ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Цифрова система обміну зашифрованими повідомленнями з використанням ASP.NET Core»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Технології цифрового розвитку»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Денис ДАЛЬКЕВИЧ
(підпис)

Виконав: здобувач вищої освіти групи ТЦР-43
Денис ДАЛЬКЕВИЧ

Керівник: Ждан ЧЕРНЯВСЬКИЙ
Ст.викл.

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Технологій цифрового розвитку

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Технології цифрового розвитку»

ЗАТВЕРДЖУЮ

Завідувач кафедри
Технологій цифрового розвитку
_____ Жебка ВІКТОРІЯ
« ____ » _____ 2026 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

Далькевичу Денису Олександровичу

1. Тема кваліфікаційної роботи: «Цифрова система обміну зашифрованими повідомленнями з використанням ASP.NET Core» керівник кваліфікаційної роботи ст. викл. Ждан ЧЕРНЯВСЬКИЙ, затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 р. № 51.

2. Строк подання кваліфікаційної роботи «18» травня 2026 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи наскрізного шифрування та протоколи захисту даних, огляд сучасних систем обміну повідомленнями (WhatsApp, Telegram, Signal), опис архітектурних підходів у веб-розробці, технічна документація з реалізації веб-застосунків на базі ASP.NET Core та Angular, вимоги до функціональності захищених комунікаційних систем, рекомендації щодо реалізації E2E-шифрування.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Аналіз предметної області: огляд існуючих систем обміну повідомленнями, порівняння протоколів real-time комунікації (WebSockets, SSE, Long Polling), аналіз механізмів наскрізного шифрування.

2. Проєктування веб-сервісу для обміну зашифрованими повідомленнями з урахуванням клієнт-серверної архітектури, моделі бази даних, REST API та механізмів автентифікації (JWT).
3. Програмна реалізація серверної частини на ASP.NET Core з використанням SignalR для real-time комунікації та клієнтської частини на Angular з реалізацією AES-256 шифрування.
4. Тестування системи: юніт-тестування криптографічних сервісів, інтеграційне тестування взаємодії клієнта та сервера, навантажувальне тестування продуктивності.

5. Перелік графічного матеріалу: презентація
   1. Аналіз існуючих систем обміну повідомленнями.
   2. Вимоги до програмного забезпечення.
   3. Програмні засоби реалізації (ASP.NET Core, Angular, SignalR).
   4. Діаграма варіантів використання.
   5. Архітектура системи та діаграма компонентів.
   6. Діаграма класів та ER-модель бази даних.
   7. Екранні форми застосунку.
   8. Результати тестування системи.
6. Дата видачі завдання «20» лютого 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів	Примітка
1	Підбір та аналіз науково-технічної літератури	20.02.2026 – 27.02.2026	виконано
2	Аналіз та дослідження існуючих аналогів	27.02.2026 – 06.03.2026	виконано
3	Проєктування криптографічного протоколу та архітектури системи захищеного обміну повідомленнями	06.03.2026 – 13.03.2026	виконано
4	Проєктування бази даних, REST API та клієнтського інтерфейсу веб-застосунку	13.03.2026 – 27.03.2026	виконано
5	Програмна реалізація застосунку	27.03.2026 – 17.04.2026	виконано
6	Тестування застосунку	17.04.2026 – 01.05.2026	виконано
7	Оформлення роботи: вступ, висновки, реферат	01.05.2026 – 04.05.2026	виконано
8	Розробка демонстраційних матеріалів	04.05.2026 – 05.05.2026	виконано
9	Попередній захист роботи	05.05.2026 – 18.05.2026	виконано

Здобувач вищої освіти _____ Денис ДАЛЬКЕВИЧ
(підпис)

Керівник
кваліфікаційної роботи _____ Ждан ЧЕРНЯВСЬКИЙ
(підпис)

РЕФЕРАТ

Обсяг роботи: 50 стор., 9 табл., 13 рис., 2 додатки, 21 джерел.

Метою роботи є розробка та практична реалізація системи обміну миттєвими повідомленнями з наскрізним шифруванням, що забезпечує конфіденційність переписки на технічному рівні незалежно від стану серверної інфраструктури.

Об'єкт дослідження – процес захищеного обміну повідомленнями в реальному часі між користувачами через веб-застосунок.

Предмет дослідження – методи та засоби реалізації наскрізного шифрування в клієнт-серверній архітектурі на базі ASP.NET Core та Angular.

Короткий зміст роботи: у кваліфікаційній роботі розроблено веб-сервіс для обміну зашифрованими повідомленнями на базі ASP.NET Core та Angular. Система реалізована за принципом анонімного доступу, без реєстрації електронної пошти чи збору персональних даних. Користувач вводить ім'я та пароль, після чого система автоматично створює обліковий запис, якщо такого користувача не існує, якщо обліковий запис вже зареєстровано, то відбувається вхід. Проведено аналіз існуючих систем обміну повідомленнями та обґрунтовано вибір технологічного стеку. Спроєктовано трирівневу архітектуру з чітким розмежуванням між шарами презентації, бізнес-логіки та доступу до даних. Розроблено криптографічний протокол на базі AES-256-CBC з динамічною генерацією параметрів IV та Salt через функцію деривації PBKDF2 для кожного окремого повідомлення – сервер зберігає виключно зашифровані дані та позбавлений технічної можливості відновити відкритий текст. Реалізовано глобальний та приватний чат, відправку файлів, редагування та видалення повідомлень зі статусами доставки. Проведено трирівневе тестування: 25 юніт-тестів криптографічного модуля з покриттям 98.48% коду, 22 інтеграційні тести API та навантажувальне тестування через Chrome DevTools.

Серверна частина реалізована на ASP.NET Core з SignalR Hub для маршрутизації повідомлень у реальному часі через WebSocket. Клієнтська

частина реалізована на Angular з наскрізним шифруванням AES-256-CBC безпосередньо в браузері. Авторизація побудована на JWT-токенах у HttpOnly cookie. Проведено інтеграційне та навантажувальне тестування – затримка доставки повідомлень менше 100 мс, частота кадрів 60 FPS при CPU throttling 4x.

Сферою використання: системи захищеного обміну повідомленнями, корпоративні та приватні комунікаційні платформи з вимогами до конфіденційності даних.

Ключові слова: ASP.NET CORE, ANGULAR, SIGNALR, AES-256-CBC, НАСКРІЗНЕ ШИФРУВАННЯ, WEBSOCKET, JWT, PBKDF2, REAL-TIME КОМУНІКАЦІЯ, АНОНІМНИЙ ДОСТУП.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AES – Advanced Encryption Standard – стандарт симетричного шифрування

API – Application Programming Interface – інтерфейс програмування застосунків

CBC – Cipher Block Chaining – режим зчеплення блоків шифру

CLS – Cumulative Layout Shift – накопичений зсув макету (метрика Web Vitals)

CORS – Cross-Origin Resource Sharing – механізм обміну ресурсами між різними джерелами

CSRF – Cross-Site Request Forgery – міжсайтова підробка запитів

DOM – Document Object Model – об'єктна модель документа

E2E – End-to-End – наскрізне (від кінця до кінця)

FPS – Frames Per Second – кількість кадрів на секунду

GLSL – OpenGL Shading Language – мова шейдерів OpenGL

HTTP – HyperText Transfer Protocol – протокол передачі гіпертексту

HTTPS – HTTP Secure – захищений протокол передачі гіпертексту

IV – Initialization Vector – вектор ініціалізації

JWT – JSON Web Token – веб-токен формату JSON

PBKDF2 – Password-Based Key Derivation Function 2 – функція деривації ключа на основі пароля

REST – Representational State Transfer – архітектурний стиль передачі стану представлення

RxJS – Reactive Extensions for JavaScript – бібліотека реактивного програмування для JavaScript

SignalR – бібліотека ASP.NET Core для real-time комунікації через WebSocket

SPA – Single Page Application – односторінковий застосунок

SQL – Structured Query Language – мова структурованих запитів

UI – User Interface – інтерфейс користувача

WebGL – Web Graphics Library – бібліотека веб-графіки

WebSocket – протокол двостороннього зв'язку між клієнтом та сервером

XSS – Cross-Site Scripting – міжсайтовий скриптинг

ВСТУП

Актуальність теми. Для сучасного цифрового світу обмін миттєвими повідомленнями став невід'ємною частиною повсякденного спілкування мільярдів користувачів. Кожен найпопулярніший месенджер, за даними Statista на 2024 рік, налічує понад два мільярди користувачів [1]. Стрімке зростання цифрової комунікації супроводжується суттєвим загостренням проблем кібербезпеки та витоком персональних даних. Несанкціонований доступ до конфіденційної інформації, приватного листування та масове стеження з боку державних структур і комерційних корпорацій вказують на критичну важливість захисту приватності користувачів. Наскрізне шифрування End-to-End Encryption, E2EE є найвищим стандартом захисту, що гарантує конфіденційність повідомлень навіть у випадку витоку даних з серверу. Водночас користувачі очікують не лише високого рівня безпеки, але й зручної, швидкодіючої та функціональної платформи для комунікації, що потребує застосування сучасних технологій “real-time” зв'язку та ефективних архітектурних рішень.

В сучасних реаліях України, з огляду на активні кіберзагрози та необхідність захисту чутливих даних, розробка захищених комунікаційних систем є особливо нагальним завданням. Вітчизняні підприємства, державні установи та громадяни потребують надійних інструментів для безпечного обміну інформацією, що не залежать від іноземної серверної інфраструктури та комерційних інтересів третіх сторін. Саме тому створення вітчизняних захищених месенджерів із відкритою архітектурою та наскрізним шифруванням є стратегічно важливим напрямом розвитку національної цифрової інфраструктури.

Мета роботи – покращення процесу обміну зашифрованими повідомленнями за допомогою розробленої системи з використанням ASP.NET Core. Проєктування веб-застосунку обміну даними з наскрізним шифруванням, що має базуватися на сучасному технологічному стеку та гарантувати високу конфіденційність, масштабованість і продуктивність комунікацій.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- проаналізувати предметну область та існуючі архітектурні рішення;
- обґрунтувати вибір протоколів комунікації в реальному часі та технології двонаправленого зв'язку ”SignalR”;
- дослідити криптографічні алгоритми зокрема, алгоритм симетричного шифрування “AES-256” та механізми динамічної генерації сесійних ключів;
- спроєктувати клієнт-серверну архітектуру та реляційну модель бази даних;
- реалізувати серверну та клієнтську частини програмної системи;
- провести комплексне тестування розробленого програмного забезпечення.

Об'єкт дослідження – процеси обміну миттєвими повідомленнями в розподілених веб-системах із високими вимогами до конфіденційності.

Предмет дослідження – методи реалізації наскрізного шифрування, протоколи двонаправленої “real-time” комунікації та архітектурні патерни побудови захищених систем.

Методи дослідження**:** системний аналіз предметної області, порівняльний аналіз технологічних рішень, методи об'єктно-орієнтованого проєктування архітектури та емпіричне тестування продуктивності.

Наукова новизна одержаних результатів полягає в розробці комплексного підходу до захисту даних, що поєднує використання протоколів двонаправленої комунікації в реальному часі для маршрутизації зашифрованих пакетів за принципом Zero-Knowledge та динамічну генерацію криптографічних параметрів сесії, вектора ініціалізації та криптографічної солі, для кожного окремого повідомлення.

Практична значущість одержаних результатів полягає у розробці системи, яка може використовуватись як основа для корпоративних

месенджерів з підвищеними вимогами до безпеки, зокрема у медичній та юридичній сферах. На відміну від більшості популярних месенджерів, дана система має реалізуватися за принципом анонімного доступу, без реєстрації email чи збору персональних даних.

Структура роботи включає вступ, три розділи, загальні висновки, перелік посилань та додатки. У першому розділі проводиться аналіз предметної області та вибір технологічного стеку. Другий розділ присвячується проєктуванню архітектури системи та моделі бази даних. У третьому розділі описується реалізація програмних компонентів та результати тестування.

Додатки містять лістинги програмного коду ключових компонентів системи: серверної частини (Program.cs, ChatHub.cs, ChatService.cs), клієнтської частини (chat.service.ts з E2E-шифруванням), конфігураційні файли та діаграми.

Апробація результатів дослідження:

1. Далькевич Д. О., Чернявський Ж. А. Проєктування системи обміну повідомленнями з наскрізним шифруванням на базі ASP.NET Core та Angular. Всеукраїнську науково-технічну конференцію «Технології цифрового розвитку: програмні системи та децентралізація», Київ, 2026. Збірник тез. – (подано до друку)
2. Далькевич Д. О., Чернявський Ж. А. Порівняльний аналіз веб-сервісів обміну повідомленнями за критеріями анонімності та захисту даних. Всеукраїнську науково-технічну конференцію «Технології цифрового розвитку: програмні системи та децентралізація», Київ, 2026. Збірник тез. – (подано до друку)

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИБІР ТЕХНОЛОГІЧНОГО СТЕКУ

1.1 Огляд сучасних систем обміну повідомленнями

Системи обміну швидкими повідомленнями, месенджери, стали невід'ємною складовою сучасної цифрової комунікації. Існуючі платформи еволюціонували від простих засобів обміну текстом до багатофункціональних середовищ, які підтримують надсилання мультимедійного контенту, відеодзвінки, голосові повідомлення та інтеграцію із зовнішніми сервісами.

Централізовані месенджери спираються на клієнт-серверну архітектуру, де всі повідомлення проходять через головні сервери розробника. До цієї групи відносяться WhatsApp, Telegram, Facebook Messenger. Перевагою централізованої архітектури є легкість синхронізації даних між різними пристроями користувача та забезпечення високої швидкості доставки повідомлень завдяки оптимізованій серверній інфраструктурі.

Відмова від централізованої інфраструктури на користь таких протоколів, як Matrix або Jabber, дозволяє побудувати стійку до зовнішнього втручання мережу, де дані не контролюються єдиним власником. Це суттєво підсилює безпеку, проте накладає додаткові вимоги до технічної підтримки вузлів. Також варто враховувати те, що через складні алгоритми маршрутизації в децентралізованих архітектурах швидкість доставки миттєвих повідомлень може бути нижчою, ніж у традиційних сервісах.

“Від користувача до користувача” месенджери забезпечують пряме з’єднання між користувачами, не залучаючи проміжні сервери. До них відносяться Tox, Briar та певні режими роботи Telegram. Такий підхід гарантує максимальну приватність, але має обмеження стосовно доставки повідомлень користувачам, які перебувають в режимі офлайн.

Корпоративні месенджери, такі як Slack, Microsoft Teams, Discord та Rocket.Chat, призначені для спільної роботи над проєктам, тематичного спілкування. Вони пропонують зручний поділ обговорень за темами, що позбавляє хаосу в листуванні. Користувачі можуть легко об'єднувати месенджер із робочим софтом, швидко знаходити потрібні файли в історії та проводити відеозустрічі, працюючи над документами в одному вікні.

Універсальні месенджери орієнтовані на широку аудиторію і підтримують як особисте, так і групове спілкування. WhatsApp, Telegram, Facebook Messenger пропонують збалансований набір функцій для різних сценаріїв використання. Ці сервіси прості і зрозумілі у використанні, працюють на різних девайсах та постійно оновлюють свої можливості. Спеціалізовані месенджери фокусуються на конкретних завданнях:

- сервіси для ігрових комунікацій;
- засоби анонімного спілкування з підвищеним рівнем захисту;
- корпоративні відеоплатформи з функціями обміну повідомленнями;
- системи в медичній сфері з підвищеним рівнем захисту персональних даних пацієнтів.

Швидкість доставки повідомлень напряму залежить від обраного мережевого протоколу. Стандартом для систем реального часу став WebSocket, який дозволяє клієнту та серверу миттєво обмінюватися інформацією. Менш ресурсомістким, але обмеженим лише одним напрямком передачі, є метод SSE. Водночас для підтримки старих браузерів, що не розуміють сучасних стандартів, у архітектуру часто закладають механізм Long Polling.

Шифрування та захищеність стали надзвичайно важливими чинниками після численних випадків несанкціонованого розголошення даних. Наскрізне кодування "End-to-End Encryption" за протоколом Signal Protocol застосовують WhatsApp, Signal та Telegram у "таємних бесідах" [2]. За цих умов, навіть постачальник послуги не володіє доступом до вмісту звісток. Інші месенджери застосовують транспортне кодування "TLS/SS", що захищає дані під час передачі, але дає змогу серверу розшифровувати вміст.

Зберігання даних може бути хмарним, з синхронізацією між пристроями, або локальним, тільки на пристрої користувача. Telegram зберігає всі дані в хмарі, забезпечуючи доступ з будь-якого пристрою та необмежений час зберігання [3]. WhatsApp використовує гібридний підхід, завдяки якому основні дані зберігаються локально на пристрої, а резервні копії передаються до хмарних сховищ Google Drive або iCloud.

Мікросервісна архітектура домінує в масштабованих месенджерах. Розподіл функцій дозволяє винести автентифікацію, чати, хмарне сховище та статистику в окремі блоки. Завдяки цьому можна швидко підсилювати лише ті компоненти, які не справляються з великою кількістю користувачів, не зачіпаючи всю систему.

Подієво-орієнтована архітектура забезпечує асинхронну обробку ключових запитів. Кожна дія, стає окремим завданням у черзі. Повідомлення розміщуються в спеціалізовану чергу на базі Apache Kafka або RabbitMQ, система може опрацьовувати мільйони подій на секунду та гарантує повну цілісність даних без втрат.

Здійснений аналіз засвідчує, що найважливішою тенденцією є суттєве посилення вимог щодо приватності. Користувачі дедалі частіше обирають месенджери, де навіть розробник не має доступу до їхніх повідомлень. Це й зумовило вибір наскрізного шифрування як необхідної умови до створеної системи.

Зіставлення проаналізованих платформ дає змогу виявити спільні вектори та вагомі розбіжності у методах впровадження захисту. WhatsApp оперує на базі Signal Protocol для E2EE, проте належить корпорації “Meta”, яка накопичує значні метадані щодо діяльності користувачів: з ким і як часто тривають бесіди, геопозиція та характеристики пристроїв. Telegram у звичайному режимі зберігає послання у власній хмарній інфраструктурі у зашифрованому, але доступному для сервера вигляді, E2EE використовується винятково у режимі “таємних бесід”, які не підтримують колективні розмови та узгодженість між приладами. Signal слугує еталоном приватності серед комерційних месенджерів, цей сервіс має найменше накопичення метаданих, відкритий код та обов’язкове шифрування для усіх видів

спілкування. Проте Signal потребує прив’язки до мобільного номера, що унеможливлює цілком анонімне застосування. Створена система усуває цей недолік, гарантуючи анонімний вхід без збору жодних особистих відомостей та зберігає шифрування даних усіх повідомлень.

1.2 Функціональні та нефункціональні вимоги до розробки чату в реальному часі

Розробка системи обміну миттєвими повідомленнями потребує чіткого визначення вимог, які поділяються на функціональні, які система має виконувати, та нефункціональні, що встановлюють критерії якості та способи їх виконання. Чітка специфікація вимог на початкових етапах проєктування допомагає мінімізувати ризики в архітектурі. Такий системний підхід гарантує, що розроблений інструментарій буде стабільним, масштабованим і повністю відповідатиме запитам аудиторії.

1.2.1 Функціональні вимоги

Програмний засіб повинен підтримувати автентифікацію за принципом анонімного доступу, за якого користувач вводить ім'я та пароль. У випадку відсутності такого ідентифікатора в базі даних профіль створюється автоматично, тоді як при знаходженні наявного імені відбувається перевірка введеного пароля. Схожа модель не вимагає підтвердження електронної пошти чи проходження складної реєстрації, що забезпечує максимально швидкий вхід у додаток.

Обмін миттєвими повідомленнями складає головну функціональність вебсервісу. Програмна розробка повинна гарантувати пересилання текстових даних у кодуванні Unicode, охоплюючи графічні символи й особливі знаки. Додатково слід запровадити засоби форматування, які покращують подання тексту за допомогою курсиву, жирного накреслення, моноширинних вставок та

гіпертекстових зв'язків. Платформа мусить забезпечувати негайне доставляння повідомлень із показом статусів: "надіслано", "доставлено", "переглянуто". Підтримка корегування надісланих повідомлень із відміткою про зміни та опція видалення повідомлень, лише для себе чи для всіх користувачів, є звичними вимогами теперішніх клієнтів.

Передача файлів розширює можливості спілкування. Система повинна підтримувати завантаження та відображення зображень з розширеннями jpg, jpeg, png, gif прямо в чаті, а також відправку файлів інших форматів з можливістю завантаження.

1.2.2 Нефункціональні вимоги

У процесі проектування системи було визначено низку важливих вимог до її функціонування. Показник продуктивності обмежено часом затримки доставки у діапазоні 100–150 мс, тоді як безпека базується на принципі Zero Knowledge, що виключає можливість дешифрування даних на сервері. Стабільність роботи додатка досягається за рахунок механізмів контролю цілісності інформації у випадках нестабільного мережевого з’єднання. Сумісність з різними браузерами та коректне відображення на різних типах пристроїв (від десктопів до смартфонів) має бути реалізовано за допомогою сітки Bootstrap Grid [4]. Особлива увага приділена сервісному обслуговуванню: впроваджена система багаторівневого логування дозволяє детально аналізувати шлях кожного повідомлення та оперативно усувати помилки в роботі криптографічних алгоритмів.

Такі вимоги формують технічне завдання для розробки системи обміну повідомленнями та служать критеріями якості на етапі тестування. Баланс між функціональними можливостями та нефункціональними характеристиками визначає успішність проекту та задоволеність користувачів.

1.3 Аналіз існуючих архітектурних рішень для асинхронної комунікації

Асинхронний зв’язок докорінно змінює логіку взаємодії клієнта з сервером. На відміну від традиційної синхронної моделі "запит-відповідь", асинхронні технології дозволяють серверу ініціювати передачу даних клієнту без попереднього запиту. Це дуже важливо для систем обміну повідомленнями, де повідомлення мають доставлятися миттєво, як тільки відправник натискає кнопку "Відправити", незалежно від того, чи виконував отримувач будь-які запити до сервера в цей момент.

1.3.1 Порівняння протоколів WebSockets, Server-Sent Events (SSE) та Long Polling

Старішим методом для отримання інформації в дійсному часі є “Long Polling”, при якому клієнт надсилає запит до сервера і чекає на відповідь до виникнення нових відомостей. Суть цього підходу полягає в тому, що клієнт надсилає HTTP-запит до сервера, але сервер не відповідає негайно, а зберігає з'єднання відкритим до моменту, поки не з'явиться свіжа інформація для передачі. Коли дані готові або закінчується час очікування, зазвичай 30-60 секунд, сервер відправляє відповідь, а клієнт миттєво створює новий запит, підтримуючи тим самим з'єднання.

Переваги “Long Polling” охоплюють універсальну підтримку будь-якими браузерами та серверами, оскільки застосовується стандартний протокол HTTP, відсутність потреби у спеціальних налаштуваннях брандмауерів та проксі-серверів, які здатні блокувати нестандартні протоколи, та легкість втілення на сервері з використанням асинхронних спроможностей будь-якої мови програмування.

Проте “Long Polling” має суттєві недоліки. Більш відчутним є висока затримка – навіть у оптимальних умовах затримка доставки повідомлення становить мінімум час виконання двох HTTP-запитів, що може сягати 200-500 мілісекунд. Також цей метод використовує неефективно ресурси – кожне з'єднання

споживає пам'ять та процесорний час сервера протягом всього часу очікування, що обмежує кількість одночасних користувачів. Крім того, виникають проблеми з масштабуванням – при використанні балансувальників навантаження можуть виникати труднощі з прив'язкою сесій, оскільки запити від одного користувача мають потрапляти на той самий сервер.

Прикладом його використання є раннє покоління веб-систем обміну повідомленнями до 2011 року, Facebook Messenger до впровадження WebSocket, деякі ігрові системи з низькою частотою повідомлень.

“Server-Sent Events” (SSE) – це стандартизований метод для односпрямованої передачі даних від сервера до клієнта. SSE застосовує звичайне HTTP-з'єднання, але зі специфічним типом контенту text/event-stream, який дає змогу серверу надсилати численні повідомлення через єдине довготривале з'єднання. Технічне виконання SSE полягає у тому, що програма в браузері дає команду створити спеціальний канал зв’язку з сервером. Сервер не закриває це з’єднання одразу, а тримає його активним і відправляє дані в текстовому форматі з префіксом data – кожне повідомлення розділяється подвійним переведенням рядка. Браузер автоматично обробляє ці повідомлення та миттєво передає їх у потрібну частину вашого коду для відображення на екрані.

Переваги SSE містять вбудовану підтримку більшістю браузерів без додаткових бібліотек, автоматичне поновлення з'єднання при обриві з механізмом повторних спроб та ідентифікатором останньої події для синхронізації, просту реалізацію серверної частини через ендпоінт HTTP з відповідними заголовками. Передача даних від сервера до клієнта відбувається з мінімальними затримками, якщо порівнювати з більш громіздким протоколом WebSocket.

Вади SSE криються у декількох обмеженнях. Ліміт браузерів на шість одночасних з’єднань із доменом, що ускладнює роботу на багатьох вкладках одночасно. Оскільки дані йдуть лише від сервера, для зворотного зв’язку користувачу доводиться постійно відправляти POST-запити. SSE передає тільки тексти, через що двійкові коди потребують додаткового кодування. Окрім того,

виникають затримки на рівні проксі-серверів, якщо вони налаштовані на буферизацію трафіку.

Найчастіше SSE використовують де йде постійний потік інформації в один бік – це оновлення котирувань акцій на фінансових порталах, стрічки новин з автоматичним оновленням, сповіщення про події у веб-застосунках та панелі моніторингу технічних даних.

“Вебсокети” являють собою повноцінний двоспрямований зв'язок обміну миттєвими повідомленнями на основі TCP. На відміну від HTTP, після встановлення з’єднання обидві сторони можуть відсилати повідомлення одна одній у будь-який момент без додаткових запитів.

З'єднання починається з HTTP-запиту на зміну протоколу, де клієнт вказує заголовки переходу протоколу та типу сполучення. Після отримання від сервера підтвердження з кодом 101, відбувається перехід до стабільного TCP-з'єднання. Це надає суттєву економію трафіку, оскільки витрати на кожен кадр становлять лише 2–6 байтів, тоді як звичні HTTP-заголовки вимагають передачі сотень байтів відомостей.

“WebSocket” надає справжній двосторонній зв'язок, завдяки якому і клієнт, і сервер можуть розпочинати передачу даних у будь-який момент без очікування запиту. Після встановлення зв'язку затримка доставляння становить лише час мережевого пересилання, що зазвичай не перевищує 50 мілісекунд. Відсутність HTTP-заголовків для кожного повідомлення значно зменшує трафік при високій частоті обміну даними. Протокол підтримує як текстові (UTF-8), так і бінарні повідомлення без додаткового кодування. Завдяки подієво-орієнтованій архітектурі один сервер здатний обслуговувати десятки тисяч одночасних з'єднань [5].

Недоліком “WebSocket” є складність реалізації асинхронного вводу-виводу на сервері, можливе блокування з'єднань корпоративними брандмауерами, відсутність автоматичного перепідключення та необхідність резервних механізмів для сумісності зі старими браузерами.

Сфера використання протоколу обміну даними в реальному часі охоплює програмні продукти, де важлива швидкість обміну даними. Зокрема, це стосується комунікаційних месенджерів, онлайн-ігор із високим ступенем інтерактивності та систем колективної роботи над документами. Технологія також забезпечує стабільне функціонування торговельних майданчиків із динамічними показниками ринку та комплексних рішень для передачі відео та аудіосигналів. Порівняльний аналіз протоколів наведено у табл. 1.1.

Таблиця 1.1.

Порівняння протоколів real-time комунікації

Характеристика	Long Polling	SSE	WebSocket
Latency	200-500 мс	50-150 мс	< 50 мс
Двосторонній зв'язок	Ні (емуляція)	Ні	Так
Підтримка браузерів	Всі	Всі крім IE	Всі сучасні
Складність реалізації	Низька	Середня	Висока
Ефективність трафіку	Низька	Середня	Висока
Масштабованість	Низька	Середня	Висока
Одночасні з'єднання	Обмежені сервером	6 на домен	Десятки тисяч

1.3.2 Обґрунтування вибору технології SignalR для реалізації двонаправленого зв'язку

SignalR – це бібліотека від Microsoft для ASP.NET Core, що реалізує абстракцію над протоколами реального часу [6]. Замість того щоб робити вибір між WebSocket, SSE та Long Polling вручну, бібліотека автоматично обирає найкращий доступний транспорт для конкретного клієнта та мережевого середовища. При цьому розробник пише єдиний код як на сервері, так і на клієнтській частині – система сама перемикається між транспортами прозоро.

З технічної точки зору бібліотека складається з двох частин: серверного Hub-класу і клієнтського пакету для TypeScript/Angular [7]. Через хаб клієнт отримує

змогу звертатись до серверної логіки напряму, тоді як сервер, у свою чергу, може у будь-який момент надіслати запит без очікування. Всі деталі протоколу і управління з'єднанням приховані всередині бібліотеки.

Для цього проєкту було обрано SignalR, оскільки він має важливі переваги. Бо є офіційною частиною платформи ASP.NET Core і дає змогу оминути встановлення зайвих сторонніх програм. Важливою рисою також є вмонтована функція самостійного поновлення зв'язку: якщо з'єднання обривається, система автоматично відновлює його без участі користувача чи додаткового коду. До того ж ця технологія виділяється своєю стійкістю у сполучених мережах: якщо захист підприємства блокує головний канал зв'язку, система автоматично обирає запасний спосіб передачі відомостей, щоб застосунок продовжував функціонувати без перешкод. Також принцип роботи з групами суттєво спрощує розповсюдження сповіщень, дозволяє надсилати повідомлення одночасно усім членам чату без ручного керування переліком з'єднань.

Порівняно з прямим використанням WebSocket, SignalR суттєво зменшує обсяг інфраструктурного коду: управління списком з'єднань, серіалізація повідомлень, heartbeat, обробка розривів – все це вирішено всередині бібліотеки. Що дозволяє більше зосередитись на бізнес-логіці, шифруванні, маршрутизації та обробці повідомлень.

1.4 Обґрунтування вибору технологічного стеку

Вибір технологічного стеку є важливим рішенням, що впливає на всі аспекти розробки, підтримки та масштабування застосунку. Для реалізації системи обміну миттєвими повідомленнями було обрано комбінацію:

- Angular для клієнтської частини;
- ASP.NET Core для серверної логіки;
- гібридний підхід до зберігання даних з використанням SQL для структурованих даних.

Angular підходить завдяки повній статичній типізації, вбудованій підтримці маршрутизації та взаємодії з програмним інтерфейсом без сторонніх бібліотек, а також бібліотеці реактивного програмування [8], що ідеально підходить для обробки асинхронних подій у месенджері [9]. Компонентна архітектура фреймворку забезпечує чітке розмежування відповідальності між елементами інтерфейсу та спрощує підтримку коду.

ASP.NET Core [10] обрано через вбудовану інтеграцію з SignalR, строгу типізацію C# та стабільність Entity Framework Core [11] для роботи з реляційною базою даних. Висока продуктивність при обробці великої кількості одночасних з'єднань є важлимим показником для систем.

Реляційна модель була обрана через те, що гарантує цілісність кожної операції, щоб жодне повідомлення не зникло під час відправки. Такі бази даних дозволяють гнучко поєднувати інформацію з різних таблиць через JOIN-запити та суворо контролювати структуру даних за допомогою зовнішніх ключів [12]. Ці властивості є дуже важливими для коректного збереження історії повідомлень та криптографічних параметрів кожної сесії.

Наявні засоби також суттєво відрізняються за технологічною базою. WhatsApp та Signal застосовують вбудовані мобільні програми. Це дає їм змогу використовувати апаратне шифрування та працювати швидше, проте звужує можливості працювати з різних пристроїв. Telegram пропонує як власні, так і веб-клієнти, зберігаючи повну функціональність у браузері. Розроблений застосунок є веб-орієнтованим SPA. Головний плюс тут у тому, що користувачу не потрібно нічого завантажувати, захищене спілкування доступне прямо в браузері. Такий підхід значно спрощує розгортання системи в корпоративних мережах та дозволяє швидко налагодити зв’язок у разі потреби. Зіставлення головних ознак показано у таблиці 1.2. Таблиця демонструє позиціонування створеного засобу стосовно аналогів за показниками приватності, збирання службових даних, виду клієнта та необхідності E2EE. Поєднання обраних інструментів дозволяє розгорнути повноцінне SPA-рішення, яке забезпечує належний рівень безпеки та швидкості, залишається максимально доступним для кінцевого користувача.

Таблиця 1.2

Порівняння характеристик систем обміну повідомленнями

Характеристика	WhatsApp	Telegram	Signal	Розроблена система
Анонімний доступ	Ні	Ні	Ні	Так
Збір метаданих	Так	Так	Мінімально	Ні
Тип клієнта	Моб. додаток	Моб. додаток + Браузерний	Моб. додаток	Браузерний (SPA)
Обов'язкове E2EE	Так	Лише секретні чати	Так	Так

1.5 Аналіз механізмів захисту та наскрізного шифрування

У сучасному цифровому світі, де кіберзагрози постійно еволюціонують, а інциденти з витоком конфіденційної інформації стають дедалі частішими, забезпечення приватності комунікацій набуває особливої ваги. Головним пріоритетом системи є збереження приватності користувачів, що реалізується через механізм End-to-End шифрування. Це найнадійніший підхід, який нівелює ризики, пов’язані з можливою компрометацією серверної сторони. Оскільки ключі зберігаються лише у відправника та отримувача, вміст повідомлень залишається недоступним для сторонніх осіб за будь-яких умов.

1.5.1 Принципи наскрізного шифрування

Наскрізне шифрування – це метод захисту даних, при якому повідомлення шифруються на пристрої відправника та дешифруються виключно на пристрої отримувача. Важливою характеристикою E2EE є те, що проміжні вузли, включаючи сервер застосунку, не мають доступу до ключів дешифрування і, відповідно, не можуть прочитати зміст повідомлень.

“Нульове Знання” – сервер працює за принципом нульового знання (Zero-Knowledge Principle), що унеможливлює технічний доступ до відкритого вмісту послань. Навіть у ситуації зловмисного втручання у серверне обладнання, хакерського нападу чи судового запиту про надання відомостей, оператор системи фізично не спроможний передати нешифровані повідомлення, адже він їх просто не має.

“Дані в Русі” (інформація під час трансферу) – це захист відомостей під час їх руху мережею. Звичний метод задіює протокол TLS/SSL, який шифрує злуку між клієнтом та сервером. Однак TLS оберігає лише шлях сполучення, тобто на сервері дані розшифровуються, обробляються й знову шифруються для пересилання наступному адресату. Це означає, що серверу доступний відкритий зміст.

“Дані в Триманні” (інформація у стані збереження) – це захист інформації, яка зберігається на сервері або пристрої користувача. Навіть, якщо сховище даних тримає сповіщення у зашифрованому вигляді, сервер володіє ключем для дешифрування і може здобути доступ до вмісту.

E2EE забезпечує захист повідомлення на всіх етапах його існування – від моменту створення на пристрої відправника до моменту прочитання на пристрої отримувача. Сервер зберігає та передає лише зашифровані дані, які він не може розшифрувати. Порівняння підходів до шифрування наведено у табл. 1.3.

Таблиця 1.3

Порівняння TLS та E2E шифрування

Характеристика	TLS/SSL шифрування	E2E шифрування
Хто має доступ до ключів	Сервер (посередник)	Лише відправник та отримувач
Захист від компрометації сервера	Ні (дані на сервері доступні)	Так (дані на сервері зашифровані)
Можливість читання оператором	Так	Ні

Продовження таблиці 1.3.

Характеристика	TLS/SSL шифрування	E2E шифрування
Відповідність на судові запити	Можливе надання даних	Технічно неможливо
Захист від MitM атак	Так (на рівні з'єднання)	Так (на рівні контенту)
Складність реалізації	Низька (стандарт для вебу)	Висока (потребує обміну ключами)

В рамках представленої праці створено модель E2EE на базі симетричного головного ключа, який зберігається винятково на стороні клієнта та ніколи не надсилається на сервер. Для кожного окремого повідомлення з головного ключа за допомогою функції деривації PBKDF2 вираховується неповторний сесійний ключ із криптографічно випадковими параметрами Salt та IV. Подібний метод забезпечує ключову характеристику наскрізного шифрування, де серверна інфраструктура оперує лише зашифрованими відомостями та не має технічної можливості відновлювати зміст повідомлень. Найважливішим напрямом подальшого вдосконалення системи визначено розширення протоколу до повного асиметричного обміну ключами за схемою “Diffie-Hellman”. Окрім технологічних переваг, E2EE гарантує дотримання нормативних приписів щодо охорони особистих даних, зокрема GDPR [13], що є європейським положенням про захист особистих даних, та HIPAA, який встановлює американський еталон захисту медичної інформації. Виконання цих вимог є необхідним для систем, що обробляють конфіденційні відомості користувачів у медичній, юридичній та бізнесовій галузях, адже нехтування нормативними положеннями може спричинити штрафні стягнення та втрату довіри.

Порівняння переваг та викликів E2EE наведено у табл. 1.4.

Таблиця 1.4

Переваги та виклики E2EE

Переваги	Виклики
Навіть оператор сервісу не має доступу до контенту повідомлень	Технічна складність реалізації та підтримки
Унеможливлює масове збирання даних без відома користувача	Повнотекстовий пошук можливий лише локально на пристрої
Відповідність регуляторним вимогам (GDPR, HIPAA)	Неможливість автоматичної модерації контенту
Прозорість щодо неможливості доступу третіх осіб	Складність відновлення доступу при втраті пристрою

1.5.2 Криптографічні алгоритми системи: AES-256-CBC та PBKDF2

Аби здійснити наскрізне шифрування у системі обміну повідомленнями, було обрано симетричний алгоритм шифрування AES-256 (Advanced Encryption Standard із 256-бітним ключем) у режимі CBC (Cipher Block Chaining) із застосуванням динамічних параметрів: вектора ініціалізації (IV) та солі (Salt) у зв'язці з функцією виведення ключа (KDF – Key Derivation Function).

Advanced Encryption Standard є симетричним блоковим шифром, затвердженим Національним інститутом стандартів і технологій США у 2001 році. AES-256 застосовує 256-бітний ключ, що гарантує виняткову криптографічну надійність.

AES-256 вважається криптографічно стійким проти всіх відомих атак, кількість можливих ключів становить 2^{256}, що робить “brute force” практично нездійсненним навіть для суперкомп'ютерів. Алгоритм схвалений NIST і використовується у протоколах TLS, WPA2/WPA3 та месенджерах Signal і WhatsApp [14]. Апаратна підтримка AES-NI у сучасних процесорах забезпечує високу швидкість без суттєвого навантаження.

У режимі CBC кожен новий фрагмент тексту комбінується з попереднім зашифрованим блоком, що унеможливлює виявлення повторюваних шаблонів.

CBC використовує вектор ініціалізації (IV) завдяки йому навіть однакові повідомлення зашифровані одним ключем мають різний шифротекст.

IV – це випадкове 128-бітне значення, унікальне для кожного повідомлення. Воно передається разом із шифротекстом у відкритому вигляді, без нього розшифрування неможливе, але знання IV без ключа не дає зловмиснику жодної переваги.

Хоча AES-256 вимагає 256-бітний ключ, пряме застосування головного ключа для шифрування кожного повідомлення створює загрозу витоку при повторному використанні однакових параметрів. Тому, щоб унеможливити відновлення майстер-ключа навіть при викритті окремого сесійного ключа, для кожного повідомлення з майстер-ключа через функцію виведення PBKDF2 обчислюється неповторний сесійний ключ із криптографічно випадковим Salt[15].

PBKDF2 виводить неповторний сесійний ключ з майстер-ключа та довільно згенерованого Salt шляхом багаторазового хешування. Неповторний Salt для кожного повідомлення унеможливлює використання “райдужних таблиць”, а велика кількість циклів робить метод повного перебору економічно недоцільним. Кількість циклів PBKDF2 у впровадженій системі встановлено на рівні 10 000, що відповідає мінімальним рекомендаціям OWASP для захисту паролів. Збільшення кількості циклів підвищує захист проте збільшує час виведення ключа. Архітектура застосунку базується на раціональному поєднанні продуктивності з вимогами безпеки. Використання індивідуальної солі (Salt) та векторів ініціалізації (IV) для кожного меседжу гарантує, що витік певного ключа чи даних не призведе до повного зламу переписки, оскільки наслідки такого інциденту будуть мінімальними.

Хоча асиметричне шифрування (RSA, ECC) також застосовується у E2EE системах для обміну ключами, для прямого шифрування вмісту віддають перевагу симетричному алгоритму. Шифрування великого повідомлення чи файлу за допомогою RSA було б неприйнятно повільним, тому обрано симетричне шифрування, яке значно швидше за асиметричне. AES-256 додає лише мінімальне навантаження на обробку даних. Симетричні алгоритми також вимагають значно

менше обчислювальних ресурсів, що гарантує високу швидкість шифрування навіть при значному завантаженні. Популярність стандарту призвела до того, що розробники впровадили підтримку шифрування безпосередньо в архітектуру процесорів. Завдяки цьому швидкість обробки даних залишається високою, а користувач не помічає жодних затримок під час обміну повідомленнями.

Оптимальним варіантом для застосунку є гібридна модель шифрування. Завдяки якому асиметричне шифрування відповідає лише за безпечну передачу сесійних ключів за протоколом “Diffie-Hellman”, після чого весь контент шифрується швидким симетричним алгоритмом AES-256. Саме такий підхід поєднує криптографічну надійність асиметричних методів із продуктивністю симетричного шифрування.

Розроблене рішення забезпечує базовий рівень наскрізного захисту, сервер отримує і зберігає лише зашифровані дані, не маючи технічної можливості відновити відкритий текст без мастер-ключа, що зберігається виключно у користувача.

Проведений аналіз предметної області допоміг сформувати необхідну базу для розробки захищеного сервісу обміну повідомленнями. Визначено функціональні та нефункціональні вимоги, обґрунтовано вибір протоколу SignalR на основі порівняння альтернативних підходів, а також сформовано технологічний стек, що найповніше відповідає вимогам безпеки, продуктивності та масштабованості системи.

2 ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Розробка архітектури програмного застосунку

Важливим етапом розробки є проєктування архітектури програмного застосунку , що визначає структуру системи, взаємодію її компонентів та можливості подальшого масштабування. Для системи обміну миттєвими повідомленнями з наскрізним шифруванням обрано трирівневу архітектуру, що забезпечує чітке розділення відповідальностей між шарами презентації, бізнес-логіки та доступу до даних.

2.1.1 Проєктування клієнт-серверної взаємодії

Архітектура системи обміну повідомленнями побудована за принципом трирівневої моделі, яка забезпечує чітке розмежування відповідальності між шарами та полегшує масштабування і підтримку системи. Діаграма компонентів (рис. 2.1) демонструє взаємозв'язок основних архітектурних елементів.

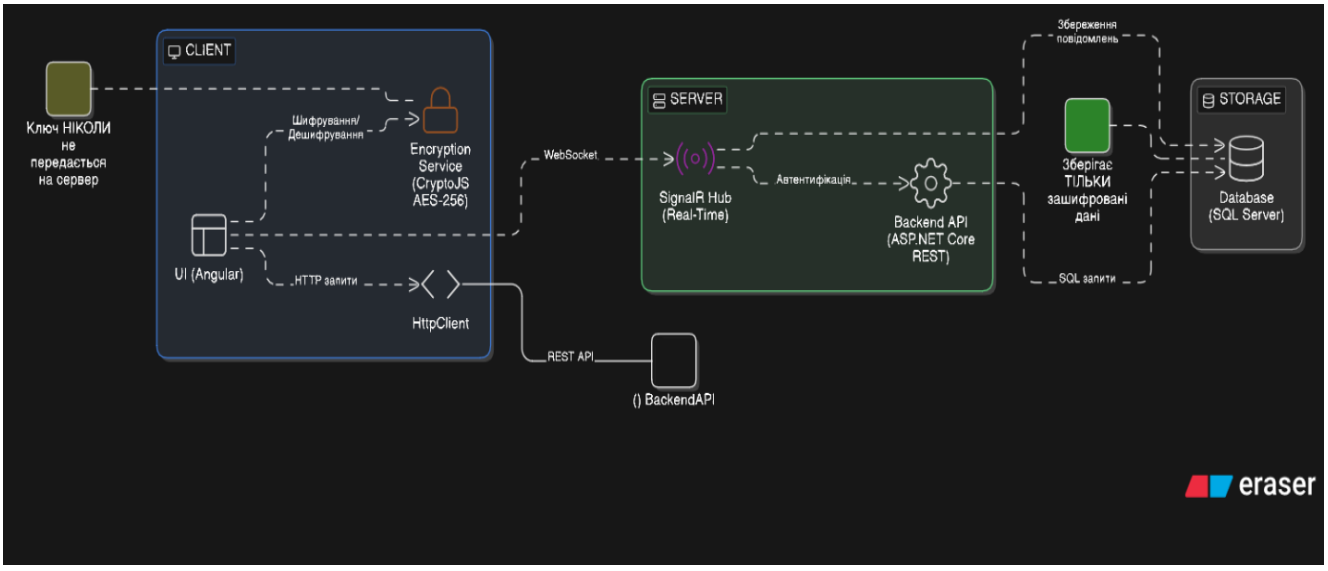


Рис. 2.1. Архітектура системи обміну повідомленнями

Як видно з діаграми, клієнтський шар (Angular) взаємодіє з серверним (ASP.NET Core) через два канали REST API для операцій запит-відповідь та SignalR для “real-time” комунікації.

Клієнтський шар є односторінковим застосунком, що функціонує у веб-браузері. Він відповідає за шифрування повідомлень перед відправкою та дешифрування при отриманні, мастер-ключ шифрування зберігається винятково в пам'яті браузера і ніколи не покидає пристрій користувача.

Серверний шар включає контролер для обробки HTTP-запитів (автентифікація, завантаження файлів), SignalR Hub для двосторонньої комунікації в реальному часі, а також сервіси бізнес-логіки для валідації та маршрутизації повідомлень. Шар даних відповідає за зберігання зашифрованих повідомлень та профілів у реляційній базі даних з гарантіями цілісності та надійності транзакцій.

Така архітектура забезпечує чітке розмежування відповідальностей між шарами. Для операцій запит-відповідь (автентифікація, завантаження історії) застосовується REST API. А для подій у реальному часі (надходження повідомлень, зміна статусів) - SignalR . Використання токенів автентифікації забезпечує роботу без збереження стану на сервері, тому сервер не зберігає сесії і може масштабуватися горизонтально без синхронізації стану між екземплярами.

2.1.2 Канали зв'язку між клієнтом та сервером

Система використовує два паралельні канали зв'язку. Для операцій запит-відповідь: автентифікація, реєстрація, завантаження історії повідомлень та файлів, застосовується HTTP-інтерфейс. SignalR задіюється тільки для миттєвих подій: доставка нових повідомлень, оновлення статусів онлайн/офлайн, підтвердження доставки та синхронізація прочитаних повідомлень. Такий розподіл знижує навантаження на з'єднання, він не перенавантажується даними історії, обробляючи лише події в реальному часі.

При підключенні користувача система зберігає ідентифікатор з'єднання та встановлює статус онлайн, сповіщаючи інших учасників. При від'єднанні статус

офлайн встановлюється лише після закриття останнього активного з'єднання, що дозволяє користувачу працювати одночасно з кількох пристроїв.

2.1.3 Моделювання функціональних вимог

Основні дії користувача в системі та їх зв'язки з обов'язковим шифруванням показано на діаграмі варіантів.

Система має двох акторів: користувача, який взаємодіє з інтерфейсом, та сервер, який відповідає за маршрутизацію повідомлень. Шифрування вбудоване в кожну дію з повідомленнями і не може бути вимкнене, коли користувач відправляє повідомлення, воно шифрується симетричним шифруванням у клієнта ще до відправки на сервер, а при завантаженні отримувачем розшифровується локально у одержувача, як відображено на рис. 2.2. Сервер на жодному з цих етапів не має доступу до відкритого тексту.

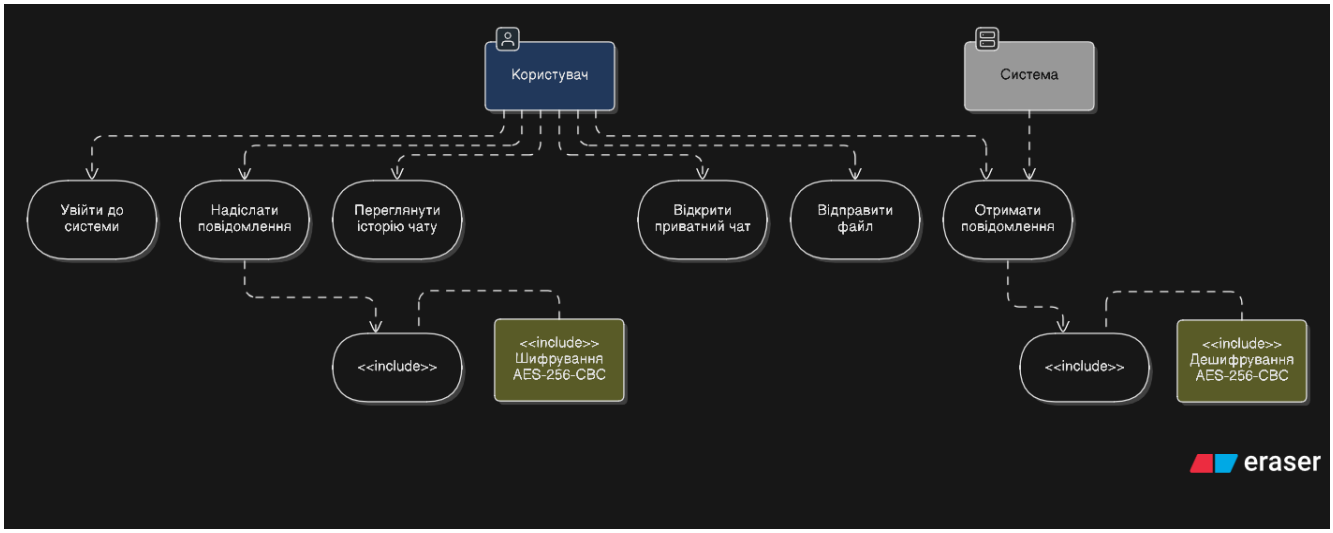


Рис. 2.2. Use Case – Шифрування повідомлень

Основні варіанти використання: вхід в систему, відправка та читання повідомлень у глобальному чаті, відкриття приватного чату з конкретним користувачем, відправка файлів, редагування і видалення власних повідомлень, відповідь на конкретне повідомлення.

2.2 Проєктування механізмів безпеки та обробки повідомлень

Безпека є центральним аспектом архітектури системи. В основу розробки покладено криптографічні методи та алгоритми, що регламентують порядок опрацювання повідомлень і гарантують високий рівень конфіденційності від моменту створення даних до їх отримання адресатом.

2.2.1 Розробка ER-моделі даних для сутностей

End-to-End шифрування є основною вимогою безпеки системи, яка гарантує конфіденційність повідомлень навіть у разі компрометації серверної інфраструктури. Шифрування та дешифрування виконуються винятково на пристроях клієнтів, сервер зберігає та маршрутизує лише зашифровані дані. Розроблена структура даних (рис. 2.3) спроєктована з урахуванням вимог наскрізного шифрування, коли сервер не має доступу до відкритого тексту повідомлень на жодному етапі їх обробки. Саме цей принцип є основою реалізованого криптографічного підходу.

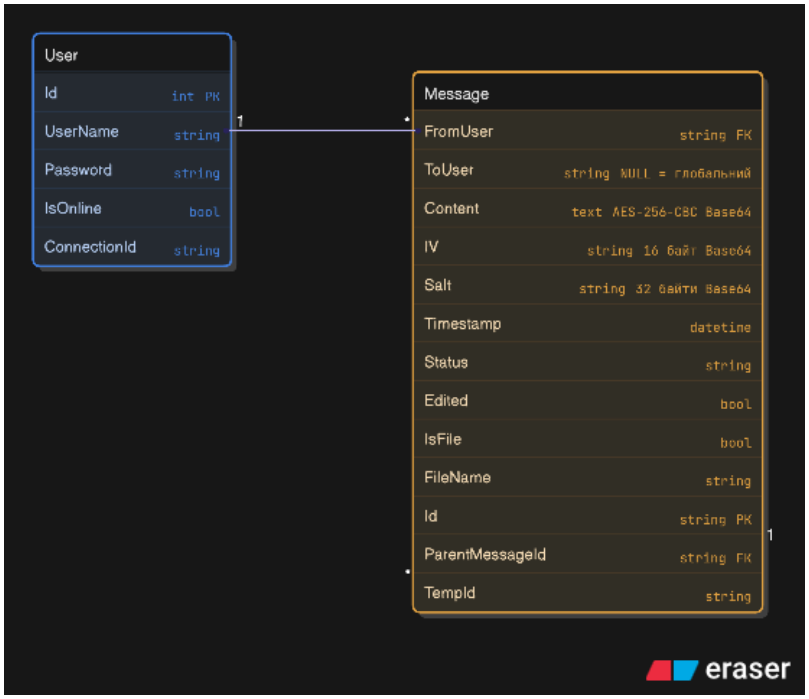


Рис. 2.3. ER-діаграма структури бази даних

Діаграма містить дві сутності: User та Message. Сутність User зберігає ім'я користувача, хешований пароль, статус онлайн та ідентифікатор SignalR з'єднання. Сутність Message містить зашифрований контент, вектор ініціалізації IV та Salt – поля необхідні для розшифрування виключно на пристрої користувача. Зв'язок між сутностями один-до-багатьох через поле відправника.

Криптографічний протокол базується на AES-256-CBC з динамічною генерацією ключів через PBKDF2, детальне обґрунтування вибору цих алгоритмів наведено в розділі 1.5.2. Ключова особливість полягає в тому, що сервер отримує і зберігає лише структуру, без мастер-ключа клієнта відновити відкритий текст технічно неможливо.

Для кожного повідомлення генеруються унікальні Salt та IV, що унеможливлює аналіз паттернів навіть при повторному шифруванні ідентичного тексту.

Формула шифрування:

$$C = AES_{256\text{-}CBC}(P, K_{session}, IV) \quad (2.1)$$

$$K_{session} = PBKDF2(K_{master}, Salt, 10000, 256), \quad (2.2)$$

де C – шифротекст (ciphertext); P – відкритий текст (plaintext); $K_{session}$ – сесійний ключ, який обчислюється за формулою (2.2); K_{master} – мастер-ключ користувача, що ніколи не покидає пристрій користувача; Salt – криптографічно випадкові 32 байти, унікальні для кожного повідомлення; IV – вектор ініціалізації 128 біт, також унікальний для кожного повідомлення.

Завдяки унікальним Salt та IV для кожного повідомлення навіть ідентичні тексти дають різний шифротекст, а безпека системи базується виключно на секретності K_master, який ніколи не покидає пристрій користувача.

Теорема: знання C, IV та Salt без знання K_master не дозволяє відновити P за поліноміальний час.

Доведення: складність повного перебору атаки на AES-256 становить 2^{256} операцій. Застосування методу повного перебору при потужності 10^{12} операцій на секунду потребувало б орієнтовно у $3{,}67 \times 10^{51}$ років обчислень. Ця величина

настільки перевищує будь-які реальні терміни, що дозволяє стверджувати про виняткову надійність обраного методу шифрування. Практичний злом AES-256 методом перебору є неможливим.

2.2.2 Процес шифрування та передачі повідомлення

Діаграма послідовності (рис. 2.4) деталізує покроковий процес надсилання повідомлення від клієнта A до клієнта B через серверну інфраструктуру.

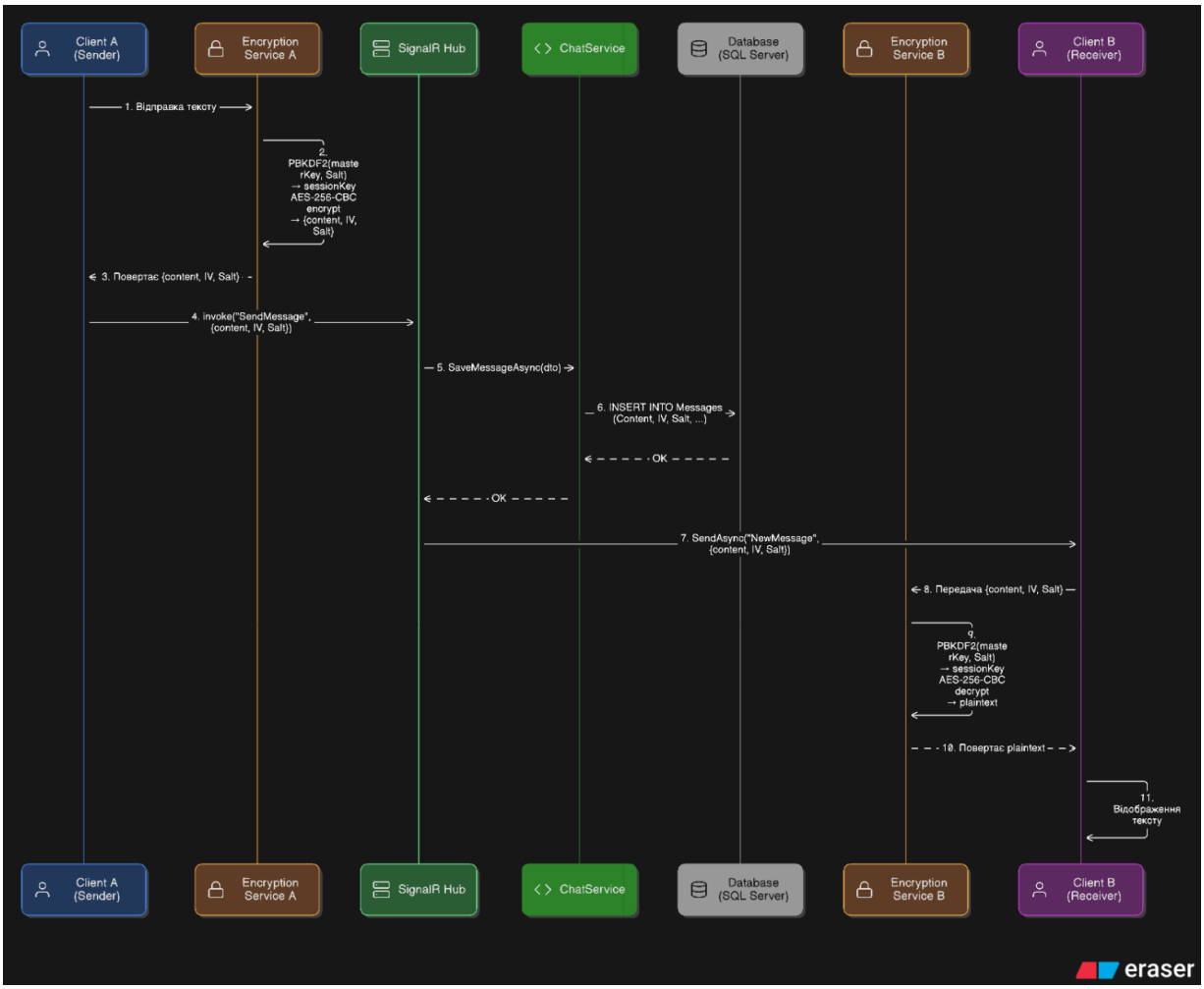


Рис. 2.4. E2EE Шифрування повідомлення

Діаграма наочно підтверджує цей принцип: відкритий текст існує виключно в пам'яті браузера відправника та отримувача, тоді як сервер на всіх проміжних етапах оперує лише зашифрованими даними. Повний лістинг ChatService з методами шифрування наведено у Додатку Б.2.1

2.2.3 Проєктування автентифікації та авторизації

Система безпеки застосунку базується на використанні веб-маркерів доступу для автентифікації програмного інтерфейсу та авторизації з'єднань реального часу. Такий підхід дозволяє серверу працювати автономно, не зберігаючи дані про стан сесій користувачів. Відсутність прив'язки до конкретного сервера (Session Persistence) робить інфраструктуру гнучкою та готовою до швидкого розширення шляхом додавання нових обчислювальних вузлів.

Веб-маркер доступу складається з трьох частин: заголовка, корисного навантаження та цифрового підпису, розділених крапками. Заголовок визначає алгоритм підпису. Корисне навантаження містить ідентифікатор користувача, його ім'я та час закінчення дії токена. Підпис є криптографічним підписом перших двох частин із серверним секретом, що унеможливлює підробку маркера.

Процес автентифікації, що передбачає використання відповідних запитів до програмного інтерфейсу реєстрації та входу, базується на єдиній логіці: якщо користувача з таким іменем не існує створюється новий запис; якщо існує порівнюється пароль. При успішному вході генерується веб-маркер доступу з терміном дії одна година та зберігається у захищеному сесійному файлі [16]. Вузол реального часу зчитує маркер при встановленні з'єднання та верифікує права доступу користувача.

Клієнт передає маркер доступу при встановленні з'єднання реального часу. Порівняння підходів до автентифікації наведено у табл. 2.1.

Таблиця 2.1.

Порівняння підходів до автентифікації

Підхід	Переваги	Недоліки	Використання в проєкті
Session-based (Cookies)	Легко інвалідувати на сервері	Складно масштабувати, CSRF вразливість	Ні
JWT (Stateless)	Легко масштабується, не вимагає серверного стану	Важко інвалідувати до закінчення терміну дії	Так (REST API та SignalR)
OpenID Connect	Делегована автентифікація через сторонні сервіси	Додаткова складність	Можлива інтеграція в майбутньому

Вибір веб-маркерів доступу зумовлений відсутністю необхідності зберігати стан сесії на сервері, що є важливим фактором для забезпечення горизонтального масштабування та коректної роботи системи реального часу у середовищі з кількома серверними вузлами. Для збереження структурованої інформації в архітектурі застосовано реляційну систему управління базами даних SQL Server.

2.3 Проєктування бази даних та сховища даних

Проектування структури бази даних є фундаментальним етапом, що визначає ефективність зберігання, швидкість доступу до даних та можливості масштабування застосунку. Для системи обміну повідомленнями, роботи з даними обрано реляційну базу даних SQL Server, що забезпечує надійне зберігання структурованих даних та інформації, які часто запитується.

2.3.1 Структура таблиць бази даних

Структура бази даних складається з двох сутностей. Сутність користувача зберігає ідентифікатор, ім'я, пароль, статус онлайн та ідентифікатор з'єднання для відстеження сесії реального часу. Сутність повідомлення містить базові атрибути ідентифікації, часову мітку створення, ідентифікатори відправника та отримувача. Ключовою особливістю є наявність трьох окремих криптографічних полів: зашифрований вміст у кодованому вигляді, вектор ініціалізації довжиною 24 символи та криптографічна сіль для деривації ключа довжиною 44 символи. Саме така структура забезпечує реалізацію принципу нульового знання – сервер зберігає лише зашифровані дані, не маючи технічної можливості відновити відкритий текст повідомлення. Математичне обґрунтування стійкості такого підходу до атак методом повного перебору наведено в розділі 2.2.1. Для прискорення вибірки повідомлень за відправником та отримувачем на відповідних полях створено індекси, що скорочує час запиту при великій кількості записів. Зв'язки та детальна конфігурація Entity Framework Core наведені у Додатку Б.1.5.

2.3.2 Схема зберігання даних про повідомлення та метадані

Файлові повідомлення позначаються спеціальним типом, при якому поле Content виступає в ролі вказівника на шлях до файлу в системному сховищі, а FileName слугує для збереження назви файлу. Використання тимчасового ідентифікатора дозволяє реалізувати механізм синхронізації при оптимістичному оновленні UI. Це забезпечує відображення повідомлення з префіксним ID, що підлягає заміні на сталий ідентифікатор після завершення транзакції в БД.

Робота з даними в месенджері підпорядковується життєвому циклу, який охоплює чотири операції. При створенні клієнт надсилає зашифрований контент з унікальними IV та Salt, сервер зберігає дані без розшифрування. Процес отримання історії передбачає повернення 100 останніх записів із сортуванням за часом. Важливою особливістю є те, що сервер передає повідомлення лише у зашифрованому вигляді, а їхнє перетворення у відкритий текст здійснюється локально у клієнтському застосунку.

Редагування передбачає перевірку авторства, відправник збігається з поточним користувачем, після чого клієнт перешифровує новий текст з новими IV та Salt, а сервер оновлює зашифрований контент разом з новими IV та Salt і позначає повідомлення як відредаговане. Видалення починається з верифікації авторства й передбачає повне вилучення запису з бази даних. Оскільки метод "м’якого видалення" не застосовується, інформація про повідомлення остаточно видаляється з цифрового реєстру.

Реалізація відповідей ґрунтується на використанні зовнішнього ключа, що вказує на ідентифікатор у тій самій таблиці Messages. Це дозволяє ефективно відтворювати структуру дерева повідомлень, уникаючи створення надлишкових таблиць у сховищі. Повна реалізація ChatService у Додатку Б.1.3.

2.4 Проєктування API та користувацького інтерфейсу

Проєктування інтерфейсу системи базується на принципі мінімальної складності для користувача, при використанні якого вся криптографічна робота відбувається у фоні без жодних додаткових дій з боку користувача. Інтерфейс складається з двох екранів: сторінки входу та головного екрану чату. Для стилізації використано Bootstrap з кастомною темою, адаптивний layout автоматично перебудовується під мобільні пристрої через Bootstrap Grid. Колірна схема інтерфейсу побудована на темній палітрі з акцентними елементами, що знижує навантаження на зір при тривалому використанні та відповідає сучасним вимогам до оформлення комунікаційних застосунків.

2.4.1 Розробка REST API-інтерфейсів

API системи побудовано за принципами REST. Ендпоінти діляться на публічні (реєстрація, вхід) та захищені, останні вимагають валідний JWT-токен у httpOnly cookie. Перелік доступних ендпоінтів наведено у табл. 2.2.

Таблиця 2.2.

Перелік REST API ендпоінтів

Метод	Ендпоінт	Авторизація	Опис
POST	/api/Chat/register	Публічний	Реєстрація або вхід, якщо ім'я нове, створюється акаунт.
POST	/api/Chat/login	Публічний	Перевірка паролю, повернення JWT у cookie.
POST	/api/Chat/logout	Захищений	Видалення JWT cookie (завершення сесії).
GET	/api/Chat/current-user	Захищений	Отримання даних поточного авторизованого користувача.
GET	/api/Chat/messages/global	Захищений	Отримання історії глобального чату (повідомлення зашифровані).

Продовження таблиці 2.2.

Метод	Ендпоінт	Авторизація	Опис
GET	/api/Chat/private/{otherUser}	Захищений	Отримання історії приватного листування з конкретним користувачем.
POST	/api/Chat/SendFile	Захищений	Завантаження та відправка файлу в приватний чат.
POST	/api/Chat/send-global-file	Захищений	Завантаження та відправка файлу в загальний чат.

При помилках валідації повертається HTTP 400 з описом конкретних порушень у полі errors. Всі захищені кінцеві точки повертають код 401 при відсутності або недійсності маркера доступу, що забезпечує єдину та передбачувану поведінку програмного інтерфейсу при несанкціонованих запитах.

Завантаження файлів реалізовано через multipart/form-data без обмеження розміру. Файли зберігаються на сервері з унікальними іменами, метадані записуються в базу даних. Ендпоінти GET повертають повідомлення виключно у зашифрованому вигляді, сервер на жодному етапі не має доступу до розшифрованого контенту. Для доступу до захищених кінцевих точок необхідно надати валідний JWT-токен, що зберігається в інкапсульованому httpOnly cookie з атрибутами Secure та SameSite=Strict. Такий підхід мінімізує ризики проведення XSS- та CSRF-атак, забезпечуючи високий рівень цілісності сесії.

2.4.2 Проєктування користувацького інтерфейсу (UX/UI)

Інтерфейс системи складається з двох екранів – екрану автентифікації та головного екрану чату. Графічний інтерфейс для великих екранів розділено на дві вертикальні зони. Ліва панель містить динамічний перелік онлайн-користувачів, а права виступає в ролі основного операційного простору, де зосереджено контейнер діалогів і панель введення даних. На мобільних пристроях Bootstrap Grid автоматично перебудовує layout у вертикальний.

Скролінг адаптований до поточної позиції перегляду, автоматичне переміщення до останніх записів виконується лише за умови перебування

користувача в нижній частині чату. Такий підхід запобігає примусовому перериванню читання історії повідомлень.

Інтерфейс підтримує контекстні дії з повідомленнями, такі як редагування, видалення та відповідь на конкретне повідомлення з відображенням цитати. Візуально розрізняються власні повідомлення користувача та повідомлення співрозмовників, а статуси доставки та прочитання відображаються в реальному часі без перезавантаження сторінки.

Проєктні рішення, представлені в роботі, стосуються всіх ключових компонентів вебсервісу від логіки збереження даних до засобів безпеки та інтерфейсу користувача. Визначено трирівневу архітектуру з чітким розмежуванням відповідальності між клієнтським, серверним та шаром даних. Розроблено криптографічний протокол на основі симетричного шифрування з динамічною генерацією параметрів через функцію деривації ключа, що унеможливлює аналіз шаблонів навіть при повторенні однакового тексту. У межах роботи було розроблено та впроваджено такі компоненти:

- реляційну модель бази даних з окремими криптографічними полями;
- механізми токенової автентифікації;
- програмний інтерфейс;
- компонентну структуру адаптивного інтерфейсу.

Розмежування між маршрутизацією повідомлень та їх шифруванням, як незалежними процесами, дозволяє розширювати криптографічний протокол без втручання в транспортну логіку системи, що в сукупності створює технічну основу для реалізації вимог щодо конфіденційності, продуктивності та надійності.

3 РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ

3.1 Реалізація серверної частини на ASP.NET Core

Серверна частина реалізована на базі ASP.NET Core та використовує дві точки входу: контролер для автентифікації та передачі файлів, і SignalR Hub для комунікації в реальному часі. Такий розподіл дозволяє чітко розмежувати операції запит-відповідь від подієво-орієнтованої “real-time” логіки, не перевантажуючи жоден з каналів зв'язку зайвими запитами.

Залежності між компонентами керуються через вбудований контейнер управління залежностями серверної платформи. Сервіси обробки повідомлень створюються окремо для кожного запиту, що унеможливлює витоки стану між різними користувачами. Сервіс автентифікації існує як єдиний екземпляр протягом усього часу роботи – це виправдано, оскільки він не зберігає даних конкретного користувача і може безпечно використовуватись одночасно з кількох потоків.

3.1.1 Розробка логіки SignalR Hubs для обробки та маршрутизації повідомлень

Вузол маршрутизації є центральним компонентом серверної архітектури, що забезпечує комунікацію між клієнтами в реальному часі через протокол двостороннього зв'язку. Головне завдання його – передача зашифрованих повідомлень без доступу до їх вмісту. Отримуючи повідомлення з технічними захисними мітками, він передає її далі без розшифрування.

При отриманні повідомлення Hub виконує валідацію відправника, генерує ідентифікатор та мітку часу, зберігає дані через сервіс обробки повідомлень та передає до отримувачів. Для глобального чату повідомлення доставляється всім учасникам групи, а для приватного – конкретному користувачу через динамічну групу. Вузол також обробляє події підключення та відключення для відстеження

онлайн-статусів, та підтримує редагування і видалення повідомлень автором та синхронізує статуси перегляду.

Повний лістинг ChatHub (основні методи Hub) у Додатку Б.1.2

3.1.2 Реалізація функціоналу обміну файлами та обробка винятків

Система підтримує завантаження файлів через HTTP-запити і одразу завантажує їх до чату. Файли зберігаються на сервері, а їхні метаданні фіксуються в реляційній базі даних.

Завантаження файлів реалізовано через два окремі канали: для глобального та приватного чату. Контролер виконує перевірку наявності файлу та автентифікацію користувача, після чого забезпечує збереження об’єкта на сервері під унікальним іменем. А такі метадані, як назва та тип, записує до бази даних. Повідомлення про файл доставляється через канал активного чату або конкретному користувачу залежно від типу розмови.

У разі відсутності файлу або надання невалідних даних сервіс генерує відповідь 400 “Bad Request”. Якщо ж автентифікація користувача не підтверджена, повертається статус 401 “Unauthorized”, тоді як виникнення непередбачених помилок супроводжується кодом 500 “Internal Server Error” із обов’язковим фіксуванням детальної інформації про інцидент. Приклад відображення файлу у чаті та інтерфейс вибору файлу перед відправкою наведено на рис. 3.1 та рис. 3.2.

Рис. 3.1. Відображення завантаженого файлу у чаті

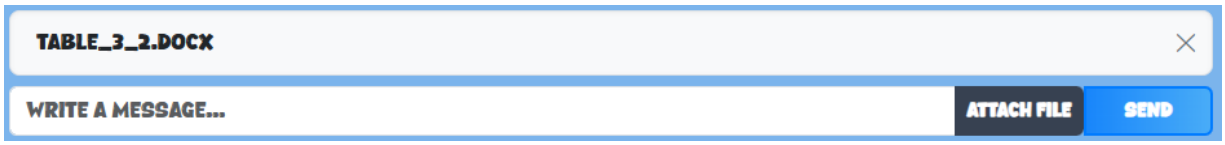


Рис. 3.2. Вибір файлу для відправки у чат

3.2 Реалізація клієнтської частини на Angular

Центральний сервіс відповідає за підключення до вузла реального часу та управління станом розмов. Усі асинхронні операції реалізовані засобами RxJS – бібліотеки реактивного програмування, яка перетворює події протоколу реального часу на потоки даних. Такий підхід дозволяє описувати логіку обробки повідомлень декларативно, без необхідності вручну керувати потоками даних у кожному компоненті.

Компоненти підписуються на потоки даних безпосередньо у шаблонах за допомогою вбудованого механізму “async pipe”, який автоматично керує життєвим циклом підписки. Це гарантує синхронне оновлення інтерфейсу при надходженні нових повідомлень від сервера. Після знищення компонента підписка розривається автоматично, що унеможливлює втрату пам'яті та забезпечує стабільну роботу додатка при тривалих сесіях.

3.2.1 Створення сервісів для взаємодії з REST API та SignalR

Клієнтська частина побудована на фреймворку Angular з використанням бібліотеки RxJS для реактивної обробки подій та CryptoJS для наскрізного шифрування [17].

Центральний сервіс клієнтської частини містить логіку підключення до вузла реального часу та координує управління станом активних розмов. Усі вхідні події від сервера перетворюються на потоки даних, на які компоненти підписуються безпосередньо у шаблонах. Завдяки цьому інтерфейс оновлюється одразу при надходженні нових даних, а після знищення компонента підписка розривається автоматично без додаткового коду.

Сервіс автентифікації обробляє два сценарії: вхід існуючого користувача та створення нового облікового запису. Якщо ім'я користувача відсутнє в базі даних, то запис створюється автоматично, після чого система одразу авторизує користувача без повторного введення даних. Інтерфейс форми автентифікації з

підтримкою автоматичної реєстрації наведено на рис. 3.3. Поточний стан сесії передається як потік даних до сервісу обміну повідомленнями, при успішному вході з'єднання з вузлом реального часу встановлюється автоматично, а при виході з'єднання розривається.

Шифрування реалізовано відповідно до криптографічного протоколу, спроєктованого в розділі 2.2.1. Для кожного повідомлення генеруються унікальні вектор ініціалізації та криптографічна сіль, що унеможливлює аналіз шаблонів навіть при повторенні однакового тексту. Для сумісності з повідомленнями, створеними до впровадження динамічних параметрів, реалізовано режим зі статичним ключем при відсутності IV та Salt – це обмеження поточної версії, заплановане до усунення у наступних релізах. Повідомлення без динамічних параметрів вразливі до атак на основі аналізу шаблонів, оскільки однаковий текст дає однаковий шифротекст при статичному IV.

Повідомлення від сервера перетворюються на потоки даних, на які компоненти підписуються безпосередньо у шаблонах – це забезпечує оновлення інтерфейсу при надходженні нових даних без явних команд та без втрати пам'яті. Повний лістинг ChatService + EncryptionService у Додатку Б.2.1

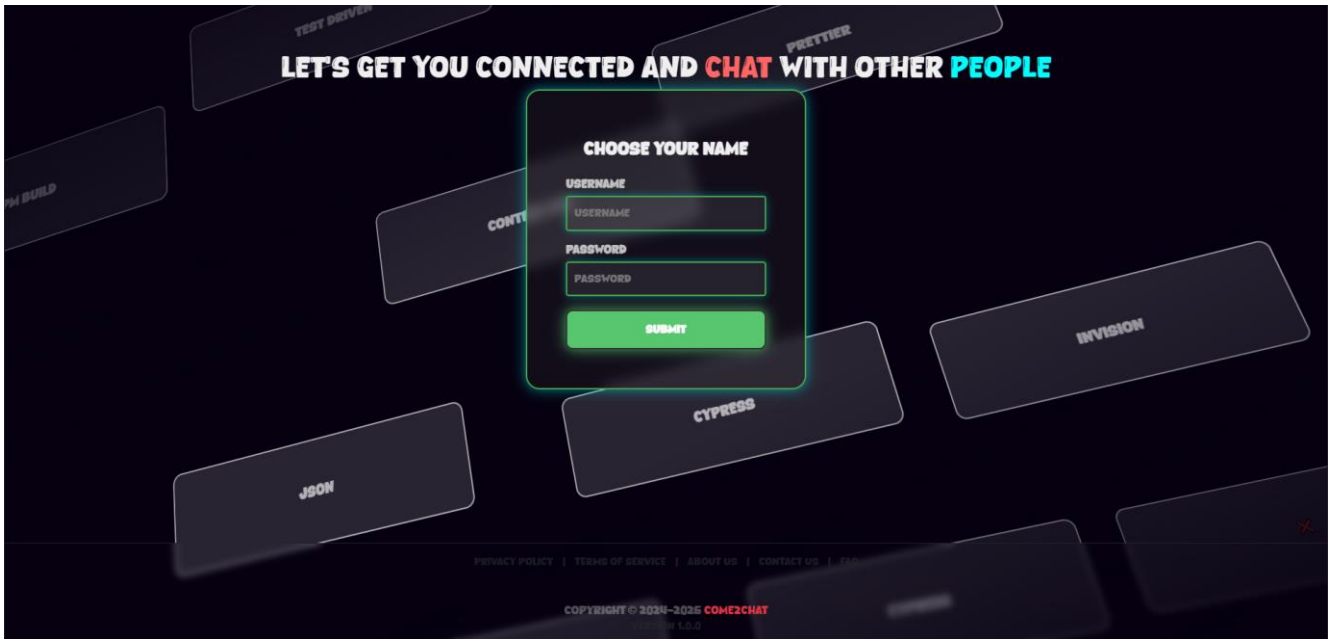


Рис. 3.3. Форма автентифікації та автоматичної реєстрації користувача

3.2.2 Компонент відображення повідомлень, відповідей та редагування

Архітектура інтерфейсу розділяє компоненти на керуючі, які відповідають за дані і логіку та відображувальні, що забезпечує продуктивність при відображенні великої історії повідомлень.

Стратегія виявлення змін “лише при явному оновленні” застосована для компонента анімованого фону, оскільки він оновлюється виключно через механізм анімаційних кадрів браузера, а не через стандартний цикл перевірки стану.

Головний компонент чату реалізує інтелектуальну систему скролінгу через вбудовані засоби DOM без CDK. Система відстежує позицію скролінгу, у випадку, якщо користувач відхилився від низу більш ніж на 50 пікселів, з'являється кнопка прокручування і вимикається автоскрол. При надходженні нового повідомлення автоскрол спрацьовує лише якщо користувач перебуває внизу чату, що дозволяє читати старі повідомлення не втрачаючи позицію. Повідомлення позначається як прочитане лише після того як воно реально відображено на екрані, а не просто отримано.

Функціонал редагування/видалення та відповідей реалізовано з умовною логікою. Кнопки редагування та видалення відображаються лише автору повідомлення. Функціонал відповідей (Reply) здійснюється шляхом пошуку батьківського повідомлення та відображення стислого прев'ю. Зовнішній вигляд інтерфейсу чату з функціями редагування та відповіді наведено на рис. 3.4 та рис. 3.5. Повні компоненти у Додатку Б.2.2

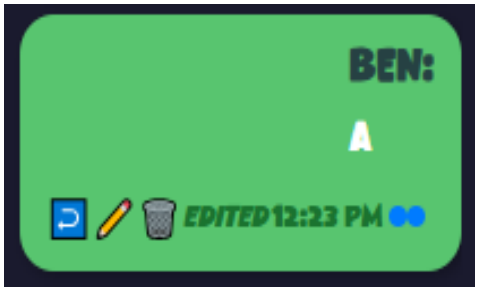


Рис. 3.4. Інтерфейс чату із кнопками редагування та видалення повідомлення

Рис. 3.5. Функція відповіді на повідомлення (Reply) з відображенням прев'ю

3.2.3 Реалізація анімованого фону на WebGL

Для реалізації анімованого фону використано бібліотеку OGL. Завдяки своїй мінімалістичності, цей інструмент дозволяє ефективно взаємодіяти з WebGL на низькому рівні, мінімізуючи навантаження на систему [18]. OGL не містить зайвих абстракцій, на відміну від повнофункціональних бібліотек на зразок Three.js, що дозволяє зменшити розмір бандлу та уникнути накладних витрат на рендеринг інтерфейсу. Даний компонент використовує WebGL2-рендерер з кастомними GLSL-шейдерами, який містить програму, що визначає форму об'єктів у просторі, і програму для пікселів, яка генерує ефект “північного сяйва” через алгоритм шумової генерації з анімацією, прив'язаною до плину часу. Для кожного кадру використовується стандартний механізм анімаційних кадрів браузера, що забезпечує плавну анімацію без блокування основного потоку інтерфейсу. Візуальний результат роботи WebGL-компонента з ефектом "північного сяйва" наведено на рис. 3.6.

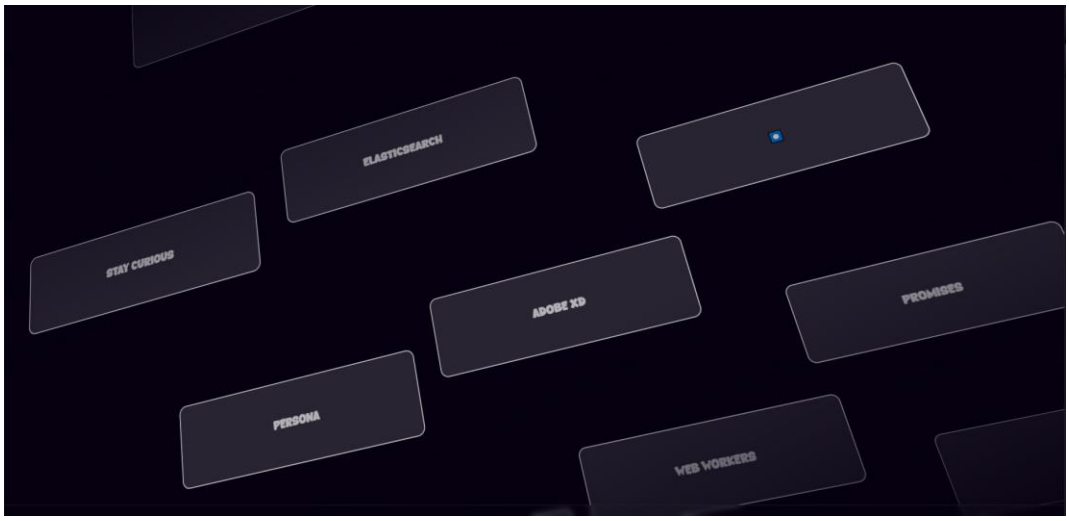


Рис. 3.6. Скріншот екрану входу або головного екрану із самим анімованим фоном (ефект "північного сяйва")

3.3 Тестування програмного застосунку

Тестування системи проводилось у три етапи відповідно до рівнів архітектури: юніт-тестування криптографічного модуля шифрування, інтеграційне тестування взаємодії клієнта з серверним API і навантажувальне тестування продуктивності інтерфейсу. Використано автоматизований фреймворк тестування з підтримкою статичної типізації для юніт-тестування та інтеграційного тестування. На кожному етапі тестування проводилася верифікація окремих аспектів розробки, зокрема оцінювалася коректність криптографічних функцій, захищеність API та відповідність нефункціональним вимогам щодо стабільності частоти кадрів і мінімізації затримок доставки повідомлень.

3.3.1 Юніт-тестування криптографічного модуля

Модульне тестування реалізовано у вигляді ізольованих тестових сценаріїв, які перевіряють роботу криптографічних функцій без залежності від сервера чи Angular-середовища. Тести охоплюють п'ять груп функцій.

Перша група перевіряє функцію encryptMessage, що відповідає за шифрування повідомлення.

Друга група тестує функцію decryptMessage, яка виконує зворотне перетворення – дешифрування зашифрованого тексту.

Третя група охоплює функцію deriveSessionKey, що генерує унікальний сесійний ключ на основі мастер-ключа та криптографічної солі за алгоритмом PBKDF2.

Четверта група перевіряє функцію createUserHash, яка створює хеш ідентифікатора користувача для безпечної ідентифікації в системі.

П'ята група тестує допоміжну функцію isBase64, що перевіряє відповідність рядка формату Base64 – це дозволяє відрізнити зашифровані дані від відкритого тексту.

Функція “encryptMessage” перевіряє, щоб шифротекст відрізнявся від відкритого тексту, є правильним форматом, містить коректні розміри IV (24 символи, 16 байт) та Salt (44 символи, 32 байти), а також щоб однаковий відкритий текст при кожному виклику давав різний шифротекст завдяки випадковій генерації IV та Salt.

У межах тестів функції “decryptMessage” підтверджено коректність round-trip перетворення та збереження Unicode-символів після дешифрування. Також перевірено поведінку при помилкових вхідних даних – функція коректно генерує виняток при отриманні невірного ключа або пошкоджених вхідних даних. Додатково перевірено режим сумісності зі старими повідомленнями без Salt через fallback на статичний ключ.

Група “deriveSessionKey” підтверджує, що функція “PBKDF2” повертає 256-бітний ключ, є детермінованою – однакові вхідні дані дають однаковий ключ та унікальною – різні Salt або різні мастер-ключі дають різні похідні ключі.

Група “createUserHash” перевіряє, якщо однаковий ідентифікатор в межах однієї секунди дає однаковий хеш, тоді як різні ідентифікатори або різні секретні ключі дають різні результати. Група “isBase64” підтверджує, що зашифрований текст завжди відповідає формату “Base64”, а звичайний текст перевірку не проходить.

За результатами виконання всі 25 тестів пройшли успішно (рис. 3.7). Покриття коду склало 98.48% рядків та 100% функцій, що підтверджує що криптографічна логіка покрита тестами повністю (рис. 3.8). Отримані результати підтверджують успішну перевірку тестового набору та високі показники покриття коду, які свідчать про належну якість криптографічного ядра застосунку. Проведене тестування підтвердило, що система готова для безпечного розгортання клієнтської частини застосунку, ризики виникнення прихованих слабких місць чи збоїв під час реального обміну повідомленнями мінімальні. Перелік тест-кейсів наведено у табл. 3.1.

Таблиця 3.1.

Перелік юніт-тестів криптографічного модуля

№	Група	Назва тесту / Сценарій	Очікуваний результат	Статус
1	encryptMessage	Шифротекст відрізняється від відкритого тексту	Різні рядки	Пройдено
2	encryptMessage	Шифротекст є валідним Base64	true	Пройдено
3	encryptMessage	IV має розмір 24 символи (16 байт)	length = 24	Пройдено
4	encryptMessage	Salt має розмір 44 символи (32 байти)	length = 44	Пройдено
5	encryptMessage	Однаковий текст дає різний шифротекст	IV/Salt унікальні	Пройдено
6	decryptMessage	Round-trip decrypt(encrypt(x)) = x	Рядки збігаються	Пройдено
7	decryptMessage	Невірний ключ викидає виняток	throw Error	Пройдено
8	decryptMessage	Пошкоджений контент викидає виняток	throw Error	Пройдено
9	decryptMessage	Fallback без Salt (старі повідомлення)	Розшифровано	Пройдено
10	deriveSessionKey	Повертає 256-бітний ключ	sigBytes = 32	Пройдено
11	deriveSessionKey	Різні Salt – різні ключі	Різні рядки	Пройдено
12	createUserHash	Різні userId – різні хеші	Різні рядки	Пройдено
13	isBase64	Зашифрований текст проходить перевірку	true	Пройдено
14	encryptMessage	Unicode + спецсимволи encrypt/decrypt	Рядки збігаються	Пройдено
15	encryptMessage	Довге повідомлення (1000 символів)	Розшифровано	Пройдено
16	encryptMessage	Статичний IV – однаковий текст дає однаковий шифротекст	enc1 = enc2	Пройдено
17	createUserHash	Хеш детермінований в межах секунди	hash1 = hash2	Пройдено
18	createUserHash	Хеш є валідним Base64 довжиною > 20 символів	length > 20	Пройдено
19	deriveSessionKey	Унікальний Salt – унікальний сесійний ключ	Ключі різні	Пройдено
20	encryptMessage	Шифротекст з динамічним IV унікальний	Різні рядки	Пройдено
21	decryptMessage	Розшифрування порожнього рядка не падає	"" або Error	Пройдено
22	isBase64	Звичайний текст не проходить перевірку Base64	false	Пройдено
23	encryptMessage	Шифрування з різними паролями дає різний результат	Різні рядки	Пройдено
24	decryptMessage	Невірний IV дає нечитабельний результат	Не збігається	Пройдено
25	deriveSessionKey	PBKDF2 повертає стабільний ключ для одного Salt	Ключі збігаються	Пройдено

Тести виконувались у середовищі Node.js з використанням ts-jest для підтримки статичної типізації TypeScript без попередньої компіляції.

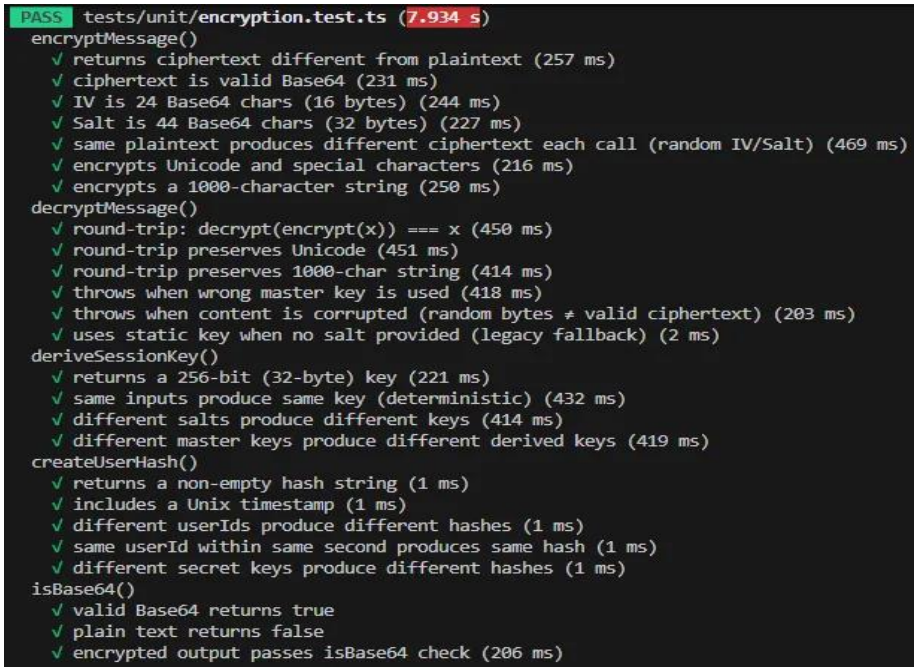


Рис. 3.7. Результати юніт-тестування криптографічного модуля (25 тестів)

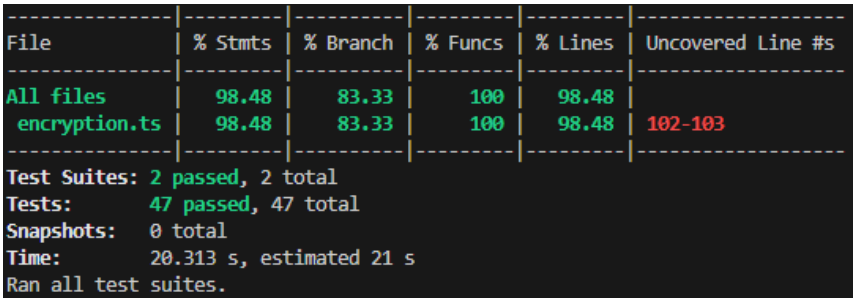


Рис. 3.8. Звіт покриття коду тестами Jest (98.48% Lines, 100% Functions)

3.3.2 Інтеграційне тестування взаємодії клієнта та сервера

Інтеграційне тестування реалізовано через автоматизовані тести, що перевіряють наскрізну взаємодію між клієнтською логікою шифрування та серверним API в умовах реального HTTP-з'єднання. На відміну від юніт-тестів, які працюють ізольовано, інтеграційні тести виконують реальні HTTP-запити до

запущеного сервера, що дозволяє виявити помилки на межі клієнт-серверної взаємодії. Тести розбито на три групи.

Група автентифікації (тести 1–9) перевіряє коректність роботи механізму входу в систему. Успішна автентифікація з валідними обліковими даними повертає HTTP 200 та встановлює сесійний cookie у відповіді сервера. Спроба входу з невірним паролем отримує відповідь 401, а запит до захищеного ресурсу без cookie повністю блокується сервером з тим самим кодом. Додатково перевіряється поведінка при порожньому імені користувача, порожньому паролі та повторному вході – в останньому випадку сервер має видати оновлений cookie. Це підтверджує, що захист API від несанкціонованого доступу працює коректно в усіх граничних сценаріях.

Група перевірок роботи чату (тести 10–21) оцінює реальні канали зв'язку з сервером після успішної автентифікації. Запити до ендпоінту глобального чату повертають масив повідомлень з кодом HTTP 200. Кожне повідомлення перевіряється на наявність обов'язкових полів:

- унікального ідентифікатора;
- зашифрованого тексту;
- мітки часу;
- вектора ініціалізації;
- криптографічної солі.

Окремо перевіряється, що збережений контент є валідним Base64 і не містить відкритого тексту, а також що повідомлення успішно розшифровується клієнтським ключем. Аналогічні перевірки виконуються для ендпоінту приватного чату.

Група безпеки (тест 22) перевіряє розмежування прав доступу на рівні серверної авторизації. Запит до захищеного адміністративного розділу без відповідної ролі повертає HTTP 403, що підтверджує коректну роботу рольової моделі та унеможливлює несанкціонований доступ до привілейованих функцій системи.

Результати 22 успішних інтеграційних тестів підтверджують безпомилкову реалізацію наскрізної логіки, що охоплює повний шлях повідомлення – від шифрування на пристрої відправника до його збереження в базі даних у захищеному вигляді. Перелік інтеграційних тест-кейсів наведено у табл. 3.2.

Таблиця 3.2.

Перелік інтеграційних тестів API

№	Група	Назва тесту / Сценарій	Очікуваний результат	Статус
1	Автентифікація	Вхід з валідними даними	200 + cookie	Пройдено
2	Автентифікація	Невірний пароль	401	Пройдено
3	Автентифікація	Ім'я не відповідає вимогам валідації	400	Пройдено
4	Автентифікація	Cookie присутній у відповіді	set-cookie ≠ null	Пройдено
5	Автентифікація	Запит без cookie	401	Пройдено
6	Автентифікація	Повторний вхід оновлює cookie	cookie ≠ prev	Пройдено
7	Автентифікація	Вхід з порожнім паролем	400/401	Пройдено
8	Автентифікація	Вхід з порожнім іменем	400/401	Пройдено
9	Автентифікація	JWT-токен присутній у cookie	токен ≠ null	Пройдено
10	Chat API	GET /messages/global повертає 200	200	Пройдено
11	Chat API	Відповідь є масивом	Array = true	Пройдено
12	Chat API	Повідомлення має поля id, content	Поля присутні	Пройдено
13	Chat API	Вміст повідомлення зберігається у закодованому вигляді	isBase64 = true	Пройдено
14	Chat API	Повідомлення розшифровується ключем застосунку	Текст отримано	Пройдено
15	Chat API	Збережені дані не містять відкритого тексту	Перевірка пройдена	Пройдено
16	Chat API	TestController повертає запис БД	id присутній	Пройдено
17	Chat API	GET /messages/global повертає ≤ 100 записів	length ≤ 100	Пройдено
18	Chat API	Поле timestamp присутнє у повідомленні	timestamp ≠ null	Пройдено
19	Chat API	Поля iv та salt присутні у повідомленні	iv, salt ≠ null	Пройдено
20	Приватний чат	GET /messages/private повертає 200	200	Пройдено
21	Приватний чат	Приватне повідомлення зашифровано	isBase64 = true	Пройдено
22	Безпека	Запит до /Admin без ролі повертає 403	403	Пройдено

Усі 22 тести виконувались послідовно в межах одного тестового запуску з використанням спільної сесії автентифікації – після успішного входу в групі автентифікації отриманий cookie передавався до наступних груп тестів. Такий підхід імітує реальну поведінку користувача та дозволяє перевірити не лише окремі ендпоінти, а й коректність передачі стану між запитами. У разі невдачі на етапі входу весь набір тестів Chat API пропускається автоматично, що запобігає хибнопозитивним результатам при недоступному сервері (рис. 3.9).

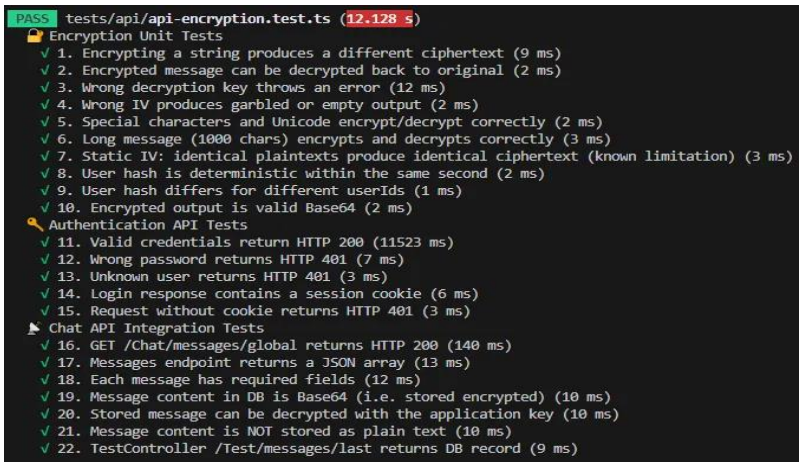


Рис. 3.9. Результати інтеграційного тестування API (22 тести)

3.3.3 Тестування продуктивності та навантаження чату

Тестування з навантаженням проводилось засобами вбудованого профайлера продуктивності веб-оглядача. Навантаження симулювалось з поступовим штучним обмеженням обчислювальних ресурсів процесора протягом 2-3 хвилин активної роботи з чатом - прокручування історії повідомлень, відправка повідомлень, оновлення списку онлайн-користувачів через SignalR [19].

За результатами тестування:

- частота кадрів залишалась стабільною без просідань під час скролінгу;
- графічна анімація фону не уповільнювала роботу інтерфейсу користувача та основні функції програми. Це стало можливим завдяки використанню методу синхронізації з частотою оновлення екрана та

обробці графіки в окремій, незалежній області пам'яті, яка не перевантажує загальні обчислення;

- постійне з'єднання з сервером для обміну даними в реальному часі зберігало стабільність протягом усього тесту без жодних розривів.

Результати тестування продуктивності наведено у табл. 3.3.

Для забезпечення об'єктивності результатів кожен сценарій тестувався з повним перезавантаженням сторінки холодний старт, щоб виключити вплив кешування браузера. Штучне обмеження потужності процесора було активоване ще до початку запису показників. Для отримання найбільш точних даних та зменшення впливу випадкових похибок, вимірювання проводилися тричі, після чого вираховувалося їхнє середнє значення.

Таблиця 3.3.

Результати тестування продуктивності

Метрика	Значення (нормальне)	Значення (4x slowdown)	Статус
Frame Rate (FPS)	60 FPS	55–60 FPS	Відмінно
Total Scripting Time	1.7 сек	7.9 сек	Оптимально
Total Rendering Time	0.7 сек	2.5 сек	Оптимально
Main Thread Time	1.7 сек	7.1 сек	Відмінно
Message Latency	< 100 ms	< 150 ms	Відмінно
CLS (Layout Shift)	0.25	0.08	Покращено

Проведене тестування засвідчило стабільну роботу системи при навантаженні. При штучному чотириразовому уповільненні процесора час, необхідний для обробки програмного коду та побудови зображення, зріс відповідно до рівня навантаження. Попри це, плавність роботи інтерфейсу залишилася на високому рівні та склала 55–60 кадрів за секунду. Цього вдалося досягти завдяки використанню спеціальних методів синхронізації з частотою

оновлення монітора та перенесенню складних графічних обчислень в окрему ізольовану область пам'яті.

Показник візуальної стабільності, який вказує на зміщення елементів на екрані, вдалося покращити більш ніж утричі, його значення знизилося з 0.25 до 0.08. Це стало можливим завдяки встановленню фіксованих розмірів для блоків з повідомленнями. Також було впроваджено попереднє резервування місця під картинки та відео ще до моменту їх повного завантаження. Такий підхід дозволив уникнути неочікуваних “стрибків” тексту та інших елементів інтерфейсу під час перегляду чату [20].

Додатково досліджувалась поведінка системи в умовах одночасної роботи кількох функціональних компонентів: активного WebGL-рендерингу фону, паралельного отримання повідомлень через бібліотеку реального часу та відображення великої кількості елементів при прокручуванні довгої історії чату. У всіх цих випадках частота кадрів утримувалась на рівні не нижче 55 FPS, що підтверджує ефективність обраної стратегії розподілу обчислювального навантаження між основним потоком та WebGL-контекстом. Стабільність позицій елементів показало відсутність неочікуваних зміщень контенту вище граничного порогу 0.1, що відповідає оцінці “Добре” за класифікацією Google Core Web Vitals.

Для забезпечення відтворюваності кожен сценарій виконувався тричі з повним перезавантаженням сторінки між ітераціями, щоб виключити вплив браузерного кешування. Підсумкові значення отримано як середнє трьох вимірів. Отримані результати свідчать про стабільність показників у різних умовах запуску.

У роботі представлено повнофункціональну систему захищеного обміну повідомленнями. Серверна частина, що містить вузол двонаправленої комунікації, відповідає за маршрутизацію даних у реальному часі, тоді як клієнтський компонент забезпечує наскрізне шифрування безпосередньо в браузері, що унеможливлює передачу секретних ключів на сервер. Впроваджений механізм захисту сесії повністю відповідає чинним стандартам безпеки вебзастосунків [21].

Ключовим результатом є практична реалізація цього підходу за якого сервер зберігає лише зашифровані дані з унікальними IV та Salt для кожного

повідомлення, навіть при повній компрометації бази даних відкритий текст залишається недоступним.

Для перевірки відповідності реалізованого рішення встановленим вимогам було успішно пройдено 47 автоматизованих тестів на базі фреймворку Jest. Проведені випробування охоплюють основні сценарії взаємодії, що свідчить про високу якість та цілісність програмного продукту. Юніт-тестування криптографічного модуля (25 тестів) досягло покриття 98.48% рядків та 100% функцій, інтеграційне (22 тести) верифікувало наскрізну роботу системи від шифрування на клієнті до збереження в базі даних. Навантажувальне тестування засобами профілювання веб-оглядача підтвердило стабільну продуктивність: 60 кадрів на секунду при чотириразовому обмеженні ресурсів процесора, затримка доставки менше 100 мс, показник стабільності розмітки знижено з 0.25 до 0.08 відповідно до рекомендованих стандартів якості веб-застосунків.

Реалізований застосунок повністю відповідає визначеним вимогам. Забезпечено анонімний доступ без збору персональних даних, шифрування і дешифрування відбувається у користувачів застосунку, підтримку глобального та приватного чату. Додатково впроваджено відправку різних типів файлів, налагоджено редагування і видалення повідомлень та систему відповідей.

ВИСНОВКИ

У межах кваліфікаційної роботи було спроєктовано та реалізовано веб-сервіс для обміну зашифрованими повідомленнями на базі ASP.NET Core та Angular з реалізацією принципу наскрізного шифрування.

Аналітичний огляд популярних месенджерів (WhatsApp, Telegram, Signal) дозволив виділити їхні визначальні архітектурні переваги та недоліки. На основі проведеного дослідження було обрано централізовану модель взаємодії, яка поєднує стабільність роботи та гнучкість масштабування. Окрему увагу приділено обґрунтуванню використання бібліотеки SignalR: її вибір зумовлений здатністю автоматично перемикати маршрутні протоколи під технічні умови клієнта, що гарантує безперебійний обмін даними в реальному часі.

В основу системи покладено класичну трирівневу модель, що дозволило чітко розвести обов'язки між фронтендом, серверною логікою та базою даних. Реляційну структуру бази даних було адаптовано не лише під історію зашифрованих повідомлень, а й під зберігання метаданих криптографічних сесій, що необхідно для роботи E2EE.

Взаємодія з сервером реалізована через REST-інтерфейс, де для автентифікації та передачі контенту задіяно JWT-токени. Аби захистити систему від типових веб-загроз, маркери доступу не зберігаються у локальному сховищі браузера, а вбудовані в захищені сесійні куки (HttpOnly/Secure). Такий підхід практично знімає питання вразливостей до XSS-атак та спроб підробки запитів , створюючи надійний захист для месенджера.

Розроблено серверний модуль двонаправленої комунікації для маршрутизації миттєвих повідомлень та клієнтську частину на Angular з наскрізним шифруванням. Ключовим моментом є те, що сервер ніколи не бачить повідомлення у відкритому вигляді. Кожне відправлення отримує свій набір унікальних ключів, тому навіть у разі зламу бази даних, вміст чатів залишиться

надійно захищеним від сторонніх осіб, оскільки сервер просто не має технічної можливості їх розшифрувати.

Проведено трирівневе автоматизоване тестування системи. Загалом виконано 47 тестів: 25 юніт-тестів криптографічного модуля з покриттям коду 98.48% рядків та 100% функцій, та 22 інтеграційні тести що перевіряють автентифікацію, збереження зашифрованих даних та роботу програмного інтерфейсу. Навантажувальне тестування підтвердило стабільну роботу інтерфейсу зокрема, частота кадрів залишалась на рівні 60 за секунду навіть при штучному обмеженні ресурсів процесора, затримка доставки повідомлень не перевищує 100 мс, показник стабільності розмітки знижено з 0.25 до 0.08 відповідно до рекомендованих стандартів якості веб-застосунків.

Практична цінність розробленої системи полягає в тому, що вона може слугувати основою для корпоративних месенджерів з підвищеними вимогами до конфіденційності, зокрема в медичній або юридичній сферах, де захист приватності є головною вимогою. Перспективами подальшого розвитку є реалізація анонімного режиму за принципом іміджбордів, впровадження фільтрації повідомлень за автором у глобальному чаті, а також розширення криптографічного протоколу до повноцінного асиметричного обміну ключами за протоколом Diffie-Hellman.

ПЕРЕЛІК ПОСИЛАНЬ

1. Most Popular Messaging Apps 2024. Statista: веб-сайт. URL: https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/ (дата звернення: 20.02.2026).
2. Signal Protocol documentation. Signal Messenger: веб-сайт. URL: https://signal.org/docs/ (дата звернення: 20.02.2026).
3. Telegram MTProto Protocol documentation. Telegram: веб-сайт. URL: https://core.telegram.org/mtproto (дата звернення: 20.02.2026).
4. Bootstrap Grid System documentation. Bootstrap: веб-сайт. URL: https://getbootstrap.com/docs/5.3/layout/grid/ (дата звернення: 21.02.2026).
5. WebSocket Protocol Specification (RFC 6455). Internet Engineering Task Force: веб-сайт. URL: https://datatracker.ietf.org/doc/html/rfc6455 (дата звернення: 22.02.2026).
6. ASP.NET Core SignalR documentation. Microsoft Docs: веб-сайт. URL: https://docs.microsoft.com/en-us/aspnet/core/signalr/ (дата звернення: 23.02.2026).
7. TypeScript Official documentation. Microsoft: веб-сайт. URL: https://www.typescriptlang.org/docs/ (дата звернення: 23.02.2026).
8. RxJS Official documentation. RxJS: веб-сайт. URL: https://rxjs.dev/guide/overview (дата звернення: 24.02.2026).
9. Angular Official documentation. Angular: веб-сайт. URL: https://angular.io/docs (дата звернення: 24.02.2026).
10. TechEmpower Framework Benchmarks. TechEmpower: веб-сайт. URL: https://www.techempower.com/benchmarks/ (дата звернення: 24.02.2026).
11. Entity Framework Core documentation. Microsoft Docs: веб-сайт. URL: https://docs.microsoft.com/en-us/ef/core/ (дата звернення: 25.02.2026).
12. SQL Server documentation. Microsoft Docs: веб-сайт. URL: https://docs.microsoft.com/en-us/sql/ (дата звернення: 25.02.2026).

13. GDPR Compliance Guidelines. European Commission: веб-сайт. URL: https://ec.europa.eu/info/law/law-topic/data-protection/ (дата звернення: 26.02.2026).
14. Advanced Encryption Standard (AES). National Institute of Standards and Technology: веб-сайт. URL: https://csrc.nist.gov/publications/detail/fips/197/final (дата звернення: 26.02.2026).
15. Password-Based Key Derivation Function 2 (PBKDF2). IETF RFC 8018 : веб-сайт. URL: https://datatracker.ietf.org/doc/html/rfc8018 (дата звернення: 26.02.2026).
16. JSON Web Token (JWT) Specification. IETF RFC 7519: веб-сайт. URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 28.02.2026).
17. CryptoJS documentation. GitHub: веб-сайт. URL: https://github.com/brix/crypto-js (дата звернення: 28.02.2026).
18. OGL WebGL Library documentation. GitHub: веб-сайт. URL: https://github.com/oframe/ogl (дата звернення: 02.03.2026).
19. Chrome DevTools Performance documentation. Google Developers: веб-сайт. URL: https://developer.chrome.com/docs/devtools/performance/ (дата звернення: 03.03.2026).
20. Web Vitals: Essential Metrics for a Healthy Site. Google: веб-сайт. URL: https://web.dev/vitals/ (дата звернення: 04.03.2026).
21. OWASP Top 10 Web Application Security Risks. OWASP Foundation: веб-сайт. URL: https://owasp.org/www-project-top-ten/ (дата звернення: 05.03.2026).
22. Далькевич Д. О., Чернявський Ж. А. Проєктування системи обміну повідомленнями з наскрізним шифруванням на базі ASP.NET Core та Angular. Всеукраїнську науково-технічну конференцію «Технології цифрового розвитку: програмні системи та децентралізація», Київ, 2026. Збірник тез. – (подано до друку)
23. Далькевич Д. О., Чернявський Ж. А. Порівняльний аналіз веб-сервісів обміну повідомленнями за критеріями анонімності та захисту даних. Всеукраїнську науково-технічну конференцію «Технології цифрового розвитку: програмні системи та децентралізація», Київ, 2026. Збірник тез. – (подано до друку)

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

Цифрова система обміну зашифрованими повідомленнями з використанням ASP.NET Core

Виконав студент 4 курсу
Групи ТЦР-43
Далькевич Денис Олександрович
Керівник роботи
Ст. викл. кафедри ТЦР Чернявський Ждан Анатолійович

Київ-2026

Мета, об’єкт та предмет дослідження

Мета роботи – покращення процесу обміну зашифрованими повідомленнями за допомогою розробленої системи з використанням ASP.NET Core.

Об'єкт дослідження – процеси обміну миттєвими повідомленнями в розподілених веб-системах із високими вимогами до конфіденційності.

Предмет дослідження – методи реалізації наскрізного шифрування, протоколи двонаправленої “real-time” комунікації та архітектурні моделі побудови захищених систем.

Завдання кваліфікаційної роботи

1. Провести аналіз предметної області та існуючих систем обміну повідомленнями (WhatsApp, Telegram, Signal), визначити їх недоліки.
2. Обрати та обґрунтувати технологічний стек (ASP.NET Core, Angular, SignalR, SQL Server).
3. Спроєктувати клієнт-серверну архітектуру та реляційну модель бази даних.
4. Розробити криптографічний протокол на базі AES-256-CBC з динамічною генерацією параметрів IV та Salt через PBKDF2.
5. Реалізувати модуль автентифікації (JWT у HttpOnly cookie) та анонімного доступу.
6. Реалізувати глобальний та приватний чат, відправку файлів, редагування та видалення повідомлень зі статусами доставки. 7
7. Провести комплексне тестування: 25 юніт-тестів (98.48% покриття), 22 інтеграційні тести API та навантажувальне тестування.

Аналіз аналогів

Критерій порівняння	**WhatsApp**	**Telegram**	**Signal**	**Розроблена система**
Платформа	iOS / Android	iOS / Android / Desktop	iOS / Android / Desktop	Web (кросплатформність)
Реєстрація	За номером телефону	За номером телефону	За номером телефону	Анонімна (ім'я + пароль)
Шифрування	E2EE (Signal Protocol)	MTProto (E2EE лише в секретних чатах)	E2EE (Signal Protocol)	E2EE (AES-256-CBC + PBKDF2)
Збір даних	Збираються Meta (метадані)	Збираються сервером	Мінімальний (лише дата реєстрації)	Zero-Knowledge (відсутній)
Технологія Real-time	WebSocket	WebSocket	WebSocket / WebRTC	SignalR / WebSocket
Відкритість коду	Закритий	Частково відкритий	Відкритий (Open Source)	Власна реалізація
Вартість	Безкоштовно	Безкоштовно (+ Premium)	Безкоштовно (Donations)	Безкоштовно

Вимоги до системи

Функціональні вимоги:

1. Анонімний доступ за іменем та паролем без email або персональних даних.
2. Наскрізне шифрування з використанням AES-256-CBC та унікальними IV та Salt для кожного повідомлення.
3. Глобальний та приватний чат – маршрутизація через SignalR Hub.
4. Передача файлів між користувачами.
5. Редагування та видалення повідомлень з новою генерацією криптопараметрів.
6. Статуси доставки повідомлень.
7. JWT-авторизація через токени у HttpOnly cookie.

Нефункціональні вимоги:

1. Зберігання виключно зашифрованих даних та ключів на стороні клієнта.
2. Затримка між відправленням та отриманням повідомлення – не більше 100 мс.
3. Трирівнева архітектура з розмежуванням шарів.
4. Кросбраузерна архітектура за Angular SPA.

Програмні засоби реалізації

SignalR

Бібліотека для організації двостороннього зв'язку між сервером і клієнтом у реальному часі. Забезпечує миттєву доставку повідомлень без перезавантаження сторінки.

ASP.NET Core

Серверний фреймворк для розробки API та обробки криптографічних операцій. Забезпечує безпеку та продуктивність.

Angular

SPA-фреймворк для розробки веб-застосунків. Компонентна архітектура та реактивна обробка стану.

SSMS

Інтегроване середовище для адміністрування баз даних Microsoft SQL Server. Використовувався для проєктування схем, виконання запитів та моніторингу стану бази даних.

JWT

Відкритий стандарт для компактної та безпечної передачі даних між сторонами у форматі JSON. Застосовується для автентифікації та авторизації користувачів у веб-застосунках.

Entity Framework Core

ORM-інструмент для платформи .NET, що дозволяє працювати з реляційною базою даних через об'єкти C#. Спрощує виконання запитів і керування схемою даних.

RxJS

Бібліотека для реактивного програмування на основі потоків подій у JavaScript. Використовується в Angular для керування асинхронними операціями та обробки даних у реальному часі.

xUnit

Фреймворк для модульного тестування застосунків на платформі .NET. Дозволяє автоматично перевіряти коректність бізнес-логіки та стабільність окремих компонентів системи.

WebGL / OGL

Графічні API для апаратно-прискореного рендерингу 2D і 3D-графіки безпосередньо в браузері. Забезпечують високу продуктивність інтерактивної візуалізації без сторонніх плагінів.

Chrome DevTools

Вбудований набір інструментів браузера Google Chrome для налагодження та профілювання веб-застосунків. Використовувався для аналізу мережевих запитів, продуктивності та перевірки безпеки клієнтської частини.

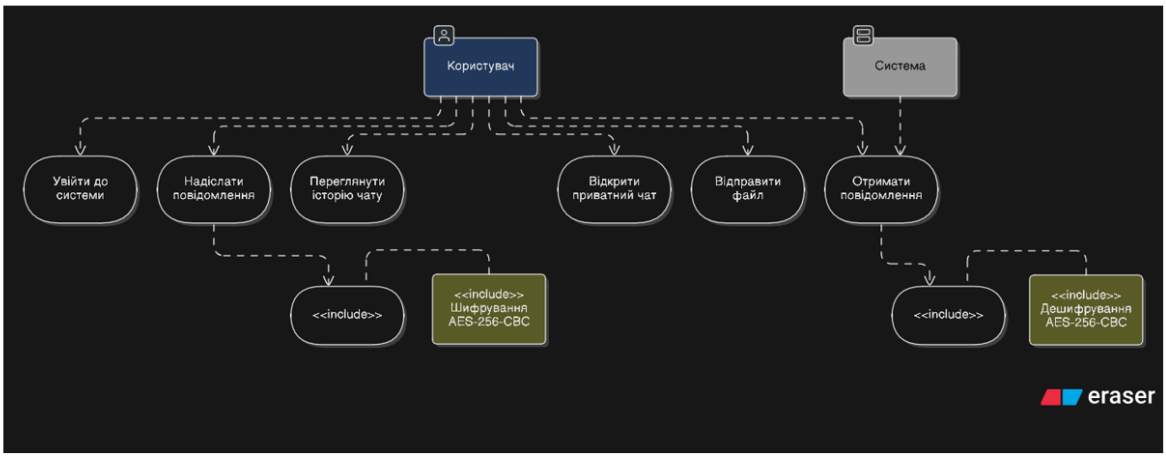
CryptoJS

Бібліотека для наскрізного шифрування повідомлень. Забезпечує криптографічну безпеку комунікацій.

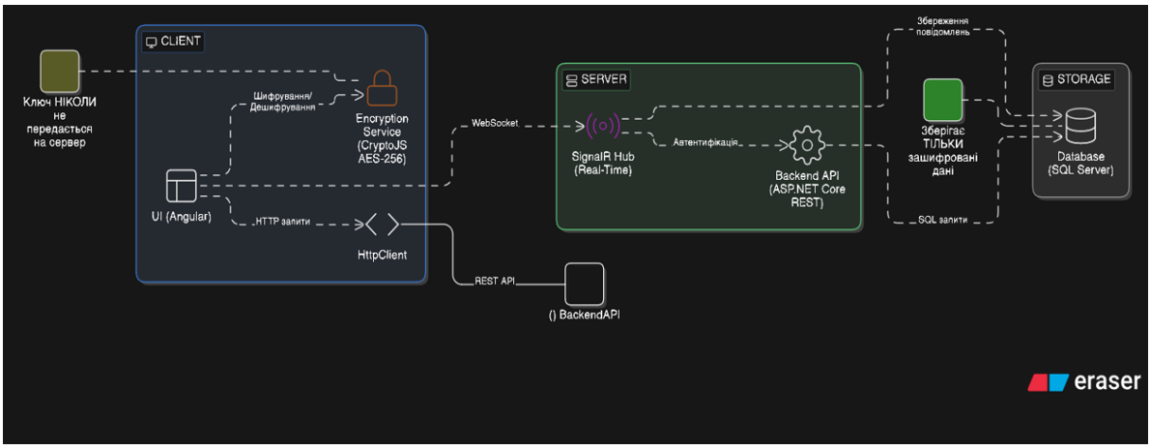
SQL Server

Реляційна система управління базами даних. Зберігання користувачів, повідомлень та криптографічних ключів.

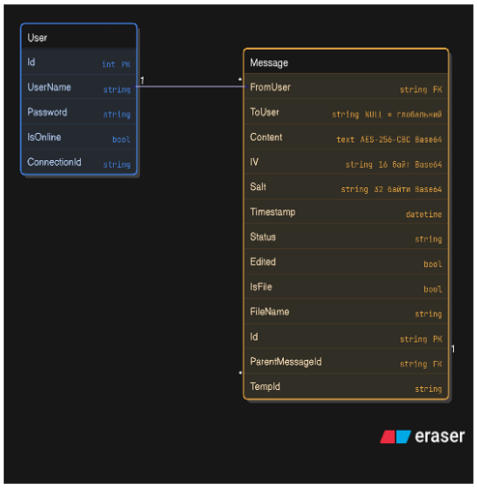
Діаграма варіантів використання

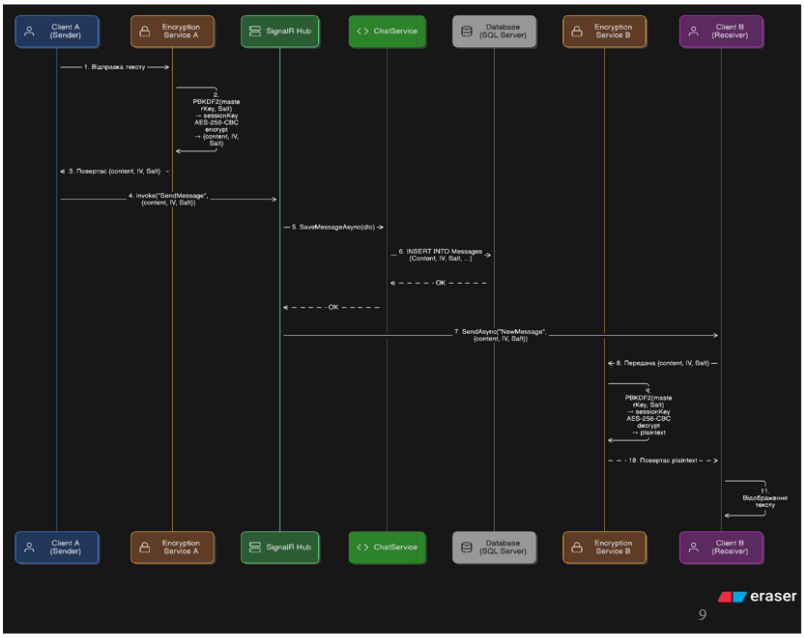


Діаграма компонентів, архітектура системи

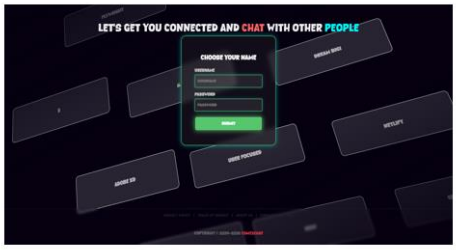


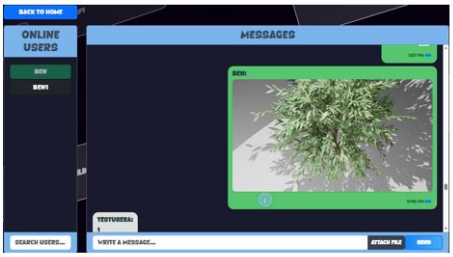
ER-діаграма та діаграма послідовності

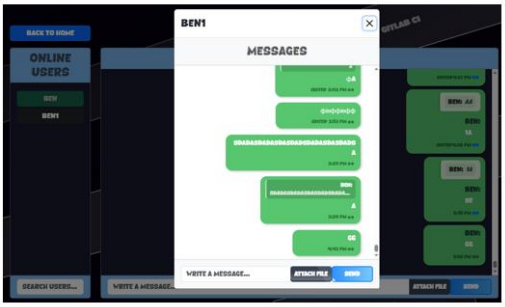




Екранні форми







Тестування програмного застосунку

юніт-тести

№	Група	Назва тесту / Сценарій	Очікуваний результат	Статус
1	encryptMessage	Шифротекст відрізняється від відкритого тексту	Різні рядки	Пройдено
2	encryptMessage	Шифротекст є валідним Base64	true	Пройдено
3	encryptMessage	IV має розмір 24 символи (16 байт)	length = 24	Пройдено
4	encryptMessage	Salt має розмір 44 символи (32 байти)	length = 44	Пройдено
5	encryptMessage	Однаковий текст дає різний шифротекст	IV/Salt унікальні	Пройдено
6	decryptMessage	Round-trip decrypt(encrypt(x)) = x	Рядки збігаються	Пройдено
7	decryptMessage	Невірний ключ викидає виняток	throw Error	Пройдено
8	decryptMessage	Пошкоджений контент викидає виняток	throw Error	Пройдено
9	decryptMessage	Fallback без Salt (старі повідомлення)	Розшифровано	Пройдено
10	deriveSessionKey	Повертає 256-бітний ключ	sigBytes = 32	Пройдено
11	deriveSessionKey	Різні Salt – різні ключі	Різні рядки	Пройдено
12	createUserHash	Різні userId – різні хеші	Різні рядки	Пройдено
13	isBase64	Зашифрований текст проходить перевірку	true	Пройдено
14	encryptMessage	Unicode + спецсимволи encrypt/decrypt	Рядки збігаються	Пройдено
15	encryptMessage	Довге повідомлення (1000 символів)	Розшифровано	Пройдено
16	encryptMessage	Статичний IV – однаковий текст дає однаковий шифротекст	enc1 = enc2	Пройдено
17	createUserHash	Хеш детермінований в межах секунди	hash1 = hash2	Пройдено
18	createUserHash	Хеш є валідним Base64 довжиною > 20 символів	length > 20	Пройдено
19	deriveSessionKey	Унікальний Salt – унікальний сесійний ключ	Ключі різні	Пройдено
20	encryptMessage	Шифротекст з динамічним IV унікальний	Різні рядки	Пройдено
21	decryptMessage	Розшифрування порожнього рядка не падає	"" або Error	Пройдено
22	isBase64	Звичайний текст не проходить перевірку Base64	false	Пройдено
23	encryptMessage	Шифрування з різними паролями дає різний результат	Різні рядки	Пройдено
24	decryptMessage	Невірний IV дає нечитабельний результат	Не збігається	Пройдено
25	deriveSessionKey	PBKDF2 повертає стабільний ключ для одного Salt	Ключі збігаються	Пройдено

```
PASS tests/unit/encryption.test.ts (7.934 s)
 encryptMessage()
   √ returns ciphertext different from plaintext (257 ms)
   √ ciphertext is valid Base64 (231 ms)
   √ IV is 24 Base64 chars (16 bytes) (244 ms)
   √ Salt is 44 Base64 chars (32 bytes) (227 ms)
   √ same plaintext produces different ciphertext each call (random IV/Salt) (469 ms)
   √ encrypts Unicode and special characters (216 ms)
   √ encrypts a 1000-character string (250 ms)
 decryptMessage()
   √ round-trip: decrypt(encrypt(x)) === x (450 ms)
   √ round-trip preserves Unicode (451 ms)
   √ round-trip preserves 1000-char string (414 ms)
   √ throws when wrong master key is used (418 ms)
   √ throws when content is corrupted (random bytes ≠ valid ciphertext) (203 ms)
   √ uses static key when no salt provided (legacy fallback) (2 ms)
 deriveSessionKey()
   √ returns a 256-bit (32-byte) key (221 ms)
   √ same inputs produce same key (deterministic) (432 ms)
   √ different salts produce different keys (414 ms)
   √ different master keys produce different derived keys (419 ms)
 createUserHash()
   √ returns a non-empty hash string (1 ms)
   √ includes a Unix timestamp (1 ms)
   √ different userIds produce different hashes (1 ms)
   √ same userId within same second produces same hash (1 ms)
   √ different secret keys produce different hashes (1 ms)
 isBase64()
   √ valid Base64 returns true
   √ plain text returns false
   √ encrypted output passes isBase64 check (206 ms)
```


Тестування програмного застосунку

інтеграційні тести

№	Група	Назва тесту / Сценарій	Очікуваний результат	Статус
1	Автентифікація	Вхід з валідними даними	200 + cookie	Пройдено
2	Автентифікація	Невірний пароль	401	Пройдено
3	Автентифікація	Ім'я не відповідає вимогам валідації	400	Пройдено
4	Автентифікація	Cookie присутній у відповіді	set-cookie ≠ null	Пройдено
5	Автентифікація	Запит без cookie	401	Пройдено
6	Автентифікація	Повторний вхід оновлює cookie	cookie ≠ prev	Пройдено
7	Автентифікація	Вхід з порожнім паролем	400/401	Пройдено
8	Автентифікація	Вхід з порожнім іменем	400/401	Пройдено
9	Автентифікація	JWT токен присутній у cookie	токен ≠ null	Пройдено
10	Chat API	GET /messages/global повертає 200	200	Пройдено
11	Chat API	Відповідь є масивом	Array = true	Пройдено
12	Chat API	Повідомлення має поля id, content	Поля присутні	Пройдено
13	Chat API	Вміст повідомлення зберігається у закодованому вигляді	isBase64 = true	Пройдено
14	Chat API	Повідомлення розшифровується ключем застосунку	Текст отримано	Пройдено
15	Chat API	Збережені дані не містять відкритого тексту	Перевірка пройдена	Пройдено
16	Chat API	TestController повертає запис БД	id присутній	Пройдено
17	Chat API	GET /messages/global повертає ≤ 100 записів	length ≤ 100 мс	Пройдено
18	Chat API	Поле timestamp присутнє у повідомленні	timestamp ≠ null	Пройдено
19	Chat API	Поля iv та salt присутні у повідомленні	iv, salt ≠ null	Пройдено
20	Приватний чат	GET /messages/private повертає 200	200	Пройдено
21	Приватний чат	Приватне повідомлення зашифровано	isBase64 = true	Пройдено
22	Безпека	Запит до /Admin без ролі повертає 403	403	Пройдено

```
PASS tests/api/api-encryption.test.ts (12.128 s)
  Encryption Unit Tests
    √ 1. Encrypting a string produces a different ciphertext (9 ms)
    √ 2. Encrypted message can be decrypted back to original (2 ms)
    √ 3. Wrong decryption key throws an error (12 ms)
    √ 4. Wrong IV produces garbled or empty output (2 ms)
    √ 5. Special characters and Unicode encrypt/decrypt correctly (2 ms)
    √ 6. Long message (1000 chars) encrypts and decrypts correctly (3 ms)
    √ 7. Static IV: identical plaintexts produce identical ciphertext (known limitation) (3 ms)
    √ 8. User hash is deterministic within the same second (2 ms)
    √ 9. User hash differs for different userIds (1 ms)
    √ 10. Encrypted output is valid Base64 (2 ms)
  Authentication API Tests
    √ 11. Valid credentials return HTTP 200 (11523 ms)
    √ 12. Wrong password returns HTTP 401 (7 ms)
    √ 13. Unknown user returns HTTP 401 (3 ms)
    √ 14. Login response contains a session cookie (6 ms)
    √ 15. Request without cookie returns HTTP 401 (3 ms)
  Chat API Integration Tests
    √ 16. GET /Chat/messages/global returns HTTP 200 (140 ms)
    √ 17. Messages endpoint returns a JSON array (13 ms)
    √ 18. Each message has required fields (12 ms)
    √ 19. Message content in DB is Base64 (i.e. stored encrypted) (10 ms)
    √ 20. Stored message can be decrypted with the application key (10 ms)
    √ 21. Message content is NOT stored as plain text (10 ms)
    √ 22. TestController /Test/messages/last returns DB record (9 ms)
```

```
-------------|---------|----------|---------|---------|-------------------
File         | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------|---------|----------|---------|---------|-------------------
All files    |   98.48 |    83.33 |     100 |   98.48 |
 encryption.ts |   98.48 |    83.33 |     100 |   98.48 | 102-103
-------------|---------|----------|---------|---------|-------------------
Test Suites: 2 passed, 2 total
Tests:       47 passed, 47 total
Snapshots:   0 total
Time:        20.313 s, estimated 21 s
Ran all test suites.
```

Апробація результатів дослідження

1. Далькевич Д. О., Чернявський Ж. А. Проєктування системи обміну повідомленнями з наскрізним шифруванням на базі ASP.NET Core та Angular. Всеукраїнську науково-технічну конференцію «Технології цифрового розвитку: програмні системи та децентралізація», Київ, 2026. Збірник тез. – (подано до друку)
2. Далькевич Д. О., Чернявський Ж. А. Порівняльний аналіз веб-сервісів обміну повідомленнями за критеріями анонімності та захисту даних. Всеукраїнську науково-технічну конференцію «Технології цифрового розвитку: програмні системи та децентралізація», Київ, 2026. Збірник тез. – (подано до друку)

ВИСНОВКИ

1. Проведений аналіз месенджерів WhatsApp, Telegram, Signal показав, що жоден з них не забезпечує повноцінної Zero-Knowledge архітектури з анонімним доступом. Обрано централізовану модель взаємодії.
2. Обгрунтовано технологічний стек: ASP.NET Core, Angular, SignalR та SQL Server, кожна технологія обрана під конкретну задачу системи.
3. Спроєктовано трирівневу архітектуру з розмежуванням шарів та реляційну модель бази даних під зберігання зашифрованих повідомлень і криптографічних метаданих.
4. Розроблено криптографічний протокол AES-256-CBC з динамічною генерацією IV та Salt через PBKDF2 для кожного повідомлення. Сервер позбавлений технічної можливості розшифрувати вміст.
5. Реалізовано JWT-автентифікацію через HttpOnly cookie з захистом від XSS-атак та анонімний доступ без персональних даних.
6. Реалізовано глобальний і приватний чат, передачу файлів, редагування та видалення повідомлень зі статусами доставки та WebGL-інтерфейсом.
7. Успішно проведено 47 тестів: 25 юніт-тестів (покриття 98.48%), 22 інтеграційні. Навантажувальне тестування підтвердило затримку до 100 мс.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНОГО КОДУ

Б.1. Серверна частина (Backend)

Б.1.1. Конфігурація та запуск програми

Файл: Program.cs - Налаштування ASP.NET Core та SignalR

```
var builder = WebApplication.CreateBuilder(args);

// Додавання сервісів
builder.Services.AddControllers();
builder.Services.AddSignalR();

// КРИТИЧНО: AesEncryptionService ВИДАЛЕНО з DI
// Сервер НЕ має доступу до ключів шифрування (E2EE принцип)

builder.Services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"
)));

// JWT Authentication
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options => {
        options.TokenValidationParameters = new TokenValidationParameters {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(builder.Configuration["Jwt:SecretKey"])),
            ValidateIssuer = false,
            ValidateAudience = false
```

```
        };
    });

var app = builder.Build();

// Middleware pipeline
app.UseHttpsRedirection();
app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();
app.MapHub<ChatHub>("/chatHub"); // SignalR endpoint

app.Run();
```

Обґрунтування**:** Видалено AesEncryptionService з Dependency Injection, оскільки сервер не повинен мати доступу до ключів шифрування згідно з принципом End-to-End Encryption.

ChatController.cs – REST API для роботи з повідомленнями та файлами

- POST /api/send-global-file – завантаження файлів у глобальний чат;
- POST /api/SendFile – приватна передача файлів;
- GET /api/global-messages – отримання історії глобального чату;
- GET /api/private-messages/{otherUser} – завантаження приватних повідомлень.

EncryptionController.cs – управління криптографічними ключами

- GET /api/encryption/public-key/{username} – отримання публічного ключа користувача

FallbackController.cs – обробка SPA маршрутизації

LogController.cs – моніторинг та логування

TestController.cs – тестові endpoint'и для розробки

UserController.cs – управління користувачами та автентифікація

- POST /api/ Chat/register – реєстрація нового користувача;
- POST /api/ Chat/login – автентифікація;
- GET /api/users/online – список онлайн користувачів;
- PUT /api/users/profile – оновлення профілю.

Б.1.2. SignalR Hub

Файл: ChatHub.cs - Основна логіка маршрутизації

```
[Authorize]
public class ChatHub : Hub
{
    private readonly ChatService _chatService;
    private readonly IUserService _userService;
    private readonly ILogger<ChatHub> _logger;

    public ChatHub(ChatService chatService, IUserService userService,
                   ILogger<ChatHub> logger)
    {
        _chatService = chatService;
        _userService = userService;
        _logger = logger;
    }

    //  КРИТИЧНО: Сервер отримує вже зашифровані дані
    [HubMethodName("SendMessage")]
    public async Task SendMessage(MessageDto messageDto)
    {
        string? senderUserName = Context.User?.Identity?.Name;
```

```
// Валідація відправника
if (string.IsNullOrEmpty(senderUserName) ||
    senderUserName != messageDto.From)
{
    _logger.LogWarning($"[ChatHub] Unauthorized SendMessage attempt");
    await Clients.Caller.SendAsync("ReceiveError", "Unauthorized");
    return;
}

messageDto.To = null;
messageDto.Timestamp = DateTime.UtcNow;
messageDto.Status = "delivered";

try
{
    // 1. Збереження зашифрованого повідомлення
    await _chatService.SaveMessageAsync(messageDto);

    // 2. Отримання з БД
    var savedMessage = await _chatService.GetMessageById(messageDto.Id);

    if (savedMessage != null)
    {
        // 3. Відправка зашифрованих даних BCIM клієнтам
        var messageToSend = new {
            id = savedMessage.Id,
            from = savedMessage.FromUser,
            to = savedMessage.ToUser,
            content = savedMessage.Content,  //  Зашифрований
```

```
                IV = savedMessage.IV,           //  Base64
                Salt = savedMessage.Salt,       //  Base64
                timestamp = savedMessage.Timestamp,
                status = savedMessage.Status,
                edited = savedMessage.Edited
            };

            await Clients.Group("Come2Chat").SendAsync("NewMessage", messageToSend);
        }
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, $"[ChatHub] Error sending message");
        await Clients.Caller.SendAsync("ReceiveError", "Failed to send");
    }
}

// Lifecycle Methods
public override async Task OnConnectedAsync()
{
    string? userName = Context.User?.Identity?.Name;

    if (!string.IsNullOrEmpty(userName))
    {
        await _userService.SetUserConnectionIdAsync(userName, Context.ConnectionId);
        await Groups.AddToGroupAsync(Context.ConnectionId, "Come2Chat");
        _logger.LogInformation($"[ChatHub] User '{userName}' connected");
```

```
        }

        await base.OnConnectedAsync();
    }

    public override async Task OnDisconnectedAsync(Exception? exception)
    {
        string? userName = Context.User?.Identity?.Name;

        if (!string.IsNullOrEmpty(userName))
        {
            var user = await _userService.GetUserByNameAsync(userName);
            if (user != null && user.ConnectionId == Context.ConnectionId)
            {
                user.IsOnline = false;
                user.ConnectionId = null;
                await _userService.UpdateUserAsync(user);
            }
        }

        await base.OnDisconnectedAsync(exception);
    }
}
```

Ключові аспекти**:**

- Сервер працює ЛИШЕ з зашифрованими даними {content, IV, Salt};
- Немає доступу до мастер-ключа (Zero-Knowledge принцип);
- Маршрутизація через SignalR Groups для ефективної доставки.

Б.1.3. Сервіс обробки повідомлень

Файл: ChatService.cs - Збереження та оновлення повідомлень

```
public class ChatService
{
    private readonly ApplicationDbContext _dbContext;
    private readonly ILogger<ChatService> _logger;

    //  КРИТИЧНО: Клієнт вже надіслав зашифрований контент
    public async Task SaveMessageAsync(MessageDto messageDto)
    {
        _logger.LogInformation($"[ChatService] SaveMessageAsync called:");
        _logger.LogInformation($"  - Message ID: {messageDto.Id}");
        _logger.LogInformation($"  - IV length: {messageDto.IV?.Length ?? 0}");
        _logger.LogInformation($"  - Salt length: {messageDto.Salt?.Length ?? 0}");

        var sender = await _dbContext.Users
            .FirstOrDefaultAsync(u => u.UserName == messageDto.From);

        if (sender == null)
        {
            throw new InvalidOperationException($"User '{messageDto.From}' not
found.");
        }

        //  Зберігаємо контент як є (вже зашифрований клієнтом)
        string contentToStore = messageDto.Content;

        var message = new Message
```

```
    {
        Id = messageDto.Id ?? Guid.NewGuid().ToString(),
        FromUser = messageDto.From,
        ToUser = messageDto.To,
        Content = contentToStore,              //  Зашифрований
        IV = messageDto.IV ?? string.Empty,     //  Base64 (16 байт)
        Salt = messageDto.Salt ?? string.Empty, //  Base64 (32 байти)
        Timestamp = messageDto.Timestamp,
        Status = messageDto.Status ?? "delivered",
        Edited = messageDto.Edited
    };

    _dbContext.Messages.Add(message);
    await _dbContext.SaveChangesAsync();

    _logger.LogInformation($"[ChatService] Message saved with IV/Salt");
}

//  КРИТИЧНО: При редагуванні НЕ змінюємо Timestamp
public async Task UpdateMessageContent(string messageId,
                                       string newEncryptedContent,
                                       string iv,
                                       string salt,
                                       string userName)
{
    var message = await _dbContext.Messages
        .FirstOrDefaultAsync(m => m.Id == messageId);

    if (message == null)
```

```
        {
            throw new InvalidOperationException($"Message {messageId} not found.");
        }

        // Перевірка авторства
        if (message.FromUser != userName)
        {
            throw new UnauthorizedAccessException(
                $"User {userName} is not authorized to edit message {messageId}.");
        }

        // Оновлюємо Content, IV та Salt (нові параметри шифрування)
        message.Content = newEncryptedContent;
        message.IV = iv ?? string.Empty;
        message.Salt = salt ?? string.Empty;
        message.Edited = true;
        //Timestamp НЕ змінюємо - залишаємо оригінальний час створення

        await _dbContext.SaveChangesAsync();

        _logger.LogInformation($"[ChatService] Message {messageId} updated with new IV/Salt");
    }
}
```

Обґрунтування:

- Сервер зберігає повідомлення БЕЗ розшифрування
- При редагуванні генеруються нові IV та Salt на клієнті
- Timestamp залишається незмінним при редагуванні

Б.1.4. Модель даних повідомлення

Файл: Message.cs - Entity для зберігання в БД

```
public class Message
{
    public string Id { get; set; }
    public string FromUser { get; set; }
    public string? ToUser { get; set; }

    //  E2EE-шифрування: поля для зберігання зашифрованих даних
    public string Content { get; set; }      // Base64-зашифрований контент
    public string IV { get; set; }           // Вектор ініціалізації (16 байт – 24 симв.)
    public string Salt { get; set; }         // Сіль для PBKDF2 (32 байти – 44 симв.)

    public DateTime Timestamp { get; set; }
    public string Status { get; set; }       // sent, delivered, read
    public bool Edited { get; set; }
    public bool IsFile { get; set; }
    public string? FileName { get; set; }
    public string? ParentMessageId { get; set; }
}
```

Математичне обґрунтування**:**

- Content: Зашифрований AES-256-CBC у форматі Base64;
- IV: 16 байт (128 біт) – 24 символи Base64;
- Salt: 32 байти (256 біт) – 44 символи Base64.

Безпека**:** Навіть знаючи {Content, IV, Salt}, без мастер-ключа K_master обчислювально неможливо відновити plaintext за поліноміальний час (складність брутфорс: 2^{256} операцій $\approx 3.67\times10^{51}$ років).

Б.1.5. База даних (Data/)

ApplicationDbContext.cs – EF Core контекст

```
public class ApplicationDbContext : DbContext {
    public DbSet<Message> Messages { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {
        modelBuilder.Entity<Message>(entity => {
            //  КРИТИЧНО: Індекс для швидкого пошуку зашифрованих повідомлень
            entity.HasIndex(m => new { m.FromUser, m.ToUser, m.Timestamp });

            //  Поля E2EE-шифрування мають NVARCHAR(MAX) для Base64
            entity.Property(m => m.Content).IsRequired();
            entity.Property(m => m.IV).HasMaxLength(24);      // 16 байт – 24 Base64
            entity.Property(m => m.Salt).HasMaxLength(44);    // 32 байти – 44 Base64
        });
    }
}
```

Б.2. Клієнтська частина (Frontend)

Б.2.1. ChatService – клієнтська частина (фрагмент)

Файл: chat.service.ts - End-to-End шифрування

```
import * as CryptoJS from 'crypto-js';
import { environment } from 'src/environments/environment';

@Injectable({ providedIn: 'root' })
```

```
export class ChatService implements OnDestroy {
  private encryptionKey!: CryptoJS.lib.WordArray;
  private encryptionIv!: CryptoJS.lib.WordArray;

  constructor(private httpClient: HttpClient, private authService: AuthService) {
    // Ініціалізація мастер-ключа (зберігається ТІЛЬКИ на клієнті)
    this.encryptionKey = CryptoJS.enc.Base64.parse(environment.encryptionKey);
    this.encryptionIv = CryptoJS.enc.Base64.parse(environment.encryptionIv);
  }

  //  PBKDF2: Деривація унікального сесійного ключа для кожного повідомлення
  private async deriveSessionKey(
    masterKey: string,
    salt: CryptoJS.lib.WordArray
  ): Promise<CryptoJS.lib.WordArray> {
    return new Promise((resolve) => {
      setTimeout(() => {
        const derivedKey = CryptoJS.PBKDF2(masterKey, salt, {
          keySize: 256 / 32,      // 32 байти = 256 біт (AES-256)
          iterations: 10000,      // OWASP-рекомендація для PBKDF2
          hasher: CryptoJS.algo.SHA256
        });
        resolve(derivedKey);
      }, 0);
    });
  }

  //  Відправка повідомлення з шифруванням
```

```
public async sendMessage(
  content: string,
  isPrivate: boolean,
  targetUser: string | null = null
): Promise<void> {
  if (!this.myName) {
    console.error('[sendMessage] User not authenticated');
    return;
  }

  const tempId = this.generateGuid();

  //  Генерація криптографічно стійких параметрів
  const iv = CryptoJS.lib.WordArray.random(16);   // 128 біт для AES
  const salt = CryptoJS.lib.WordArray.random(32); // 256 біт для PBKDF2

  try {
    //  Деривація сесійного ключа через PBKDF2
    const sessionKey = await this.deriveSessionKey(
      environment.encryptionKey,
      salt
    );

    //  Шифрування AES-256-CBC з деривованим ключем
    const encryptedContent = CryptoJS.AES.encrypt(content, sessionKey, {
      iv: iv,
      mode: CryptoJS.mode.CBC,
      padding: CryptoJS.pad.Pkcs7
    }).toString();
```

```
  // Конвертація у Base64 для передачі
  const ivBase64 = iv.toString(CryptoJS.enc.Base64);
  const saltBase64 = salt.toString(CryptoJS.enc.Base64);

  const messageToSend: Message = {
    id: tempId,
    from: this.myName,
    to: targetUser ?? undefined,
    content: encryptedContent,  //  Зашифрований
    iv: ivBase64,               //  16 байт – 24 символи
    salt: saltBase64,           //  32 байти – 44 символи
    timestamp: new Date(),
    status: 'sent',
    edited: false
  };

  // Показуємо plain text в UI (локально)
  const displayMessage = { ...messageToSend, content: content };
  this.addOrUpdateMessage(displayMessage, true);
  // Відправляємо зашифроване через SignalR
  if (isPrivate) {
    await this.chatConnection?.invoke('SendPrivateMessage', messageToSend);
  } else {
    await this.chatConnection?.invoke('SendMessage', messageToSend);
  }

} catch (error) {
  console.error('[sendMessage] Encryption failed:', error);
```

```
  }
}

//  Розшифрування отриманих повідомлень
public async decryptContent(
  encryptedText: string,
  ivBase64?: string,
  saltBase64?: string
): Promise<string> {
  if (!encryptedText || !this.encryptionKey) {
    return '[ПОМИЛКА: Відсутні дані]';
  }

  try {
    // КРОК 1: Визначаємо IV
    const ivToUse = ivBase64
      ? CryptoJS.enc.Base64.parse(ivBase64)
      : this.encryptionIv; // Fallback для старих повідомлень

    // КРОК 2: Визначаємо ключ (деривований або статичний)
    let keyToUse: CryptoJS.lib.WordArray;

    if (saltBase64) {
      //  Новий формат: деривуємо ключ з Salt
      const salt = CryptoJS.enc.Base64.parse(saltBase64);
      keyToUse = await this.deriveSessionKey(environment.encryptionKey, salt);
      console.log('[decryptContent] Using PBKDF2-derived key');
    } else {
      //  Старий формат: статичний ключ (сумісність)
```

```
      keyToUse = this.encryptionKey;
      console.log('[decryptContent] Using static key (legacy)');
    }

    // КРОК 3: Розшифрування AES-256-CBC
    const decryptedBytes = CryptoJS.AES.decrypt(
      encryptedText,
      keyToUse,
      {
        iv: ivToUse,
        mode: CryptoJS.mode.CBC,
        padding: CryptoJS.pad.Pkcs7
      }
    );

    if (decryptedBytes.sigBytes === 0) {
      throw new Error('Decryption returned 0 bytes');
    }
    const decryptedContent = decryptedBytes.toString(CryptoJS.enc.Utf8);
    if (!decryptedContent?.trim()) {
      throw new Error('Decrypted content is empty');
    }
    console.log('[decryptContent]  Success');
    return decryptedContent;
  } catch (error) {
    console.error('[decryptContent] All attempts failed:', error);
    return '[ПОМИЛКА РОЗШИФРУВАННЯ]';
  }
}
```

}

Ключові аспекти:

1. PBKDF2 генерує унікальний сесійний ключ для кожного повідомлення
2. Динамічні IV/Salt забезпечують унікальність шифротексту
3. Мастер-ключ зберігається ВИКЛЮЧНО на клієнті (Zero-Knowledge)
4. Fallback для розшифрування старих повідомлень без Salt

Б.2.2. Редагування повідомлень з новими параметрами

Файл: chat.service.ts - Метод editPrivateMessage

```
public async editPrivateMessage(
  toUser: string,
  messageId: string,
  newContent: string
): Promise<void> {
  if (!this.myName || !this.chatConnection) {
    throw new Error('User not authenticated or no connection');
  }
  // Оптимістичне оновлення UI
  this.updateMessageContentAndEditedStatusInClient(messageId, newContent, true, toUser);
  //  Генерація НОВИХ криптографічних параметрів
  const iv = CryptoJS.lib.WordArray.random(16);
  const salt = CryptoJS.lib.WordArray.random(32);
  const sessionKey = await this.deriveSessionKey(environment.encryptionKey, salt);
  //  Шифрування з новим ключем
  const encryptedContent = CryptoJS.AES.encrypt(newContent, sessionKey, {
    iv: iv,
    mode: CryptoJS.mode.CBC,
    padding: CryptoJS.pad.Pkcs7
```

```
  }).toString();
  // Конвертація в Base64
  const ivBase64 = iv.toString(CryptoJS.enc.Base64);
  const saltBase64 = salt.toString(CryptoJS.enc.Base64);

  try {
    //  Відправка з НОВИМИ IV та Salt
    await this.chatConnection.invoke(
      'EditPrivateMessage',
      messageId,          // ID повідомлення
      encryptedContent,   // Новий зашифрований контент
      ivBase64,           //  Новий IV
      saltBase64,         //  Новий Salt
      toUser              // Отримувач
    );

    console.log(`[ChatService]  Edited message with new IV/Salt`);
  } catch (error) {
    console.error(`[ChatService] Edit failed:`, error);
    this.revertLocalMessageEdit(messageId, toUser);
  }
}
```

Обґрунтування**:** При редагуванні генеруються нові IV та Salt, що забезпечує унікальність шифротексту навіть якщо новий контент ідентичний попередньому.

Б.3. Angular Application Architecture

Б.3.1. Конфігурація шифрування на клієнті

Файл: environment.ts - Зберігання мастер-ключа

```
export const environment = {
  production: false,
  apiUrl: 'http://localhost:5051',

  // КРИТИЧНО: Ключі зберігаються ТІЛЬКИ на клієнті
  // Сервер НЕ має доступу до цих значень (E2EE принцип)
  encryptionKey: 'your-master-key-base64-here',
  encryptionIv: '5nQIu+58DHvGuTmoH5Wf0A==',

  userHashSecretKey: 'your-secret-key-here'
};
```

Безпека**:** Ці ключі НЕ передаються на сервер. Вони використовуються виключно для локального шифрування/дешифрування на клієнтському пристрої.

Б.3.2. Конфігурація бази даних

Файл: appsettings.json - Backend налаштування

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DefaultConnection":
"Server=localhost;Database=ChatApp;Trusted_Connection=True;"
  },
```

```
  "Jwt": {
    "SecretKey": "your-jwt-secret-key-minimum-32-characters",
    "Issuer": "ChatApp",
    "Audience": "ChatAppUsers",
    "ExpirationMinutes": 60
  },
  "ChatSettings": {
    "UserHashSecretKey": "your-hash-secret-key",
    "UserHashExpirationMinutes": 5
  }
}
```

Важливо**:** Секція "Encryption" з ключами AES-256 **ВИДАЛЕНА**, оскільки сервер не повинен мати доступу до ключів шифрування згідно з E2EE принципом.

Основні модулі**:**

- Core Module – сервіси для роботи з API та шифруванням;
- Shared Module – багаторазові компоненти;
- Feature Modules – функціональні модулі (Chat, Auth).

Ключові сервіси**:**

- chat.service.ts – SignalR, маршрутизація та E2EE шифрування;
- auth.service.ts – JWT-автентифікація.

Компоненти**:**

- chat-window.component.ts – головне вікно чату;
- message-item.component.ts – відображення повідомлення;
- aurora-background.component.ts – WebGL анімований фон (OGL).