НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ
Завідувач кафедри
комп'ютерних наук
(назва кафедри)

_____ Голуб Б.Л.
(підпис) (ПІБ)

«__» _____2025 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему

«Програмне забезпечення веб сайту для обміну повідомленнями в режимі реального часу з підтримкою основних функцій мессенджера»

Спеціальність 121 «Інженерія програмного забезпечення»

Гарант освітньої програми
к.т.н., доцент _____ Вайганг Г.О.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Керівник бакалаврської кваліфікаційної роботи
асистент _____ Баранова Т.А.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Консультант бакалаврської кваліфікаційної роботи
к.т.н., доцент _____ Даков С.Ю.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ Дударенко Н.І.
(підпис) (ПІБ студента)

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних наук
(назва кафедри)

_____ Голуб Б.Л.
(підпис) (ПІБ)

«__» _____2025 р.

З А В Д А Н Н Я
на виконання бакалаврської кваліфікаційної роботи студенту
Дударенко Нікіті Івановичу
(прізвище, ім'я, по батькові)

Спеціальність 121 – «Інженерія програмного забезпечення»
Тема бакалаврської кваліфікаційної роботи Програмне забезпечення веб сайту для обміну повідомленнями в режимі реального часу з підтримкою основних функцій мессенджера
затверджена наказом ректора НУБіП України від "16" 12 2024р. №2248 "С"
Термін подання завершеної роботи на кафедру 2025.05.
(рік, місяць, число)
Вихідні дані до бакалаврської кваліфікаційної роботи
опис програмного забезпечення
Перелік питань, які потрібно розробити:
Аналіз предметної області, проектування програмного забезпечення, розробка програмного забезпечення, впровадження
Дата видачі завдання "_16_" __12_____2024 р.
Керівник бакалаврської кваліфікаційної роботи
асистент _____ Баранова Т.А.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Консультант бакалаврської кваліфікаційної роботи
к.т.н., доцент _____ Даков С.Ю.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Завдання прийняв до виконання _____ Дударенко Н.І.
(підпис) (ПІБ студента)

ЗМІСТ

ВСТУП

У сучасному світі комунікаційні застосунки є важливою частиною повсякденного життя — як у особистому, так і в професійному плані. Користувачам потрібні швидка доставка повідомлень, надійний обмін інформацією, можливість керувати списком контактів і знати, хто є онлайн. Це створює потребу в рішеннях для інтеграції обміну повідомленнями у системи підприємств або в застосунки, доступні для обмеженого кола користувачів.

Зокрема, потреба виникає у веб-застосунках обміну повідомленнями, які не вимагають встановлення додатка, завдяки чому доступ до ними є легким та кросплатформеним. Ще одним важливим питанням є надійне керування доступом, яке необхідне для підтримки приватності користувачів та захисту даних. Це робить створення кастомного рішення для обміну повідомленнями важливим завданням, яке дозволяє реалізувати специфічні вимоги.

Мета розробки — створення сучасного веб-застосунку для обміну повідомленнями, який підтримуватиме основний функціонал типової системи обміну повідомленнями: можливість авторизуватись, реєструватись, створювати приватні й групові чати, писати текстові повідомлення, переглядати профілі користувачів, змінювати налаштування облікового запису тощо.

Програмне рішення має забезпечити:

- відгук на дії користувача у реальному часі за допомогою WebSocket-з’єднань (SignalR);
- безпечне збереження даних з використанням сучасної ORM (Entity Framework + PostgreSQL);
- можливість масштабування у майбутньому;
- інтуїтивно зрозумілий інтерфейс без надмірної складності.

Завдяки застосуванню Blazor Server користувач отримує відчуття роботи з настільним додатком при збереженні усіх переваг веб-архітектури. Розробка

програмного забезпечення виконувалася із використанням сучасного стеку технологій, орієнтованого на створення продуктивного, безпечного та зручного у використанні веб-додатку.

Як платформа клієнт-сервера була використана Blazor Server, що в практиці реалізовує динамічні інтерфейси за допомогою C# з найменшою можливістю використання JavaScript. SignalR бібліотека допомагає обміну даними в реальному часі, завдяки чому учасники чата отримують повідомлення миттєво.

Для доступу до бази даних було вибрано ORM Entity Framework Core, який працює у поєднанні з PostgreSQL через Npgsql — офіційний драйвер. Користувачам мається можливістю використовувати кращий підхід за допомогою DbContext для пошуку даних і таким чином зберігати чисту структуру коду (clean architecture) проекту. Де зберігається інформація — це про користувачів, чати, повідомлення та інші дані, які пов’язані з автентифікацією.

Через інтегровану систему ASP.NET Core Identity, в процесі мають змогу працювати з авторизованими користувачами, тобто одночасно реєстрація, авторизація, підтвердження електронної пошти, зміна паролю, інші операції веб контролю належать обліковим записам користувачів. Реалізаціяінтерфейсної частини продукту виконана за допомогою Bootstrap and CSS-стилів, що дозволяє адаптивність інтерфейсу та захищає комфортність користувача.

Усі вибрані технології поєднуються та надають ефективну роботу системи, її надійність, масштабованість.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис предметної області

В умовах стрімкого росту цифрових інтерфес технологій, а також в результаті переходу до онлайн-комунікацій особливу вагу набувають такі системи, як системи миттєвого обміну повідомленнями — месенджери. Ці системи єневід'ємнимидля особистого спілкування, корпоративних підприємств, освітньої галузі, служби підтримки та багатьох інших сфер, де надзвичайно важливо забезпечити оперативну та надійну передачу інформації між користувачами в реальному часі.

У цьому віці месенджери можуть бути корисними як інструменти пошуку і обміну ідеями, так і вирішення подібних проблем. Наприклад, діти можуть бути стимульовані для навчання, якщо знайдуть спосіб веселим та цікавим. В освітній сфері месенджер може бути корисний як спосіб здобуття професійних навичок по вирішенню проблем хоча б і в певній мірі.

В рамках данної дипломної роботи реалізован веб-додаток для обміну повідомленнями в реальному часі, створена за допомогою технології Blazor Server — сучасного інструмента розробки від компанії Microsoft, що дозволяє будувати інтерактивні веб-інтерфейси з використанням мови програмування C# без потреби залучення JavaScript. Для зберігання даних використовується СУБД PostgreSQL, а для реалізації двостороннього обміну даними між клієнтом і сервером застосовано бібліотеку SignalR, що дозволяє легко використовувати технологію WebSocket, що дозволяє досягти високої чуйності та мінімізації затримок у передачі інформації.

Серед основної реалізованої функціональності можна виділити: можливість авторизації, створення нового аккаунту, створення приватних та групових чатів, можливість знаходити потрібних користувачів по логіну, обмін

повідомленнями з автоматичним оновленням інтерфейсу, перегляд профілю, можливість змінювати своє ім’я, логін та пароль, вихід з чатів, модальне управління профілем та ще декілька дрібних можливостей щоб закрити базові потреби користувача месенджера.

Таким чином, предметною областю дослідження є створення інформаційної системи миттєвого обміну повідомленнями, яка відповідає сучасним викликам цифрового середовища. Враховуючи високу конкуренцію у сфері месенджерів,доцільним є аналіз існуючих аналогів, на основі якого можливо виокремити оптимальні рішення та сформулювати вимоги до власної програмної системи.

1.2 Огляд інформаційних джерел та існуючих рішень

Разглядаючи сучасні системи обміну миттєвими повідомленнями, ми можемо відзначити, що відповідні системи різних категорій користувачів, кожна має свої особливості, які призвели їх для різних груп користувачів популярними для обміну миттєвими повідомленнями.

Основною особливістю Telegram є використання власного протоколу MTProto, завдяки чому він є швидким, надійним і відносно безпечним, навіть у разі великого завантаження інфраструктури можна використовувати програму без проблем. Підтримує хмарне сховище, має можливість забезпечити енд-ту-енд шифрування в секретному чаті. Використовує відкрите API для розробників, що є конкурентною перевагою в сучасному світі, так як це означає здатність створення як публічних, так і приватних ботів для автоматизації деяких процесів, безкоштовну інтернет-модерацію та розвиток значних спільнот.

Discord є платформою, що почала свій шлях як інструмент комунікації геймерів і перетворилася на масштабуючу ідеологію приватних соціальних спільнот. Використовує протокол WebRTC, що дозволяє знизити навантаження на мережу і знизити затримки часу розмови, що дозволяє і далі працювати навіть при великому навантаженні на мережу платформи. Основну структуру програми

реалізовано в основному за допомогою вебсокетів, що об'єднують юзера з гільдіями, каналами і ще з іншими користувачами. Все це надає можливість створювати власні серверні кластери для зборів великовеличезних конференцій, а також комунікації з спікером великої аудиторії.

WhatsApp використує понад два мільярди людей та став надзвичайно популярним. Це — сучасне програмне забезпечення, яке дозволяє обмінюватися миттєвими повідомленнями зі своїми знайомими та поширювати медіафайли. Він використовує протокол Signal Protocol як метод захисту повідомлень та забезпечує надійне криптографічне шифрування за замовчуванням. Вбудованих ресурсів достатньо для повного продуктивного управління груповими чатами та безпеці каналу спільного зв’язку.

Signal — служба миттєвого спілкування, що працює через мобільний додаток чи веб-рішення. Сфокусований на безпеці користувачів та їхньої конфіденційності, використовує свій власний протокол для забезпечення надійного криптографічного шифрування – Signal Protocol. Відсутність збирання метаданих про користувачів, всі збережені інформації є закодовані сильними асиметричними алгоритмами шифрування. Єдиний недолік будується на тім, що він не має найсучасніших функцій автоматизації чи модерування, а експлуатується на повному відсутність впливу на користувачів та забезпечення швидкого обміну миттєвими повідомленнями.

Facebook Messenger об'єднується з соціальною мережею Facebook, тому користувачам надаються різноманітні можливості, від текстових та мультимедійних повідомлень до відеодзвінків та взаємодії з ботами. Відтоді, як у грудні 2022 року Messenger запровадив за замовчуванням кінцевого-до-кінця шифрування, конфіденційність вкотре покотилася на нові рубежі. Проте залишається проблемою те, що наскрізне шифрування працює тільки в спеціальному режимі, і це визначний недолік для тих, хто шукає збільшену конфіденційність.

Таким чином, аналіз сучасних систем обміну повідомленнями дозволяє виявити такі ключові особливості, які потрібно враховувати при розробці власного програмного рішення: б розмір та охорона інформації, масштабність та продуктивність, інтеграції та інтероперабельність, зручність користування та досвід користувача. Ці аспекти складають прийняття рішень, кількість пропозицій і вартість розробутого зараз рушникового засобу для обміну повідомленнями.

1.3 Аналіз вимог до програмної системи

Проектування системи миттєвого обміну повідомленнями вимагає чіткого визначення функціональних і нефункціональних вимог, які охоплюють її цілі, можливості, очікування кінцевих користувачів щодо взаємодії з нею, а також параметри якості, які безпосередньо впливають на її коректність, надійність і по-треби хостингу та розгортання. У цьому розділі систематизовано основні функціональні та нефункціональні вимоги до системи миттєвих повідомлень з урахуванням передових стандартів розробки

1.3.1 Функціональні вимоги

Функціональні вимоги визначають основні дії, які система повинна виконувати для забезпечення потреб користувача. У межах реалізованої програмної системи обміну повідомленнями в реальному часі було виділено такі ключові функціональні можливості:

- Реєстрація та авторизація користувачів: Користувач повинен мати змогу створити обліковий запис за допомогою введення унікального логіну, паролю та адреси електронної пошти. Після цього можливий вхід до системи шляхом авторизації. Всі операції захищено за допомогою JWT-токенів та HTTPS.

- Обмін повідомленнями у режимі реального часу: Після авторизації користувач отримує доступ до інтерфейсу чату, де може надсилати й отримувати текстові повідомлення без перезавантаження сторінки. Передавання повідомлень відбувається миттєво завдяки використанню WebSocket, що забезпечує двосторонній зв’язок між клієнтом і сервером.
- Створення та перегляд чатів: Користувач може створювати нові приватні чати (на двох осіб) та переглядати список доступних чатів. Інтерфейс автоматично оновлюється при отриманні нових повідомлень або зміні даних чату.
- Перегляд історії листування**:** Після входу в чат користувач має змогу переглянути попередні повідомлення. Дані завантажуються з бази даних PostgreSQL, і підтримується коректне форматування повідомлень із зазначенням часу надсилання.
- Відображення активних чатів та повідомлень: Головна сторінка месенджера містить список чатів з останніми повідомленнями, що дозволяє швидко орієнтуватися в активному листуванні. Індикатори нового повідомлення або непрочитаних діалогів можуть бути реалізовані за потреби.
- Можливість змінити ім’я користувача або аватар: Базовий функціонал профілю передбачає можливість оновлення деяких даних користувача через інтерфейс налаштувань.
- Збереження повідомлень і стану чатів у базі даних: Всі повідомлення зберігаються у структурованому вигляді в базі даних, що забезпечує цілісність та можливість подальшого аналізу чи розширення функціоналу (наприклад, фільтрація або пошук).
- Обробка помилок і повідомлення користувача: У разі виникнення технічних проблем (наприклад, втрата з'єднання з WebSocket або неправильні дані авторизації) система інформує користувача відповідним повідомленням.

1.3.2 Нефункціональні вимоги

Нефункціональні вимоги визначають характеристики, які не пов'язані безпосередньо з поведінкою функцій системи, але суттєво впливають на її зручність, ефективність, підтримку та масштабування. Для розробленої системи обміну повідомленнями в реальному часі визначено такі важливі атрибути якості:

Система повинна мати високу продуктивність і забезпечувати миттєвий обмін повідомленнями. Для цього реалізовано двосторонній обмін даними між клієнтом і сервером за допомогою WebSocket, що дозволяє мінімізувати затримки в порівнянні з традиційними HTTP-запитами. Оскільки кешування даних не реалізовано, швидкість відповіді залежить від продуктивності бази даних та загального навантаження на сервер.

Система повинна бути надійною – повідомлення відразу ж додаватися в базу даних, що повинно забезпечити збереження історії навіть у випадку перезапуску сервера або оновлення сторінки користувачем. Механізми обробки помилок будуть показані користувачу у випадку проблем з мережею або неправильним вводом, що також підвищує надійність системи.

Безпека реалізована на за допомогою передачі даних через HTTPS та авторизації із використанням JWT токенів. Усі запити авторизованих користувачів перевіряються, що виключає неавторизований доступ до повідомлень. Збереження чутливої інформації клієнта real-time (останній активний чат) реалізовано через protected storage - зменшує ризики втрати або крадіжки публічною інформацією зловмисників.

Хоча масштабованість не реалізована у повному обсязі, система має модульну архітектуру, що дозволяє при потребі розширювати функціонал або адаптувати її до зростання кількості користувачів (наприклад, шляхом додавання контейнеризації у майбутньому).

Масштабованість не реалізована у повноцінна, але система має модульну архітектуру, що дозволяє при потребі розширити функціонал або адаптувати її

до зростання кількості користувачів (наприклад, додавання контейнеризації у майбутньому).

Інтерфейс орієнтований для персональних комп’ютерів і забезпечує зрозумілу навігацію, просту структуру сторінок і добре візуальне розділення елементів інтерфейсу. Адаптивність для мобільних пристроїв на даний момент не передбачена.

З точки зору підтримки, програмну систему створено на базі фреймворків .NET та Blazor Server, що спрощує модифікацію коду, а також тестування та доопрацювання. Використано архітектурні підходи, що забезпечують розділення областей відповідальності окремих компонентів програмної системи, робота з базою даних виділена в окремий клас на базі DbContext.

Таким чином, реалізована система демонструє достатній рівень якості для локального або внутрішньомережевого використання з можливістю подальшого масштабування, розширення функціональності та впровадження інструментів забезпечення високої доступності та продуктивності.

1.4 Постановка завдання

Метою цього дипломного проєкту є створення програмного забезпечення для обміну повідомленнями в реальному часі, яке повинно надавати основні функції месенджера, а саме: текстові чати, групове спілкування та зручний інтерфейс.

Для досягнення цієї мети планується розробка веб-додатку на основі клієнт-серверної архітектури з використанням новітніх технологій. Система повинна обробляти повідомлення з мінімальними затримками, забезпечувати високий рівень безпеки та бути зручною для кінцевого користувача.

Щоб реалізувати цю мету, продумано виконання таких завдань:

- визначити функціональні та нефункціональні вимоги до програмної системи;

- проаналізувати існуючі рішення у сфері обміну повідомленнями та виділити найбільш релевантні підходи;
- спроєктувати архітектуру програмної системи з урахуванням вимог до масштабованості, продуктивності та підтримуваності;
- реалізувати серверну частину системи з використанням ASP.NET Core та WebSocket для забезпечення обміну повідомленнями в реальному часі;
- реалізувати клієнтську частину за допомогою Blazor Server, забезпечивши динамічне оновлення повідомлень та збереження контексту обраного чату;
- забезпечити авторизацію та автентифікацію користувачів;
- протестувати основні функції системи.

1.5 Моделювання предметної області

1.5.1 Загальні положення

Аналіз предметної області в процесі розробки системи дозволяє виокремити основні компоненти системи, визначити логіку їх роботи та взаємодії, а також визначити межі та загальну структуру програми.

Моделювання предметної області дозволяє розробникам системи та користувачу програмного забезпечення отримати розуміння принципів роботи системи. Завдяки чому вдається узгодити вимоги до програмного продукту, знизити кількість логічних помилок та спростити всі наступні етапи розробки – від написання коду, до його тестування та підтримки.

В даному пункті описано моделі, які відображають основні процеси та взаємозв’язки для системи обміну повідомленнями в реальному часі. Ці моделі дозволяють сформувати архітектуру додатку та визначити його основні компоненти для реалізації основного функціоналу.

1.5.2 Діаграми прецедентів

Діаграми варіантів використання використовуються для візуалізації потреб клієнтів у системі, яка підлягає розробці. Вони відображають функціональність з точки зору кінцевого користувача і пов'язані з існуючими бізнес-сценаріями або процесами. Це засіб для комунікації між кінцевим користувачем та проектувальником системи.

Для проекту системи обміну миттєвими повідомленнями в реальному часі, було підготовлено кілька діаграм прецедентів (варіантів використання) цієї системи. Діаграми варіантів використання можна розділити на декілька функціональних областей, які містять логічний набір варіантів використання цієї функціональної частини системи, що підлягає розробці. Існують три загальні області: аутентифікація, чати та зміни профілю.

На діаграмі прецедентів «Аутентифікація» (рис. 1) показано дії, що виконуються користувачем під час взаємодії із системою на при вході або реєстрації в системі. Головним актором є користувач, який може Зареєструватися, Авторизуватися та Вийти з акаунту. Для того, щоб були введені коректні дані під час реєстрації та авторизації реалізовано прецедент Перевірити введені дані, який включається (<<include>>) в обидва головні випадки. Тобто вже на початку взаємодії з системою забезпечується перевірка обов’язкових полів та перевірка введеної інформації, це підвищує загальну надійність програмного забезпечення.

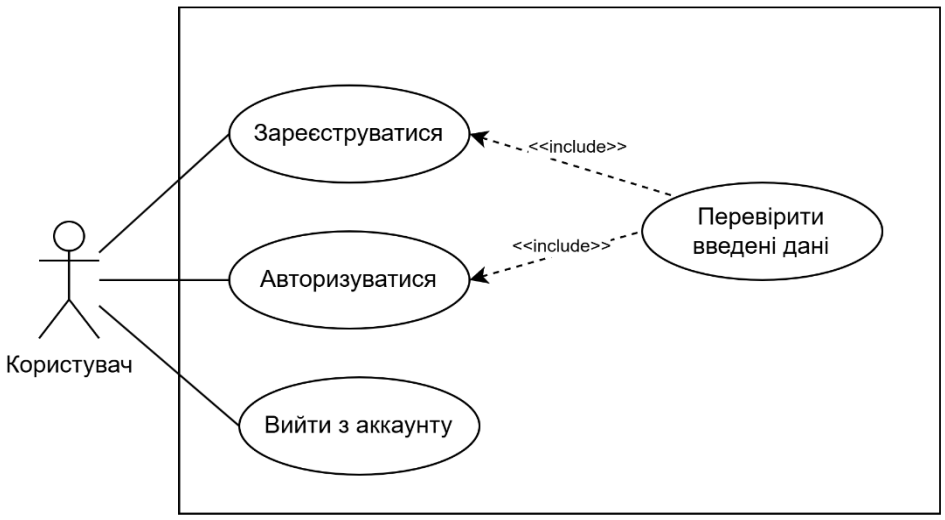

Рис. 1 Діаграма прецедентів: Аутентифікація

На діаграмі прецедентів «Чати» (рис. 2) відображені сценарії, що стосуються використання чатів. Єдиним актором є користувач, він взаємодіє з широким спектром дій, таких як: створити приватний чат, створити груповий чат, відкрити приватний чат, відкрити груповий чат, переглянути історію повідомлень, написати повідомлення, видалити повідомлення, видалити чат, вийти з групового чату, додати учасника до групового чату. Така деталізація показує основні і допоміжні операції, що можуть виконуватись користувачем під час спілкування в чатах. Варіанти використання охоплюють весь життєвий цикл чату – від його створення для приватного чи групового спілкування до можливого видалення або виходу з нього, а також операції з управління вмістом повідомлень.

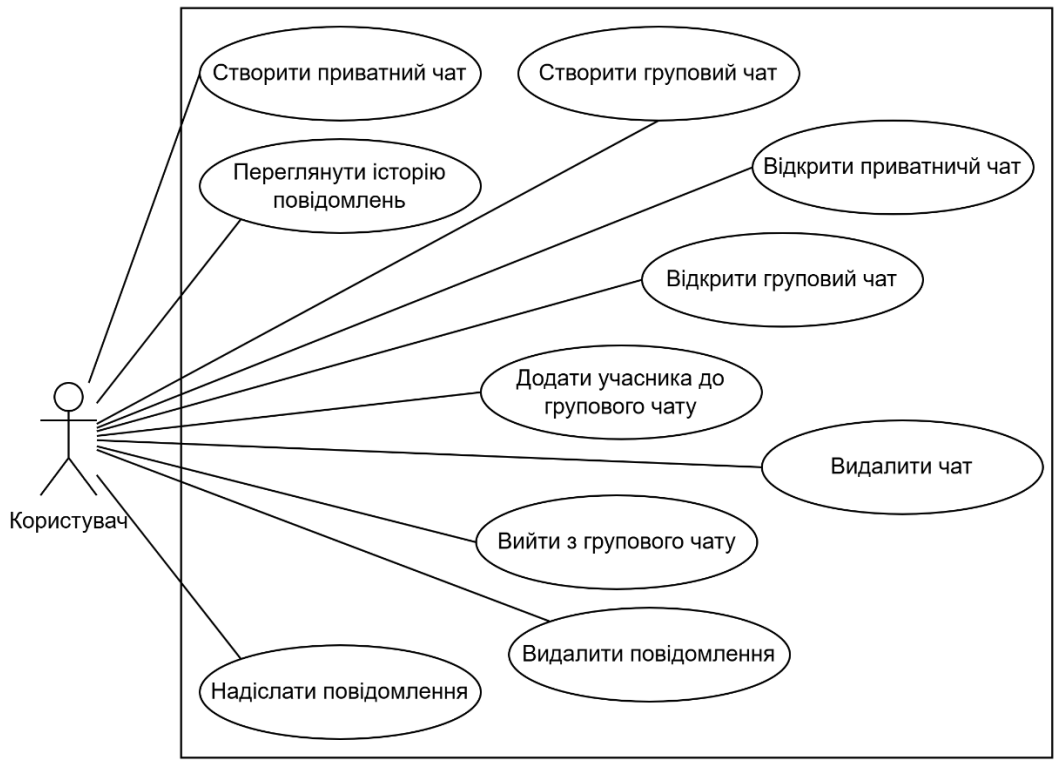

Рис. 2 Діаграма прецедентів: Чати

На діаграмі прецедентів «Налаштування профілю» (рис. 3) розглянуто дії, які дозволяють керувати персональними налаштуваннями. Доступні сценарії це: Переглянути профіль користувача, Змінити відображуване ім’я, Змінити пароль, Змінити пошту, Підтвердити пошту, Додати опис користувача.

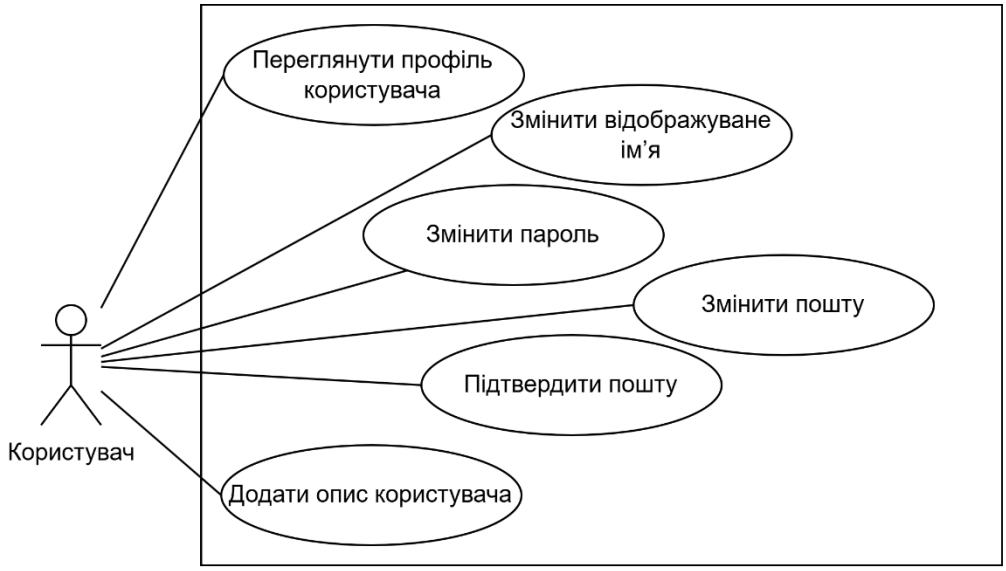

Рис. 3 Діаграма прецедентів: Налаштування профілю

Для більш детального опису прецедентів див. додаток А.

1.5.3 Діаграми послідовності

Більш детально описувати процеси обміну повідомленнями між об'єктами системи під час виконання того чи іншого випадку використання слід за допомогою діаграм послідовності (англ. Sequence diagram). Це дозволяє описати порядок відправлення повідомлень актором в ході виконання випадку використання, а також те, як ці повідомлення обробляються у системі. Такі діаграми можуть створюватися для опису конкретних випадків використання зі складною логікою взаємодії з користувачем, які потребують виконання додаткових кроків валідації, перевірки тощо.

В системі було розроблено чотири діаграми послідовності, які описують наступні випадки використання: реєстрація, авторизація, створення чату та відправлення повідомлень, а також додавання користувача до групового чату. Опис діаграм наведено нижче.

На діаграмі послідовності «Реєстрація» (Рис. 4) зображено процес багаторівневої взаємодії між актором Користувачем та Системою у сценарії створення нового облікового запису. Користувач спочатку заповнює форму реєстрації (внутрішня дія актору), після чого надсилає дані в систему. Система

виконує перевірку правильності заповнення даних (внутрішня обробка) й у разі невдалого вводу повертає повідомлення про помилку. Користувач здійснює повторне введення, на цей раз система виявляє, що даний користувач вже зареєстрований, й знову повертає повідомлення про помилку. Лише після третьої спроби, коли дані є ідентифіковані як коректні й унікальні, система зберігає обліковий запис та підтверджує повідомленням успішне завершення операції.

Рис. 4 Діаграма послідовності – Реєстрація

На діаграмі послідовності «Авторизація» (Рис. 5) передбачено два цикли взаємодії. Користувач вводить логін і пароль, які надсилає до системи. Система виконує перевірку й повертає повідомлення про некоректні дані. Після повторного введення та перевірки, якщо інформація є достовірною, система дозволяє доступ до облікового запису та підтверджує успішну авторизацію. Усі

основні кроки діалогу між користувачем та системою реалізовано за допомогою послідовних обмінів повідомленнями.

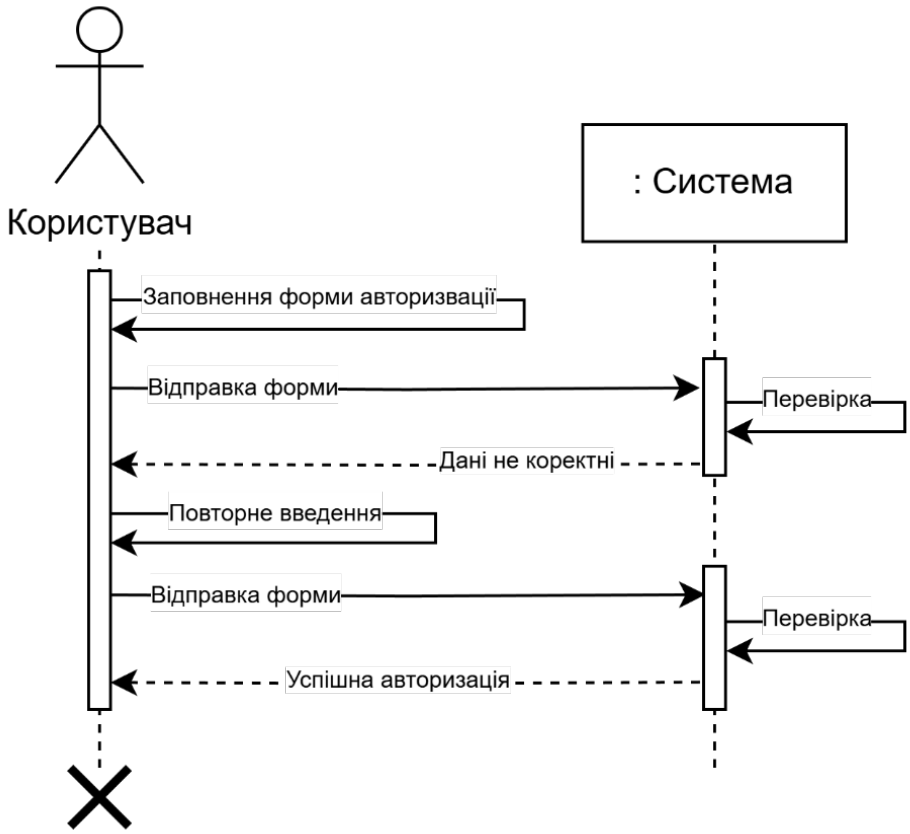

Рис. 5 Діаграма послідовності: Авторизація

На діаграмі послідовності «Створення чату і відправка повідомлення» (Рис. 6), беруть участь три учасники: Користувач 1, Користувач 2 та Система. Користувач 1 ініціює пошук іншого користувача (внутрішня дія), після чого надсилає запит системі на створення приватного чату. Система обробляє запит, створює новий чат і надсилає обом учасникам повідомлення про його появу. Далі Користувач 1 надсилає повідомлення в чат, система його зберігає й оперативно передає обом користувачам оновлення інтерфейсу із відображенням нового повідомлення. Таким чином, діаграма демонструє як ініціацію зв’язку, так і основну функцію обміну повідомленнями.

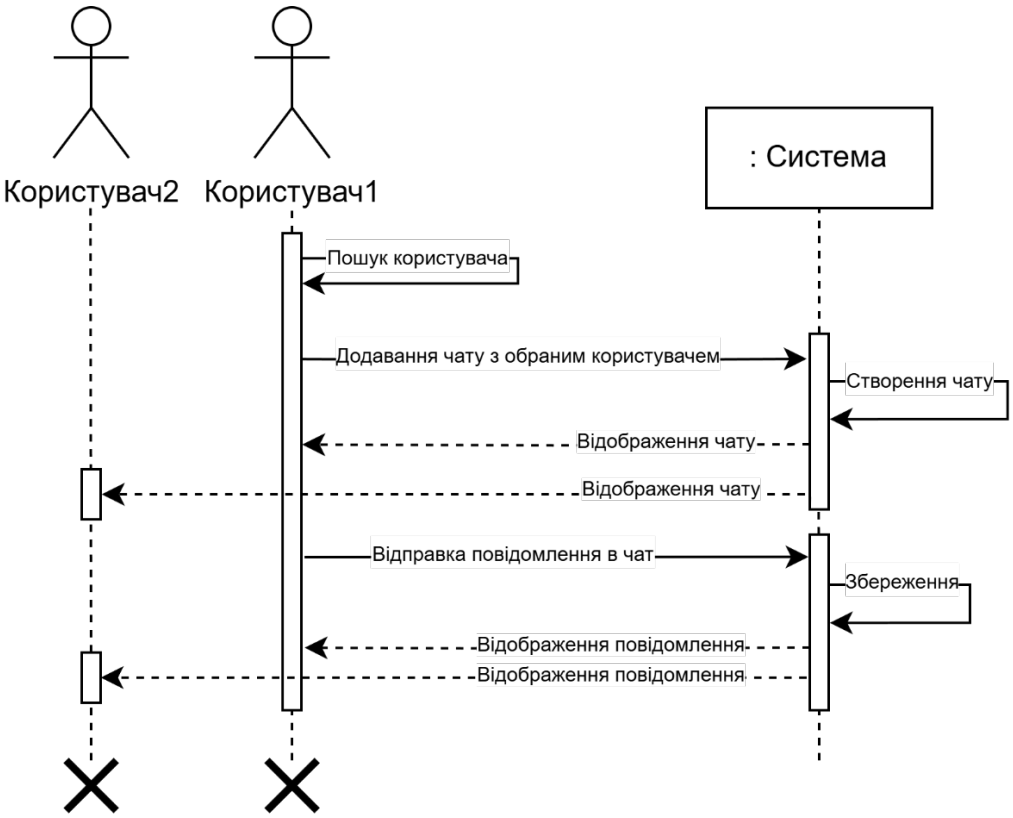

Рис. 6 Діаграма послідовності: Реєстрація

На завершення, діаграма «Додавання користувача до групового чату» (Рис. 7) відображає процес додавання нового учасника до наявного групового чату. У цьому процесі беруть участь Користувач 1, Користувач 2 та Система. Користувач 1 передає запит із зазначенням імені користувача. Система виконує пошук, повертає список можливих кандидатів. Після вибору одного з них, Користувач 1 підтверджує додавання, система зберігає зміну та оновлює відображення – Користувачу 1 показується оновлений список учасників, а Користувачу 2 – оновлений інтерфейс групового чату, що свідчить про його успішне приєднання.

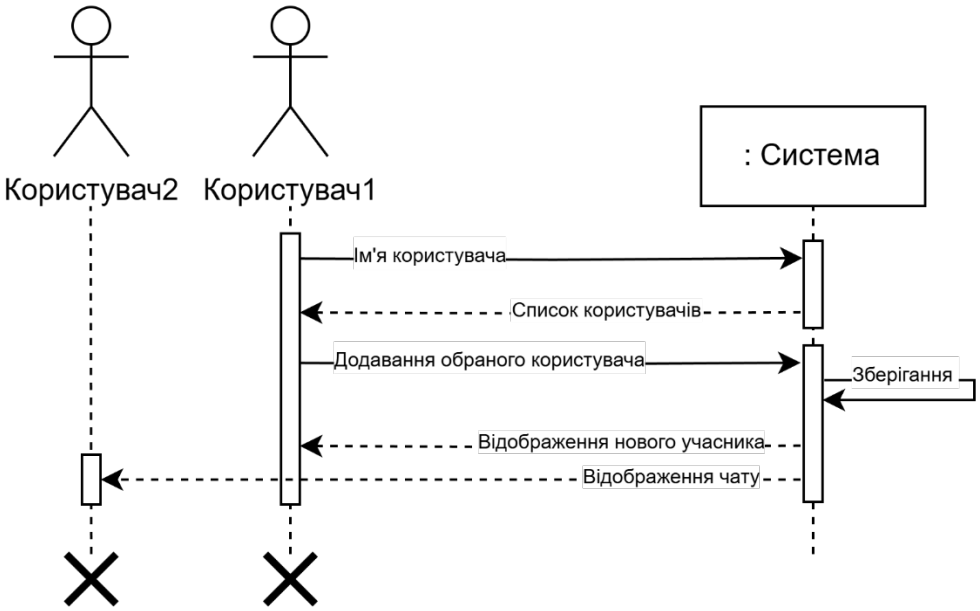

Рис. 7 Діаграма послідовності: Авторизація

1.5.4 Діаграма активності

Діаграми активності використовуються для моделювання потоків управління та логіки виконання процесу в межах окремих варіантів використання функціоналу системи. Такі діаграми особливо корисні для ілюстрації покрокового виконання алгоритмів або дій, які мають умовні переходи чи повторення або розгалуження. Таким чином, можна зрозуміти, як саме відбувається взаємодія між користувачем та системою у відповідь на певні дії, а також які перевірки здійснює та які рішення приймає система в процесі виконання певної задачі.

На Рис. 8 представлено діаграму активності, що моделює процес зміни налаштувань профілю користувача, а саме – редагування відображуваного імені.

Діаграма має дві часові доріжки (swimlanes), які відповідають за дії Користувача та Системи. Взаємодія починається на стороні користувача з перегляду сторінки чатів, після чого ініціюється перехід до налаштувань профілю. Користувач вводить нове відображуване ім’я та підтверджує зміну.

Після цього управління передається до системи, яка виконує перевірку коректності введених даних. Далі відбувається розгалуження:

- у випадку некоректного введення система виводить відповідне повідомлення про помилку і повертає користувача назад до налаштувань для повторного введення;
- у разі коректного введення система зберігає зміни, оновлює ім’я користувача, після чого передає управління назад користувачу.

Під кінець користувач переглядає оновлену інформацію у своєму профілі, чим і завершується діаграма.

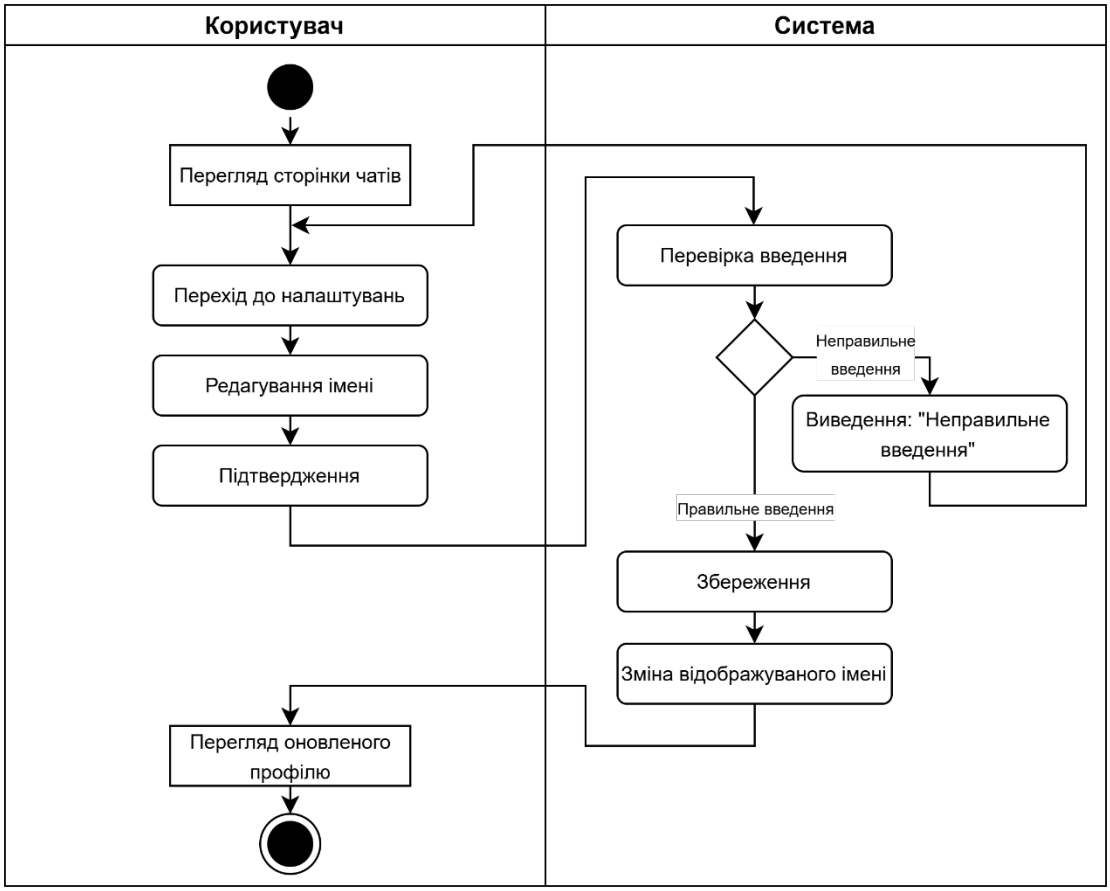

Рис. 8 Діаграма Активності: Зміна налаштувань

1.5.5 Абстракції

Під час проєктування інформаційної системи варто визначитися з базовими абстракціями, що представляють ключові сутності, а також з базовими дії, що відображають поведінку цих сутностей. Абстракції дозволяють сформувати високорівневе бачення структури системи, відокремити суттєві

характеристики від не суттєвих і сформулювати типові дії, що можуть виконуватись над цими сутностями.

На Рис. 9 показано основні абстракції, які використовуються у системі обміну повідомленнями. Вони включають інформаційні компоненти (наприклад, Повідомлення, Профіль, Налаштування) і логічні одиниці взаємодії (наприклад, Користувач, Персональний чат, Груповий чат). Кожна з абстракцій має набір властивостей (полів), які описують її стан, а також дії (операції), що визначають її поведінку.

Рис. 9 Абстракції

Користувач є головною абстракцією системи. Він має доступ до персональних і групових чатів, пов'язаний з профілем та налаштуваннями. Основні дії це: створення персонального чи групового чату, видалення чату, зміна налаштувань, перегляд профілю.

Профіль містить базову інформацію про користувача: логін, ім’я, опис («про себе») і дату реєстрації. Дії над профілем прямо не передбачаються, тому, що зміни відбуваються через відповідні налаштування.

Персональний чат включає назву, перелік повідомлень та профіль співрозмовника. Доступна дія – перегляд профілю іншого учасника.

Груповий чат зберігає назву, список повідомлень та учасників. Дії включають зміну назви та додавання нових учасників.

Учасник є представленням користувача в межах групового чату й пов’язаний з його профілем. Доступна дія – перегляд профілю.

Повідомлення – абстракція, що включає вміст, час надсилання, відправника, прив’язку до чату та тип повідомлення. Основні дії: надсилання, видалення, перегляд повідомлення.

Налаштування зберігають особисту інформацію користувача (ім’я, опис, пошту, пароль, дату реєстрації) і дозволяють виконати дії: вийти з облікового запису, змінити персональні дані.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Етап проектування програмного забезпечення дуже важливий у розробці інформаційної системи, оскільки він формує архітектуру, структуру даних, логіку взаємодії компонентів і функціональне призначення системи. Хороший дизайн зробить майбутню систему ефективною, масштабованою, зручною для обслуговування та простою у підтримці.

У цьому розділі розроблено ER-діаграму для представлення логічної моделі даних. Вона зображує сутності та їхні атрибути, а також зв’язки, які є основою для роботи системи обміну повідомленнями. Структура програмних об’єктів представлена діаграмою класів. Інші чотири діаграми – кооперацій, пакетів, компонентів і розгортання – дають детальну інформацію про те, як різні частини системи взаємодіють на різних рівнях абстракції.

Ці діаграми показують архітектурні рішення, зроблені під час створення веб-програми. Вони служать основою для впровадження програмного забезпечення, описаного в наступному розділі.

2.1 Логічна модель даних у вигляді ER-діаграми

Логічна модель даних є одним з найважливіших аспектів під час розроблення та проектування інформаційної системи, коли відбувається її аналіз, оформлення й моделювання предметної області. Вона дає змогу зібрати та визначити інформаційні об’єкти, так звані сутності, атрибутивний склад цих сутностей та різноманітні функціональні і нефункціональні види залежностей особливо між сутностями та їх атрибутами. Оскільки логічна модель є багаторівневою абстракцією концептуальної моделі, вона є незалежною від певного середовища впровадження (проєкції) інформаційної системи, тому

можливо зосередитись на взаємодії даних без необхідності враховувати технічні обмеження, з якими можна зіткнутися.

В ході розробки, було побудовано логічну модель даних, яка приведена на рис. 10. На даній діаграмі показані сутності і їх зв'язки, вона дозволяє уявити логічну структуру та адекватність даної системи, і служить основою для побудови фізичної моделі бази даних на наступних кроках.

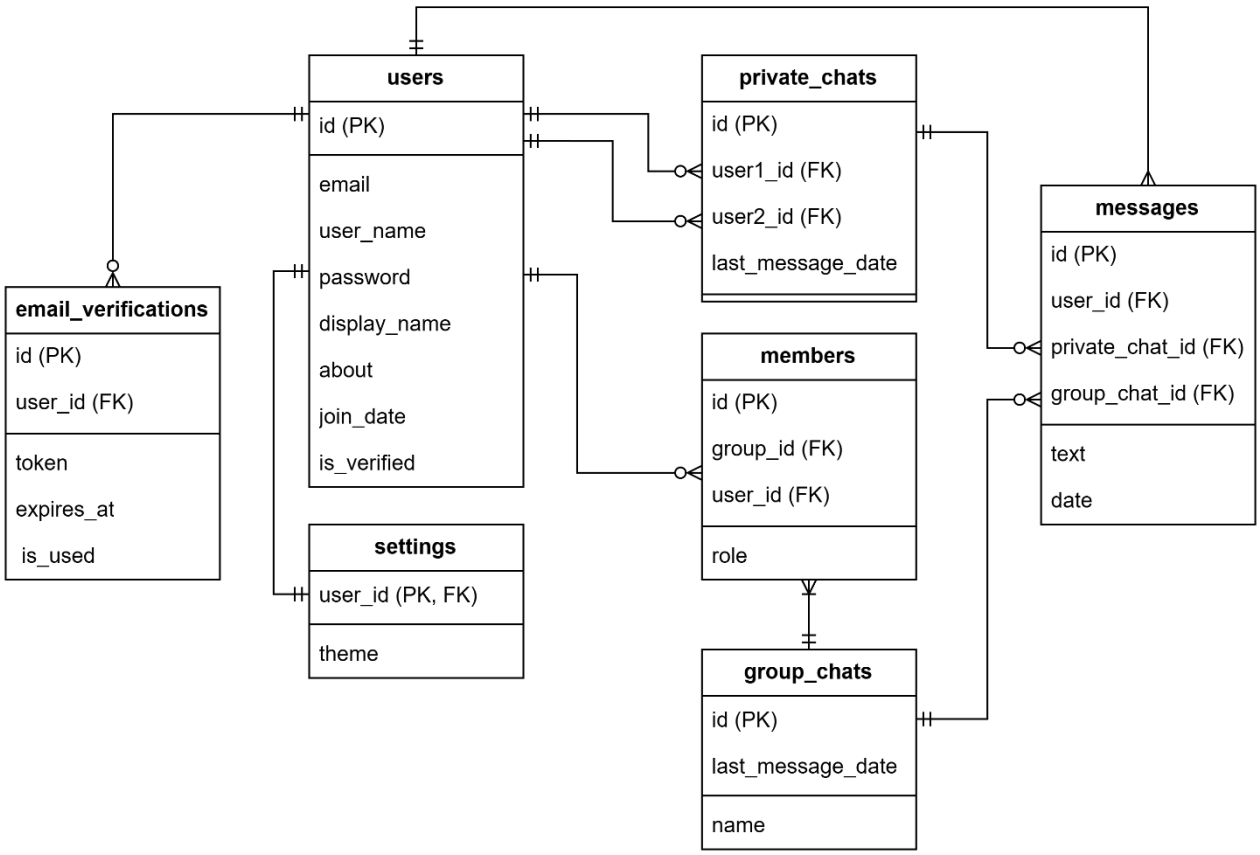

Рис. 10 Логічна модель даних

Модель враховує ключові аспекти системи:

Користувачі (Users) – основна таблиця, яка зберігає облікові записи користувачів. Поля цієї таблиці:

- унікальні ідентифікатори (id),
- електронна пошта
- ім'я користувача (user_name),
- пароль (у вигляді хешу),
- відображуване ім’я (display_name),

- дані «про себе» (about),
- дата реєстрації (join_date),
- статус перевірки (is_verified).

Налаштування – об'єкт, пов'язаний з кожним користувачем у відношенні 1:1 та містить атрибут налаштування теми. Відношення користувача визначається через зовнішній ключ user_id.

Приватні чати (private_chats) – об'єкт для індивідуальних розмов двох користувачів. Містить посилання на обох учасників чату (user1_id, user2_id) та дату останнього повідомлення. Щоб уникнути дублікатів, забезпечується унікальність для пар користувачів.

Групові чати (group_chats) відстежують назву чату та дату надсилання останнього повідомлення. Один чат може включати кількох учасників, а зв'язок з цими особами встановлюється через окрему сутність.

Учасники (members) – це допоміжна таблиця, що сприяє зв'язку «багато до багатьох» між груповими чатами та користувачами. Вона має додатковий атрибут ролі (admin, moderator, user), який визначає рівень доступу, що надається кожному учаснику.

Повідомлення (messages) – це сутність, яка містить надіслані повідомлення. Повідомлення належить або приватному чату, або груповому, але не обом одночасно. Це забезпечується логічним обмеженням (умова CHECK). Ключові атрибути:

- текст повідомлення
- дата та час відправлення
- ідентифікатор відправника (user_id),
- прив'язка до приватного або групового чату.

Перевірка електронної пошти (Email_verifications) – це об'єкт для зберігання одноразових токенів підтвердження електронної пошти користувачів. Зберігає дату закінчення терміну дії токена, чи був він використаний, та посилання на користувача.

Взаємодія між об'єктами здійснюється за допомогою зовнішніх ключів, які забезпечують цілісність даних. Коли записи користувачів видаляються, це каскадно видаляє пов'язані об'єкти (налаштування, повідомлення, перевірки тощо).

2.2 Діаграма класів та кооперацій

Діаграми класів та кооперації використовуються при проєктуванні програмного забезпечення для зображення його об'єктно-орієнтованої моделі та динамічної взаємодії об'єктів. Вони допомагають визначити основні компоненти системи, їхні властивості, методи та взаємодії між собою. Ці діаграми є основою для подальшої реалізації програмної логіки та надають формальне уявлення про внутрішню архітектуру системи.

У цьому розділі розглядаються два важливі аспекти: статична структура системи, яку показує діаграма класів, та динамічні сценарії взаємодії об'єктів, які показані на діаграмі кооперації. Нижче подано діаграму класів, яка описує основні об'єкти програмної частини розроблюваної системи обміну повідомленнями.

2.2.1 Діаграма класів

Діаграма класів є одним з основних інструментів об'єктно-орієнтованого проектування програмного забезпечення. Вона відображає статичну структуру системи, включаючи основні класи, їхні атрибути, методи, а також зв'язки між об'єктами. Побудова діаграми класів дозволяє формалізувати архітектурупрограмної системи та забезпечує підґрунтя для реалізації логіки взаємодії між її складовими частинами.

У контексті розроблюваного веб-застосунку для обміну повідомленнями, діаграма класів демонструє структуру основних програмних сутностей, які забезпечують автентифікацію, управління сесіями, створення та обробку чатів, обмін повідомленнями, керування профілем користувача, а також налаштування облікового запису.

Головною точкою входу в систему є клас Program, який запускає логіку авторизації за допомогою сервісу AuthService та встановлює поточного користувача у випадку успішної авторизації. Клас AuthService реалізує основні методи автентифікації, такі як вхід у систему, реєстрація нового користувача, а також відновлення сесії. Для роботи з сесіями використовується окремий клас SessionManager, який відповідає за створення, перевірку й знищення сесій користувачів. Зберігання сесій реалізоване через словник пар токенів і відповідних ідентифікаторів користувачів.

Клас User виконує роль головного об'єкта, що показує зареєстрованого користувача системи. Він зберігає посилання на власні налаштування (Settings), особистий профіль (Profile), а також колекції персональних (PrivateChat) і групових чатів (GroupChat), до яких належить користувач. За допомогою відповідних методів користувач може створювати нові чати, видаляти приватні, залишати групові, а також відкривати профілі співрозмовників.

Клас Settings відповідає за управління обліковими параметрами, зокрема ім’ям відображення, описом, електронною поштою та паролем. Додатково передбачено методи для верифікації електронної пошти та виходу з облікового запису. Особисті дані користувача винесено в окремий клас Profile, який містить публічну інформацію: ім’я користувача, ім’я відображення, опис і дату реєстрації.

Чатова функціональність системи показана двома окремими сутностями: PrivateChat для приватних розмов і GroupChat для спілкування в групах. Обидва класи мають спільну функціональність щодо створення та видалення повідомлень, проте GroupChat додатково реалізує методи керування учасниками, правами доступу та зміни назви чату. Приватний чат містить лише одного співрозмовника, інформація про якого зберігається у вигляді профілю. Клас Member відображає учасника групового чату і зберігає роль у чаті (наприклад, «admin» або «user»), а також посилання на його профіль.

Об’єкти класу Message інкапсулюють інформацію про окремі повідомлення, які надсилаються у чатах. Кожне повідомлення має текстове поле,

дату надсилання та зберігається у відповідному чаті. Зв’язки між класами демонструють не лише приналежність об’єктів один до одного, а й характер використання методів, наприклад створення, перегляд, або керування.

Взаємозв’язки між описаними класами, а також їхня внутрішня структура наведені на діаграмі класів, зображеній на рис. 11. Ця діаграма є основою для подальшого переходу до розробки програмних модулів та формалізує логічну структуру програмного забезпечення.

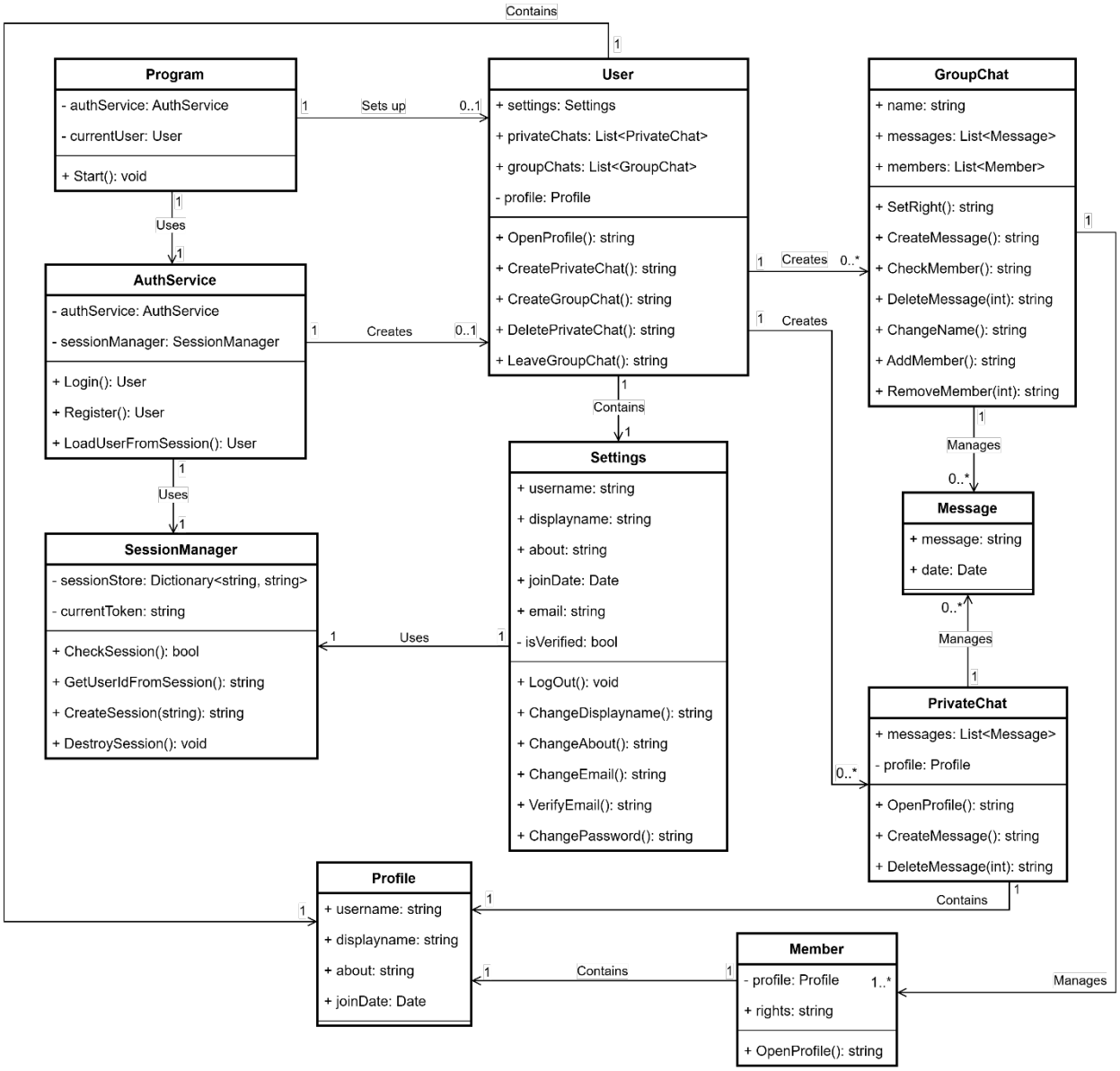

Рис. 11 Діаграма класів

2.2.2 Діаграми кооперацій

Діаграми кооперації (діаграми співпраці, collaboration diagrams) описують, які об'єкти яким чином взаємодіють для реалізації сценаріїв виконання функцій. На відміну від діаграм класів, які описують статичний аспект об'єктно-орієнтованої системи, діаграми співпраці описують динамічний аспект, тобто взаємодії об'єктів у системі. Діаграма співпраці зображає об'єкти і зв'язки (асоціації, композиції, агрегації і залежності та ін.). Таким чином, діаграма співпраці показує, яким чином виконуються окремі функції користувача програмного продукту.

Було створено три діаграми кооперацій, які описують процес авторизації користувача, створення та використання приватного чату, зміну налаштувань користувача з можливістю виходу із системи. Кожна з діаграм відображає об'єкти, які задіяні в сценарії, а також зв’язки між ними (асоціації, композиції, агрегації та залежності), на основі принципів об'єктно-орієнтованого проєктування.

Перша діаграма кооперації (рис. 12) показує послідовність дій під час авторизації користувача. Початковий об'єкт Program ініціює логіку аутентифікації за допомогою сервісу AuthService, котрий, у свою чергу, взаємодіє з SessionManager для перевірки наявної сесії або створення нової. В результаті успішної авторизації створюється об'єкт користувача (User) з відповідним профілем (Profile), що зберігає особисту інформацію користувача. Зв’язки між об’єктами реалізовано через композицію (між Program та User), агрегацію (Profile → User, SessionManager → AuthService) та залежність (AuthService → User).

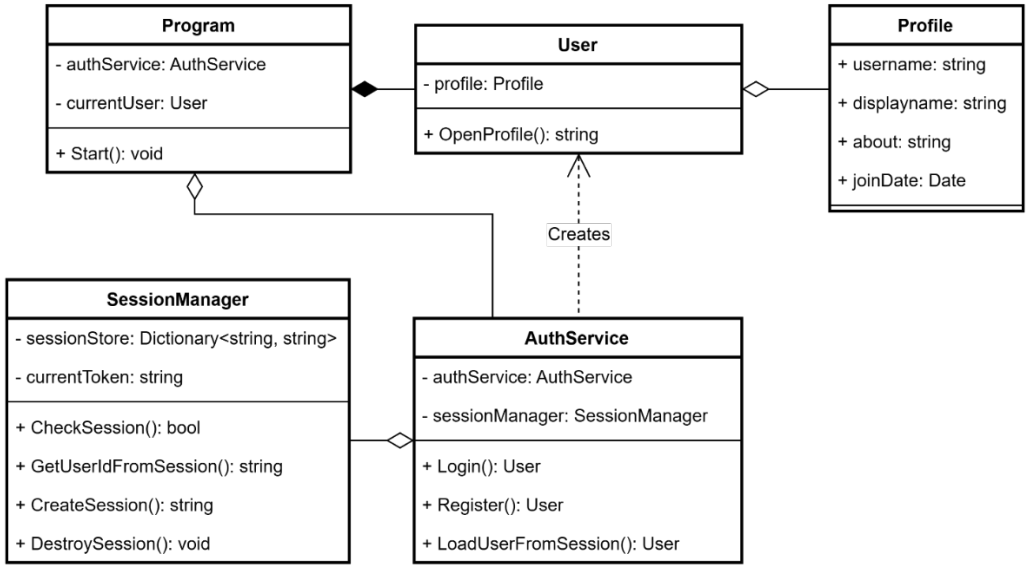

Рис. 12 Діаграма кооперації: Авторизація користувача

Друга діаграма (рис. 13) показує взаємодію об'єктів у процесі створення приватного чату, надсилання повідомлення та його подальшого видалення. Користувач за допомогою методу CreatePrivateChat() ініціює створення об'єкта PrivateChat, який зберігається у списку приватних чатів. Кожен чат містить перелік повідомлень (Message), які створюються методом CreateMessage(), а також можуть бути видалені методом DeleteMessage(). Усі зв'язки між об'єктами виконані через композицію, що відображає сильний взаємозв'язок та життєвий цикл об'єктів у межах конкретного користувача.

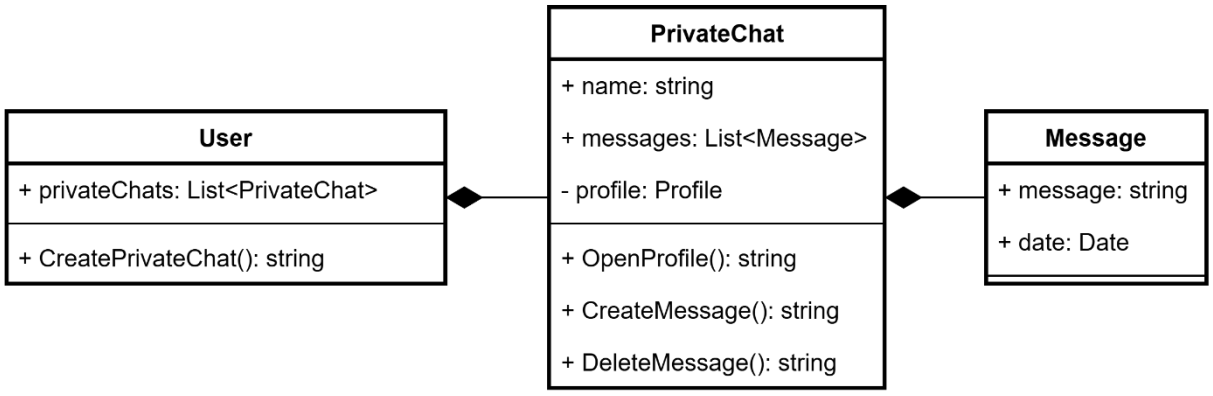

Рис. 13 Діаграма кооперації: Створення приватного чату, надсилання та видалення повідомлення

Третя діаграма (рис. 14) показує сценарій зміни налаштувань користувача та завершення сеансу роботи. Клас Settings, що є складовою об'єкта користувача, забезпечує набір методів для зміни параметрів облікового запису, таких як ім’я, опис, електронна пошта, а також керування верифікацією. Метод Logout() запускає знищення сесії через об’єкт SessionManager, що дозволяє завершити поточний сеанс. Взаємозв’язки в даній кооперації представлено через композицію між Settings та User, а також асоціацію між Settings та SessionManager, котра показує одноразове використання функціоналу з іншого об’єкта.

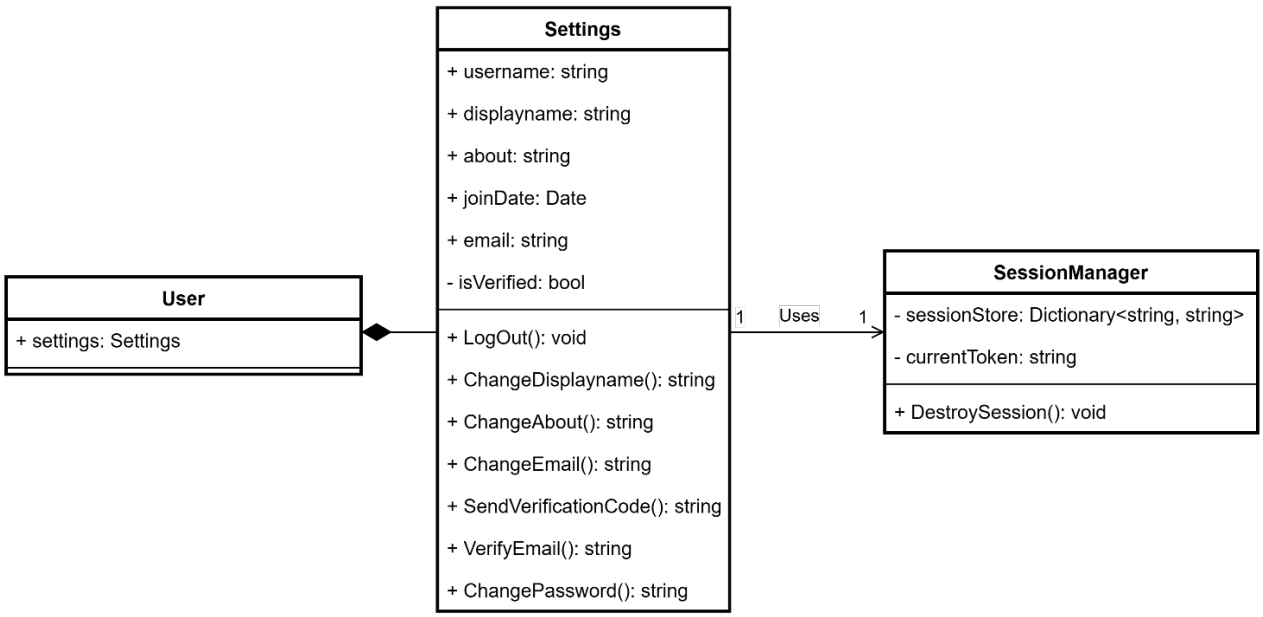

Рис. 14 Діаграма кооперації: Зміна налаштувань та вихід із системи

2.3 Діаграма пакетів

Діаграма пакетів – це діаграма у структурному моделюванні, яка розподіляє компоненти системи на модулі, звані пакетами та відстежує об’єкти залежності між ними на вищому рівні абстракції. Це допомагає показати

відносно велику систему на більш простому шаблоні, що дає краще розуміння розподілу обов’язків між компонентами.

У системі було виділено чотири логічні шари, кожен з яких реалізує окремий рівень відповідальності:

Application layer (шар прикладної логіки) містить пакети Program та Services. Пакет Program включає головний клас Program.cs, який відповідає за ініціалізацію програми. Пакет Services містить сервіси прикладної логіки, такі як AuthService та SessionManager.

Infrastructure layer (інфраструктурний шар) об’єднує пакети Data та API. Пакет Data містить клас DBContext, а також вкладений пакет DBModels, який реалізує структури даних, необхідні для роботи з базою даних.

Domain layer (шар предметної області) представлений пакетом Models, що включає основні сутності системи, зокрема класи User, Settings, PrivateChat, Profile, Message, GroupChat і Member.

Presentation layer (шар представлення) включає пакет UI(pages), який реалізує користувацький інтерфейс.

На основі побудованої діаграми пакетів (рис. 10) можна простежити залежності між компонентами. Клас Program.cs використовує пакети UI(pages) та Services, що вказує на його роль як точки входу в систему та координатора основних дій. Сервіси (Services) взаємодіють з інфраструктурними компонентами: викликають API (<<use>>), звертаються до бази даних через пакет Data (<<call>>) та імпортують моделі доменної області (<<import>>). Пакети API та Data також імпортують спільні доменні моделі, що надає цілісність представлення даних у різних шарах. Користувацький інтерфейс (UI(pages)) використовує сервіси для виконання дій, що ініціюються користувачем, дотримуючись принципів розділення обов’язків і слабкого зв’язку між шарами.

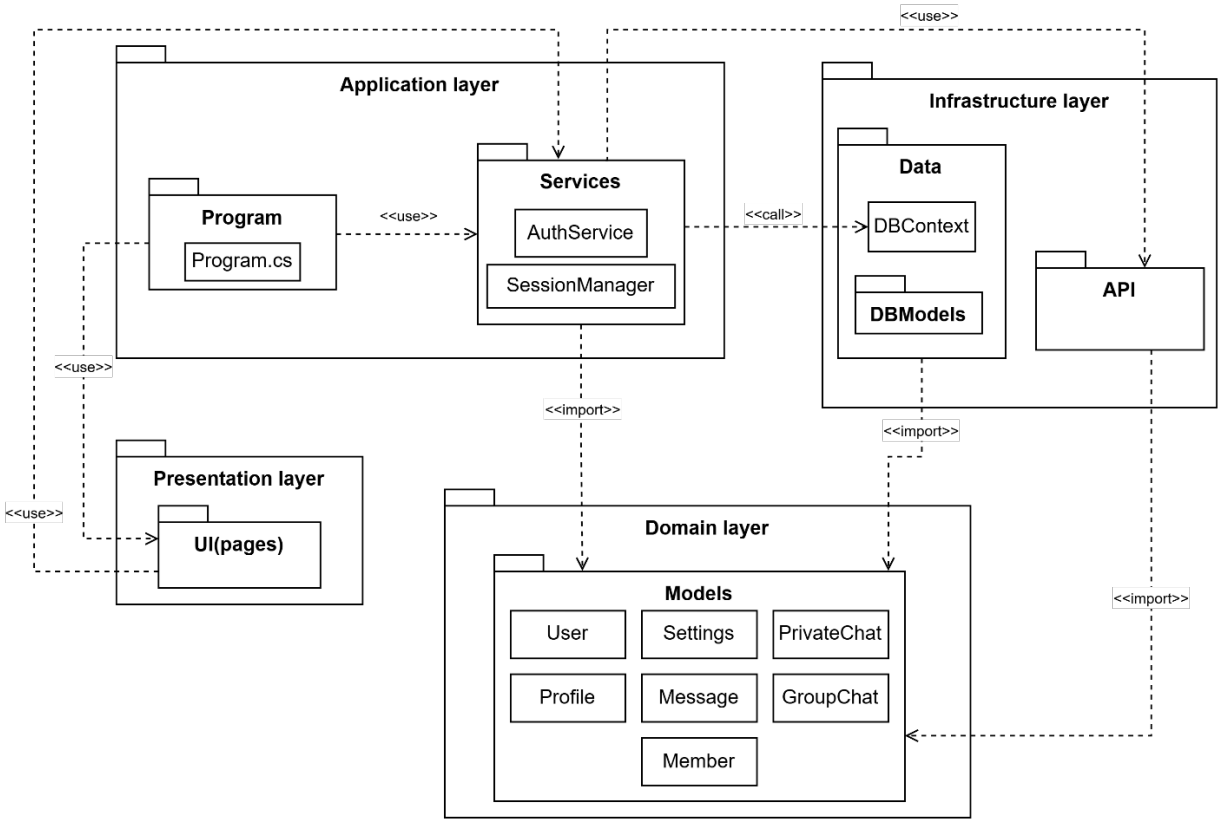

Рис. 15 Діаграма пакетів

2.4 Діаграма компонентів

Діаграма компонентів є суттєвим інструментом моделювання структури, в якому зображується фізична архітектура програмного забезпечення. Вона надає уявлення щодо ключових програмних компонентів системи та їх інтерфейсів, а також їх взаємозв'язків та співвідношень. Такий підхід забезпечує чітке уявлення модульної структури застосунку, спрощує процес розробки, тестування та супроводу програмного забезпечення, а також ефективно підтримує принципи інкапсуляції та підвищення повторного використання коду.

В розробленому застосунку виділено шість основних компонентів, кожен з яких виконує певну функцію:

Blazor UI - компонент для інтерфейсу користувача, розроблений на Blazor. Цей компонент взаємодіє з іншими частинами системи через необхідні інтерфейси IUserService, IAuthService та IChatService, які забезпечуються відповідними сервісами.

UserService – сервіс, що відповідає за керування користувачем та пов’язаними налаштуваннями. Він реалізує інтерфейс IUserService та потребує доступ до інтерфейсів IDatabaseAccess та ISessionManager для своєї роботи.

AuthService – компонент, що відповідає за авторизацію та реєстрацію користувачів. Він реалізує інтерфейс IAuthService та залежить від інтерфейсів IDatabaseAccess і ISessionManager.

ChatService – компонент, що відповідає за логіку обробки чатів та повідомлень. Цей сервіс реалізує інтерфейс IChatService та потребує доступу до бази даних через інтерфейс IDatabaseAccess.

SessionManager – компонент, що реалізує управління сесіями користувачів. Він реалізує інтерфейс ISessionManager, що використовується сервісами авторизації та профілю.

DBService – компонент, що забезпечує доступ до бази даних та реалізує інтерфейс IDatabaseAccess. Його використовують усі сервіси, яким потрібна взаємодія з шаром даних.

На діаграмі компонентів (рис. 16) показано залежності між цими компонентами. Усі зв’язки втілено як залежності від необхідних інтерфейсів до реалізованих, що сприяє слабкому зв'язку між компонентами та дотриманню принципу інверсії залежностей.

Загальна структура системи, відображена на діаграмі компонентів, демонструє високий модульність та відповідає принципам об’єктно-орієнтованого дизайну. Цей підхід сприяє масштабованості системи, дозволяє легко додавати нові функції або замінювати окремі компоненти без значного впливу на інші частини застосунку.

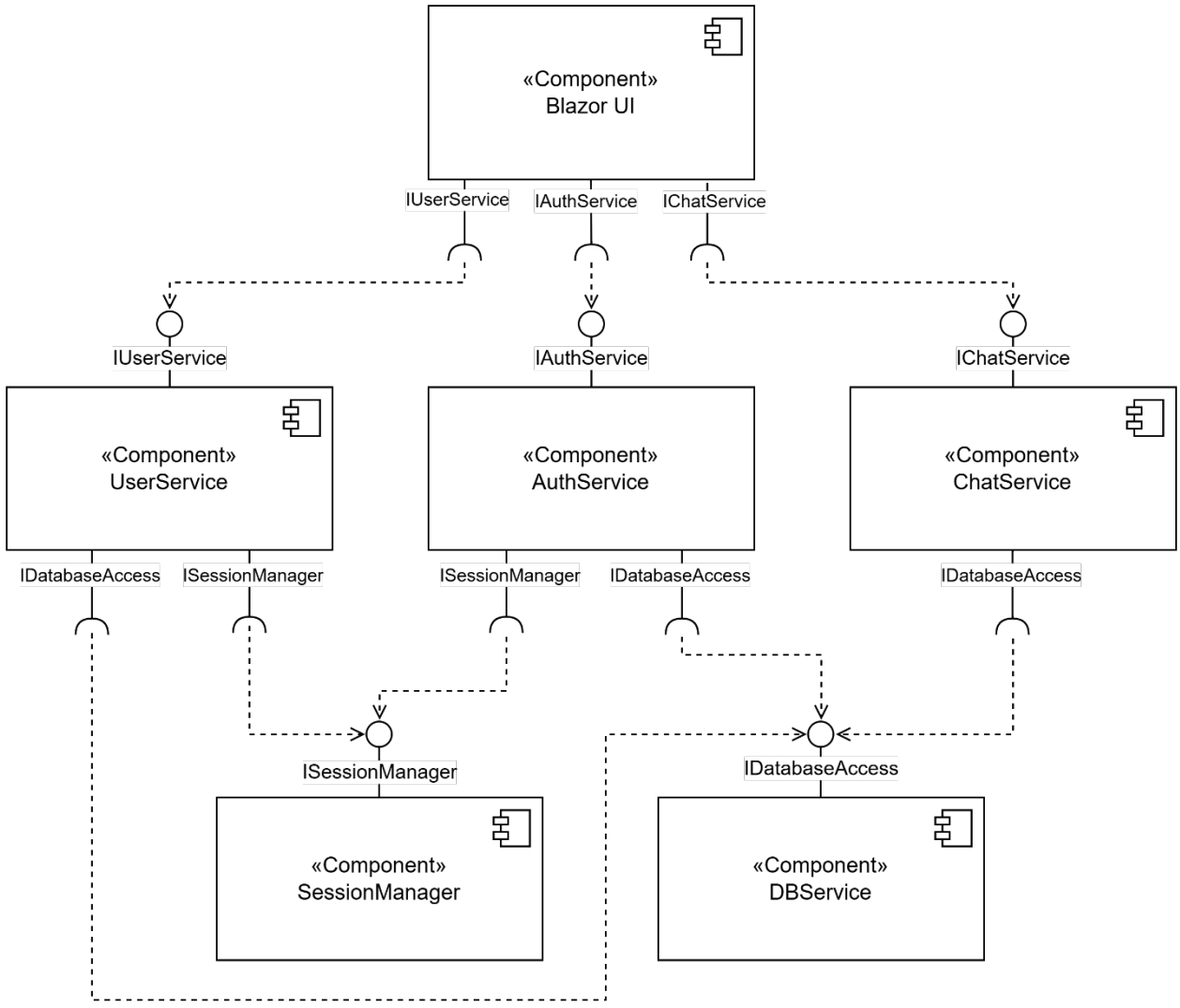

Рис. 16 Діаграма компонентів

2.5 Діаграма розгортання

Діаграма розгортання є невід'ємною частиною проєктування програмного забезпечення, оскільки вона відображає фізичне розгортання компонентів системи у середовищі виконання. Такий тип діаграми дозволяє відобразити як різні частини застосунку взаємодіють між собою на рівні апаратних та програмних вузлів, які ресурси використовуються, та яким чином реалізовано обмін даними між клієнтом, сервером і базою даних.

У системі продумано три основні вузли: клієнтський пристрій, веб-сервер, на якому розгорнуто застосунок Blazor Server, і сервер бази даних, що працює

під на PostgreSQL. Взаємодія між ними відбувається відповідно до стандартних архітектурних принципів клієнт-серверної моделі.

На стороні клієнта функціонують артефакти «Web Browser» і «SignalR Client», які забезпечують візуалізацію інтерфейсу та реєстрацію подій у режимі реального часу за допомогою WebSocket-з'єднання. Через HTTP(S)-протокол клієнт надсилає запити до представницького шару сервера.

На веб-сервері розгорнуто кілька логічних компонентів, поділених на шари:

- UI Layer, який містить інтерфейсні компоненти (MainLayout.razor, Chats.razor, Login.razor, Register.razor) і відповідає за взаємодію з користувачем.
- Business Logic Layer, до складу якого входять сервіси (AuthService.cs, ChatService.cs) та SessionManager.cs, що обробляють запити, керують сесіями та забезпечують логіку роботи месенджера.
- Infrastructure Layer, де реалізовано ініціалізацію додатку (Program.cs), доступ до бази даних (DBContext.cs) і обробку реального часу за допомогою SignalR Hub.

Окремим вузлом виступає сервер бази даних, реалізований на основі PostgreSQL. Він містить відповідні таблиці (users, messages, group_chats, private_chats, members, settings, user_verifications), до яких звертається застосунок через об’єкт DBContext.cs за допомогою SQL-запитів.

На рисунку 17 представлено діаграму розгортання, що відображає описану інфраструктуру програмного забезпечення, включаючи взаємозв’язки між усіма компонентами системи та механізми комунікації між ними.

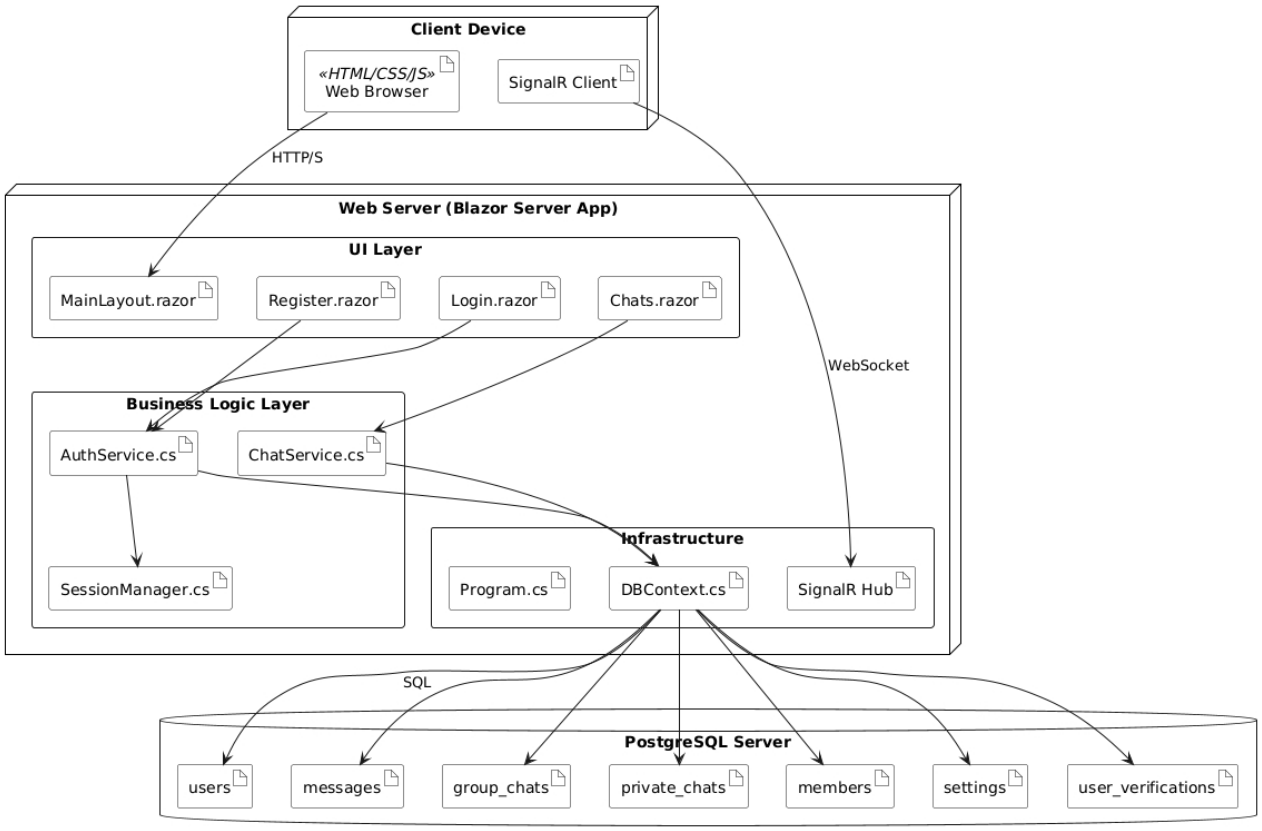

Рис. 17 Діаграма розгортання

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Система управління інформаційною базою

Для забезпечення збереження, цілісності та ефективної обробки інформації в межах розроблюваної системи було обрано систему управління базами даних PostgreSQL. Це сучасна, потужна об'єктно-реляційна СУБД з відкритим вихідним кодом, що підтримує повний набір SQL-стандартів та забезпечує широкі можливості для масштабування, розширення і оптимізації продуктивності. PostgreSQL активно використовується в промислових та наукових проєктах, які потребують високої надійності та продуктивності, зокрема в системах, що функціонують у режимі реального часу.

Однією з ключових переваг PostgreSQL є підтримка транзакцій з повною відповідністю принципам ACID, що гарантує надійність операцій зі збереження даних навіть у випадку збоїв. Завдяки цьому в розробленій системі можна безпечно виконувати послідовні дії, наприклад, додавання повідомлення та одночасне оновлення часу останньої активності чату. Крім того, PostgreSQL забезпечує підтримку індексів, тригерів, зовнішніх ключів та складних запитів, що дозволяє гнучко організувати структуру інформаційної бази та забезпечити ефективну обробку даних незалежно від їхнього обсягу.

З точки зору масштабованості, PostgreSQL дозволяє ефективно працювати як у невеликих локальних проєктах, так і в розподілених хмарних середовищах. Це особливо важливо для системи обміну повідомленнями, яка потенційно може обслуговувати велику кількість користувачів одночасно.

Для зручності взаємодії з базою даних у середовищі розробки було обрано технологію об'єктно-реляційного відображення (ORM) — Entity Framework Core. Цей інструментарій, розроблений корпорацією Microsoft, дозволяє розробникам взаємодіяти з базою даних за допомогою мовних конструкцій C#, не вдаючись

безпосередньо до SQL-запитів. Завдяки цьому зменшується кількість шаблонного коду, підвищується продуктивність розробки та забезпечується краща читабельність і підтримуваність програмного коду.

EF Core також забезпечує підтримку міграцій, що дозволяє керувати версіями схеми бази даних у процесі розробки. Це дає змогу автоматично створювати або оновлювати таблиці, залежно від змін у класах моделі, що спрощує командну розробку та розгортання проєкту.

Загалом, поєднання PostgreSQL як надійної СУБД та EF Core як гнучкого ORM-рішення забезпечує стабільну, масштабовану та ефективну інформаційну основу для реалізації функціональності системи обміну повідомленнями в реальному часі.

3.2 Розробка інформаційної бази

Інформаційна база розробленої системи є фундаментальною складовою, яка забезпечує структуроване збереження, обробку та взаємозв’язок даних між ключовими компонентами мессенджера. Архітектура бази даних спроєктована відповідно до вимог, сформульованих на етапах аналізу предметної області та логічного моделювання, з урахуванням сучасних принципів проєктування реляційних БД.

До основних сутностей бази даних належать:

- Користувачі, які мають унікальний ідентифікатор, адресу електронної пошти, логін, пароль (у вигляді хешу), відображуване ім’я, опис, дату приєднання до системи та статус верифікації.
- Приватні чати, які є двосторонніми зв’язками між двома унікальними користувачами, з перевіркою на недопущення створення ідентичних чатів між тими ж самими учасниками.
- Групові чати, які включають назву та дату останнього повідомлення, з підтримкою багатьох учасників.

- Учасники груп, для яких зберігається роль (адміністратор, модератор, користувач), що дозволяє реалізувати контроль доступу до окремих функцій у групових чатах в майбутньому.
- Повідомлення, що можуть належати як приватному, так і груповому чату, але не обом одночасно, що забезпечується відповідним обмеженням на рівні таблиці.
- Налаштування користувача, які дають змогу зберігати переваги, зокрема вибрану тему інтерфейсу.
- Верифікації електронної пошти, необхідні для підтвердження адреси користувача, з урахуванням терміну дії токена та його повторного використання.

Між сутностями реалізовано такі типи зв’язків: «один до одного» (наприклад, між користувачем і його налаштуваннями), «один до багатьох» (наприклад, між користувачем і повідомленнями), а також «багато до багатьох» (через таблицю members, що зв’язує користувачів із груповими чатами). Поля, які можуть містити відсутні значення (nullable), застосовано лише там, де це логічно виправдано — зокрема, для вказівки типу чату у повідомленнях (груповий або приватний).

Фізична модель бази даних реалізована у вигляді набору таблиць PostgreSQL, визначених SQL-інструкціями. Для кожної таблиці задані типи даних, первинні та зовнішні ключі, унікальні обмеження, а також перевірки цілісності (CHECK-умови). На рисунку 3.1 наведено фізичну модель бази даних, що демонструє структуру таблиць, їхні атрибути, типи даних і зв’язки між сутностями.

У реалізації моделі передбачено забезпечення каскадного видалення залежних записів, що дозволяє підтримувати цілісність даних без ручного втручання. Наприклад, при видаленні користувача автоматично видаляються всі пов’язані з ним повідомлення, налаштування та участь у чатах.

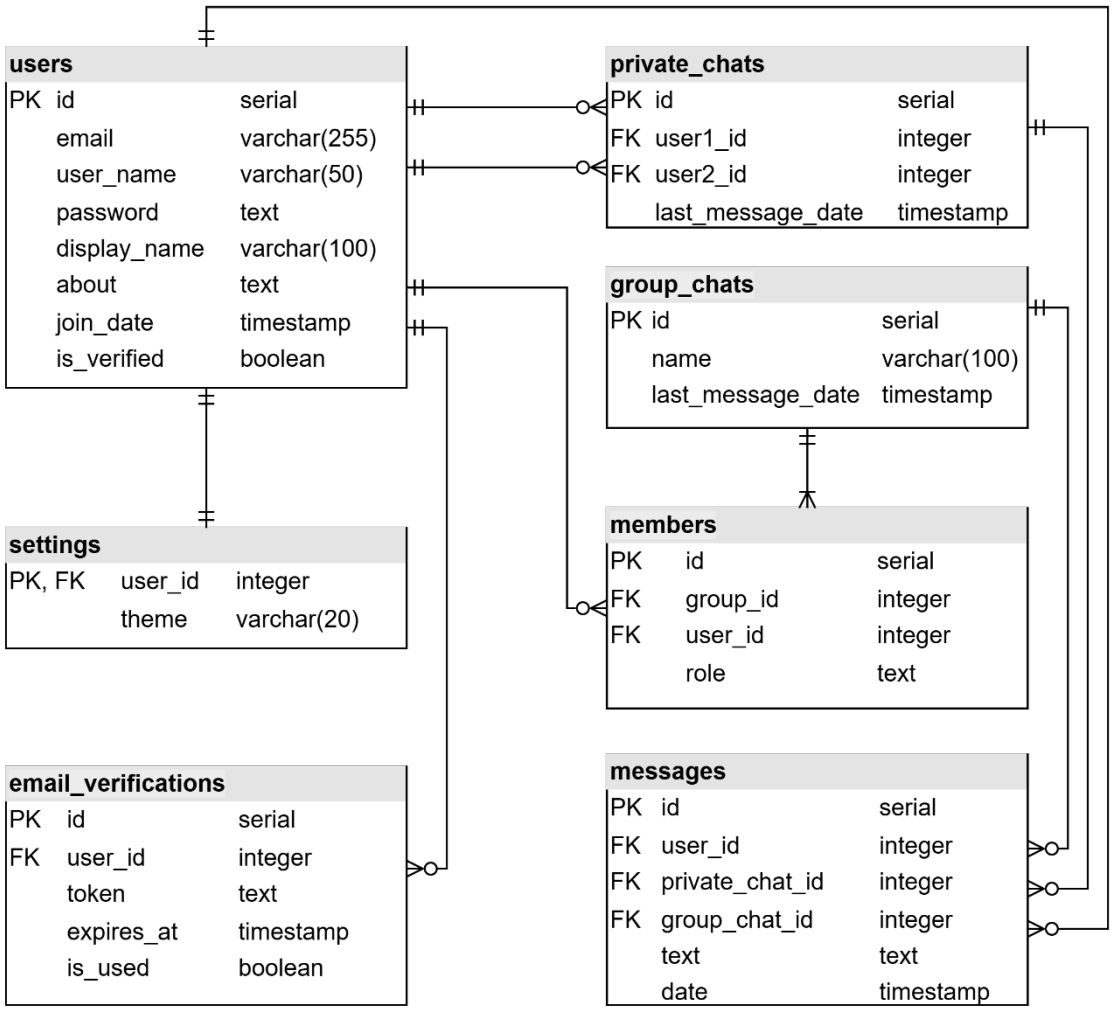

Рис. 18 Фізична модель бази даних

SQL-структура, що була застосована для створення бази даних, містить докладне визначення кожної таблиці, включаючи всі необхідні обмеження. Код охоплює створення таблиць users, settings, private_chats, group_chats, members, messages, а також email_verifications. Усі необхідні SQL-запити створення таблиць описано в Додатку Б, на який можна орієнтуватися при необхідності аналізу повної реалізації.

Ініціалізація бази даних у проєкті відбувається за допомогою засобів ORM-технології Entity Framework Core, а саме через механізм міграцій. Детальний опис цього процесу наведено у підрозділі 3.4.

Отже, побудована інформаційна база забезпечує повноцінну основу для реалізації функціональності системи, відповідаючи вимогам до продуктивності,

узгодженості та масштабованості, що є критично важливими для систем обміну повідомленнями в реальному часі.

3.3 Вибір інструментальних засобів розробки

Для реалізації програмного забезпечення системи обміну повідомленнями в реальному часі було обрано сучасні та ефективні засоби розробки, що забезпечують високу продуктивність, масштабованість та зручність у підтримці програмного коду. Основу проєкту становить платформа .NET 9, яка є актуальною версією фреймворку від компанії Microsoft з відкритим вихідним кодом та широким спектром інструментів для розробки вебзастосунків.

У якості основної мови програмування обрано C#, яка надає розвинену об’єктно-орієнтовану модель, високу безпечність типів, підтримку асинхронного програмування та широкий інструментарій для розробки сучасних вебсервісів.

Середовище розробки Microsoft Visual Studio – забезпечує зручний графічний інтерфейс, інтеграцію з системами контролю версій, підтримку налагодження на різних етапах, а також вбудовані засоби роботи з базами даних, API та серверною логікою. Visual Studio оптимально підходить для реалізації проєктів на платформі .NET та взаємодії з базами даних через ORM.

Ключовим компонентом архітектури клієнтської частини застосунку є Blazor Server – технологія від Microsoft, яка дозволяє створювати інтерактивні вебінтерфейси за допомогою C# замість JavaScript. На відміну від Blazor WebAssembly, який виконується на стороні клієнта, Blazor Server працює з рендерингом на сервері через SignalR – бібліотеку для двостороннього обміну повідомленнями у реальному часі на основі WebSocket-з'єднань. Це дозволяє досягти доброго відкліку інтерфейсу навіть при складній логіці обробки даних, зберігаючи при цьому повний контроль над станом компонента. Такий підхід дуже полегшує реалізацію реального часу в месенджері, зокрема для миттєвого відображення нових повідомлень, змін статусу користувачів, оновлення списків чатів тощо.

Для реалізації доступу до реляційної бази даних було обрано об’єктно-реляційний мапінг Entity Framework Core (EF Core), який підтримує генерацію моделей, міграції, LINQ-запити та зменшує необхідність прямої роботи з SQL-кодом. Це дозволяє значно підвищити швидкість розробки та уніфікувати логіку взаємодії з базою даних на рівні C#-класів.

У зв’язці з EF Core використовується драйвер Npgsql, що є офіційним ADO.NET-провайдером для PostgreSQL – обраної системи управління базами даних. Npgsql забезпечує повну підтримку типів даних PostgreSQL, транзакцій, оптимізованих запитів, а також надає розширені можливості для асинхронного доступу до даних.

Завдяки поєднанню Blazor Server, SignalR та Entity Framework Core проєкт має гнучку модульну структуру, здатну обробляти великі обсяги повідомлень із мінімальною затримкою. Такий вибір інструментарію дозволив створити динамічний вебзастосунок, що підтримує реальну часову взаємодію, відповідає сучасним стандартам безпеки, має масштабовану архітектуру та може бути розгорнутий у різних середовищах – як локально, так і в хмарній інфраструктурі.

3.4 Реалізація прикладної логіки

3.4.1 Початок роботи з базою даних

Перед реалізацією логіки роботи із базою даних необхідно було налаштувати підключення до системи керування базами даних PostgreSQL та згенерувати модель контексту даних, яка відображає структуру таблиць. Для цього у конфігураційному файлі appsettings.json було додано рядок з’єднання у секції ConnectionStrings. Приклад запису:

```
"ConnectionStrings": {
"DefaultConnection": "Host=localhost; Database=messenger;
                      Username=postgres; Password=1"
}
```

На основі цього з’єднання було виконано команду генерації моделей та контексту бази даних за допомогою інструменту Entity Framework Core. Використана команда Scaffold-DbContext у вигляді:

```
dotnet ef dbcontext scaffold "Name=DefaultConnection" Npgsql.EntityFrameworkCore.PostgreSQL --output-dir Models --context-dir Data --context MyDbContext --force
```

У результаті виконання цієї команди автоматично сформовано класи, що відповідають таблицям бази даних, а також створено клас MyDbContext, який є контекстом доступу до бази даних і реалізує шаблон Unit of Work.

Після генерації необхідно зареєструвати DbContextFactory у контейнері залежностей у файлі Program.cs. Це дозволяє забезпечити ефективне створення екземплярів контексту в середовищі Blazor Server. Реєстрація відбувається наступним чином:

```
builder.Services.AddDbContextFactory<MyDbContext>(options =>
{
   options.UseNpgsql(builder.Configuration.GetConnectionString(
         "DefaultConnection"));
});
```

Отже, створено механізм доступу до бази даних, який відповідає сучасним практикам розробки у середовищі ASP.NET Core із застосуванням технологій Entity Framework Core та PostgreSQL. Докладну структуру згенерованих моделей таблиць та контексту MyDbContext, що використовується для взаємодії з базою даних, наведено в Додатку В.

3.4.2 Облікові записи та сесії

3.4.2.1 Реєстрація користувачів

У межах даного проєкту процес реєстрації реалізовано за допомогою інструментів платформи Blazor Server, з використанням валідації на основі атрибутів, інтерактивного оброблення помилок та збереження стану користувача в захищеному локальному сховищі.

Інтерфейс сторінки реєстрації реалізовано у вигляді компонента Blazor з маршрутним маркером /register. У структурі форми застосовується клас EditForm, що дозволяє інтегрувати автоматичну валідацію введених даних через атрибути DataAnnotations. Наприклад, для полів електронної пошти, імені користувача та пароля визначено обмеження на довжину та обов’язковість, а також додано користувацький атрибут StrictEmail для валідації електронної пошти через регулярний вираз.

Повний вміст моделі RegisterModel, що використовується для перевірки введених даних, наведено у Додатку В.

У разі невідповідності даних система одразу виводить відповідні повідомлення про помилки.

Перед створенням нового користувача здійснюється перевірка унікальності електронної пошти та імені користувача в базі даних за допомогою асинхронних запитів до контексту DbContext. У разі виявлення конфлікту виводиться відповідне повідомлення:

- bool emailExists = await DbContext.Users.AnyAsync(u => u.Email == registerModel.Email);
- bool usernameExists = await DbContext.Users.AnyAsync(u => u.UserName == registerModel.Username);

Після проходження всіх перевірок формується об’єкт користувача User, що містить базову інформацію про новий обліковий запис, включаючи дату створення, стан верифікації, а також типову тему оформлення в налаштуваннях:

```
var newUser = new User {
    Email = ...,
    UserName = ...,
    Password = ...,
    JoinDate = DateTime.UtcNow,
    IsVerified = false,
    Setting = new Setting { Theme = "white" }
};
```

Дані зберігаються до бази за допомогою методу SaveChangesAsync, обгорнутого в блок try-catch, що дозволяє обробити можливі виняткові ситуації, пов’язані з доступом до БД.

Після успішної реєстрації виконується переадресація на головну сторінку застосунку за допомогою сервісу NavigationManager. Окрім цього, на початку ініціалізації компонента перевіряється, чи вже авторизований користувач, і у разі виявлення активної сесії — автоматично здійснюється перенаправлення, що запобігає повторному доступу до сторінки реєстрації:

```
var authState = await AuthProvider.GetAuthenticationStateAsync();
if (user.Identity?.IsAuthenticated == true) {
   Navigation.NavigateTo("/", forceLoad: true);
}
```

Також при першому рендерингу компонента, якщо користувач ще не авторизований, із захищеного локального сховища (ProtectedLocalStorage) видаляється попередній стан чату, що дозволяє уникнути неконсистентного завантаження даних при першому вході в систему.

Загалом, обробка реєстрації в даній реалізації побудована на поєднанні серверної логіки перевірки з інтерактивним інтерфейсом, що забезпечує користувачу інтуїтивно зрозумілий і безпечний досвід створення облікового запису.

3.4.2.2 Авторизація користувачів

Вхід в систему реалізовано через окрему сторінку /login, яка використовує серверний рендеринг (Blazor Server) з інтерактивною логікою, що дозволяє здійснювати перевірку облікових даних у режимі реального часу без перезавантаження сторінки.

Сторінка авторизації побудована на основі компонента EditForm, який використовує модель LoginModel. Ця модель визначає обов’язкові поля електронної пошти та пароля, що підтверджується атрибутами валідації ([Required]). Повний вміст цієї моделі наведено у Додатку В.

Після заповнення форми та підтвердження даних метод HandleValidSubmit здійснює звернення до бази даних через контекст MyDbContext. Перевірка включає пошук користувача за введеною електронною поштою та перевірку відповідності пароля. У разі невірних облікових даних система повертає відповідне повідомлення про помилку, яке відображається на сторінці.

У випадку успішної авторизації створюється об’єкт ClaimsPrincipal, що містить набір claims (утверджень), зокрема унікальний ідентифікатор користувача (ClaimTypes.NameIdentifier). Цей об’єкт асоціюється з cookie-ідентифікацією через CookieAuthenticationDefaults.AuthenticationScheme. Подальший вхід у систему здійснюється за допомогою HttpContext.SignInAsync(principal), після чого користувача автоматично перенаправляють на головну сторінку.

Крім цього, у життєвому циклі компонента враховується поточний стан автентифікації: якщо користувач уже авторизований, система здійснює негайну переадресацію без потреби повторного входу. Такий підхід покращує зручність взаємодії та скорочує кількість непотрібних дій зі сторони користувача.

Як і у випадку з реєстрацією, після першого рендерингу компонента виконується очищення локального сховища (ProtectedLocalStorage) — зокрема, видаляється збережене значення "lastChat", якщо користувач ще не авторизований. Це дозволяє уникнути конфліктів або небажаних даних при зміні стану сесії.

Повний вихідний код сторінки авторизації наведено у Додатку Е.

3.4.2.3 Робота з сесіями

У системі реалізовано повноцінну підтримку сесій користувача на основі механізмів автентифікації та авторизації, що надаються платформою ASP.NET Core. Цей функціонал забезпечує контроль доступу до ресурсів додатку залежно від статусу авторизації користувача та гарантує безпечну взаємодію з внутрішніми компонентами системи.

Після успішної авторизації облікові дані користувача зберігаються у вигляді cookie-файлу, який має назву auth_token. Реєстрація цього механізму здійснюється у файлі Program.cs за допомогою методів AddAuthentication та AddCookie, де також задаються параметри: шлях до сторінки входу (LoginPath = "/login"), тривалість сесії (MaxAge), а також використання ковзаючого терміну дії (SlidingExpiration = true). Це дозволяє підтримувати активну сесію при умові періодичної активності користувача, не змушуючи його повторно входити в систему.

Додатково підключаються служби авторизації (AddAuthorization) та поширення стану автентифікації між компонентами (AddCascadingAuthenticationState), що необхідно для коректної роботи внутрішнього механізму AuthenticationStateProvider в середовищі Blazor Server.

З боку маршрутизації компонентів, у файлі App.razor (у проєкті – Routes.razor) використовується тег <AuthorizeRouteView>, який забезпечує динамічний контроль доступу до сторінок на основі поточного стану автентифікації. Це дозволяє обмежувати доступ до сторінок, які потребують авторизації, а саме головної сторінки чатів.

Усередині сторінок реєстрації (Register.razor) та авторизації (Login.razor) реалізовано логіку перевірки, яка автоматично переадресовує авторизованого користувача на головну сторінку (за маршрутом /), що запобігає повторному доступу до сторінок, призначених лише для неавторизованих відвідувачів. Це здійснюється шляхом отримання поточного стану автентифікації за допомогою AuthenticationStateProvider в методі OnInitializedAsync.

Для головної сторінки чатів (Main.razor) застосовано атрибут [Authorize], який є декларативним способом обмеження доступу на рівні компонента. У випадку, якщо неавторизований користувач намагається отримати доступ до сторінки, система автоматично перенаправляє його до сторінки входу.

Процедура виходу з облікового запису реалізована у вигляді функціоналу в модальному вікні налаштувань користувача, описаному в Додатку Ж.

3.4.3 Приватні чати

Підсистема приватних чатів є невід'ємною частиною функціональності розробленого месенджера, яка реалізує двосторонню комунікацію між двома користувачами. Її інтеграція у загальну архітектуру забезпечує автоматичне підвантаження існуючих чатів при завантаженні головної сторінки (див. фрагмент сторінки у додатку Ж).

Створення нового приватного чату

Ініціація створення нового чату відбувається за допомогою натискання кнопки додавання, яка відкриває модальне вікно пошуку користувачів. Повний код компонента модального вікна наведено в додатку Ж, а його роботу описано в наступних пунктах.

Пошук реалізовано через періодичні запити до бази даних із затримкою в одну секунду після введення символів. Це дозволяє оптимізувати навантаження на сервер і забезпечити динамічне формування списку користувачів, з якими ще не створено приватного чату.

Якщо натиснути на користувача зі списку, то відобразиться його профіль у новому модальному вікні. При натисканні кнопки плюса біля обраного користувача створюється відповідний запис у таблиці PrivateChats, і чат відображається в інтерфейсі поточного користувача, стаючи першим у списку чатів. Одночасно, за допомогою технології SignalR, другий учасник отримує оновлення про новий чат у реальному часі. Механізм обміну повідомленнями через хаб реалізовано у компоненті NewChatHub (див. додаток И).

Ключовий фрагмент коду, що відповідає за створення нового приватного чату, виглядає наступним чином:

```
var newChat = new PrivateChat
{
   User1Id = currentUserId,
   User2Id = selectedUser.Id,
   LastMessageDate = DateTime.UtcNow
};
```

```
DbContext.PrivateChats.Add(newChat);
await DbContext.SaveChangesAsync();

await NewChatConnection.SendAsync("SendPrivateChat", currentUserId, currentChatModel);
await NewChatConnection.SendAsync("SendPrivateChat", selectedUser.Id, otherChatModel);
```

Цей код створює новий запис у базі даних, а також надсилає оновлення через хаб до обох учасників.

Відображення та управління приватними чатами

Після створення або вибору чату користувач потрапляє до інтерфейсу перегляду повідомлень. Одночасно відкриваються два ключових компоненти: компонент повідомлень та деталі приватного чату. Останній виводить основну інформацію про співрозмовника: тег (username), відображуване ім’я, короткий опис та дату реєстрації (див. додаток К).

У головній частині сторінки чати організовано у вигляді списку, що автоматично оновлюється у разі надходження або видалення чатів:

```
@if (privateChats.Any())
{
   <ul class="list-group">
      @foreach (var chat in privateChats)
      {
<li class="chat-element @((currentChatId == chat.ChatId) && currentChatType == "private" ? "selected-chat" : "")"
@onclick="@(() => OpenPrivateChat((chat.ChatId, chat.UserId, chat.ChatName)))">
                <div class="icon-div">
                   @chat.ChatName
                </div>
             </li>
      }
   </ul>
}
```

Реактивність через SignalR

Для обміну інформацією між клієнтами застосовується SignalR. Його використання забезпечує миттєве оновлення інтерфейсу у разі створення або видалення чату:

```
newChatConnection.On<PrivateChatModel>("ReceivePrivateChat", (chat) =>
{
   AddPrivateChat(chat);
   InvokeAsync(StateHasChanged);
});

newChatConnection.On<int>("RemovePrivateChat", (chatId) =>
{
   privateChats.RemoveAll(x => x.ChatId == chatId);
   if(currentChatType == "private" && currentChatId == chatId)
   {
      isChatSelected = false;
   }
   InvokeAsync(StateHasChanged);
});
```

Таким чином, усі дії з приватними чатами синхронізуються між користувачами, що суттєво покращує взаємодію та зменшує ризик розсинхронізації клієнтів.

Видалення приватного чату

Функціональність видалення чату також реалізована через відповідний метод хабу:

```
public Task DeletePrivateChat(int userId, int chatId)
{
   return Clients.Group($"getchat_{userId}").SendAsync(
      "RemovePrivateChat" , chatId);
}
```

Після видалення чату один користувач миттєво бачить зміни, а другий отримує відповідне повідомлення, що видаляє чат зі списку

У результаті реалізована підсистема приватних чатів дозволяє користувачам швидко знаходити співрозмовників, починати з ними діалог, переглядати деталі, обмінюватися повідомленнями у режимі реального часу та за потреби видаляти чати. Уся ця функціональність базується на комбінації

Blazor Server, EF Core для взаємодії з БД та SignalR для двосторонньої передачі даних між клієнтами.

3.4.4 Групові чати

Групові чати реалізовані як окремий тип чату в межах основної сторінки застосунку (див. ДОДАТОК Е), займаючи другу панель зліва після приватних чатів. Їх створення, управління та оновлення здійснюється через збереження даних у базі даних та використання механізмів SignalR для синхронізації змін у режимі реального часу (ДОДАТОК И).

На відміну від приватних чатів, створення групового чату не передбачає вибір учасника — при натисканні кнопки створення чат одразу додається до бази даних з дефолтною назвою, а користувач, який його створив, автоматично додається як перший учасник із роллю адміністратора. Цей підхід дозволяє в майбутньому реалізувати функціонал керування правами доступу. Код функції створення нового групового чату виглядає наступним чином:

```
private async Task AddGroupChat()
{
   var newChat = new GroupChat{ Name = "New chat",
        LastMessageDate = DateTime.UtcNow
   };
   DbContext.GroupChats.Add(newChat);
   await DbContext.SaveChangesAsync();

   var newMember = new Member{
      GroupId = newChat.Id,
      UserId = currentUser.Id,
      Role = "admin"
   };

   DbContext.Members.Add(newMember);
   await DbContext.SaveChangesAsync();

   GroupChatModel model = new GroupChatModel{
      ChatId = newChat.Id,
      ChatName = newChat.Name,
      Role = newMember.Role
```

```
    };

    groupChats.Insert(0, model);
    await OpenGroupChat((model.ChatId, model.ChatName));
}
```

Детальна модель GroupChat, як і зв’язок через таблицю Members, подані в ДОДАТКУ В. Всі дані зберігаються з урахуванням останньої дати повідомлення, що дозволяє сортувати чати за активністю.

Після створення чату або його вибору користувачеві відкривається права частина інтерфейсу, де відображається компонент GroupChatDetails. Саме він відповідає за управління назвою чату, відображенням учасників та надає можливість залишити чат або додати нових користувачів (див. ДОДАТОК Е та ДОДАТОК Ж). Код зміни назви чату ініціюється за допомогою логіки редагування вводу, після чого оновлення передається усім учасникам через SignalR:

```
newChatConnection.On<int, string>("ChangeGroupChatName", (chatId,
chatName) =>
{
    var chat = groupChats.FirstOrDefault(c => c.ChatId == chatId);
    if (chat != null)
    {
        chat.ChatName = chatName;

        if(currentChatId == chatId && currentChatType == "group")
        {
            currentChatName = chatName;
        }
        InvokeAsync(StateHasChanged);
    }
});
```

Список учасників групового чату оновлюється у всіх користувачів у реальному часі. При додаванні нового учасника, він одразу отримує доступ до групового чату, а у вже наявних учасників список оновлюється через відповідні виклики SignalR. Відповідна логіка оновлення SignalR винесена у ДОДАТОК И, а сервіси, які реалізують основні операції — у ДОДАТОК Л.

Користувач може залишити чат, натиснувши відповідну кнопку, яка відкриває модальне вікно підтвердження (ConfirmMW). Після підтвердження користувач видаляється зі списку учасників, і ця дія синхронізується з іншими клієнтами:

```
<ConfirmMW @ref="deleteChatDialog"
   ConfirmationTitle="Leave group chat"
   ConfirmationMessage="Are you sure you want to leave this chat?"
   OnConfirm="LeaveGroupChat"
   OnCancel="CancelLeaveChat" />
```

Додатково реалізована можливість відкриття профілю будь-якого учасника чату шляхом натискання на його ім’я в списку. Форма пошуку нових учасників до групи працює за тим самим принципом, що і пошук користувачів при створенні приватного чату (цей функціонал буде описано у наступному підрозділі, а реалізацію можна переглянути в ДОДАТКУ Ж).

У коді головної сторінки можна побачити, що групові чати обробляються окремо від приватних — зокрема, вони оновлюються лише у випадку, коли повідомлення надсилає сам користувач, на відміну від приватних чатів, де будь-яке нове повідомлення пересуває чат догори списку. Це дозволяє уникнути плутанини у великій кількості групових чатів із неактуальними обговореннями.

На завершення, важливо відзначити, що всі компоненти групового чату реалізовані із дотриманням принципів розділення відповідальності. Вся взаємодія з даними реалізується через контекст Entity Framework Core (ДОДАТОК В), структури об'єктів чітко визначені через моделі (ДОДАТОК Д), а компоненти Blazor створені таким чином, щоб їх можна було легко розширювати або повторно використовувати (ДОДАТКИ К та Е). Такий підхід забезпечує зручність підтримки, масштабованість та надійність функціоналу групових чатів.

3.4.5 Пошук користувачів в системі

Функціональність пошуку користувачів реалізована у двох модальних компонентах: вікні створення приватного чату та вікні додавання учасників до

групового чату. Обидва компоненти мають спільну логіку пошуку користувачів у базі даних, яка виконується асинхронно з використанням таймеру та автоматично оновлює результати у міру введення тексту пошуку. Алгоритм пошуку користувача представлено у вигляді блок-схеми на рисунку 19.

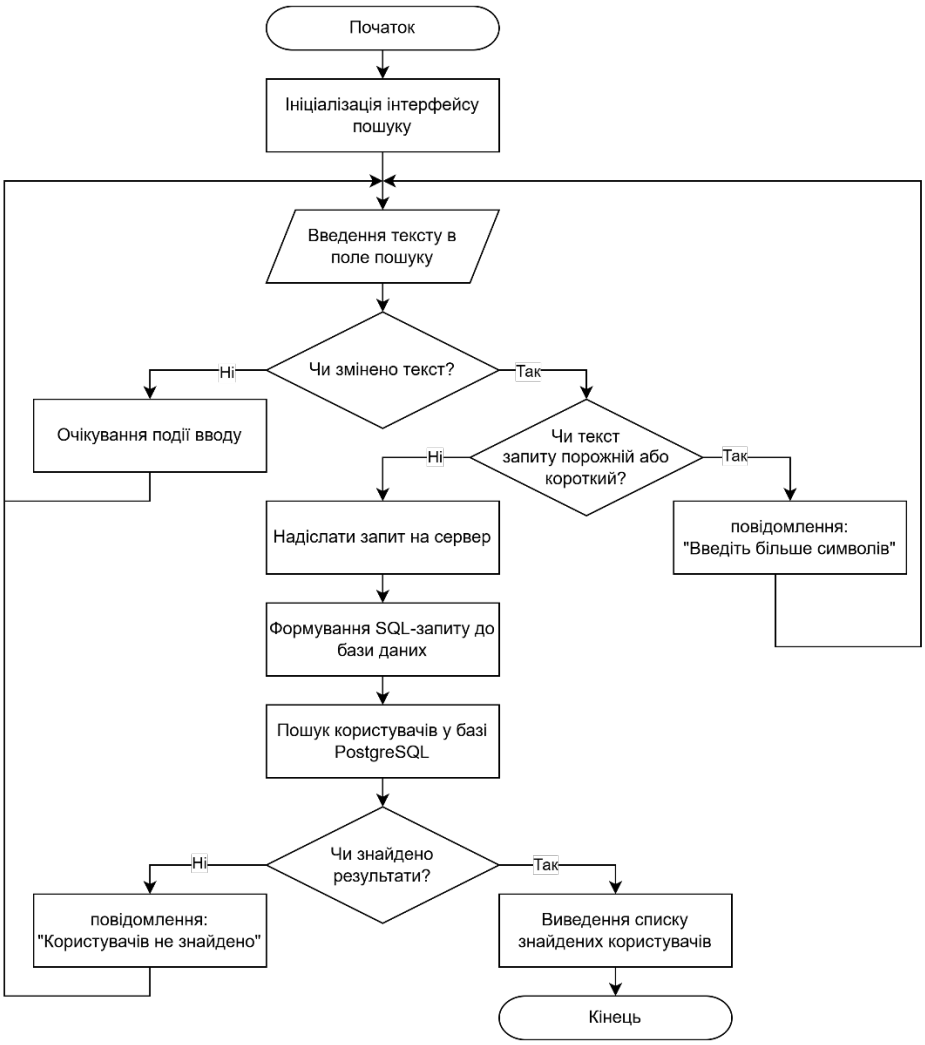

Рис. 19 Блок схема пошуку користувачів

Модальні вікна відкриваються при натисканні відповідних кнопок, після чого у поле введення можна ввести частину імені користувача або його унікального псевдоніму (UserName). Компонент ініціює таймер, який кожну секунду перевіряє введене значення, і в разі його зміни виконує запит до бази даних. У цьому запиті здійснюється пошук користувачів за умовою часткового

збігу імені, а також виключаються користувачі, які вже є учасниками поточного чату або мають існуючий приватний чат з поточним користувачем.

Основна логіка пошуку в обох компонентах реалізована через метод SearchUsers, який можна переглянути у Додатку Ж. Наприклад, у компоненті створення приватного чату він має наступний вигляд:

```
FoundUsers = await DbContext.Users
    .Where(u =>
        u.UserName.ToLower().Contains(SearchText.ToLower()) &&
        u.Id != currentUserId &&
        !existingChatUserIds.Contains(u.Id))
    .Select(u => new User
    {
        Id = u.Id,
        UserName = u.UserName,
        DisplayName = u.DisplayName
    })
    .Take(10)
    .ToListAsync();
```

Аналогічна логіка реалізована і для компонента додавання до групового чату, проте там фільтрація відбувається на основі списку поточних учасників групи:

```
FoundUsers = await context.Users
    .Where(u =>
        u.UserName.ToLower().Contains(SearchText.ToLower()) &&
        u.Id != currentUserId &&
        !existingMemberIds.Contains(u.Id))
    .Select(u => new User
    {
        Id = u.Id,
        UserName = u.UserName,
        DisplayName = u.DisplayName
    })
    .Take(10)
    .ToListAsync();
```

Результати пошуку відображаються у вигляді списку, де кожен елемент містить кнопку перегляду профілю користувача та кнопку додавання. Натискання на ім’я користувача відкриває окреме модальне вікно з його

профілем. Детальний опис перегляду профілю користувача наведено у наступному пункті.

Додавання приватного чату або нового учасника до групи супроводжується оновленням інтерфейсу інших користувачів за допомогою технології SignalR. У разі створення нового чату, повідомлення надсилається двом учасникам:

```
await NewChatConnection.SendAsync("SendPrivateChat", currentUserId, currentChatModel);
await NewChatConnection.SendAsync("SendPrivateChat", selectedUser.Id, otherChatModel);
```

У разі додавання до групового чату аналогічно розсилається інформація про нового учасника:

```
await NewMemberConnection.SendAsync("SendMember", CurrentChatId, memberModel);
await NewChatConnection.SendAsync("SendGroupChat", selectedUser.Id, new GroupChatModel
{
   ChatId = CurrentChatId,
   ChatName = CurrentChatName
});
```

Повний код логіки модальних вікон представлений у Додатку Ж, а логіка обміну повідомленнями через SignalR у Додатку И. Робота з контекстом бази даних реалізована з використанням MyDbContext, опис якого є Додатку В.

Таким чином, реалізація пошуку забезпечує зручний інтерфейс для взаємодії користувачів з системою та автоматично виключає вже наявні зв’язки, запобігаючи дублюванню чатів чи учасників.

3.4.6 Вікно профіля користувачів

Функціональність перегляду профілю користувача реалізована у вигляді модального вікна, яке відображається у відповідь на взаємодію користувача з елементами інтерфейсу, пов’язаними з аватаром або обліковим записом інших користувачів. Зокрема, це модальне вікно може бути відкрито в таких випадках: під час додавання нового чату або додавання учасників до існуючого; при

натисканні на блок з інформацією про поточного користувача (у правому нижньому куті інтерфейсу); а також при перегляді деталей приватного чату — через аватар співрозмовника у верхній частині діалогу. Такий підхід забезпечує зручну та контекстну взаємодію з профілями користувачів.

Модальне вікно динамічно завантажує інформацію про користувача з бази даних на основі його унікального ідентифікатора. Основна логіка завантаження даних розміщена в компоненті, код якого наведено в Додатку Ж. Для збереження читабельності та модульності, усі параметри компонента передаються через вхідні властивості, а саме: IsOpen — прапорець відображення вікна, userId — ідентифікатор користувача, та OnClose — колбек для закриття вікна.

При ініціалізації компонента викликається метод OnParametersSetAsync, у якому створюється контекст бази даних через фабрику IDbContextFactory<MyDbContext>, а потім здійснюється запит на отримання об’єкта UserDetailsDto. Отримані дані містять такі поля, як відображуване ім’я (DisplayName), ім’я користувача (UserName), опис профілю (About) і дата приєднання до системи (JoinDate). Формування цього об’єкта відображено в наступному фрагменті коду:

```
userDetails = await context.Users
    .AsNoTracking()
    .Where(x => x.Id == userId)
    .Select(x => new UserDetailsDto
    {
        DisplayName = x.DisplayName,
        UserName = x.UserName,
        About = x.About,
        JoinDate = x.JoinDate
    })
    .FirstOrDefaultAsync();
```

Відповідна структура UserDetailsDto, що використовується для передачі даних у представлення, описана у Додатку Д.

Інтерфейс модального вікна реалізований у вигляді діалогового блоку, який містить іконку користувача, його ім’я, коротку біографічну інформацію та

дату приєднання. Якщо поле «Про себе» відсутнє, виводиться повідомлення про відсутність опису. Закриття модального вікна викликає метод Close, що, у свою чергу, ініціює подію OnClose, яка визначається зовнішнім компонентом:

```
private void Close(){
   OnClose.InvokeAsync();
}
```

Сама структура модального вікна детально представлена у Додатку Е.

Для взаємодії з базою даних у компоненті використовується контекст, визначений у Додатку В, що дозволяє ефективно виконувати запити з використанням AsNoTracking() для підвищення продуктивності та безпечного отримання даних.

3.4.7 Повідомлення

Функціональність обміну повідомленнями є центральним елементом програмного забезпечення, що реалізує миттєву комунікацію між користувачами. Архітектура цієї підсистеми побудована із застосуванням SignalR-хабу, двох Blazor-компонентів, сервісу для зв’язку з хабом, а також відповідної моделі повідомлень, що зберігається у базі даних.

Передача повідомлень у системі здійснюється через компонент MessageBox, який відповідає за формування та відправлення повідомлення до відповідного хабу. Інший компонент — MessagesList — відповідає за відображення історії листування у межах вибраного чату та обробку нових або видалених повідомлень у реальному часі (Додаток К).

Комунікація клієнта з сервером реалізована через ChatHubService, який ініціалізується на сторінці разом із компонентами, пов’язаними з чатами. Під час зміни активного чату сервіс виконує вихід із попередньої групи в хабі SignalR та приєднується до нової групи. Важливо, що ідентифікатори груп у хабі формуються за шаблоном {тип чату}_{ідентифікатор чату}. Це дозволяє уникнути перетинів між приватними та груповими чатами, навіть якщо їхні числові ідентифікатори збігаються, запобігаючи потенційному витоку даних (Додаток И).

Основні методи сервісу ChatHubService, який забезпечує підключення до хабу та відправку повідомлень, показано нижче:

```
public async Task SendMessageAsync(MessageModel message)
{
    await _hubConnection!.InvokeAsync("SendMessage", message);
}

public async Task JoinChatAsync(string chatId)
{
    if (_currentChatId != null)
        await _hubConnection!.InvokeAsync("LeaveChat", _currentChatId);

    _currentChatId = chatId;
    await _hubConnection!.InvokeAsync("JoinChat", chatId);
}
```

На стороні сервера розташований SignalR-хаб ChatHub, який обробляє приєднання до груп, відправлення повідомлень та повідомлення про їхнє видалення. Код хабу наведено у Додатку И. Ключовий метод, що відповідає за доставку повідомлень усім учасникам групи:

```
public async Task SendMessage(MessageModel mModel)
{
    await Clients.Group($"{mModel.ChatType}_{mModel.ChatId}")
        .SendAsync("ReceiveMessage", mModel);
}
```

Компонент MessageBox здійснює відправлення повідомлення. Під час цього повідомлення записується до бази даних, а також створюється екземпляр моделі MessageModel, який надсилається через сервіс у хаб:

```
var messageModel = new MessageModel
{
    MessageId = messageEntity.Id,
    ChatId = CurrentChatId,
    SenderId = CurrentUserId,
    SenderDisplayName = CurrentUserDisplayName ?? string.Empty,
    Text = messageEntity.Text,
    SentAt = messageEntity.Date,
    ChatType = CurrentChatType
};
```

```
await ChatHub.SendMessageAsync(messageModel);
```

Повний код компонента наведено у Додатку К. Також у процесі надсилання повідомлення оновлюється дата останнього повідомлення в чаті у відповідній таблиці бази даних (PrivateChats або GroupChats), що відображено у моделі та контексті бази (Додаток В).

Компонент відображення повідомлень (ChatMessages) містить логіку завантаження історії повідомлень при відкритті чату, обробки подій отримання або видалення повідомлень, а також забезпечує динамічне оновлення інтерфейсу. Для групових чатів реалізовано виведення імені відправника, тоді як у приватному чаті ця інформація прихована:

```
@if (ChatType == "group" && CurrentUserId != msg.SenderId)
{
   <p class="m-name-other">@msg.SenderDisplayName</p>
}
```

Під час ініціалізації компонент приєднується до хабу та підписується на події отримання повідомлень:

```
protected override async Task OnInitializedAsync()
{
   ChatHub.OnMessageReceived += HandleIncomingMessage;
   ChatHub.OnMessageDeleted += HandleMessageDeleted;
   await ChatHub.StartAsync();
}
```

Після отримання нового повідомлення, воно додається до списку, і виконується оновлення інтерфейсу користувача та прокрутка донизу. У випадку видалення повідомлення, воно також видаляється з поточного списку.

Таким чином, модуль повідомлень забезпечує передачу, збереження та динамічне відображення повідомлень із підтримкою реального часу. Усі аспекти цієї логіки охоплюють компоненти інтерфейсу (Додаток Е), модальні вікна (Додаток Ж), хаб SignalR (Додаток И), допоміжні компоненти (Додаток К), моделі даних (Додаток Д) та відповідні сервіси (Додаток Л).

3.4.8 Налаштування користувача

Модальне вікно налаштувань призначене для редагування особистої інформації користувача, зокрема його відображуваного імені, опису (about), електронної пошти, а також зміни пароля. Реалізація цього функціоналу спрямована на забезпечення зручного та безпечного способу зміни персональних даних користувачем, із урахуванням обмежень і валідації введених значень.

Інтерфейс вікна представлено у вигляді окремого компонента Blazor (див. ДОДАТОК Ж), що активується при відповідній події та містить форми для введення нових значень. Для кожного поля редагування передбачена логіка активації за допомогою кнопки редагування. У разі її натискання, активується відповідне текстове поле, а замість кнопки редагування з’являються кнопки підтвердження або скасування змін.

Щоб запобігти одночасному редагуванню кількох полів та уникнути можливих конфліктів у стані компонента, реалізовано загальний логічний прапорець isEditing, який блокує доступ до редагування інших полів під час поточного редагування. Такий підхід забезпечує логічну цілісність стану користувацького інтерфейсу та сприяє уникненню помилок взаємодії (див. фрагмент реалізації логіки на прикладі імені користувача у ДОДАТОК Ж):

```
private async Task EnableDisplaynameEditing()
{
    if (!isEditing)
    {
        isEditing = true;
        isDisplaynameEditable = true;
        await Task.Yield();
        await JS.InvokeVoidAsync("focusElement", displaynameInputRef);
    }
}
```

Уведені дані піддаються перевірці на коректність перед збереженням. Наприклад, ім’я користувача має містити від 3 до 100 символів, опис — не перевищувати 1000 символів, а електронна пошта — відповідати шаблону регулярного виразу. У разі порушення обмежень користувач отримує відповідне

повідомлення про помилку. Типовий приклад перевірки на коректність імені наведено нижче:

```
if (displaynameInputValue.Length < 3)
{
   displaynameError = "Display name can't be shorter than 3 symbols";
   return;}
```

Після підтвердження зміни, оновлена інформація зберігається в базі даних за допомогою контексту DbContext, який реалізує взаємодію з таблицею користувачів (див. ДОДАТОК В). Зміна, наприклад, імені користувача виглядає наступним чином:

```
userProfile.DisplayName = displaynameInputValue;
DbContext.Users.Update(userProfile);
await DbContext.SaveChangesAsync();
```

Окрема увага приділена редагуванню електронної пошти. У разі зміни адреси поле IsVerified у моделі користувача скидається в значення false, що ініціює повторну процедуру верифікації (сам процес описується у наступному підпункті). Перевірка унікальності нової адреси електронної пошти реалізується через запит до бази даних:

```
bool emailExists = await DbContext.Users.AnyAsync(u => u.Email == emailInputValue);
if(emailExists)
{
   emailError = "This email already exists";
   return;
}
```

Також у цьому вікні користувач може змінити пароль, ввівши поточний пароль та двічі новий. Реалізована перевірка на відповідність і наявність усіх необхідних полів (детальніше про логіку обробки у ДОДАТКУ Ж).

Під час взаємодії з інтерфейсом активно використовується сервіс UserStateService, який забезпечує доступ до поточного профілю користувача

(див. ДОДАТОК Л), а також сервіси збереження стану в браузері ProtectedLocalStorage, що підвищує безпеку взаємодії з персональними даними.

Таким чином, модальне вікно налаштувань є повноцінним функціональним модулем системи, який забезпечує зручний і безпечний спосіб редагування особистої інформації користувача. Його реалізація є важливою складовою загального UX та демонструє застосування як принципів реактивного програмування, так і належної роботи з базою даних.

3.4.9 Верифікація пошти

У реалізованому програмному забезпеченні підтвердження електронної пошти здійснюється шляхом надсилання спеціального листа на вказану користувачем адресу. У листі міститься посилання з унікальним токеном, який генерується під час ініціалізації процесу верифікації.

Після генерації токену він зберігається у відповідній таблиці бази даних разом із датою завершення його дії та позначкою використання. Ці дані моделюються у базі даних, як представлено у ДОДАТКУ В та ДОДАТКУ Д.

Для надсилання верифікаційного листа використовується сервіс EmailService, реалізований у ДОДАТКУ Л. Він працює з SMTP-сервером Gmail, параметри доступу до якого зберігаються у конфігураційному файлі appsettings.json. Витяг з конфігурації наведено нижче:

```
"Email": {
  "Smtp": {
    "Host": "smtp.gmail.com",
    "Port": "587",
    "Username": "nd120704@gmail.com",
    "Password": "**** **** **** ****"
  }
}
```

Ключовий метод сервісу, що відповідає за надсилання листів, наведено нижче:

```
public async Task SendEmailAsync(string toEmail, string subject, string body)
{
    var smtpClient = new SmtpClient
    {
```

```
            Host = _configuration["Email:Smtp:Host"],
            Port = int.Parse(_configuration["Email:Smtp:Port"]!),
            EnableSsl = true,
            Credentials = new NetworkCredential(
                _configuration["Email:Smtp:Username"],
                _configuration["Email:Smtp:Password"]
            )
        };

        var mailMessage = new MailMessage
        {
            From = new MailAddress(_configuration["Email:Smtp:Username"]!),
            Subject = subject,
            Body = body,
            IsBodyHtml = true
        };

        mailMessage.To.Add(toEmail);
        await smtpClient.SendMailAsync(mailMessage);
    }
```

Сторінка, що відповідає за обробку підтвердження, розміщена за адресою /verify-email. Її код представлений у ДОДАТКУ Е, і вона автоматично зчитує токен з параметрів URL. Далі відбувається перевірка токену в базі даних: він повинен існувати, не бути використаним раніше та бути дійсним за часом. Якщо всі умови виконано, обліковий запис користувача позначається як верифікований, а сам токен вважається використаним.

Ключова частина логіки перевірки виглядає наступним чином:

```
var record = await DbContext.EmailVerifications
    .Include(v => v.User)
    .FirstOrDefaultAsync(v => v.Token == Token && !v.IsUsed);

if (record == null)
{
    Message = "This verification link is invalid or already used.";
    IsSuccess = false;
    StateHasChanged();
    return;
}
```

```
if (record.ExpiresAt < DateTime.UtcNow)
{
    Message = "This verification link has expired.";
    IsSuccess = false;
    StateHasChanged();
    return;
}

record.IsUsed = true;
record.User.IsVerified = true;
await DbContext.SaveChangesAsync();
UserState.SetIsVerified(true);
Message = "Your email has been successfully verified!";
IsSuccess = true;
```

Оскільки підтвердження відбувається на окремій сторінці, для відображення змін у модальному вікні налаштувань користувача після повернення до нього реалізовано спеціальний сервіс UserStateService (ДОДАТОК Л). Він використовує подієву модель, яка дозволяє оновити інтерфейс налаштувань у реальному часі після завершення верифікації. Приклад коду сервісу:

```
public class UserStateService
{
    public bool IsVerified { get; private set; }
    public event Action? OnChange;

    public void SetIsVerified(bool isVerified)
    {
        IsVerified = isVerified;
        OnChange?.Invoke();
    }
}
```

У візуальному інтерфейсі користувачеві, чия пошта ще не була підтверджена, відображається відповідна кнопка. При її натисканні з’являється повідомлення про те, що лист для підтвердження був надісланий.

4 ВПРОВАДЖЕННЯ СИСТЕМИ

4.1 Тестування системи

На завершальному етапі розробки програмного забезпечення були проведені тести з метою перевірки його функціональності, відповідності заявленим вимогам та виявлення існуючих багів. Тести проводились у тестовому середовищі, у межах однієї локальної машини із використанням актуальної версії веб-застосунку на етапі розробки.

Найбільшу увагу було надано тестуванню основних функціональних модулів застосунка: реєстрації та входу у систему, пересилки повідомлень у реальному часі, створення та видалення чатів і перевірки даних у базі даних PostgreSQL. Зважаючи на особливості обраного підходу до розробки, який включає компоненти Blazor Server та WebSocket-підключення, особливу увагу також було надано стійкості роботи з'єднання та швидкості надсилання та отримання повідомлень між користувачами.

Тестування відбувалось у ручному режимі, шляхом імітації стандартної поведінки користувачів під час роботи із застосунком. Була проведена реєстрація декількох тестових облікових записів з метою перевірки сценаріїв входу у систему та перевірки роботи логіки, пов'язаної із введенням некоректних даних. Усі перевірені сценарії завершилися успіхом, що свідчить про правильність написаного коду.

Окрім вище зазначеного функціонального тестування, була проведена перевірка бази даних на цілісність збережених даних. Було переконанося у тому, що при аналогічних вище описаних сценаріях використання, відбувається коректне збереження, оновлення та видалення необхідних записів про повідомлення, чати та користувачів. Було проведено тестування навантажень у рамках наявного тестового середовища: при одночасній роботі декількох

користувачів система демонструвала стабільну роботу, без помітного зниження продуктивності.

В цілому проведене тестування показало, щ програмне забезпечення задовольняє вимоги, що були закладені при дизайні. Система працює коректно, а виявлені дрібні недоліки були усунені в процесі доопрацювання. Тобто, розроблена система є готовою до подальшого впровадження.

4.2 Вимоги до апаратного та програмного забезпечення

Для того щоб система працювала стабільно та ефективно, важливо дотримуватися певних апаратних і програмних вимог як на сервері, так і на клієнтських пристроях. Архітектура застосунку побудована на Blazor Server, що забезпечує активну взаємодію між клієнтом і сервером у реальному часі за допомогою SignalR.

Серверна частина системи виконує важливі функції: обробляє запити, взаємодіє з базою даних PostgreSQL, відповідає за автентифікацію користувачів, а також керує чатами та повідомленнями. Рекомендується, щоб сервер мав такі апаратні характеристики: процесор не нижче Intel Core i5 10-го покоління, 16 ГБ оперативної пам’яті, 200 ГБ SSD та стабільне інтернет-з’єднання з швидкістю від 100 Мбіт/с.Програмне забезпечення сервера повинно включати:

- Операційну систему Windows Server 2019+ або Ubuntu Server 20.04+;
- Середовище .NET 8.0+;
- СУБД PostgreSQL 14+;
- Веб-сервер IIS, Nginx або Apache з підтримкою SSL-сертифікатів.

Клієнтська частина була створена на Blazor і доступна через браузер. Для користувачів достатньо мати пристрій, який підтримує сучасний веб-браузер та має доступ до Інтернету. Мінімальні вимоги для клієнтських пристроїв:

- Процесор: Intel Pentium G4400 або аналогічний;
- ОЗП: 2 ГБ;

- Вільне місце: 200 МБ;
- Інтернет: від 2 Мбіт/с.

Підтримувані браузери: Google Chrome, Microsoft Edge, Mozilla Firefox, Safari, Opera (версії 88.0+ або новіші). Для повноцінного використання необхідно мати обліковий запис, увімкнений JavaScript у браузері та доступ до електронної пошти щоб підтвердження реєстрації.

4.3 Візуальна демонстрація роботи програми

У цьому підрозділі наведено візуальну демонстрацію ключових етапів взаємодії користувача з веб-застосунком, що реалізує функціонал обміну повідомленнями в реальному часі.

На рисунку 20 зображено форму реєстрації насторінці реєстрації, де необхідно ввести логін, електронну пошту та пароль.

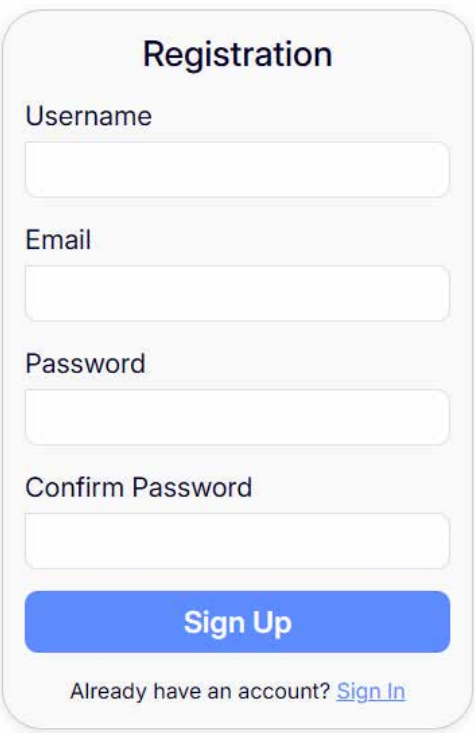

Рис. 20 Сторінка реєстрації

Інтерфейс передбачає зручне відображення помилок при некоректному введенні даних, що сприяє покращенню користувацького досвіду (рис. 21).

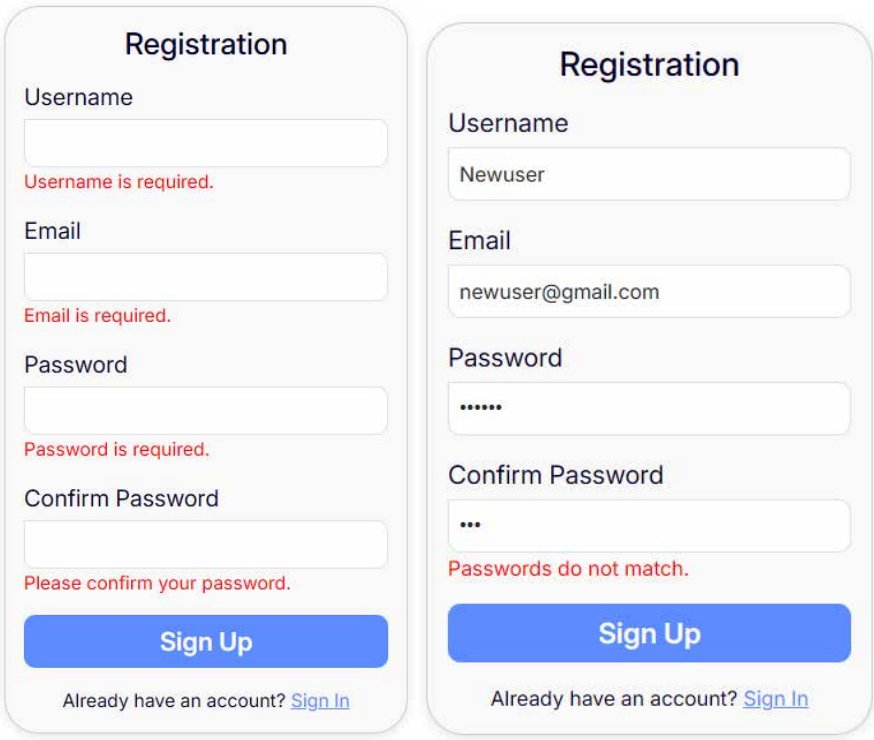

Рис. 21 Помилки при реєстрації

На рис. 22 представлена форма авторизації користувача при введенні даних. Після введення правильних облікових даних користувач отримує доступ до системи.

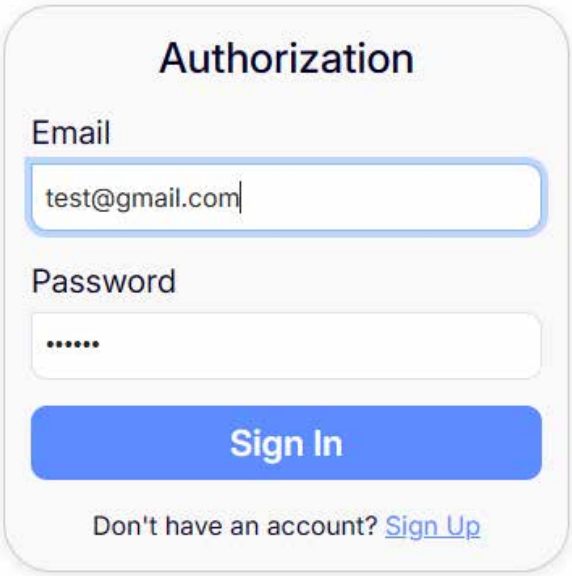

Рис. 22 Форма авторизації

Після успішного входу до системи користувач потрапляє на головну сторінку застосунку, що зображена на рисунку 23. На цій сторінці відображаються наявні чати, а також надається можливість створити новий чат або групу.

Рис. 23 Головна сторінка

На рисунку 24 показано модальне вікно пошуку користувачів для створення нового чату. Це вікно виглядає так само і для додавання учасника в груповий чат. Пошук здійснюється у реальному часі за частковим збігом імені користувача, що значно полегшує взаємодію.

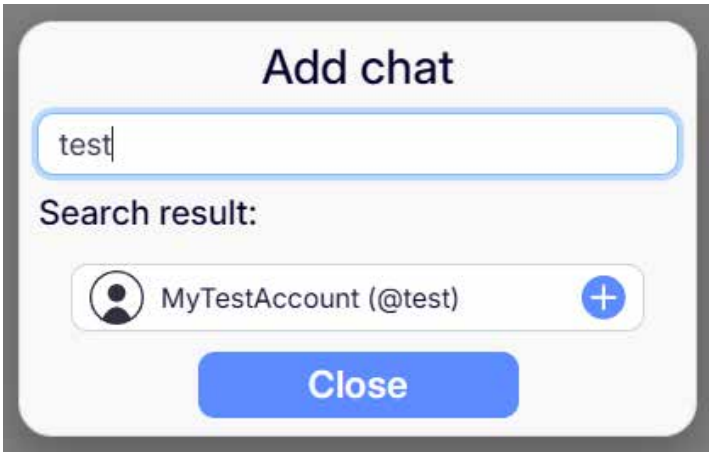

Рис. 24 Вікно пошуку користувачів

Після створення групового чату реалізована можливість перегляду учасників чату, як показано на рисунку 25.

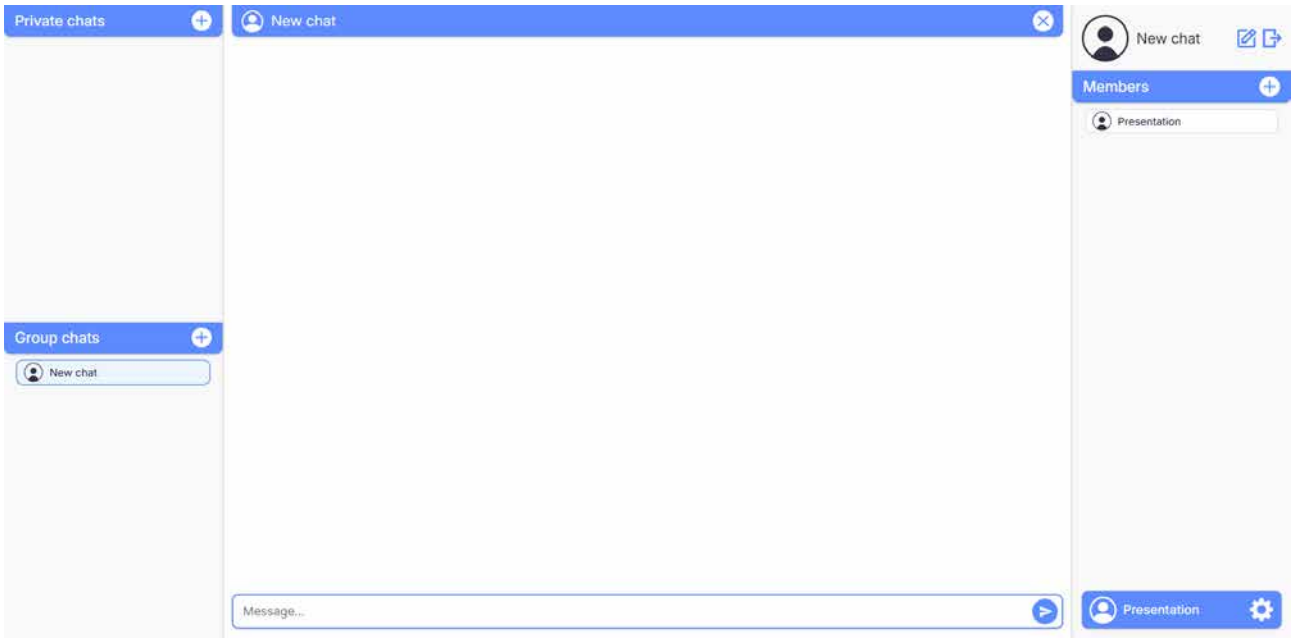

Рис. 25 Створений груповий чат та деталі чату

На рисунку 26 продемонстровано інтерфейс налаштувань профілю, де користувач може переглядати свої дані та змінювати персональні параметри. Додатково показано редагування імені. Один із прикладів зміни даних – редагування імені, представлено на рисунку 27.

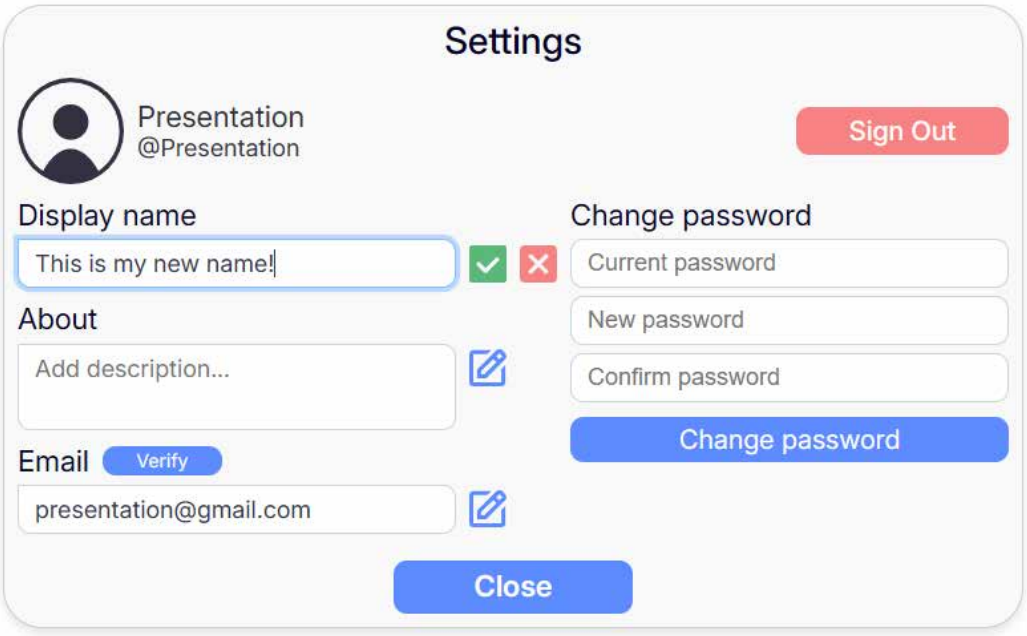

Рис. 26 Модальне вікно налаштувань, та редагування імені

Рисунок 27 показує верифікацію електронної пошти. Після реєстрації користувачеві надсилається лист із підтвердженням, а в інтерфейсі з'являється повідомлення про необхідність підтвердження. Після переходу за відповідним посиланням у листі обліковий запис активується.

Рис. 27 Вікно після успішної верифікації

Рисунок 28 показує повідомлення в груповому чаті. За різного вмісту повідомлень добре видно як ведуть себе блоки з повідомленнями.

Рис. 28 Повідомлення в груповому чаті

ВИСНОВОК

При виконанні даної дипломної роботи було розроблено програмний веб-додаток для обміну повідомленнями в режимі реального часу. Після аналізу предметной області були визначені головні функціональні та нефункціональні вимоги, що дозволило сформувати повністю продумане технічне завдання на стадії проектування.

У розділі проектуання було побудовано модель майбутньої системи обміну повідомленнями в реальному часі використовуючи діаграми UML, які включали логічну модель, архітектурні компоненти та сценарії взаємодії. Цей підхід надав послідовність розробки та дозволив ефективно організувати архітектуру і структуру коду програми.

Процес вибору влаштовуючого набору технологій, а саме - . NET 9, Blazor Server, PostgreSQL, Entity Framework Core, та SignalR – які надають чудову продуктивність, легкіцсть розробки, та підтримку дій в реальному часі були виправдані на етапі розробки програмного забезпечення. Побудові бази даних та впровадженні авторизації, реєстрації, чатів, і управління користувачами було надано особливу увагу.

В наслідок впровадження системи, проведено тестування що підтвердило правильність реалізації основних функцій. Розглянуто вимоги до апаратного і програмного забезпечення від клієнтів й сервера; це дозволяє масштабувати систему в залежності від зростаючого навантаження. Через візуальну демонстрацію показано ключові частини інтерфейсу користувача, включаючи процеси реєстрації входу взаємодії в чатах та управління профілем.

Отже, цілі в роботі досягнуті повністю. Створений додаток можна використовувати як основу для створення системи миттєвого обміну повідомленнями з можливістю покращення, інтеграції нових функцій та збільшення масштабу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ChatGPT - text generation tool [Електронний ресурс] – Режим доступу до ресурсу: https://chatgpt.com/
2. GeeksforGeeks. Use Case Diagram – Unified Modeling Language (UML) [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/use-case-diagram/?utm_source=chatgpt.com
3. Visual Paradigm. What is Use Case Diagram? [Електронний ресурс] – Режим доступу до ресурсу: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/?utm_source=chatgpt.com
4. GeeksforGeeks. Sequence Diagrams – Unified Modeling Language (UML) [Електронний ресурс] – Режим доступу до ресурсу: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/?utm_source=chatgpt.com
5. Visual Paradigm. What is Sequence Diagram? [Електронний ресурс] – Режим доступу до ресурсу: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/?utm_source=chatgpt.com
6. GeeksforGeeks. Activity Diagrams – Unified Modeling Language (UML) [Електронний ресурс] – Режим доступу до ресурсу: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/?utm_source=chatgpt.com
7. GeeksforGeeks. Domain Modeling – Software Engineering [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/software-engineering-domain-modeling/?utm_source=chatgpt.com
8. ThoughtWorks. Domain Modeling: What you need to know before coding [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/software-engineering-domain-modeling/?utm_source=chatgpt.com

9. Telegram. MTProto Mobile Protocol [Електронний ресурс] – Режим доступу до ресурсу: https://blogfork.telegram.org/mtproto
10. Discord. How Discord Handles Two and Half Million Concurrent Voice Users using WebRTC [Електронний ресурс] – Режим доступу до ресурсу: https://discord.com/blog/how-discord-handles-two-and-half-million-concurrent-voice-users-using-webrtc/
11. WhatsApp Data Security: End-to-End Encryption and Backups [Електронний ресурс] – Режим доступу до ресурсу: https://www.wati.io/blog/understanding-whatsapp-data-security-understand-end-to-end-encryption-and-backups/.
12. Signal. GitHub - signalapp/libsignal [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/signalapp/libsignal
13. Meta. End-to-End Encryption on Messenger Explained [Електронний ресурс] – Режим доступу до ресурсу: https://about.fb.com/news/2024/03/end-to-end-encryption-on-messenger-explained/
14. UML Diagrams. Overview of ER Diagrams, Class Diagrams, Component and Deployment Diagrams [Електронний ресурс] – Режим доступу до ресурсу: https://www.uml-diagrams.org/
15. Lucidchart. ER Diagram Tutorial: How to Make an ER Diagram [Електронний ресурс] – Режим доступу до ресурсу: https://www.lucidchart.com/pages/er-diagram-symbols-and-examples
16. Microsoft Docs. Entity Framework Core - Modeling and Mapping [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/ef/core/modeling/
17. Visual Paradigm. UML Class Diagram Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://online.visual-paradigm.com/diagrams/tutorials/class-diagram-tutorial/
18. Creately. UML Collaboration Diagrams Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://creately.com/blog/diagrams/uml-collaboration-diagram/

19. Lucidchart. Component Diagram Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://www.lucidchart.com/pages/uml-component-diagram
20. Visual Paradigm. Deployment Diagram Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://online.visual-paradigm.com/diagrams/tutorials/deployment-diagram-tutorial/
21. Microsoft Docs. Blazor Architecture Overview [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/aspnet/core/blazor/architecture
22. Microsoft Docs. ASP.NET Core SignalR overview [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction
23. PostgreSQL Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/docs/current/index.html
24. SignalR у Blazor Server: офіційне керівництво [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/aspnet/core/blazor/tutorials/signalr-blazor?view=aspnetcore-9.0
25. Integrating PostgreSQL with .NET 9 using EF Core: A Step-by-Step Guide [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@vosarat1995/integrating-postgresql-with-net-9-using-ef-core-a-step-by-step-guide-a773768777f2
26. Building a Real-Time Chat Service with .NET Core and SignalR [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@sayyedulawwab/building-a-real-time-chat-service-with-net-core-and-signalrintroduction-f65943dba9f0
27. Entity Framework Core with Blazor Server: офіційна документація [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/aspnet/core/blazor/blazor-ef-core?view=aspnetcore-9.0

28. Npgsql.EntityFrameworkCore.PostgreSQL – офіційна сторінка пакета NuGet [Електронний ресурс] – Режим доступу до ресурсу: https://www.nuget.org/packages/npgsql.entityframeworkcore.postgresql
29. Software Testing Fundamentals. Manual Testing Overview [Електронний ресурс] – Режим доступу до ресурсу: https://softwaretestingfundamentals.com/manual-testing/
30. Guru99. Manual Testing Tutorial for Beginners [Електронний ресурс] – Режим доступу до ресурсу: https://www.guru99.com/manual-testing.html

ДОДАТОК А

Опис прецедентів

Сторінок 2

Київ-2025

Опис прецедентів

Таблиця 1

№	Назва прецеденту	Актор	Опис прецеденту
1	Зареєструватися	Користувач	Створення нового облікового запису з перевіркою правильності пошти, логіна та паролю.
2	Авторизуватися	Користувач	Аутентифікація користувача через введення електронної пошти та паролю.
3	Вийти з аккаунту	Користувач	Завершення поточної сесії та вихід із системи.
4	Переглянути список чатів	Користувач	Виведення переліку всіх доступних приватних та групових чатів користувача.
5	Переглянути учасників групового чату	Користувач	Відображення списку учасників конкретного групового чату.
6	Отримати історію повідомлень	Користувач	Завантаження попередніх повідомлень у вибраному чаті.
7	Пошук користувачів	Користувач	Здійснення пошуку інших користувачів за ім’ям або поштою для ініціації чату.
8	Створити приватний чат	Користувач	Ініціація нового діалогу з вибраним користувачем у форматі приватного чату.
9	Створити груповий чат	Користувач	Створення нового чату з можливістю додати декілька учасників.
10	Відкрити чат	Користувач	Завантаження вмісту вибраного чату для подальшого перегляду та взаємодії.
11	Переглянути історію повідомлень	Користувач	Перегляд раніше надісланих повідомлень у рамках конкретного чату.
12	Додати учасника до групового чату	Користувач	Додавання одного або кількох нових учасників до наявного групового чату.
13	Видалити чат	Користувач	Повне видалення приватного або групового чату з інтерфейсу користувача.

Таблиця 1 (закінчення)

№	Назва прецеденту	Актор	Опис прецеденту
14	Вийти з групового чату	Користувач	Самостійний вихід користувача з групи без видалення чату для інших учасників.
15	Надіслати повідомлення	Користувач	Створення та надсилання текстового повідомлення у вибраний чат.
16	Видалити повідомлення	Користувач	Видалення раніше надісланого власного повідомлення.
17	Переглянути профіль користувача	Користувач	Перегляд інформації профілю іншого або власного користувача.
18	Змінити відображуване ім’я	Користувач	Редагування імені, яке буде відображатись іншим користувачам у чатах.
19	Змінити пароль	Користувач	Заміна чинного паролю для підвищення безпеки облікового запису.
20	Змінити пошту	Користувач	Оновлення електронної адреси з повторною перевіркою та верифікацією.
21	Підтвердити пошту	Користувач	Процедура верифікації пошти після реєстрації або зміни адреси.
22	Додати опис користувача	Користувач	Додавання або редагування текстового опису профілю (біографія, статус тощо).

ДОДАТОК Б

SQL код створення таблиць

Сторінок 1

Київ-2025

```
CREATE TABLE users (
   id SERIAL PRIMARY KEY,
   email VARCHAR(255) UNIQUE NOT NULL,
   user_name VARCHAR(50) UNIQUE NOT NULL,
   password TEXT NOT NULL,
   display_name VARCHAR(100),
   about TEXT,
   join_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   is_verified BOOLEAN DEFAULT FALSE);
CREATE TABLE settings (
   user_id INT PRIMARY KEY REFERENCES users(id) ON DELETE CASCADE,
   theme VARCHAR(20));
CREATE TABLE private_chats (
   id SERIAL PRIMARY KEY,
   user1_id INT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
   user2_id INT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
   last_message_date TIMESTAMP WITH TIME ZONE NOT NULL,
   CONSTRAINT unique_private_chat UNIQUE (user1_id, user2_id),
   CHECK (user1_id <> user2_id));
CREATE TABLE group_chats (
   id SERIAL PRIMARY KEY,
   name VARCHAR(100) NOT NULL,
   last_message_date TIMESTAMP WITH TIME ZONE NOT NULL,);
CREATE TABLE members (
   id SERIAL PRIMARY KEY,
   group_id INT NOT NULL REFERENCES group_chats(id) ON DELETE CASCADE,
   user_id INT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
   role VARCHAR(20) NOT NULL CHECK (role IN ('admin', 'moderator', 'user')),
   CONSTRAINT unique_member_in_group UNIQUE (group_id, user_id));
CREATE TABLE messages (
   id SERIAL PRIMARY KEY,
   user_id INT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
   private_chat_id INT REFERENCES private_chats(id) ON DELETE CASCADE,
   group_chat_id INT REFERENCES group_chats(id) ON DELETE CASCADE,
   text TEXT NOT NULL,
   date TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
   CHECK (
      (private_chat_id IS NOT NULL AND group_chat_id IS NULL)
      OR
      (private_chat_id IS NULL AND group_chat_id IS NOT NULL)
   ));
CREATE TABLE email_verifications (
   id SERIAL PRIMARY KEY,
   user_id INTEGER NOT NULL REFERENCES users(id) ON DELETE CASCADE,
   token TEXT NOT NULL,
   expires_at TIMESTAMP WITH TIME ZONE NOT NULL,
   is_used BOOLEAN NOT NULL DEFAULT FALSE);
```

ДОДАТОК В

Моделі таблиць та контекст роботи з базою даних

Сторінок 4

Київ-2025

Модель верифікацій електронної пошти:

```
public partial class EmailVerification {
    public int Id { get; set; }
    public int UserId { get; set; }
    public string Token { get; set; } = null!;
    public DateTime ExpiresAt { get; set; }
    public bool IsUsed { get; set; }
    public virtual User User { get; set; } = null!; }
```

Модель групових чатів:

```
public partial class GroupChat {
    public int Id { get; set; }
    public string Name { get; set; } = null!;
    public DateTime LastMessageDate { get; set; }
    public virtual ICollection<Member> Members { get; set; } = new List<Member>();
    public virtual ICollection<Message> Messages { get; set; } = new List<Message>();}
```

Модель учасників:

```
public partial class Member {
    public int Id { get; set; }
    public int GroupId { get; set; }
    public int UserId { get; set; }
    public string Role { get; set; } = null!;
    public virtual GroupChat Group { get; set; } = null!;
    public virtual User User { get; set; } = null!; }
```

Модель повідомлення:

```
public partial class Message {
    public int Id { get; set; }
    public int UserId { get; set; }
    public int? PrivateChatId { get; set; }
    public int? GroupChatId { get; set; }
    public string Text { get; set; } = null!;
    public DateTime Date { get; set; }
    public virtual GroupChat? GroupChat { get; set; }
    public virtual PrivateChat? PrivateChat { get; set; }
    public virtual User User { get; set; } = null!; }
```

Модель приватних чатів:

```
public partial class PrivateChat {
    public int Id { get; set; }
    public int User1Id { get; set; }
    public int User2Id { get; set; }
    public DateTime LastMessageDate { get; set; }
    public virtual ICollection<Message> Messages { get; set; } = new List<Message>();
    public virtual User User1 { get; set; } = null!;
```

```
    public virtual User User2 { get; set; } = null!;}
```


Модель налаштувань:

```
public partial class Setting {
    public int UserId { get; set; }
    public string Theme { get; set; } = null!;
    public virtual User User { get; set; } = null!;}
```

Модель користувача:

```
public partial class User {
    public int Id { get; set; }
    public string Email { get; set; } = null!;
    public string UserName { get; set; } = null!;
    public string Password { get; set; } = null!;
    public string DisplayName { get; set; } = null!;
    public string? About { get; set; }
    public DateTime JoinDate { get; set; }
    public bool IsVerified { get; set; }
    public virtual ICollection<EmailVerification> EmailVerifications { get; set; } = new List
<EmailVerification>();
    public virtual ICollection<Member> Members { get; set; } = new List<Member>();
    public virtual ICollection<Message> Messages { get; set; } = new List<Message>();
    public virtual ICollection<PrivateChat> PrivateChatUser1s { get; set; } = new List<Privat
eChat>();
    public virtual ICollection<PrivateChat> PrivateChatUser2s { get; set; } = new List<Privat
eChat>();
    public virtual Setting? Setting { get; set; }}
```

Контекст роботи з бд:

```
public partial class MyDbContext : DbContext{
   public MyDbContext(){}
   public MyDbContext(DbContextOptions<MyDbContext> options) : base(options) {}
   public virtual DbSet<EmailVerification> EmailVerifications { get; set; }
   public virtual DbSet<GroupChat> GroupChats { get; set; }
   public virtual DbSet<Member> Members { get; set; }
   public virtual DbSet<Message> Messages { get; set; }
   public virtual DbSet<PrivateChat> PrivateChats { get; set; }
   public virtual DbSet<Setting> Settings { get; set; }
   public virtual DbSet<User> Users { get; set; }
   protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
      => optionsBuilder.UseNpgsql("Name=DefaultConnection");
   protected override void OnModelCreating(ModelBuilder modelBuilder) {
      modelBuilder.Entity<EmailVerification>(entity =>{
         entity.HasKey(e => e.Id).HasName("email_verifications_pkey");
         entity.ToTable("email_verifications");
         entity.Property(e => e.Id).HasColumnName("id");
         entity.Property(e => e.ExpiresAt).HasColumnName("expires_at");
         entity.Property(e => e.IsUsed).HasDefaultValue(false)
            .HasColumnName("is_used");
```

```
        entity.Property(e => e.Token).HasColumnName("token");
```



```
    entity.Property(e => e.UserId).HasColumnName("user_id");
        entity.HasOne(d => d.User).WithMany(p => p.EmailVerifications)
            .HasForeignKey(d => d.UserId)
            .HasConstraintName("email_verifications_user_id_fkey");});
    modelBuilder.Entity<GroupChat>(entity =>{
        entity.HasKey(e => e.Id).HasName("group_chats_pkey");
        entity.ToTable("group_chats");
        entity.Property(e => e.Id).HasColumnName("id");
        entity.Property(e => e.LastMessageDate)
            .HasDefaultValueSql("CURRENT_TIMESTAMP")
            .HasColumnName("last_message_date");
        entity.Property(e => e.Name).HasMaxLength(100)
            .HasColumnName("name");});
    modelBuilder.Entity<Member>(entity =>{
        entity.HasKey(e => e.Id).HasName("members_pkey");
        entity.ToTable("members");
entity.HasIndex(e => new { e.GroupId, e.UserId }, "unique_member_in_group").IsUnique();
        entity.Property(e => e.Id).HasColumnName("id");
entity.Property(e => e.GroupId).HasColumnName("group_id");
        entity.Property(e => e.Role).HasMaxLength(20)
            .HasDefaultValueSql("'member'::character varying").HasColumnName("role");
        entity.Property(e => e.UserId).HasColumnName("user_id");
        entity.HasOne(d => d.Group).WithMany(p => p.Members)
            .HasForeignKey(d => d.GroupId).HasConstraintName("members_group_id_fkey");
        entity.HasOne(d => d.User).WithMany(p => p.Members).HasForeignKey(d => d.UserId)
            .HasConstraintName("members_user_id_fkey");});
    modelBuilder.Entity<Message>(entity =>{
        entity.HasKey(e => e.Id).HasName("messages_pkey");
        entity.ToTable("messages");
        entity.Property(e => e.Id).HasColumnName("id");
        entity.Property(e => e.Date)
            .HasDefaultValueSql("CURRENT_TIMESTAMP").HasColumnName("date");
        entity.Property(e => e.GroupChatId).HasColumnName("group_chat_id");
        entity.Property(e => e.PrivateChatId).HasColumnName("private_chat_id");
        entity.Property(e => e.Text).HasColumnName("text");
        entity.Property(e => e.UserId).HasColumnName("user_id");
        entity.HasOne(d => d.GroupChat).WithMany(p => p.Messages)
            .HasForeignKey(d => d.GroupChatId).OnDelete(DeleteBehavior.Cascade)
            .HasConstraintName("messages_group_chat_id_fkey");
        entity.HasOne(d => d.PrivateChat).WithMany(p => p.Messages)
            .HasForeignKey(d => d.PrivateChatId).OnDelete(DeleteBehavior.Cascade)
            .HasConstraintName("messages_private_chat_id_fkey");
        entity.HasOne(d => d.User).WithMany(p => p.Messages)
            .HasForeignKey(d => d.UserId).HasConstraintName("messages_user_id_fkey");});
    modelBuilder.Entity<PrivateChat>(entity =>{
        entity.HasKey(e => e.Id).HasName("private_chats_pkey");
        entity.ToTable("private_chats");
        entity.HasIndex(e => new { e.User1Id, e.User2Id }, "unique_private_chat").IsUnique();
 entity.Property(e => e.Id).HasColumnName("id");
```

```
            entity.Property(e => e.LastMessageDate)
```



```
                .HasDefaultValueSql("CURRENT_TIMESTAMP")
                .HasColumnName("last_message_date");
entity.Property(e => e.User1Id).HasColumnName("user1_id");
            entity.Property(e => e.User2Id).HasColumnName("user2_id");
            entity.HasOne(d => d.User1).WithMany(p => p.PrivateChatUser1s)
                .HasForeignKey(d => d.User1Id)
                .HasConstraintName("private_chats_user1_id_fkey");
            entity.HasOne(d => d.User2).WithMany(p => p.PrivateChatUser2s)
                .HasForeignKey(d => d.User2Id)
                .HasConstraintName("private_chats_user2_id_fkey");});
        modelBuilder.Entity<Setting>(entity =>{
            entity.HasKey(e => e.UserId).HasName("settings_pkey");
            entity.ToTable("settings");
            entity.Property(e => e.UserId).ValueGeneratedNever().HasColumnName("user_id");
            entity.Property(e => e.Theme).HasMaxLength(20)
                .HasDefaultValueSql("'white'::character varying").HasColumnName("theme");
            entity.HasOne(d => d.User).WithOne(p => p.Setting)
                .HasForeignKey<Setting>(d => d.UserId)
                .HasConstraintName("settings_user_id_fkey");});
        modelBuilder.Entity<User>(entity =>{
            entity.HasKey(e => e.Id).HasName("users_pkey");
            entity.ToTable("users");
            entity.HasIndex(e => e.Email, "users_email_key").IsUnique();
            entity.HasIndex(e => e.UserName, "users_user_name_key").IsUnique();
            entity.Property(e => e.Id).HasColumnName("id");
            entity.Property(e => e.About).HasColumnName("about");
            entity.Property(e => e.DisplayName).HasMaxLength(100)
                .HasColumnName("display_name");
            entity.Property(e => e.Email).HasMaxLength(255).HasColumnName("email");
            entity.Property(e => e.IsVerified).HasDefaultValue(false).HasColumnName("is_verified");
            entity.Property(e => e.JoinDate).HasDefaultValueSql("CURRENT_TIMESTAMP")
                .HasColumnName("join_date");
            entity.Property(e => e.Password).HasColumnName("password");
            entity.Property(e => e.UserName).HasMaxLength(50)
                .HasColumnName("user_name");});
        OnModelCreatingPartial(modelBuilder); }
    partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
}
```

ДОДАТОК Д

Моделі даних

Сторінок 2

Київ-2025

Модель реєстрації:

```
public class RegisterModel {
    [Required(ErrorMessage = "Username is required.")]
    [StringLength(50, MinimumLength = 3)]
    public string Username { get; set; } = string.Empty;

    [Required(ErrorMessage = "Email is required.")]
    [StrictEmail(ErrorMessage = "Email is not valid.")]
    public string Email { get; set; } = string.Empty;

    [Required(ErrorMessage = "Password is required.")]
    [StringLength(100, MinimumLength = 6)]
    public string Password { get; set; } = string.Empty;

    [Required(ErrorMessage = "Please confirm your password.")]
    [Compare(nameof(Password), ErrorMessage = "Passwords do not match.")]
    public string ConfirmPassword { get; set; } = string.Empty;
}

public class StrictEmailAttribute : ValidationAttribute {
    public override bool IsValid(object? value){
        if (value is not string email) return false;
        var pattern = @"^[^@\s]+@[^@\s]+\.[^@\s]{2,}$";
        return Regex.IsMatch(email, pattern);
    }
}
```

Модель авторизації:

```
public class LoginModel {
    [Required(ErrorMessage = "Email is required.")]
    public string Email { get; set; } = string.Empty;

    [Required(ErrorMessage = "Password is required.")]
    public string Password { get; set; } = string.Empty;
}
```

Модель приватних чатів:

```
public class PrivateChatModel {
    public int ChatId { get; set; }
    public string ChatName { get; set; } = string.Empty;
    public int UserId { get; set; }
}
```

Модель групових чатів:

```
public class GroupChatModel {
    public int ChatId { get; set; }
    public string ChatName { get; set; } = string.Empty;
```

```
    public string Role { get; set; } = string.Empty;}
```


Модель останнього обраного чату:

```
public class LastOpenedChat {
    public int ChatId { get; set; }
    public string ChatType { get; set; } = string.Empty;
}
```

Модель учасника групового чату:

```
public class MemberModel {
    public int UserId { get; set; }
    public string? DisplayName { get; set; }
    public string? Role { get; set; }
}
```

Модель повідомлення:

```
public class MessageModel {
    public int MessageId { get; set; }
    public int ChatId { get; set; }
    public int SenderId { get; set; }
    public string? SenderDisplayName { get; set; }
    public string Text { get; set; } = string.Empty;
    public DateTime SentAt { get; set; }
    public string? ChatType { get; set; }
}
```

Модель деталей користувача:

```
public class UserDetailsDto {
    public string DisplayName { get; set; } = "";
    public string UserName { get; set; } = "";
    public string About { get; set; } = "";
    public DateTime JoinDate { get; set; }
}
```

ДОДАТОК Е

Код сторінок та взаємодія з інтерфейсом

Сторінок 10

Київ-2025

1. Сторінка реєстрації

```
@page "/register"
@rendermode InteractiveServer
@inject NavigationManager Navigation
@inject MyDbContext DbContext
@inject AuthenticationStateProvider AuthProvider
@inject ProtectedLocalStorage LocalStorage
@using System.ComponentModel.DataAnnotations
@using DiplomaProgram.Models

<EditForm Model="registerModel" OnValidSubmit="HandleValidSubmit">
    <DataAnnotationsValidator />
    <label>Username</label>
    <InputText @bind-Value="registerModel.Username" class="form-control" />
    <ValidationMessage For="@(() => registerModel.Username)" />
    <label>Email</label>
    <InputText @bind-Value="registerModel.Email" class="form-control" />
    <ValidationMessage For="@(() => registerModel.Email)" />
    <label>Password</label>
    <InputText @bind-Value="registerModel.Password" type="password" class="form-control" />
    <ValidationMessage For="@(() => registerModel.Password)" />
    <label>Confirm Password</label>
    <InputText @bind-Value="registerModel.ConfirmPassword" type="password" class="form-
control" />
    <ValidationMessage For="@(() => registerModel.ConfirmPassword)" />
    @if (!string.IsNullOrEmpty(errorMessage)){
        <div class="text-danger">@errorMessage</div>
    }
    <button type="submit">Sign Up</button>
</EditForm>

@code {
    private RegisterModel registerModel = new RegisterModel();
    private string? errorMessage;
    private bool isRedirecting = false;
    private bool isAuthorized = false;
    private bool hasCleared = false;

    protected override async Task OnInitializedAsync() {
        var authState = await AuthProvider.GetAuthenticationStateAsync();
        var user = authState.User;
        if (user.Identity?.IsAuthenticated == true && !isRedirecting) {
            isAuthorized = true;
            isRedirecting = true;
            await Task.Yield();
            Navigation.NavigateTo("/", forceLoad: true);}
    }

    protected override async Task OnAfterRenderAsync(bool firstRender){
```

```
        if (firstRender){
            if (!isAuthorized && !hasCleared){
                await LocalStorage.DeleteAsync("lastChat");
                hasCleared = true;}
        }
    }

    private async Task HandleValidSubmit(){
        if (registerModel.Password != registerModel.ConfirmPassword){
            errorMessage = "Passwords do not match.";
            return;}
        bool emailExists = await DbContext.Users.AnyAsync(u => u.Email == registerModel.Email);
        bool usernameExists = await DbContext.Users.AnyAsync(u => u.UserName ==
registerModel.Username);
        if (emailExists){
            errorMessage = "This email is already registered.";
            return;}
        if (usernameExists){
            errorMessage = "This username is already taken.";
            return;}
        var newUser = new User{
                Email = registerModel.Email,
                UserName = registerModel.Username,
                Password = registerModel.Password,
                DisplayName = registerModel.Username,
                About = "",
                JoinDate = DateTime.UtcNow,
                IsVerified = false};
        newUser.Setting = new Setting { Theme = "white" };
        DbContext.Users.Add(newUser);
        try{
            await DbContext.SaveChangesAsync();}
        catch (Exception){
            errorMessage = "Registration failed. Please try again later.";}
        errorMessage = "";
        Navigation.NavigateTo("/");
    }
}
```

2. Сторінка авторизації

```
@page "/login"
@inject NavigationManager Navigation
@inject MyDbContext DbContext
@inject AuthenticationStateProvider AuthProvider
@inject ProtectedLocalStorage LocalStorage
@using System.ComponentModel.DataAnnotations
@using System.Security.Claims
@using DiplomaProgram.Models
@using Microsoft.AspNetCore.Authentication
```

```
@using Microsoft.AspNetCore.Authentication.Cookies
```



```
<EditForm Model="loginModel" OnValidSubmit="HandleValidSubmit">
    <DataAnnotationsValidator />
    <label>Email</label>
    <InputText @bind-Value="loginModel.Email" class="form-control" />
    <ValidationMessage For="@(() => loginModel.Email)" />
    <label>Password</label>
    <InputText @bind-Value="loginModel.Password" type="password"/>
    <ValidationMessage For="@(() => loginModel.Password)" />
    @if (!string.IsNullOrEmpty(errorMessage)){
        <div class="text-danger">@errorMessage</div>
    }
    <button type="submit">Sign In</button>
</EditForm>

@code {
    [CascadingParameter]
    public HttpContext? HttpContext { get; set; }
    [SupplyParameterFromForm]
    private LoginModel loginModel { get; set; } = new();
    private string? errorMessage;
    private bool isRedirecting = false;
    private bool isAuthorized = false;
    private bool hasCleared = false;

    protected override async Task OnInitializedAsync(){
        var authState = await AuthProvider.GetAuthenticationStateAsync();
        var user = authState.User;
        if (user.Identity?.IsAuthenticated == true && !isRedirecting){
            isAuthorized = true;
            isRedirecting = true;
            await Task.Yield();
            Navigation.NavigateTo("/");}
    }

    protected override async Task OnAfterRenderAsync(bool firstRender){
        if (firstRender){
            if (!isAuthorized && !hasCleared){
                await LocalStorage.DeleteAsync("lastChat");
                hasCleared = true;}}
    }

    private async Task HandleValidSubmit(){
        var userAccount = DbContext.Users.Where(x => x.Email ==
loginModel.Email).FirstOrDefault();
        if (userAccount is null || userAccount.Password != loginModel.Password){
            errorMessage = "Invalid email or password.";
            return;}
        var claims = new List<Claim>{
            new Claim(ClaimTypes.NameIdentifier, userAccount.Id.ToString())};
```

```
   var identity = new ClaimsIdentity(claims, CookieAuthenticationDefaults.AuthenticationScheme);
   var principal = new ClaimsPrincipal(identity);
      await HttpContext.SignInAsync(principal);
      Navigation.NavigateTo("/");}
}
```

3. Сторінка виходу з аккаунту

```
@page "/logout"
@inject AuthenticationStateProvider AuthProvider
@inject NavigationManager Navigation
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Authentication

@code {
        [CascadingParameter]
        public HttpContext HttpContext { get; set; }
        protected override async Task OnInitializedAsync(){
                await base.OnInitializedAsync();
                if (HttpContext.User.Identity.IsAuthenticated){
                        HttpContext.Response.Cookies.Delete("auth_token");
                        await HttpContext.SignOutAsync();}
                Navigation.NavigateTo("/login", forceLoad: true);
        }
}
```

4. Сторінка верифікації пошти

```
@page "/verify-email"
@rendermode InteractiveServer
@using DiplomaProgram.Services
@inject MyDbContext DbContext
@inject NavigationManager Nav
@inject UserStateService UserState

@if (!string.IsNullOrEmpty(Message)){
   <div class="@(IsSuccess ? "text-success" : "text-danger")">@Message</div>
}
else{ <div>Verifying your email, please wait...</div> }

@code {
   [Parameter]
   [SupplyParameterFromQuery(Name = "token")]
   public string? Token { get; set; }
   private string Message = string.Empty;
   private bool IsSuccess = false;
   private bool isInitialized = false;
```

```
    protected override async Task OnAfterRenderAsync(bool firstRender){
        if (firstRender && !isInitialized){
```



```
            isInitialized = true;
            if (string.IsNullOrWhiteSpace(Token)){
                Message = "Verification token is missing.";
                IsSuccess = false;
                StateHasChanged();
                return;}
            var record = await DbContext.EmailVerifications
                .Include(v => v.User)
                .FirstOrDefaultAsync(v => v.Token == Token && !v.IsUsed);
            if (record == null){
                Message = "This verification link is invalid or already used.";
                IsSuccess = false;
                StateHasChanged();
                return;}
            if (record.ExpiresAt < DateTime.UtcNow){
                Message = "This verification link has expired.";
                IsSuccess = false;
                StateHasChanged();
                return;}
            record.IsUsed = true;
            record.User.IsVerified = true;
            await DbContext.SaveChangesAsync();
            UserState.SetIsVerified(true);
            Message = "Your email has been successfully verified!";
            IsSuccess = true;
            StateHasChanged();
        }
    }
}
```

5. Сторінка чатів

```
@page "/"
@attribute [Authorize]
@inject MyDbContext DbContext
@inject AuthenticationStateProvider AuthProvider
@inject NavigationManager Navigation
@inject UserStateService UserState
@inject ProtectedLocalStorage LocalStorage

<PageTitle>Chats</PageTitle>

<section class="main-section">
    <section class="left-section">
        <div><h2>Private chats</h2>
            <button @onclick="ToggleSearchDialog">Add</button>
            @if (privateChats.Any()){<ul>
                    @foreach (var chat in privateChats){
```

```
                <li @onclick="@(() => OpenPrivateChat((chat.ChatId, chat.UserId,
chat.ChatName)))">@chat.ChatName
</li> }</ul>}</div>
```



```
      <div><h2>Group chats</h2>
         <button @onclick="AddGroupChat">Add</button>
         @if (groupChats.Any()){<ul>
               @foreach (var chat in groupChats){
                  <li @onclick="@(() => OpenGroupChat((chat.ChatId, chat.ChatName)))">
                     @chat.ChatName
</li>}</ul>}</div></section>

   <section class="center-section">
      @if (isChatSelected){
         <div>
            <button @onclick="() => OpenUserProfileMW(currentOtherUserId)">Profile</button>
            <h2>@currentChatName</h2>
           <button @onclick="ConfirmChatDelete">Delete</button></div>
         <ChatMessages ChatId="@currentChatId"
         CurrentUserId="@(currentUser?.Id ?? 0)" ChatType="@currentChatType" />
         <MessageBox CurrentUserId="@(currentUser?.Id ?? 0)"
         CurrentUserDisplayName="@(currentUser?.DisplayName ?? "0")"
         CurrentChatId="@currentChatId" CurrentChatType="@currentChatType"
         OnChatMovedToTop="HandleMoveChatToTop" />}
   </section>

   <section class="right-section">
      @if (isChatSelected){
         @if (currentChatType == "private"){
            <PrivateChatDetails userId="@currentOtherUserId" />}
         else{
            <GroupChatDetails ChatId="@currentChatId"
           ChatName="@currentChatName" RemoveGroupChat="RemoveGroupChat"
           CurrentChatType="@currentChatType" CurrentUserId="@(currentUser?.Id ?? 0)"
           NewChatConnection="@newChatConnection" />}}
      <div>
         <button @onclick="() => OpenUserProfileMW(currentUser?.Id ?? 1)">Profile</button>
         <span>@currentUser?.DisplayName</span>
         <button @onclick="ToggleSettingsMW">Settings</button>
</div></section></section>

<AddChatMW IsOpen="isSearchDialogOpen" OnClose="ToggleSearchDialog"
        OpenPrivateChat="OpenPrivateChat"
        NewChatConnection="newChatConnection"
        CurrentUserDisplayName="@currentUser?.DisplayName" />
<UserProfileMW IsOpen="isUserProfileOpen" OnClose="CloseUserProfileMW"
userId="idToSearch" />
<SettingsMW IsOpen="isSettingsOpen" OnClose="ToggleSettingsMW" userId="currentUser?.Id
?? 1" />
<ConfirmMW @ref="deleteChatDialog" ConfirmationTitle="Chat deletion"
        ConfirmationMessage="Are you sure you want to delete this chat?"
```

```
        OnConfirm="DeletePrivateChat" OnCancel="CancelChatDelete" />
```



```
@code {
    [Inject] IJSRuntime JS { get; set; }
    private User? currentUser;
    private HubConnection? newChatConnection;
    private bool isSearchDialogOpen = false;
    private bool isUserProfileOpen = false;
    private bool isSettingsOpen = false;
    private ConfirmMW? deleteChatDialog;
    private string searchText = string.Empty;
    private int idToSearch = 0;
    private List<PrivateChatModel> privateChats = new();
    private List<GroupChatModel> groupChats = new();
    private bool isChatSelected = false;
    private string currentChatType = "private";
    private string currentChatName = "";
    private int currentChatId = 0;
    private int currentOtherUserId = 0;

    protected override async Task OnInitializedAsync(){
        var authState = await AuthProvider.GetAuthenticationStateAsync();
        var user = authState.User;
        if (user.Identity?.IsAuthenticated == true){
            var userId = Convert.ToInt32(user.FindFirst(ClaimTypes.NameIdentifier)?.Value);
            currentUser = await DbContext.Users.FirstOrDefaultAsync(x => x.Id == userId);
            if (currentUser != null){
                privateChats = await DbContext.PrivateChats
                    .Where(c => c.User1Id == currentUser.Id || c.User2Id == currentUser.Id)
                    .Include(c => c.User1).Include(c => c.User2)
                    .OrderByDescending(c => c.LastMessageDate)
                    .Select(c => new PrivateChatModel{
                            ChatId = c.Id,
                            ChatName = c.User1Id == currentUser.Id ? c.User2.DisplayName :
c.User1.DisplayName,
                            UserId = c.User1Id == currentUser.Id ? c.User2.Id : c.User1.Id
                        }).ToListAsync();
                groupChats = await DbContext.Members
                    .Where(m => m.UserId == currentUser.Id)
                    .Include(m => m.Group)
                    .OrderByDescending(m => m.Group.LastMessageDate)
                    .Select(m => new GroupChatModel{
                        ChatId = m.Group.Id,
                        ChatName = m.Group.Name,
                        Role = m.Role
                    }).ToListAsync();}
            newChatConnection = new HubConnectionBuilder()
                .WithUrl(Navigation.ToAbsoluteUri("/newchathub"))
```

```
            .WithAutomaticReconnect().Build();
        newChatConnection.On<PrivateChatModel>("ReceivePrivateChat", (chat) =>{
            AddPrivateChat(chat);
            InvokeAsync(StateHasChanged);});
```



```
newChatConnection.On<int>("RemovePrivateChat", (chatId) =>{
            privateChats.RemoveAll(x => x.ChatId == chatId);
            if(currentChatType == "private" && currentChatId == chatId){
                isChatSelected = false;}
            InvokeAsync(StateHasChanged);});
        newChatConnection.On<GroupChatModel>("ReceiveGroupChat", (chat) =>{
            if (!groupChats.Any(u => u.ChatId == chat.ChatId)){
                groupChats.Insert(0, chat);
                InvokeAsync(StateHasChanged);}});
        newChatConnection.On<int, string>("ChangeGroupChatName", (chatId, chatName) =>{
            var chat = groupChats.FirstOrDefault(c => c.ChatId == chatId);
            if (chat != null){
                chat.ChatName = chatName;
                if(currentChatId == chatId && currentChatType == "group"){
                    currentChatName = chatName;}
              InvokeAsync(StateHasChanged);}});
        await newChatConnection.StartAsync();
        await newChatConnection.SendAsync("JoinGroup", userId);}
 }

 protected override async Task OnAfterRenderAsync(bool firstRender){
    if (firstRender){
        await JS.InvokeVoidAsync("resizeTextarea", "msgTextarea");
        await JS.InvokeVoidAsync("focusElement", "msgTextarea");
        await TryOpenLastChatAsync();
        StateHasChanged();}
 }

 private void ConfirmChatDelete(){ deleteChatDialog?.Show(); }
 private Task CancelChatDelete(){ return Task.CompletedTask; }

 private async Task OpenPrivateChat((int chatId, int userId, string chatName) chatInfo){
    currentChatType = "private";
    (currentChatId, currentOtherUserId, currentChatName) = chatInfo;
    isChatSelected = true;
    await JS.InvokeVoidAsync("focusElement", "msgTextarea");
    await LocalStorage.SetAsync("lastChat", new LastOpenedChat{
            ChatId = currentChatId,
            ChatType = currentChatType});
 }

 private async Task OpenGroupChat((int chatId, string chatName) chatInfo){
    currentChatType = "group";
    (currentChatId, currentChatName) = chatInfo;
    isChatSelected = true;
    await JS.InvokeVoidAsync("focusElement", "msgTextarea");
```

```
    await LocalStorage.SetAsync("lastChat", new LastOpenedChat{
            ChatId = currentChatId,
            ChatType = currentChatType});
 }
```



```
private void HandleMoveChatToTop((int ChatId, string ChatType) chat){
    if (chat.ChatType == "private"){
       if (privateChats.Count > 0 && privateChats[0].ChatId == chat.ChatId)
          return;
    var chatToMove = privateChats.FirstOrDefault(c => c.ChatId == chat.ChatId);
       if (chatToMove != null){
          privateChats.Remove(chatToMove);
          privateChats.Insert(0, chatToMove);
          StateHasChanged();}}
    else{
       if (groupChats.Count > 0 && groupChats[0].ChatId == chat.ChatId)
          return;
       var chatToMove = groupChats.FirstOrDefault(c => c.ChatId == chat.ChatId);
       if (chatToMove != null){
          groupChats.Remove(chatToMove);
          groupChats.Insert(0, chatToMove);
          StateHasChanged();}}
 }

 private async Task TryOpenLastChatAsync(){
    var result = await LocalStorage.GetAsync<LastOpenedChat>("lastChat");
    if (result.Success && result.Value.ChatType == "private"){
       var chat = privateChats.FirstOrDefault(c => c.ChatId == result.Value.ChatId);
       if (chat != null){
          await OpenPrivateChat((chat.ChatId, chat.UserId, chat.ChatName));
          return;}}
    else if (result.Success && result.Value.ChatType == "group"){
       var chat = groupChats.FirstOrDefault(c => c.ChatId == result.Value.ChatId);
       if (chat != null){
          await OpenGroupChat((chat.ChatId, chat.ChatName));
          return;}}
    var firstChat = privateChats.FirstOrDefault();
    if (firstChat != null){
      await OpenPrivateChat((firstChat.ChatId, firstChat.UserId, firstChat.ChatName));}
 }

 private void ToggleSearchDialog(){isSearchDialogOpen = !isSearchDialogOpen;}

 private void AddPrivateChat(PrivateChatModel chat){
    if (!privateChats.Any(u => u.ChatId == chat.ChatId)){
       privateChats.Insert(0, chat);}}

 private async Task DeletePrivateChat(){
    var chat = await DbContext.PrivateChats
       .FirstOrDefaultAsync(c => c.Id == currentChatId);
    if (chat != null){
```

```
            DbContext.PrivateChats.Remove(chat);
            await DbContext.SaveChangesAsync();}
        await newChatConnection.SendAsync("DeletePrivateChat", currentUser.Id, currentChatId);
```



```
      await newChatConnection.SendAsync("DeletePrivateChat", currentOtherUserId,
currentChatId);
    }

    private async Task AddGroupChat(){
            var newChat = new GroupChat{
            Name = "New chat", LastMessageDate = DateTime.UtcNow};
        DbContext.GroupChats.Add(newChat);
        await DbContext.SaveChangesAsync();
        var newMember = new Member{GroupId = newChat.Id,
            UserId = currentUser.Id, Role = "admin" };
        DbContext.Members.Add(newMember);
        await DbContext.SaveChangesAsync();
        GroupChatModel model = new GroupChatModel{ChatId = newChat.Id,
            ChatName = newChat.Name, Role = newMember.Role};
        groupChats.Insert(0, model);
        await OpenGroupChat((model.ChatId, model.ChatName));}
    private void RemoveGroupChat(int id){
        groupChats.RemoveAll(x => x.ChatId == id);
        isChatSelected = false;}
    private void OpenUserProfileMW(int id){
        idToSearch = id;
        isUserProfileOpen = true;}
    private void CloseUserProfileMW(){ isUserProfileOpen = false; }
    private void ToggleSettingsMW(){ isSettingsOpen = !isSettingsOpen;}
    public async ValueTask DisposeAsync(){
        if (newChatConnection is not null){
            await newChatConnection.DisposeAsync();}}
}
```

ДОДАТОК Ж

Модальні вікна

Сторінок 14

Київ-2025

1. Модальне вікно додавання чатів

```
@using System.Security.Claims
@using DiplomaProgram.Hubs
@using DiplomaProgram.Models
@using DiplomaProgram.Services
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.SignalR
@using Microsoft.AspNetCore.SignalR.Client
@inject IServiceProvider serviceProvider

@if (IsOpen){
<div class="modal-background">
<div class="my-dialog">
        <h3>Add chat</h3>
        <input @ref="searchInputRef" id="chat-user-search" @bind="SearchText"
@bind:event="oninput" placeholder="@@Username" />
        @if (FoundUsers.Any()){
        <h5>Search result:</h5>
                <ul class="list-group">
                        @foreach (var user in FoundUsers){
                        <li class="search-element" @onclick="() => OpenUserProfileMW(user.Id)">
                        <div class="icon-div">
                        <UserIcon />
                        @user.DisplayName (@("@" + user.UserName))
                        </div>
                        <span @onclick:stopPropagation>
                        <Tooltip Title="Add this user" role="button" @onclick="() =>
AddPrivateChat(user)" Class="tooltip-general">
                        <AddIcon /></Tooltip></span></li>}</ul>}
                else if (!string.IsNullOrEmpty(SearchText)){
                <p class="mt-2">No results.</p>}
        <button class="close-button" @onclick="Close">Close</button>
</div></div>}

<UserProfileMW IsOpen="isUserProfileOpen" OnClose="CloseUserProfileMW"
userId="idToSearch" />
<UserProfileMW IsOpen="isUserProfileOpen" OnClose="CloseUserProfileMW"
userId="idToSearch" />

@code {
   [Parameter] public bool IsOpen { get; set; }
   [Parameter] public EventCallback OnClose { get; set; }
   [Parameter] public EventCallback<(int, int, string)> OpenPrivateChat { get; set; }
   [Parameter] public HubConnection? NewChatConnection { get; set; }
   [Parameter] public string? CurrentUserDisplayName { get; set; }
   [Inject] private AuthenticationStateProvider AuthProvider { get; set; }
   [Inject] IJSRuntime? JS { get; set; }
   [Inject] public MyDbContext DbContext { get; set; } = default!;
   private ElementReference searchInputRef;
```

```
private string SearchText { get; set; } = string.Empty;
 private string lastSearchText = string.Empty;
 private List<User> FoundUsers { get; set; } = new();
 private System.Threading.Timer? searchTimer;
 private bool isUserProfileOpen = false;
 private int idToSearch = 0;
 private int currentUserId;

 protected override async Task OnAfterRenderAsync(bool firstRender){
    if (IsOpen){
       await JS.InvokeVoidAsync("focusElement", searchInputRef);
       var authState = await AuthProvider.GetAuthenticationStateAsync();
       var user = authState.User;
       if (user.Identity?.IsAuthenticated == true){
          currentUserId = Convert.ToInt32(user.FindFirst(ClaimTypes.NameIdentifier)?.Value);}}
 }

 protected override void OnParametersSet(){
    if (IsOpen && searchTimer == null){
       searchTimer = new System.Threading.Timer(async _ =>{
          if (!string.IsNullOrWhiteSpace(SearchText)){
             await InvokeAsync(SearchUsers); }
       }, null, 0, 1000);}
    else if (!IsOpen){DisposeSearch();}
 }

 private void DisposeSearch(){
    searchTimer?.Dispose();
    searchTimer = null;
    SearchText = string.Empty;
    lastSearchText = string.Empty;
    FoundUsers.Clear();
 }

 private async Task AddPrivateChat(User selectedUser){
    var existingChat = await DbContext.PrivateChats
       .Where(c =>
          (c.User1Id == currentUserId && c.User2Id == selectedUser.Id) ||
          (c.User1Id == selectedUser.Id && c.User2Id == currentUserId))
       .FirstOrDefaultAsync();

    if (existingChat == null){
       var newChat = new PrivateChat{
             User1Id = currentUserId,
             User2Id = selectedUser.Id,
             LastMessageDate = DateTime.UtcNow};
       DbContext.PrivateChats.Add(newChat);
       await DbContext.SaveChangesAsync();
       var currentChatModel = new PrivateChatModel{
             ChatId = newChat.Id,
```

```
                ChatName = selectedUser.DisplayName,
                UserId = selectedUser.Id};
            var otherChatModel = new PrivateChatModel{
                ChatId = newChat.Id,
                ChatName = CurrentUserDisplayName ?? "Unknown chat",
                UserId = currentUserId};
            if (NewChatConnection is not null){
                await NewChatConnection.SendAsync("SendPrivateChat", currentUserId,
currentChatModel);
                await NewChatConnection.SendAsync("SendPrivateChat", selectedUser.Id,
otherChatModel);
                await OpenPrivateChat.InvokeAsync((newChat.Id, selectedUser.Id,
selectedUser.DisplayName));}}
        await OnClose.InvokeAsync();
        DisposeSearch();
    }

    private async Task SearchUsers(){
        if (SearchText.Trim() == lastSearchText.Trim()) return;
        lastSearchText = SearchText.Trim();
        var existingChatUserIds = await DbContext.PrivateChats
            .Where(c => c.User1Id == currentUserId || c.User2Id == currentUserId)
            .Select(c => c.User1Id == currentUserId ? c.User2Id : c.User1Id)
            .ToListAsync();
        FoundUsers = await DbContext.Users
            .Where(u =>
                u.UserName.ToLower().Contains(SearchText.ToLower()) &&
                u.Id != currentUserId &&
                !existingChatUserIds.Contains(u.Id)).Select(u => new User{
                    Id = u.Id,
                    UserName = u.UserName,
                    DisplayName = u.DisplayName}).Take(10)
            .ToListAsync();
        StateHasChanged();
    }

    private void OpenUserProfileMW(int id){
        idToSearch = id;
        isUserProfileOpen = true;
    }

    private void CloseUserProfileMW(){isUserProfileOpen = false;}

    private void Close(){
        OnClose.InvokeAsync();
        DisposeSearch();
    }
}
```

2. Модальне вікно додавання учасників

```
@if (IsOpen){
   <div class="modal-background"><div class="my-dialog"><h3>Add chat</h3>
         <input @ref="searchInputRef" type="text" autocomplete="off" @bind="SearchText"
@bind:event="oninput" placeholder="@@Username" />
         @if (FoundUsers.Any()){<h5>Search result:</h5>
            <ul class="list-group">
               @foreach (var user in FoundUsers){
                  <li class="search-element" @onclick="() => OpenUserProfileMW(user.Id)">
                     <div class="icon-div"><svg class="user-icon"></svg>
                        @user.DisplayName (@("@" + user.UserName))
                     </div>
                     <span @onclick:stopPropagation>
                        <Tooltip Title="Add this user" @onclick="() => AddMember(user)"
Class="tooltip-general"><svg class="add-icon-search"></svg></Tooltip></span></li>}</ul>}
         else if (!string.IsNullOrWhiteSpace(SearchText)){
            <p class="mt-2">No results.</p>}

         <button class="close-button" @onclick="Close">Close</button>
</div></div>
}

@code {
   [Parameter] public bool IsOpen { get; set; }
   [Parameter] public EventCallback OnClose { get; set; }
   [Parameter] public HubConnection? NewMemberConnection { get; set; }
   [Parameter] public HubConnection? NewChatConnection { get; set; }
   [Parameter] public int CurrentChatId { get; set; }
   [Parameter] public string? CurrentChatName { get; set; }
   [Inject] private AuthenticationStateProvider AuthProvider { get; set; }
   [Inject] IJSRuntime? JS { get; set; }
   [Inject] public MyDbContext DbContext { get; set; } = default!;
   [Inject] public IDbContextFactory<MyDbContext> DbFactory { get; set; } = default!;
   private ElementReference searchInputRef;
   private string SearchText { get; set; } = string.Empty;
   private string lastSearchText = string.Empty;
   private List<User> FoundUsers { get; set; } = new();
   private System.Threading.Timer? searchTimer;
   private bool isUserProfileOpen = false;
   private int idToSearch = 0;
   private int currentUserId;

   protected override async Task OnAfterRenderAsync(bool firstRender){
      if (IsOpen){
         await JS.InvokeVoidAsync("focusElement", searchInputRef);
         var authState = await AuthProvider.GetAuthenticationStateAsync();
         var user = authState.User;
         if (user.Identity?.IsAuthenticated == true){
            currentUserId = Convert.ToInt32(user.FindFirst(ClaimTypes.NameIdentifier)?.Value);}}}
```

```
    protected override void OnParametersSet(){
        if (IsOpen && searchTimer == null){
            searchTimer = new System.Threading.Timer(async _ =>{
                if (!string.IsNullOrWhiteSpace(SearchText)){await InvokeAsync(SearchUsers);}
            }, null, 0, 1000);}
        else if (!IsOpen){DisposeSearch();}
    }

    private void DisposeSearch(){
        searchTimer?.Dispose();
        searchTimer = null;
        SearchText = string.Empty;
        lastSearchText = string.Empty;
        FoundUsers.Clear();
    }

    private async Task AddMember(User selectedUser){
        await using var context = await DbFactory.CreateDbContextAsync();
        var newMember = new Member{
                UserId = selectedUser.Id,
                GroupId = CurrentChatId,
                Role = "admin"};
        var exists = await context.Members
            .AnyAsync(m => m.GroupId == CurrentChatId && m.UserId == selectedUser.Id);
        if (!exists){
            context.Members.Add(newMember);
            await context.SaveChangesAsync();}
        var memberModel = new MemberModel{
                UserId = selectedUser.Id,
                DisplayName = selectedUser.DisplayName,
                Role = "admin"};
        if (NewMemberConnection is not null){
            await NewMemberConnection.SendAsync("SendMember", CurrentChatId,
memberModel);}
        if (NewChatConnection is not null){
            await NewChatConnection.SendAsync("SendGroupChat", selectedUser.Id, new
GroupChatModel{
                    ChatId = CurrentChatId,
                    ChatName = CurrentChatName});}
        await OnClose.InvokeAsync();
        DisposeSearch();
    }

    private async Task SearchUsers(){
        if (SearchText.Trim() == lastSearchText.Trim()) return;
        lastSearchText = SearchText.Trim();
        await using var context = await DbFactory.CreateDbContextAsync();
        var existingMemberIds = await context.Members
            .Where(m => m.GroupId == CurrentChatId)
            .Select(m => m.UserId).ToListAsync();
```

```
            FoundUsers = await context.Users
            .Where(u =>
               u.UserName.ToLower().Contains(SearchText.ToLower()) &&
               u.Id != currentUserId &&
               !existingMemberIds.Contains(u.Id))
            .Select(u => new User{
                  Id = u.Id,
                  UserName = u.UserName,
                  DisplayName = u.DisplayName}).Take(10)
            .ToListAsync();
         StateHasChanged();
      }

      private void OpenUserProfileMW(int id){
         idToSearch = id;
         isUserProfileOpen = true;
      }

      private void CloseUserProfileMW(){isUserProfileOpen = false;}

      private void Close(){
         OnClose.InvokeAsync();
         DisposeSearch();
      }
}
```

3. Модальне вікно діалогу підтвердження дії

```
@if (IsVisible){
   <div class="modal-backdrop"><div class="modal-window">
         <h4 class="header-text">@ConfirmationTitle</h4>
         <div class="modal-body">@ConfirmationMessage</div>
         <div class="modal-footer">
            <button @onclick="OnConfirmClicked" class="confirm-button">Confirm</button>
            <button @onclick="OnCancelClicked" class="cancel-button">Cancel</button>
</div></div></div>}

@code {
   [Parameter] public string ConfirmationTitle { get; set; } = "Confirmation";
   [Parameter] public string ConfirmationMessage { get; set; } = "Are you sure?";
   [Parameter] public EventCallback OnConfirm { get; set; }
   [Parameter] public EventCallback OnCancel { get; set; }
   private bool IsVisible { get; set; } = false;
   public void Show() => IsVisible = true;
   public void Hide() => IsVisible = false;

   private async Task OnConfirmClicked(){
      await OnConfirm.InvokeAsync();
      Hide();
   }
```

```
    private async Task OnCancelClicked(){
    await OnCancel.InvokeAsync();
    Hide();
  }
}
```

4. Модальне вікно профілю користувача

```
@using DiplomaProgram.Models
@inject IDbContextFactory<MyDbContext> DbFactory

@if (IsOpen){
  @if (userDetails is not null){
    <div class="modal-background"><div class="my-dialog"><h3>Profile</h3>
        <div class="icon-row"><div class="icon-div">
            <svg class="user-icon"></svg>
          </div>
          <div class="user-name-div">
            <h5 class="icon-text">@userDetails.DisplayName</h5>
            <h6 class="icon-text">@("@" + userDetails.UserName)</h6>
          </div>
        </div>
        <div class="content-row"><h5>About</h5>
          <div class="text-scroll-div">
            <div class="overflow-div scrollbar-clean">
              @(string.IsNullOrWhiteSpace(userDetails.About)
                ? "This user has not added a description."
                : userDetails.About)
        </div></div></div>
        <div class="content-row"><h5>Join date</h5>
          <div class="text-div">@userDetails.JoinDate.ToString("dd-MM-yyyy")</div>
        </div>

        <button class="close-button" @onclick="Close">Close</button>
    </div></div>}
  else{<p>Loading user details...</p>}}

@code {
  [Parameter] public bool IsOpen { get; set; }
  [Parameter] public EventCallback OnClose { get; set; }
  [Parameter] public int userId { get; set; }
  private UserDetailsDto? userDetails;

  protected override async Task OnParametersSetAsync()
  {
    await using var context = await DbFactory.CreateDbContextAsync();
    userDetails = await context.Users
      .AsNoTracking()
      .Where(x => x.Id == userId)
      .Select(x => new UserDetailsDto {
```

```
            DisplayName = x.DisplayName,
                UserName = x.UserName,
                About = x.About,
                JoinDate = x.JoinDate
            }).FirstOrDefaultAsync();
    }

    private void Close() => OnClose.InvokeAsync();
}
```

5. Модальне вікно налаштувань користувача

```
@using System.Text.RegularExpressions;
@using System.Security.Cryptography
@using DiplomaProgram.Services
@using Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage
@inject NavigationManager Navigation
@inject UserStateService UserState
@inject ProtectedLocalStorage LocalStorage

@if (IsOpen){
    <div><div class="my-dialog"><h3>Settings</h3>
            <div class="icon-row"><div class="icon-div">
                <svg class="user-icon" viewBox="0 0 57 57" fill="none"
xmlns="http://www.w3.org/2000/svg"></svg>
              </div>
              <div>
                <h5 class="icon-text">@(userProfile?.DisplayName ?? string.Empty)</h5>
                <h6 class="icon-text">@("@" + userProfile?.UserName ?? string.Empty)</h6>
              </div>
              <div class="logout-div">
                <button class="logout-button" @onclick="Logout">Sign Out</button>
              </div>
            </div>
            <div>
              <div class="sub-container"><div class="content-row"><h5>Display name</h5>
                <div class="input-buttons">
                  <div class="input-and-message">
                    <input @ref="displaynameInputRef" @bind="displaynameInputValue"
disabled="@(!isDisplaynameEditable)" class="@(isDisplaynameEditable ? "focus" : "")"
type="text" autocomplete="off" placeholder="New display name..." />
                    @if(displaynameError != string.Empty){
                      <p class="error-message">@displaynameError</p>}
                  </div>
                  <div class="buttons">
                    @if (!isDisplaynameEditable){
                      <Tooltip Title="Edit display name" role="button"
@onclick="EnableDisplaynameEditing" Class="tooltip-general"></Tooltip>}
                    else{
```

```
    <Tooltip Title="Confirm" role="button" @onclick="ChangeDisplayname" Class="tooltip-
general"></Tooltip>
                    <Tooltip Title="Cancel" role="button"
@onclick="DisableDisplaynameEditing" Class="tooltip-general"></Tooltip>}
                </div>
            </div></div>
            <div class="content-row"><h5>About</h5>
                <div class="input-buttons">
                    <div class="input-and-message">
                        <textarea @ref="descriptionInputRef" @bind="descriptionInputValue"
                        disabled="@(!isDescriptionEditable)"
                        class="@(isDescriptionEditable ? "focus" : "")"
                        rows="1" @oninput="ResizeTextArea" id="aboutTextarea"
                        placeholder="Add description..." >
                        </textarea>
                        @if (descriptionError != string.Empty){
                            <p class="error-message">@descriptionError</p>}
                    </div>
                    <div class="buttons">
                        @if (!isDescriptionEditable){
                            <Tooltip Title="Edit description" role="button"
@onclick="EnableDescriptionEditing" Class="tooltip-general"></Tooltip>}
                        else{
                            <Tooltip Title="Confirm" role="button" @onclick="ChangeDescription"
Class="tooltip-general"></Tooltip>
                            <Tooltip Title="Cancel" role="button"
@onclick="DisableDescriptionEditing" Class="tooltip-general"></Tooltip>}
                    </div>
                </div>
            </div>
            <div class="content-row">
                <div class="input-header">
                    <h5>Email</h5>
                    @if(!userProfile?.IsVerified ?? false){
                        <button class="close-button regular-button"
@onclick="VerifyEmail">Verify</button>}
                    else{
                        <Tooltip Title="Email is verified" Class="tooltip-general" Style="margin-left:
3px;"></Tooltip>}
                </div>
                <div class="input-buttons">
                    <div class="input-and-message">
                        <input @ref="emailInputRef" @bind="emailInputValue"
disabled="@(!isEmailEditable)" class="@(isEmailEditable ? "focus" : "")" type="text"
autocomplete="off" placeholder="Error loading...">
                        @if (emailError != string.Empty){
                            <p class="error-message">@emailError</p>}
                        else if (emailMessage != string.Empty){
                            <p class="error-message input-message">@emailMessage</p>}
                    </div>
```

```
                <div class="buttons">
                        @if (!isEmailEditable){
                            <Tooltip Title="Edit display name" role="button"
@onclick="EnableEmailEditing" Class="tooltip-general"> </Tooltip>}
                        else {
                            <Tooltip Title="Confirm" role="button" @onclick="ChangeEmail"
Class="tooltip-general"> </Tooltip>
                            <Tooltip Title="Cancel" role="button" @onclick="DisableEmailEditing"
Class="tooltip-general"></Tooltip>}
                    </div></div></div></div>
            <div class="sub-container"><div class="content-row password-row">
                <h5>Change password</h5>
                <div class="input-buttons">
                    <div class="input-and-message">
                        <input @bind="currentPasswordInputValue" class="password-input"
type="password" autocomplete="off" placeholder="Current password" />
                        <input @bind="newPasswordInputValue" class="password-input"
type="password" autocomplete="off" placeholder="New password" />
                        <input @bind="confirmPasswordInputValue" class="password-input"
type="password" autocomplete="off" placeholder="Confirm password" />
                        @if (passwordError != string.Empty){
                            <p class="error-message">@passwordError</p>}
                        else if (passwordMessage != string.Empty){
                            <p class="error-message input-message">@passwordMessage</p>}

                        <button class="close-button change-password-button"
@onclick="ChangePassword">Change password</button>
                    </div></div></div></div></div>
          <button class="close-button" @onclick="Close">Close</button>
        </div></div>}

@code {
    [Parameter] public bool IsOpen { get; set; }
    [Parameter] public EventCallback OnClose { get; set; }
    [Parameter] public int userId { get; set; }
    [Inject] IJSRuntime JS { get; set; } = default!;
    [Inject] public MyDbContext DbContext { get; set; } = default!;
    [Inject] public EmailService EmailService { get; set; } = default!;
    private User? userProfile;
    private bool isEditing = false;
    private ElementReference displaynameInputRef, descriptionInputRef, emailInputRef;
    private string displaynameInputValue = "", displaynameError = "", descriptionInputValue = "",
descriptionError = "";
    private string emailInputValue = "", emailError = "", emailMessage = "";
    private string currentPasswordInputValue = "", newPasswordInputValue = "",
confirmPasswordInputValue = "", passwordError = "", passwordMessage = "";
    private bool isDisplaynameEditable, isDescriptionEditable, isEmailEditable;

    private async Task EnableDisplaynameEditing(){
        passwordMessage = string.Empty;
```

```
    if (!isEditing){
        emailMessage = string.Empty;
        isEditing = true;
        isDisplaynameEditable = true;
        await Task.Yield();
        await JS.InvokeVoidAsync("focusElement", displaynameInputRef);}
}

private async Task ChangeDisplayname(){
    passwordMessage = string.Empty;
    if (displaynameInputValue.Length < 3) {
        displaynameError = "Display name can't be shorter than 3 symbols";
        return;}
    if (displaynameInputValue.Length > 100){
        displaynameError = "Display name can't be longer than 100 symbols";
        return;}

    if (userProfile is not null){
        userProfile.DisplayName = displaynameInputValue;
        DbContext.Users.Update(userProfile);
        await DbContext.SaveChangesAsync();}
    isEditing = false;
    isDisplaynameEditable = false;
    displaynameInputValue = string.Empty;
    displaynameError = string.Empty;
}

private void DisableDisplaynameEditing(){
    passwordMessage = string.Empty;
    isEditing = false;
    isDisplaynameEditable = false;
    displaynameInputValue = string.Empty;
    displaynameError = string.Empty;
}

private async Task EnableDescriptionEditing(){
    passwordMessage = string.Empty;
    if (!isEditing){
        emailMessage = string.Empty;
        isEditing = true;
        isDescriptionEditable = true;
        await Task.Yield();
        await JS.InvokeVoidAsync("focusElement", descriptionInputRef);}
}

private async Task ChangeDescription(){
    passwordMessage = string.Empty;
    if (descriptionInputValue.Length > 1000){
        descriptionError = "Description can't be longer than 1000 symbols";
        return;}
```

```
  if (userProfile is not null){
      userProfile.About = descriptionInputValue;
      DbContext.Users.Update(userProfile);
      await DbContext.SaveChangesAsync();
      descriptionInputValue = userProfile?.About ?? string.Empty;}
   isEditing = false;
   isDescriptionEditable = false;
   descriptionError = string.Empty;
}

private void DisableDescriptionEditing(){
   passwordMessage = string.Empty;
   isEditing = false;
   isDescriptionEditable = false;
   descriptionError = string.Empty;
   descriptionInputValue = userProfile?.About ?? string.Empty;
}

private async Task EnableEmailEditing(){
   passwordMessage = string.Empty;
   if (!isEditing){
      isEditing = true;
      isEmailEditable = true;
      await Task.Yield();
      await JS.InvokeVoidAsync("focusElement", emailInputRef);}
}

private async Task ChangeEmail(){
   passwordMessage = string.Empty;
   if (!Regex.IsMatch(emailInputValue, @"^[^@\s]+@[^@\s]+\.[^@\s]{2,}$")){
      emailError = "Email is invalid";return;}
   bool emailExists = await DbContext.Users.AnyAsync(u => u.Email == emailInputValue);
   if(emailExists){
      emailError = "This email already exists";return;}
   if (userProfile is not null){
      userProfile.Email = emailInputValue;
      userProfile.IsVerified = false;
      DbContext.Users.Update(userProfile);
      await DbContext.SaveChangesAsync();
   }
   isEditing = false;
   isEmailEditable = false;
   emailError = string.Empty;
   emailMessage = string.Empty;
}

private void DisableEmailEditing(){
   passwordMessage = string.Empty;
   isEditing = false;
   isEmailEditable = false;
```

```
 emailInputValue = userProfile?.Email ?? string.Empty;
    emailError = string.Empty;
}

public async Task VerifyEmail(){
    passwordMessage = string.Empty;
    emailMessage = "Check your email to continue verification";
    var token = Convert.ToBase64String(RandomNumberGenerator.GetBytes(32));
    var verification = new EmailVerification{
            UserId = userProfile.Id,
            Token = token,
            ExpiresAt = DateTime.UtcNow.AddHours(24),
            IsUsed = false};
    DbContext.EmailVerifications.Add(verification);
    await DbContext.SaveChangesAsync();
    var link = $"https://localhost:7292/verify-email?token={Uri.EscapeDataString(token)}";
    await EmailService.SendEmailAsync(userProfile.Email, "Verify your email", $@"
    <p>Click to verify: <a href=""{link}"">{link}</a></p>");
}

private async Task ChangePassword(){
    if (currentPasswordInputValue == string.Empty ||
        newPasswordInputValue == string.Empty ||
        confirmPasswordInputValue == string.Empty){
            passwordError = "All fields must be filled";
            return;}
    if (currentPasswordInputValue != userProfile.Password){
        passwordError = "Current password is incorrect"; return;}
    if (newPasswordInputValue.Length < 6){
        passwordError = "Password mustn't be shorter than 6 symbols"; return;}
    if (newPasswordInputValue != confirmPasswordInputValue){
        passwordError = "New password and confirmation do not match"; return;}
    if (userProfile is not null){
        userProfile.Password = newPasswordInputValue;
        DbContext.Users.Update(userProfile);
        await DbContext.SaveChangesAsync();}
    currentPasswordInputValue = string.Empty;
    newPasswordInputValue = string.Empty;
    confirmPasswordInputValue = string.Empty;
    passwordError = string.Empty;
    passwordMessage = "Pasword changed.";
}

protected override async Task OnAfterRenderAsync(bool firstRender){
    await JS.InvokeVoidAsync("resizeTextarea", "aboutTextarea");
}

protected override async Task OnParametersSetAsync(){
    if (IsOpen){
        userProfile = await DbContext.Users.FirstOrDefaultAsync(x => x.Id == userId);
```

```
            UserState.OnChange += UpdateUser;
            if (!string.IsNullOrWhiteSpace(userProfile?.About))
                descriptionInputValue = userProfile.About;
            emailInputValue = userProfile?.Email ?? "Error...";}
    }

    private void UpdateUser(){
        if (userProfile is not null){
            userProfile.IsVerified = UserState.IsVerified;
            emailMessage = "Email has been verified!";
            InvokeAsync(StateHasChanged);}
    }

    public void Dispose() => UserState.OnChange -= UpdateUser;

    private async Task ResizeTextArea(ChangeEventArgs e){
        await JS.InvokeVoidAsync("resizeTextarea", "aboutTextarea");
    }

    private async Task Logout(){
        await LocalStorage.DeleteAsync("lastChat");
        Navigation.NavigateTo("/logout", forceLoad: true);
    }

    private void Close() => OnClose.InvokeAsync();
}
```

ДОДАТОКИ

SignalR хаби

Сторінок 1

Київ-2025

Хаб повідомлень:

```
public class ChatHub : Hub {
   public async Task JoinChat(string chatId) {
      await Groups.AddToGroupAsync(Context.ConnectionId, chatId);}
   public async Task LeaveChat(string chatId) {
      await Groups.RemoveFromGroupAsync(Context.ConnectionId, chatId);}
   public async Task SendMessage(MessageModel mModel) {
      await Clients.Group($"{mModel.ChatType}_{mModel.ChatId}" )
         .SendAsync("ReceiveMessage", mModel);}
   public async Task MessageDeleted(int messageId, string chatId) {
      await Clients.Group(chatId).SendAsync("MessageDeleted", messageId);}
}
```

Хаб роботи з чатами:

```
public class NewChatHub : Hub {
   public Task JoinGroup(int userId) {
      return Groups.AddToGroupAsync(Context.ConnectionId, $"getchat_{userId}");}
   public Task LeaveGroup(int userId) {
      return Groups.RemoveFromGroupAsync(Context.ConnectionId, $"getchat_{userId}");}
   public Task SendPrivateChat(int userId, PrivateChatModel chat) {
      return Clients.Group($"getchat_{userId}").SendAsync("ReceivePrivateChat", chat);}
   public Task DeletePrivateChat(int userId, int chatId) {
      return Clients.Group($"getchat_{userId}").SendAsync("RemovePrivateChat", chatId);}
   public Task SendGroupChat(int userId, GroupChatModel chat) {
      return Clients.Group($"getchat_{userId}").SendAsync("ReceiveGroupChat", chat);}
   public Task DeleteGroupChat(int userId, int chatId) {
      return Clients.Group($"getchat_{userId}").SendAsync("RemoveGroupChat", chatId);}
   public Task NotifyGroupChatNameChange(int userId, int chatId, string chatName)  {
      return Clients.Group($"getchat_{userId}").SendAsync("ChangeGroupChatName", chatId,
chatName);}
}
```

Хаб роботи з новими учасниками групових чатів:

```
public class NewMemberHub : Hub {
   public Task JoinGroup(int chatId){
      return Groups.AddToGroupAsync(Context.ConnectionId, $"member_{chatId}");}
   public Task LeaveGroup(int chatId){
      return Groups.RemoveFromGroupAsync(Context.ConnectionId, $"member_{chatId}");}
   public Task SendMember(int chatId, MemberModel message){
      return Clients.Group($"member_{chatId}").SendAsync("ReceiveMember", message);}
   public Task NotifyRemove(int chatId, int userId) {
      return Clients.Group($"member_{chatId}").SendAsync("RemoveMember", userId);}
}
```

ДОДАТОК К

Додаткові компоненти

Сторінок 8

Київ-2025

1. Компонент учасників групового чату

```
@using DiplomaProgram.Components.ModalWindows
@using DiplomaProgram.Models
@using Microsoft.AspNetCore.SignalR.Client
@using System.Security.Claims
@inject IDbContextFactory<MyDbContext> DbFactory
@inject NavigationManager Navigation

<div class="group-chat-header">
   <div><svg class="user-icon-top" viewBox="0 0 57 57" fill="none"
xmlns="http://www.w3.org/2000/svg"></svg></div>
   <input id="nameInputId" @bind="chatNameInput" class="chat-name-input"
disabled="@(!isEditingName)" />
   @if (!isEditingName){
      <Tooltip Title="Change name" role="button" @onclick="EditName" Class="tooltip-
general"></Tooltip>}
   else{
      <Tooltip Title="Confirm" role="button" @onclick="ChangeChatName" Class="tooltip-
general"></Tooltip>
      <Tooltip Title="Cancel" role="button" @onclick="() => isEditingName = false"
Class="tooltip-general"></Tooltip>}
   <Tooltip Title="Leave group" role="button" @onclick="() => deleteChatDialog?.Show()"
Class="tooltip-general"></Tooltip></div>
<div class="members-section">
   <div class="members-header">
      <h2>Members</h2>
      <Tooltip Title="Add member" role="button" @onclick="ToggleSearchDialog" Class="tooltip-
general"></Tooltip>
   </div>
   @if (members.Any()){
      <ul class="list-group">
         @foreach (var member in members){
            <li class="chat-element" @onclick="() => OpenUserProfileMW(member.UserId)">
               <div class="icon-div">
                  <svg class="user-icon" viewBox="0 0 57 57" fill="none"
xmlns="http://www.w3.org/2000/svg"></svg>
                  @member.DisplayName
               </div></li>}</ul>}</div>
<ConfirmMW @ref="deleteChatDialog"
   ConfirmationTitle="Leave group chat"
   ConfirmationMessage="Are you sure you want to leave this chat?"
   OnConfirm="LeaveGroupChat"
   OnCancel="CancelLeaveChat" />
<AddMemberMW IsOpen="isSearchDialogOpen"
   OnClose="ToggleSearchDialog"
   NewMemberConnection="newMemberConnection"
   CurrentChatId="ChatId"
   NewChatConnection="NewChatConnection"
   CurrentChatName="@chatNameInput" />
```

```
<UserProfileMW IsOpen="isUserProfileOpen" OnClose="CloseUserProfileMW"
userId="idToSearch" />

@code {
    [Parameter] public int ChatId { get; set; }
    [Parameter] public string? ChatName { get; set; }
    [Parameter] public EventCallback<int> RemoveGroupChat { get; set; }
    [Parameter] public string? CurrentChatType { get; set; }
    [Parameter] public int CurrentUserId { get; set; }
    [Parameter] public HubConnection? NewChatConnection { get; set; }
    [Inject] IJSRuntime? JS { get; set; }
    private List<MemberModel> members = new();
    private string? chatNameInput = "Error";
    private HubConnection? newMemberConnection;
    private bool isUserProfileOpen = false;
    private bool isSearchDialogOpen = false;
    private int idToSearch = 0;
    private bool isEditingName = false;
    private ConfirmMW? deleteChatDialog;
    private int previousChatId = 0;

    protected override async Task OnInitializedAsync(){
        newMemberConnection = new HubConnectionBuilder()
            .WithUrl(Navigation.ToAbsoluteUri("/newmemberhub"))
            .WithAutomaticReconnect().Build();
        newMemberConnection.On<MemberModel>("ReceiveMember", (message) =>{
            if (members.Count() > 0){
                members.Insert(1, message);
                InvokeAsync(StateHasChanged);}});
        newMemberConnection.On<int>("RemoveMember", (userId) =>{
            members.RemoveAll(x => x.UserId == userId);
            InvokeAsync(StateHasChanged);});
        await newMemberConnection.StartAsync();
    }

    protected override async Task OnParametersSetAsync(){
        if (CurrentChatType == "group"){
            members.Clear();
            chatNameInput = ChatName;
            await newMemberConnection.SendAsync("LeaveGroup", previousChatId);
            await newMemberConnection.SendAsync("JoinGroup", ChatId);
            previousChatId = ChatId;
            await using var context = await DbFactory.CreateDbContextAsync();
            var dbMembers = await context.Members
                .Where(m => m.GroupId == ChatId).Include(m => m.User).ToListAsync();
            members = dbMembers
                .OrderByDescending(m => m.UserId == CurrentUserId)
                .Select(m => new MemberModel{
                        UserId = m.User.Id,
                        DisplayName = m.User.DisplayName,
```

```
                Role = m.Role}).ToList();}
    }

    private void OpenUserProfileMW(int id){
        idToSearch = id;
        isUserProfileOpen = true;
    }

    private void CloseUserProfileMW(){isUserProfileOpen = false;}

    private void ToggleSearchDialog(){isSearchDialogOpen = !isSearchDialogOpen;}

    private async Task ChangeChatName(){
        isEditingName = false;
        await using var context = await DbFactory.CreateDbContextAsync();
        var chat = await context.GroupChats.FirstOrDefaultAsync(c => c.Id == ChatId);
        if (chat != null){
            chat.Name = chatNameInput.Trim();
            await context.SaveChangesAsync();
            foreach(var member in members){
                await NewChatConnection.SendAsync("NotifyGroupChatNameChange",
member.UserId, ChatId, chatNameInput);}
        }
    }

    private async Task LeaveGroupChat(){
        await newMemberConnection.SendAsync("NotifyRemove", ChatId, CurrentUserId);
        await using var context = await DbFactory.CreateDbContextAsync();
        var member = await context.Members
            .FirstOrDefaultAsync(m => m.GroupId == ChatId && m.UserId == CurrentUserId);
        if (member != null){
            context.Members.Remove(member);
            await context.SaveChangesAsync();}
        bool hasMembers = await context.Members
            .AnyAsync(m => m.GroupId == ChatId);
        if (!hasMembers){
            var chat = await context.GroupChats.FindAsync(ChatId);
            if (chat != null){
                context.GroupChats.Remove(chat);
                await context.SaveChangesAsync();}}
        await RemoveGroupChat.InvokeAsync(ChatId);
    }

    private Task CancelLeaveChat(){ return Task.CompletedTask; }

    private async Task EditName(){
        isEditingName = true;
        await Task.Delay(30);
        await JS.InvokeVoidAsync("focusElement", "nameInputId");
    }}
```

2. Компонент деталей приватного чату

```
@using DiplomaProgram.Models
@inject IDbContextFactory<MyDbContext> DbFactory

@if (userDetails is not null){
   <div class="private-chat-container"><div class="icon-row"><div class="icon-div"></div>
         <div class="user-name-div">
            <h5 class="icon-text">@userDetails.DisplayName</h5>
            <h6 class="icon-text">@("@" + userDetails.UserName)</h6>
         </div>
      </div>
      <div class="content-row"><h5>About</h5>
         <div class="text-scroll-div">
            <div class="overflow-div scrollbar-clean">
               @(string.IsNullOrWhiteSpace(userDetails.About)
                     ? "This user has not added a description."
                     : userDetails.About)
            </div></div></div>
      <div class="content-row">
         <h5>Join date</h5>
         <div class="text-div">@userDetails.JoinDate.ToString("dd-MM-yyyy")</div>
      </div>
   </div>}
else{<p>Loading user details...</p>}

@code {
   [Parameter] public int userId { get; set; }
   private UserDetailsDto? userDetails;

   protected override async Task OnParametersSetAsync(){
      await using var context = await DbFactory.CreateDbContextAsync();
      userDetails = await context.Users
         .AsNoTracking().Where(x => x.Id == userId)
         .Select(x => new UserDetailsDto{
               DisplayName = x.DisplayName,
               UserName = x.UserName,
               About = x.About,
               JoinDate = x.JoinDate
            }).FirstOrDefaultAsync();
   }
}
```

3. Компонента повідомлення

```
@using DiplomaProgram.Models
@using DiplomaProgram.Services
@inject IDbContextFactory<MyDbContext> DbFactory
@inject ChatHubService ChatHub
```

```
<textarea @bind="messageText" rows="1" @oninput="ResizeTextArea" id="msgTextarea"
placeholder="Message..."></textarea>
<div class="icon-container">
   <Tooltip Title="Send" role="button" @onclick="SendMessage" Class="tooltip-
general"></Tooltip>
</div>

@code {
   [Parameter] public int CurrentUserId { get; set; }
   [Parameter] public string? CurrentUserDisplayName { get; set; }
   [Parameter] public int CurrentChatId { get; set; }
   [Parameter] public string CurrentChatType { get; set; } = "private";
   [Parameter] public EventCallback<(int ChatId, string ChatType)> OnChatMovedToTop { get;
set; }
   [Inject] IJSRuntime JS { get; set; }
   private string messageText = string.Empty;
   private MyDbContext? DbContext;

   protected override async Task OnInitializedAsync(){
      DbContext = await DbFactory.CreateDbContextAsync();
   }

   private async Task SendMessage(){
      if (!string.IsNullOrWhiteSpace(messageText) && DbContext != null){
         var now = DateTime.UtcNow;
         var messageEntity = new Message{
               UserId = CurrentUserId,
               Text = messageText,
               Date = now};
         if (CurrentChatType == "private")
         {
            messageEntity.PrivateChatId = CurrentChatId;

            var privateChat = await DbContext.PrivateChats.FindAsync(CurrentChatId);
            if (privateChat != null)
               privateChat.LastMessageDate = now;
         }
         else{
            messageEntity.GroupChatId = CurrentChatId;
            var groupChat = await DbContext.GroupChats.FindAsync(CurrentChatId);
            if (groupChat != null)
               groupChat.LastMessageDate = now;}
         DbContext.Messages.Add(messageEntity);
         await DbContext.SaveChangesAsync();
         var messageModel = new MessageModel{
               MessageId = messageEntity.Id,
               ChatId = CurrentChatId,
               SenderId = CurrentUserId,
```

```
            SenderDisplayName = CurrentUserDisplayName ?? string.Empty,
            Text = messageEntity.Text,
```



```
            SentAt = messageEntity.Date,
            ChatType = CurrentChatType};
        await ChatHub.SendMessageAsync(messageModel);
        await OnChatMovedToTop.InvokeAsync((CurrentChatId, CurrentChatType));
        messageText = string.Empty;
        StateHasChanged();
        await ResizeTextArea();
        await JS.InvokeVoidAsync("focusElement", "msgTextarea");}
    }

    private async Task ResizeTextArea(){
        await JS.InvokeVoidAsync("resizeTextarea", "msgTextarea");
        await JS.InvokeVoidAsync("scrollToBottom", "chatScrollContainer");
    }
}
```

4. Компонента повідомлення

```
@using DiplomaProgram.Components.ModalWindows
@using DiplomaProgram.Models
@using DiplomaProgram.Services
@inject ChatHubService ChatHub
@inject MyDbContext DbContext

<ul class="message-list">
    @foreach (var msg in messages){
        <li class="@(CurrentUserId == msg.SenderId ? "message-current" : "message-other")">
            @if (msg.SenderId == CurrentUserId){
                <svg @onclick="() => ConfirmDelete(msg.MessageId)" class="message-delete-button"
viewBox="0 0 33 37" fill="none" xmlns="http://www.w3.org/2000/svg"></svg>}
            <div class="message-info">
                <div>
                    @if (ChatType == "group" && CurrentUserId != msg.SenderId){
                        <p class="m-name-other">@msg.SenderDisplayName</p>}
                </div>
                <div class="time-wrapper">
                    <p class="message-time">@FormatMessageTime(msg.SentAt)</p>
                </div>
            </div>
            <p class="message-text">@msg.Text</p></li>}</ul>
<ConfirmMW @ref="deleteDialog"
    ConfirmationTitle="Message deletion"
    ConfirmationMessage="Are you sure you want to delete this message?"
    OnConfirm="DeleteConfirmed"
    OnCancel="CancelDelete" />

@code {
```

```
[Parameter] public int ChatId { get; set; }
[Parameter] public int CurrentUserId { get; set; }
```



```
[Parameter] public string? ChatType { get; set; }
[Inject] private IJSRuntime JSRuntime { get; set; }
private ConfirmMW? deleteDialog;
private int messageIdToDelete;
private List<MessageModel> messages = new();
private bool _shouldScrollToBottom = false;

private void ConfirmDelete(int id){
    messageIdToDelete = id;
    deleteDialog?.Show();}

private Task CancelDelete(){
    messageIdToDelete = 0;
    return Task.CompletedTask;
}

protected override async Task OnParametersSetAsync(){
    messages.Clear();
    List<MessageModel> dbMessages;
    if (ChatType == "private"){
        dbMessages = await DbContext.Messages
            .Where(m => m.PrivateChatId == ChatId)
            .OrderBy(m => m.Date)
            .Select(m => new MessageModel{
                    MessageId = m.Id,
                    ChatId = m.PrivateChatId ?? 0,
                    SenderId = m.UserId,
                    SenderDisplayName = m.User.DisplayName,
                    Text = m.Text,
                    SentAt = m.Date}).ToListAsync();}
    else if (ChatType == "group"){
        dbMessages = await DbContext.Messages
            .Where(m => m.GroupChatId == ChatId).OrderBy(m => m.Date)
            .Select(m => new MessageModel{
                    MessageId = m.Id,
                    ChatId = m.GroupChatId ?? 0,
                    SenderId = m.UserId,
                    SenderDisplayName = m.User.DisplayName,
                    Text = m.Text,
                    SentAt = m.Date
                }).ToListAsync();}
    else{dbMessages = new();}
    messages.AddRange(dbMessages);
    await ChatHub.JoinChatAsync($"{ChatType}_{ChatId}");
    _shouldScrollToBottom = true;
}

protected override async Task OnAfterRenderAsync(bool firstRender){
```

```
        if (_shouldScrollToBottom){
            _shouldScrollToBottom = false;
```



```
            await JSRuntime.InvokeVoidAsync("scrollToBottom", "chatScrollContainer");}
    }

    protected override async Task OnInitializedAsync(){
        ChatHub.OnMessageReceived += HandleIncomingMessage;
        ChatHub.OnMessageDeleted += HandleMessageDeleted;
        await ChatHub.StartAsync();
    }

    private void HandleIncomingMessage(MessageModel msg){
        if (msg.ChatId == ChatId){
            messages.Add(msg);
            InvokeAsync(async () =>{
                StateHasChanged();
                await Task.Yield();
                await Task.Delay(20);
                await JSRuntime.InvokeVoidAsync("scrollToBottom", "chatScrollContainer");});}
    }

    private void HandleMessageDeleted(int messageId){
        var toRemove = messages.FirstOrDefault(m => m.MessageId == messageId);
        if (toRemove != null){
            messages.Remove(toRemove);
            InvokeAsync(StateHasChanged);}
    }
    private async Task DeleteConfirmed(){
        var message = await DbContext.Messages.FindAsync(messageIdToDelete);
        if (message != null){
            DbContext.Messages.Remove(message);
            await DbContext.SaveChangesAsync();
            await ChatHub.NotifyMessageDeletedAsync(messageIdToDelete,
$"{ChatType}_{ChatId}");}
        messageIdToDelete = 0;
    }
    private string FormatMessageTime(DateTime utcTimestamp){
        var localTime = TimeZoneInfo.ConvertTimeFromUtc(
            utcTimestamp,
            TimeZoneInfo.FindSystemTimeZoneById("FLE Standard Time"));
        var now = TimeZoneInfo.ConvertTimeFromUtc(
            DateTime.UtcNow,
            TimeZoneInfo.FindSystemTimeZoneById("FLE Standard Time"));
        if (localTime.Date == now.Date){return localTime.ToString("HH:mm");}
        else if (localTime.Date == now.Date.AddDays(-1)){return $"yesterday at
{localTime:HH:mm}";}
        else{return localTime.ToString("dd.MM.yyyy HH:mm");}
    }
    public void Dispose(){
        ChatHub.OnMessageReceived -= HandleIncomingMessage;
```

```
        ChatHub.OnMessageDeleted -= HandleMessageDeleted; }
}
```

ДОДАТОК Л

Сервіси

Київ-2025

Сервіс обробки повідомлень:

```
public class ChatHubService{
   private readonly NavigationManager _navigationManager;
   private HubConnection? _hubConnection;
   private string? _currentChatId;
   public event Action<MessageModel>? OnMessageReceived;
   public event Action<int>? OnMessageDeleted;

   public ChatHubService(NavigationManager navigationManager){
      _navigationManager = navigationManager;
   }

   public async Task StartAsync(){
      if (_hubConnection != null)
         return;
      _hubConnection = new HubConnectionBuilder()
         .WithUrl(_navigationManager.ToAbsoluteUri("/chathub"))
         .WithAutomaticReconnect().Build();
      _hubConnection.On<MessageModel>("ReceiveMessage", (message) =>{
         OnMessageReceived?.Invoke(message);});
      _hubConnection.On<int>("MessageDeleted", messageId =>{
         OnMessageDeleted?.Invoke(messageId);});
      await _hubConnection.StartAsync();
   }

   public async Task JoinChatAsync(string chatId){
      if (_currentChatId != null)
     await _hubConnection!.InvokeAsync("LeaveChat", _currentChatId);
      _currentChatId = chatId;
      await _hubConnection!.InvokeAsync("JoinChat", chatId);
   }

   public async Task SendMessageAsync(MessageModel message){
      await _hubConnection!.InvokeAsync("SendMessage", message);
   }

   public async Task NotifyMessageDeletedAsync(int messageId, string chatId){
      if (_hubConnection is not null){
         await _hubConnection.InvokeAsync("MessageDeleted", messageId, chatId);}
   }

   public async Task StopAsync(){
      if (_hubConnection is not null){
         await _hubConnection.StopAsync();
         await _hubConnection.DisposeAsync();}
      _hubConnection = null;
```

```
    }
}
```


Сервіс оновлення елементів верифікації пошти:

```
public class UserStateService{
    public bool IsVerified { get; private set; }
    public event Action? OnChange;
    public void SetIsVerified(bool isVerified){
        IsVerified = isVerified;
        OnChange?.Invoke();
    }
}
```

Сервіс роботи з електроними листами для верифікації пошти:

```
public class EmailService{
    private readonly IConfiguration _configuration;
    public EmailService(IConfiguration configuration){
        _configuration = configuration;
    }

    public async Task SendEmailAsync(string toEmail, string subject, string body){
        var smtpClient = new SmtpClient{
            Host = _configuration["Email:Smtp:Host"],
            Port = int.Parse(_configuration["Email:Smtp:Port"]!),
            EnableSsl = true,
            Credentials = new NetworkCredential(
                _configuration["Email:Smtp:Username"],
                _configuration["Email:Smtp:Password"]
            )
        };

        var mailMessage = new MailMessage{
            From = new MailAddress(_configuration["Email:Smtp:Username"]!),
            Subject = subject,
            Body = body,
            IsBodyHtml = true
        };
        mailMessage.To.Add(toEmail);
        await smtpClient.SendMailAsync(mailMessage);
    }
}
```