ...partnerFromChat };

  // 1. Пытаемся получить свежие данные через относительный роут (без хардкода портов)
  try {
    const token = localStorage.getItem('token');
    if (token) {
      const res = await fetch(`/api/users/${partnerFromChat._id}`, {
        headers: {
          'Authorization': `Bearer ${token}`,
          'Content-Type': 'application/json'
        }
      });
    });

    if (res.ok) {
      const responseData = await res.json();
      console.log('Fresh data from server:', responseData);
      if (responseData.success && responseData.user) {

```

```

        fullPartnerData = responseData.user;
    }
    } else {
        console.warn(`Server returned status ${res.status}. Trying local fallback...`);
    }
}
} catch (err) {
    console.error('Failed to fetch user profiles:', err);
}

if (!fullPartnerData.bio && currentUser.contacts) {
    // Ищем в массиве контактов текущего пользователя
    const contactMatch = currentUser.contacts.find(c =>
        (c._id === partnerFromChat._id || c === partnerFromChat._id || (c && c._id === fullPartnerData._id))
    );

    if (contactMatch && contactMatch.bio) {
        console.log('Found bio in currentUser.contacts:', contactMatch.bio);
        fullPartnerData.bio = contactMatch.bio;
    }
}

console.log('FINAL PARTNER DATA FOR MODAL:', fullPartnerData);

// Определяем, является ли пользователем другом
const isFriend = Array.isArray(currentUser.contacts)
    ? currentUser.contacts.some(c => {
        const cId = (typeof c === 'object' && c !== null) ? c._id : c;
        return cId === fullPartnerData._id;
    })
    : false;

// Открываем твою оригинальную модальку
openContactModal({
    _id: fullPartnerData._id,
    displayName: fullPartnerData.displayName || partnerFromChat.displayName,
    nickname: fullPartnerData.nickname || partnerFromChat.nickname,
    tag: fullPartnerData.tag || partnerFromChat.tag,
    bio: fullPartnerData.bio, // Если био нет ни на сервере, ни в контактах — сработает заглушка "Біографія відсутня"
    avatar: fullPartnerData.avatar || partnerFromChat.avatar,
    isFriend: isFriend
});
}
// ЗАВАНТАЖЕННЯ АВАТАРА

async function uploadProfileAvatar(file) {
    const token = localStorage.getItem('token');
    if (!token) {
        alert(t('mustLoginError'));
        return;
    }

    const formData = new FormData();
    formData.append('avatar', file);

    const avatarEl = document.getElementById('profileAvatar');
    if (avatarEl) {
        avatarEl.textContent = '🕒';
    }
}

```

```

    }

    try {
        const res = await fetch('/api/users/avatar', {
            method: 'POST',
            headers: {
                Authorization: `Bearer ${token}`
            },
            body: formData
        });

        const data = await res.json();

        if (!res.ok || !data.success) {
            throw new Error(data.error || t('uploadError'));
        }

        // Показуємо нове зображення одразу
        updateProfileAvatar(data.avatar);

        // Оновлюємо localStorage
        const storedUser = JSON.parse(localStorage.getItem('user') || '{}');
        storedUser.avatar = data.avatar;
        localStorage.setItem('user', JSON.stringify(storedUser));

    } catch (err) {
        console.error(err);
        alert(err.message || t('uploadAvatarError'));
        if (avatarEl) {
            avatarEl.textContent = '👤';
        }
    }
}

// Функція відображення аватара
function updateProfileAvatar(avatarUrl) {
    const avatarEl = document.getElementById('profileAvatar');
    if (!avatarEl) return;

    if (avatarUrl) {
        avatarEl.style.backgroundImage = `url("${avatarUrl}?t=${Date.now()}")`;
        avatarEl.style.backgroundSize = 'cover';
        avatarEl.style.backgroundPosition = 'center';
        avatarEl.innerHTML = `<div class="avatar-edit-icon"></div>`;
    } else {
        avatarEl.style.backgroundImage = "";
        avatarEl.innerHTML = `👤 <div class="avatar-edit-icon"></div>`;
    }

    checkSaveButton();
}

// ===== PROFILE =====

let originalProfile = {
    tag: "",

```



```

    displayName: "",
    bio: "",
    avatar: ""
  };

  // Завантаження профілю
  async function loadProfile() {
    const token = localStorage.getItem('token');
    if (!token) return;

    try {
      const res = await fetch('/api/users/me', {
        headers: {
          Authorization: `Bearer ${token}`
        }
      });

      const user = await res.json();

      if (!res.ok) {
        throw new Error(user.error || t('profileLoadError'));
      }

      // Заповнюємо поля
      const tagInput = document.getElementById('userTag');
      const displayNameInput = document.getElementById('userDisplayName');
      const bioInput = document.getElementById('userBio');

      if (tagInput) tagInput.value = user.tag || "";
      if (displayNameInput) {
        displayNameInput.value = user.displayName || user.nickname || "";
      }
      if (bioInput) bioInput.value = user.bio || "";

      // Показуємо аватар
      if (user.avatar) {
        updateProfileAvatar(user.avatar);
      }

      // Зберігаємо початковий стан
      originalProfile = {
        tag: user.tag || "",
        displayName: user.displayName || user.nickname || "",
        bio: user.bio || "",
        avatar: user.avatar || ""
      };

      // Оновлюємо localStorage
      localStorage.setItem('user', JSON.stringify(user));

      checkSaveButton();

    } catch (err) {
      console.error('loadProfile error:', err);
    }
  }

  // Перевірка змін
  function checkSaveButton() {
    const tagInput = document.getElementById('userTag');

```

```

const displayNameInput = document.getElementById('userDisplayName');
const bioInput = document.getElementById('userBio');
const saveBtn = document.getElementById('saveProfileBtn');
const avatarEl = document.getElementById('profileAvatar');

if (!saveBtn) return;

const currentAvatar = avatarEl?.dataset?.avatar || '';

const changed =
  (tagInput?.value.trim() || '') !== originalProfile.tag ||
  (displayNameInput?.value.trim() || '') !== originalProfile.displayName ||
  (bioInput?.value.trim() || '') !== originalProfile.bio ||
  currentAvatar !== originalProfile.avatar;

saveBtn.disabled = !changed;
}

// Збереження профілю
async function saveProfile() {
  const token = localStorage.getItem('token');
  if (!token) return;

  const saveBtn = document.getElementById('saveProfileBtn');

  const payload = {
    tag: document.getElementById('userTag').value.trim(),
    displayName: document.getElementById('userDisplayName').value.trim(),
    bio: document.getElementById('userBio').value.trim()
  };

  try {
    saveBtn.disabled = true;
    saveBtn.textContent = t('saving');

    const res = await fetch('/api/users/me', {
      method: 'PUT',
      headers: {
        'Content-Type': 'application/json',
        Authorization: `Bearer ${token}`
      },
      body: JSON.stringify(payload)
    });

    const data = await res.json();

    if (!res.ok || !data.success) {
      throw new Error(data.error || t('saveError'));
    }

    const user = data.user;

    // Оновлюємо localStorage
    localStorage.setItem('user', JSON.stringify(user));

    // Оновлюємо початкові дані
    originalProfile = {
      tag: user.tag || '',
      displayName: user.displayName || user.nickname || '',
      bio: user.bio || ''
    };
  }
}

```

```

        avatar: user.avatar || "
    };

    checkSaveButton();

    saveBtn.textContent = t('savedSuccess');

    setTimeout(() => {
        saveBtn.textContent = t('saveBtn');
        checkSaveButton();
    }, 1500);

} catch (err) {
    console.error('saveProfile error:', err);
    alert(err.message);
    saveBtn.textContent = t('saveBtn');
    checkSaveButton();
}
}

// Ініціалізація слухачів
function initProfileListeners() {
    const tagInput = document.getElementById('userTag');
    const displayNameInput = document.getElementById('userDisplayName');
    const bioInput = document.getElementById('userBio');
    const saveBtn = document.getElementById('saveProfileBtn');

    if (!tagInput || tagInput.dataset.initialized) return;

    tagInput.addEventListener('input', checkSaveButton);
    displayNameInput.addEventListener('input', checkSaveButton);
    bioInput.addEventListener('input', checkSaveButton);
    saveBtn.addEventListener('click', saveProfile);

    tagInput.dataset.initialized = 'true';
}

// ===== MESSENGER =====

// Основна функція відкриття чату
async function openChat(userId) {
    if (!userId || !currentUser) return;

    const token = localStorage.getItem('token');
    if (!token) return;

    try {
        const res = await fetch('/api/chats', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
                Authorization: `Bearer ${token}`
            },
            body: JSON.stringify({ userId })
        });

        const data = await res.json();
        if (!data.success || !data.chat) throw new Error(t('openChatError'));
    }

```

```

currentChat = data.chat;
currentChatId = data.chat._id;

const otherUser = currentChat.members.find(m => m._id !== currentUser._id);
if (!otherUser) return;

// === ТІЛЬКИ ДЛЯ ЦЬОГО ЧАТУ оновлюємо timestamp ===
const lastMsgTime = data.chat.lastMessage?.createdAt
  ? new Date(data.chat.lastMessage.createdAt).getTime()
  : Date.now();

lastReadTimestamps[currentChatId] = lastMsgTime;
saveLastReadTimestamps();

console.log(`☑ Чат ${currentChatId} відкрито і позначено як прочитаний`);

// UI
const chatHeader = document.getElementById('chatHeader');
if (chatHeader) chatHeader.classList.add('active');

const chatHeaderName = document.getElementById('chatHeaderName');
if (chatHeaderName) {
  chatHeaderName.textContent = otherUser.displayName || otherUser.nickname || t('chat');
}

const avatarEl = document.getElementById('chatAvatar');
if (avatarEl) {
  if (otherUser.avatar) {
    avatarEl.style.backgroundImage = `url("${otherUser.avatar}?t=${Date.now()}")`;
    avatarEl.textContent = "";
  } else {
    avatarEl.style.backgroundImage = "";
    avatarEl.textContent = '👤';
  }
}

hideNoChatPlaceholder();
await loadChatMessages(currentChatId);
showMediaForCurrentChat();

loadChats();

} catch (err) {
  console.error('Open chat error:', err);
}
}

// Обробка нового повідомлення
function handleNewMessage(chatId, messageTimestamp = Date.now()) {
  if (!chatId) return;

  // Если новое сообщение пришло в текущий открытый чат
  if (chatId === currentChatId) {
    lastReadTimestamps[chatId] = Math.max(
      lastReadTimestamps[chatId] || 0,
      messageTimestamp
    );
    saveLastReadTimestamps();

    if (typeof removeNewBadgeFromChat === 'function') {

```

```

        removeNewBadgeFromChat(chatId);
    }

    if (typeof loadMediaForChat === 'function') {
        loadMediaForChat(currentChatId);
    }
}

// Откладаємо оновлення списку чатів, щоб DOM успів отримати
setTimeout(() => {
    if (typeof loadChats === 'function') loadChats();
}, 80);
}

// Позначити чат як прочитаний
async function markChatAsRead(chatId) {
    if (!chatId) return;

    const token = localStorage.getItem('token');
    try {
        await fetch(`/api/chats/${chatId}/read`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
                Authorization: `Bearer ${token}`
            }
        });
    } catch (e) {
        console.error('Failed to mark chat as read:', e);
    }
}

// Допоміжна функція
function removeNewBadgeFromChat(chatId) {
    if (!chatId) return;

    const chatItems = document.querySelectorAll(`.chat-item[data-chat-id="${chatId}"]`);
    chatItems.forEach(item => {
        const badge = item.querySelector('.new-badge');
        if (badge) badge.remove();
    });

    // Також прибираємо з активного чату
    const activeItem = document.querySelector('.chat-item.active');
    if (activeItem) {
        const badge = activeItem.querySelector('.new-badge');
        if (badge) badge.remove();
    }
}

// Нова допоміжна функція
function removeNewBadgeFromCurrentChat() {
    const activeChatItem = document.querySelector('.chat-item.active');
    if (activeChatItem) {
        const newBadge = activeChatItem.querySelector('.new-badge');
        if (newBadge) newBadge.remove();
    }
}

// Скидання стану чату (коли повертаємося до "виберіть чат")

```

```

function resetChatView() {
  const chatHeader = document.getElementById('chatHeader');
  if (chatHeader) chatHeader.classList.remove('active');

  const placeholder = document.getElementById('noChatPlaceholder');
  if (placeholder) {
    placeholder.classList.add('show'); // використовуємо клас
    placeholder.style.display = 'flex';
  }

  const messages = document.getElementById('messages');
  if (messages) messages.style.display = 'none';

  const inputArea = document.getElementById('messageInputArea');
  if (inputArea) inputArea.style.display = 'none';

  // Медіа
  const mediaPlaceholder = document.getElementById('noChatMediaPlaceholder');
  if (mediaPlaceholder) mediaPlaceholder.style.display = 'flex';

  const mediaContent = document.getElementById('mediaContent');
  if (mediaContent) mediaContent.style.display = 'none';
}

// Завантаження повідомлень конкретного чату
async function loadChatMessages(chatId) {
  console.log('LOAD CHAT:', chatId);
  console.log('messagesContainer:', messagesContainer);

  if (!chatId || !messagesContainer) return;

  messagesContainer.innerHTML = `
    <div style="text-align:center;padding:60px;color:#64748b;">
      ${t('loading')}
    </div>
  `;

  try {
    const res = await fetch(`/api/messages?chatId=${chatId}`, {
      headers: {
        Authorization: `Bearer ${localStorage.getItem('token')}`
      }
    });

    const data = await res.json();

    messagesContainer.innerHTML = "";

    if (data.success && data.messages && data.messages.length > 0) {

      data.messages.forEach(msg => renderMessage(msg));

      // === ПОЗНАЧАЄМО ЧАТ ПРОЧИТАНИМ ===
      const lastMsg = data.messages[data.messages.length - 1];

      if (lastMsg.createdAt) {
        lastReadTimestamps[chatId] = new Date(lastMsg.createdAt).getTime();
        saveLastReadTimestamps();
      }
    }
  }
}

```

```

        scrollToBottom();

    } catch (err) {
        console.error(err);

        messagesContainer.innerHTML = `
            <div style="color:#ef4444;text-align:center;padding:60px;">
                ${t('profileLoadError')}
            </div>
        `;
    }
}

// Рендер повідомлення
function renderMessage(msg) {
    const emptyState = messagesContainer.querySelector('.empty-state, .no-messages');

    if (emptyState) {
        emptyState.remove();
    }

    if (!msg || !messagesContainer) {
        console.log('MSG EMPTY', msg);
        return;
    }

    console.log('RENDER MSG:', msg);

    const senderId =
        msg.sender?._id ||
        msg.senderId?._id ||
        msg.senderId ||
        msg.sender;

    const isMine = String(senderId) === String(currentUser?._id);

    const div = document.createElement('div');
    div.className = `message ${isMine ? 'sent' : 'received'}`;

    // Динамічна локаль для часу на основі обраної мови (за замовчуванням 'uk')
    const currentLang = typeof getCurrentLanguage === 'function' ? getCurrentLanguage() : 'uk';
    const time = msg.createdAt
        ? new Date(msg.createdAt).toLocaleTimeString(currentLang, {
            hour: '2-digit',
            minute: '2-digit'
        })
        : "";

    let html = "";

    if (msg.text) {
        html += `<div class="message-text">${msg.text}</div>`;
    }

    const file = msg.file || null;

    if (file?.url) {
        // Додаємо timestamp, щоб браузер не кешував 404
        let fileUrl = file.url;

```

```

    if (!fileUrl.includes('t=')) {
        fileUrl += (fileUrl.includes('?') ? '&' : '?') + 't=' + Date.now();
    }

    if (file.mimetype?.startsWith('image/')) {
        html += ``;
    }
    else if (file.mimetype?.startsWith('video/')) {
        html += `
        <video
            src="${fileUrl}"
            controls
            class="message-video"
            preload="metadata"
            onclick="openLightboxFromMessage(this, event)">
        </video>`;
    }
    else {
        const icon = getFileIcon((file?.originalName || file?.name || t('file')) || "");

        html += `
        <a href="${fileUrl}" download class="message-file">
            <span class="file-icon">${icon}</span>
            <span class="file-name">
                ${file?.originalName || file?.name || t('file')}
            </span>
        </a>
        `;
    }
}

html += `<span class="time">${time}</span>`;

div.innerHTML = html;
messagesContainer.appendChild(div);
scrollToBottom();
}

function scrollToBottom() {
    if (messagesContainer) {
        setTimeout(() => {
            messagesContainer.scrollTop = messagesContainer.scrollHeight;
        }, 100);
    }
}

// Відкриття lightbox з повідомлення
function openLightboxFromMessage(element, event) {
    if (event) event.preventDefault?().();
    if (event) event.stopPropagation?().();

    pauseAllVideos();

    if (element && element.tagName === 'VIDEO') {
        element.pause();
        element.currentTime = 0;
    }
}

```



```

const url = element.src || element.currentSrc;
const isVideo = element.tagName === 'VIDEO';

const items = [{
  url: url,
  mimetype: isVideo ? 'video/mp4' : 'image/jpeg',
  originalName: 'media'
}];

openLightbox(items, 0);
}

// Завантаження списку чатів
async function loadChats() {
  const chatList = document.getElementById('chatList');
  if (!chatList) return;

  const token = localStorage.getItem('token');
  if (!token) return;

  try {
    const res = await fetch('/api/chats', {
      cache: 'no-store',
      headers: {
        Authorization: `Bearer ${token}`
      }
    });
    const data = await res.json();
    chatList.innerHTML = "";

    if (!data.success || !data.chats.length) {
      chatList.innerHTML = `<div style="text-align:center;padding:60px;color:#64748b;">
<p>${t('noChats')}</p></div>`;
      return;
    }

    data.chats.forEach(chat => {
      const otherUser = chat.members.find(m => m._id !== currentUser._id);
      if (!otherUser) return;

      const isActive = chat._id === currentChatId;

      const last = chat.lastMessage || {};
      let lastMsgText = t('writeFirst');
      let lastMsgTime = "";

      if (last.text) {
        lastMsgText = last.text;
      } else if (last.file) {
        lastMsgText = last.file.mimetype?.startsWith('image/') ? `🖼️ ${t('photo')}` :
          last.file.mimetype?.startsWith('video/') ? `🎥 ${t('video')}` : `📎 ${t('file')}`;
      }

      const currentLang = typeof getLanguage === 'function' ? getLanguage() : 'uk';
      if (last.createdAt) {
        lastMsgTime = new Date(last.createdAt).toLocaleTimeString(currentLang, {
          hour: '2-digit', minute: '2-digit'
        });
      }
    });
  }
}

```

```

// === ПОЗУМНА ЛОГІКА NEW ===
const lastReadTime = lastReadTimestamps[chat._id] || 0;
const lastMessageTime = last.createdAt ? new Date(last.createdAt).getTime() : 0;

const lastSenderId =
  last.sender?._id ||
  last.senderId?._id ||
  last.senderId ||
  last.sender;

const isMyMessage = String(lastSenderId) === String(currentUser?._id);

const hasNewMessage = lastMessageTime > lastReadTime && !isMyMessage;
const showNewBadge = hasNewMessage && !isActive;

// === ДІАГНОСТИКА ===
console.log(` 🗨 Chat ${chat._id.slice(-6)} | NEW: ${hasNewMessage} | lastMsg: ${lastMessageTime} |
lastRead: ${lastReadTime} | isMy: ${isMyMessage} | active: ${isActive}`);

const chatItem = document.createElement('div');
chatItem.className = `chat-item ${isActive ? 'active' : ''}`;
chatItem.setAttribute('data-chat-id', chat._id);

chatItem.innerHTML = `
  <div class="avatar" style="${otherUser.avatar ? `background-
image:url('${otherUser.avatar}')?t=${Date.now()}` : ''}">
    ${!otherUser.avatar ? '👤' : ''}
  </div>

  <div class="chat-preview">
    <div class="chat-name">${otherUser.displayName || otherUser.nickname}</div>
    <div class="last-message">${lastMsgText}</div>
  </div>

  <div class="chat-meta">
    <span class="chat-time">${lastMsgTime}</span>
    ${showNewBadge ? `<span class="new-badge">${t('newBadge')}</span>` : ''}
  </div>
`;

chatItem.addEventListener('click', () => openChat(otherUser._id));
chatList.appendChild(chatItem);
});

} catch (err) {
  console.error('Load chats error:', err);
}
}

// Переключення вкладок медіа
let currentMediaTab = "photos";

function switchMediaTab(tab) {
  const slider = document.getElementById("mediaSlider");
  console.log("TAB INPUT:", tab);
  if (!slider) return;

  const tabs = document.querySelectorAll(".tab");

```

```

// нормалізуємо вхід (ВАЖНО)
tab = (tab || "").trim().toLowerCase();

let foundTab = null;

tabs.forEach(t => {
  t.classList.remove("active");
  const tName = (t.dataset.tab || "").trim().toLowerCase();
  if (tName === tab) {
    foundTab = t;
  }
});

if (foundTab) {
  foundTab.classList.add("active");
} else {
  console.warn("TAB NOT FOUND:", tab);
  return;
}

const indexMap = {
  photos: 0,
  videos: 1,
  files: 2
};

slider.style.transform = `translateX(-${(indexMap[tab] ?? 0) * 100}%)`;
}

// Показати медіа панель для поточного чату
function showMediaForCurrentChat() {
  const mediaTabs = document.getElementById('mediaTabs');
  const mediaContent = document.getElementById('mediaContent');
  const placeholder = document.getElementById('noChatMediaPlaceholder');

  if (mediaTabs) mediaTabs.style.display = 'flex';
  if (placeholder) placeholder.style.display = 'none';
  if (mediaContent) mediaContent.style.display = 'flex';

  loadMediaForChat(currentChatId);
}

async function loadMediaPreview() {
  console.log('Медіа для чата', currentChatId, 'загружается...');
}

// Завантаження медіа для поточного чату
async function loadMediaForChat(chatId) {
  if (!chatId) return;

  try {
    const res = await fetch(`/api/messages?chatId=${chatId}`, {
      headers: { Authorization: `Bearer ${localStorage.getItem('token')}` }
    });

    const data = await res.json();
    if (!data.success || !data.messages) return;

    const photos = [];
    const videos = [];

```

```

const files = [];

data.messages.forEach(msg => {
  if (!msg.file?.url) return;

  const fileInfo = {
    url: msg.file.url + (msg.file.url.includes('?') ? '&' : '?') + 't=' + Date.now(),
    originalName: msg.file.originalName || t('file'),
    mimetype: msg.file.mimetype,
    size: msg.file.size
  };

  if (msg.file.mimetype.startsWith('image/')) photos.push(fileInfo);
  else if (msg.file.mimetype.startsWith('video/')) videos.push(fileInfo);
  else files.push(fileInfo);
});

renderMediaGrid('photosGrid', photos, 'image');
renderMediaGrid('videosGrid', videos, 'video');
renderFilesList('filesList', files);

} catch (err) {
  console.error('Media load error:', err);
}
}

// Допоміжна функція для рендера медіа-панелі
function renderMediaGrid(containerId, items, type) {
  const container = document.getElementById(containerId);
  if (!container) return;

  container.innerHTML = "";

  if (items.length === 0) {
    // Просто берем перевод из t(). Если ключа нет, t() сама вернет имя ключа как фоллбек.
    const noMediaText = type === 'image' ? t('noPhotos') : t('noVideos');
    container.innerHTML = `
      <div style="padding:60px 20px; text-align:center; color:#64748b;">
        <div style="font-size:42px; margin-bottom:12px;"> 📭 </div>
        ${noMediaText}
      </div>`;
    return;
  }

  items.forEach((item, index) => {
    const el = document.createElement('div');
    el.className = 'media-item';

    if (type === 'image') {
      el.innerHTML = ``;
    } else {
      el.innerHTML = `
        <video src="${item.url}" muted></video>
        <div class="media-overlay">▶</div>
      `;
    }

    // Відкриваємо lightbox
    el.addEventListener('click', () => {
      openLightbox(items, index);
    });
  });
}

```

```

    });

    container.appendChild(el);
  });
}

function renderFilesList(containerId, items) {
  const container = document.getElementById(containerId);
  if (!container) return;

  container.innerHTML = "";

  if (items.length === 0) {

    container.innerHTML = `<div style="padding:20px; color:#64748b;">${t('noFiles')}</div>`;
    return;
  }

  items.forEach(item => {
    const div = document.createElement('div');
    div.className = 'file-item';

    const icon = getFileIcon(item.originalName || item.name || "");

    div.innerHTML = `
      <div class="file-icon">${icon}</div>
      <div class="file-info">
        <div class="file-name">${item.originalName}</div>
        <small class="file-time">${item.time || ""}</small>
      </div>
    `;

    container.appendChild(div);
  });
}

function getLastMessagePreview(lastMessage) {

  if (!lastMessage) return t('noMessages');

  switch (lastMessage.type) {
    case "image":
      return `🖼️ ${t('photos')}`;
    case "video":
      return `🎥 ${t('videos')}`;
    case "file":
      return `📁 ${t('files')}`;
    default:
      return lastMessage.text || "";
  }
}

function getFileIcon(filename) {
  const ext = filename.split('.').pop().toLowerCase();

  switch (ext) {
    case "zip":
    case "rar":
    case "7z":
      return "📦 ";
  }
}

```

```

    case "xls":
    case "xlsx":
    case "csv":
        return "📊";
    case "doc":
    case "docx":
        return "📄";
    case "txt":
        return "📝";
    case "pdf":
        return "📖";
    case "mp3":
    case "wav":
        return "🎵";
    case "mp4":
    case "mov":
        return "🎬";
    case "jpg":
    case "jpeg":
    case "png":
        return "🖼️";
    case "js":
    case "ts":
    case "html":
    case "css":
        return "💻";
    default:
        return "📁";
}
}

function showFilePreview(file) {
    const previewContainer = document.getElementById('filePreviewContainer');
    const previewName = document.getElementById('filePreviewName');
    const fileIcon = previewContainer.querySelector('.file-icon');

    if (!previewContainer || !previewName) return;

    previewName.textContent = file.name;
    fileIcon.textContent = getFileIcon(file.name);

    previewContainer.style.display = 'flex';
}

function clearFilePreview() {
    const previewContainer = document.getElementById('filePreviewContainer');
    const fileInput = document.getElementById('messageFile');

    if (fileInput) fileInput.value = '';
    if (previewContainer) previewContainer.style.display = 'none';
}

// Инициализация превью файла
function initFilePreview() {
    const fileInput = document.getElementById('messageFile');
    if (!fileInput) return;

    // Убираем старые слушатели, чтобы не дублировались
    fileInput.removeEventListener('change', handleFileSelect);

```

```

    fileInput.addEventListener('change', handleFileSelect);
}

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

    showFilePreview(file);
}

function getLastMessageText(chat) {
    const msg = chat.lastMessage;

    if (!msg) return 'Немає повідомлень';

    if (msg.text) return msg.text;

    if (msg.file) {
        if (msg.file.mimetype.startsWith('image/')) return '🖼️ Фото';
        if (msg.file.mimetype.startsWith('video/')) return '📺 Відео';
        return '📎 Файл';
    }

    return 'Повідомлення';
}

// ===== MAIN APP INITIALIZATION =====
async function initializeApp() {
    if (appInitialized) return;
    appInitialized = true;

    const token = localStorage.getItem('token');
    hideAllAuth();

    if (!token) {
        showLogin();
        initMenu();
        return;
    }

    try {
        const user = await loadCurrentUser();
        if (!user) throw new Error('No user');

        loadLastReadTimestamps(); // ← завантажується для поточного користувача

        showMessenger();
        loadMyContacts();
    } catch (err) {
        console.error('Auto login failed:', err);
        localStorage.clear();
        showLogin();
    }
}

let chatsPolling = null;

function startChatsPolling() {

```

```

    if (chatsPolling) {
      clearInterval(chatsPolling);
    }

    chatsPolling = setInterval(() => {
      // Не завантажувемо, якщо користувач не авторизований
      if (!localStorage.getItem('token')) return;

      loadChats();
    }, 3000); // кожні 3 секунди
  }

// ===== SOCKET.IO =====
let socket = null;

function initSocket() {
  if (typeof io === 'undefined') {
    console.warn('⚠ Socket.io не підключено (io is not defined)');
    return;
  }

  if (socket) socket.disconnect();

  socket = io('/', {
    auth: { token: localStorage.getItem('token') },
    reconnection: true,
    timeout: 10000
  });

  socket.on('connect', () => {
    console.log('✅ Socket connected successfully');
    if (currentUser?._id) {
      socket.emit('join', currentUser._id);
    }
  });

  socket.on('newMessage', (message) => {
    console.log('✉ New message via socket:', message);
    const chatId = message.chatId || message.chat?._id;
    if (chatId === currentChatId) {
      renderMessage(message);
      scrollToBottom();
    }
    handleNewMessage(chatId, new Date(message.createdAt).getTime());
  });

  socket.on('connect_error', (err) => {
    console.error('Socket connection error:', err.message);
  });
}

// ===== LIGHTBOX GALLERY =====
let currentMediaItems = [];
let currentMediaIndex = 0;

function openLightbox(items, startIndex = 0) {
  pauseAllVideos(); // Зупиняємо всі фонові відео в чаті

  currentMediaItems = items;
  currentMediaIndex = startIndex;
}

```



```

const lightbox = document.getElementById('mediaLightbox');
const container = document.getElementById('lightboxMediaContainer');

container.innerHTML = "";
showCurrentMedia();

lightbox.classList.add('show');
document.body.style.overflow = 'hidden';
}

function showCurrentMedia() {
const container = document.getElementById('lightboxMediaContainer');
const item = currentMediaItems[currentMediaIndex];

container.innerHTML = "";

if (item.mimetype.startsWith('image/')) {
const img = document.createElement('img');
img.src = item.url;
container.appendChild(img);
} else if (item.mimetype.startsWith('video/')) {
const video = document.createElement('video');
video.src = item.url;
video.controls = true;
video.autoplay = true;
video.loop = false;
container.appendChild(video);
}
}

function closeLightbox() {
pauseAllVideos(); // Зупиняємо плеєр всередині lightbox при закритті

const lightbox = document.getElementById('mediaLightbox');
lightbox.classList.remove('show');
document.body.style.overflow = 'auto';
}

function prevMedia() {
if (currentMediaIndex > 0) {
currentMediaIndex--;
showCurrentMedia();
}
}

function nextMedia() {
if (currentMediaIndex < currentMediaItems.length - 1) {
currentMediaIndex++;
showCurrentMedia();
}
}

function downloadCurrentMedia() {
const item = currentMediaItems[currentMediaIndex];
const link = document.createElement('a');
link.href = item.url;
link.download = item.originalName || 'media';
link.click();
}

```

```

function goToMessageFromLightbox() {
  // Використовуємо локалізоване повідомлення про майбутній функціонал
  alert(t('featureInDevelopment') || 'Функціонал у розробці');
  closeLightbox();
}

// Закриття галереї по натисканню Escape
document.addEventListener('keydown', (e) => {
  if (e.key === 'Escape') {
    const lightbox = document.getElementById('mediaLightbox');
    if (lightbox && lightbox.classList.contains('show')) closeLightbox();
  }
});

// ===== VIDEO CONTROL =====

// Зупиняємо абсолютно всі відеоелементи на сторінці
function pauseAllVideos() {
  document.querySelectorAll('video').forEach(video => {
    video.pause();
    video.currentTime = video.currentTime;
  });
}

function resumeAllVideos() {
  // Резервна функція відновлення відтворення (якщо знадобиться автоплей)
}

// ===== THEME =====

function applyTheme(theme) {
  if (theme === 'light') {
    document.body.classList.add('light-theme');
  } else {
    document.body.classList.remove('light-theme');
  }
}

// Получаем userId
const user =
  JSON.parse(localStorage.getItem('user') || '{}');

const userId = user._id;

// Сохраняем тему отдельно для каждого юзера
if (userId) {
  localStorage.setItem(
    `chatty_theme_${userId}`,
    theme
  );
}

const toggle = document.getElementById('themeToggle');

if (toggle) {
  toggle.checked = theme === 'light';
}
}

```

```

function loadSavedTheme() {
  const user =
    JSON.parse(localStorage.getItem('user') || '{}');

  if (!user._id) {
    applyTheme('dark');
    return;
  }

  const savedTheme =
    localStorage.getItem(`chatty_theme_${user._id}`) || 'dark';

  applyTheme(savedTheme);
}

// ===== СМЕНА ПАРОЛЯ (ЛОГИКА ШАГОВ) =====

// 1. Открытие модального окна
window.openPasswordModal = function() {
  // Сбрасываем инпуты
  document.getElementById('oldPasswordInput').value = '';
  document.getElementById('newPasswordInput').value = '';
  document.getElementById('confirmPasswordInput').value = '';

  // Скрываем алерты
  const alertBox = document.getElementById('passwordModalAlert');
  alertBox.style.display = 'none';
  alertBox.textContent = '';
  alertBox.style.color = '#ef4444'; // возвращаем красный цвет по дефолту

  // Показываем Шаг 1 и скрываем Шаг 2
  document.getElementById('passwordStep1').style.display = 'block';
  document.getElementById('passwordStep2').style.display = 'none';

  // Показываем само модальное окно (используем тот же класс анимации, что и для контактов)
  document.getElementById('passwordModal').classList.add('show');
};

// 2. Закрытие модального окна
window.closePasswordModal = function() {
  document.getElementById('passwordModal').classList.remove('show');
};

// 3. Шаг 1: Проверка старого пароля на бэкенде
window.verifyOldPassword = async function() {
  const oldPassword = document.getElementById('oldPasswordInput').value.trim();
  const alertBox = document.getElementById('passwordModalAlert');

  if (!oldPassword) {
    alertBox.textContent = 'Будь ласка, введіть поточний пароль';
    alertBox.style.display = 'block';
    return;
  }

  try {
    const token = localStorage.getItem('token');

    const res = await fetch('/api/users/change-password/verify', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',

```

```

        'Authorization': `Bearer ${token}`
      },
      body: JSON.stringify({ oldPassword })
    });

    const data = await res.json();

    if (!data.success) {
      alertBox.textContent = data.error || 'Невірний старий пароль';
      alertBox.style.display = 'block';
      return;
    }

    // Если пароль верный, скрываем ошибки и переключаем шаг
    alertBox.style.display = 'none';
    document.getElementById('passwordStep1').style.display = 'none';
    document.getElementById('passwordStep2').style.display = 'block';

  } catch (err) {
    console.error('Verify password error:', err);
    alertBox.textContent = 'Помилка з'єднання з сервером';
    alertBox.style.display = 'block';
  }
};

// 4. Шаг 2: Валидация и сохранение нового пароля
window.saveNewPassword = async function() {
  const oldPassword = document.getElementById('oldPasswordInput').value.trim();
  const newPassword = document.getElementById('newPasswordInput').value.trim();
  const confirmPassword = document.getElementById('confirmPasswordInput').value.trim();
  const alertBox = document.getElementById('passwordModalAlert');

  if (!newPassword || !confirmPassword) {
    alertBox.style.color = '#ef4444';
    alertBox.textContent = 'Заповніть усі поля';
    alertBox.style.display = 'block';
    return;
  }

  if (newPassword.length < 6) {
    alertBox.style.color = '#ef4444';
    alertBox.textContent = 'Новий пароль має бути не менше 6 символів';
    alertBox.style.display = 'block';
    return;
  }

  if (newPassword !== confirmPassword) {
    alertBox.style.color = '#ef4444';
    alertBox.textContent = 'Паролі не збігаються';
    alertBox.style.display = 'block';
    return;
  }

  try {
    const token = localStorage.getItem('token');

    const res = await fetch('/api/users/change-password/save', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',

```

```

        'Authorization': `Bearer ${token}`
      },
      body: JSON.stringify({ oldPassword, newPassword })
    });

    const data = await res.json();

    if (!data.success) {
      alertBox.style.color = '#ef4444';
      alertBox.textContent = data.error || 'Помилка зміни пароля';
      alertBox.style.display = 'block';
      return;
    }

    alertBox.style.color = '#10b981';
    alertBox.textContent = 'Пароль успішно змінено!';
    alertBox.style.display = 'block';

    setTimeout(() => {
      closePasswordModal();
    }, 1500);

  } catch (err) {
    console.error('Save password error:', err);
    alertBox.style.color = '#ef4444';
    alertBox.textContent = 'Помилка при збереженні пароля';
    alertBox.style.display = 'block';
  }
};

function initThemeToggle() {
  const toggle = document.getElementById('themeToggle');

  if (!toggle || toggle.dataset.initialized) return;

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

  const savedTheme =
    localStorage.getItem(`chatty_theme_${user._id}`) || 'dark';

  toggle.checked = savedTheme === 'light';

  toggle.addEventListener('change', () => {
    applyTheme(toggle.checked ? 'light' : 'dark');
  });

  toggle.dataset.initialized = 'true';
}

// ===== ГЛОБАЛЬНЫЕ ФУНКЦИИ ДЛЯ HTML onclick =====
window.login = login;
window.showLogin = showLogin;
window.showRegister = showRegister;
window.goToCodeVerification = goToCodeVerification;
window.verifyCode = verifyCode;

```

```
window.finishRegistration = finishRegistration;
window.logout = logout;
window.startChatWith = startChatWith;
window.addContact = addContact;
window.removeContact = removeContact;
window.closeContactModal = closeContactModal;
window.showChatInfo = showChatInfo;
window.closeContactModal = closeContactModal;
window.openPasswordModal = openPasswordModal;
window.closePasswordModal = closePasswordModal;
window.verifyOldPassword = verifyOldPassword;
window.saveNewPassword = saveNewPassword;
```

```
window.showRecoveryStep1 = showRecoveryStep1;
window.showRecoveryStep2 = showRecoveryStep2;
window.showRecoveryStep3 = showRecoveryStep3;
window.sendRecoveryEmail = sendRecoveryEmail;
window.verifyRecoveryCode = verifyRecoveryCode;
window.submitNewPassword = submitNewPassword;
```

```
// ===== START APP =====
document.addEventListener('DOMContentLoaded', () => {
  initializeApp();
  initMenu();
});
```

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

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGO_URI);

    console.log(`✅ MongoDB підключён: ${conn.connection.host}`);
    return conn;
  } catch (error) {
    console.error("❌ Ошибка подключения к MongoDB:", error.message);
    process.exit(1); // завершаем процесс при ошибке
  }
};

module.exports = connectDB;
```

ДОДАТОК Д

```
const Message = require('../models/Message');
const upload = require('../middleware/upload'); // или где у тебя multer

exports.getMessages = async (req, res) => {
  try {
    const messages = await Message.find().sort({ createdAt: 1 });
    res.json(messages);
  } catch (err) {
    res.status(500).json({ success: false, message: err.message });
  }
};

exports.sendMessage = [
  upload.single('file'),
  async (req, res) => {
    try {
      const { text } = req.body;
      const file = req.file;

      const messageData = {
        text: text || "",
        sender: null, // позже добавим auth
        chatId: 'general'
      };

      if (file) {
        const folder = file.mimetype.startsWith('image/') ? 'images' :
          file.mimetype.startsWith('video/') ? 'videos' : 'files';

        messageData.file = {
          url: `/uploads/chats/general/${folder}/${file.filename}`,
          originalName: file.originalname,
          mimetype: file.mimetype,
          size: file.size
        };
      }

      const message = await Message.create(messageData);
      res.status(201).json({ success: true, message });
    } catch (err) {
      console.error(err);
      res.status(500).json({ success: false, message: 'Ошибка при отправке' });
    }
  }
];
```



```

const jwt = require('jsonwebtoken');

module.exports = function (req, res, next) {
  console.log('🔑 === AUTH MIDDLEWARE STARTED ===');
  console.log('Method:', req.method, 'URL:', req.url);
  console.log('Headers:', req.headers);

  const header = req.headers.authorization;

  if (!header) {
    console.log('❌ No Authorization header found');
    return res.status(401).json({ error: 'No token provided' });
  }

  const token = header.split(' ')[1];

  if (!token) {
    console.log('❌ Token not found after "Bearer"');
    return res.status(401).json({ error: 'No token provided' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.userId = decoded.id;

    console.log('✅ Auth SUCCESS');
    console.log('User ID:', req.userId);
    console.log('=== AUTH MIDDLEWARE FINISHED ===\n');

    next();
  } catch (err) {
    console.error('❌ JWT verification failed:', err.message);
    return res.status(401).json({ error: 'Invalid token' });
  }
};

```

```

const multer = require('multer');
const fs = require('fs');
const path = require('path');
const iconv = require('iconv-lite');

const ROOT = path.resolve(__dirname, '..', '..');

const storage = multer.diskStorage({

  destination: (req, file, cb) => {

    const chatId = req.query.chatId;

    if (!chatId) {
      return cb(new Error('chatId missing'));
    }

    let folder = 'files';

    if (file.mimetype.startsWith('image/')) {
      folder = 'images';
    } else if (file.mimetype.startsWith('video/')) {
      folder = 'videos';
    }

    const dir = path.join(
      ROOT,
      'uploads',
      'chats',
      chatId,
      folder
    );
    fs.mkdirSync(dir, { recursive: true });
    console.log('📁 SAVE:', dir);
    cb(null, dir);
  },
  filename: (req, file, cb) => {
    const originalName = Buffer.from(file.originalname, 'latin1').toString('utf8');
    const name =
      Date.now() +
      '-' +
      Math.round(Math.random() * 1e9) +
      path.extname(file.originalname);

    cb(null, name);
  }
});

module.exports = multer({
  storage,
  limits: {
    fileSize: 50 * 1024 * 1024
  }
});

```

```

const mongoose = require('mongoose');

const chatSchema = new mongoose.Schema(
{
  members: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'User'
    }
  ],

  isGroup: {
    type: Boolean,
    default: false
  },

  name: {
    type: String,
    default: ""
  },

  lastMessage: {
    text: {
      type: String,
      default: ""
    },

    file: {
      mimetype: String
    },

    createdAt: {
      type: Date,
      default: null
    }
  },
},
{
  timestamps: true
});

module.exports = mongoose.model('Chat', chatSchema);

```

ДОДАТОК К

```
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');
const userSchema = new mongoose.Schema({
  nickname: {
    type: String,
    required: true,
    unique: true,
    trim: true
  },
  displayName: {
    type: String,
    default: ""
  },
  tag: {
    type: String,
    unique: true,
    trim: true,
    default: ""
  },
  email: {
    type: String,
    required: true,
    unique: true,
    trim: true,
    lowercase: true
  },
  phone: {
    type: String,
    default: ""
  },
  password: {
    type: String,
    required: true
  },
  avatar: {
    type: String,
    default: ""
  },
  bio: {
    type: String,
    default: ""
  },
  contacts: [ {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User'
  } ],
  createdAt: {
    type: Date,
    default: Date.now
  }
});
userSchema.pre('save', async function () {
  if (!this.isModified('password')) return;
  this.password = await bcrypt.hash(this.password, 10);
});
module.exports = mongoose.model('User', userSchema);
```

```

const sendCode = require('../utils/sendMail');

// Хранилище кодов: Map(email => { code, expiresAt, purpose })
const verificationCodes = new Map();

const verificationService = {
  // Создать и отправить код
  async createCode(email, purpose = 'register', lang = 'uk') {
    const code = Math.floor(100000 + Math.random() * 900000).toString(); // 6 цифр
    const expiresAt = Date.now() + 10 * 60 * 1000; // 10 минут

    verificationCodes.set(email, { code, expiresAt, purpose });

    // Передаем тип действия и язык в модуль отправки
    await sendCode(email, code, purpose, lang);
    return code;
  },

  // Проверить код с привязкой к конкретной цели
  verifyCode(email, code, purpose = 'register') {
    const data = verificationCodes.get(email);

    if (!data) return { success: false, message: "Код не знайдено" };

    // Защита от подмены назначения кода
    if (data.purpose !== purpose) {
      return { success: false, message: "Невірне призначення коду" };
    }

    if (Date.now() > data.expiresAt) {
      verificationCodes.delete(email);
      return { success: false, message: "Код прострочено" };
    }

    if (data.code !== code) {
      return { success: false, message: "Невірний код" };
    }

    // Код верный, но для сброса пароля мы удалим его только ПОСЛЕ сохранения нового пароля.
    // Для регистрации — удаляем сразу.
    if (purpose === 'register') {
      verificationCodes.delete(email);
    }

    return { success: true };
  },

  // Удалить код из памяти вручную (вызывается после успешного сброса пароля)
  removeCode(email) {
    verificationCodes.delete(email);
  }
};

module.exports = verificationService;

```

```

const router = require('express').Router();
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const User = require('../models/User');
const verificationService = require('../services/verificationService');

// РЕГИСТРАЦИЯ — ШАГ 1
router.post('/register', async (req, res) => {
  try {
    const { nickname, displayName, email, phone, lang } = req.body;
    const clientLang = lang === 'en' ? 'en' : 'uk';

    if (!nickname || !email) {
      return res.status(400).json({ error: "Нікнейм та Email обов'язкові" });
    }

    const exists = await User.findOne({ $or: [{ email: email.toLowerCase().trim() }, { nickname: nickname.trim() }] });
    if (exists) {
      return res.status(400).json({ error: "Такий користувач вже існує" });
    }

    // Передаємо цель 'register' і мову
    await verificationService.createCode(email.toLowerCase().trim(), 'register', clientLang);

    res.json({ success: true, message: "Код відправлено", email });
  } catch (err) {
    console.error("Register error:", err);
    res.status(500).json({ error: "Внутрішня помилка сервера" });
  }
});

// ПОДТВЕРЖДЕНИЕ КОДА (Для регистрации)
router.post('/verify-code', (req, res) => {
  try {
    const { email, code } = req.body;
    const result = verificationService.verifyCode(email.toLowerCase().trim(), code, 'register');

    if (!result.success) {
      return res.status(400).json({ error: result.message });
    }

    res.json({ success: true, message: "Email підтверджено" });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

// ЗАВЕРШЕНИЕ РЕГИСТРАЦИИ
router.post('/finish-registration', async (req, res) => {
  try {
    console.log("🚀 Получены данные на финальную регистрацию:", req.body);

    const { email, password, nickname, displayName, phone } = req.body;

    if (!email || !password || !nickname) {
      return res.status(400).json({ error: "Відсутні обов'язкові поля" });
    }
  }
});

```

```

const cleanEmail = email.toLowerCase().trim();

// ИСПРАВЛЕНИЕ: НЕ хэшируем пароль здесь вручную!
// Хук pre('save') в модели User.js сделает это сам. Если хэшировать и тут, будет двойной хэш.
const cleanDisplayName = displayName?.trim() || nickname.trim();

const user = await User.create({
  nickname: nickname.trim(),
  displayName: cleanDisplayName,
  tag: `@${cleanDisplayName.toUpperCase()}`,
  email: cleanEmail,
  phone: phone || "",
  password: password // Передаем чистый пароль!
});

const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, { expiresIn: '7d' });

console.log("✅ Пользователь успешно создан:", user.nickname);

res.json({
  success: true,
  user: {
    id: user._id,
    nickname: user.nickname,
    displayName: user.displayName,
    email: user.email,
    avatar: user.avatar || ""
  },
  token
});
} catch (err) {
  console.error("❌ Finish Registration Error:", err.message);
  res.status(500).json({ error: err.message || "Помилка створення користувача" });
}
});

// ЛОГИН
router.post('/login', async (req, res) => {
  try {
    const { identifier, password } = req.body;

    if (!identifier || !password) {
      return res.status(400).json({ error: "Заповніть всі поля" });
    }

    const cleanIdentifier = identifier.trim().toLowerCase();

    // Важно: если в модели стоит select: false для пароля, не забудь сделать .select('+password')
    const user = await User.findOne({
      $or: [{ email: cleanIdentifier }, { nickname: identifier.trim() }]
    }).select('+password');

    if (!user) return res.status(400).json({ error: "Користувача не знайдено" });

    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) return res.status(400).json({ error: "Невірний пароль" });

    const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, { expiresIn: '7d' });

```

```

    res.json({
      success: true,
      user: {
        id: user._id,
        nickname: user.nickname,
        displayName: user.displayName || user.nickname,
        email: user.email,
        avatar: user.avatar || ""
      },
      token
    });
  } catch (err) {
    console.error("Login error:", err);
    res.status(500).json({ error: "Помилка сервера" });
  }
});

// ===== ВОССТАНОВЛЕНИЕ ПАРОЛЯ =====

// 1. ЗАПРОС КОДА ВОССТАНОВЛЕНИЯ НА EMAIL
router.post('/forgot-password', async (req, res) => {
  try {
    const { email, lang } = req.body;
    const clientLang = lang === 'en' ? 'en' : 'uk';

    if (!email) {
      return res.status(400).json({ error: "Email обов'язковий" });
    }

    const cleanEmail = email.toLowerCase().trim();

    const user = await User.findOne({ email: cleanEmail });
    if (!user) {
      return res.status(404).json({ error: clientLang === 'en' ? "User with this Email was not found" : "Користувача з таким Email не знайдено" });
    }

    // Генеруємо код конкретно для 'recovery'
    await verificationService.createCode(cleanEmail, 'recovery', clientLang);

    res.json({ success: true, message: "Код відновлення відправлено" });
  } catch (err) {
    console.error("Forgot password error:", err);
    res.status(500).json({ error: "Внутрішня помилка сервера" });
  }
});

// 2. ПРОВЕРКА КОДА ВОССТАНОВЛЕНИЯ
router.post('/verify-recovery-code', (req, res) => {
  try {
    const { email, code } = req.body;

    if (!email || !code) {
      return res.status(400).json({ error: "Заповніть всі поля" });
    }

    const cleanEmail = email.toLowerCase().trim();
    const result = verificationService.verifyCode(cleanEmail, code, 'recovery');

    if (!result.success) {
      return res.status(400).json({ error: result.message });
    }
  }
});

```



```

    }

    res.json({ success: true, message: "Код успішно підтверджено" });
  } catch (err) {
    console.error("Verify recovery code error:", err);
    res.status(500).json({ error: "Внутрішня помилка сервера" });
  }
});

// 3. СОХРАНЕНИЕ НОВОГО ПАРОЛЯ ИЗ ЭКРАНА ВОССТАНОВЛЕНИЯ
router.post('/reset-password', async (req, res) => {
  try {
    const { email, code, newPassword } = req.body;

    if (!email || !code || !newPassword) {
      return res.status(400).json({ error: "Відсутні обов'язкові дані" });
    }

    const cleanEmail = email.toLowerCase().trim();

    // Проверяем код еще раз перед сохранением для безопасности
    const isVerified = verificationService.verifyCode(cleanEmail, code, 'recovery');
    if (!isVerified.success) {
      return res.status(400).json({ error: "Сесія перевірки коду застаріла. Спробуйте ще раз" });
    }

    const user = await User.findOne({ email: cleanEmail });
    if (!user) return res.status(404).json({ error: "Користувача не знайдено" });

    // Передаем ЧИСТЫЙ пароль. Хук модели User его захэширует автоматически.
    user.password = newPassword;
    await user.save();

    // После успешного сохранения удаляем код из памяти
    verificationService.removeCode(cleanEmail);

    res.json({ success: true, message: "Пароль успішно змінено" });
  } catch (err) {
    console.error("Reset password error:", err);
    res.status(500).json({ error: err.message });
  }
});

module.exports = router;

```

ДОДАТОК М

```

const router = require('express').Router();
const Chat = require('../models/Chat');
const auth = require('../middleware/authMiddleware');

const Message = require('../models/Message');

const fs = require('fs');
const path = require('path');

// Создать или получить чат с пользователем
router.post('/', auth, async (req, res) => {
  try {

```

```

const { userId } = req.body;

if (!userId || userId === req.userId.toString()) {
  return res.status(400).json({ error: 'Некорректный userId' });
}

// Ищем существующий чат
let chat = await Chat.findOne({
  isGroup: false,
  members: { $all: [req.userId, userId] }
});

if (!chat) {
  chat = await Chat.create({
    members: [req.userId, userId],
    isGroup: false
  });
}

// Популяризуем участников
chat = await Chat.findById(chat._id).populate('members', 'nickname displayName avatar tag');

res.json({ success: true, chat });
} catch (err) {
  console.error(err);
  res.status(500).json({ error: err.message });
}
});

// Получить все чаты пользователя (УЛУЧШЕННАЯ ВЕРСИЯ)
router.get('/', auth, async (req, res) => {
  try {

    const chats = await Chat.find({
      members: req.userId
    })
    .populate('members', 'nickname displayName avatar tag')
    .lean();

    const chatsWithMeta = await Promise.all(
      chats.map(async (chat) => {

        const lastMessage = await Message.findOne({ chatId: chat._id })
          .sort({ createdAt: -1 })
          .lean();

        const unreadCount = await Message.countDocuments({
          chatId: chat._id,
          senderId: { $ne: req.userId }
        });

        return {
          ...chat,
          lastMessage,
          unreadCount
        };
      })
    );

    res.json({

```

```

    success: true,
    chats: chatsWithMeta
  });

} catch (err) {
  console.error(err);
  res.status(500).json({ error: err.message });
}
});

// Перевірка статусу чату перед видаленням контакту
router.get('/check-status/:userId', auth, async (req, res) => {
  try {
    const { userId } = req.params;

    // Шукаємо приватний чат між користувачами
    const chat = await Chat.findOne({
      isGroup: false,
      members: { $all: [req.userId, userId] }
    });

    if (!chat) {
      return res.json({ success: true, hasChat: false, hasMessages: false });
    }

    // Перевіряємо, чи є в чаті хоча б одне повідомлення
    const messageCount = await Message.countDocuments({ chatId: chat._id });

    res.json({
      success: true,
      hasChat: true,
      chatId: chat._id,
      hasMessages: messageCount > 0
    });
  } catch (err) {
    console.error('Check chat status error:', err);
    res.status(500).json({ success: false, error: err.message });
  }
});

// Повне видалення чату, повідомлень та медіафайлів із сервера
router.delete('/:chatId', auth, async (req, res) => {
  try {
    const { chatId } = req.params;

    // Перевіряємо, чи користувач є учасником чату
    const chat = await Chat.findOne({ _id: chatId, members: req.userId });
    if (!chat) {
      return res.status(404).json({ success: false, error: 'Чат не знайдено або доступ заборонено' });
    }

    // 1. Фізично видаляємо папку з файлами чату на сервері (якщо вона існує)
    const chatFolder = path.join(__dirname, '..', '..', 'uploads', 'chats', chatId.toString());
    if (fs.existsSync(chatFolder)) {
      try {
        fs.rmSync(chatFolder, { recursive: true, force: true });
        console.log('🗑️ Папку медіафайлів чату ${chatId} успішно видалено з сервера');
      } catch (dirErr) {
        console.error('Не вдалося видалити папку чату:', dirErr);
      }
    }
  }
});

```

```
// 2. Видаляємо всі повідомлення чату з бази даних
await Message.deleteMany({ chatId });

// 3. Видаляємо сам чат
await Chat.findByIdAndDelete(chatId);

// 4. Оповіщаємо інших учасників через Socket.io про видалення чату
const io = req.app.get('io');
if (io && chat.members) {
  chat.members.forEach(memberId => {
    io.to(memberId.toString()).emit('chatDeleted', { chatId });
  });
}

res.json({ success: true, message: 'Чат та всі його дані успішно видалено' });
} catch (err) {
  console.error('Delete chat error:', err);
  res.status(500).json({ success: false, error: err.message });
}
});

module.exports = router;
```

ДОДАТОК Н

```
const Message = require('../models/Message');
const Chat = require('../models/Chat');

// ===== GET =====
exports.getAllMessages = async (req, res) => {
  try {

    const { chatId } = req.query;

    if (!chatId) {
      return res.status(400).json({
        error: 'chatId required'
      });
    }

    const messages = await Message.find({ chatId })
      .sort({ createdAt: 1 })
      .limit(100)
      .populate('senderId', 'nickname displayName avatar');

    res.json({
      success: true,
      messages
    });

  } catch (err) {
    console.error(err);

    res.status(500).json({
      error: err.message
    });
  }
};

// ===== POST =====
exports.createMessage = async (req, res) => {
  try {
    const text = req.body.text || '';
    const chatId = req.query.chatId;
    const file = req.file;

    if (!chatId) {
      return res.status(400).json({ success: false, error: 'chatId missing' });
    }

    const messageData = {
      chatId,
      senderId: req.userId,
      text
    };

    if (file) {
      let folder = 'files';
      if (file.mimetype.startsWith('image/')) folder = 'images';
      else if (file.mimetype.startsWith('video/')) folder = 'videos';

      messageData.file = {
        url: `uploads/chats/${chatId}/${folder}/${file.filename}`,
      };
    }
  }
};
```

```

        originalName: Buffer.from(file.originalname, 'latin1').toString('utf8'),
        mimetype: file.mimetype,
        size: file.size
    };
}

const message = await Message.create(messageData);

await Chat.findByIdAndUpdate(chatId, {
    lastMessage: {
        text: text || "",
        file: file ? { mimetype: file.mimetype } : null,
        createdAt: new Date()
    },
    updatedAt: new Date()
});

const populated = await Message.findById(message._id)
    .populate('senderId', 'nickname displayName avatar');

// ===== SOCKET.IO =====
const io = req.app.get('io');
if (io) {
    // Завантажуємо чат зі всіма учасниками
    const chat = await Chat.findById(chatId).select('members');

    if (chat && chat.members && chat.members.length > 0) {
        const messageToSend = populated.toObject();

        chat.members.forEach(memberId => {
            if (memberId) {
                io.to(memberId.toString()).emit('newMessage', messageToSend);
                console.log('📡 Socket sent to user: ${memberId}');
            }
        });
    }
}

res.status(201).json({
    success: true,
    message: populated
});

} catch (err) {
    console.error('Create message error:', err);
    res.status(500).json({ success: false, error: err.message });
}
};

```

ДОДАТОК 3

```

const mongoose = require('mongoose');

const MessageSchema = new mongoose.Schema({
    chatId: {
        type: String,

```

```
    default: 'general'
  },
  senderId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    default: null // ← сделали необязательным
  },
  text: {
    type: String,
    trim: true
  },
  file: {
    url: String,
    originalName: String,
    mimetype: String,
    size: Number
  }
}, {
  timestamps: true
});

module.exports = mongoose.model('Message', MessageSchema);
```

ДОДАТОК II

```
const multer = require('multer');
const fs = require('fs');
const path = require('path');
const bcrypt = require('bcryptjs');

const router = require('express').Router();
const User = require('../models/User');
const auth = require('../middleware/authMiddleware');

// ===== UPLOAD AVATAR =====

const avatarStorage = multer.diskStorage({
  destination: (req, file, cb) => {
    // uploads/avatars/<userId>/
    const userFolder = path.join(
      __dirname,
      '..',
      '..',
      'uploads',
      'avatars',
      req.userId.toString()
    );
  },

  // Создаём папку пользователя
  fs.mkdirSync(userFolder, { recursive: true });

  // Удаляем старые файлы, чтобы хранить только одну актуальную аватарку
  const existingFiles = fs.readdirSync(userFolder);
  existingFiles.forEach(fileName => {
    try {
      fs.unlinkSync(path.join(userFolder, fileName));
    } catch (err) {
      console.error('Не удалось удалить старый аватар:', err);
    }
  });

  cb(null, userFolder);
},

filename: (req, file, cb) => {
  const ext = path.extname(file.originalname).toLowerCase();

  // avatar-1716234567890.jpg
  const filename = `avatar-${Date.now()}${ext}`;

  cb(null, filename);
}
});

const avatarUpload = multer({
  storage: avatarStorage,
  limits: {
    fileSize: 5 * 1024 * 1024 // 5 MB
  },
  fileFilter: (req, file, cb) => {
    if (!file.mimetype.startsWith('image/')) {
      return cb(new Error('Only images are allowed'));
    }
  },
  cb(null, true);
});
```



```
}  
});
```

// получить свой профиль

```
router.get('/me', auth, async (req, res) => {  
  try {  
    const user = await User.findById(req.userId).select('-password');  
    res.json(user);  
  } catch (err) {  
    res.status(500).json({ error: err.message });  
  }  
});
```

// добавить в контакты

// загрузка аватарки

```
router.post('/avatar', auth, avatarUpload.single('avatar'), async (req, res) => {  
  try {  
    if (!req.file) {  
      return res.status(400).json({ error: 'Файл не завантажено' });  
    }  
  }  
}
```

```
const avatarUrl = `/uploads/avatars/${req.userId}/${req.file.filename}`;
```

```
console.log(' 📁 Аватарка сохранена по пути:', avatarUrl);
```

```
const user = await User.findByIdAndUpdate(  
  req.userId,  
  { avatar: avatarUrl },  
  { new: true, returnDocument: 'after' }  
).select('-password');
```

```
res.json({  
  success: true,  
  avatar: avatarUrl,  
  user  
});  
} catch (err) {  
  console.error('Avatar upload error:', err);  
  res.status(500).json({ error: err.message });  
}  
});
```

// обновление своего профиля (ИСПРАВЛЕНО: Синхронизация tag и nickname)

```
router.put('/me', auth, async (req, res) => {  
  try {  
    const { tag, displayName, bio } = req.body;  
  
    const updatedData = {  
      bio: (bio || "").trim()  
    };  
  
    if (displayName !== undefined && displayName.trim() !== "") {  
      updatedData.displayName = displayName.trim();  
    }  
  
    if (tag !== undefined && tag.trim() !== "") {
```

```

// Очищаем тег от лишних символов, если пользователь случайно ввел их
let cleanTag = tag.trim().toLowerCase();
if (cleanTag.startsWith('@')) {
  cleanTag = cleanTag.slice(1);
}

if (cleanTag !== '') {
  // Проверяем, не занят ли этот тег/никнейм кем-то другим
  const exists = await User.findOne({
    _id: { $ne: req.userId },
    $or: [{ nickname: cleanTag }, { tag: `@${cleanTag}` }]
  });

  if (exists) {
    return res.status(400).json({ success: false, error: 'Цей нікнейм вже зайнятий' });
  }

  // Синхронизируем оба поля! Старый никнейм стирается навсегда
  updatedData.nickname = cleanTag;
  updatedData.tag = `@${cleanTag}`;
}

const user = await User.findByIdAndUpdate(
  req.userId,
  updatedData,
  { new: true }
).select('-password');

res.json({
  success: true,
  user
});
} catch (err) {
  console.error('Profile update error:', err);
  res.status(500).json({
    success: false,
    error: err.message
  });
}
});

// получить контакты
router.get('/contacts', auth, async (req, res) => {
  try {
    const user = await User.findById(req.userId)
      .populate('contacts', 'nickname displayName tag avatar bio');

    res.json({
      success: true,
      contacts: user.contacts || []
    });
  } catch (err) {
    console.error('Contacts error:', err);
    res.status(500).json({
      success: false,
      error: err.message
    });
  }
}
}

```

```

});

// поиск пользователей
// Пошук користувачів із роздільною логікою для контактів та інших людей
router.get('/search', auth, async (req, res) => {
  try {
    let query = (req.query.query || '').trim();

    if (!query) {
      return res.json({
        success: true,
        users: { friends: [], nonFriends: [] }
      });
    }

    // Якщо пошук починається з @, зрізаємо його, щоб регулярка відпрацьовувала чисто
    if (query.startsWith('@')) {
      query = query.slice(1);
    }

    const searchRegex = { $regex: query, $options: 'i' };

    // 1. Отримуємо профіль поточного користувача та список його контактів
    const me = await User.findById(req.userId);
    const contactIds = (me.contacts || []).map(id => id.toString());

    // 2. Робимо вибірку з бази даних (включаючи себе).
    // Шукаємо по tag та displayName (поле nickname/login взагалі ігноруємо, щоб уникнути багів старого нікнейму)
    const users = await User.find({
      _id: { $ne: req.userId },
      $or: [
        { tag: searchRegex },
        { displayName: searchRegex }
      ]
    })
    .select('nickname displayName tag avatar bio')
    .limit(100);

    const friends = [];
    const nonFriends = [];

    // 3. Розподіляємо користувачів за твоєю логікою
    users.forEach(user => {
      const isFriend = contactIds.includes(user._id.toString());
      const userData = {
        ...user.toObject(),
        isFriend
      };

      if (isFriend) {
        // ДЛЯ КОНТАКТІВ: додаємо, якщо збігся або тег, або displayName (це вже відпрацювало у вибірці вище)
        friends.push(userData);
      } else {
        // ДЛЯ ІНШИХ КОРИСТУВАЧІВ: додаємо ТІЛЬКІ якщо запит збігається з тегом!
        // Перевіряємо, чи є підстрока запиту в його актуальному тегу
        const cleanUserTag = (user.tag || '').replace('@', '').toLowerCase();
        if (cleanUserTag.includes(query.toLowerCase())) {
          nonFriends.push(userData);
        }
      }
    });
  }
});

```

```

});

res.json({
  success: true,
  users: {
    friends,
    nonFriends
  }
});
} catch (err) {
  console.error('Search error:', err);
  res.status(500).json({
    success: false,
    error: err.message
  });
}
});

// додати контакт
router.post('/add-contact', auth, async (req, res) => {
  try {
    const { userId } = req.body;

    if (!userId || userId === req.userId) {
      return res.status(400).json({
        success: false,
        error: 'Некоректний userId'
      });
    }

    const me = await User.findById(req.userId);

    const alreadyExists = me.contacts.some(
      id => id.toString() === userId
    );

    if (!alreadyExists) {
      me.contacts.push(userId);
      await me.save();
    }

    res.json({
      success: true
    });
  } catch (err) {
    console.error('Add contact error:', err);
    res.status(500).json({
      success: false,
      error: err.message
    });
  }
});

// видалити контакт ОДНОЧАСНО З ЧАТОМ ТА ВСІМА ДАНИМИ
router.post('/remove-contact', auth, async (req, res) => {
  try {
    const { userId } = req.body;
    if (!userId) {
      return res.status(400).json({ success: false, error: 'Відсутній userId' });
    }
  }

```

```

// 1. Видаляємо користувача з масиву контактів поточного юзера
const me = await User.findById(req.userId);
if (me) {
  me.contacts = (me.contacts || []).filter(
    id => id.toString() !== userId.toString()
  );
  await me.save();
}

// 2. Шукаємо приватний чат між цими двома користувачами
const Chat = require('../models/Chat'); // Переконайся, що моделі імпортовані
const Message = require('../models/Message');
const fs = require('fs');
const path = require('path');

const chat = await Chat.findOne({
  isGroup: false,
  members: { $all: [req.userId, userId] }
});

if (chat) {
  const chatId = chat._id;

  // А. Фізично видаляємо папку з медіафайлами чату на сервері (якщо є)
  const chatFolder = path.join(__dirname, '..', '..', 'uploads', 'chats', chatId.toString());
  if (fs.existsSync(chatFolder)) {
    try {
      fs.rmSync(chatFolder, { recursive: true, force: true });
      console.log(` 🗑 Папку медіафайлів чату ${chatId} успішно видалено`);
    } catch (dirErr) {
      console.error('Не вдалося видалити папку чату:', dirErr);
    }
  }

  // Б. Видаляємо всі повідомлення чату з бази даних
  await Message.deleteMany({ chatId });

  // В. Видаляємо сам запис чату
  await Chat.findByIdAndDelete(chatId);

  // Г. Оповіщаємо іншого учасника через Socket.io про те, що чат видалено
  const io = req.app.get('io');
  if (io && chat.members) {
    chat.members.forEach(memberId => {
      io.to(memberId.toString()).emit('chatDeleted', { chatId });
    });
  }
  console.log(` 🗑 Чат ${chatId} та всі його дані повністю знищено.`);
}

res.json({
  success: true
});
} catch (err) {
  console.error('Remove contact error:', err);
  res.status(500).json({
    success: false,
    error: err.message
  });
}
}

```

```

// ===== GET USER BY ID =====
router.get('/:id', auth, async (req, res) => {
  try {
    const user = await User.findById(req.params.id)
      .select('nickname displayName tag avatar bio contacts');

    if (!user) {
      return res.status(404).json({
        success: false,
        error: 'User not found'
      });
    }

    res.json({
      success: true,
      user
    });
  } catch (err) {
    console.error('Get user by id error:', err);
    res.status(500).json({
      success: false,
      error: err.message
    });
  }
});

// ===== 1. ПРОВЕРКА СТАРОГО ПАРОЛЯ =====
router.post('/change-password/verify', auth, async (req, res) => {
  try {
    const { oldPassword } = req.body;
    if (!oldPassword) {
      return res.status(400).json({ success: false, error: 'Введіть старий пароль' });
    }

    // Принудительно запрашиваем поле password
    const user = await User.findById(req.userId).select('+password');
    if (!user) {
      return res.status(404).json({ success: false, error: 'Користувача не знайдено' });
    }

    // ВАЖНО: Проверяем, что используется именно bcrypt (так как вверху файла ты импортировал его как
    const bcrypt = require('bcryptjs');
    const isMatch = await bcrypt.compare(oldPassword, user.password);
    if (!isMatch) {
      return res.status(400).json({ success: false, error: 'Невірний старий пароль' });
    }

    res.json({ success: true, message: 'Старий пароль підтверджено' });
  } catch (err) {
    // ЭТОТ ЛОГ ПОКАЖЕТ ВСЁ В ТЕРМИНАЛЕ СЕРВЕРА:
    console.error('=== КРИТИЧЕСКАЯ ОШИБКА НА БЭКЕНДЕ ===');
    console.error(err);
    res.status(500).json({ success: false, error: err.message });
  }
});

// ===== 2. СОХРАНЕНИЕ НОВОГО ПАРОЛЯ =====
router.post('/change-password/save', auth, async (req, res) => {
  try {
    const { oldPassword, newPassword } = req.body;

    if (!oldPassword || !newPassword) {
      return res.status(400).json({ success: false, error: 'Усі поля обов'язкові' });
    }
  }
});

```

```

    }

    if (newPassword.length < 6) {
        return res.status(400).json({ success: false, error: 'Пароль має бути не менше 6 символів' });
    }

    // Здець тоже принудительно запрашиваем текущий пароль для повторной проверки
    const user = await User.findById(req.userId).select('+password');
    if (!user) {
        return res.status(404).json({ success: false, error: 'Користувача не знайдено' });
    }

    const isMatch = await bcrypt.compare(oldPassword, user.password);
    if (!isMatch) {
        return res.status(400).json({ success: false, error: 'Сесія безпеки порушена. Невірний старий пароль' });
    }

    // Передаємо чистий текст, хук в моделі сам його правильно захешує
    user.password = newPassword;
    await user.save();

    res.json({ success: true, message: 'Пароль успішно змінено' });
  } catch (err) {
    console.error('Save new password error:', err);
    res.status(500).json({ success: false, error: err.message });
  }
});

module.exports = router;

```

ДОДАТОК Т

```

const dns = require('dns');

dns.setDefaultResultOrder('ipv4first');
dns.setServers(['8.8.8.8', '8.8.4.4', '1.1.1.1']);

const express = require('express');
const http = require('http'); // ← HOBE
const { Server } = require('socket.io'); // ← HOBE

const cors = require('cors');
const dotenv = require('dotenv');
const path = require('path');

dotenv.config();

const connectDB = require('./config/database');
const upload = require('./middleware/upload');
const auth = require('./middleware/authMiddleware');

const app = express();

// ===== MIDDLEWARE =====
app.use(cors());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

```

```

// ===== STATIC =====
const ROOT = path.resolve(__dirname, '..');
const UPLOADS = path.join(ROOT, 'uploads');
const PUBLIC = path.join(ROOT, 'public');

app.use(express.static(PUBLIC));
app.use('/uploads', express.static(UPLOADS, {
  index: false,
  etag: false,
  lastModified: false,
  maxAge: 0,
  setHeaders: (res) => {
    res.set('Cache-Control', 'no-store');
    res.set('Access-Control-Allow-Origin', '*');
  }
}));

// ===== ROUTES =====
app.use('/api/auth', require('./routes/auth'));
app.use('/api/users', require('./routes/users'));
app.use('/api/chats', require('./routes/chats'));

const messageRoutes = require('./routes/messages');

app.post('/api/messages', auth, upload.single('file'), messageRoutes.createMessage);
app.get('/api/messages', auth, messageRoutes.getAllMessages);

// ===== SOCKET.IO =====
const server = http.createServer(app); // ← ЗМІНИЛИ

const io = new Server(server, {
  cors: {
    origin: "*", // Для розробки
    methods: ["GET", "POST"]
  }
});

// Підключення сокетів
io.on('connection', (socket) => {
  console.log('Користувач підключився:', socket.id);

  // Приєднуємо користувача до його особистої кімнати
  socket.on('join', (userId) => {
    socket.join(userId);
    console.log(`User ${userId} joined his room`);
  });

  socket.on('disconnect', () => {
    console.log('Користувач відключився:', socket.id);
  });
});

// Зробимо io доступним у всіх роутах
app.set('io', io); // ← Важливо!

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

(async () => {
  await connectDB();

```



```
server.listen(PORT, () => {      // ← server.listen замість app.listen
  console.log(🚀 Server: http://localhost:${PORT});
  console.log('📁 Uploads:', UPLOADS);
});
})();
```

ДОДАТОК С

```
const sgMail = require('@sendgrid/mail');

// Инициализируем SendGrid ключом из переменных окружения
// Переменную SENDGRID_API_KEY нужно обязательно добавить в Dashboard на Render
sgMail.setApiKey(process.env.SENDGRID_API_KEY);

/**
 * Универсальная отправка кода через SendGrid API
 * @param {string} email - Кому отправляем
 * @param {string} code - 6-значный код
 * @param {string} type - 'register' или 'recovery'
 * @param {string} lang - 'uk' или 'en'
 */
async function sendCode(email, code, type = 'register', lang = 'uk') {
  try {
    let subject = 'Chatty';
    let header = '';
    let bodyText = '';
    let ignoreText = '';

    if (lang === 'en') {
      subject = type === 'recovery' ? 'Password reset — Chatty' : 'Registration confirmation code — Chatty';
      header = type === 'recovery' ? 'Password recovery request' : 'Your registration confirmation code';
      bodyText = type === 'recovery' ? 'You received this email because a password reset request was made for your account.' : 'Thank you for choosing Chatty. Use this code to complete your registration.';
      ignoreText = 'If you did not request this code, please ignore this email.';
    } else { // 'uk' по умолчанию
      subject = type === 'recovery' ? 'Скидання пароля — Chatty' : 'Код підтвердження — Chatty';
      header = type === 'recovery' ? 'Запит на відновлення пароля' : 'Ваш код підтвердження';
      bodyText = type === 'recovery' ? 'Ви отримали цей лист, тому що зробили запит на відновлення доступу до аккаунту.' : 'Дякуємо, що обрали Chatty. Використовуйте цей код для завершення реєстрації.';
      ignoreText = 'Якщо ви не запитували цей код — просто ігноруйте це повідомлення.';
    }

    const msg = {
      to: email,
      // Важно: Этот Email должен быть верифицирован в вашем аккаунте SendGrid (Single Sender Verification)
      from: {
        email: process.env.EMAIL_USER,
        name: 'Chatty'
      },
      subject: subject,
      html: `
        <div style="font-family: 'Segoe UI', Arial, sans-serif; max-width: 500px; margin: 0 auto; border: 1px solid #e2e8f0; border-radius: 12px; padding: 25px; background-color: #ffffff; color: #1e293b;">
          <div style="text-align: center; margin-bottom: 20px;">
            <h1 style="color: #3b82f6; margin: 0; font-size: 28px;">Chatty</h1>
          </div>
          <h3 style="color: #0f172a; margin-top: 0;">${header}</h3>
          <p style="font-size: 15px; line-height: 1.5; color: #475569;">${bodyText}</p>
          <div style="text-align: center; margin: 30px 0; background: #f8f9fc; padding: 15px; border-radius: 8px;">
            <span style="font-family: monospace; color: #2563eb; font-size: 38px; font-weight: bold; letter-spacing: 6px; display: inline-block;">${code}</span>
          </div>
          <p style="font-size: 13px; color: #94a3b8; margin-bottom: 0;">${ignoreText}</p>
        </div>
      `
    };
  }
}
```

```
// Отправка через HTTP API (порт 443)
await sgMail.send(msg);

    console.log(  [${type.toUpperCase()}] [${lang.toUpperCase()}] Код ${code} успешно отправлен через
SendGrid API на ${email}`);
} catch (error) {
    console.error("  Ошибка отправки email через SendGrid:");
    if (error.response) {
        console.error(error.response.body);
    } else {
        console.error(error);
    }
    throw error;
}
}

module.exports = sendCode;
```

```

let translations = {};
let currentLanguage = 'uk';

async function loadLanguage(lang) {
  try {
    const response = await fetch(`/locales/${lang}.json`);
    if (!response.ok) throw new Error('Language file not found');

    translations = await response.json();
    currentLanguage = lang;

    applyTranslations(); // сразу переводим DOM

    if (typeof buildMenuContent === 'function') {
      buildMenuContent(); // пересоздаём меню
    }

    setTimeout(() => {
      applyTranslations(); // повтор после DOM rebuild
    }, 0);

  } catch (error) {
    console.error('Language loading error:', error);
  }
}

function t(key) {
  return translations[key] || key;
}

function applyTranslations() {
  // ===== TEXT =====
  document.querySelectorAll('[data-i18n]').forEach(el => {
    const key = el.getAttribute('data-i18n');
    if (!key) return;
    el.textContent = t(key);
  });

  // ===== PLACEHOLDER =====
  document.querySelectorAll('[data-i18n-placeholder]').forEach(el => {
    const key = el.getAttribute('data-i18n-placeholder');
    if (!key) return;
    el.placeholder = t(key);
  });

  // ===== TITLE =====
  document.querySelectorAll('[data-i18n-title]').forEach(el => {
    const key = el.getAttribute('data-i18n-title');
    if (!key) return;
    el.title = t(key);
  });
}

async function setLanguage(lang) {
  try {
    localStorage.setItem('chatty_language', lang);

    await loadLanguage(lang);
  }
}

```

```

const select = document.getElementById('languageSelect');
if (select) select.value = lang;

// 1) Пересобираем динамическое меню настроек/профиля
if (typeof buildMenuContent === 'function') {
    buildMenuContent();
}

// 2) Применяем переводы к статическим элементам DOM (data-i18n)
applyTranslations();

// 🔥 3) МГНОВЕННЫЙ ПЕРЕВОРОТ ДИНАМИЧЕСКИХ ДАННЫХ ИЗ app.js:

// Перерисовываем список чатов, чтобы обновить "Нет сообщений" или превью ("📷 Фото")
if (typeof renderChats === 'function' && window.currentChatsList) {
    renderChats(window.currentChatsList);
}

// Перерисовываем медиа-панели (Фото / Видео / Файлы), чтобы "No videos" перевелось на ленту
if (typeof currentChatMediaData !== 'undefined' && window.currentChatMediaData) {
    if (typeof renderMediaGrid === 'function') {
        renderMediaGrid('photosGrid', window.currentChatMediaData.images || [], 'image');
        renderMediaGrid('videosGrid', window.currentChatMediaData.videos || [], 'video');
    }
    if (typeof renderFilesList === 'function') {
        renderFilesList('filesList', window.currentChatMediaData.files || []);
    }
}

} catch (err) {
    console.error('Language switch failed:', err);

    if (lang !== 'uk') {
        await loadLanguage('uk');
    }
}

}

function getCurrentLanguage() {
    return currentLanguage;
}

window.addEventListener('DOMContentLoaded', async () => {
    const saved = localStorage.getItem('chatty_language') || 'uk';
    await loadLanguage(saved);
});

window.t = t;
window.setLanguage = setLanguage;
window.getCurrentLanguage = getCurrentLanguage;
window.applyTranslations = applyTranslations;

```

ДОДАТОК Х

```
{
  "login": "Увійти",
  "register": "Зареєструватися",
  "registration": "Реєстрація",
  "back": "Назад",
  "continue": "Далі",
  "verify": "Перевірити",

  "nicknameOrEmail": "Нікнейм або Email",
  "password": "Пароль",
  "repeatPassword": "Повторіть пароль",
  "nickname": "Нікнейм",
  "displayName": "Ім'я (як вас бачать)",
  "phone": "Телефон",
  "email": "Email",
  "enterCode": "Введіть код",

  "noAccount": "Немає акаунта? Зареєструйтесь!",
  "confirmation": "Підтвердження",
  "passwordCreate": "Придумайте пароль",

  "codeSent": "Ми відправили код на",
  "resendCode": "Відправити код ще раз",

  "chooseChat": "Оберіть чат",
  "chooseChatDescription": "Виберіть контакт справа, щоб почати розмову",

  "selectChat": "Виберіть чат",
  "chatMediaHere": "Тут буде медіа вибраного чату",

  "messagePlaceholder": "Напишіть повідомлення...",

  "sending": "Відправляється...",
  "sendMessage": "Повідомлення відправляється...",
  "sendingFile": "Файл відправляється...",

  "loading": "Завантаження...",
  "noChats": "Немає чатів",
  "writeFirst": "Напишіть першим...",

  "photos": "Фото",
  "videos": "Відео",
  "files": "Файли",

  "noPhotos": "Немає фото",
  "noVideos": "Немає відео",
  "noFiles": "Немає файлів",

  "chats": "Чати",

  "myProfile": "Мій профіль",
  "contacts": "Контакти",
  "settings": "Налаштування",
  "logout": "Вихід",

  "openProfile": "Відкрити профіль",
  "bioEmpty": "Біографія відсутня",
```

"goToMessage": "Перейти до повідомлення",
"download": "Завантажити",

"writeMessage": "Написати повідомлення",
"addContact": "Додати до контактів",
"removeContact": "Видалити контакт та чат",

"contactsEmpty": "Контактів поки немає",
"nothingFound": "Нічого не знайдено",

"serverError": "Помилка сервера",
"connectionError": "Помилка з'єднання",

"selectChatFirst": "Спочатку виберіть чат!",

"saved": "Збережено ✓",
"saving": "Збереження...",

"deleteChatConfirm": "Видалити чат та всі повідомлення?",
"deleteContactConfirm": "Видалити контакт і весь чат з цим користувачем?",

"user": "Користувач",

"registrationSuccess": "Реєстрація успішна! 🎉",

"newBadge": "NEW",

"enterLoginData": "Введіть дані для входу",
"loggingIn": "Входимо...",
"invalidCredentials": "Невірні дані",
"nicknameEmailRequired": "Нікнейм та Email обов'язкові",
"sendingCode": "Відправляємо код...",
"registrationError": "Помилка реєстрації",

"enter6DigitCode": "Введіть 6-значний код",
"invalidCode": "Невірний код",
"codeVerificationError": "Помилка перевірки коду",

"passwordsDoNotMatch": "Паролі не співпадають",
"passwordTooShort": "Пароль повинен бути не менше 6 символів",
"finishingRegistration": "Завершуємо...",
"finishRegistrationError": "Помилка завершення реєстрації",
"sendMessageFailed": "Не вдалося відправити повідомлення",
"menu": "Меню",
"tag": "Тег",
"bio": "Біографія",
"save": "Зберегти",
"yourName": "Ваше ім'я",
"tellAboutYourself": "Розкажіть про себе",
"contactSearch": "Пошук контактів",
"contactSearchPlaceholder": "Введіть ім'я, нікнейм або @тег...",
"searchResults": "Результати пошуку",
"theme": "Тема",
"sound": "Звук",
"language": "Мова",
"changePassword": "Змінити пароль",
"changePhoto": "Натисніть, щоб змінити фото",

"deleteFromContacts": "Видалити з контактів",

"bioAbsent": "Біографія відсутня",

```

"loadedTimestamps": "Завантажено timestamps для користувача",
"failedLoadTimestamps": "Не вдалося завантажити timestamps",
"failedSaveTimestamps": "Не вдалося зберегти timestamps",
"addToContacts": "Додати до контактів",

"finishing": "Завершуємо...",
"next": "Далі",
"noMessages": "Немає повідомлень",
"noContactsYet": "Контактів поки немає",
"forgotPassword": "Забули пароль?",
"passwordRecovery": "Відновлення пароля",
"enterYourEmail": "Ваш Email",
"sendCodeBtn": "Надіслати код",
"emailConfirmation": "Підтвердження Email",
"codeFromEmail": "Код з пошти",
"verifyCodeBtn": "Перевірити код",
"newPasswordTitle": "Новий пароль",
"recoveryNewPasswordPlaceholder": "Новий пароль (від 6 символів)",
"recoveryConfirmPasswordPlaceholder": "Повторіть новий пароль",
"savePasswordBtn": "Зберегти пароль",
"userNotFound": "Користувача з таким Email не знайдено",
"passwordResetSuccess": "Пароль успішно змінено! 🎉",
"emailSubjectRegister": "Код підтвердження реєстрації — Chatty",
"emailSubjectRecovery": "Скидання пароля — Chatty",
"emailHeaderRegister": "Ваш код підтвердження реєстрації",
"emailHeaderRecovery": "Запит на відновлення пароля",
"emailBodyRecovery": "Ви отримали цей лист, тому що зробили запит на відновлення доступу до акаунта.",
"saveBtn": "Зберегти",
"savedSuccess": "Збережено! ✓",
"myContactsSection": "Ваші Контакти",
"otherUsersSection": "Інші користувачі",
"deleteActionTitle": "Видалення розмови",
"deleteActionDesc": "Цей чат містить історію повідомлень. Оберіть варіант видалення:",
"deleteOnlyChat": "✗ Видалити тільки чат",
"deleteChatAndContact": "🗑️ Видалити чат та контакт"
}

```



```
{
  "login": "Accedi",
  "register": "Registrati",
  "registration": "Registrazione",
  "back": "Indietro",
  "continue": "Continua",
  "verify": "Verifica",

  "nicknameOrEmail": "Nickname o Email",
  "password": "Password",
  "repeatPassword": "Ripeti la password",
  "nickname": "Nickname",
  "displayName": "Nome visualizzato (come ti vedono gli altri)",
  "phone": "Telefono",
  "email": "Email",
  "enterCode": "Inserisci il codice",

  "noAccount": "Non hai un account? Registrati!",
  "confirmation": "Conferma",
  "passwordCreate": "Crea una password",

  "codeSent": "Abbiamo inviato un codice a",
  "resendCode": "Invia di nuovo il codice",

  "chooseChat": "Seleziona una chat",
  "chooseChatDescription": "Seleziona un contatto a destra per iniziare una conversazione",

  "selectChat": "Seleziona chat",
  "chatMediaHere": "I file multimediali della chat selezionata appariranno qui",

  "messagePlaceholder": "Scrivi un messaggio...",

  "sending": "Invio in corso...",
  "sendMessage": "Invio del messaggio...",
  "sendingFile": "Invio del file...",

  "loading": "Caricamento...",
  "noChats": "Nessuna chat disponibile",
  "writeFirst": "Scrivi per primo...",

  "photos": "Foto",
  "videos": "Video",
  "files": "File",

  "noPhotos": "Nessuna foto",
  "noVideos": "Nessun video",
  "noFiles": "Nessun file",

  "chats": "Chat",

  "myProfile": "Il mio profilo",
  "contacts": "Contatti",
  "settings": "Impostazioni",
  "logout": "Disconnetti",

  "openProfile": "Apri profilo",

  "bioEmpty": "Biografia non presente",
```

"goToMessage": "Vai al messaggio",
"download": "Scarica",

"writeMessage": "Scrivi un messaggio",
"addContact": "Aggiungi ai contatti",
"removeContact": "Elimina contatto e chat",

"contactsEmpty": "Nessun contatto presente",
"nothingFound": "Nessun risultato trovato",

"serverError": "Errore del server",
"connectionError": "Errore di connessione",

"selectChatFirst": "Per favore, seleziona prima una chat!",

"saved": "Salvato",
"savedSuccess": "Salvataggio...",
"saveBtn": "Salva",

"deleteChatConfirm": "Eliminare la chat e tutti i messaggi?",
"deleteContactConfirm": "Eliminare il contatto e l'intera chat con questo utente?",

"user": "Utente",

"registrationSuccess": "Registrazione completata con successo! 🎉",

"newBadge": "NUOVO",

"enterLoginData": "Inserisci i dati di accesso",
"loggingIn": "Accesso in corso...",
"invalidCredentials": "Credenziali non valide",
"nicknameEmailRequired": "Nickname ed Email sono obbligatori",
"sendingCode": "Invio del codice...",
"registrationError": "Errore di registrazione",

"enter6DigitCode": "Inserisci il codice a 6 cifre",
"invalidCode": "Codice non valido",
"codeVerificationError": "Errore di verifica del codice",

"passwordsDoNotMatch": "Le password non corrispondono",
"passwordTooShort": "La password deve contenere almeno 6 caratteri",
"finishingRegistration": "Completamento...",
"finishRegistrationError": "Impossibile completare la registrazione",
"sendMessageFailed": "Impossibile inviare il messaggio",
"menu": "Menu",
"tag": "Tag",
"bio": "Biografia",
"save": "Salva",
"yourName": "Il tuo nome",
"tellAboutYourself": "Racconta qualcosa di te",
"contactSearch": "Cerca contatti",
"contactSearchPlaceholder": "Inserisci nome, nickname o @tag...",
"searchResults": "Risultati della ricerca",
"theme": "Tema",
"sound": "Suono",
"language": "Lingua",
"changePassword": "Cambia password",
"changePhoto": "Clicca per cambiare la foto",

"deleteFromContacts": "Rimuovi dai contatti",

```

"bioAbsent": "Biografia non presente",

"loadedTimestamps": "Timestamp caricati per l'utente",
"failedLoadTimestamps": "Impossibile caricare i timestamp",
"failedSaveTimestamps": "Impossibile salvare i timestamp",
"addToContacts": "Aggiungi ai contatti",

"finishing": "Completamento...",
"next": "Avanti",
"noMessages": "Nessun messaggio",
"noContactsYet": "Nessun contatto presente",
"forgotPassword": "Password dimenticata?",
"passwordRecovery": "Recupero password",
"enterYourEmail": "La tua Email",
"sendCodeBtn": "Invia codice",
"emailConfirmation": "Conferma Email",
"codeFromEmail": "Codice dalla mail",
"verifyCodeBtn": "Verifica codice",
"newPasswordTitle": "Nuova password",
"recoveryNewPasswordPlaceholder": "Nuova password (min. 6 caratteri)",
"recoveryConfirmPasswordPlaceholder": "Ripeti la nuova password",
"savePasswordBtn": "Salva password",
"userNotFound": "Utente con questa Email non trovato",
"passwordResetSuccess": "Password modificata con successo! 🎉",
"emailSubjectRegister": "Codice di conferma registrazione — Chatty",
"emailSubjectRecovery": "Reimpostazione della password — Chatty",
"emailHeaderRegister": "Il tuo codice di conferma registrazione",
"emailHeaderRecovery": "Richiesta di recupero password",
"emailBodyRecovery": "Hai ricevuto questa email perché è stato richiesto il recupero dell'accesso al tuo account.",
"myContactsSection": "I tuoi contatti",
"otherUsersSection": "Altri utenti",
"deleteActionTitle": "Elimina conversazione",
"deleteActionDesc": "Questa chat contiene la cronologia dei messaggi. Scegli un'opzione di eliminazione:",
"deleteOnlyChat": "✖ Elimina solo la chat",
"deleteChatAndContact": "🗑 Elimina chat e contatto"
}

```

```

{
  "login": "Log In",
  "register": "Register",
  "registration": "Registration",
  "back": "Back",
  "continue": "Continue",
  "verify": "Verify",

  "nicknameOrEmail": "Nickname or Email",
  "password": "Password",
  "repeatPassword": "Repeat password",
  "nickname": "Nickname",
  "displayName": "Display Name (as seen by others)",
  "phone": "Phone",
  "email": "Email",
  "enterCode": "Enter code",

  "noAccount": "Don't have an account? Register!",
  "confirmation": "Confirmation",
  "passwordCreate": "Create a password",

  "codeSent": "We sent a code to",
  "resendCode": "Resend code",

  "chooseChat": "Select a chat",
  "chooseChatDescription": "Select a contact from the right to start a conversation",

  "selectChat": "Select chat",
  "chatMediaHere": "Shared media will appear here",

  "messagePlaceholder": "Write a message...",

  "sending": "Sending...",
  "sendMessage": "Sending message...",
  "sendingFile": "Sending file...",

  "loading": "Loading...",
  "noChats": "No chats available",
  "writeFirst": "Be the first to write...",

  "photos": "Photos",
  "videos": "Videos",
  "files": "Files",

  "noPhotos": "No photos",
  "noVideos": "No videos",
  "noFiles": "No files",

  "chats": "Chats",

  "myProfile": "My Profile",
  "contacts": "Contacts",
  "settings": "Settings",
  "logout": "Log Out",

  "openProfile": "Open profile",

  "bioEmpty": "No biography added",

```

"goToMessage": "Go to message",
"download": "Download",

"writeMessage": "Write a message",
"addContact": "Add to contacts",
"removeContact": "Delete contact and chat",

"contactsEmpty": "No contacts yet",
"nothingFound": "Nothing found",

"serverError": "Server error",
"connectionError": "Connection error",

"selectChatFirst": "Please select a chat first!",

"saving": "Saving... ",

"saveBtn": "Save",
"savedSuccess": "Saved! ✓",

"deleteChatConfirm": "Delete chat and all messages?",
"deleteContactConfirm": "Delete contact and entire chat with this user?",

"user": "User",

"registrationSuccess": "Registration successful! 🎉",

"newBadge": "NEW",

"enterLoginData": "Enter your login details",
"loggingIn": "Logging in...",
"invalidCredentials": "Invalid credentials",
"nicknameEmailRequired": "Nickname and Email are required",
"sendingCode": "Sending code...",
"registrationError": "Registration error",

"enter6DigitCode": "Enter 6-digit code",
"invalidCode": "Invalid code",
"codeVerificationError": "Code verification error",

"passwordsDoNotMatch": "Passwords do not match",
"passwordTooShort": "Password must be at least 6 characters",
"finishingRegistration": "Finishing up...",
"finishRegistrationError": "Failed to complete registration",
"sendMessageFailed": "Failed to send message",
"menu": "Menu",
"tag": "Tag",
"bio": "Biography",
"save": "Save",
"yourName": "Your name",
"tellAboutYourself": "Tell something about yourself",
"contactSearch": "Search contacts",
"contactSearchPlaceholder": "Enter name, nickname or @tag...",
"searchResults": "Search results",
"theme": "Theme",
"sound": "Sound",
"language": "Language",
"changePassword": "Change password",
"changePhoto": "Click to change photo",

"deleteFromContacts": "Delete from contacts",

```

"bioAbsent": "No biography available",

"loadedTimestamps": "Loaded timestamps for user",
"failedLoadTimestamps": "Failed to load timestamps",
"failedSaveTimestamps": "Failed to save timestamps",
"addToContacts": "Add to contacts",

"finishing": "Finishing...",
"next": "Next",
"noMessages": "No messages",
"noContactsYet": "No contacts yet",
"forgotPassword": "Forgot password?",
"passwordRecovery": "Password recovery",
"enterYourEmail": "Your Email",
"sendCodeBtn": "Send code",
"emailConfirmation": "Email confirmation",
"codeFromEmail": "Code from email",
"verifyCodeBtn": "Verify code",
"newPasswordTitle": "New password",
"recoveryNewPasswordPlaceholder": "New password (min 6 characters)",
"recoveryConfirmPasswordPlaceholder": "Confirm new password",
"savePasswordBtn": "Save password",
"userNotFound": "User with this Email not found",
"passwordResetSuccess": "Password changed successfully! 🎉",
"emailSubjectRegister": "Registration confirmation code — Chatty",
"emailSubjectRecovery": "Password reset — Chatty",
"emailHeaderRegister": "Your registration confirmation code",
"emailHeaderRecovery": "Password recovery request",
"emailBodyRecovery": "You received this email because you requested a password reset for your account.",
"myContactsSection": "Your Contacts",
"otherUsersSection": "Other Users",

"deleteActionTitle": "Delete Conversation",
"deleteActionDesc": "This chat contains message history. Choose a deletion option:",
"deleteOnlyChat": "✖ Delete chat only",
"deleteChatAndContact": "🗑 Delete chat and contact"
}

```

```
{
    "login": "Einloggen",
    "register": "Registrieren",
    "registration": "Registrierung",
    "back": "Zurück",
    "continue": "Weiter",
    "verify": "Überprüfen",

    "nicknameOrEmail": "Benutzername oder E-Mail",
    "password": "Passwort",
    "repeatPassword": "Passwort wiederholen",
    "nickname": "Benutzername",
    "displayName": "Anzeigename (wie andere Sie sehen)",
    "phone": "Telefonnummer",
    "email": "E-Mail",
    "enterCode": "Code eingeben",

    "noAccount": "Noch kein Konto? Registrieren!",
    "confirmation": "Bestätigung",
    "passwordCreate": "Passwort erstellen",

    "codeSent": "Wir haben einen Code gesendet an",
    "resendCode": "Code erneut senden",

    "chooseChat": "Chat auswählen",
    "chooseChatDescription": "Wählen Sie rechts einen Kontakt aus, um ein Gespräch zu beginnen",

    "selectChat": "Chat auswählen",
    "chatMediaHere": "Medien des ausgewählten Chats werden hier angezeigt",

    "messagePlaceholder": "Nachricht schreiben...",

    "sending": "Senden...",
    "sendingMessage": "Nachricht wird gesendet...",
    "sendingFile": "Datei wird gesendet...",

    "loading": "Laden...",
    "noChats": "Keine Chats vorhanden",
    "writeFirst": "Schreiben Sie als Erster...",

    "photos": "Fotos",
    "videos": "Videos",
    "files": "Dateien",

    "noPhotos": "Keine Fotos",
    "noVideos": "Keine Videos",
    "noFiles": "Keine Dateien",

    "chats": "Chats",

    "myProfile": "Mein Profil",
    "contacts": "Kontakte",
    "settings": "Einstellungen",
    "logout": "Abmelden",

    "openProfile": "Profil öffnen",

    "bioEmpty": "Keine Biografie vorhanden",
}
```

"goToMessage": "Zur Nachricht gehen",
"download": "Herunterladen",

"writeMessage": "Nachricht schreiben",
"addContact": "Zu Kontakten hinzufügen",
"removeContact": "Kontakt und Chat löschen",

"contactsEmpty": "Noch keine Kontakte vorhanden",
"nothingFound": "Nichts gefunden",

"serverError": "Serverfehler",
"connectionError": "Verbindungsfehler",

"selectChatFirst": "Bitte wählen Sie zuerst einen Chat aus!",

"saving": "Speichern...",
"saveBtn": "Speichern",
"savedSuccess": "Gespeichert! ✓",

"deleteChatConfirm": "Chat und alle Nachrichten löschen?",
"deleteContactConfirm": "Kontakt und den gesamten Chat mit diesem Benutzer löschen?",

"user": "Benutzer",

"registrationSuccess": "Registrierung erfolgreich! 🎉",

"newBadge": "NEU",

"enterLoginData": "Geben Sie Ihre Zugangsdaten ein",
"loggingIn": "Einloggen...",
"invalidCredentials": "Ungültige Zugangsdaten",
"nicknameEmailRequired": "Benutzername und E-Mail sind erforderlich",
"sendingCode": "Code wird gesendet...",
"registrationError": "Registrierungsfehler",

"enter6DigitCode": "Geben Sie den 6-stelligen Code ein",
"invalidCode": "Ungültiger Code",
"codeVerificationError": "Fehler bei der Codeüberprüfung",

"passwordsDoNotMatch": "Passwörter stimmen nicht überein",
"passwordTooShort": "Das Passwort muss mindestens 6 Zeichen lang sein",
"finishingRegistration": "Registrierung wird abgeschlossen...",
"finishRegistrationError": "Registrierung konnte nicht abgeschlossen werden",
"sendMessageFailed": "Nachricht konnte nicht gesendet werden",
"menu": "Menü",
"tag": "Tag",
"bio": "Biografie",
"save": "Speichern",
"yourName": "Ihr Name",
"tellAboutYourself": "Erzählen Sie etwas über sich",
"contactSearch": "Kontakte suchen",
"contactSearchPlaceholder": "Name, Nickname oder @Tag eingeben...",
"searchResults": "Suchergebnisse",
"theme": "Design",
"sound": "Ton",
"language": "Sprache",
"changePassword": "Passwort ändern",
"changePhoto": "Klicken, um das Foto zu ändern",

"deleteFromContacts": "Aus Kontakten entfernen",


```

"bioAbsent": "Keine Biografie vorhanden",

"loadedTimestamps": "Zeitstempel für Benutzer geladen",
"failedLoadTimestamps": "Zeitstempel konnten nicht geladen werden",
"failedSaveTimestamps": "Zeitstempel konnten nicht gespeichert werden",
"addToContacts": "Zu Kontakten hinzufügen",

"finishing": "Wird abgeschlossen...",
"next": "Weiter",
"noMessages": "Keine Nachrichten",
"noContactsYet": "Noch keine Kontakte",
"forgotPassword": "Passwort vergessen?",
"passwordRecovery": "Passwort-Wiederherstellung",
"enterYourEmail": "Ihre E-Mail-Adresse",
"sendCodeBtn": "Code senden",
"emailConfirmation": "E-Mail-Bestätigung",
"codeFromEmail": "Code aus der E-Mail",
"verifyCodeBtn": "Code überprüfen",
"newPasswordTitle": "Neues Passwort",
"recoveryNewPasswordPlaceholder": "Neues Passwort (mind. 6 Zeichen)",
"recoveryConfirmPasswordPlaceholder": "Neues Passwort wiederholen",
"savePasswordBtn": "Passwort speichern",
"userNotFound": "Benutzer mit dieser E-Mail wurde nicht gefunden",
"passwordResetSuccess": "Passwort erfolgreich geändert! 🎉",
"emailSubjectRegister": "Registrierungs-Bestätigungscode — Chatty",
"emailSubjectRecovery": "Passwort zurücksetzen — Chatty",
"emailHeaderRegister": "Ihr Registrierungs-Bestätigungscode",
"emailHeaderRecovery": "Anforderung zur Passwort-Wiederherstellung",
"emailBodyRecovery": "Sie erhalten diese E-Mail, da eine Passwort-Wiederherstellung für Ihr Konto angefordert wurde.",

"myContactsSection": "Ihre Kontakte",
"otherUsersSection": "Andere Benutzer",

"deleteActionTitle": "Konversation löschen",
"deleteActionDesc": "Dieser Chat enthält Nachrichtenverlauf. Wählen Sie eine Löschoption:",
"deleteOnlyChat": "✖ Nur Chat löschen",
"deleteChatAndContact": "🗑 Chat und Kontakt löschen"
}

```

ДОДАТОК Я

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Chatty</title>
  <link rel="stylesheet" href="style.css">
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/cropperjs/1.5.13/cropper.min.css">
  <script src="https://cdnjs.cloudflare.com/ajax/libs/cropperjs/1.5.13/cropper.min.js"></script>
</head>
<body>

<form id="loginForm" class="auth-wrapper" onsubmit="event.preventDefault(); login();">
  <div class="auth-logo">
    <h1>Chatty</h1>
  </div>

  <div class="auth-container">
    <input type="text" id="loginIdentifier"
      autocomplete="username"
      data-i18n-placeholder="nicknameOrEmail"
      placeholder="Нікнейм або Email" required>

    <input type="password" id="loginPassword"
      autocomplete="current-password"
      data-i18n-placeholder="password"
      placeholder="Пароль" required>

    <button type="submit" data-i18n="login">
      Увійти
    </button>

    <div id="forgotPasswordLink" class="link"
      onclick="showRecoveryStep1()"
      data-i18n="forgotPassword"
      style="display: none; margin-top: 10px;">
      Забули пароль?
    </div>

    <div class="link" onclick="showRegister()" data-i18n="noAccount">
      Немає аккаунта? Зареєструйтесь!
    </div>
  </div>
</form>

<form id="registerStep1" class="auth-wrapper" style="display:none;" onsubmit="event.preventDefault();
goToCodeVerification();">
  <div class="auth-logo">
    <h1>Chatty</h1>
  </div>

  <div class="auth-container">
    <h2 data-i18n="registration">Реєстрація</h2>

    <input type="text" id="regNickname" autocomplete="username" data-i18n-placeholder="nickname"
      placeholder="Нікнейм" required>
    <input type="text" id="regDisplayName" autocomplete="name" data-i18n-placeholder="displayName"
      placeholder="Ім'я (як вас бачать)">
    <input type="tel" id="regPhone" autocomplete="tel" data-i18n-placeholder="phone" placeholder="Телефон">
```

```

        <input type="email" id="regEmail" autocomplete="email" data-i18n-placeholder="email" placeholder="Email"
required>

        <button type="submit" data-i18n="register">
            Зареєструватися
        </button>

        <div class="link" onclick="showLogin()" data-i18n="back">
            Назад
        </div>
    </div>
</form>

<form id="registerStep2" class="auth-wrapper" style="display:none;" onsubmit="event.preventDefault();
verifyCode();">
    <div class="auth-logo">
        <h1>Chatty</h1>
    </div>

    <div class="auth-container">
        <h2 data-i18n="confirmation">Підтвердження</h2>
        <p style="text-align: center; margin-bottom: 15px;">
            <span data-i18n="codeSent">Ми відправили код на</span> <strong id="sentTo"></strong>
        </p>

        <input type="text" id="verificationCode" maxlength="6" pattern="\d{6}" data-i18n-placeholder="enterCode"
placeholder="Введіть код" required style="text-align: center; font-size: 1.3rem; letter-spacing: 4px;">

        <button type="submit" data-i18n="verify">
            Перевірити
        </button>

        <div id="resendBlock" style="margin-top:25px; text-align:center; font-size:0.92rem;">
            <span id="resendText" class="link disabled" data-i18n="resendCode">
                Відправити код ще раз (60)
            </span>
        </div>

        <div class="link" onclick="showRegister()" data-i18n="back">
            Назад
        </div>
    </div>
</form>

<form id="registerStep3" class="auth-wrapper" style="display:none;" onsubmit="event.preventDefault();
finishRegistration();">
    <div class="auth-logo">
        <h1>Chatty</h1>
    </div>

    <div class="auth-container">
        <h2 data-i18n="passwordCreate">Придумайте пароль</h2>

        <input type="password" id="regPassword" autocomplete="new-password" data-i18n-placeholder="password"
placeholder="Пароль" required>
        <input type="password" id="regPasswordRepeat" autocomplete="new-password" data-i18n-
placeholder="repeatPassword" placeholder="Повторіть пароль" required>

        <button type="submit" data-i18n="continue">
            Далі
        </button>

```

```

        <div class="link" onclick="goBackToCode()" data-i18n="back">
            Назад
        </div>
    </div>
</form>

<form id="recoveryStep1" class="auth-wrapper" style="display:none;" onsubmit="event.preventDefault();
sendRecoveryEmail();">
    <div class="auth-logo">
        <h1>Chatty</h1>
    </div>

    <div class="auth-container">
        <h2 data-i18n="passwordRecovery">Відновлення пароля</h2>

        <input type="email" id="recoveryEmail" autocomplete="email" data-i18n-placeholder="enterYourEmail"
placeholder="Ваш Email" required>

        <button type="submit" data-i18n="sendCodeBtn">
            Надіслати код
        </button>

        <div class="link" onclick="showLogin()" data-i18n="back">
            Назад
        </div>
    </div>
</form>

<form id="recoveryStep2" class="auth-wrapper" style="display:none;" onsubmit="event.preventDefault();
verifyRecoveryCode();">
    <div class="auth-logo">
        <h1>Chatty</h1>
    </div>

    <div class="auth-container">
        <h2 data-i18n="emailConfirmation">Підтвердження Email</h2>

        <input type="text" id="recoveryCode" maxlength="6" pattern="\d{6}" data-i18n-placeholder="codeFromEmail"
placeholder="Код з пошти" required style="text-align: center; font-size: 1.3rem; letter-spacing: 4px;">

        <button type="submit" data-i18n="verifyCodeBtn">
            Перевірити код
        </button>

        <div class="link" onclick="showRecoveryStep1()" data-i18n="back">
            Назад
        </div>
    </div>
</form>

<form id="recoveryStep3" class="auth-wrapper" style="display:none;" onsubmit="event.preventDefault();
submitNewPassword();">
    <div class="auth-logo">
        <h1>Chatty</h1>
    </div>

    <div class="auth-container">
        <h2 data-i18n="newPasswordTitle">Новий пароль</h2>

```

```
<input type="password" id="recoveryNewPassword" autocomplete="new-password" data-i18n-
placeholder="recoveryNewPasswordPlaceholder" placeholder="Новий пароль (від 6 символів)" required>
<input type="password" id="recoveryConfirmPassword" autocomplete="new-password" data-i18n-
placeholder="recoveryConfirmPasswordPlaceholder" placeholder="Повторіть новий пароль" required>
```

```
<button type="submit" data-i18n="savePasswordBtn">
    Зберегти пароль
</button>
</div>
</form>
```

```
<div id="messenger" style="display:none;">
```

```
<div class="side-panel left-panel">
```

```
<div id="noChatMediaPlaceholder" class="media-placeholder">
    <div class="placeholder-content">
        <div class="placeholder-icon"> 📁 </div>
        <h3 data-i18n="selectChat">Вибери чат</h3>
        <p data-i18n="chatMediaHere">Тут буде медіа вибраного чату</p>
    </div>
</div>
```

```
<div id="mediaContent" class="media-content-wrapper" style="display:none;">
```

```
<div class="media-tabs">
    <div class="tab active" data-tab="photos" data-i18n="photos"
onclick="switchMediaTab('photos')">Фото</div>
    <div class="tab" data-tab="videos" data-i18n="videos" onclick="switchMediaTab('videos')">Відео</div>
    <div class="tab" data-tab="files" data-i18n="files" onclick="switchMediaTab('files')">Файли</div>
</div>
```

```
<div class="media-slider-wrapper">
    <div class="media-slider" id="mediaSlider">

        <div class="media-page">
            <div class="media-grid" id="photosGrid"></div>
        </div>

        <div class="media-page">
            <div class="media-grid" id="videosGrid"></div>
        </div>

        <div class="media-page">
            <div class="files-list" id="filesList"></div>
        </div>

    </div>
</div>
```

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

```
<div class="chat-container">
```

```
<div class="chat-header" id="chatHeader">
    <div onclick="showChatInfo()" class="chat-avatar" id="chatAvatar"> 👤 </div>
    <div class="chat-info">
        <h3 id="chatHeaderName">Чат не вибрано</h3>
        <span id="chatHeaderStatus" class="status online">онлайн</span>
    </div>
```

```

        <div class="chat-header-actions"></div>
    </div>

    <div id="noChatPlaceholder" class="no-chat-placeholder">
        <div class="placeholder-content">
            <div class="placeholder-icon">🗉</div>
            <h3 data-i18n="chooseChat">Оберіть чат</h3>
            <p data-i18n="chooseChatDescription">Виберіть контакт справа,<br>щоб почати розмову</p>
        </div>
    </div>

    <div class="messages" id="messages" style="display: none;"></div>

    <div id="filePreviewContainer" class="file-preview-bar" style="display: none;">
    <div class="file-preview-content">
        <span class="file-icon">📎</span>
        <span id="filePreviewName">photo_2026.jpg</span>
    </div>
    <button type="button" onclick="clearFilePreview()" class="file-preview-remove">X</button>
</div>

    <div class="input-area" id="messageInputArea" style="display: none;">
        <button type="button" id="attachBtn">📎</button>
        <input type="file" id="messageFile" style="display: none;">

        <div id="filePreview" class="file-preview" style="display: none;">
            <div id="filePreviewContent" class="file-preview-content"></div>
        </div>

        <input type="text" id="messageText" data-i18n-placeholder="messagePlaceholder" placeholder="Напишіть повідомлення..." autocomplete="off">
        <button type="button" onclick="sendMessage()">➤</button>
    </div>
</div>

<div class="side-panel right-panel">
    <div class="panel-header">
        <h3 id="panelTitle" data-i18n="chats">Чату</h3>
        <button id="menuToggle" class="menu-btn">☰</button>
    </div>

    <div class="chat-list-container">
        <div class="chat-list" id="chatList"></div>
    </div>

    <div id="menuLayer" class="menu-layer">
        <div class="menu-content">
            <div class="menu-panel active" id="menuContent">
                <div class="menu-item" data-panel="profileMenu">👤 <span data-i18n="myProfile">Мій профіль</span></div>
                <div class="menu-item" data-panel="contactsMenu">👥 <span data-i18n="contacts">Контакти</span></div>
                <div class="menu-item" data-panel="settingsMenu">⚙️ <span data-i18n="settings">Налаштування</span></div>
                <div class="menu-item exit">🚪 <span data-i18n="logout">Вийти</span></div>
            </div>

            <div class="menu-panel" id="profileMenu"></div>
            <div class="menu-panel" id="contactsMenu"></div>
        </div>
    </div>

```

```
        <div class="menu-panel" id="settingsMenu"></div>
    </div>
</div>
</div>

<div id="avatarModal" class="modal" style="display: none;">...</div>

<script src="/i18n.js"></script>
<script src="app.js"></script>
<script src="/socket.io/socket.io.js"></script>
```

```
<div id="contactModal" class="contact-modal">
    <div class="contact-modal-content">
        <button class="contact-modal-close" onclick="closeContactModal()">X</button>
        <div id="modalAvatar" class="modal-avatar">  </div>
        <div id="modalName" class="modal-name"></div>
        <div id="modalTag" class="modal-tag"></div>
        <div id="modalBio" class="modal-bio" data-i18n="bioEmpty">Біографія відсутня</div>
        <div id="modalActions" class="modal-actions"></div>
    </div>
</div>
```

```
<div id="mediaLightbox" class="media-lightbox">
    <div class="lightbox-overlay" onclick="closeLightbox()"></div>
```

```
    <div class="lightbox-content">
        <button class="lightbox-close" onclick="closeLightbox()">X</button>
        <div id="lightboxMediaContainer" class="lightbox-media-container"></div>
```

```
    <div class="lightbox-actions">
        <button onclick="goToMessageFromLightbox()" class="lightbox-btn" data-i18n="goToMessage">
            Перейти до повідомлення
        </button>
        <button onclick="downloadCurrentMedia()" class="lightbox-btn" data-i18n="download">
            Завантажити
        </button>
    </div>
```

```
    <button class="lightbox-nav prev" onclick="prevMedia()">&#8249;</button>
    <button class="lightbox-nav next" onclick="nextMedia()">&#8250;</button>
```

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

```
<div id="passwordModal" class="contact-modal">
    <div class="contact-modal-overlay" onclick="closePasswordModal()"></div>
    <div class="contact-modal-content">
        <button class="contact-modal-close" onclick="closePasswordModal()">X</button>

        <div class="modal-name" style="margin-bottom: 20px;">Зміна пароля</div>
```

```
        <div id="passwordModalAlert" style="color: #ef4444; font-size: 0.9rem; margin-bottom: 15px; text-align: center;
display: none;"></div>
```

```
<div id="passwordStep1">
  <div style="margin-bottom: 15px; text-align: left;">
    <label style="display: block; margin-bottom: 5px; font-size: 0.85rem; color: #94a3b8;">Поточний
    пароль</label>
    <input type="password" id="oldPasswordInput" style="width: 100%; padding: 10px; background: #1e293b;
    border: 1px solid #334155; border-radius: 8px; color: #fff; font-size: 1rem;" placeholder="Введіть старий пароль">
  </div>
  <button onclick="verifyOldPassword()" class="password-submit-btn" style="width: 100%; margin-top:
  10px;">Далі</button>
</div>

<div id="passwordStep2" style="display: none;">
  <div style="margin-bottom: 12px; text-align: left;">
    <label style="display: block; margin-bottom: 5px; font-size: 0.85rem; color: #94a3b8;">Новий
    пароль</label>
    <input type="password" id="newPasswordInput" style="width: 100%; padding: 10px; background: #1e293b;
    border: 1px solid #334155; border-radius: 8px; color: #fff; font-size: 1rem;" placeholder="Не менше 6 символів">
  </div>
  <div style="margin-bottom: 15px; text-align: left;">
    <label style="display: block; margin-bottom: 5px; font-size: 0.85rem; color: #94a3b8;">Повторіть новий
    пароль</label>
    <input type="password" id="confirmPasswordInput" style="width: 100%; padding: 10px; background:
    #1e293b; border: 1px solid #334155; border-radius: 8px; color: #fff; font-size: 1rem;" placeholder="Повторіть
    пароль">
  </div>
  <button onclick="saveNewPassword()" class="password-submit-btn" style="width: 100%; margin-top: 10px;
  background: linear-gradient(135deg, #10b981, #059669); box-shadow: 0 4px 12px rgba(16, 185, 129,
  0.2);">Зберегти новий пароль</button>
</div>
</div>

</body>
</html>
```