Національний технічний університет України
“Київський політехнічний інститут
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В. о. завідувача кафедри
_____ Артем ВОЛОКИТА
(підпис)
“__” _____2026 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Комп’ютерні системи та мережі”

спеціальності 123 “Комп’ютерна інженерія”

на тему: Система обміну повідомленнями з шифруванням

Виконав : студент __4__ курсу, групи _ІО-22
(шифр групи)

_____Білик Максим Євгенович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____доцент, к.т.н. Волокита А. М._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) асистент кафедри ОТ Коренко Д. В._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент __доц., к.т.н., доцент кафедри АУТС Павло Катін_____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2026 р.

Національний технічний університет України
“Київський політехнічний інститут
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Комп’ютерні системи та мережі”

спеціальності 123 “Комп’ютерна інженерія”

ЗАТВЕРДЖУЮ
В. о. завідувача кафедри
_____ Артем ВОЛОКИТА
(підпис)
“__” _____2026 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Білика Максим Євгеновича

1. Тема проєкту *Система обміну повідомленнями з шифруванням*
керівник проєкту Волокита Артем Миколайович, доцент, к.т.н.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 11 травня 2026 року №1139-с

2. Термін здачі студентом закінченого проєкту *29 травня 2026 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

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

Розділ 1. Аналіз предметної області та огляд існуючих рішень.

Розділ 2. Аналіз вимог та огляд технологій для розробки системи.

Розділ 3. Проектування системи.

Розділ 4. Дослідження та аналіз розробленої системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, функціональна схема (діаграма класів), алгоритм дій програмного забезпечення.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Коренко Д. В.		

7. Дата видачі завдання «20» січня 2026 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*4.02.2026-11.02.2026*	
2.	*Вивчення та аналіз завдання*	*11.02.2026-14.02.2026*	
3.	*Розробка архітектури та загальної структури системи*	*14.02.2026-28.02.2026*	
4.	*Розробка структур окремих підсистем*	*28.02.2026-10.03.2026*	
5.	*Програмна реалізація системи*	*10.03.2026-20.04.2026*	
6.	*Оформлення пояснювальної записки*	*20.04.2026-30.04.2026*	
7.	*Передзахист*	*10.06.2026*	
8.	*Захист*	*15.06.2026*	

Студент-дипломник _____ Максим БІЛИК
(підпис)

Керівник проєкту _____ Артем ВОЛОКИТА
(підпис)

АНОТАЦІЯ

Об'єктом розробки є система обміну повідомленнями з багаторівневим шифруванням даних. Метою проєкту є підвищення конфіденційності та безпеки електронних комунікацій шляхом створення системи, що забезпечує захист даних як на рівні клієнтського застосунку, так і на рівні зберігання в базі даних. У проєкті реалізовано застосунок для обміну текстовими повідомленнями з клієнтським шифруванням на основі протоколів Double Ratchet та Sender Key, серверним шифруванням через ChaCha20-Poly1305 та RSA, мікросервісною архітектурою на .NET 8 з використанням RabbitMQ, Redis і PostgreSQL, а також клієнтською частиною на Angular. Система забезпечує неможливість доступу до вмісту повідомлень навіть з боку сервера. Програмний продукт розроблено на мовах C# та TypeScript.

Ключові слова: наскрізне шифрування, E2EE, ChaCha20-Poly1305, RSA, мікросервісна архітектура, .NET 8, Angular.

ANNOTATION

The object of development is a messaging system with multi-level data encryption. The aim of the project is to enhance the confidentiality and security of electronic communications by creating a system that ensures data protection both at the client application level and at the database storage level. The project implements a messaging application with client-side encryption based on Double Ratchet and Sender Key protocols, server-side encryption via ChaCha20-Poly1305 and RSA, microservice architecture on .NET 8 using RabbitMQ, Redis and PostgreSQL, and a client-side application built with Angular. The system ensures that message content is inaccessible even from the server side. The software product was developed in C# and TypeScript.

Key words: end-to-end encryption, E2EE, ChaCha20-Poly1305, RSA, microservice architecture, .NET 8, Angular.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467100.002 ТЗ***	Система обміну	3		
			повідомленнями			
			з шифруванням			
			Технічне завдання			
	А4	***ІАЛЦ.467100.003 ПЗ***	Система обміну	114		
			повідомленнями			
			з шифруванням			
			Пояснювальна записка			
	А4	***ІАЛЦ.467100.004 Д1***	Система обміну	1		
			повідомленнями			
			з шифруванням			
			Структурна схема			
			системи			
	А4	***ІАЛЦ.467100.005 Д2***	Система обміну	3		
			повідомленнями			
			з шифруванням			
			Функціональна схема			
			(діаграма класів)			
	А4	***ІАЛЦ.467100.006 Д3***	Система обміну	1		
			повідомленнями			
			з шифруванням			
			Алгоритм дій програмного			
			забезпечення			
	А4	***ІАЛЦ.467100.007 Д4***	Система обміну	42		
			повідомленнями			
			з шифруванням			
			Текст програмного коду			
			основних компонентів			
			системи			

Зм	***Лист***	***№ докум.***	***Підп***	***Дата***	***ІАЛЦ.467100.001 ОА***	
Розроб		Білик М.Є.			*Система обміну повідомленнями з шифруванням* Опис альбому	Літ. / Аркуш: 1 / Аркушів: 1
Перев		Волокита А.М.				***КПІ ім. Ігоря Сікорського ФІОТ ІО-22***

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Система обміну повідомленнями з шифруванням»

Київ – 2026

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку системи захищеного обміну повідомленнями з наскрізним шифруванням, а також на подальшу підтримку та вдосконалення розробленої системи.

Областю застосування даної системи є корпоративні та приватні комунікації з підвищеними вимогами до конфіденційності, передача службової та комерційно чутливої інформації, захищений обмін повідомленнями в організаціях державного та приватного секторів, а також будь-яке середовище де критично важливо унеможливити несанкціонований доступ до змісту переписки з боку третіх осіб, включаючи адміністраторів серверної інфраструктури.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для виконання даної роботи є завдання на здобуття кваліфікаційно-освітнього рівня «бакалавр комп'ютерної інженерії», затверджене кафедрою обчислювальної техніки факультету інформатики та обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є поліпшення рівня конфіденційності та цілісності переданих даних засобами системи обміну повідомленнями із застосуванням криптографічного шифрування..

4 ДЖЕРЕЛА РОЗРОБКИ

Інформаційну базу дипломного проєкту складають офіційна технічна документація, інтернет-публікації та статті за відповідною тематикою, а також науково-технічна література.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Простий і інтуїтивно-зрозумілий інтерфейс системи.
- Надати можливість користувачам передавати повідомлення.
- Надати можливість користувачам переглядати розшифровані повідомлення.
- Надати вичерпну та зрозумілу документацію для розробленого продукту.

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- Visual Studio IDE або JetBrains Rider IDE.

5.3. Вимоги до апаратної частини

- ЦП будь-якої архітектури (x-86 або ARM).
- ROM не менше ніж 64 ГБ.
- **RAM не менше ніж 4 ГБ.**

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	*4.02.2026-11.02.2026*
Вивчення та аналіз завдання	*11.02.2026-14.02.2026*
Розробка архітектури та загальної структури системи	*14.02.2026-28.02.2026*
Розробка структур окремих частин системи	*28.02.2026-10.03.2026*
Програмна реалізація системи	*10.03.2026-20.04.2026*
Виправлення помилок	*20.04.2026-23.04.2026*
Оформлення пояснювальної записки	*23.04.2026-30.04.2026*

Пояснювальна записка

до дипломного проєкту

на тему: «Система обміну повідомленнями з шифруванням»

Київ – 2026

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

2FA	(Two-Factor Authentication) Двофакторна автентифікація
AEAD	(Authenticated Encryption with Associated Data) Автентифіковане шифрування з асоційованими даними
AES	(Advanced Encryption Standard) Стандарт симетричного шифрування
AMQP	(Advanced Message Queuing Protocol) Протокол черг повідомлень
API	(Application Programming Interface) Інтерфейс програмування застосунків
ARX	(Add-Rotate-XOR) Операції додавання, зсуву та XOR
E2EE	(End-to-End Encryption) Наскрізне шифрування
FSD	(Feature-Sliced Design) Архітектурна методологія для фронтенду
HTTP	(HyperText Transfer Protocol) Протокол передачі гіпертексту
HTTPS	(HyperText Transfer Protocol Secure) Захищений протокол передачі гіпертексту
JWT	(JSON Web Token) Токен автентифікації
PFS	(Perfect Forward Secrecy) Пряма секретність
RSA	(Rivest-Shamir-Adleman) Алгоритм асиметричного шифрування
SHA	(Secure Hash Algorithm) Алгоритм безпечного хешування
SPA	(Single Page Application) Односторінковий застосунок
TCP	(Transmission Control Protocol) Протокол керування передачею
TLS	(Transport Layer Security) Протокол захисту транспортного рівня
X3DH	(Extended Triple Diffie-Hellman) Розширений потрійний протокол Діффі-Хеллмана
ACID	(Atomicity, Consistency, Isolation, Durability) Властивості транзакцій бази даних

GCM	(Galois/Counter Mode) Режим шифрування з автентифікацією
OAEP	(Optimal Asymmetric Encryption Padding) Оптимальне асиметричне заповнення для шифрування
NGINX	(Engine X) Веб-сервер та зворотний проксі-сервер
ORM	(Object-Relational Mapping) Об'єктно-реляційне відображення
SSL	(Secure Sockets Layer) Протокол захищених з'єднань
SIM	(Subscriber Identity Module) Модуль ідентифікації абонента
IDE	(Integrated Development Environment) Інтегроване середовище розробки
XMPP	(Extensible Messaging and Presence Protocol) Протокол обміну повідомленнями
SGX	(Software Guard Extensions) Розширення захисту програмного забезпечення Intel

ВСТУП

В умовах інтенсивного розвитку інформаційно-комунікаційних технологій системи обміну повідомленнями займають важливе місце у забезпеченні взаємодії між користувачами в цифровому середовищі. Месенджери широко застосовуються для передачі текстових повідомлень та мультимедійних даних, що зумовлює зростання вимог до захисту переданих даних. У зв’язку з цим питання забезпечення конфіденційності та цілісності інформації в системах обміну повідомленнями набувають особливої актуальності.

Актуальність даної роботи обумовлена постійним зростанням кількості кіберзагроз, спрямованих на перехоплення, несанкціонований доступ або модифікацію повідомлень у процесі їх передачі мережею. Використання незахищених або недостатньо захищених каналів зв’язку створює ризики витоку персональних та службових даних, що може призвести до фінансових, репутаційних та правових наслідків. Тому впровадження сучасних криптографічних методів захисту інформації є необхідною умовою функціонування безпечних месенджерів.

Особливе значення у контексті інформаційної безпеки мають системи обміну повідомленнями з підтримкою наскрізного шифрування, при якому доступ до змісту повідомлень мають виключно безпосередні учасники комунікації. Такий підхід дозволяє усунути можливість доступу до конфіденційних даних з боку третіх осіб, у тому числі адміністраторів серверної інфраструктури, та забезпечує високий рівень довіри до системи. Разом із цим виникає необхідність у комплексному підході до проєктування архітектури месенджера з урахуванням вимог до управління ключами, автентифікації користувачів і захисту каналів зв’язку.

Об’єктом дослідження у даній роботі є процеси обміну повідомленнями в клієнт-серверних інформаційних системах. Предметом дослідження є методи та засоби забезпечення інформаційної безпеки в месенджерах з використанням криптографічних алгоритмів шифрування та механізмів автентифікації.

Метою дипломного проєкту є покращення підходів до забезпечення інформаційної безпеки в системах обміну повідомленнями на основі використання сучасних криптографічних методів захисту даних.

Для досягнення поставленої мети у роботі необхідно вирішити такі основні завдання: проаналізувати існуючі рішення у сфері захищених месенджерів; дослідити сучасні криптографічні алгоритми та протоколи, що застосовуються для захисту даних; спроєктувати архітектуру системи обміну повідомленнями з урахуванням вимог інформаційної безпеки; реалізувати програмний продукт та провести його тестування з метою оцінки ефективності запропонованих рішень.

Таким чином, предметна область даного дослідження охоплює сукупність теоретичних і практичних аспектів забезпечення інформаційної безпеки в системах обміну повідомленнями, зокрема застосування сучасних криптографічних методів, механізмів автентифікації та захисту каналів зв’язку. Увага зосереджується на підходах, що дозволяють гарантувати конфіденційність, цілісність і достовірність переданої інформації в умовах зростання кіберзагроз. Розуміння зазначених принципів є основою для подальшого обґрунтування архітектурних рішень та формування ефективної моделі безпечної системи обміну повідомленнями.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Аналіз предметної області

У сучасних умовах розвитку технологій системи віддаленого обміну повідомленнями відіграють ключову роль у забезпеченні оперативної взаємодії між користувачами в мережевих середовищах. Зростання обсягів цифрової комунікації супроводжується підвищенням вимог до рівня захисту інформації, що передається, зокрема щодо конфіденційності, цілісності та автентичності повідомлень. У зв'язку з цим предметна область розробки захищених месенджерів поєднує технічні, організаційні та криптографічні аспекти побудови надійних систем обміну даними.

Базовим елементом, що формує функціональне ядро досліджуваної системи, є повідомлення як структурована цифрова сутність, що формується користувачем і передається через мережу до кінцевого адресата. Повідомлення можуть містити текстову або мультимедійну інформацію.

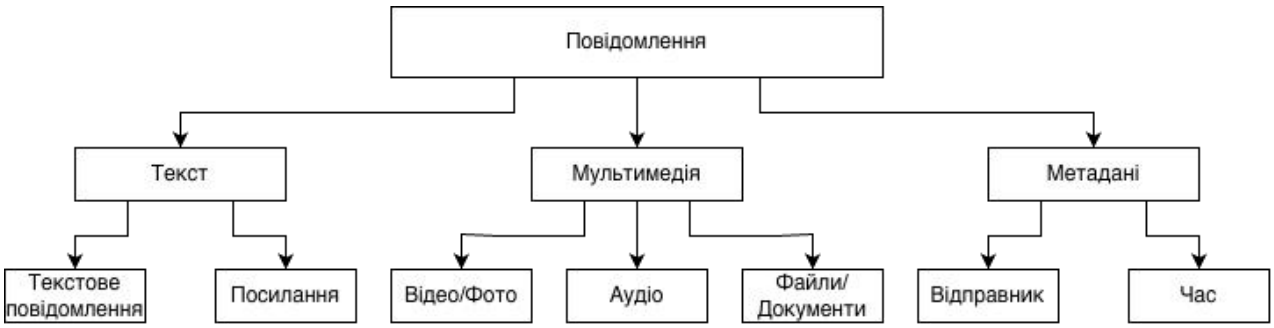

Рисунок 1.1 – Структура цифрової сутності

Крім того, вони мають обмежений життєвий цикл, який включає етапи створення, шифрування, передачі, зберігання, дешифрування та відображення.

Це вимагає впровадження механізмів, здатних гарантувати цілісність даних та виключати можливість несанкціонованого втручання.

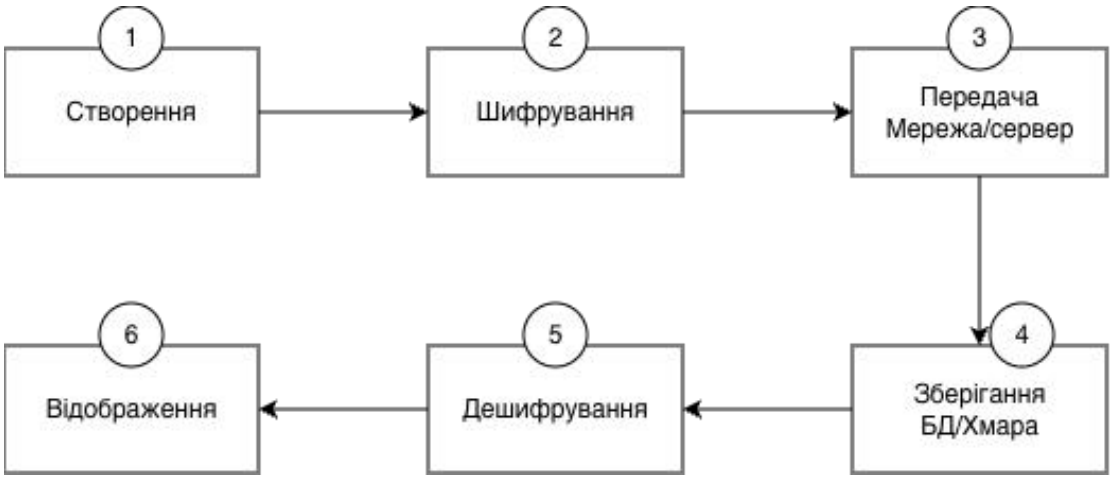

Рисунок 1.2 – Життєвий цикл повідомлення

Важливим аспектом предметної області є забезпечення інформаційної безпеки в процесі обміну повідомленнями. Це передбачає реалізацію механізмів, які унеможливлюють несанкціонований доступ до змісту повідомлень, їх підміну або спотворення. Захист інформації повинен бути незалежним від рівня довіри до мережевої інфраструктури та проміжних компонентів системи, що відповідає сучасним концепціям побудови безпечних інформаційних систем.

Окрему роль у предметній області відіграють механізми автентифікації та контролю доступу, які дозволяють однозначно ідентифікувати учасників обміну повідомленнями. Коректна реалізація цих механізмів сприяє зменшенню ризиків внутрішніх загроз і підвищує загальний рівень довіри до системи.

Предметна область також охоплює питання зберігання даних та управління історією повідомлень. У системах обміну повідомленнями важливо забезпечити баланс між доступністю інформації для користувачів і дотриманням вимог до конфіденційності. Це передбачає використання підходів, які дозволяють зберігати дані у захищеному вигляді та обмежувати доступ до них виключно авторизованим учасникам. У різних юрисдикціях

діють нормативно-правові акти, які регламентують порядок обробки, зберігання та передачі інформації, що може містити персональні або конфіденційні відомості. Тому при проєктуванні системи необхідно враховувати принципи мінімізації збирання даних, обмеження доступу та прозорості обробки інформації, що безпосередньо впливає на архітектурні рішення в межах предметної області.

Суттєве значення має і клієнтська частина системи, яка відповідає за взаємодію користувача з програмним продуктом. Інтерфейс повинен забезпечувати зручність використання, своєчасне відображення отриманих повідомлень та коректну обробку помилкових ситуацій без зниження рівня безпеки.

Окремої уваги в межах предметної області потребують питання масштабованості та адаптивності систем. У процесі експлуатації кількість користувачів, обсяг переданих повідомлень та навантаження на інфраструктуру можуть суттєво змінюватися, що вимагає можливості еволюційного розвитку системи без порушення її цілісності та безпеки. Архітектура таких систем повинна передбачати розширення функціональних можливостей, інтеграцію додаткових сервісів та підтримку зростаючих обсягів даних без необхідності повної перебудови програмного продукту.

Таким чином, предметна область даного дослідження охоплює сукупність організаційних та технологічних аспектів побудови захищених систем обміну повідомленнями, спрямованих на забезпечення конфіденційності, цілісності та автентичності інформації, а також на підтримку надійної та безперервної взаємодії між користувачами в умовах цифрового середовища. Розуміння зазначених принципів є підґрунтям для подальшого проєктування архітектури системи та реалізації її функціональних компонентів.

1.2 Огляд існуючих рішень

У сучасному цифровому середовищі, де значна частина особистої, ділової та організаційної комунікації здійснюється в електронному форматі, системи обміну повідомленнями стали важливою складовою інформаційної інфраструктури. Вони використовуються для оперативної взаємодії між користувачами, координації робочих процесів та передавання службової інформації, яка може мати конфіденційний або критичний характер, що зумовлює підвищені вимоги до її захисту.

Із поширенням цифрових каналів комунікації зростає кількість загроз, пов’язаних із несанкціонованим доступом, перехопленням, модифікацією або підміною повідомлень. У централізованих системах значна частина контролю зосереджена на серверній стороні, що підвищує ризики порушення конфіденційності та залежності від єдиного центру управління. У зв’язку з цим особливої актуальності набуває забезпечення цілісності та приватності переданих даних.

Ключовим аспектом предметної області є формування довіри між учасниками комунікації. Користувачі повинні мати впевненість у тому, що повідомлення не були змінені в процесі передавання, а особа відправника може бути коректно ідентифікована. Порушення цих вимог може призвести до витоку інформації та негативних наслідків.

Окрім безпекових характеристик, системи обміну повідомленнями повинні відповідати вимогам надійності та доступності, забезпечувати стабільну роботу за умов зростання навантаження та підтримувати асинхронну взаємодію між користувачами. Сукупність зазначених факторів визначає необхідність подальшого вдосконалення підходів до проєктування захищених систем обміну повідомленнями.

Крім того, важливими характеристиками таких систем є

масштабованість та стійкість до збоїв, що забезпечують безперервність комунікації.

1.2.1 Telegram

Telegram - це хмарний багатоплатформний месенджер, заснований братами Миколою та Павлом Дуровими у 2013 році. Головна концепція, покладена в основу продукту - надання користувачам максимальної свободи та конфіденційності в умовах, коли великі технологічні корпорації активно збирали персональні дані своїх клієнтів[1].

Наприкінці 2023 - на початку 2024 року дослідники у сфері кібербезпеки виявили масштабний витік облікових даних, що поширювався через мережу Telegram-каналів. Загальний обсяг архіву склав 122 ГБ, а відомий дослідник безпеки Трой Хант підтвердив, що він містив 361 мільйон унікальних електронних адрес у поєднанні з паролями та іншими ідентифікаційними даними.

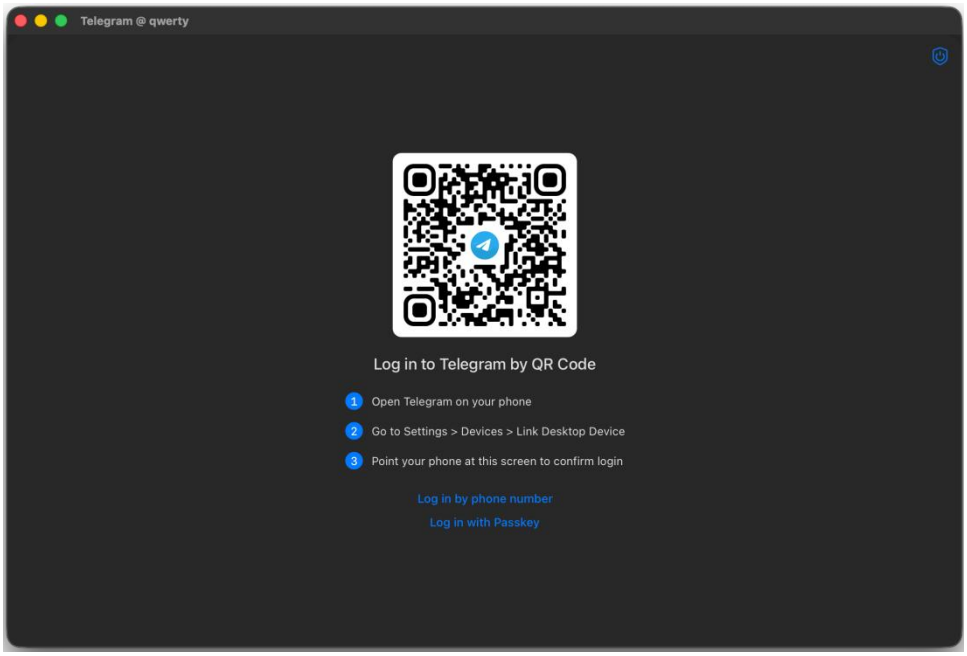

Рисунок 1.3 – Інтерфейс автентифікації у Telegram Desktop

1.2.1.1 Захист даних та шифрування

Технічною основою платформи є власний протокол шифрування MTProto (Mobile Transport Protocol), розроблений командою Дурова. Цей протокол поєднує алгоритми симетричного та асиметричного шифрування для захисту даних під час передачі[1]. Однак саме цей аспект викликає найбільше дискусій серед фахівців у сфері кібербезпеки.

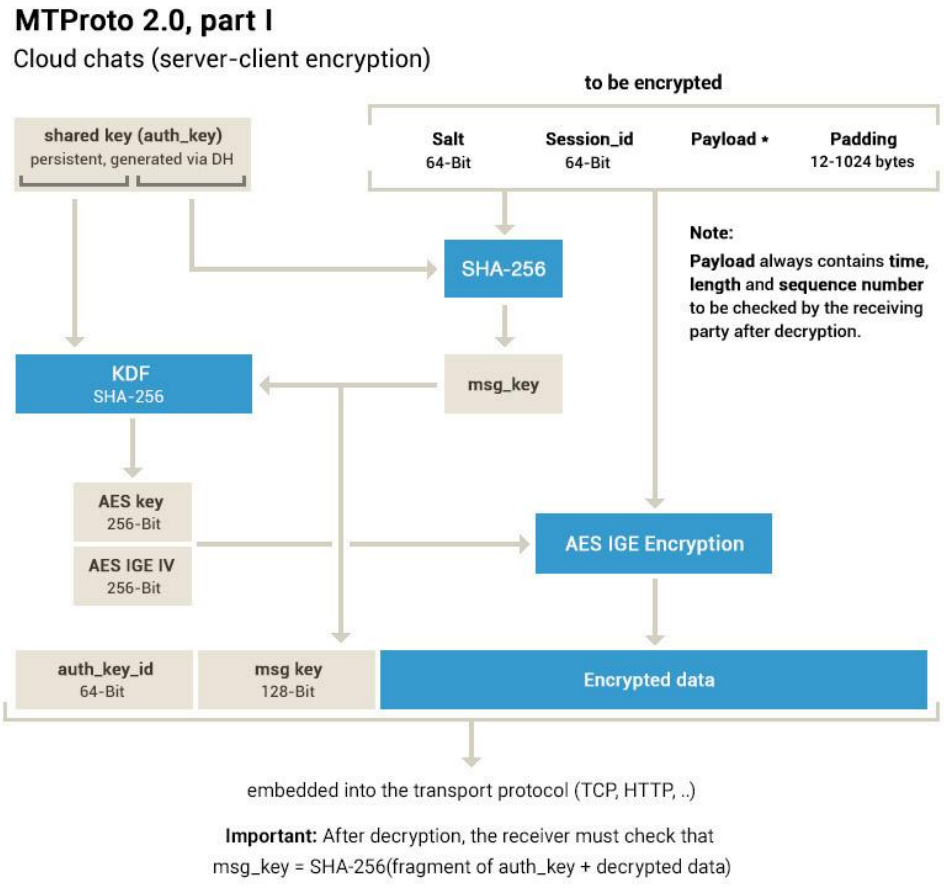

Рисунок 1.4 – Блок схема MTProto

Протокол, є нестандартним рішенням, що викликає закономірне занепокоєння серед криптографів та фахівців з кібербезпеки. Загальноприйнята практика у сфері захисту інформації передбачає використання добре вивчених і широко перевірених протоколів - наприклад, протоколу Signal (використовується у WhatsApp) або TLS. Розробка власного криптографічного протоколу є складним завданням, що вимагає тривалої

незалежної перевірки [5]. Він зазнавав критики за відсутність повноцінних та регулярних незалежних аудитів безпеки. Більшість аналізів, доступних публічно, проводились окремими дослідниками, а не авторитетними організаціями чи дослідницькими групами у рамках систематичної перевірки. Відсутність систематичного аудиту не дозволяє зробити однозначний висновок про надійність протоколу [5].

Незважаючи на позиціонування як захищеного месенджера, Telegram мав у своїй історії ряд задокументованих вразливостей. У 2018 році дослідники виявили, що десктопна версія Telegram розкривала реальні IP-адреси під час ініціювання однорангових голосових дзвінків. Ще одна вразливість дозволяла зловмисникам впроваджувати шкідливий код через текстові рядки з написанням справа наліво (RTL Override).

Питання захисту даних є центральним у позиціонуванні Telegram. Платформа пропонує кілька рівнів безпеки залежно від типу переписки. Для стандартних хмарних чатів застосовується шифрування типу «клієнт-сервер», яке базується на 256-бітному симетричному алгоритмі AES, 2048-бітному шифруванні RSA та захищеному обміну ключами за протоколом Діффі-Хеллмана [2]. Ці стандарти є промисловими та загальновизнаними в криптографічній спільноті.

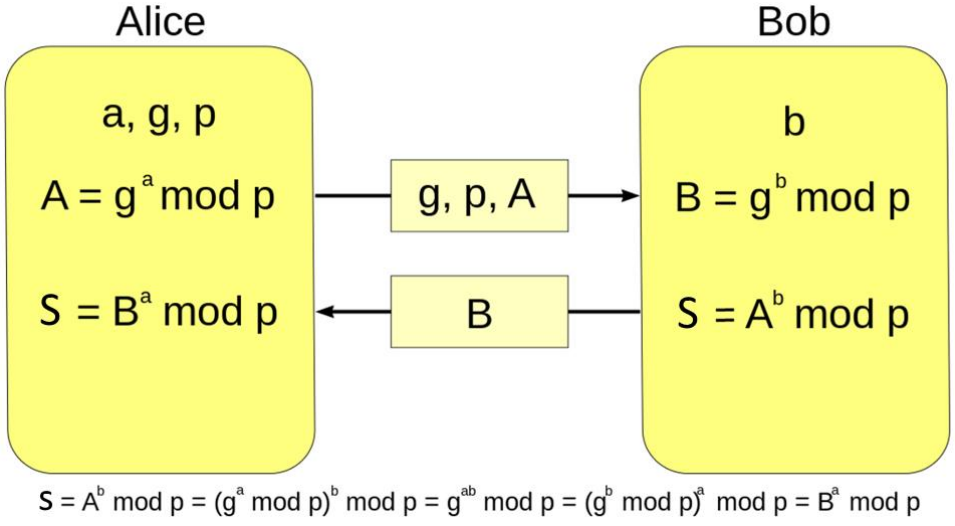

Рисунок 1.5 – Блок схема протоколу Діффі-Хеллмана

Окремою функцією є «Секретні чати» (Secret Chats), що реалізують повноцінне наскрізне шифрування (End-to-End Encryption, E2EE). У такому режимі повідомлення шифруються безпосередньо на пристрої відправника і можуть бути розшифровані виключно на пристрої одержувача. Навіть сервери Telegram не мають технічної можливості отримати доступ до вмісту таких переписок. Секретні чати не зберігаються у хмарі та прив'язані до конкретного пристрою [3].

Для захисту облікового запису від несанкціонованого доступу Telegram підтримує двофакторну автентифікацію (2FA). Після її активації для входу в обліковий запис на новому пристрої, окрім одноразового коду, надісланого на номер телефону, потрібно ввести додатковий пароль. Це суттєво ускладнює захоплення облікового запису навіть у разі, якщо зловмисник отримає доступ до SIM-карти користувача через атаку типу SIM Swap[3].

Важливо зазначити, що 2FA у Telegram є необов'язковою функцією і не активована за замовчуванням. Користувачі, які не усвідомлюють цього ризику, можуть залишатися вразливими до атак, спрямованих на перехоплення SMS-кодів підтвердження.

1.2.1.2 Серверна інфраструктура Telegram

Telegram використовує децентралізовану розподілену інфраструктуру дата-центрів (Data Centers, DC), що дозволяє забезпечити низьку затримку передачі даних для користувачів у різних регіонах світу та стійкість системи до збоїв [6].

Станом на поточний момент Telegram підтримує 5 основних дата-центрів (DC1-DC5), кожен з яких може функціонувати автономно [7]:

- DC1 і DC3 - розташовані у місті Маямі (США);
- DC2 і DC4 - розташовані в Амстердамі (Нідерланди);

- DC5 - розташований у Сінгапурі.

Кожен обліковий запис прив'язується до конкретного дата-центру в момент реєстрації. Клієнт підключається до найближчого вузла за допомогою протоколу MTProto - власного протоколу Telegram, оптимізованого для мобільних пристроїв. Зміна дата-центру може відбутися автоматично при тривалому використанні з іншого географічного регіону [6].

MTProto підтримує кілька транспортних протоколів для з'єднання з серверами: TCP, HTTP, HTTPS, WebSocket (WS) та WebSocket over HTTPS (WSS). Це забезпечує сумісність з різними мережевими середовищами та дозволяє обходити певні мережеві обмеження [8].

1.2.1.3 Недоліки платформи

Найсуттєвішим і найбільш обговорюваним недоліком Telegram є відсутність наскрізного шифрування (E2EE) у стандартних хмарних чатах. На відміну від Signal та WhatsApp, де E2EE застосовується автоматично для всіх переписок, у Telegram ця функція доступна лише у «Секретних чатах», активувати яку необхідно вручну.

Стандартні чати, груповий обмін, а також канали - все це зберігається на серверах Telegram у зашифрованому вигляді, однак компанія теоретично має технічну можливість розшифрувати ці дані. Переважна більшість пересічних користувачів не підозрює про цю відмінність і помилково вважає всі свої переписки захищеними рівнозначно [2].

Практична проблема полягає також у тому, що «Секретні чати» мають суттєві функціональні обмеження: вони недоступні з веб-версії Telegram і прив'язані до конкретного пристрою, що унеможливлює їх перегляд на іншому гаджеті. Така незручність змушує багатьох користувачів відмовлятися від захищеного режиму на користь зручнішого, але менш захищеного.

Telegram тривалий час критикували за недостатню модерацію незаконного контенту. Платформа використовувалася для розповсюдження матеріалів, пов'язаних з незаконною торгівлею, координацією злочинної діяльності та поширенням дезінформації. Хоча компанія вживає певних заходів для боротьби з цими явищами, ефективність цих заходів залишається предметом критики з боку правоохоронних органів та дослідників.

Попри відкритість клієнтської частини, серверний код Telegram залишається закритим. Це означає, що незалежна перевірка того, яким чином насправді обробляються та зберігаються дані на серверах компанії, є неможливою. Будь-які заяви компанії щодо безпеки серверної обробки даних неможливо підтвердити або спростувати без доступу до відповідного коду [4].

Закритість серверного коду є фундаментальним обмеженням з точки зору верифікованої безпеки. Відкрита модель безпеки (Security Through Transparency) передбачає, що незалежні дослідники можуть перевіряти як клієнтську, так і серверну частини системи. Telegram же пропонує лише часткову прозорість, що є компромісним рішенням між повною відкритістю та традиційною закритою моделлю.

1.2.2 WhatsApp

WhatsApp - один із найпопулярніших месенджерів, заснований у 2009 році Яном Коумом та Браяном Ектоном. У 2014 році компанія Facebook (нині Meta Platforms) придбала платформу. Ця угода суттєво вплинула на його подальший розвиток з точки зору обробки персональних даних.[9]

Технічною основою безпеки WhatsApp є протокол Signal - розробка з відкритим вихідним кодом, яку вважають одним із найнадійніших криптографічних рішень для захищеного обміну повідомленнями. Протокол реалізує повноцінне наскрізне шифрування (End-to-End Encryption, E2EE) за

замовчуванням для всіх переписок, а також голосових і відеодзвінків. Саме використання цього протоколу формує міцну технічну основу безпеки платформи.

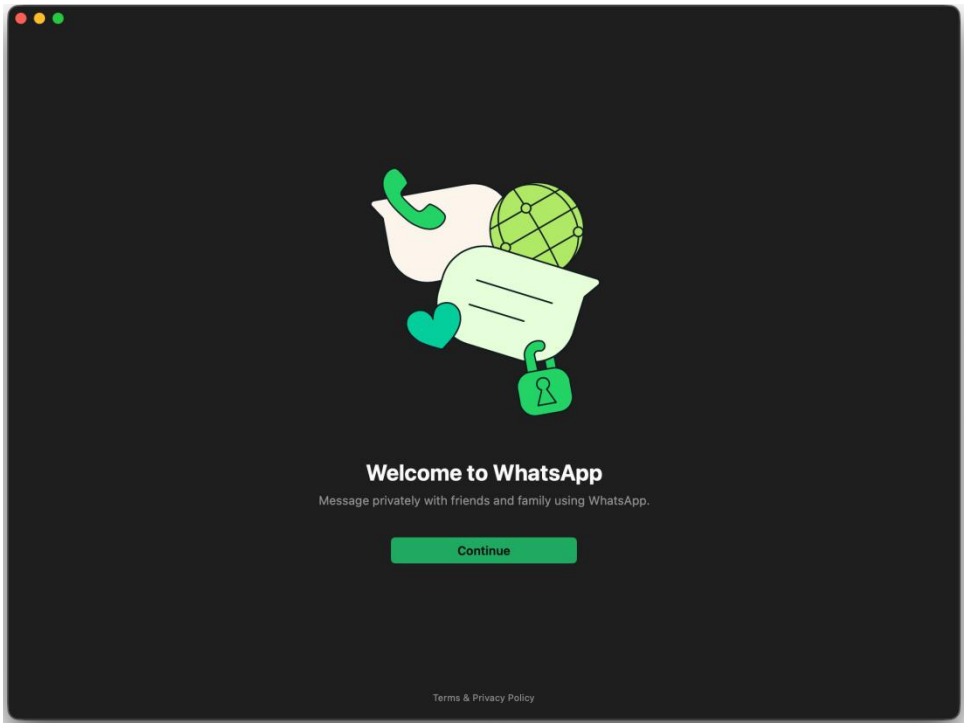

Рисунок 1.6 – Інтерфейс автентифікації у WhatsApp Desktop

Водночас приналежність WhatsApp до екосистеми Meta накладає суттєві обмеження на рівень конфіденційності. Попри те що вміст повідомлень залишається захищеним, компанія збирає значний обсяг метаданих, які можуть розкривати інформацію про поведінку та зв'язки користувача навіть без доступу до змісту переписок.

1.2.2.1 Захист даних та шифрування

Головною технічною перевагою WhatsApp є те, що наскрізне шифрування активоване для всіх переписок автоматично, без будь-яких дій з боку користувача. Протокол, що лежить в основі системи шифрування, реалізує подвійний механізм «Double Ratchet Algorithm» у поєднанні з X3DH (Extended Triple Diffie-Hellman), що забезпечує пряму секретність (Forward Secrecy) -

компрометація одного ключа не дозволяє розшифрувати попередні повідомлення [10].

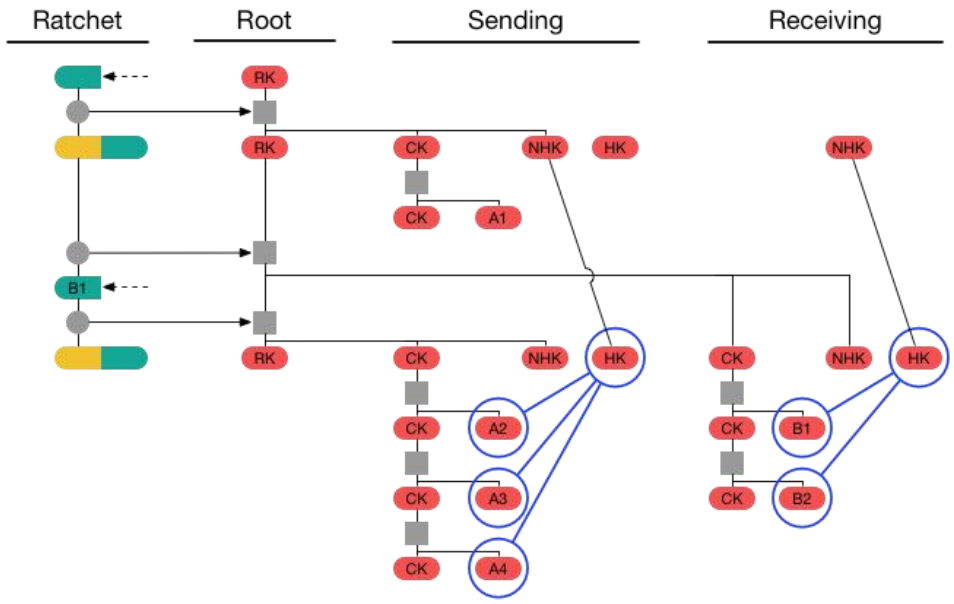

Рисунок 1.7 – Блок схема Double Ratchet Algorithm

Суттєвою перевагою порівняно з рядом конкурентів, зокрема Telegram, є наявність наскрізного шифрування, де E2EE у групових чатах відсутнє. Згідно з офіційною позицією компанії, навіть співробітники WhatsApp та Meta не мають технічної можливості отримати доступ до вмісту зашифрованих переписок.

У 2021 році WhatsApp запровадив функцію наскрізного шифрування резервних копій переписок у хмарних сховищах (Google Drive та iCloud). До цього резервні копії зберігалися у відкритому вигляді, що створювало значну вразливість - правоохоронні органи могли отримати вміст переписок від Google або Apple без відома WhatsApp. Після оновлення 2021 року резервні копії з увімкненим E2EE стали недоступними навіть для хмарних провайдерів. У жовтні 2025 року Meta оголосила про можливість увімкнення E2EE для резервних копій за допомогою ключів-паролів (passkeys) [11].

WhatsApp підтримує двофакторну автентифікацію (2FA), яка вимагає введення шестизначного PIN-коду при реєстрації на новому пристрої. Це суттєво знижує ризик несанкціонованого захоплення облікового запису через атаку типу SIM Swap. Додатково платформа повідомляє користувача про кожну нову реєстрацію за допомогою його номера телефону [10].

1.2.2.2 Серверна інфраструктура WhatsApp

WhatsApp функціонує на базі власної інфраструктури Meta Platforms. Після придбання у 2014 році сервіс поступово мігрував із хмари IBM SoftLayer до власних дата-центрів Meta[12]. Сьогодні всі серверні потужності WhatsApp розміщені у дата-центрах Meta, що дозволяє використовувати переваги масштабної корпоративної інфраструктури.

Мережа дата-центрів Meta охоплює 24 майданчики з загальним обсягом інвестицій близько 30 мільярдів доларів. У США розміщено 20 дата-центрів, ще 4 - у Європі та Азіатсько-Тихоокеанському регіоні [13]. Ключові локації включають:

- США: Принвілл (Орегон), Форт-Ворт (Техас), Форест-Сіті (Північна Кароліна), Канзас-Сіті (Міссурі) та інші;
- Європа: Лулео (Швеція), Оденсе (Данія), Клоніфарт (Ірландія);
- Азіатсько-Тихоокеанський регіон: Сінгапур.

Серверна частина WhatsApp побудована на базі мови програмування Erlang/OTP - платформи, розробленої ще у 1980-х роках для телекомунікаційних систем Ericsson. Головна перевага Erlang полягає в моделі легковагових процесів: кожне з'єднання з користувачем працює як окремий ізольований процес у середовищі виконання BEAM. Один сервер здатен обробляти від 1 до 15 мільйонів одночасних з'єднань [14].

Для маршрутизації повідомлень WhatsApp використовує кастомізований XMPP-сервер на базі Ejabberd - відкритого програмного забезпечення, значно модифікованого командою WhatsApp. Власний бінарний протокол FunXMPP, що є стиснутою версією стандартного XMPP, скорочує використання трафіку на 50-70% порівняно з текстовим XMPP - критично важлива оптимізація для мобільних мереж [15].

Операційна система серверів - FreeBSD, обрана завдяки перевагам мережевого стеку при роботі з великою кількістю одночасних з'єднань. Для зберігання маршрутних даних і черг повідомлень використовується Mnesia - розподілена in-memory база даних, тісно інтегрована з Erlang. Для зберігання структурованих даних - профілів користувачів та метаданих - застосовується MySQL із горизонтальним шардингом [15].

Важливий принцип архітектури: WhatsApp не зберігає повідомлення на сервері після їх успішного доставлення. Повідомлення перебувають у черзі лише до моменту, поки одержувач не підключиться і не отримає їх. Після підтвердження доставки вміст видаляється з серверів, що суттєво спрощує інфраструктуру зберігання [16].

Для медіаконтенту (фото, відео, документи) WhatsApp використовує CDN (Content Delivery Network) - розподілену мережу серверів для прискорення завантаження файлів у різних регіонах. Медіафайли зберігаються у зашифрованому вигляді та видаляються через 30 днів після відправки, якщо не були отримані [10].

1.2.2.3 Недоліки платформи

Найсерйознішим недоліком WhatsApp з точки зору конфіденційності є збір та обробка значного обсягу метаданих. Незважаючи на те, що вміст повідомлень захищений шифруванням, платформа збирає широкий спектр

непрямих даних: ідентифікатори пристрою та мережі, IP-адресу, інформацію про частоту та тривалість взаємодій, геолокаційні дані, відомості про групове членство, використання функцій застосунку та поведінкові патерни.

Відповідно до оновленої політики конфіденційності 2021 року та умов використання WhatsApp Business, зібрані метадані передаються іншим сервісам Meta. Це дозволяє Meta будувати детальні поведінкові профілі користувачів, навіть не маючи доступу до вмісту їхніх переписок. Для рекламних систем платформи метадані є надзвичайно цінним ресурсом для таргетування.

У 2024 році Meta відреагувала на понад 78% запитів від правоохоронних органів, що стосувалися WhatsApp. Хоча за наявності E2EE компанія може надавати лише метадані (а не вміст переписок), сам факт настільки активної співпраці з органами влади є суттєвою відмінністю від більш закритих конкурентів.

Попри загалом надійну архітектуру шифрування, WhatsApp неодноразово ставав об'єктом атак та виявлення серйозних вразливостей. У 2025 році у застосунку була виявлена критична вразливість CVE-2025-30259, пов'язана з обробкою PDF-файлів: шкідливий PDF-документ міг обходити механізм пісочниці (sandboxing) та дозволяв виконання довільного коду на пристрої жертви [17].

Попри те що WhatsApp використовує відкритий протокол Signal, власний код застосунку WhatsApp є закритим. Незалежна перевірка того, чи реалізований протокол коректно і чи немає прихованих компонентів, які могли б передавати дані Meta або третім сторонам, є практично неможливою.

Придбання WhatsApp компанією Meta у 2014 році стало чинником постійного репутаційного ризику для платформи. Корпорація Meta має значну кількість гучних скандалів у сфері захисту персональних даних, включаючи «Cambridge Analytica» (2018), численні штрафи від регуляторів ЄС та США, а також

звинувачення у маніпулюванні суспільною думкою через алгоритми соціальних мереж.

1.2.3 Signal

Signal - це некомерційний месенджер з повністю відкритим вихідним кодом, що є продуктом Signal Foundation. Фундаментальна концепція платформи полягає у розробці технологій відкритого стандарту для захисту конфіденційного спілкування, відмові від реклами та збору персональних даних. Фінансування здійснюється виключно за рахунок пожертвувань від користувачів та організацій. За оцінками компанії, операційні витрати Signal становитимуть близько 50 мільйонів доларів на рік до 2025 року [18].

Signal широко визнаний серед криптографів та фахівців з кібербезпеки як еталонний засіб захищеного спілкування. Месенджер підтримує роботу на платформах iOS, Android, Windows, macOS та Linux. Станом на початок 2025 року платформою активно користується близько 70 мільйонів осіб щомісяця [19].

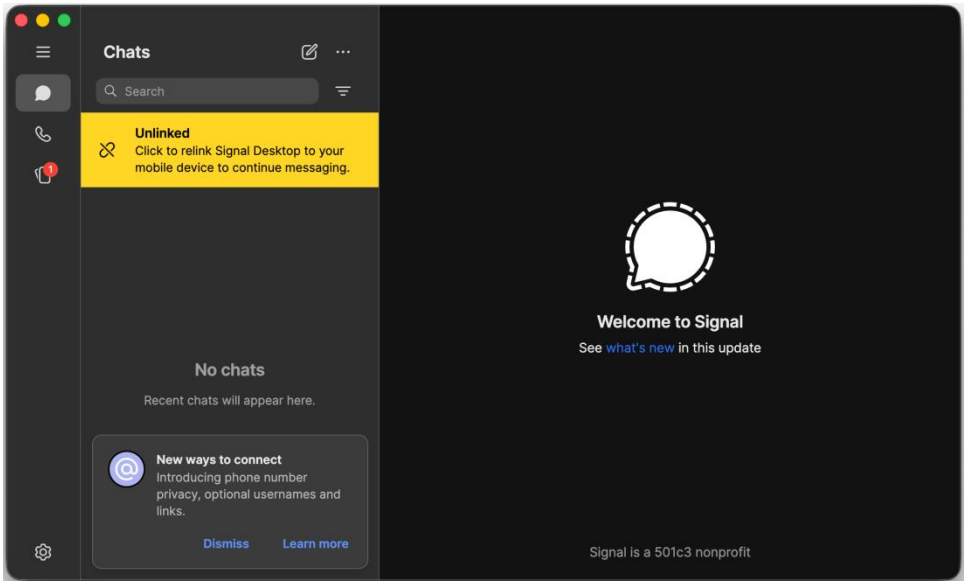

Рисунок 1.8 – Інтерфейс автентифікації у Signal Desktop

Протокол Signal здобув таке визнання у криптографічній спільноті, що був інтегрований WhatsApp, Facebook Messenger та Skype для шифрування власних переписок. Це є унікальним свідченням якості та надійності криптографічного дизайну платформи.

1.2.3.1 Захист даних та шифрування

В основі Signal лежить однойменний відкритий криптографічний протокол, що поєднує кілька передових алгоритмів. Центральними компонентами є алгоритм X3DH (Extended Triple Diffie-Hellman) та алгоритм Double Ratchet. X3DH забезпечує асинхронне встановлення сеансового ключа між двома сторонами без необхідності їхньої одночасної онлайн-присутності. Double Ratchet відповідає за безперервне оновлення ключів шифрування в процесі переписки [20].

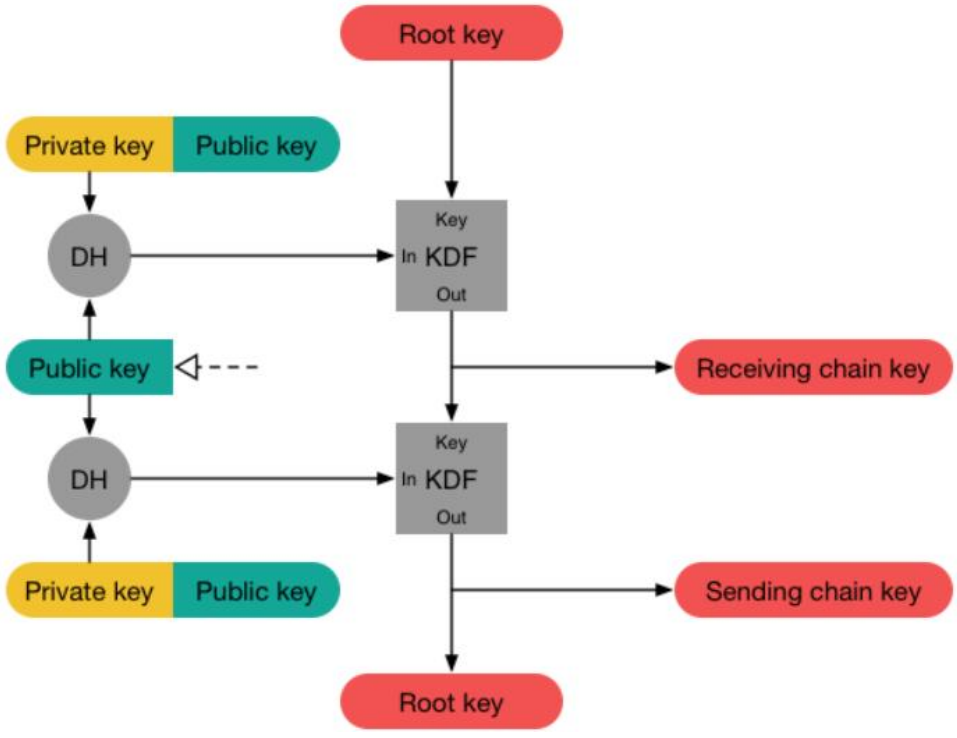

Рисунок 1.9 – Блок схема протоколу Signal

Алгоритм шифрування повідомлень - AES-256 (Advanced Encryption Standard з 256-бітним ключем) у режимі CBC з 128-бітним вектором

ініціалізації та аутентифікацією HMAC-SHA256. Голосові та відеодзвінки шифруються за допомогою протоколу ZRTP, що генерує короткий автентифікаційний рядок (SAS) для верифікації захищеності з'єднання. Усі повідомлення та дзвінки захищені властивістю Perfect Forward Secrecy (PFS): компрометація довгострокового ключа не дозволяє розшифрувати раніше перехоплені повідомлення [20].

У 2024 році протокол Signal було розширено алгоритмом PQXDH (Post-Quantum Extended Diffie-Hellman), що додає постквантовий рівень захисту. Ця особливість забезпечує стійкість до потенційних атак з боку квантових комп'ютерів у майбутньому [21]. Формальний аналіз безпеки протоколу Signal, проведений у 2016 році групою дослідників з чотирьох країн, не виявив жодних суттєвих вразливостей у його криптографічному дизайні. Протокол пройшов кілька незалежних аудитів безпеки - жоден з них не змінив загального висновку про криптографічну коректність [20].

1.2.3.2 Серверна інфраструктура Signal

На відміну від Telegram та WhatsApp, Signal не має власних фізичних дата-центрів. Платформа повністю базується на хмарній інфраструктурі - насамперед Amazon Web Services (AWS) та Microsoft Azure [22]. Це дозволяє Signal динамічно масштабувати потужності без капітальних витрат на власне обладнання, що критично важливо для некомерційної організації. Для зберігання черги повідомлень Signal використовує тимчасове шифроване сховище: після успішної доставки повідомлення одержувачу воно видаляється з черги. Незашифрована інформація на серверах практично відсутня. За даними самої компанії, витрати на зберігання даних складають близько 1,3 мільйона доларів на рік - суттєво менше, ніж у конкурентів, що зберігають дані постійно [18].

Ключовою технічною особливістю інфраструктури Signal є використання технології Intel SGX (Software Guard Extensions) для реалізації захищених анклавів (Secure Enclaves) на серверній стороні. Анклав - це ізольована область процесора, доступ до якої не мають ні операційна система, ні адміністратори серверів, ні навіть сам Signal Foundation. Дані всередині анклаву зашифровані на апаратному рівні, а ключі шифрування ніколи не покидають процесор [23]. Signal використовує технологію SGX-анклавів для двох критично важливих сервісів:

- Contact Discovery Service (виявлення контактів) - дозволяє визначити, які контакти з адресної книги користувача вже зареєстровані в Signal, не розкриваючи цю інформацію серверу. Контакти обробляються виключно всередині анклаву у зашифрованому вигляді. Клієнт виконує Remote Attestation - криптографічну перевірку того, що код, який виконується в анклаві, відповідає опублікованому відкритому коду [23];
- Secure Value Recovery (безпечне відновлення даних) - дозволяє зберігати зашифрований мастер-ключ для відновлення PIN-коду з обмеженою кількістю спроб підбору. Алгоритм Path ORAM (Oblivious RAM) приховує навіть патерни доступу до пам'яті всередині анклаву [24].

Оскільки SGX-анклави з підтримкою Remote Attestation доступні в хмарі переважно через Microsoft Azure (DCsv2/DCsv3 VM), саме ця хмарна платформа є основною для критично важливих приватних сервісів Signal. AWS використовується для інших компонентів інфраструктури [22].

Весь вихідний код серверної частини Signal опублікований у відкритому доступі на GitHub - це унікальна властивість серед масових месенджерів. Будь-який дослідник може перевірити відповідність коду, що виконується на серверах, опублікованому вихідному коду через механізм Remote Attestation [25].

1.2.3.3 Недоліки платформи

Найбільш відчутним практичним недоліком Signal є значно менша база користувачів порівняно з масовими месенджерами - близько 70 мільйонів активних користувачів проти понад 3 мільярдів у WhatsApp та 1 мільярда у Telegram [19]. Феномен мережевого ефекту є структурним обмеженням: цінність платформи для конкретного користувача прямо залежить від кількості його контактів на цій платформі.

Signal все ще вимагає номер телефону для первинної реєстрації. Для частини користувачів, що прагнуть повної анонімності, це є суттєвим обмеженням, оскільки SIM-карта реєструється на паспорт у більшості юрисдикцій. Функція Registration Lock (двофакторна автентифікація) суттєво знижує ризик захоплення облікового запису, проте сама вимога номера телефону залишається теоретичним вектором деанонімізації.

На iOS офіційного механізму хмарного резервного копіювання переписок не існує: у разі втрати пристрою вся історія переписок безповоротно зникає. На Android підтримується локальне шифроване резервне копіювання та резервування на Google Drive, проте процедура відновлення потребує 30-значного ключа безпеки, що є серйозним практичним бар'єром для нетехнічних користувачів.

Порівняно з конкурентами Signal має функціональні обмеження: групові чати обмежені 1000 учасниками (проти 200 000 у Telegram), відсутня система каналів та ботів, відсутній веб-клієнт. Кожна нова функція ретельно аналізується з точки зору конфіденційності, що зумовлює більш повільний темп розвитку функціоналу.

Платформа є централізованим сервісом: усі повідомлення проходять через сервери Signal Foundation, розташовані переважно у США. Хоча ці сервери не мають технічного доступу до вмісту зашифрованих переписок, централізована архітектура утворює єдину точку потенційного

юрисдикційного тиску. Некомерційна модель фінансування несе структурний ризик фінансової нестійкості при оцінюваних витратах 50 мільйонів доларів на рік [18].

1.3 Порівняння існуючих рішень

Для наочного зіставлення розглянутих платформ за ключовими технічними та функціональними параметрами нижче наведено порівняльну таблицю. Вона охоплює критерії, що безпосередньо впливають на вибір месенджера у контексті захищених комунікацій: масштаб аудиторії, тип протоколу шифрування, наявність наскрізного шифрування, прозорість коду, збір метаданих та наявність двофакторної автентифікації.

Таблиця 1.1 – Порівняльна характеристика месенджерів Telegram, WhatsApp та Signal

Критерій	Telegram	WhatsApp	Signal
Активних користувачів	> 1 млрд	> 2,7 млрд	≈ 70 млн
Протокол шифрування	MTProto (власний)	Signal Protocol	Signal Protocol (X3DH + Double Ratchet)
E2EE за замовчуванням	Лише у «Секретних чатах»	Так (вміст), метадані - ні	Так (вміст + метадані)
E2EE у групових чатах	-	+	+

Кінець таблиці 1.1

Критерій	Telegram	WhatsApp	Signal
Відкритий код клієнта	+	-	+
Відкритий код сервера	-	-	+
Збір метаданих	Мінімальний	Розширений (IP, геолокація, поведінка)	Мінімальний (лише номер та дата реєстрації)
Двофакторна автентифікація	Так (необов’язкова)	Так (необов’язкова)	Так (необов’язкова)
Незалежний аудит протоколу	Несистематичний	Так (протокол Signal)	Так (багаторазовий)
Відсутність зовнішньої політики конфіденційності	-	-	-
Відсутність прив'язки до номера телефону	-	-	-

З наведеної таблиці видно, що платформи суттєво розрізняються за підходом до захисту даних. Signal забезпечує найвищий рівень конфіденційності: наскрізне шифрування поширюється на всі типи комунікацій, збір даних зведено до мінімуму, а відкритий код як клієнтської,

так і серверної частини допускає повноцінну незалежну перевірку. WhatsApp забезпечує E2EE вмісту за замовчуванням на базі протоколу Signal, однак розширений збір метаданих у межах екосистеми Meta суттєво знижує реальний рівень приватності. Telegram виявляється найслабшим за критерієм шифрування: E2EE доступне лише у «Секретних чатах», групові переписки його позбавлені, а власний протокол MTProto не проходив систематичного незалежного аудиту.

Попри відмінності у підходах до шифрування, всі три платформи мають спільні структурні обмеження. Всі три вимагають прив'язки до номера телефону при реєстрації. Крім того, кожна з платформ контролюється приватною комерційною організацією, що несе ризик односторонньої зміни умов використання або припинення роботи сервісу.

Зазначені обмеження обумовлюють актуальність побудови власної системи захищеного обміну повідомленнями, яка позбавлена перелічених недоліків: підтримує розгортання на власній інфраструктурі, не вимагає прив'язки до номера телефону та забезпечує повний контроль користувача над криптографічними ключами.

ВИСНОВОК ДО РОЗДІЛУ 1

Проведений аналіз показав, що попри широке використання популярних месенджерів, таких як Signal, WhatsApp, чи Telegram, вони не завжди забезпечують повний контроль над конфіденційністю та незалежну перевірку цілісності переданих даних. У більшості випадків довіра до повідомлень залежить від централізованих серверів або інфраструктури провайдера, що створює потенційні вразливості для несанкціонованого доступу, підміни чи перехоплення інформації. Навіть за наявності наскрізного шифрування або базових механізмів автентифікації, користувачі не можуть незалежно перевірити достовірність переданих повідомлень або їхню незмінність після доставки.

У цьому контексті застосування сучасних криптографічних алгоритмів пропонує концептуально інший підхід. Вбудовані механізми шифрування на різних рівнях системи - від клієнтських застосунків до проміжних сервісів - дозволяють захистити дані незалежно від наявності довіри до центральних компонентів інфраструктури. Використання криптографічних відбитків і механізмів контролю цілісності гарантує, що будь-яка модифікація або спроба підміни повідомлення стане одразу помітною, а автентичність відправника може бути підтверджена без стороннього втручання.

Втім, аналіз наявних рішень показав, що навіть найбільш поширені месенджери обмежені у функціоналі щодо гнучкого застосування шифрування та контролю доступу, особливо у корпоративних або регульованих середовищах. Часто немає можливості налаштувати політики доступу, ведення криптографічних журналів або перевірку автентичності на всіх етапах життєвого циклу повідомлення. Це обмежує їхнє використання у випадках, де критично важливо забезпечити юридичну значущість, доказовість або конфіденційність переданих даних.

З огляду на вищезазначене, виникає потреба у вдосконалені існуючих систем обміну повідомленнями з інтегрованим комплексним криптографічним захистом. Така система має забезпечувати наскрізне шифрування, автентифікацію користувачів та контроль цілісності даних на всіх рівнях обробки, поєднуючи переваги сучасної цифрової комунікації. Гнучкість застосованих криптографічних механізмів дозволить адаптувати систему під різні сценарії використання - від приватних комунікацій до корпоративних або державних структур із підвищеними вимогами до безпеки та юридичної значущості обміну даними.

РОЗДІЛ 2
АНАЛІЗ ВИМОГ ТА ОГЛЯД ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ СИСТЕМИ

2.1 Постановка вимог до системи

Перед проєктуванням архітектури та вибором технологій необхідно визначити вимоги до розроблюваної системи. Вимоги поділяються на функціональні, що описують конкретну поведінку системи, та нефункціональні, що визначають якісні характеристики її роботи. Сукупність цих вимог є основою для обґрунтування архітектурних та технологічних рішень, розглянутих у підрозділі 2.3.

2.1.1 Функціональні вимоги

Функціональні вимоги описують конкретну поведінку системи та функції, які вона має виконувати. На основі аналізу предметної області визначено наступні функціональні вимоги до розроблюваної системи.

Система має забезпечувати реєстрацію нового користувача з введенням електронної пошти та пароля без прив'язки до номера телефону, а також надсилати підтвердження реєстрації на вказану електронну адресу. Користувач має можливість входити в систему за своїми обліковими даними та підтримувати активну сесію між запитами. Також має бути можливість змінювати персональні дані: ім'я, аватар, нікнейм, логін та електронну пошту.

Система має підтримувати відправку та отримання приватних повідомлень між двома користувачами в режимі реального часу, а також групових повідомлень з підтримкою кількох учасників одночасно. Усі

повідомлення мають бути наскрізно зашифровані на стороні клієнта перед збереженням, а закритий ключ користувача має зберігатися виключно на його пристрої і не передаватися на сервер.

Система має забезпечувати збереження історії повідомлень у зашифрованому вигляді та надавати можливість їх перегляду після входу в систему. Також має підтримуватися завантаження та перегляд медіафайлів у межах чатів.

Внутрішні компоненти системи мають обмінюватися подіями асинхронно таким чином, щоб тимчасова недоступність одного компонента не блокувала роботу інших та не призводила до втрати даних.

2.1.2 Нефункціональні вимоги

Нефункціональні вимоги визначають якісні характеристики системи, що описують не що вона робить, а як вона це робить.

З точки зору безпеки система має гарантувати, що вміст повідомлень є недоступним для будь-якої третьої сторони, включаючи адміністраторів серверної інфраструктури. Закриті ключі шифрування мають зберігатися виключно на пристроях користувачів і не передаватися на сервер. Паролі користувачів мають зберігатися у захищеному хешованому вигляді.

З точки зору продуктивності система має забезпечувати доставку повідомлень у режимі реального часу з мінімальною затримкою. Серверна частина має бути здатна обробляти велику кількість одночасних з'єднань без надмірного споживання ресурсів. Повторні запити до однакових даних мають оброблятися без звернення до бази даних.

З точки зору надійності доставка внутрішніх подій між компонентами системи має гарантуватися навіть у разі тимчасової недоступності одного з них, з автоматичною повторною обробкою у разі збою. Збереження даних має

відповідати вимогам атомарності, узгодженості, ізольованості та довговічності транзакцій.

З точки зору масштабованості архітектура системи має допускати горизонтальне масштабування окремих компонентів незалежно один від одного без порушення їх взаємодії.

З точки зору супроводжуваності кожен компонент системи має бути покритий автоматизованими тестами, а структура коду має забезпечувати незалежність бізнес-логіки від інфраструктурних деталей.

З точки зору розгортання система має підтримувати контейнеризацію, що дозволяє відтворити середовище виконання незалежно від операційної системи хостової машини.

2.2 Огляд алгоритмів шифрування

Сучасні системи захищеного обміну повідомленнями будуються на основі комбінації декількох криптографічних примітивів, кожен з яких вирішує окрему задачу в загальній архітектурі безпеки. Жоден окремо взятий алгоритм не здатний забезпечити повний спектр вимог, що висуваються до захищених комунікацій: конфіденційність, цілісність, автентифікацію та захист від компрометації в майбутньому. Саме тому на практиці застосовується гібридний підхід, у якому різні алгоритми доповнюють один одного.

У рамках даної роботи в основу об'єкту розробки покладено три ключових аспекти: симетричний потоковий шифр ChaCha20-Poly1305, асиметричний алгоритм RSA та протокол наскрізного шифрування E2EE. Вибір саме цього набору обумовлений їхньою взаємодоповнюваністю: RSA забезпечує безпечний обмін ключами між сторонами, не знайомими одна з одною; ChaCha20-Poly1305 відповідає за швидке та надійне шифрування

безпосередньо повідомлень; E2EE як концепція об'єднує ці складові в єдину архітектуру, де навіть оператор сервісу не має доступу до вмісту переписки.

Нижче наведено детальний огляд кожного з перелічених алгоритмів та технологій з точки зору їхніх принципів роботи, математичного підґрунтя, а також місця в загальній архітектурі розроблюваної системи.

2.2.1 ChaCha20-Poly1305

ChaCha20-Poly1305 є алгоритмом автентифікованого шифрування з асоційованими даними (AEAD), що поєднує потоковий шифр ChaCha20 з кодом автентифікації повідомлень Poly1305 [27].

ChaCha20 є потоковим шифром, що оперує 256-бітним ключем та 96-бітним одноразовим значенням (nonce) [28]. Внутрішня функція шифру виконує серію операцій додавання, XOR та побітового зсуву (ARX: add-rotate-XOR) над 512-бітним станом протягом 20 раундів, після чого генерує псевдовипадковий потік бітів, що накладається на відкритий текст операцією XOR [28]. Ключовою перевагою ARX-архітектури є стійкість до атак на основі часових сторонніх каналів, оскільки всі операції виконуються за константний час незалежно від оброблюваних даних [28].

Poly1305 відповідає за цілісність та автентичність даних. При спільному використанні з ChaCha20 одноразовий ключ для Poly1305 генерується з перших 32 байт вихідного потоку шифру, що унеможливлює його повторне використання [27]. Poly1305 обчислює 128-бітний тег автентифікації як для зашифрованих даних, так і для асоційованих даних, що не підлягають шифруванню, але мають бути захищені від модифікації [27].

Алгоритм широко прийнятий у стандартах TLS та SSH як альтернатива існуючим наборам шифрів завдяки поєднанню високих показників безпеки та продуктивності на широкому спектрі програмних платформ. На відміну від AES-GCM, ChaCha20-Poly1305 забезпечує вищу швидкість роботи без

апаратного прискорення, що є критичним для мобільних пристроїв та середовищ без підтримки AES-NI [27]. Саме тому цей алгоритм обрано як основний механізм симетричного шифрування в розроблюваній системі [29].

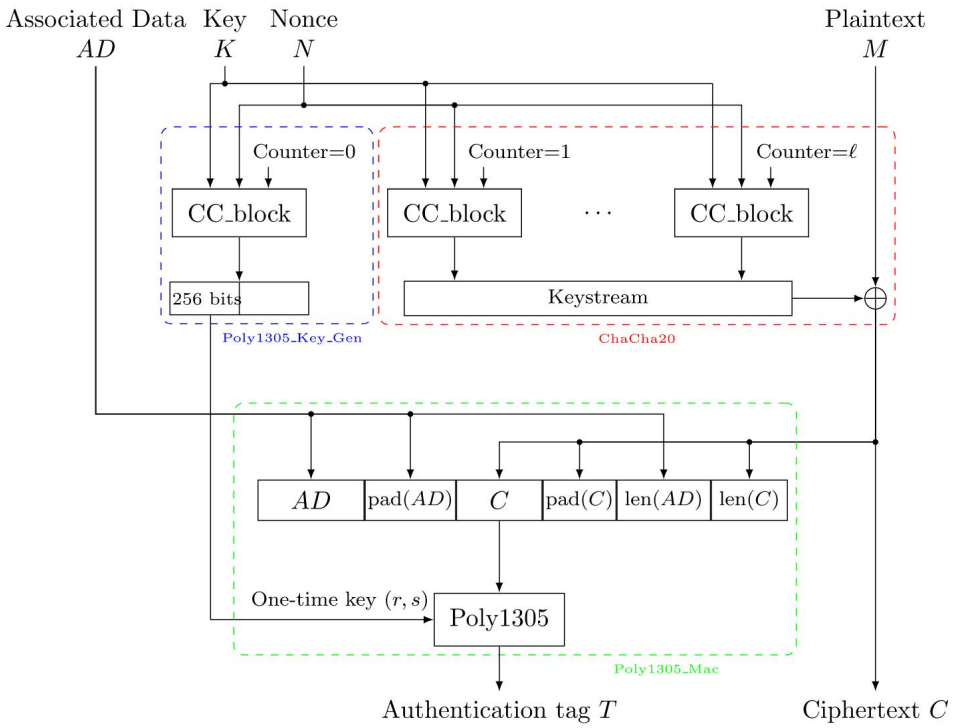

Рисунок 2.1 – Блок схема ChaCha20-Poly1305

Вибір саме ChaCha20-Poly1305 замість альтернативного алгоритму AES-GCM обумовлений нефункціональною вимогою щодо продуктивності. AES-GCM демонструє високу швидкість лише за наявності апаратного прискорення AES-NI, тоді як ChaCha20-Poly1305 забезпечує порівнянну або вищу продуктивність на програмному рівні без залежності від апаратних розширень процесора. Це є критичним для мобільних пристроїв та середовищ, де апаратне прискорення недоступне. Крім того, ARX-архітектура алгоритму природно стійка до атак на основі часових сторонніх каналів, що відповідає нефункціональній вимозі безпеки системи.

2.2.2 RSA

RSA (Rivest-Shamir-Adleman) є алгоритмом асиметричної або публічної криптографії, що оперує двома різними ключами: відкритим та закритим. Відкритий ключ використовується для шифрування і є загальновідомим, тоді як закритий ключ використовується для дешифрування і зберігається у таємниці отримувачем [31].

Процес роботи RSA складається з трьох основних етапів: генерація ключів, шифрування та дешифрування [31]. На етапі генерації ключів обираються два великих простих числа p та q, обчислюється їх добуток n = p * q, а також функція Ейлера φ(n) = (p-1)(q-1). Далі обирається відкрита експонента e така, що 1 < e < φ(n) та gcd(e, φ(n)) = 1. Закритий ключ d обчислюється як мультиплікативний обернений елемент e за модулем φ(n), тобто (d * e) ≡ 1 mod φ(n). Таким чином, відкритий ключ складає пара (n, e), а закритий (n, d) [31].

Шифрування повідомлення M виконується за формулою C = M^e mod n, де C є шифротекстом. Дешифрування здійснюється за формулою M = C^d mod n [31]. Безпека алгоритму ґрунтується на обчислювальній складності задачі факторизації великих цілих чисел: знаючи лише n та e, практично неможливо відновити p і q за розумний час [32].

RSA є асиметричним методом шифрування, що забезпечує захист завдяки тому, що лише власник закритого ключа може розшифрувати дані, навіть якщо відкритий ключ відомий іншим особам. Окрім шифрування, RSA широко застосовується для формування цифрових підписів та безпечного обміну симетричними ключами [33].

Вибір RSA для асиметричного обміну ключами між сервісами обумовлений насамперед зрілістю алгоритму та його широкою підтримкою в стандартній бібліотеці .NET — System.Security.Cryptography. Альтернативні

алгоритми, зокрема ECDH або X25519, забезпечують вищу продуктивність при менших розмірах ключів, однак RSA є більш дослідженим з точки зору безпеки та має тривалу історію практичного застосування у промислових системах. Практичним обмеженням RSA є висока обчислювальна складність операцій з великими числами, що робить алгоритм відносно повільним для шифрування великих обсягів даних [32]. Тому у розроблюваній системі RSA використовується виключно для захищеної передачі симетричних ключів між сервісами, а безпосереднє шифрування повідомлень виконується швидшим симетричним алгоритмом ChaCha20-Poly1305. Оскільки операція обміну ключами виконується одноразово при встановленні захищеного каналу, а не при кожному повідомленні, відносна повільність RSA не є критичним обмеженням для розроблюваної системи.

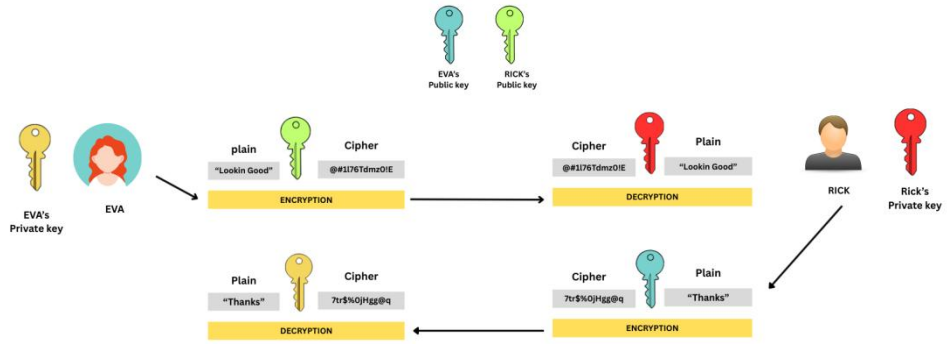

Рисунок 2.2 – Блок схема RSA

2.2.3 E2EE

E2EE (End-to-End Encryption, наскрізне шифрування) є процесом захищеної передачі даних, при якому інформація шифрується на пристрої відправника та залишається зашифрованою аж до пристрою отримувача[35]. Жодна третя сторона, включаючи уряди та сервери, через які проходять дані, не може отримати доступ до їхнього вмісту[37].

E2EE використовує поєднання асиметричної та симетричної криптографії для забезпечення безпеки, цілісності та конфіденційності комунікацій[36]. Процес встановлення захищеного з'єднання відбувається таким чином: кожен учасник генерує пару криптографічних ключів — відкритий та закритий. Відкритий ключ є загальнодоступним, тоді як закритий ключ зберігається виключно на пристрої користувача. При надсиланні даних пристрій відправника шифрує їх відкритим ключем отримувача, перетворюючи на нечитабельний шифротекст, і лише відповідний закритий ключ здатний його розшифрувати[38].

Ключовою відмінністю E2EE від традиційного шифрування під час передачі є те, що при стандартному підході дані розшифровуються на проміжних серверах, а потім шифруються знову. При E2EE постачальник послуг не тримає закритих ключів, а отже не може читати, сканувати або передавати вміст повідомлень навіть за умови юридичного запиту[38].

Системи E2EE застосовують як симетричне, так і асиметричне шифрування: пара відкритого та закритого ключів використовується для безпечного обміну сесійним ключем, тимчасовим симетричним ключем, яким безпосередньо шифруються дані. Такий гібридний підхід дозволяє поєднати переваги обох методів: надійність асиметричного шифрування при обміні ключами та швидкодію симетричного шифрування при роботі з даними [36].

E2EE має особливо важливе значення в областях, де конфіденційність є критичною: захищені комунікації, що перебувають під загрозою перехоплення, передача військової інформації, а також сфери, що стосуються чутливих даних про здоров'я або персональних відомостей[37]. Саме тому E2EE покладено в основу архітектури розроблюваної системи захищеного обміну повідомленнями.

Реалізація E2EE безпосередньо випливає з нефункціональної вимоги безпеки: вміст повідомлень має бути недоступним для будь-якої третьої сторони, включаючи адміністраторів серверної інфраструктури. Жоден інший

підхід до шифрування не задовольняє цю вимогу в повній мірі: шифрування на транспортному рівні (TLS) захищає лише канал передачі, але не вміст повідомлень на сервері, тоді як E2EE гарантує, що сервер у жодному разі не має доступу до відкритого тексту повідомлень.

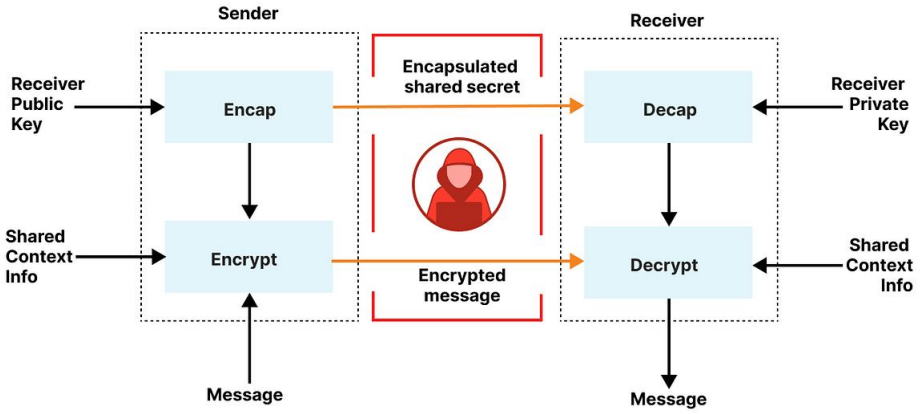

Рисунок 2.3 – Блок схема E2EE алгоритму

2.3 Технологічний стек та архітектурні рішення

2.3.1 .NET 8

.NET 8 є сучасною кросплатформеною платформою загального призначення зі статусом LTS-версії, що гарантує підтримку, оновлення та виправлення помилок щонайменше протягом трьох років з моменту виходу [45]. Саме стабільність та довгострокова підтримка стали одним із ключових аргументів на користь її вибору для розроблюваної системи.

З точки зору криптографії .NET 8 надає зрілу бібліотеку System.Security.Cryptography, що містить готові реалізації алгоритмів, використаних у системі: ChaCha20-Poly1305 для симетричного шифрування та RSA для асиметричного обміну ключами [45]. Це дозволяє уникнути залежності від сторонніх криптографічних бібліотек та спиратися на перевірені реалізації, що пройшли аудит безпеки.

Для веб-месенджера критично важливою є здатність сервера обробляти велику кількість одночасних підключень. .NET 8 надає розвинений асинхронний стек на основі async/await, що дозволяє ефективно обробляти тисячі одночасних WebSocket-з'єднань без надмірного споживання ресурсів [45]. Платформа також забезпечує засоби для побудови захищених мережевих з'єднань через TLS, що є необхідною умовою для транспортного рівня захисту на додачу до E2EE [45].

Платформа ASP.NET Core, що входить до складу .NET 8, забезпечує інтеграцію з SignalR для комунікації в реальному часі та з Entity Framework Core для роботи з PostgreSQL [47].

Порівняно з альтернативами, зокрема Node.js або Go, платформа .NET 8 у поєднанні з C# надає суттєві переваги саме для даного проєкту. Строга статична типізація C# знижує ймовірність помилок при роботі з криптографічними примітивами, де некоректна передача типів може призвести до вразливостей безпеки [39]. Node.js, попри високу продуктивність при обробці вхідних з'єднань, не має зрілої вбудованої криптографічної бібліотеки рівня System.Security.Cryptography та поступається C# у типобезпеці. Go є сильним конкурентом у сфері мікросервісів, однак значно менша екосистема бібліотек та відсутність вбудованої підтримки SignalR і Entity Framework роблять його менш зручним у контексті обраного технологічного стеку. Таким чином, .NET 8 є оптимальним вибором для розроблюваної системи, де безпека, типобезпека та зрілість криптографічної екосистеми є пріоритетними вимогами.

Водночас платформа має і певні обмеження. Порівняно з нативно компільованими мовами, такими як Go або Rust, продуктивність .NET є дещо нижчою через проміжний рівень IL та роботу JIT-компілятора [45]. Автоматичне збирання сміття, хоча і спрощує розробку, може вносити короткочасні паузи у роботу застосунку в моменти своєї активності, що потенційно впливає на затримку доставки повідомлень під високим

навантаженням [46]. Проте в контексті розроблюваної системи ці обмеження не є критичними, оскільки пріоритетами є надійність, безпека та зручність супроводу коду.

2.3.2 SignalR

SignalR є бібліотекою з відкритим вихідним кодом, що призначена для реалізації функціоналу реального часу у вебзастосунках. [48] Основна ідея бібліотеки полягає у можливості серверного коду надсилати повідомлення підключеним клієнтам у момент виникнення події, без необхідності клієнту самостійно опитувати сервер щодо наявності нових даних. [49]

SignalR автоматично вибирає оптимальний транспортний протокол залежно від можливостей клієнта та сервера. У першу чергу використовується протокол WebSockets, який забезпечує постійне двостороннє з'єднання між клієнтом та сервером. У випадку, коли WebSockets недоступний, бібліотека автоматично переходить на альтернативні методи передачі даних, зберігаючи при цьому незмінним програмний код застосунку. [49]

Центральним поняттям у SignalR є концепція хабу. Хаб є спеціальним класом на стороні сервера, що виступає посередником між сервером та клієнтами і дозволяє викликати методи на стороні клієнта безпосередньо з серверного коду. [48] Хаби SignalR можуть бути написані на C# і додаватися до ASP.NETCore застосунку поряд зі сторінками та API. Проста модель програмування органічно інтегрується з іншими механізмами ASP.NETCore, зокрема впровадженням залежностей, автентифікацією, авторизацією та масштабуванням. [49]

SignalR надає клієнтські SDK для широкого спектру платформ. Завдяки підтримці JavaScript, .NET, Java і Swift розробники можуть підключатися до хабу та отримувати повідомлення в реальному часі на вебсторінках, мобільних пристроях, десктопних застосунках та у середовищах розробки ігор. [49]

Бібліотека призначена не лише для реалізації чатів, хоча цей сценарій часто наводиться як типовий приклад. SignalR застосовується у панелях моніторингу, картах із динамічним оновленням, онлайн-іграх та будь-яких системах, де потрібна висока частота оновлення даних з боку сервера. [49]

У контексті системи обміну повідомленнями з шифруванням SignalR забезпечує транспортний рівень для доставки зашифрованих повідомлень між користувачами у режимі реального часу, що робить його ключовим компонентом клієнт-серверної взаємодії у цьому проєкті.

Вибір SignalR замість прямого використання WebSocket API або бібліотеки Socket.IO обумовлений кількома факторами. По-перше, SignalR автоматично обирає оптимальний транспортний протокол залежно від можливостей клієнта та сервера, що забезпечує сумісність без додаткового коду. По-друге, на відміну від Socket.IO, що є JavaScript-орієнтованою бібліотекою, SignalR органічно інтегрується з екосистемою ASP.NET Core та підтримує типізовані хаби на C#, що відповідає обраному технологічному стеку. По-третє, SignalR надає вбудовані механізми автентифікації та авторизації через стандартні механізми ASP.NET Core, що спрощує реалізацію вимог безпеки.

2.3.3 Redis

Redis (Remote Dictionary Server) є відкритим програмним забезпеченням, що являє собою рушій зберігання даних у пам'яті, який може виступати одночасно як база даних, кеш та брокер повідомлень [51]. Завдяки тому, що всі дані зберігаються безпосередньо в оперативній пам'яті, Redis забезпечує надзвичайно низьку затримку при зверненні до даних: читання та запис виконуються за мікросекунди, а не мілісекунди [51]. Саме ця властивість робить Redis одним із найпоширеніших інструментів у сучасних високонавантажених системах.

Ключовою особливістю Redis є підтримка різноманітних структур даних. На відміну від простих сховищ типу "ключ значення", Redis надає рядки, хеші, списки, множини, впорядковані множини, потоки та JSON-документи [51]. Завдяки цьому Redis підходить для широкого спектра завдань: від зберігання сесій користувачів до побудови черг задач, систем рейтингування та пошуку за векторами [51].

Важливим механізмом Redis є модель публікації та підписки (Pub/Sub). Ця модель дозволяє компонентам розподіленої системи обмінюватися повідомленнями через іменовані канали: видавці надсилають повідомлення у канал, а передплатники отримують їх у реальному часі [51]. Pub/Sub є асинхронною і масштабованою схемою комунікації, що відділяє виробників повідомлень від їхніх споживачів, що особливо корисно при побудові застосунків із подієвою архітектурою та систем сповіщень у реальному часі [51].

Redis також підтримує збереження даних на диску, що дозволяє системі відновлюватися після перебоїв. Механізм збереження реалізується через регулярні знімки стану бази даних, а також через журнал операцій, що дозволяє відновити дані з точністю до останньої виконаної команди [51]. Це вигідно відрізняє Redis від простих кешів, які втрачають усі дані при перезапуску.

З точки зору масштабування Redis підтримує реплікацію та кластеризацію. Redis Cluster дозволяє розподіляти дані між кількома вузлами, долаючи обмеження пам'яті одного сервера [51]. Для сценаріїв із підвищеними вимогами до доступності Redis підтримує реплікацію з декількома репліками, а також механізм Sentinel для автоматичного відновлення після відмови основного вузла.

У контексті розроблюваної системи Redis використовується для кешування історії повідомлень приватних та групових чатів. При першому запиті до історії результат зберігається у Redis на визначений проміжок часу,

що дозволяє уникнути повторних звернень до PostgreSQL при завантаженні тієї самої сторінки повідомлень. Такий підхід безпосередньо реалізує нефункціональну вимогу щодо продуктивності: повторні запити до однакових даних обробляються без звернення до бази даних.

2.3.4 RabbitMQ

RabbitMQ є одним із найпоширеніших брокерів повідомлень з відкритим вихідним кодом, що реалізує протокол AMQP (Advanced Message Queuing Protocol). Система призначена для організації асинхронного обміну повідомленнями між компонентами розподілених застосунків, забезпечуючи надійну доставку, маршрутизацію та буферизацію повідомлень [53].

В основі архітектури RabbitMQ лежить модель, в якій виробники повідомлень публікують їх до обмінників, а не безпосередньо у черги. Обмінник розподіляє копії повідомлень до черг за правилами, що називаються прив'язками, після чого споживачі отримують повідомлення або через підписку, або шляхом явного запиту [53]. RabbitMQ підтримує чотири типи обмінників: прямий, широкомовний, тематичний та заголовковий, кожен з яких реалізує власну логіку маршрутизації залежно від потреб застосунку [53].

Оскільки мережі є ненадійними, протокол AMQP передбачає механізм підтверджень: після доставки повідомлення споживач надсилає брокеру підтвердження про успішну обробку. Якщо повідомлення не може бути доставлено, воно може бути повернуто видавцю або переміщено до черги недоставлених повідомлень [53]. Черги підтримують налаштування стійкості, ексклюзивності та автовидалення, що дозволяє гнучко адаптувати поведінку системи під конкретні вимоги [53].

Вибір RabbitMQ як брокера повідомлень обумовлений порівнянням з основною альтернативою — Apache Kafka. Kafka є оптимальним рішенням для обробки великих потоків даних та журналювання подій з довгостроковим

зберіганням, однак має значно складнішу модель розгортання та налаштування. У розроблюваній системі брокер використовується для передачі подій між трьома мікросервісами з відносно невеликою кількістю повідомлень та складною логікою маршрутизації, що є саме тим сценарієм для якого RabbitMQ є більш підходящим рішенням. Додатково RabbitMQ має широку підтримку клієнтських бібліотек для .NET, що органічно вписується в обраний технологічний стек.

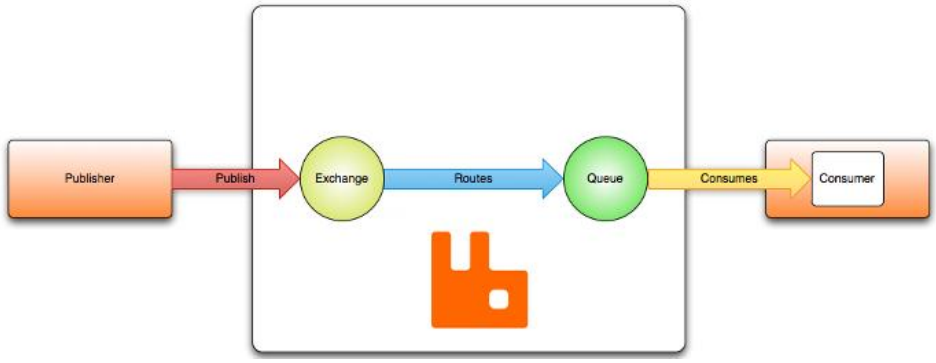

Рисунок 2.4 – Архітектура передачі повідомлень у RabbitMQ

2.3.5 PostgreSQL

PostgreSQL є об'єктно-реляційною системою управління базами даних з відкритим вихідним кодом, що бере початок з 1986 року як проєкт POSTGRES в Університеті Каліфорнії в Берклі та має майже 40 років активної розробки [53]. Система є ACID-сумісною, що гарантує атомарність, узгодженість, ізольованість та довговічність транзакцій, а відповідність стандарту SQL:2023 підтверджена підтримкою щонайменше 170 із 177 обов'язкових вимог [53].

Управління конкурентним доступом реалізовано через механізм MVCC (Multiversion Concurrency Control): кожна транзакція отримує власний знімок стану бази даних, завдяки чому операції читання не блокують запис і навпаки [54]. Надійність даних забезпечується механізмом Write-Ahead Logging (WAL), що фіксує всі зміни у журналі до їх застосування, а також підтримкою

реплікації та Point-In-Time Recovery для відновлення бази до конкретного моменту часу [54].

З точки зору безпеки PostgreSQL підтримує автентифікацію через SCRAM-SHA-256, сертифікати та OAuth 2.0, а також розмежування доступу на рівні окремих рядків і стовпців таблиць [53]. Система є високо розширюваною: підтримуються користувацькі типи даних, збережені процедури та функції на мовах PL/pgSQL, Python, Perl, Java та Rust без необхідності перекомпіляції ядра [53]. PostgreSQL масштабується як по обсягу даних, так і по кількості одночасних підключень, при цьому активні кластери у продакшн-середовищах обробляють дані обсягом у петабайти [53].
Наведені характеристики визначили вибір PostgreSQL як основного сховища даних у розроблюваній системі захищеного обміну повідомленнями.

2.3.6 Unit and Integration тести

Юніт-тести (unit tests) є методом тестування, при якому окремі компоненти програми, такі як функції, методи або класи, перевіряються в ізоляції від решти системи; зовнішні залежності при цьому замінюються підробленими об'єктами (mock/stub), що дозволяє виявляти помилки на ранніх етапах розробки [56].

Інтеграційні тести (integration tests) перевіряють коректність взаємодії між двома і більше модулями системи, використовуючи реальні або наближені до реальних зовнішні ресурси, зокрема бази даних та мережеві з'єднання [56].

Обидва типи тестів є взаємодоповнювальними: юніт-тести забезпечують швидке виявлення дефектів у конкретній логіці, тоді як інтеграційні тести підтверджують коректну роботу системи в цілому [55].

Для тестування логіки в серверній частині проекту було використано такі бібліотеки: Xunit, Microsoft.AspNetCore.Mvc.Testing, Moq, DotNet.Testcontainers.Builders, DotNet.Testcontainers.Containers.

xUnit це фреймворк для написання та запуску автоматизованих тестів на C# і .NET, що підтримує .NET 8 [57]. Він виконує роль основи тестової інфраструктури: надає механізми для оголошення тестових методів, організації тестових класів, параметризації тестів та формування звітів про результати їх виконання [57].

Moq є найпопулярнішою бібліотекою мокінгу для .NET, що дозволяє замінювати реальні залежності підробленими об'єктами під час юніт-тестування [58]. Бібліотека побудована на основі lambda-виразів та LINQ, завдяки чому синтаксис налаштування поведінки mock-об'єктів є лаконічним та типобезпечним.

Microsoft.AspNetCore.Mvc.Testing надає клас WebApplicationFactory, що запускає весь ASP.NET Core додаток в оперативній пам'яті без реального мережевого порту [59]. Це дозволяє надсилати HTTP-запити до ендпоінтів API в рамках інтеграційних тестів і перевіряти повний ланцюжок обробки запиту: маршрутизацію, middleware, контролери та сервіси [59].

Бібліотеки DotNet.Testcontainers.Builders та DotNet.Testcontainers.Containers є частиною Testcontainers для .NET — бібліотеки, що дозволяє програмно запускати Docker-контейнери безпосередньо під час виконання тестів [60]. У проєкті вони використовуються для піднімання реального екземпляру PostgreSQL у контейнері перед запуском інтеграційних тестів та автоматичного його видалення після їх завершення, що забезпечує тестування в середовищі, максимально наближеному до продакшн [60].

Для тестування фронтенду в свою чергу використовувалися Jasmine та Karma. Jasmine є фреймворком для тестування JavaScript з відкритим вихідним кодом, що підтримує методологію Behavior-Driven Development (BDD) і є стандартним інструментом тестування в Angular [61]. Він складається з трьох частин: бібліотеки з функціями для побудови тестів, рушія виконання тестів та рушія формування звітів. Синтаксис Jasmine базується на блоках describe для

групування тестів, it для опису окремих тест-кейсів та функціях expect для перевірки очікуваних результатів [61]. Karma у свою чергу є test runner — інструментом, що запускає браузер, виконує в ньому тести Jasmine та повертає результати до командного рядка [62]. У проєкті Angular CLI автоматично запускає Karma через команду ng test, що усуває необхідність ручного налаштування тестового середовища [62].

2.3.7 Angular

Angular є відкритим фреймворком для побудови вебзастосунків на стороні клієнта, що підтримується командою Google [63]. Фреймворк призначений для створення швидких, надійних та масштабованих застосунків і надає розробникам широкий набір інструментів, API та бібліотек для спрощення процесу розробки [63].

Основою Angular є компонентна архітектура. Кожен компонент складається з HTML-шаблону, що визначає інтерфейс, класу TypeScript, який містить логіку поведінки, та CSS-селектора для стилізації [64]. Компоненти об'єднуються у модулі, які групують пов'язані між собою частини застосунку, що спрощує організацію та підтримку коду [64]. Така модульність дозволяє членам команди вносити зміни незалежно, без необхідності повного оновлення проєкту [64].

Важливою концепцією Angular є двостороннє зв'язування даних. Зміни у моделі автоматично відображаються у поданні і навпаки, що суттєво скорочує кількість повторюваного коду та прискорює розробку інтерфейсу [64]. Шаблони Angular використовують розширений синтаксис HTML з підтримкою директив, які розширюють можливості стандартних HTML-елементів. Структурні директиви додають або видаляють елементи з DOM, а атрибутні директиви змінюють поведінку чи зовнішній вигляд існуючих елементів [64].

Сервіси у Angular надають повторно використовувану логіку та дані між різними компонентами застосунку. Система впровадження залежностей, вбудована у фреймворк, автоматично надає компонентам необхідні сервіси, що зменшує зв'язаність між частинами застосунку та спрощує тестування [64]. Вбудована система маршрутизації Angular дозволяє реалізовувати навігацію у односторінкових застосунках з підтримкою захисників маршрутів, лінивого завантаження та розпізнавання даних [63].

Angular підтримує первинну мову програмування TypeScript, яка надає статичну типізацію та перевірку помилок на етапі компіляції [64]. Це покращує читабельність коду, зменшує кількість помилок у виконанні та полегшує налагодження у великих командах. Angular CLI спрощує типові задачі, такі як генерація компонентів, налаштування маршрутизації та розгортання застосунку, і дозволяє виконувати їх за допомогою однієї команди у терміналі [64].

Компіляція AOT (Ahead-of-Time) є ще однією ключовою особливістю: шаблони застосунку попередньо компілюються під час збірки, а не під час виконання. Це зменшує розмір бандлу, прискорює завантаження сторінки та мінімізує кількість помилок виконання [64]. Для реактивного програмування Angular використовує бібліотеку RxJS, яка забезпечує зручну обробку потоків даних та асинхронних подій, що є особливо корисним при роботі із запитами до API та оновленнями у реальному часі [64].

Вибір Angular замість альтернативних фреймворків React або Vue обумовлений кількома факторами. По-перше, Angular є повноцінним фреймворком що включає вбудовані рішення для маршрутизації, управління формами, HTTP-клієнта та впровадження залежностей, тоді як React та Vue вимагають залучення додаткових бібліотек для вирішення тих самих задач. По-друге, Angular використовує TypeScript як основну мову за замовчуванням, що забезпечує типобезпеку на рівні фронтенду та узгоджується з підходом до типізації на серверній частині. По-третє, вбудована підтримка RxJS спрощує

роботу з асинхронними потоками даних від SignalR, що є критичним для відображення повідомлень у режимі реального часу.

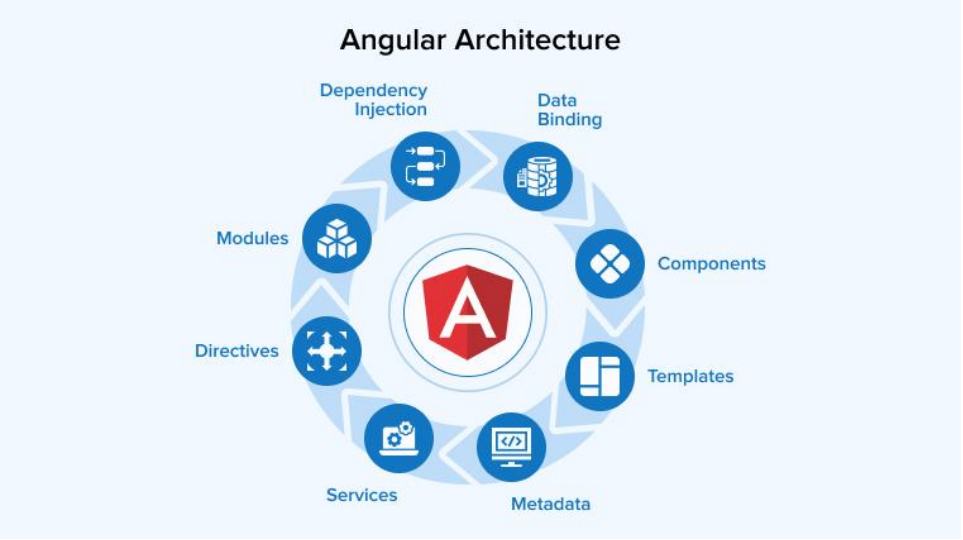

Рисунок 2.5 – Архітектура Angular

Angular є потужним та комплексним фреймворком для розробки клієнтської частини вебзастосунків, що поєднує компонентну архітектуру, двостороннє зв'язування даних, вбудовану систему маршрутизації та впровадження залежностей в єдине рішення. Використання TypeScript як основної мови програмування забезпечує типобезпеку та зменшує кількість помилок на етапі розробки, а компіляція AOT підвищує продуктивність застосунку під час виконання. Інтеграція з бібліотекою RxJS робить Angular особливо доцільним вибором для застосунків із реактивною обробкою даних у реальному часі, що в сукупності обумовлює його використання у даному проєкті.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі сформульовано вимоги до розроблюваної системи, розглянуто криптографічні алгоритми, що лягли в її основу, та обґрунтовано вибір технологічного стеку.

На початку розділу визначено функціональні та нефункціональні вимоги до системи. Функціональні вимоги охоплюють реєстрацію та автентифікацію користувачів, відправку приватних і групових повідомлень у режимі реального часу, наскрізне шифрування вмісту на стороні клієнта, збереження зашифрованої історії повідомлень та підтримку медіафайлів. Нефункціональні вимоги визначають критерії безпеки, продуктивності, надійності, масштабованості системи. Сукупність цих вимог стала основою для подальшого вибору алгоритмів і технологій.

У підрозділі огляду криптографічних алгоритмів розглянуто три ключові механізми захисту. ChaCha20-Poly1305 обрано як основний алгоритм симетричного шифрування повідомлень завдяки високій продуктивності на програмному рівні без залежності від апаратного прискорення та природній стійкості до атак на основі часових сторонніх каналів. RSA використовується для захищеної передачі симетричних ключів між сервісами, що унеможливлює їх перехоплення під час транзиту. E2EE як архітектурна концепція об'єднує обидва алгоритми, гарантуючи що сервер у жодному разі не має доступу до відкритого тексту повідомлень.

У підрозділі технологічного стеку обґрунтовано вибір кожного компонента системи. Платформа .NET 8 обрана для серверної частини завдяки зрілій криптографічній бібліотеці, строгій типізації та розвиненому асинхронному стеку. SignalR забезпечує доставку повідомлень у режимі реального часу через WebSockets з вбудованою інтеграцією в екосистему ASP.NET Core. Redis використовується для кешування історії повідомлень, що

зменшує навантаження на базу даних при повторних запитах. RabbitMQ забезпечує надійний асинхронний обмін подіями між мікросервісами з гарантованою доставкою та механізмом повторної обробки у разі збою. PostgreSQL обрано як основне сховище даних завдяки ACID-сумісності та розширеним механізмам безпеки. Для клієнтської частини обрано Angular, який забезпечує типобезпеку через TypeScript та зручну роботу з асинхронними потоками даних від SignalR через RxJS.

Окремо розглянуто підхід до тестування: для серверної частини використовуються xUnit, Moq, Microsoft.AspNetCore.Mvc.Testing та Testcontainers, що забезпечують як юніт-, так і інтеграційне тестування в середовищі, максимально наближеному до продакшн. Для фронтенду застосовуються Jasmine та Karma як стандартні інструменти тестування Angular-застосунків.

РОЗДІЛ 3
ПРОЕКТУВАННЯ СИСТЕМИ

Проведений аналіз криптографічних алгоритмів та сформульовані вимоги до системи створюють необхідне підґрунтя для переходу до практичної фази. Реалізація системи захищеного обміну повідомленнями із застосуванням описаних криптографічних механізмів потребує детального опису архітектури системи, визначення її основних структурних компонентів та схем їх взаємодії, що і є предметом розгляду цього розділу.

3.1 Архітектура серверної частини

Серверна частина системи побудована на основі мікросервісної архітектури, яка передбачає декомпозицію застосунку на сукупність незалежних, слабко зв'язаних сервісів, кожен з яких відповідає за чітко визначену функціональну область та може розроблятися, розгортатися і масштабуватися автономно[66]. На відміну від монолітного підходу, де всі компоненти системи об'єднані в єдиний розгортуваний артефакт, мікросервісна архітектура дозволяє ізолювати критично важливі підсистеми, зокрема модуль шифрування та модуль автентифікації, від решти компонентів системи, що підвищує як безпеку, так і відмовостійкість продукту в цілому[67]. Взаємодія між сервісами здійснюється через чітко визначені інтерфейси, що забезпечує незалежність внутрішньої реалізації кожного з них[66].

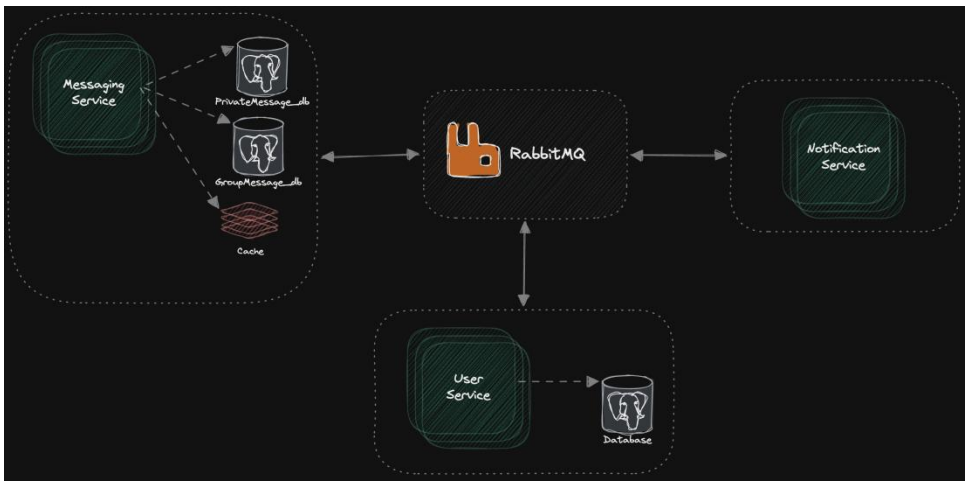

Рисунок 3.1 – Мікросервісна архітектура серверної частини системи

Для забезпечення якості внутрішньої структури кожного сервісу в проєкті застосовано підхід Clean Architecture, який детально розглянуто в наступному підрозділі. Загальна структурна схема системи наведена в Додатку 1.

3.1.1 Clean Architecture в сервісах

Кожен мікросервіс системи побудований відповідно до принципів Clean Architecture, запропонованих Робертом Мартіном. Основна ідея полягає у структуруванні коду у вигляді концентричних шарів, де кожен шар має єдину зону відповідальності, а залежності між шарами спрямовані виключно всередину: зовнішні шари залежать від внутрішніх, але не навпаки [68].

Архітектура складається з чотирьох шарів. Найглибший шар — шар сутностей (Entities) — містить базові класи та інтерфейси доменної моделі і не залежить від жодного іншого шару [68]. Шар варіантів використання (Use Cases) містить всю бізнес-логіку застосунку та виступає посередником між контролерами та інфраструктурним шаром. Шар контролерів відповідає за приймання вхідних HTTP-запитів, їх валідацію та передачу до відповідного варіанту використання. Зовнішній інфраструктурний шар відповідає за

взаємодію з базою даних, зовнішніми сервісами та іншими технічними деталями реалізації [68].

Для розв'язання конфліктів залежностей між шарами застосовуються два ключові принципи: інверсія залежностей (Dependency Inversion) через абстрактні інтерфейси та ін'єкція залежностей (Dependency Injection), що дозволяє передавати реалізацію ззовні, не прив'язуючи шар бізнес-логіки до конкретної технології [68]. Завдяки цьому, наприклад, заміна PostgreSQL на іншу базу даних потребувала б змін лише в інфраструктурному шарі, не зачіпаючи бізнес-логіку.

Застосування Clean Architecture в кожному сервісі забезпечує високу тестованість коду, оскільки бізнес-логіка повністю ізольована від зовнішніх залежностей і може перевірятися за допомогою mock-об'єктів, а також спрощує підтримку та масштабування системи в майбутньому.

Відповідно до принципу розподілу відповідальностей у мікросервісній архітектурі, в проєкті було виділено три окремих сервіси: сервіс відправки e-mail повідомлень (MessagingSystem.Services.Notification), сервіс для роботи з чатами та SignalR хабом (MessagingSystem.Services.Messaging) та сервіс реєстрації, авторизації і управління даними користувача (MessagingSystem.Services.User).

MessagingSystem.Services.Notification відповідає за відправку e-mail повідомлень користувачам системи і для цього інтегрується зі стороннім API MailJet. Як видно з рисунку 3.2, сервіс структурований відповідно до принципів Clean Architecture і містить п'ять основних директорій.

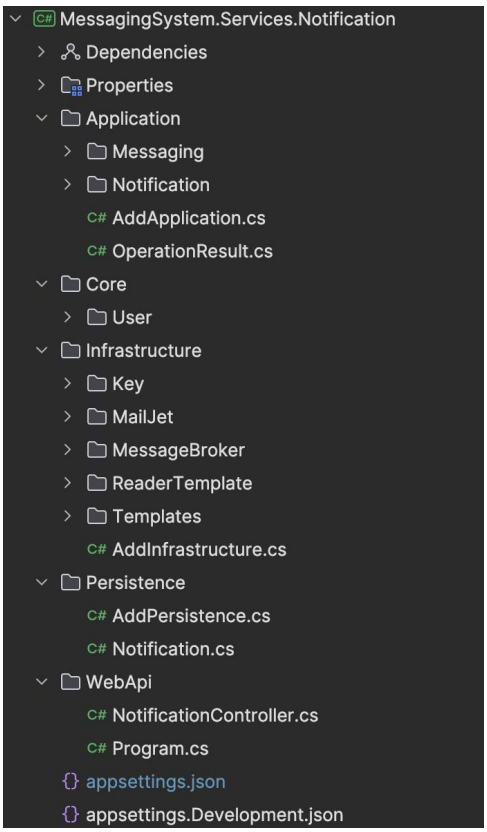

Рисунок 3.2 – Архітектура MessagingSystem.Services.Notification

Директорія Core містить доменні сутності сервісу, зокрема модель користувача, і не залежить від жодного іншого шару. Директорія Application містить бізнес-логіку у вигляді підкаталогів Messaging та Notification, а також файли AddApplication.cs для реєстрації залежностей та OperationResult.cs для уніфікованого представлення результатів операцій. Директорія Infrastructure реалізує взаємодію із зовнішніми системами: Key відповідає за відправлення ключа іншим сервісам за допомогою RabbitMQ, MailJet містить інтеграцію з поштовим API, MessageBroker забезпечує взаємодію з RabbitMQ, ReaderTemplate та Templates відповідають за зберігання та читання шаблонів листів. Директорія Persistence містить файл AddPersistence.cs для реєстрації залежностей та клас Notification.cs, який відповідає за підстановку даних з моделі в зчитаний шаблон. Директорія WebApi є точкою входу сервісу і містить NotificationController.cs для обробки вхідних запитів та Program.cs для конфігурації додатку (рис. Д2.1).

MessagingSystem.Services.Messaging є центральним сервісом системи, що відповідає за роботу з чатами, обробку повідомлень та управління SignalR хабом для забезпечення комунікації в реальному часі. Як видно з рисунку 3.3, сервіс має розширену структуру порівняно з Notification і містить шість основних директорій, що відображає більшу функціональну складність цього компонента.

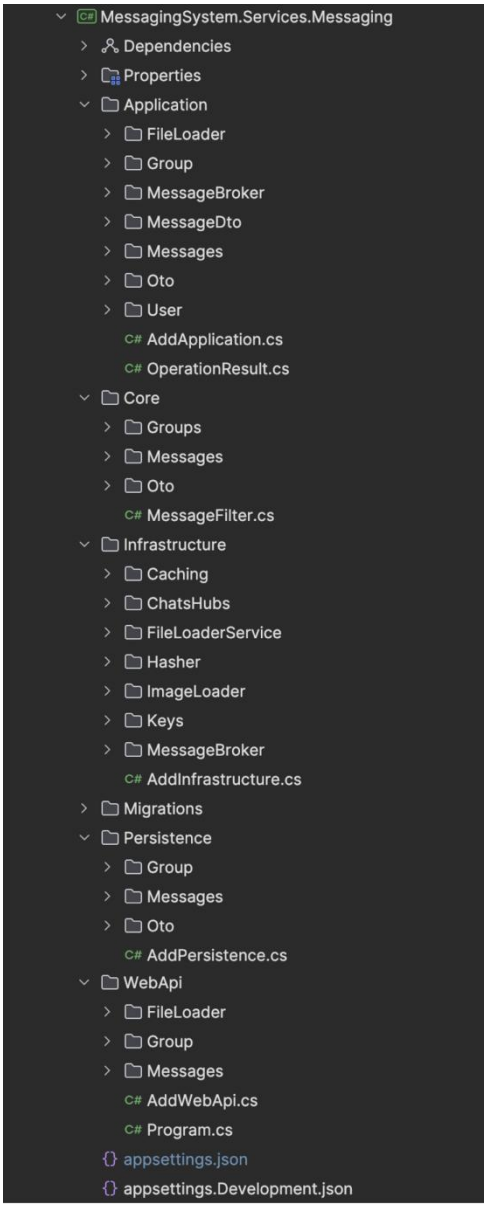

Рисунок 3.3 – Архітектура MessagingSystem.Services.Messaging

Директорія Core містить доменні сутності, розподілені за трьома підкаталогами: Groups - сутності групових чатів, учасників групи та групові повідомлення, Messages - абстракна сутність повідомлень, Oto - сутності

персональних чатів, нікнейми та аватари користувачів та персональні повідомлення, а також файл MessageFilter.cs для фільтрації повідомлень. Директорія Application містить бізнес-логіку сервісу і включає підкаталоги FileLoader, Group, MessageBroker, MessageDto, Messages, Oto та User, а також файли AddApplication.cs для реєстрації залежностей та OperationResult.cs для уніфікованого представлення результатів операцій. Директорія Infrastructure реалізує взаємодію із зовнішніми системами та містить: Caching для кешування даних, ChatHubs з реалізацією SignalR хабу, FileLoaderService для роботи з файлами, Hasher для хешування чутливих даних, ImageLoader для завантаження зображень, Keys для обміну ключами між сервісами та MessageBroker для взаємодії з RabbitMQ. Директорія Migrations містить міграції бази даних. Директорія Persistence відповідає за збереження даних і структурована за сутностями Group, Messages та Oto з файлом AddPersistence.cs для реєстрації залежностей. Директорія WebApi є точкою входу сервісу і містить контролери FileLoader, Group та Messages, а також файли AddWebApi.cs та Program.cs для конфігурації додатку (рис. Д2.2).

MessagingSystem.Services.User є сервісом, що відповідає за реєстрацію, авторизацію та управління даними користувача. Як видно з рисунку 3.4, сервіс також дотримується принципів Clean Architecture і містить шість основних директорій.

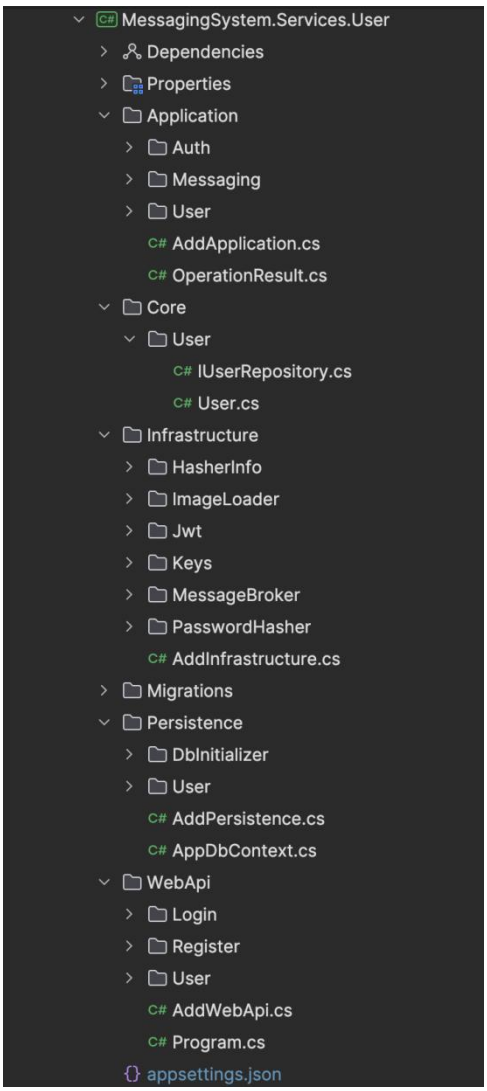

Рисунок 3.4 – Архітектура MessagingSystem.Services.User

Директорія Core містить доменну модель користувача: інтерфейс IUserRepository.cs, що визначає контракт для роботи з даними користувача, та сутність User.cs. Директорія Application містить бізнес-логіку, розподілену за трьома підкаталогами: Auth - логіка реєстрації та авторизації, Messaging - взаємодія з брокером повідомлень, User - логіка управління даними користувача, а також файли AddApplication.cs та OperationResult.cs.

Infrastructure реалізує технічні деталі і включає: HasherInfo та PasswordHasher для хешування паролів, Jwt для генерації та валідації JWT-токенів, Keys для обміну ключами між сервісами, ImageLoader для роботи із зображеннями профілю та MessageBroker для взаємодії з RabbitMQ.

Директорія Migrations містить міграції бази даних. Persistence відповідає за збереження даних і включає DbInitializer для початкового заповнення бази даних, підкаталог User з реалізацією репозиторію та AddPersistence.cs для реєстрації залежностей. Директорія WebApi є точкою входу сервісу і містить контролери Login, Register та User для обробки відповідних HTTP-запитів, а також AddWebApi.cs та Program.cs для конфігурації додатку (рис. Д2.3).

3.2 Архітектура клієнтської частини

Клієнтська частина системи реалізована на базі фреймворку Angular з використанням мови TypeScript. На відміну від серверної частини, де застосовано мікросервісну декомпозицію, клієнтська частина є єдиним застосунком, внутрішня структура якого організована відповідно до методології Feature-Sliced Design (FSD). Такий підхід забезпечує чіткий поділ коду за функціональними областями та рівнями абстракції, що спрощує масштабування та підтримку застосунку детальний код наведено в Додатку 4.

3.2.1 FSD

FSD є архітектурною методологією для фронтенд-застосунків, що організовує код за трьома вимірами: шари (layers), слайси (slices) та сегменти (segments). Основна ідея полягає у тому, що модулі верхніх шарів можуть залежати від модулів нижніх шарів, але не навпаки - це унеможливлює циклічні залежності та робить кодову базу передбачуваною і стабільною [69]. Методологія визначає фіксований набір шарів у порядку від верхнього до нижнього, що наочно відображено на рисунку 3.5.

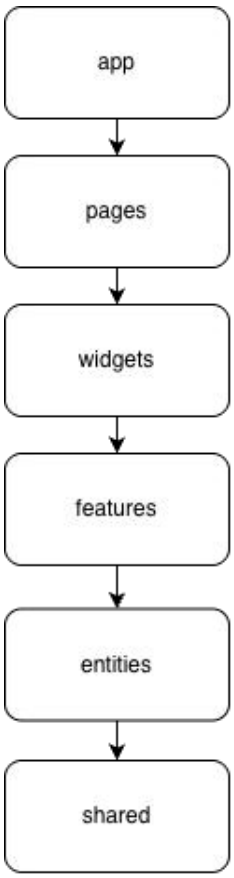

Рисунок 3.5 – Шари методології Feature-Sliced Design

Застосування FSD у розроблюваній системі забезпечує чіткі межі між функціональними модулями - зокрема між модулем шифрування, модулем чатів та модулем автентифікації, - що відповідає загальному принципу ізоляції критично важливих підсистем, прийнятому в архітектурі системи в цілому.

Відповідно до принципів FSD структура клієнтської частини системи організована наступним чином. Шар app є точкою входу застосунку і містить guards з auth.guard.ts для захисту маршрутів від неавторизованого доступу, layouts з реалізацією головного макету сторінок, файли конфігурації app.config.ts та маршрутизації app.routes.ts, інтерсептори unauthorized-redirect для автоматичного перенаправлення при закінченні сесії. Директорія assets містить підкаталог i18n з файлами локалізації для чотирьох мов: англійської, української, німецької та іспанської, а також custom-language-loader.ts для їх динамічного завантаження.

Шар entities містить доменні моделі та базові API-сервіси: session для моделей сесії та сервісу автентифікації, user для моделей користувача, search-user для моделі пошуку, oto-message та group-message для моделей приватних та групових повідомлень відповідно. Шар features містить функціональні модулі: auth для логіки реєстрації та автентифікації, file-sender для відправки медіафайлів, keys-generator для генерації та управління криптографічними ключами E2EE з використанням бібліотеки libsodium та search-user для пошуку користувачів системи.

Шар pages містить компоненти окремих сторінок: login та register для сторінок автентифікації, email-confirmation для підтвердження електронної пошти, oto-chat та group-chat для сторінок приватного та групового чату відповідно, settings для налаштувань облікового запису та user-profile для перегляду профілю користувача. Шар widgets містить великі самодостатні блоки інтерфейсу: auth-layout для макету сторінок автентифікації, chat-layout для загального макету чату, chat-messages для відображення списку повідомлень та navigation-side-bar для бічної панелі навігації.

Шар shared містить перевикористовуваний інфраструктурний код: api-result та api-urls для централізованого управління HTTP-запитами, ui-elements з набором перевикористовуваних компонентів інтерфейсу, realtime для інкапсуляції логіки SignalR-з'єднання, send-message-area для компонента введення та відправки повідомлення, setting-key для управління налаштуваннями ключів шифрування, storage для роботи з локальним сховищем браузера де зберігаються криптографічні ключі користувача, validators для кастомних валідаторів форм та styles для глобальних стилів на основі Tailwind CSS. Директорія types містить TypeScript-декларацію для криптографічної бібліотеки libsodium-wrappers-sumo, що використовується у модулі E2EE шифрування.

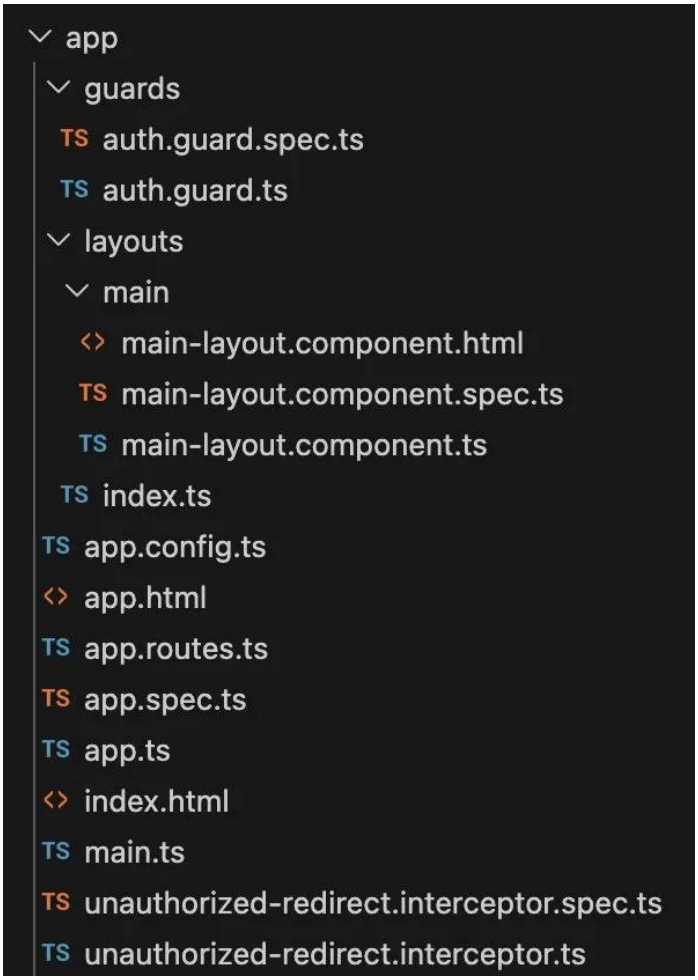

Рисунок 3.6 – Шар app

Шар app є точкою входу застосунку і відповідає за його глобальну конфігурацію та ініціалізацію. Директорія guards містить auth.guard.ts — захисник маршрутів, що перевіряє наявність активної сесії користувача перед наданням доступу до захищених сторінок. Директорія layouts містить підкаталог main з компонентом main-layout, що визначає загальний макет застосунку після авторизації. Файл app.config.ts містить глобальні провайдери та конфігурацію залежностей застосунку, app.routes.ts визначає всі маршрути та правила навігації між сторінками, а main.ts є точкою запуску Angular-застосунку. Файл unauthorized-redirect.interceptor.ts є HTTP-інтерсептором, що автоматично перенаправляє користувача на сторінку входу у разі отримання відповіді з кодом 401 від сервера.

Рисунок 3.7 – Шар pages

Шар pages містить компоненти окремих сторінок застосунку, кожна з яких організована у власний слайс з файлом index.ts для публічного експорту. Слайс email-confirmation містить service з логікою підтвердження електронної пошти та ui з відповідним компонентом. Слайси group-chat та oto-chat є найбільш розгалуженими і містять три підкаталоги: api для HTTP-запитів до серверного API, model для моделей даних чату та ui для компонентів відображення. Слайси login та register містять model з контрактами форм та ui

з компонентами відповідних сторінок автентифікації. Слайс settings містить лише ui з компонентом налаштувань облікового запису. Слайс user-profile містить api для завантаження даних профілю, model для їх представлення та ui з компонентом сторінки профілю користувача.

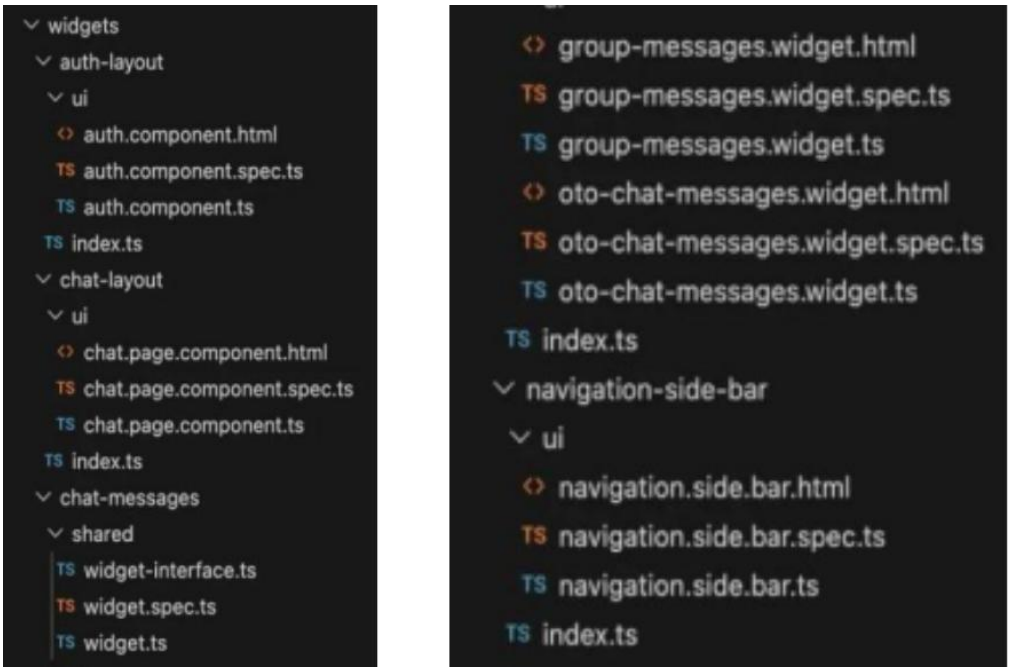

Рисунок 3.8 – Шар widgets

Шар widgets містить великі самодостатні блоки інтерфейсу що компонують елементи нижніх шарів. Слайс auth-layout містить ui з компонентом auth.component, що визначає загальний макет сторінок автентифікації - обгортку з логотипом та формою входу або реєстрації. Слайс chat-layout містить ui з компонентом chat.page.component, що визначає загальний макет сторінки чату з бічною панеллю та областю повідомлень. Слайс chat-messages є найбільш складним: підкаталог shared містить widget-interface.ts з абстрактним інтерфейсом віджету повідомлень та widget.ts з базовою реалізацією, а підкаталог ui містить два окремих віджети - group-messages.widget для відображення повідомлень групового чату та oto-chat-messages.widget для приватного чату, кожен з яких має власний HTML-шаблон та TypeScript-клас. Слайс navigation-side-bar містить ui з компонентом navigation.side.bar, що реалізує бічну панель навігації між чатами.

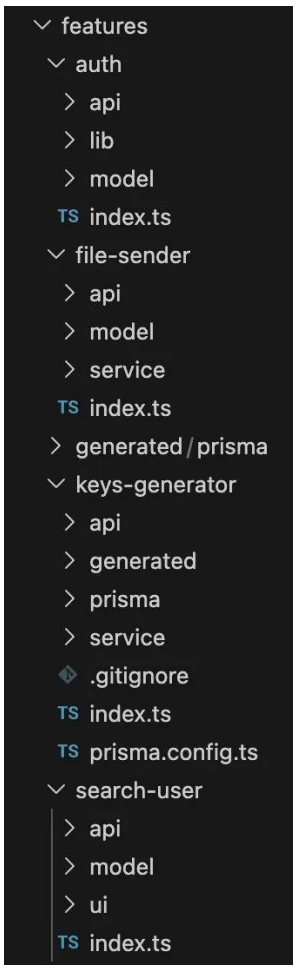

Рисунок 3.9 – Шар features

Шар features містить функціональні модулі що реалізують конкретні продуктові можливості системи. Слайс auth містить три підкаталоги: api для HTTP-запитів до ендпоінтів автентифікації, lib з допоміжною логікою та model з контрактами даних. Слайс file-sender відповідає за відправку медіафайлів у чатах і містить api для взаємодії з серверним API, model для моделей файлів та service з логікою підготовки та відправки файлів. Слайс keys-generator є ключовим модулем системи з точки зору безпеки - він відповідає за генерацію та управління криптографічними ключами E2EE і містить api для взаємодії з сервером, generated та prisma з автоматично згенерованим кодом для роботи з локальним сховищем ключів, service з логікою генерації ключів на основі бібліотеки libsodium та файл prisma.config.ts з конфігурацією. Слайс search-

user містить api для пошукових запитів, model для моделі результатів пошуку та ui з компонентом відображення списку знайдених користувачів.

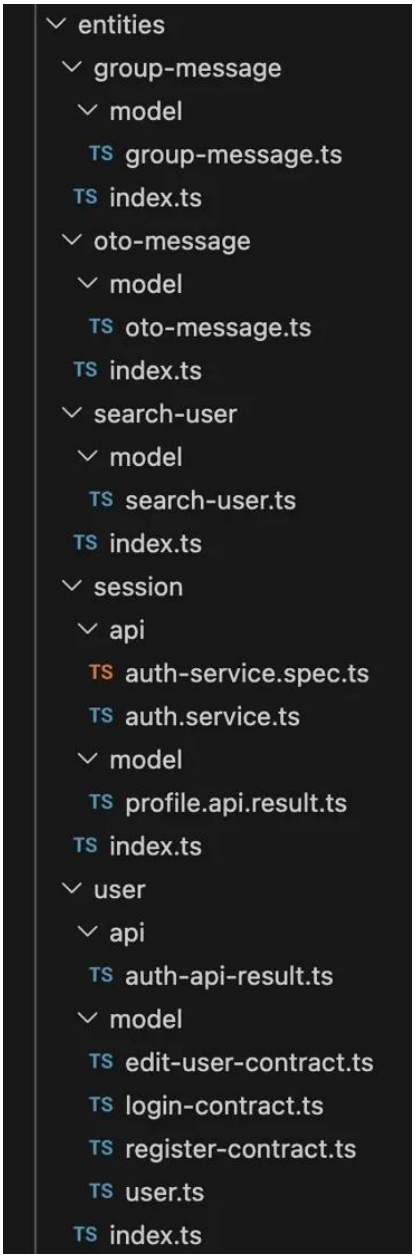

Рисунок 3.10 – Шар entities

Шар entities містить доменні моделі та базові API-сервіси, кожен з яких організований у окремий слайс з власним файлом index.ts для публічного експорту. Слайс group-message містить модель group-message.ts що описує структуру групового повідомлення. Слайс oto-message містить модель oto-message.ts для приватних повідомлень між двома користувачами. Слайс search-user містить модель search-user.ts для представлення результатів пошуку

користувачів. Слайс session містить API-сервіс auth.service.ts для взаємодії з ендпоінтами автентифікації та модель profile.api.result.ts що описує структуру відповіді сервера з даними профілю. Слайс user містить API-результат auth-api-result.ts та чотири моделі: edit-user-contract.ts для запиту зміни даних користувача, login-contract.ts та register-contract.ts для запитів входу і реєстрації відповідно, та user.ts як основну доменну модель користувача.

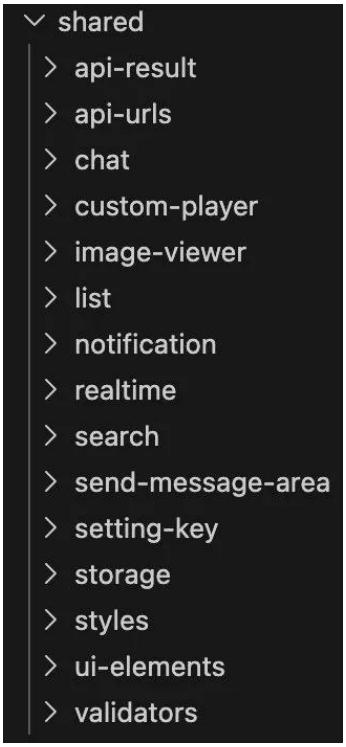

Рисунок 3.11 – Шар shared

Шар shared містить перевикористовуваний інфраструктурний код що не прив'язаний до конкретної функціональності. Підкаталоги api-result та api-urls централізують визначення типів відповідей API та URL-адрес ендпоінтів відповідно. Підкаталог chat містить спільні компоненти та сервіси для роботи з чатами. Підкаталоги custom-player та image-viewer реалізують відтворення медіаконтенту - аудіо, відео та зображень. Підкаталог list містить перевикористовуваний компонент списку. Підкаталог notification містить сервіс для відображення системних сповіщень користувачу. Підкаталог realtime інкапсулює всю логіку SignalR-з'єднання та підписки на події реального часу. Підкаталог search містить компонент пошуку. Підкаталог

send-message-area реалізує компонент області введення та відправки повідомлення з підтримкою медіафайлів. Підкаталог setting-key відповідає за управління налаштуваннями криптографічних ключів користувача. Підкаталог storage інкапсулює роботу з локальним сховищем браузера де зберігаються ключі шифрування. Підкаталог styles містить глобальні стилі застосунку на основі Tailwind CSS. Підкаталог ui-elements містить бібліотеку перевикористовуваних UI-компонентів: кнопки, поля введення, перемикачі, сповіщення тощо. Підкаталог validators містить кастомні валідатори форм Angular.

3.3 Реалізація основних компонентів системи

Архітектура та структура сервісів, описані в попередніх підрозділах, є організаційною основою системи. Проте ключову роль у забезпеченні безпеки та функціональності месенджера відіграють конкретні компоненти, що реалізують криптографічні механізми, автентифікацію та комунікацію в реальному часі. У цьому підрозділі детально розглянуто реалізацію основних компонентів: модуля шифрування та дешифрування повідомлень, модуля автентифікації користувачів, а також механізму взаємодії між сервісами через брокер повідомлень.

3.3.1 Модуль Encryptor та Decryptor

Модуль шифрування та дешифрування реалізований у вигляді двох класів - EncryptionInfo та DecryptionInfo, що реалізують інтерфейси IEncryptionInfo та IDecryptionInfo відповідно. Обидва класи розташовані в окремому проєкті Encryptor та підключаються до кожного мікросервісу як

залежність через механізм ін'єкції залежностей (додаток 4, Encryptor.cs, Decryptor.cs).

При ініціалізації обох класів виконується однакова послідовність дій. З конфігурації зчитується рядковий ключ ChaChaKey, який хешується алгоритмом SHA-256 для отримання 256-бітного ключа шифрування ChaCha20-Poly1305. Паралельно перевіряється наявність файлів RSA-ключів у директорії Keys: якщо файли відсутні, автоматично генерується нова пара ключів та зберігається на диск. Після цього завантажується закритий RSA-ключ для подальшого використання в операціях дешифрування та підпису.

Клас EncryptionInfo реалізує чотири групи операцій. Метод Encrypt виконує симетричне шифрування рядка алгоритмом ChaCha20-Poly1305: генерується випадковий 12-байтний nonce, текст шифрується з тегом автентифікації розміром 128 біт, після чого nonce та шифротекст об'єднуються і кодуються у Base64. Метод EncryptRsa виконує асиметричне шифрування рядка відкритим RSA-ключем отримувача з використанням схеми заповнення OAEP-SHA256. Методи EncryptObjectStrings та EncryptObjectStringsForUpdate виконують масове шифрування всіх рядкових властивостей довільного об'єкта через рефлексію, пропускаючи поля що містять хеші та ідентифікатори. Метод EncryptRsaObjectStrings аналогічно виконує масове RSA-шифрування властивостей об'єкта відкритим ключем отримувача. Метод GetPublicKey повертає вміст файлу з відкритим RSA-ключем сервісу для передачі іншим учасникам обміну.

Клас DecryptionInfo реалізує симетричні операції дешифрування. Метод Decrypt виконує зворотну операцію до Encrypt: з вхідного Base64-рядка виділяється nonce перших 12 байт та шифротекст, після чого ChaCha20-Poly1305 дешифрує дані та перевіряє тег автентифікації. Метод DecryptRsa дешифрує рядок закритим RSA-ключем поточного сервісу. Методи DecryptObjectStrings та DecryptRsaObjectStrings виконують масове дешифрування властивостей об'єкта через рефлексію, пропускаючи поля з

хешами. В методі DecryptRsaObjectStrings помилки дешифрування окремих полів перехоплюються та логуються без переривання обробки решти властивостей, що забезпечує стійкість до часткового збою.

Таким чином модуль реалізує гібридну схему шифрування: ChaCha20-Poly1305 використовується для ефективного шифрування основного обсягу даних, що зберігаються у базі даних, тоді як RSA застосовується для захищеної передачі повідомлень між сервісами. Винесення модуля в окремий проєкт дозволяє підключати його до будь-якого мікросервісу системи без дублювання коду (рис. Д2.4).

3.3.2 Модуль реєстрації та автентифікації користувача

Реєстрація користувача реалізована у класі RegisterOrchestrator, що відповідає за координацію всіх операцій пов'язаних зі створенням нового облікового запису. Клас отримує всі необхідні залежності через ін'єкцію залежностей, що відповідає принципам Clean Architecture (додаток 4, RegisterOrchestrator.cs).

Процес реєстрації починається з отримання відкритих RSA-ключів сервісів Notification та Messaging зі сховища публічних ключів. Якщо ключі відсутні - реєстрація переривається, оскільки без них неможливо безпечно передати дані іншим сервісам. Після цього виконується валідація вхідних даних через FluentValidation і у разі помилок повертається відповідний результат.

Далі виконується підготовка даних користувача: пароль хешується алгоритмом з сіллю через IHasherPassword, а поля login, email та nickname хешуються через IHasher для збереження у базі даних у вигляді хешів - це унеможливлює відновлення оригінальних значень навіть при компрометації бази. Після маппінгу у доменну модель та встановлення хешів усі рядкові поля

об'єкта шифруються алгоритмом ChaCha20-Poly1305 через метод EncryptObjectStrings.

Перед збереженням користувача система перевіряє його унікальність через запит до MessagingSystem.Services.Messaging за допомогою MassTransit request-response: хеш нікнейму та аватар шифруються відкритим RSA-ключем Messaging сервісу та надсилаються запитом AddUserRequest. Якщо користувач вже існує - реєстрація переривається.

Після успішного збереження користувача у базі формується подія ConfirmUserEmail з іменем, зашифрованою електронною адресою та хешем нікнейму, яка шифрується відкритим RSA-ключем Notification сервісу та публікується у чергу RabbitMQ. Сервіс Notification отримує цю подію, дешифрує дані своїм закритим ключем та надсилає лист підтвердження на електронну пошту користувача.

Метод ConfirmEmailAsync обробляє підтвердження електронної пошти: декодує хеш нікнейму з URL-encoded формату, знаходить користувача в базі та встановлює прапорець EmailConfirmed у значення true.

Автентифікація користувача реалізована у класі LoginOrchestrator (додаток 4, LoginOrchestrator.cs, JwtService.cs). Процес входу починається з валідації вхідних даних через FluentValidation. Після успішної валідації виконується пошук користувача в базі даних за хешем логіну - оскільки логін зберігається у хешованому вигляді, пошук виконується через hasher.Hash(loginDto.Login), що унеможливлює пошук за оригінальним значенням навіть всередині сервісу.

Далі виконується серія перевірок: якщо користувача не знайдено - повертається помилка, якщо електронна пошта не підтверджена - вхід блокується до підтвердження, якщо хеш введеного нікнейму не збігається з хешем у базі - повертається відповідна помилка. Така багаторівнева перевірка унеможливлює вхід навіть при знанні лише логіну та пароля без нікнейму.

Фінальним кроком є автентифікація пароля: збережений хешований пароль дешифрується з бази даних методом Decrypt, після чого IJwtService порівнює його з введеним паролем та у разі успіху генерує JWT-токен і встановлює його у cookie. Таким чином пароль у відкритому вигляді ніколи не зберігається в базі - він шифрується ChaCha20-Poly1305 і лише тимчасово дешифрується в пам'яті для порівняння.

3.3.3 Модуль комунікації в реальному часі

Комунікація в реальному часі реалізована через два окремих SignalR хаби: OtoChatHub для приватних повідомлень між двома користувачами та GroupChatHub для групових чатів (додаток 4, OtoChatHub.cs, GroupChatHub.cs). Обидва хаби захищені атрибутом Authorize, що означає що підключення без валідного JWT-токена буде відхилено ще до виклику будь-якого методу хабу.

Ідентифікація користувача в обох хабах відбувається через JWT-claim типу ClaimTypes.UserData, з якого витягується нікнейм поточного користувача. При підключенні кожен користувач автоматично додається до SignalR-групи зі своїм нікнеймом через метод OnConnectedAsync, а при відключенні видаляється з неї через OnDisconnectedAsync. Це дозволяє адресно надсилати повідомлення конкретному користувачу через Clients.Group(nickname).SendAsync незалежно від кількості його активних підключень.

OtoChatHub реалізує повний цикл роботи з приватними повідомленнями. Метод SendMessageAsync отримує нікнейм отримувача та зашифрований на стороні клієнта вміст повідомлення, зберігає його через IMessageOrchestrator та сповіщає обох учасників через допоміжний метод NotifyUsersAsync, що надсилає подію ReceivePrivateMessage одночасно відправнику та отримувачу. Метод DeleteMessageAsync підтримує два режими видалення: soft-delete, що

позначає повідомлення як видалене зі збереженням у базі, та hard-delete, що повністю видаляє запис. В обох випадках обидва учасники отримують сповіщення MessageDeleted з типом видалення. Метод LoadChatHistoryAsync завантажує історію повідомлень з підтримкою пагінації через параметри skip та take. Метод ReplyToMessageAsync спочатку зберігає нове повідомлення, після чого прив'язує його як відповідь на вказане повідомлення.

GroupChatHub реалізує розширений набір функцій для групових чатів. Окрім стандартних операцій з повідомленнями - відправка, редагування, видалення та відповідь - хаб забезпечує управління групами: створення через CreateGroupAsync, редагування через EditGroupAsync, видалення через DeleteGroupAsync, а також додавання та видалення учасників через AddMembersToGroupAsync та RemoveMembersFromGroupAsync. При видаленні учасників кожен з них отримує персональне сповіщення UserRemovedFromGroup через допоміжний метод NotifyRemovedUsersAsync. Сповіщення учасників групи реалізовано через метод NotifyUsersInGroupAsync, що отримує актуальний список учасників групи та надсилає подію кожному з них через їх персональну SignalR-групу. Метод FindMessageByFilterAsync дозволяє фільтрувати повідомлення за відправником, отримувачем та датою.

Важливою особливістю обох хабів є те, що вміст повідомлень не дешифрується на стороні сервера - хаб отримує вже зашифровані дані від клієнта-відправника та передає їх отримувачу у незмінному вигляді. Дешифрування відбувається виключно на пристрої отримувача, що забезпечує реалізацію принципу E2EE на рівні транспортного шару системи.

3.3.4 Модуль E2EE на клієнтській частині

Наскрізне шифрування на клієнтській частині реалізовано у сервісі E2eeService, що використовує криптографічну бібліотеку libsodium-wrappers-

sumo та bip39 для генерації мнемонічних фраз (додаток 4, e2ee.ts). Сервіс реалізує два окремих протоколи шифрування: Double Ratchet для приватних чатів та Sender Key для групових.

Генерація ключів починається зі створення мнемонічної фрази з 256 біт ентропії через bip39. З мнемоніки через функцію mnemonicToSeed32 отримується детермінований 32-байтний seed, з якого генерується пара ключів Ed25519 через keypairFromSeed32. Оскільки libsodium використовує Curve25519 для операцій Діффі-Хеллмана, Ed25519-ключі конвертуються у X25519-формат через convertEdToX. Таким чином користувач має дві пари ключів: Ed25519 для підписів та X25519 для шифрування. Згенеровані ключі зберігаються виключно в пам'яті сервісу, а стан ratchet додатково зберігається у sessionStorage браузера для відновлення між навігаціями. Для резервного копіювання ключів реалізовано метод createEncryptedBackupAndDownload, що шифрує мнемоніку та закритий ключ паролем користувача через алгоритм Argon2id та завантажує зашифрований файл на пристрій.

Для приватних повідомлень реалізовано протокол Double Ratchet, що поєднує ланцюжок Діффі-Хеллмана з симетричним ratchet-ланцюжком. Ініціалізація ratchet для відправника виконується через initRatchetAsSender: генерується ефемерна пара ключів, виконуються три операції Діффі-Хеллмана між різними комбінаціями ключів відправника та отримувача, результати об'єднуються та хешуються для отримання кореневого ключа rootKey. З нього деривуються sendingChainKey для відправки та receivingChainKey для отримання. Отримувач виконує дзеркальні операції через initRatchetAsReceiver для отримання ідентичних ключів.

Шифрування повідомлення через encryptMessageWithHistory виконується таким чином: з поточного sendingChainKey та номера повідомлення деривується унікальний messageKey, яким шифрується текст через crypto_secretbox_easy з випадковим nonce. Після шифрування sendingChainKey оновлюється хешуванням для forward secrecy. Кожні 50

повідомлень автоматично виконується DH-ratchet: генерується нова ефемерна пара ключів, оновлюється rootKey та sendingChainKey. Дешифрування через decryptMessage враховує три сценарії: новий ratchet-ідентифікатор що означає повне оновлення стану, зміна ефемерного ключа що означає DH-крок, та звичайне повідомлення в поточному ланцюжку. Пропущені повідомлення обробляються через збереження їх ключів у messageKeysCache для подальшого дешифрування.

Для групових чатів реалізовано протокол Sender Key, де кожен учасник має власний ланцюжок ключів. Через generateSenderKey відправник створює випадковий chainKey та унікальний senderKeyId. Цей ключ шифрується для кожного учасника групи окремо через encryptSenderKeyForMember: генерується ефемерна пара ключів, виконується операція Діффі-Хеллмана з відкритим ключем учасника, результат використовується як ключ шифрування bundle з chainKey через crypto_secretbox_easy. Отримувач дешифрує bundle через decryptSenderKeyFromMember власним закритим ключем.

Шифрування групового повідомлення через encryptGroupMessage деривує messageKey з поточного chainKey та chainIndex, після чого оновлює chainKey хешуванням. Дешифрування через decryptGroupMessage відновлює messageKey з отриманого chainKey та chainIndex відправника. При видаленні учасника з групи виконується ротація ключів через rotateSenderKey з новим senderKeyId, що позбавляє видаленого учасника можливості дешифрувати подальші повідомлення.

ВИСНОВОК ДО РОЗДІЛУ 3

У розділі 3 було здійснено всебічне проєктування інформаційної системи захищеного обміну повідомленнями із наскрізним шифруванням, що відображає її функціональну, логічну та технологічну структуру. Проєктування почалося з формування архітектурної моделі серверної частини, в основі якої закладено мікросервісну архітектуру. Така архітектура дозволяє чітко розподілити зони відповідальності між трьома незалежними сервісами: MessagingSystem.Services.User, що забезпечує реєстрацію та автентифікацію, MessagingSystem.Services.Messaging, що відповідає за комунікацію в реальному часі, та MessagingSystem.Services.Notification, що реалізує поштові сповіщення. Важливо, що кожен із сервісів побудований відповідно до принципів Clean Architecture, що забезпечує чітке розмежування шарів бізнес-логіки, інфраструктури та представлення і дозволяє замінювати окремі технічні компоненти без впливу на доменну логіку.

Технічне рішення щодо реалізації криптографічних механізмів ґрунтується на гібридній схемі шифрування: алгоритм ChaCha20-Poly1305 використовується для ефективного захисту даних, що зберігаються в базі даних, тоді як RSA із схемою OAEP-SHA256 застосовується для безпечної міжсервісної передачі ключів. Модуль Encryptor, винесений в окремий проєкт, підключається до кожного мікросервісу як залежність, що виключає дублювання коду та забезпечує єдину криптографічну політику в межах усієї системи. Механізм реєстрації та автентифікації додатково підсилено хешуванням чутливих полів користувача, що унеможливлює відновлення оригінальних значень навіть у разі компрометації бази даних.

На клієнтській стороні реалізовано SPA-застосунок на основі Angular з методологією Feature-Sliced Design, що забезпечує чіткий поділ коду за функціональними областями та рівнями абстракції. Ізоляція модуля

шифрування, модуля чатів та модуля автентифікації у межах FSD-шарів відповідає загальному архітектурному принципу незалежності критично важливих підсистем. Застосування TypeScript і Tailwind CSS разом із централізованою підтримкою чотирьох мов через i18n забезпечує зручний та адаптивний інтерфейс для кінцевого користувача.

Окрему увагу приділено реалізації механізму наскрізного шифрування на клієнтській стороні за допомогою бібліотеки libsodium. Для приватних чатів застосовано протокол Double Ratchet, що поєднує ланцюжок Діффі-Хеллмана із симетричним ключовим ланцюжком і забезпечує forward secrecy на рівні кожного повідомлення. Для групових чатів реалізовано протокол Sender Key з автоматичною ротацією ключів при видаленні учасників. Комунікація в реальному часі здійснюється через два захищені SignalR-хаби, при цьому сервер оперує виключно зашифрованими даними, а дешифрування відбувається лише на пристрої отримувача.

Загалом, результати цього розділу демонструють технічну і концептуальну готовність системи до реалізації. Обрані інструменти, архітектурні рішення та принципи побудови повністю відповідають поставленим вимогам до безпеки, ефективності та масштабованості. Запропонована система має потенціал для подальшого розвитку і впровадження у сферах, де потрібен надійний захист приватної комунікації від несанкціонованого доступу без довіри до серверної інфраструктури.

РОЗДІЛ 4
ДОСЛІДЖЕННЯ ТА АНАЛІЗ РОЗРОБЛЕННОЇ СИСТЕМИ

Базуючись на розробленій системі захищеного обміну повідомленнями, метою даного розділу є показати результати її роботи та підтвердити відповідність сформульованим вимогам. Для цього проведено автоматизоване тестування серверної та клієнтської частин системи, наведено опис інтерфейсу користувача з основними сценаріями використання, проаналізовано схему бази даних, а також сформульовано рекомендації щодо подальшого розвитку системи.

4.1 Unit та Integration тести

Тестування розробленої системи проводилось на двох рівнях: серверній та клієнтській частинах. Для кожної з них застосовувався відповідний набір інструментів, описаних у підрозділі 2.3.6.

Серверна частина покрита 1329 автоматизованими тестами, розподіленими між п'ятьма тестовими проєктами.

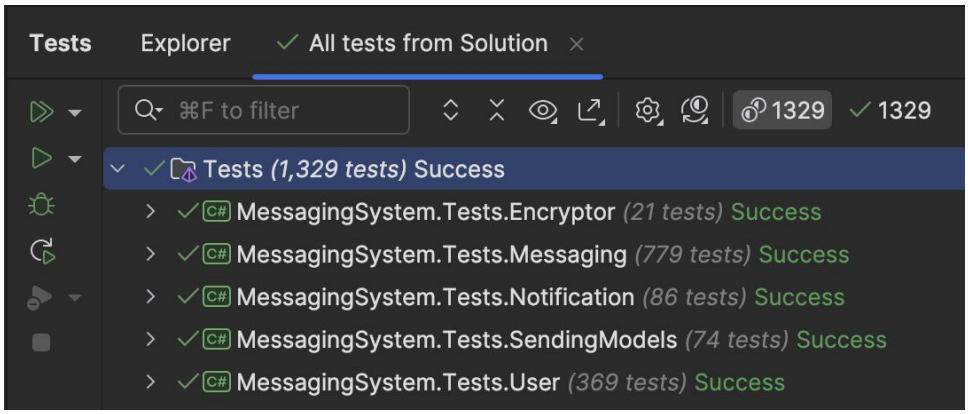

Рисунок 4.1 – Тести серверної частини

MessagingSystem.Tests.Encryptor містить 21 тест що перевіряє коректність роботи модуля шифрування та дешифрування, зокрема симетричне шифрування через ChaCha20-Poly1305 та асиметричне через RSA. MessagingSystem.Tests.Messaging є найбільшим тестовим проєктом і налічує 779 тестів що охоплюють логіку роботи з приватними та груповими чатами, SignalR хабами та кешуванням через Redis. MessagingSystem.Tests.Notification містить 86 тестів для перевірки логіки відправки електронних листів через MailJet. MessagingSystem.Tests.SendingModels містить 74 тести для перевірки моделей повідомлень що передаються між сервісами через RabbitMQ. MessagingSystem.Tests.User налічує 369 тестів що перевіряють логіку реєстрації, автентифікації та управління даними користувача. Всі 1329 тестів виконались успішно, що підтверджує коректність реалізації бізнес-логіки кожного мікросервісу.

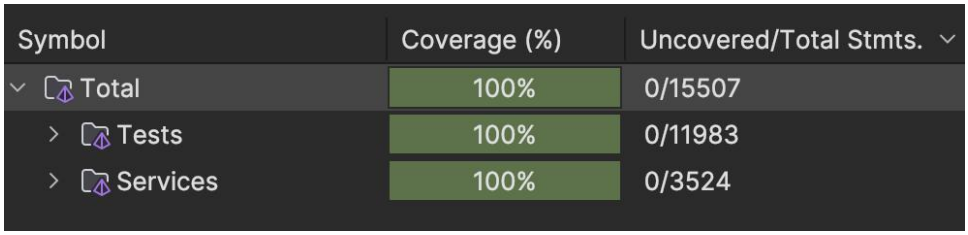

Рисунок 4.2 – Результати тестування серверної частини

Клієнтська частина тестувалась за допомогою Jasmine та Karma у браузері Chrome 143.0.0.0. Загалом виконано 2307 тестів, всі успішно. За результатами аналізу покриття коду досягнуто 100% по всіх чотирьох метриках: покриття інструкцій становить 100% (5688 з 5688), покриття гілок - 100% (1890 з 1890), покриття функцій - 100% (1487 з 1487), покриття рядків - 100% (5266 з 5266). Загальний час виконання тестів склав 8.787 секунди.

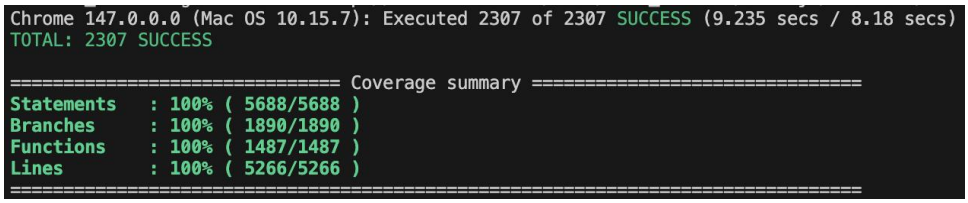

Рисунок 4.3 – Результати тестування клієнтської частини

Досягнуте 100% покриття коду на клієнтській частині та повне проходження всіх серверних тестів підтверджують відповідність реалізованої системи сформульованим функціональним та нефункціональним вимогам.

4.2 Інструкція користувача

З метою забезпечення ефективного використання розробленої системи захищеного обміну повідомленнями, у цьому підрозділі подано інструкцію для кінцевого користувача. Наведені відомості охоплюють основні сценарії роботи із системою, включаючи процедури реєстрації та авторизації, обміну приватними та груповими повідомленнями, а також керування профілем і налаштуваннями інтерфейсу. Інструкція покликана зробити процес користування інтуїтивно зрозумілим та зменшити кількість помилок при роботі з системою.

4.2.1 Реєстрація та авторизація користувача

Першим етапом взаємодії з системою WhisperNet є процедура реєстрації або авторизації користувача. Форма реєстрації, зображена на рисунку 4.4, містить обов'язкові поля: ім'я, прізвище, логін, електронну пошту, нікнейм, пароль, а також поле для завантаження аватара у форматах JPEG, PNG, GIF, WebP або SVG розміром до 5 МБ.

Після заповнення форми виконується клієнтська валідація введених даних - у разі некоректного заповнення поля підсвічуються з відповідним повідомленням про помилку, що зображено на рисунку 4.5. На серверній стороні перевіряється унікальність логіну, електронної пошти та нікнейму користувача - їх хешовані значення порівнюються з наявними записами в базі даних.

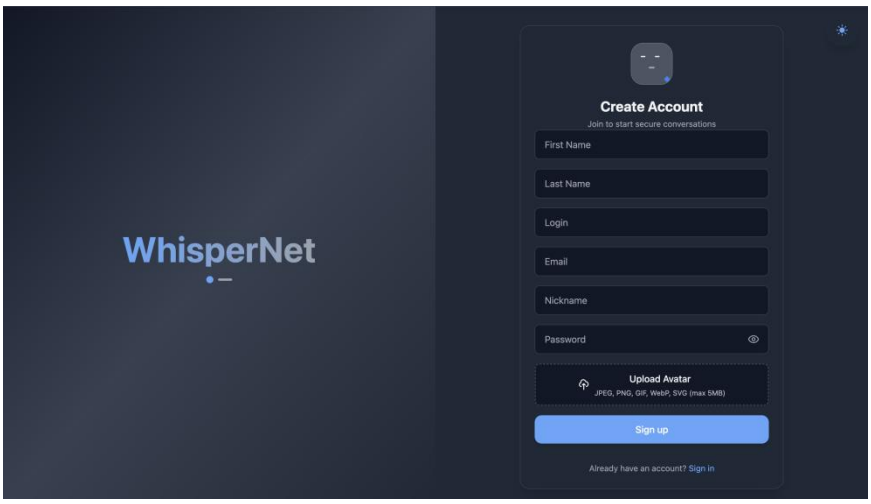

Рисунок 4.4 – Форма реєстрації користувача

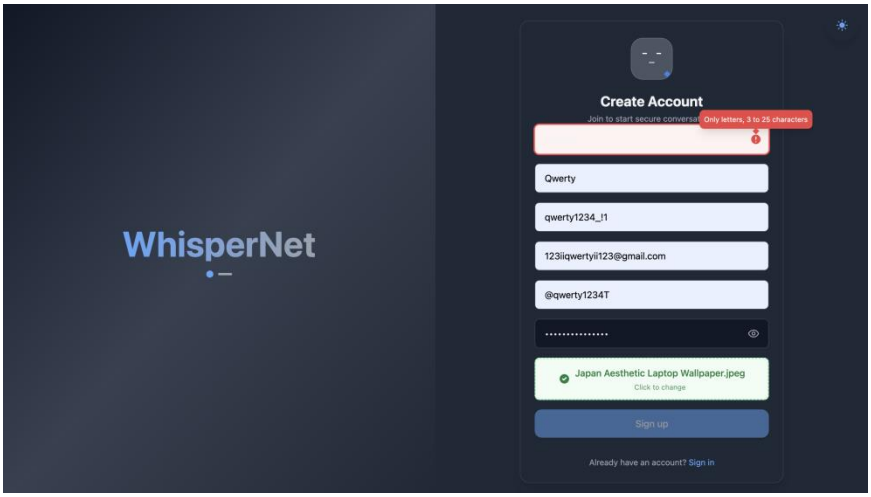

Рисунок 4.5 – Валідація даних користувача

Після успішної реєстрації система генерує унікальну мнемонічну фразу з 24 слів, що зображено на рисунку 4.6. Ця фраза є єдиним способом відновлення облікового запису та криптографічних ключів користувача, тому система наполегливо рекомендує її зберегти. Для підтвердження збереження фрази користувач має ввести три випадково обрані слова з неї, що зображено на рисунку 4.7.

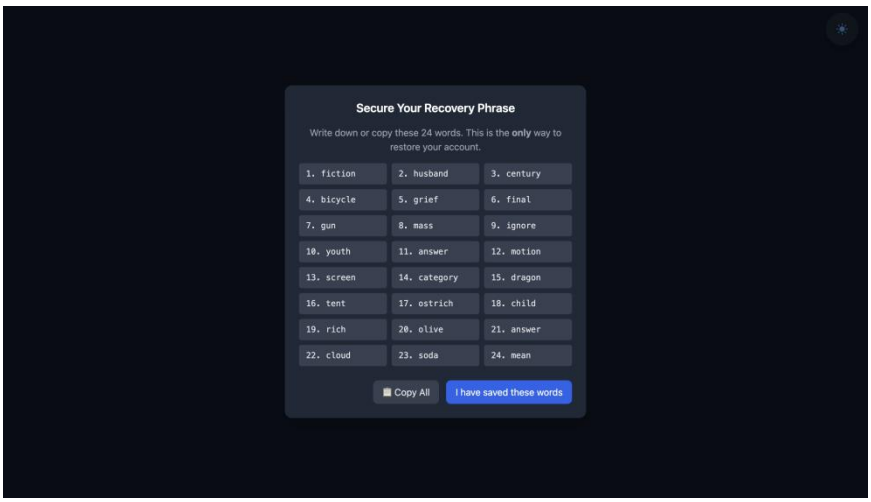

Рисунок 4.6 – Генерація мнемонічної фрази відновлення

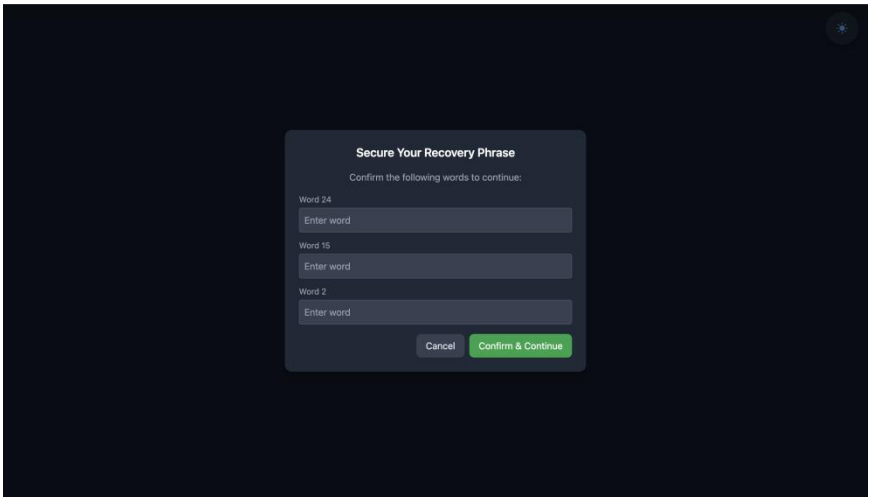

Рисунок 4.7 – Підтвердження збереження мнемонічної фрази

Такий механізм гарантує що користувач дійсно зберіг фразу перед продовженням. Наступним кроком є створення резервної копії ключів шифрування, зображеного на рисунку 4.8 - користувач може захистити файл резервної копії додатковим паролем або пропустити цей крок, після чого JSON-файл з ключами автоматично завантажується на пристрій.

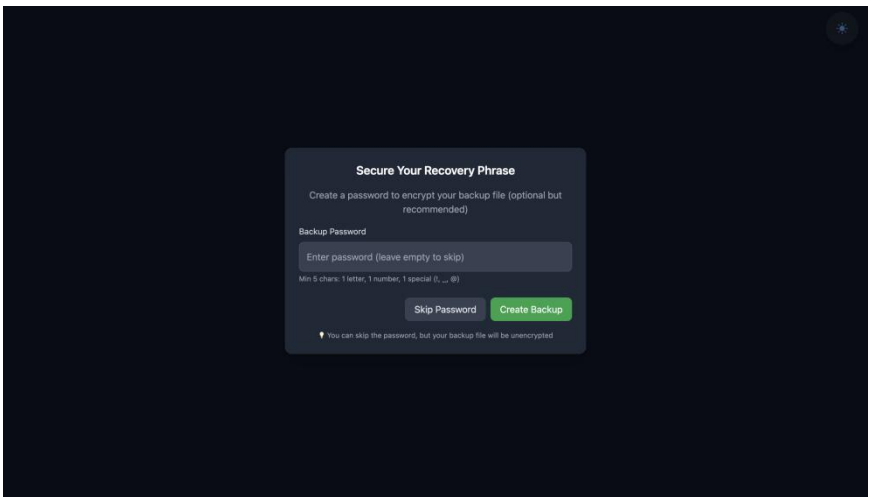

Рисунок 4.8 – Створення резервної копії ключів шифрування

При повторному вході до системи користувач вводить логін, нікнейм та пароль у формі авторизації, зображеній на рисунку 4.9. Після перевірки пароля система відображає модальне вікно відновлення ключів шифрування, де користувач завантажує збережений раніше JSON-файл резервної копії та вводить пароль захисту (якщо він був встановлений). Це забезпечує відновлення криптографічного стану сесії та доступ до зашифрованого листування.

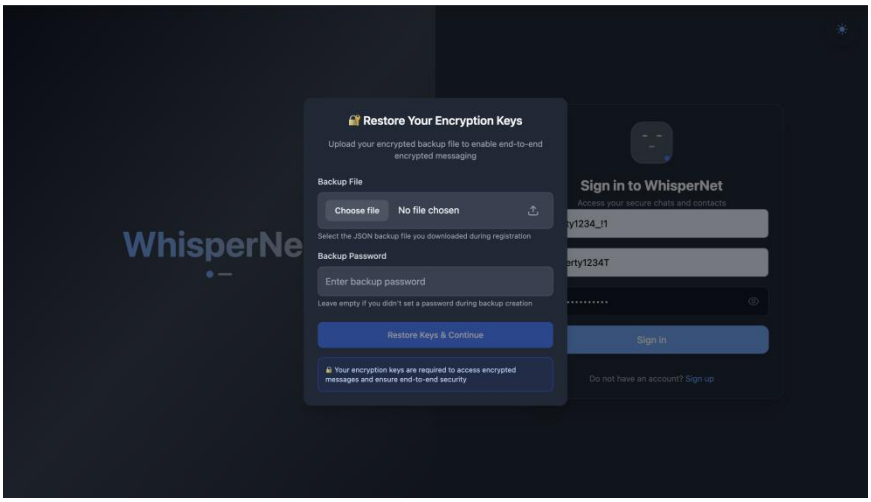

Рисунок 4.9 – Завантаження резервної копії ключів при авторизації

4.2.2 Робота з повідомленнями

Розділ повідомлень є основним функціональним модулем системи. Після авторизації користувач потрапляє до списку приватних чатів, зображеного на рисунку 4.10, де відображаються всі наявні діалоги. Пошук по контактах здійснюється через рядок пошуку у верхній частині панелі.

Рисунок 4.10 – Список приватних чатів

При виборі чату відкривається його інтерфейс, зображений на рисунку 4.11, що містить історію повідомлень, поле для введення тексту та кнопки для відправлення повідомлення і прикріплення файлів. Усі повідомлення передаються у зашифрованому вигляді відповідно до протоколу Double Ratchet та дешифруються виключно на пристрої отримувача.

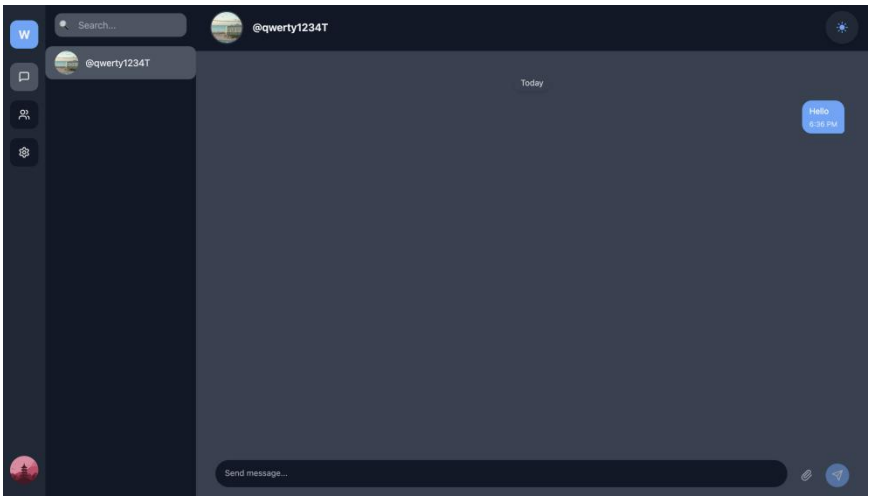

Рисунок 4.11 – Інтерфейс приватного чату

Для взаємодії з окремим повідомленням користувач викликає контекстне меню, зображене на рисунку 4.12, що містить три дії: відповісти, редагувати та видалити.

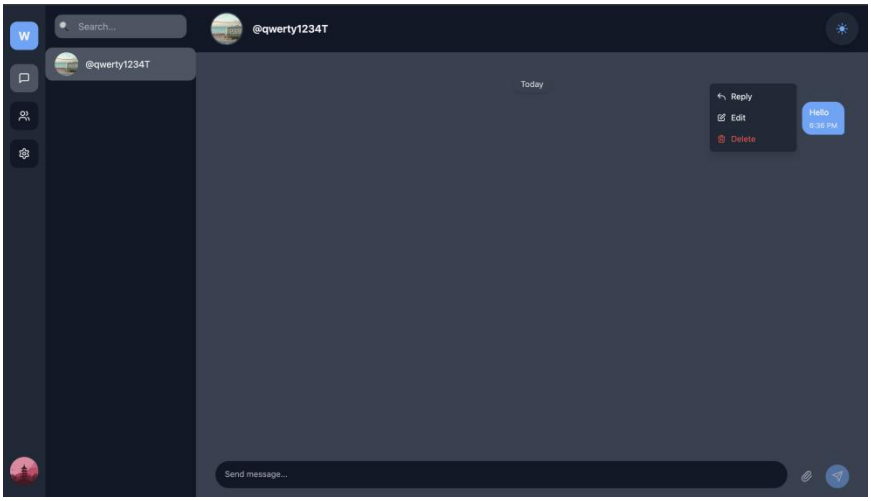

Рисунок 4.12 – Контекстне меню повідомлення

При виборі видалення система відображає модальне вікно підтвердження, зображене на рисунку 4.13, де користувач може видалити повідомлення лише для себе. Якщо активувати опцію «Also delete for both», зображену на рисунку 4.14, повідомлення буде видалено і для співрозмовника. Функція відповіді на повідомлення, зображена на рисунку 4.15, дозволяє процитувати конкретне повідомлення у відповіді, що забезпечує зручність ведення діалогу.

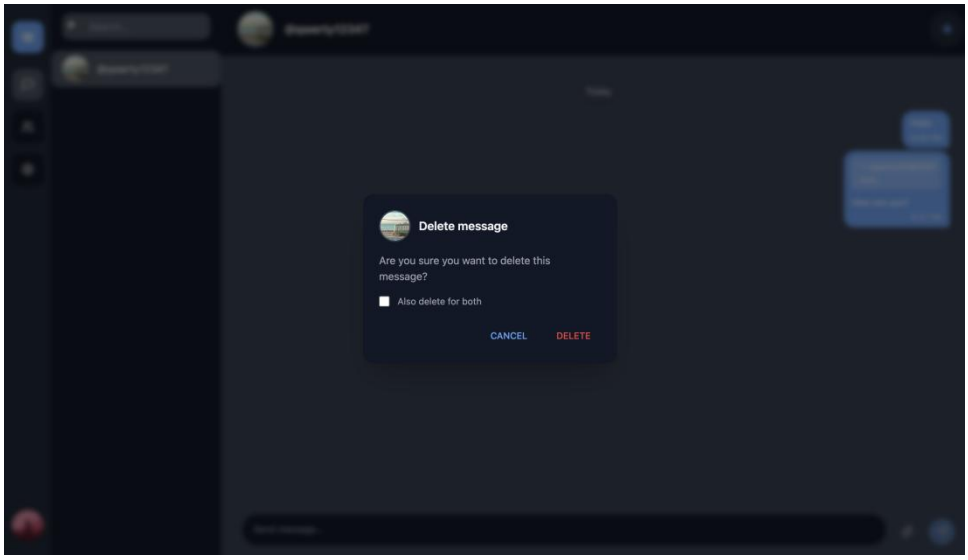

Рисунок 4.13 – Видалення повідомлення для себе

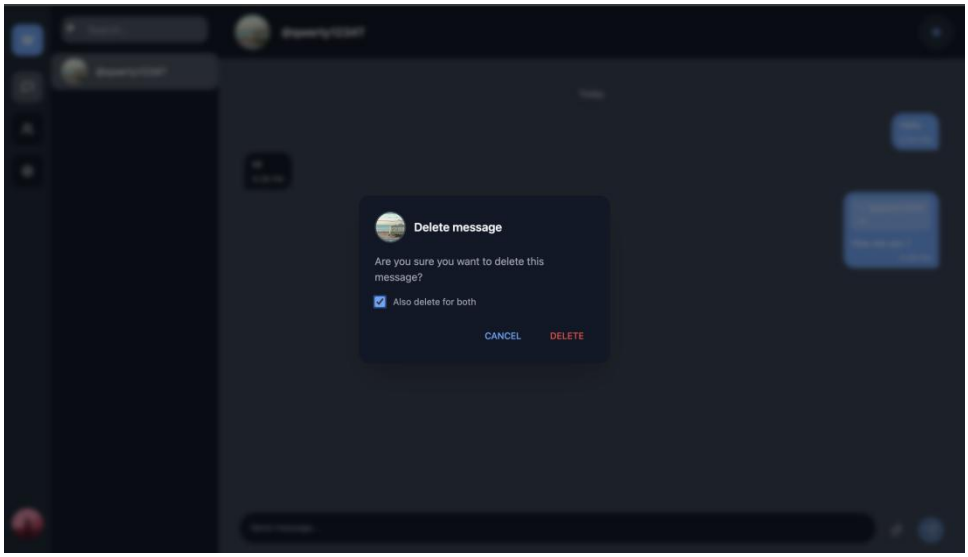

Рисунок 4.14 – Видалення повідомлення для обох учасників

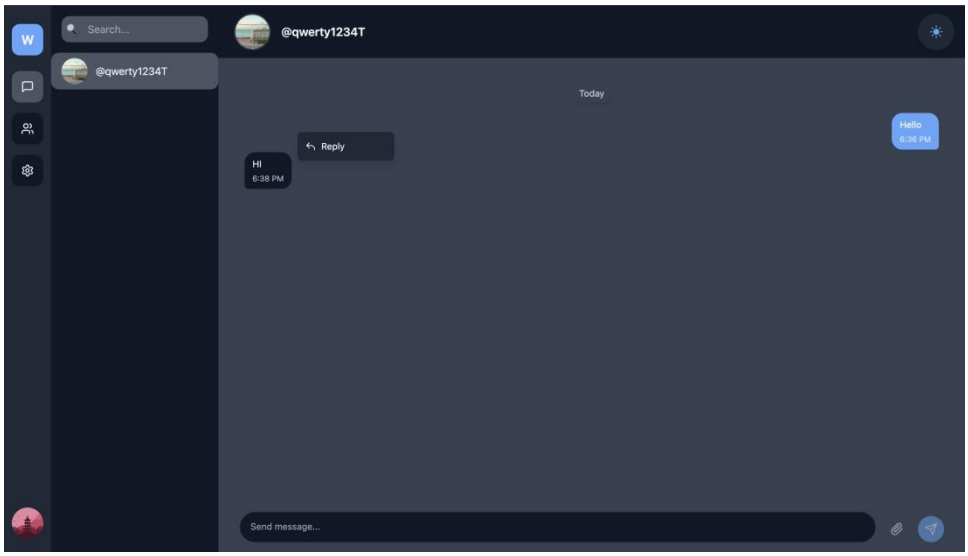

Рисунок 4.15 – Відповідь на повідомлення

Система підтримує відправлення файлів різних типів. При натисканні кнопки прикріплення відкривається модальне вікно, зображене на рисунку 4.16, де відображається обраний файл, його розмір та поле для додавання підпису. Після відправлення файли відображаються безпосередньо у стрічці повідомлень - документи у вигляді картки з назвою та розширенням, зображення та відео у вигляді мініатюри, аудіофайли у вигляді програвача, що зображено на рисунках 4.17 та 4.18.

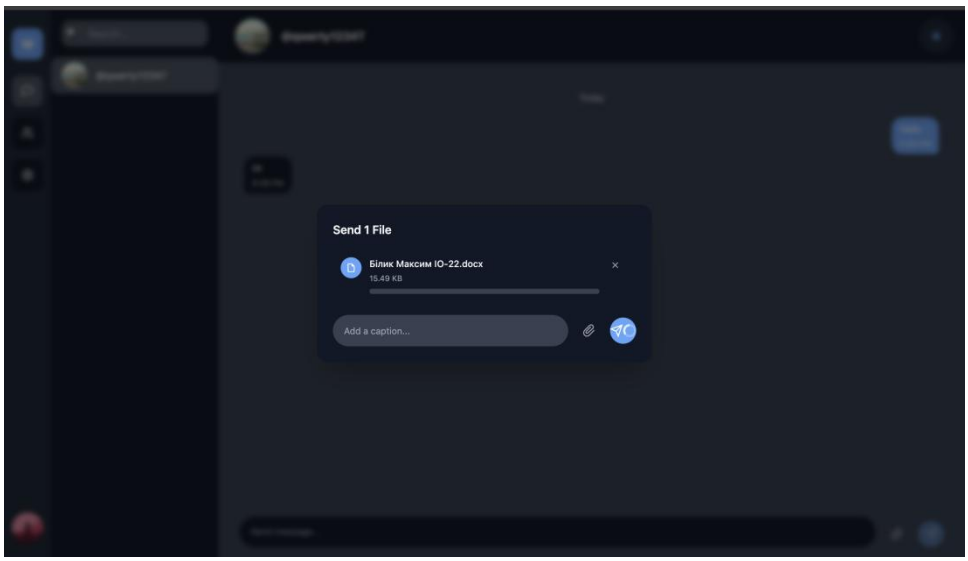

Рисунок 4.16 – Відправлення файлу

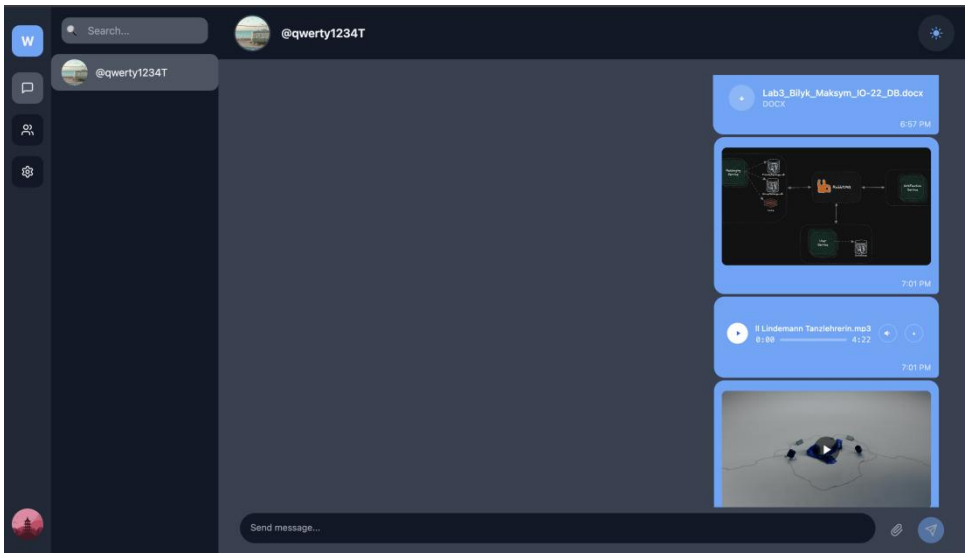

Рисунок 4.17 – Отримані медіафайли та документи

Окрім приватних чатів система підтримує групове спілкування - інтерфейс групового чату, зображений на рисунку 4.19, відображає нікнейм відправника біля кожного повідомлення, підтримує відповіді на повідомлення з цитуванням та пошук по групах у верхній панелі, а також підтримує повний функціонал роботи з повідомленнями - відповідь, редагування, видалення для себе або для всіх учасників та відправлення файлів різних типів.

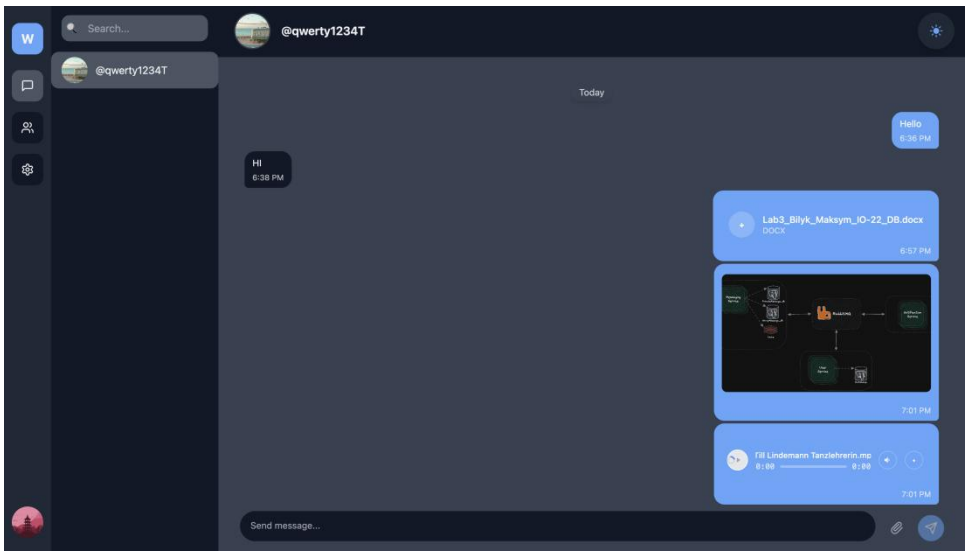

Рисунок 4.18 – Інтерфейс приватного чату з медіафайлами

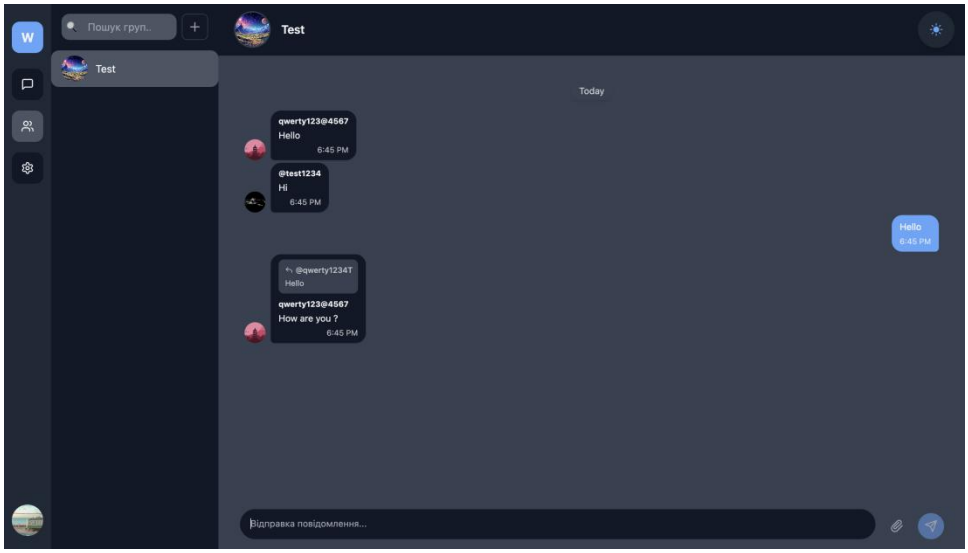

Рисунок 4.19 – Інтерфейс групового чату

Детальний алгоритм відправки зашифрованого повідомлення наведено в Додатку 3

4.2.3 Керування групою

Для створення нового групи користувач переходить до розділу груп та натискає кнопку “+”, після чого відкривається форма, зображена на рисунку 4.20. Форма містить поле для завантаження аватара групи, поля для введення

назви та опису, а також рядок пошуку користувачів для додавання учасників. Пошук здійснюється в реальному часі - при введенні нікнейму система відображає відповідні результати, а обрані учасники фіксуються у вигляді тегів під рядком пошуку з можливістю їх видалення до завершення створення групи. Після натискання кнопки «Create Group» група створюється.

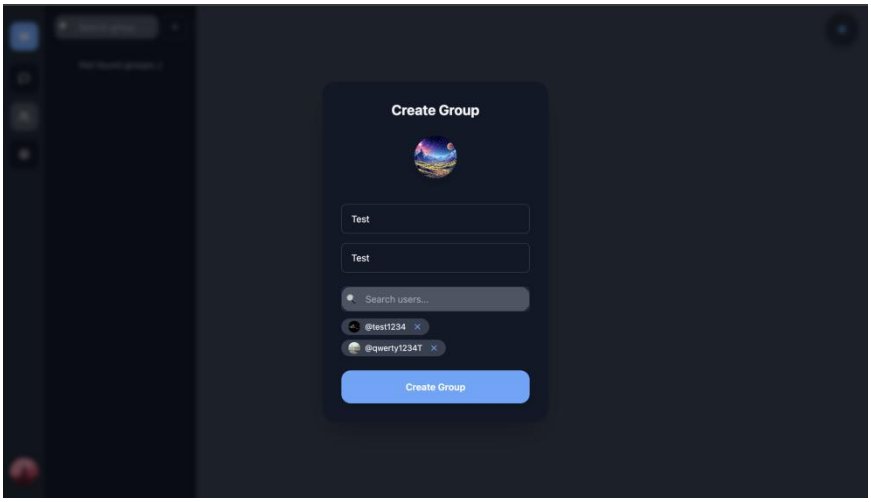

Рисунок 4.20 – Форма створення групи

Інтерфейс групового чату, зображений на рисунку 4.21, відображає повідомлення всіх учасників з їхніми нікнеймами та аватарами, що дозволяє легко ідентифікувати відправника у багатоучасниковому діалозі. Повідомлення поточного користувача відображаються праворуч, повідомлення інших учасників - ліворуч, що є стандартною та інтуїтивно зрозумілою схемою розташування. Контекстне меню повідомлення, що викликається натисканням на нього, підтримує ті самі дії що й у приватному чаті - відповідь з цитуванням, редагування тексту та видалення як для себе так і для всіх учасників групи. Усі повідомлення у груповому чаті шифруються за протоколом Sender Key, що забезпечує конфіденційність переписки навіть у разі компрометації серверної інфраструктури.

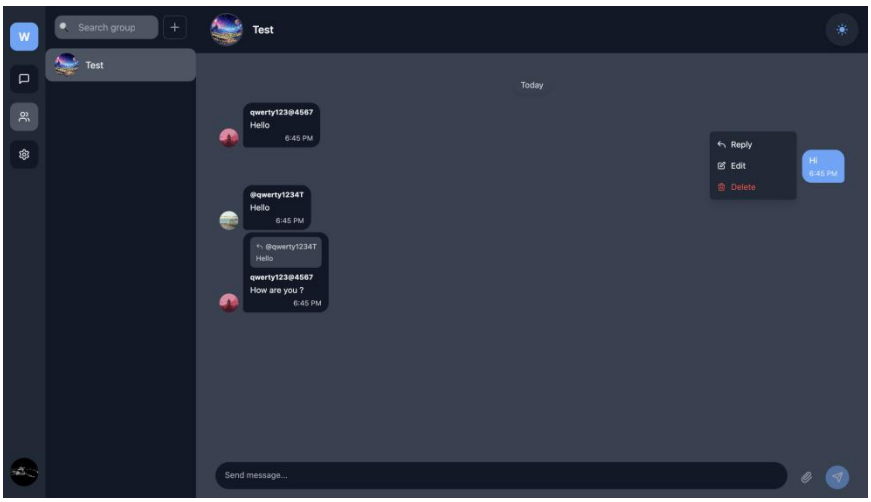

Рисунок 4.21 – Контекстне меню повідомлення у груповому чаті

4.2.4 Інформація про користувача та кастомізація

WhisperNet надає користувачу можливість кастомізації інтерфейсу та керування особистими даними через розділ налаштувань. Інтерфейс підтримує світлу та темну теми - перемикач розташований у правому верхньому куті кожної сторінки, що забезпечує швидкий доступ незалежно від поточного розділу. Світла тема інтерфейсу зображена на рисунку 4.22..

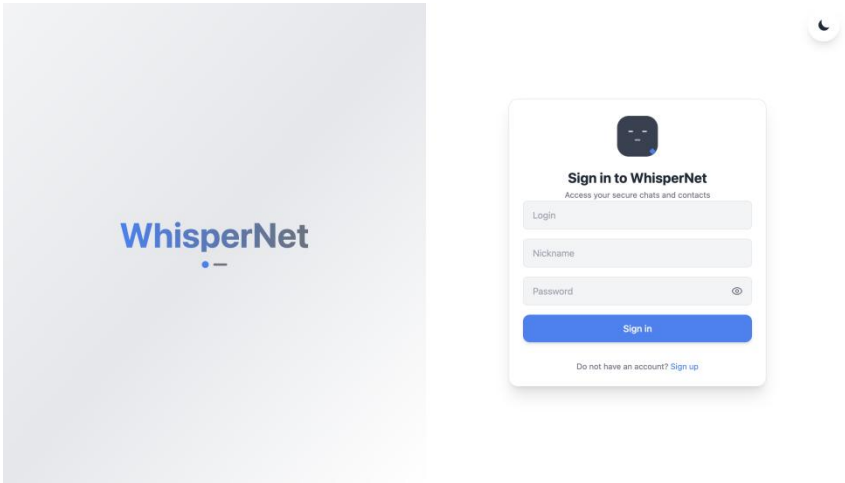

Рисунок 4.22 – Світла тема інтерфейсу

Розділ налаштувань, зображений на рисунку 4.23, містить три секції. Секція Notifications дозволяє вмикати або вимикати сповіщення про нові повідомлення. Секція Appearance надає вибір з восьми колірних тем

інтерфейсу, що дозволяє адаптувати зовнішній вигляд застосунку до індивідуальних уподобань користувача. Секція Language містить перемикач мови інтерфейсу з підтримкою чотирьох мов: англійської, української, німецької та іспанської.

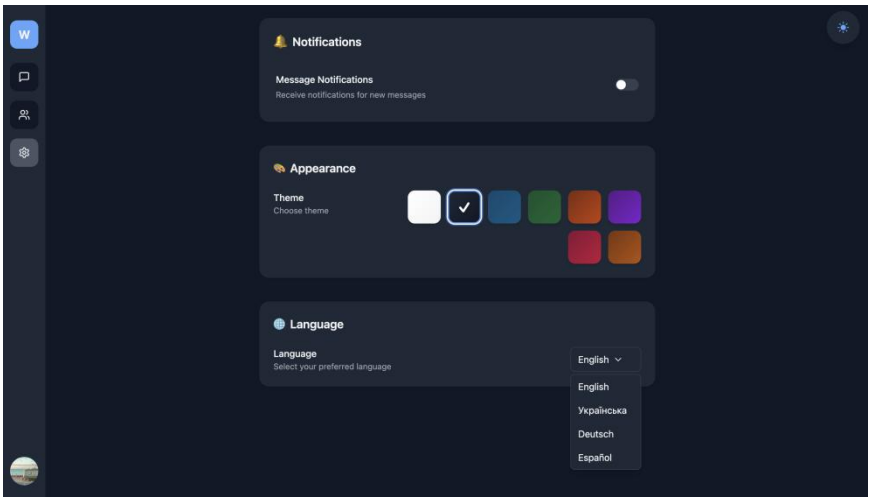

Рисунок 4.23 – Налаштування сповіщень, теми та мови інтерфейсу

Розділ профілю користувача відображає аватар та нікнейм. Нижче розташована картка персональних даних у режимі перегляду, зображена на рисунку 4.24, де у зручному табличному форматі відображаються ім'я, прізвище, нікнейм та електронна пошта користувача. У нижній частині картки розташована кнопка «Delete Account», що ініціює процедуру повного видалення облікового запису та всіх пов'язаних даних з системи.

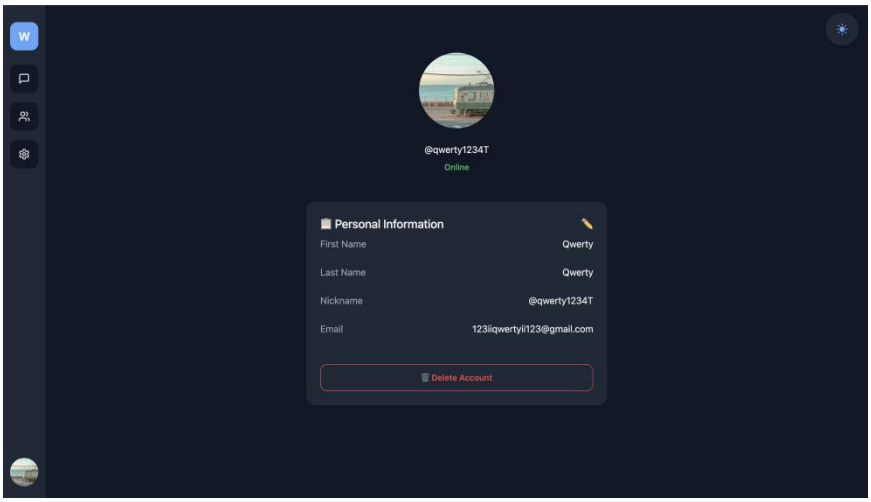

Рисунок 4.24 – Перегляд профілю користувача

При натисканні кнопки редагування, розташованої у правому верхньому куті картки, поля форми переходять у режим введення, що зображено на рисунку 4.25.

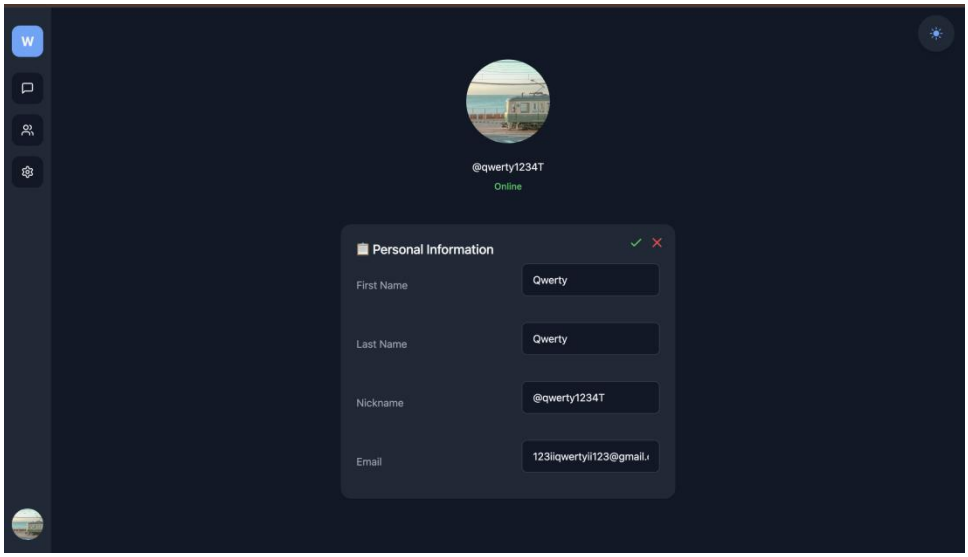

Рисунок 4.25 – Редагування персональних даних користувача

У цьому режимі користувач може змінити ім'я, прізвище, нікнейм або електронну пошту. Після внесення змін доступні дві дії - підтвердити зміни натисканням зеленої галочки або скасувати їх натисканням червоного хрестика, що повертає форму до попередніх значень без збереження. Система виконує клієнтську валідацію введених даних у реальному часі - при некоректному заповненні поле підсвічується червоним кольором з відповідним повідомленням про вимоги до формату, що зображено на рисунку 4.26. Зокрема, поле імені приймає лише літери довжиною від 3 до 25 символів, що унеможливлює збереження некоректних даних до їх виправлення користувачем.

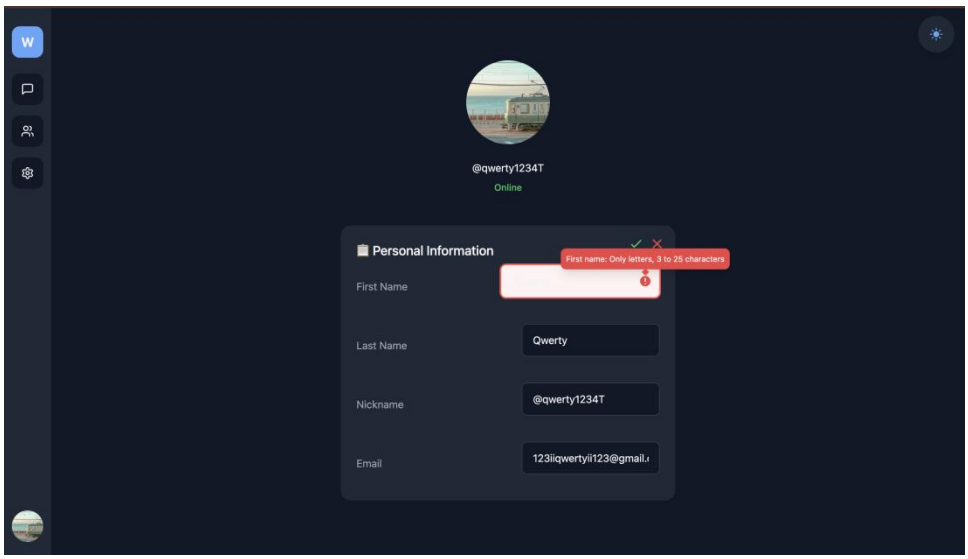

Рисунок 4.26 – Валідація персональних даних користувача

Таким чином, розділ налаштувань та профілю користувача забезпечує повний контроль над персональними даними, зовнішнім виглядом та поведінкою інтерфейсу, що робить систему гнучкою та зручною для широкого кола користувачів з різними уподобаннями та потребами.

4.3 Огляд схеми бази даних

Відповідно до принципу Database per Service мікросервісної архітектури, кожен сервіс системи має власну ізольовану базу даних PostgreSQL, що унеможливлює пряму міжсервісну залежність на рівні даних та дозволяє масштабувати і розгортати кожен сервіс незалежно. Взаємодія з базами даних реалізована через Entity Framework Core з окремим DbContext для кожної схеми (додаток 4, AppDbContext.cs, OtoAppDbContext.cs, GroupAppDbContext.cs).

Перша схема описує структуру бази даних сервісу користувачів і керується через AppDbContext (додаток 4, AppDbContext.cs). Таблиця Users зберігає профільну інформацію: Login та NickName обмежені 80 символами, поле Image - 500 символами. Принципово важливим архітектурним рішенням є зберігання чутливих ідентифікаційних даних виключно у хешованому

вигляді - поля HashLogin, HashEmail та HashNickName обмежені 120 символами і мають унікальні індекси, що одночасно забезпечує унікальність користувачів у системі та унеможливлює відновлення оригінальних значень навіть у разі компрометації бази даних. Таблиця також містить розширений набір полів ASP.NET Identity: PasswordHash, SecurityStamp, ConcurrencyStamp, TwoFactorEnabled, LockoutEnd та AccessFailedCount, що забезпечують повноцінне управління обліковими записами та безпекою автентифікації.

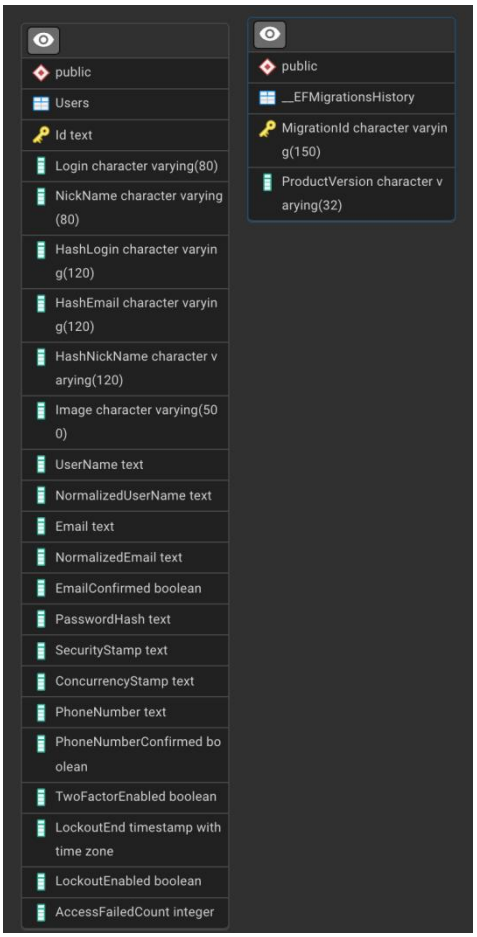

Рисунок 4.4 – Схема бази даних сервісу користувачів

Друга схема описує структуру бази даних сервісу повідомлень для приватних чатів і керується через OtoAppDbContext (додаток 4, OtoAppDbContext.cs). Таблиця UsersMessages зберігає зашифровані повідомлення між двома користувачами: поля Sender та Recipient обмежені 120 символами, Content - 20000 символами. Чутливі ідентифікатори зберігаються у хешованому вигляді через поля SenderHash та RecipientHash

обмежені 120 символами, на яких встановлено індекси для ефективного пошуку повідомлень без розкриття оригінальних нікнеймів. Службові поля IsDeleted, IsEdited, EditDate та ReplyFor забезпечують підтримку редагування, видалення та відповідей на повідомлення, а поле SendTime фіксує мітку часу відправлення. Таблиця Images кешує зображення профілів користувачів - поле NickNameHash обмежене 120 символами має унікальний індекс, а Image - 500 символами.

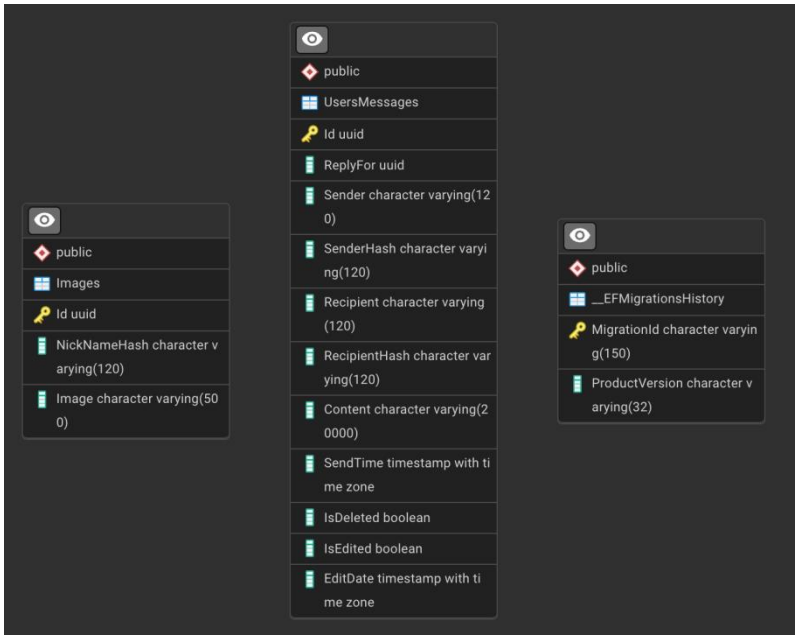

Рисунок 4.5 – Схема бази даних сервісу повідомлень для приватних чатів

Третя схема описує структуру бази даних сервісу повідомлень для групових чатів і керується через GroupAppDbContext (додаток 4, GroupAppDbContext.cs). Таблиця GroupInfos зберігає метадані групи: GroupName обмежене 500 символами, Description - 600, Image - 900, Admin - 80 символами. Чутливі ідентифікатори адміністратора та назви групи зберігаються у хешованому вигляді через поля AdminHash та GroupNameHash з унікальними індексами. Поле RowVersion налаштовано як IsConcurrencyToken, що забезпечує механізм оптимістичного блокування та запобігає конфліктам при одночасному редагуванні групи кількома учасниками. Таблиця GroupMembers містить дані учасників: UserNickName та

UserNickNameHash обмежені 120 символами, Image - 600 символами, а поле JoinedTime фіксує час приєднання до групи. Зв'язок з GroupInfos реалізовано через зовнішній ключ GroupId з каскадним видаленням - при видаленні групи автоматично видаляються всі її учасники. Таблиця GroupMessages зберігає зашифровані повідомлення: Sender та SenderHash обмежені 120 символами, Content - 20000 символами. Зв'язок з GroupInfos реалізовано через зовнішній ключ GroupId, а поля IsDeleted, IsEdited, EditTime та ReplyFor забезпечують підтримку редагування, видалення та відповідей на повідомлення.

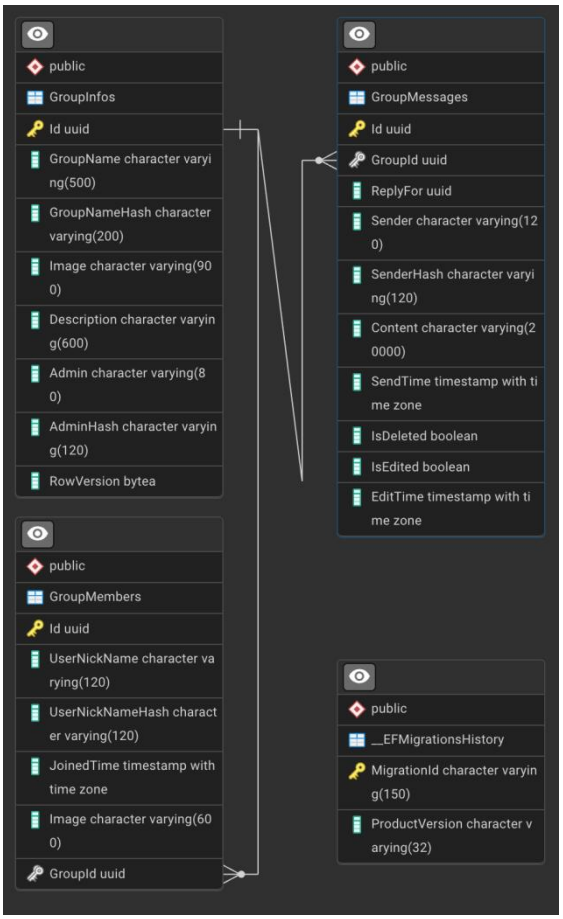

Рисунок 4.6 – Схема бази даних сервісу повідомлень для групових чатів

Четверта схема описує структуру бази даних клієнтської частини системи, реалізованої через Prisma ORM з підключенням до PostgreSQL. На відміну від серверних схем, ця база даних обслуговує протокол Double Ratchet безпосередньо на стороні клієнта. Таблиця User зберігає нікнейм та публічний RSA-ключ користувача з унікальними обмеженнями на обох полях. Таблиця

Message містить зашифровані повідомлення: поля encryptedContent, ephemeralKey та nonce забезпечують зберігання даних у форматі сумісному з протоколом Double Ratchet, а поля messageNumber та previousChainN відстежують стан ланцюжка шифрування. Підтримка редагування та видалення реалізована через поля isEdited, editedAt, deleteType та deletedAt, а відповіді на повідомлення - через самореференційний зв'язок replyToId.

Таблиця MessageKey зберігає сесійні ключі для кожного учасника повідомлення: encryptedKey містить зашифрований ключ, ephemeralPublicKey - ефемерний публічний ключ, chainKeySnapshot - знімок стану ланцюжка, а keyIndex - порядковий номер ключа. Унікальний індекс на парі messageId та userId гарантує що кожен учасник має лише один ключ для кожного повідомлення. Таблиця RatchetState фіксує повний стан ratchet-сесії між двома користувачами: rootKey, sendingChainKey та receivingChainKey описують поточний стан ключових ланцюжків, поля dhRatchetPrivate, dhRatchetPublic та remotePublicKey - стан ключів Діффі-Хеллмана, а sendMessageNumber та recvMessageNumber - лічильники відправлених та отриманих повідомлень. Унікальний індекс на парі userId та contactId гарантує єдиність стану сесії між кожною парою користувачів.

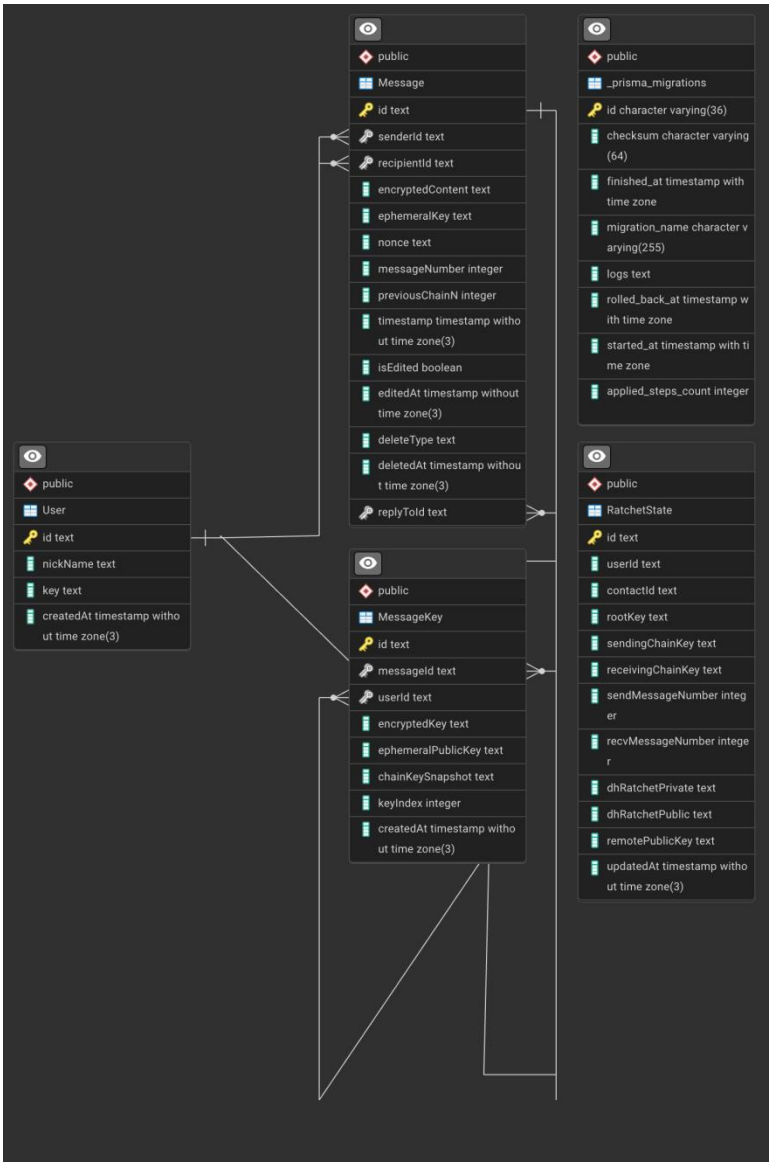

Рисунок 4.7 – Схема бази даних E2EE ключів

Наскрізним принципом організації усіх чотирьох баз даних є зберігання чутливих даних виключно у зашифрованому або хешованому вигляді. Жодна з серверних таблиць не містить повідомлень чи ідентифікаторів користувачів у відкритому вигляді, а клієнтська база даних зберігає криптографічний стан сесій Double Ratchet локально на пристрої користувача. Така архітектура зберігання даних у поєднанні з наскрізним шифруванням унеможливлює відновлення вмісту переписки навіть у разі одночасної компрометації серверної інфраструктури та бази даних.

4.4 Рекомендації щодо розвитку та вдосконалення системи

Розроблена система захищеного обміну повідомленнями є повнофункціональним програмним продуктом, що реалізує всі визначені функціональні та нефункціональні вимоги. Водночас у процесі розробки було визначено ряд напрямків, реалізація яких дозволить суттєво розширити можливості системи та підвищити її готовність до промислового використання. Одним із пріоритетних напрямків є розробка мобільного додатку для платформ iOS та Android. На відміну від поточної веббраузерної реалізації, нативний мобільний додаток забезпечить можливість отримання push-сповіщень про нові повідомлення у фоновому режимі, авторизацію на кількох пристроях одночасно з синхронізацією стану сесії - аналогічно до підходу реалізованого у Signal. Для реалізації крос-платформеного мобільного додатку доцільно розглянути використання фреймворку .NET MAUI або Flutter, що дозволить повторно використати частину існуючої бізнес-логіки.

Важливим напрямком є розгортання системи на хмарній або власній інфраструктурі з урахуванням вимог до масштабованості та відмовостійкості. Для оркестрації мікросервісів рекомендується використання Kubernetes, що забезпечить автоматичне горизонтальне масштабування окремих сервісів залежно від навантаження, автоматичний перезапуск сервісів у разі збою та керування розгортанням нових версій без переривання роботи системи. Для балансування навантаження між екземплярами сервісів доцільно застосувати NGINX як Ingress Controller у кластері Kubernetes, що забезпечить рівномірний розподіл запитів та SSL-термінацію на рівні вхідної точки системи.

Окремим напрямком вдосконалення є розширення функціоналу самої системи: додавання голосових та відеодзвінків із наскрізним шифруванням на основі протоколу WebRTC, реалізація механізму самознищення повідомлень після визначеного проміжку часу, а також підтримка верифікації особи

користувача через цифровий підпис для підвищення рівня довіри між учасниками комунікації.

Перспективним напрямком є також розширення форматів медіаконтенту в межах чатів. Зокрема, доцільним є додавання підтримки голосових повідомлень - запису та відтворення аудіо безпосередньо в інтерфейсі чату, аналогічно до реалізації у WhatsApp та Telegram. Голосові повідомлення перед відправленням мають шифруватись за тими самими механізмами що й текстові, що забезпечить їх конфіденційність у рамках існуючої криптографічної архітектури системи. Додатково можлива реалізація коротких відеоповідомлень з автоматичним стисненням та потоковою передачею, що суттєво розширить засоби комунікації між учасниками без шкоди для рівня захисту даних.

ВИСНОВОК ДО РОЗДІЛУ 4

У розділі 4 було проведено комплексне дослідження та аналіз розробленої системи захищеного обміну повідомленнями WhisperNet, що охопило тестування, опис інтерфейсу користувача, огляд схем баз даних та рекомендації щодо подальшого розвитку.

За результатами автоматизованого тестування серверна частина системи покрита 1329 тестами розподіленими між п'ятьма тестовими проєктами, всі з яких виконались успішно. Клієнтська частина протестована 2307 тестами з досягненням 100% покриття коду за всіма чотирма метриками - інструкціями, гілками, функціями та рядками. Отримані результати підтверджують коректність реалізованої бізнес-логіки та відповідність системи сформульованим функціональним і нефункціональним вимогам.

В рамках інструкції користувача описано чотири ключові сценарії взаємодії з системою. Процедура реєстрації реалізована з обов'язковою генерацією мнемонічної фрази відновлення та створенням резервної копії ключів шифрування, що забезпечує криптографічну незалежність користувача від серверної інфраструктури. Модуль повідомлень підтримує приватні та групові чати з повним набором функцій редагування, видалення та відповідей на повідомлення, а також відправлення файлів різних типів. Розділ налаштувань надає можливість кастомізації теми, мови інтерфейсу та керування персональними даними профілю.

Огляд схем баз даних продемонстрував послідовне застосування принципу Database per Service - кожен мікросервіс має власну ізольовану базу даних PostgreSQL з окремим DbContext. Наскрізним принципом усіх схем є зберігання чутливих даних виключно у хешованому або зашифрованому вигляді, що унеможливлює відновлення персональних даних та вмісту переписки навіть у разі компрометації інфраструктури.

На завершення розділу сформульовано рекомендації щодо подальшого розвитку системи, зокрема розробку нативного мобільного додатку, розгортання на хмарній інфраструктурі з оркестрацією через Kubernetes, а також розширення функціоналу за рахунок голосових та відеодзвінків із наскрізним шифруванням і механізму самознищення повідомлень.

ВИСНОВКИ

У процесі виконання дипломної роботи на тему «Система обміну повідомленнями з шифруванням» створено функціонально повноцінний, безпечний та технічно обґрунтований програмний застосунок для захищеної комунікації між користувачами у приватних та групових чатах. В основі системи лежить поєднання мікросервісної архітектури з сучасними криптографічними протоколами Double Ratchet та ChaCha20-Poly1305, що дозволило забезпечити наскрізне шифрування повідомлень без можливості їх перехоплення чи розшифрування на серверній стороні.

Під час реалізації проєкту було послідовно виконано всі етапи розробки від аналізу предметної області й постановки вимог до системи, до проєктування її архітектури, вибору технологій та безпосередньої реалізації серверної і клієнтської частин. Система побудована на основі чітко визначених функціональних вимог, серед яких: реєстрація та автентифікація користувачів без прив’язки до номеру телефона, наскрізне шифрування приватних та групових повідомлень, підтримка передачі файлів різних типів, а також керування профілем та кастомізація інтерфейсу. Усі ці компоненти були впроваджені відповідно до вимог проєктного завдання. Разом із тим нефункціональні вимоги, зокрема безпека, масштабованість та відмовостійкість, також були враховані у технічній реалізації системи, що дозволяє гарантувати її стабільну роботу в умовах сучасного цифрового середовища.

Система, описана в межах даного дипломного проєкту, передбачає побудову захищеної моделі зберігання та передачі даних, в якій жоден компонент серверної інфраструктури не має доступу до вмісту повідомлень у відкритому вигляді. Чутливі ідентифікаційні дані користувачів зберігаються виключно у хешованому вигляді, криптографічний стан сесій Double Ratchet

зберігається локально на пристрої користувача, а дешифрування повідомлень відбувається виключно на стороні отримувача. Таке рішення сприяє не лише підвищенню рівня конфіденційності, а й забезпеченню стійкості системи до компрометації серверної інфраструктури.

До основних переваг запропонованого застосунку належать: наскрізне шифрування повідомлень з forward secrecy на рівні кожного повідомлення завдяки протоколу Double Ratchet; захист групових чатів через протокол Sender Key з автоматичною ротацією ключів при зміні складу учасників; зберігання чутливих даних виключно у хешованому або зашифрованому вигляді на всіх рівнях системи; мікросервісна архітектура з принципом Database per Service та підтверджена якість реалізації через 100% покриття коду на клієнтській частині та 1329 автоматизованих тестів на серверній.

Застосування сучасних криптографічних протоколів у межах цієї розробки дозволило реалізувати механізм захищеної комунікації, що не потребує довіри до серверної інфраструктури. Запропоноване рішення дозволяє не лише передавати повідомлення у зашифрованому вигляді, а й гарантувати їх конфіденційність навіть у разі компрометації бази даних або серверів, забезпечуючи тим самим повний контроль користувача над власними даними. Технічна реалізація підтверджує доцільність обраного підходу, оскільки система демонструє високу продуктивність, стійкість до зовнішніх загроз і логічну завершеність інтерфейсу користувача.

Таким чином, можна стверджувати, що всі цілі дипломного проєкту повністю досягнуто. Тема дослідження розкрита у повному обсязі, її актуальність доведено практичними результатами реалізації, а запропоноване технічне рішення може бути адаптоване до реальних потреб у сфері захищеної комунікації як у корпоративному, так і у приватному секторі. Отримані результати є підґрунтям для подальших досліджень у напрямі розвитку децентралізованих та криптографічно захищених систем комунікації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Telegram [Електронний ресурс] – Режим доступу до ресурсу: https://telegram.org.
2. Telegram Security [Електронний ресурс] – Режим доступу до ресурсу: https://telegram.org/faq#security.
3. Telegram Secret Chats [Електронний ресурс] – Режим доступу до ресурсу: https://telegram.org/faq#secret-chats.
4. Telegram Open Source Policy [Електронний ресурс] – Режим доступу: https://telegram.org/apps#source-code.
5. Cybernews Signal founder roasts Telegram, says there's nothing private about it [Електронний ресурс]. – Режим доступу: https://cybernews.com/security/signal-founder-telegram-privacy-pavel-durov/.
6. Telegram Working with Different Data Centers [Електронний ресурс] – Режим доступу: https://core.telegram.org/api/datacenter.
7. Coxxs Mysteries of Telegram DC [Електронний ресурс] – Режим доступу: https://dev.moe/en/3025.
8. Telegram MTProto Transports [Електронний ресурс] – Режим доступу до ресурсу: https://core.telegram.org/mtproto/mtproto-transports.
9. Getstream.io How WhatsApp Works - Architecture Deep Dive on 100 Billion Messages [Електронний ресурс] – Режим доступу до ресурсу: https://getstream.io/blog/whatsapp-works.
10. WhatsApp Security Whitepaper - Meta Platforms [Електронний ресурс] – Режим доступу до ресурсу: https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf.
11. WhatsApp End-to-End Encrypted Backups on WhatsApp - Meta Platforms, 2021 [Електронний ресурс] – Режим доступу до ресурсу: https://faq.whatsapp.com/490592613091019.

12. CNBC Facebook is planning to move WhatsApp off IBM's public cloud. - 2017 [Електронний ресурс] – Режим доступу до ресурсу: https://www.cnbc.com/2017/06/07/facebook-planning-to-move-whatsapp-off-ibms-public-cloud.html.
13. Dgtl Infra Meta's Data Center Locations for Facebook and Instagram - 2024 [Електронний ресурс] – Режим доступу до ресурсу: https://dgtlinfra.com/meta-data-center-locations-facebook.
14. DEV Community The WhatsApp-First Paradigm and the Revolution of Conversational Systems - 2026 [Електронний ресурс] – Режим доступу до ресурсу: https://dev.to/fullagenticstack/the-whatsapp-first-paradigm-and-the-revolution-of-conversational-systems-an-analysis-of-the-1g5l.
15. CometChat Understanding WhatsApp's Architecture & System Design - 2021 [Електронний ресур] – Режим доступу до ресурсу: https://www.cometchat.com/blog/whatsapps-architecture-and-system-design.
16. ByteByteGo How WhatsApp Handles 40 Billion Messages Per Day - 2025 [Електронний ресурс] – Режим доступу до ресурсу: https://blog.bytebytego.com/p/how-whatsapp-handles-40-billion-messages.
17. WhatsApp Security Advisories - Meta Platforms, 2025 [Електронний ресурс] – Режим доступу до ресурсу: https://www.whatsapp.com/security/advisories/2025.
18. Signal Foundation Privacy is Priceless, but Signal is Expensive - Signal Blog, 2023 [Електронний ресурс] – Режим доступу до ресурсу: https://signal.org/blog/signal-is-expensive.
19. Wikipedia Signal (software) - 2025 [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Signal_(software).
20. Signal Foundation Signal Protocol Documentation - 2024 [Електронний ресурс] – Режим доступу до ресурсу: https://signal.org/docs.

21. Sathyanarayanan S Inside the Signal Protocol's Security Architecture: A Technical Deep Dive - profincognito.me, 2024 [Електорнний ресурс] – Режим доступу до ресурсу: https://profincognito.me/blog/security/signal-security-architecture.
22. Data Center Dynamics Encrypted messaging service Signal goes down, as millions drop WhatsApp - 2021 [Електронний ресурс] – Режим доступу до ресурсу: https://www.datacenterdynamics.com/en/news/encrypted-messaging-service-signal-goes-down-millions-drop-whatsapp.
23. Signal Foundation Technology preview: Private contact discovery for Signal - Signal Blog, 2017 [Електронний ресурс] – Режим доступу до ресурсу: https://signal.org/blog/private-contact-discovery.
24. Signal Foundation Technology Deep Dive: Building a Faster ORAM Layer for Enclaves - Signal Blog, 2022 [Електронний ресурс] – Режим доступу до ресурсу: https://signal.org/blog/building-faster-oram.
25. signalapp Signal Server Source Code - GitHub, 2024 [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/signalapp/Signal-Server.
26. Signal Foundation Grand Jury Subpoena for Signal User Data, Eastern District of Virginia - Signal Blog, 2016 [Електронний ресурс] – Режим доступу до ресурсу: https://signal.org/bigbrother/eastern-virginia-grand-jury.
27. Medium ApptasticCoder Have you heard Chacha20-Poly1305? [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/apptastic-coder/have-you-heard-chacha20-poly1305-580586e47347.
28. Yashraj Singh Tomar ChaCha20: A Modern Stream Cipher Explained [Електронний ресурс] – Режим досутпу до ресурсу: https://www.linkedin.com/pulse/chacha20-modern-stream-cipher-explained-yashraj-singh-tomar-rs7qe/.

29. Toma A. et al. ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications [Електронний ресурс] – Режим доступу: https://ieeexplore.ieee.org/document/7927078/.
30. SECURITY Cryptographic Algorithms [Електронний ресурс] – Режим доступу: https://docs.phase.dev/security/cryptography.
31. RSA Algorithm in Cryptography [Електронний ресусрс] – Режим доступу: https://www.geeksforgeeks.org/computer-networks/rsa-algorithm-cryptography/.
32. RSA Encryption: Beginner's Guide [Електронний ресур] – Режим доступу: https://medium.com/@Espress0/rsa-encryption-beginners-guide-6caee3e8cd80.
33. What is RSA Encryption? [Електронний ресурс] – Режим доступу: https://www.cyberark.com/what-is/rsa-encryption/.
34. Ahsan Mubariz End-to-End Encryption (E2EE) [Електронний ресурс] – Режим доступу: https://medium.com/@ahsanmubariz/end-to-end-encryption-e2ee-a-deep-dive-with-simple-example-in-javascript-d97b1b880d4d.
35. What is end-to-end encryption (E2EE)? [Електронний ресурс] – Режим доступу: https://www.ibm.com/think/topics/end-to-end-encryption.
36. End-to-End Encryption or Gateway-to-Gateway Encryption? [Електронний ресурс] – Режим доступу: https://www.nospamproxy.de/en/end-to-end-encryption-gateway-to-gateway-encryption/.
37. What is E2EE (End to End Encryption)? [Електронний ресурс] – Режим доступу: https://www.geeksforgeeks.org/computer-networks/what-is-e2eeend-to-end-encryption/.
38. What is end-to-end encryption, and how does it work? [Електронний ресурс] – Режим доступу: https://proton.me/learn/encryption/types-of-encryption/what-is-end-to-end.

39. Introduction to C# [Електронний ресурс] – Режим доступу: https://www.w3schools.com/cs/cs_intro.php.
40. A tour of the C# language [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/overview.
41. What is C# (C Sharp) programming language? [Електронний ресурс] – Режим доступу: https://hicronsoftware.com/blog/what-is-c-sharp-programming-language.
42. Hejlsberg A., Torgersen M., Wiltamuth S., Golde P. The C# Programming Language. 4th ed. — Upper Saddle River : Addison-Wesley, 2010. — 864 p.
43. Troelsen A., Japikse P. Pro C# 7: With .NET and .NET Core. — New York : Apress, 2017. — 1372 p.
44. Albahari J., Albahari B. C# 7.0 in a Nutshell: The Definitive Reference. — Sebastopol : O'Reilly Media, 2017. — 1088 p.
45. What's new in .NET 8 [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-8/overview.
46. Announcing .NET 8 [Електронний ресурс] – Режим доступу: https://devblogs.microsoft.com/dotnet/announcing-dotnet-8/.
47. Getting started with .NET 8.0 on RHEL 8 [Електронний ресурс] – Режим доступу: https://docs.redhat.com/documentation/net/8.0.
48. SignalR Microsoft Learn [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/aspnet/signalr/.
49. Real-time ASP.NET with SignalR .NET [Електронний ресурс] – Режим доступу: https://dotnet.microsoft.com/en-us/apps/aspnet/signalr.
50. Updating client application data from the server by using SignalR DWKit Documentation [Електронний ресурс] – Режим доступу: https://dwkit.com/documentation/dwkit-solution/signalr.
51. RabbitMQ Documentation [Електронний ресурс] – Режим доступу: https://www.rabbitmq.com/docs.

52. RabbitMQ Linuxfabrik Documentation [Електронний ресурс] – Режим доступу: https://docs.linuxfabrik.ch/software/rabbitmq.html.
53. About PostgreSQL [Електронний ресурс] – Режим доступу: https://www.postgresql.org/about/.
54. What is PostgreSQL? [Електронний ресурс] – Режим доступу: https://www.ibm.com/think/topics/postgresql.
55. Integration Tests vs Unit Tests: What Are They And Which One to Use? [Електронний ресурс] – Режим доступу: https://testrigor.com/blog/integration-tests-vs-unit-tests-what-are-they-and-which-one-to-use/.
56. Difference between Unit Testing and Integration Testing [Електронний ресурс] – Режим доступу: https://www.geeksforgeeks.org/software-engineering/difference-between-unit-testing-and-integration-testing/.
57. xUnit.net [Електронний ресурс] – Режим доступу: https://xunit.net/.
58. Unit Testing: Moq Framework [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/shows/visual-studio-toolbox/unit-testing-moq-framework.
59. Integration tests in ASP.NET Core [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/test/integration-tests.
60. Testcontainers for .NET [Електронний ресурс] – Режим доступу: https://dotnet.testcontainers.org/.
61. Test suites with Jasmine [Електронний ресурс] – Режим доступу: https://testing-angular.com/test-suites-with-jasmine/.
62. Jasmine & Karma [Електронний ресурс] – Режим доступу: https://codecraft.tv/courses/angular/unit-testing/jasmine-and-karma/.
63. What is Angular? [Електронний ресурс] – Режим доступу: https://angular.dev/overview.
64. What is Angular in Web Development? [Електронний ресурс] – Режим доступу: https://hicronsoftware.com/blog/what-is-angular/.

65. Angular Introduction [Електронний ресурс] – Режим доступу: https://www.w3schools.com/angular/angular_intro.asp.
66. Microservices Architecture Style [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices.
67. Microservices Martin Fowler [Електронний ресурс] – Режим доступу: https://www.martinfowler.com/articles/microservices.html.
68. Rudraksh Nanavaty Clean Architecture [Електронний ресурс] – Режим доступу: https://medium.com/@rudrakshnanavaty/clean-architecture-7c1b3b4cb181.
69. FSD [Електронний ресурс] – Режим доступу: https://fsd.how/docs/get-started/overview/.
70. Репозиторій серверної частини [Електронний ресурс] – Режим доступу: https://github.com/Namelesa/MessagingSystem.
71. Репозиторій клієнтської частини [Електронний ресурс] – Режим доступу: https://github.com/Namelesa/MessagingSystemFront.

Додаток 1

Система обміну повідомленнями з шифруванням

Структурна схема системи

ІАЛЦ.467100.004 Д1

Аркушів 1

Київ 2026 р

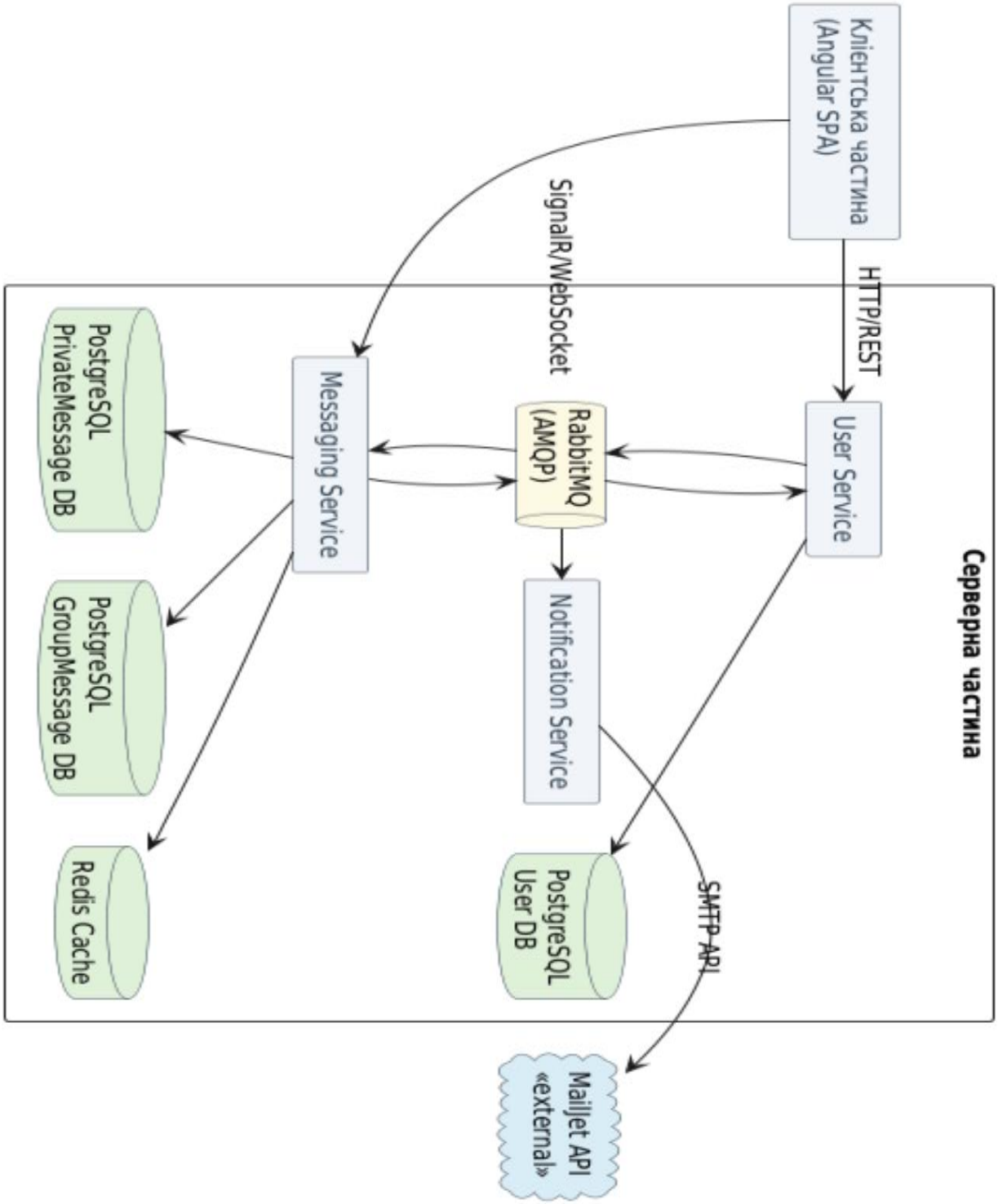

Рисунок Д1.1 – Структурна схема системи

					ІАЛЦ.467100.004 Д1		
		№ докум.	Підпис	Дата			
Розробив		Білик М. Є.			**Система обміну повідомленнями з шифруванням** Структурна схема системи	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Волокита А. М.					
						КПІ ім. Ігоря Сікорського, ФІОТ, ІО-22	
Н. Контр.		Коренко Д. В.					
Затвердив							

Додаток 2

Система обміну повідомленнями з шифруванням

Функціональна схема (діаграма класів)

ІАЛЦ.467100.005 Д2

Аркушів 3

Київ 2026 р

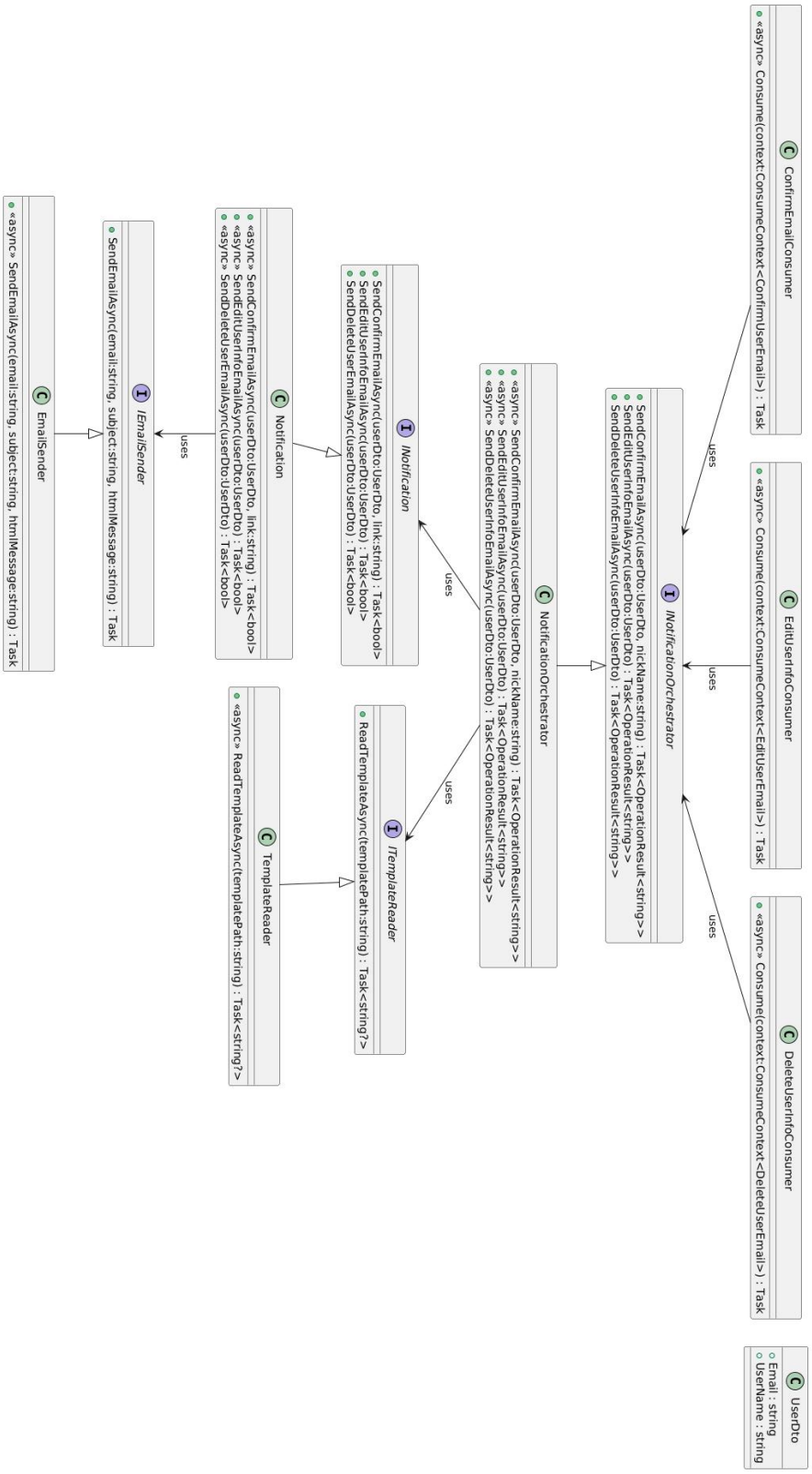

Рисунок Д2.1 – Діаграма класів сервісу Notification

					ІАЛЦ.467100.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Білик М. Є.			**Система обміну повідомленнями з шифруванням** Функціональна схема (діаграма класів)	Літ. / Аркуш 1 / Аркушів 3
Перевірив		Волокита А. М.				
Н. Контр.		Коренко Д. В.				КПІ ім. Ігоря Сікорського, ФІОТ, ІО-22
Затвердив						

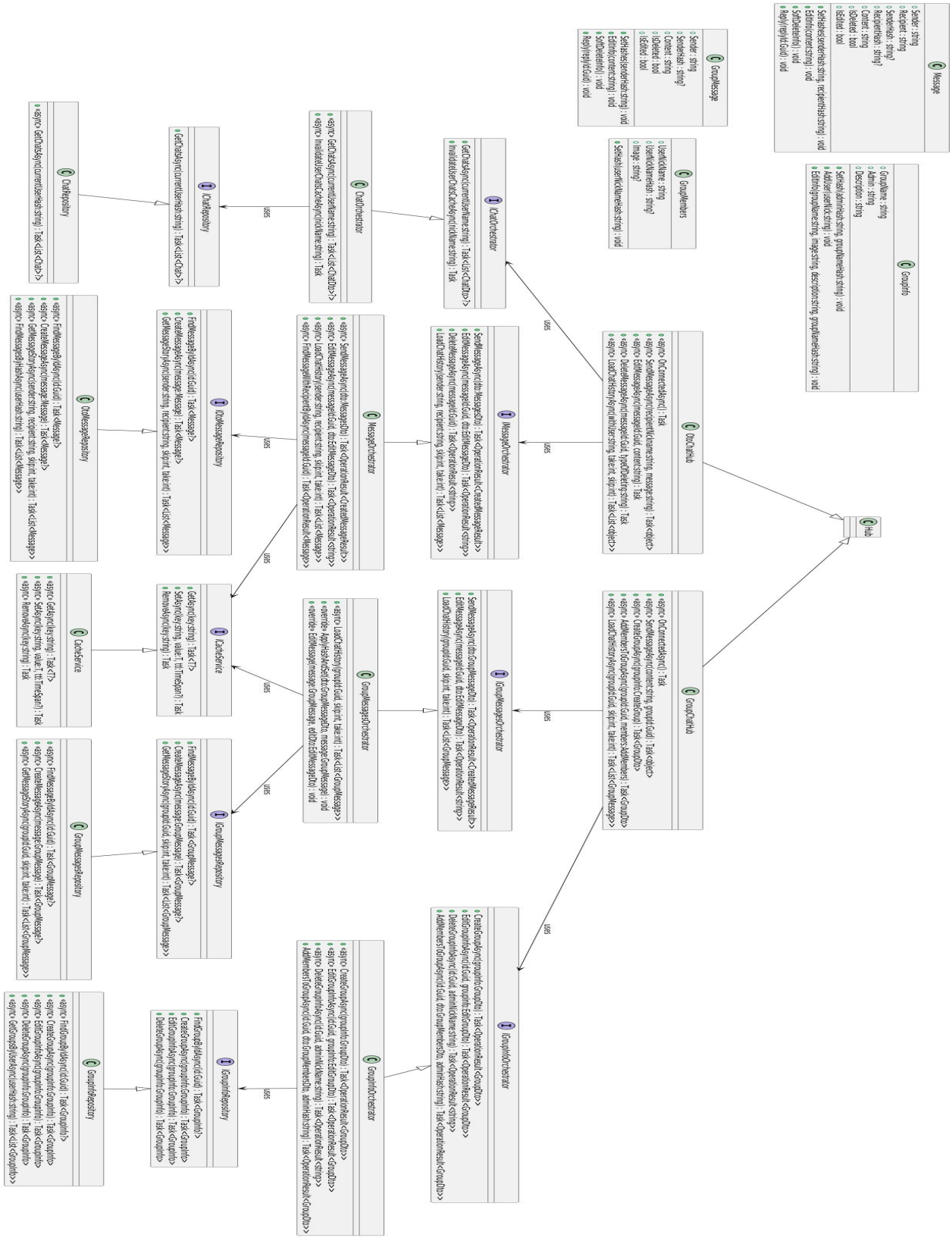

Рисунок Д2.2 – Діаграма класів сервісу Messagin

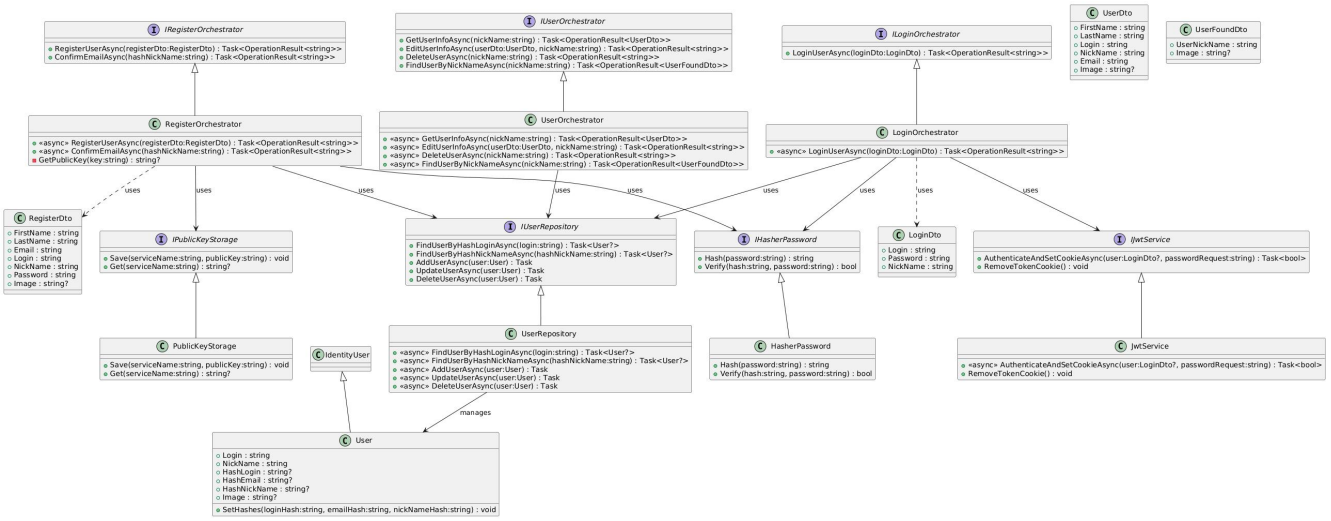

Рисунок Д2.3 — Діаграма класів сервісу User

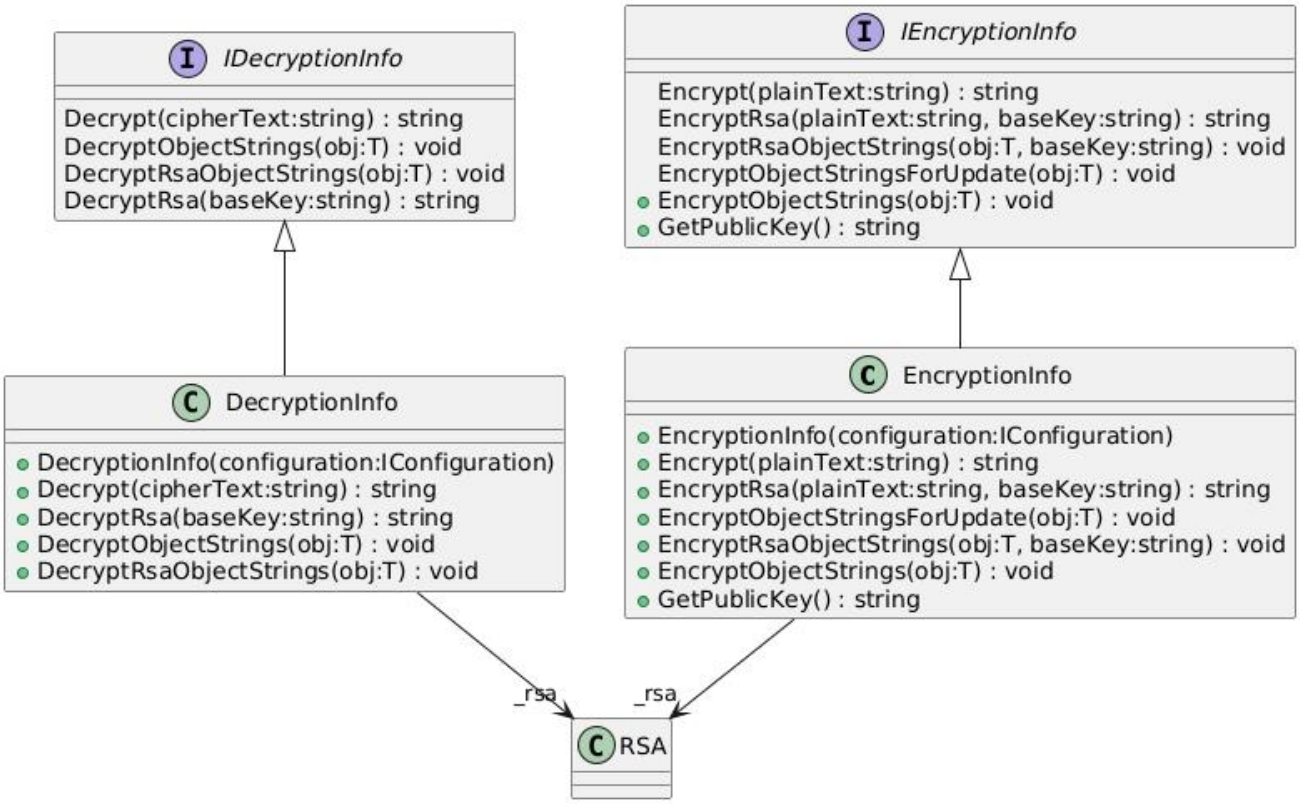

Рисунок Д2.4 – Діаграма класів модуля шифрування та дешифрування

Додаток 3

Система обміну повідомленнями з шифруванням

Алгоритм дій програмного забезпечення

ІАЛЦ.467100.006 Д3

Аркушів 1

Київ 2026 р

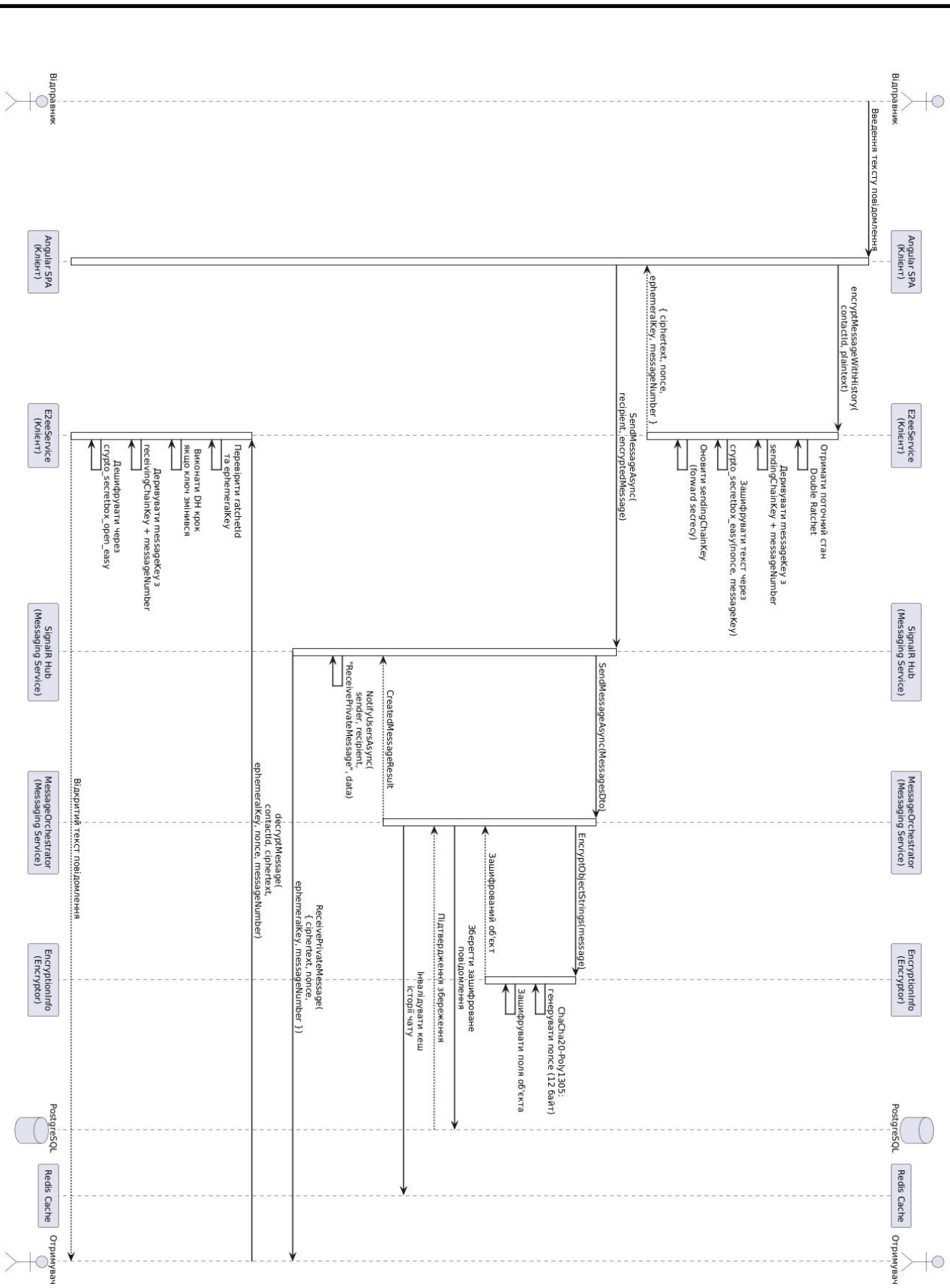

Рисунок Д3.1 – Алгоритм дії програмного забезпечення

					ІАЛЦ.467100.006 Д3	
		№ докум.	Підпис	Дата		
Розробив		Білик М. Є.			**Система обміну повідомленнями з шифруванням** Алгоритм дій програмного забезпечення	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Волокита А. М.				
						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-22
Н. Контр.		Коренко Д. В.				
Затвердив						

Додаток 4

Система обміну повідомленнями з шифруванням

Текст програмного коду основних компонентів системи

ІАЛЦ.467100.007 Д4

Аркушів 42

Київ 2026

Encryptor.cs

```
using System.Security.Cryptography;
using System.Text;
using Encryptor.Publisher;
using Microsoft.Extensions.Configuration;
using Org.BouncyCastle.Crypto.Parameters;
using ChaCha20Poly1305 = Org.BouncyCastle.Crypto.Modes.ChaCha20Poly1305;

namespace Encryptor.Encryption;

public class EncryptionInfo : IEncryptionInfo
{
    private readonly byte[] _key;
    private readonly RSA _rsa;

    public EncryptionInfo(IConfiguration configuration)
    {
        var key = configuration["Encryption:ChaChaKey"];
        if (string.IsNullOrWhiteSpace(key))
            throw new InvalidOperationException("Encryption key not configured");

        _key = SHA256.HashData(Encoding.UTF8.GetBytes(key));

        var publicPath = Path.Combine(AppContext.BaseDirectory, "Keys", "public.key");
        var privatePath = Path.Combine(AppContext.BaseDirectory, "Keys", "private.key");

        if (!File.Exists(publicPath) || !File.Exists(privatePath))
        {
            var generatedKeys = RsaKeyPair.Generate();
            Directory.CreateDirectory(Path.GetDirectoryName(publicPath) ?? throw new InvalidOperationException());
            RsaKeyPair.SaveToFiles(generatedKeys, publicPath, privatePath);
        }

        var rsaKeys = RsaKeyPair.LoadFromFiles(publicPath, privatePath);

        _rsa = RSA.Create();
        _rsa.ImportRSAPrivateKey(Convert.FromBase64String(rsaKeys.PrivateKey), out _);
    }
    public string Encrypt(string plainText)
    {
        var nonce = RandomNumberGenerator.GetBytes(12);
        var plaintextBytes = Encoding.UTF8.GetBytes(plainText);

        var cipher = new ChaCha20Poly1305();
        var parameters = new AeadParameters(new KeyParameter(_key), 128, nonce, null);
        cipher.Init(true, parameters);

        var output = new byte[cipher.GetOutputSize(plaintextBytes.Length)];
        var len = cipher.ProcessBytes(plaintextBytes, 0, plaintextBytes.Length, output, 0);
        cipher.DoFinal(output, len);

        var result = new byte[nonce.Length + output.Length];
        Buffer.BlockCopy(nonce, 0, result, 0, nonce.Length);
        Buffer.BlockCopy(output, 0, result, nonce.Length, output.Length);

        return Convert.ToBase64String(result)
```

```
public string EncryptRsa(string plainText, string baseKey)
{
    using var rsa = RSA.Create();
    rsa.ImportRSAPublicKey(Convert.FromBase64String(baseKey), out _);
    var encrypted = rsa.Encrypt(Encoding.UTF8.GetBytes(plainText), RSAEncryptionPadding.OaepSHA256);
    return Convert.ToBase64String(encrypted);
}
public void EncryptObjectStringsForUpdate<T>(T obj)
{
    var excludedProps = new[] { "Id", "HashNickName", "HashLogin", "PasswordHash", "HashEmail" };

    var props = typeof(T).GetProperties()
        .Where(p =>
            p is { CanRead: true, CanWrite: true } &&
            p.PropertyType == typeof(string) &&
            !excludedProps.Contains(p.Name));

    foreach (var prop in props)
    {
        var value = prop.GetValue(obj) as string;
        if (!string.IsNullOrEmpty(value))
        {
            prop.SetValue(obj, Encrypt(value));
        }
    }
}
public void EncryptRsaObjectStrings<T>(T obj, string baseKey)
{
    var excludedProps = new[] { "NickName" };
    var props = typeof(T).GetProperties()
        .Where(p =>
            p is { CanRead: true, CanWrite: true } &&
            p.PropertyType == typeof(string) &&
            !excludedProps.Contains(p.Name));

    foreach (var prop in props)
    {
        var value = prop.GetValue(obj) as string;
        if (!string.IsNullOrEmpty(value))
        {
            prop.SetValue(obj, EncryptRsa(value, baseKey));
        }
    }
}
public void EncryptObjectStrings<T>(T obj)
{
    var excludedProps = new[]
    {
        "HashLogin", "HashEmail", "HashNickName", "SenderHash", "RecipientHash", "AdminHash"
        , "GroupNameHash", "FileOriginalHash"
    };

    var props = typeof(T).GetProperties()
        .Where(p =>
            p is { CanRead: true, CanWrite: true } &&
            p.PropertyType == typeof(string) &&
            !excludedProps.Contains(p.Name));
```

```
        foreach (var prop in props)
        {
            var value = prop.GetValue(obj) as string;
            if (!string.IsNullOrEmpty(value))
            {
                prop.SetValue(obj, Encrypt(value));
            }
        }
    }
    public string GetPublicKey()
    {
        var publicKeyPath = Path.Combine(AppContext.BaseDirectory, "Keys", "public.key");

        if (!File.Exists(publicKeyPath))
        {
            throw new FileNotFoundException("Public key not found");
        }

        return File.ReadAllText(publicKeyPath);
    }
}
```

Decryptor.cs

```
using System.Security.Cryptography;
using System.Text;
using Encryptor.Publisher;
using Microsoft.Extensions.Configuration;
using Org.BouncyCastle.Crypto.Parameters;
using ChaCha20Poly1305 = Org.BouncyCastle.Crypto.Modes.ChaCha20Poly1305;

namespace Encryptor.Decryption;

public class DecryptionInfo : IDecryptionInfo
{
    private readonly byte[] _key;
    private readonly RSA _rsa;

    public DecryptionInfo(IConfiguration configuration)
    {
        var key = configuration["Encryption:ChaChaKey"];
        if (string.IsNullOrWhiteSpace(key))
            throw new InvalidOperationException("Encryption key not configured");

        _key = SHA256.HashData(Encoding.UTF8.GetBytes(key));

        var publicPath = Path.Combine(AppContext.BaseDirectory, "Keys", "public.key");
        var privatePath = Path.Combine(AppContext.BaseDirectory, "Keys", "private.key");

        if (!File.Exists(publicPath) || !File.Exists(privatePath))
        {
            var generatedKeys = RsaKeyPair.Generate();
            Directory.CreateDirectory(Path.GetDirectoryName(publicPath) ?? throw new InvalidOperationException());
            RsaKeyPair.SaveToFiles(generatedKeys, publicPath, privatePath);
        }
```

```
var rsaKeys = RsaKeyPair.LoadFromFiles(publicPath, privatePath);

        _rsa = RSA.Create();
        _rsa.ImportRSAPrivateKey(Convert.FromBase64String(rsaKeys.PrivateKey), out _);
    }
    public string Decrypt(string cipherText)
    {
        var input = Convert.FromBase64String(cipherText);
        var nonce = input[..12];
        var ciphertextBytes = input[12..];

        var cipher = new ChaCha20Poly1305();
        var parameters = new AeadParameters(new KeyParameter(_key), 128, nonce, null);
        cipher.Init(false, parameters);

        var output = new byte[cipher.GetOutputSize(ciphertextBytes.Length)];
        var len = cipher.ProcessBytes(ciphertextBytes, 0, ciphertextBytes.Length, output, 0);
        cipher.DoFinal(output, len);

        return Encoding.UTF8.GetString(output);
    }
    public string DecryptRsa(string baseKey)
    {
        var encryptedBytes = Convert.FromBase64String(baseKey);
        var decrypted = _rsa.Decrypt(encryptedBytes, RSAEncryptionPadding.OaepSHA256);
        return Encoding.UTF8.GetString(decrypted);
    }
    public void DecryptObjectStrings<T>(T obj)
    {
        var excludedProps = new[]
        {
            "SenderHash", "RecipientHash", "GroupNameHash", "AdminHash", "UserNickNameHash"
            ,"HashEmail", "HashLogin", "HashNickName", "RowVersion"
        };

        var props = typeof(T).GetProperties()
            .Where(p =>
                p is { CanRead: true, CanWrite: true } &&
                p.PropertyType == typeof(string) &&
                !excludedProps.Contains(p.Name));

        foreach (var prop in props)
        {
            var value = prop.GetValue(obj) as string;
            if (!string.IsNullOrEmpty(value))
            {
                prop.SetValue(obj, Decrypt(value));
            }
        }
    }
```

```
public void DecryptRsaObjectStrings<T>(T obj)
  {
     var props = typeof(T).GetProperties()
        .Where(p =>
           p is { CanRead: true, CanWrite: true } &&
           p.PropertyType == typeof(string));

     foreach (var prop in props)
     {
        var value = prop.GetValue(obj) as string;
        if (string.IsNullOrEmpty(value)) continue;
        try
        {
           var decrypted = DecryptRsa(value);
           prop.SetValue(obj, decrypted);
        }
        catch(Exception e)
        {
           Console.WriteLine(e);
        }
     }
  }
}
```

RegisterOrchestrator.cs

```
using AutoMapper;
using Encryptor.Encryption;
using FluentValidation;
using MassTransit;
using MessagingSystem.SendingModels.UserMessaging.Add;
using MessagingSystem.SendingModels.UserNotification;
using MessagingSystem.Services.User.Application.Auth.Register.Dto;
using MessagingSystem.Services.User.Core.User;
using MessagingSystem.Services.User.Infrastructure.HasherInfo;
using MessagingSystem.Services.User.Infrastructure.Keys;
using MessagingSystem.Services.User.Infrastructure.PasswordHasher;

namespace MessagingSystem.Services.User.Application.Auth.Register;

public class RegisterOrchestrator(
   IUserRepository userRepository,
   IMapper mapper,
   IValidator<RegisterDto> validator,
   IHasherPassword hasherPassword,
   IEncryptionInfo encryptInfo,
   IHasher hasher,
   IPublishEndpoint publishEndpoint,
   IPublicKeyStorage publicKeyStorage,
   IRequestClient<AddUserRequest> client) : IRegisterOrchestrator
{
   public async Task<OperationResult<string>> RegisterUserAsync(RegisterDto registerDto)
   {
      var publicKeyNotification = GetPublicKey("Notification");
      var publicKeyMessaging = GetPublicKey("Messaging");

      if (publicKeyNotification == null || publicKeyMessaging == null)
         return OperationResult<string>.Fail("Public key for Notification or Messaging service not found");
```

```
        var validationResult = await validator.ValidateAsync(registerDto);
        if (!validationResult.IsValid)
            return OperationResult<string>.Fail(string.Join("; ", validationResult.Errors));

        registerDto.Password = hasherPassword.Hash(registerDto.Password);

        var hashLogin = hasher.Hash(registerDto.Login);
        var hashEmail = hasher.Hash(registerDto.Email);
        var hashNickName = hasher.Hash(registerDto.NickName);

        var user = mapper.Map<Core.User.User>(registerDto);
        user.SetHashes(hashLogin, hashEmail, hashNickName);

        encryptInfo.EncryptObjectStrings(user);

        try
        {
            if(user.HashNickName == null || user.Image == null || user.UserName == null || user.Email == null)
                return OperationResult<string>.Fail("User can not have null properties");

            var existingUserResponse = await client.GetResponse<AddUserResponse>(
                new AddUserRequest(encryptInfo.EncryptRsa(user.HashNickName, publicKeyMessaging),
                    encryptInfo.EncryptRsa(user.Image, publicKeyMessaging)));

            if (!existingUserResponse.Message.Success)
                return OperationResult<string>.Fail("User already exists");

            await userRepository.AddUserAsync(user);

            var confirmUserEmail = new ConfirmUserEmail(user.UserName, user.Email,
Uri.EscapeDataString(hashNickName));
            encryptInfo.EncryptRsaObjectStrings(confirmUserEmail, publicKeyNotification);

            await publishEndpoint.Publish(confirmUserEmail);

            return OperationResult<string>.Ok("User registered and need to confirm email");
        }
        catch (Exception e)
        {
            return OperationResult<string>.Fail($"User can not be added {e.Message}");
        }
    }

    public async Task<OperationResult<string>> ConfirmEmailAsync(string hashNickName)
    {
        var decodedHash = Uri.UnescapeDataString(hashNickName);
        var user = await userRepository.FindUserByHashNickNameAsync(decodedHash);

        if (user == null)
            return OperationResult<string>.Fail("User not found");

        if(user.EmailConfirmed)
            return OperationResult<string>.Fail("Email already confirmed");

        user.EmailConfirmed = true;
```

```
try
        {
            await userRepository.UpdateUserAsync(user);
            return OperationResult<string>.Ok(user.HashNickName);
        }
        catch (Exception e)
        {
            return OperationResult<string>.Fail($"Error {e}");
        }
    }
    private string? GetPublicKey(string key)
        => publicKeyStorage.Get(key);
}
```

LoginOrchestrator.cs

```
using Encryptor.Decryption;
using FluentValidation;
using MessagingSystem.Services.User.Application.Auth.Login.Dto;
using MessagingSystem.Services.User.Core.User;
using MessagingSystem.Services.User.Infrastructure.HasherInfo;
using MessagingSystem.Services.User.Infrastructure.Jwt;

namespace MessagingSystem.Services.User.Application.Auth.Login;

public class LoginOrchestrator(
    IUserRepository userRepository,
    IValidator<LoginDto> validator,
    IDecryptionInfo decryptionInfo,
    IJwtService jwtService,
    IHasher hasher) : ILoginOrchestrator
{
    public async Task<OperationResult<string>> LoginUserAsync(LoginDto loginDto)
    {
        var validationResult = await validator.ValidateAsync(loginDto);
        if (!validationResult.IsValid)
            return OperationResult<string>.Fail(string.Join("; ", validationResult.Errors));

        var user = await userRepository.FindUserByHashLoginAsync(hasher.Hash(loginDto.Login));
        if (user == null)
            return OperationResult<string>.Fail("User not found");

        if(!user.EmailConfirmed) return OperationResult<string>.Fail("Please confirm email");

        if (user.HashNickName != hasher.Hash(loginDto.NickName))
            return OperationResult<string>.Fail("Input your real nick name");

        var res = user.PasswordHash != null &&
                  await                                    jwtService.AuthenticateAndSetCookieAsync(loginDto,
decryptionInfo.Decrypt(user.PasswordHash));

        return OperationResult<string>.Ok(res.ToString());
    }
}
```

JwtService.cs

```
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
using Encryptor.Encryption;
using MessagingSystem.Services.User.Application.Auth.Login.Dto;
using Microsoft.AspNetCore.Identity;
using Microsoft.IdentityModel.Tokens;

namespace MessagingSystem.Services.User.Infrastructure.Jwt;

public class JwtService(
    IConfiguration config,
    ILogger<JwtService> logger,
    IEncryptionInfo encryptInfo,
    IHttpContextAccessor httpContextAccessor) : IJwtService
{
    private readonly IConfiguration _config = config
                                      ?? throw new ArgumentNullException(nameof(config));
    private readonly ILogger<JwtService> _logger = logger
                                        ?? throw new ArgumentNullException(nameof(logger));
    private readonly IEncryptionInfo _encryptInfo = encryptInfo
                                       ?? throw new ArgumentNullException(nameof(encryptInfo));
    private readonly IHttpContextAccessor _httpContextAccessor = httpContextAccessor
                                                 ?? throw new ArgumentNullException(nameof(httpContextAccessor));
    private readonly PasswordHasher<LoginDto> _passwordHasher = new();

    private async Task<string?> AuthenticateAsync(LoginDto? user, string passwordRequest)
    {
        if (!ValidateUserCredentials(user, passwordRequest))
        {
            _logger.LogWarning("Authentication failed: invalid credentials for user {UserLogin}", user?.Login);
            return null;
        }

        try
        {
            var token = _encryptInfo.Encrypt(GenerateJwtToken(user));
            return await Task.FromResult(token);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error occurred while generating JWT token for user {UserLogin}", user?.Login);
            return null;
        }
    }
```

```
public async Task<bool> AuthenticateAndSetCookieAsync(LoginDto? user, string passwordRequest)
{
    var token = await AuthenticateAsync(user, passwordRequest);
    if (token == null)
        return false;

    var cookieOptions = new CookieOptions
    {
        HttpOnly = true,
        Secure = true,
        SameSite = SameSiteMode.None,
        Expires = DateTimeOffset.UtcNow.AddMinutes(_config.GetValue<int>("JWTConfig:TokenValidityMinutes"))
    };

    _httpContextAccessor.HttpContext?.Response.Cookies.Append("access_token", token, cookieOptions);
    return true;
}

public void RemoveTokenCookie()
{
    _httpContextAccessor.HttpContext?.Response.Cookies.Delete("access_token");
}

private bool ValidateUserCredentials(LoginDto? user, string passwordRequest)
{
    if (user?.Password == null)
        return false;

    var passwordVerificationResult = _passwordHasher.VerifyHashedPassword(user, passwordRequest,
user.Password);
    return passwordVerificationResult == PasswordVerificationResult.Success;
}

private string GenerateJwtToken(LoginDto? user)
{
    ArgumentNullException.ThrowIfNull(user);

    var issuer = _config["JWTConfig:Issuer"];
    var audience = _config["JWTConfig:Audience"];
    var key = _config["JWTConfig:Key"];
    var tokenMin = _config.GetValue<int>("JWTConfig:TokenValidityMinutes");

    if (string.IsNullOrWhiteSpace(key))
        throw new InvalidOperationException("JWT key is not configured properly.");

    var issuedAt = DateTime.UtcNow;
    var expiresAt = issuedAt.AddMinutes(tokenMin);
    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(key));

    var claims = new List<Claim>
    {
        new (JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
        new (JwtRegisteredClaimNames.Iat, EpochTime.GetIntDate(issuedAt).ToString(),
ClaimValueTypes.Integer64),
        new (ClaimTypes.Name, user.Login),
        new (ClaimTypes.UserData, user.NickName)
    };
```

```
        var tokenDescriptor = new SecurityTokenDescriptor
        {
            Subject = new ClaimsIdentity(claims),
            Expires = expiresAt,
            IssuedAt = issuedAt,
            NotBefore = issuedAt,
            Issuer = issuer,
            Audience = audience,
            SigningCredentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256Signature)
        };

        var tokenHandler = new JwtSecurityTokenHandler();
        var token = tokenHandler.CreateToken(tokenDescriptor);
        return tokenHandler.WriteToken(token);
    }
}
```

OtoChatHub.cs

```
using System.Security.Claims;
using MessagingSystem.Services.Messaging.Application.MessageDto;
using MessagingSystem.Services.Messaging.Application.Oto.OtoChats;
using MessagingSystem.Services.Messaging.Application.Oto.OtoChats.Dto;
using MessagingSystem.Services.Messaging.Application.Oto.OtoMessages;
using MessagingSystem.Services.Messaging.Application.Oto.OtoMessages.Dto;
using MessagingSystem.Services.Messaging.Core;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.SignalR;

namespace MessagingSystem.Services.Messaging.Infrastructure.ChatsHubs.Oto;

[Authorize]
public class OtoChatHub(
    ILogger<OtoChatHub> logger,
    IMessageOrchestrator messageOrchestrator,
    IChatOrchestrator chatOrchestrator)
    : Hub
{
    private string CurrentUserNickname =>
        Context.User?.Claims.FirstOrDefault(c => c.Type == ClaimTypes.UserData)?.Value
        ?? throw new HubException("Unauthorized");

    public override async Task OnConnectedAsync()
    {
        var nickname = CurrentUserNickname;

        logger.LogInformation("User {Nickname} connected to chat hub", nickname);
        await Groups.AddToGroupAsync(Context.ConnectionId, nickname);

        await base.OnConnectedAsync();
    }
    public override async Task OnDisconnectedAsync(Exception? exception)
    {
        var nickname = CurrentUserNickname;
        await Groups.RemoveFromGroupAsync(Context.ConnectionId, nickname);

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

```
  public async Task<object> SendMessageAsync(string recipientNickname, string message)
  {
      var sender = CurrentUserNickname;

      logger.LogInformation("Sending message from {Sender} to {Recipient}", sender, recipientNickname);

      var result = await messageOrchestrator.SendMessageAsync(new MessagesDto(sender, recipientNickname,
message));
      if (!result.Success || result.Data == null)
          throw new HubException(result.Message ?? "Message could not be saved");

      var messageData = new
      {
          messageId = result.Data.MessageId,
          sender,
          recipient = recipientNickname,
          content = message,
          sentAt = result.Data.SentTime
      };

      await NotifyUsersAsync(sender, recipientNickname, "ReceivePrivateMessage", messageData);
      return messageData;
  }
  public async Task EditMessageAsync(Guid messageId, string content)
  {
      var sender = CurrentUserNickname;

      var editResult = await messageOrchestrator.EditMessageAsync(messageId, new EditMessageDto(content));
      if (!editResult.Success)
          throw new HubException(editResult.Message);

      var messageOwner = await messageOrchestrator.FindMessageWithRecipientByIdAsync(messageId);
      if (messageOwner.Data == null) return;
      var editInfo = new
      {
          messageId,
          newContent = content,
          editedAt = DateTime.UtcNow
      };

      await NotifyUsersAsync(sender, messageOwner.Data.Recipient, "MessageEdited", editInfo);
  }
  public async Task DeleteMessageAsync(Guid messageId, string typeOfDeleting)
  {
      var sender = CurrentUserNickname;
      var deletedAt = DateTime.UtcNow;

      if (typeOfDeleting.Equals("soft", StringComparison.OrdinalIgnoreCase))
      {
          var softDeleteResult = await messageOrchestrator.SoftDeleteMessageAsync(messageId);
          if (!softDeleteResult.Success)
              throw new HubException(softDeleteResult.Message ?? "Soft delete failed");

          var messageOwner = await messageOrchestrator.FindMessageWithRecipientByIdAsync(messageId);
          if (messageOwner.Data == null)
              throw new HubException("Message not found");
```

```
  await NotifyUsersAsync(sender, messageOwner.Data.Recipient, "MessageDeleted", new
        {
            messageId,
            deletedAt,
            type = "soft"
        });
    }
    else
    {
        var messageOwner = await messageOrchestrator.FindMessageWithRecipientByIdAsync(messageId);
        if (messageOwner.Data == null)
            throw new HubException("Message not found");

        var deleteResult = await messageOrchestrator.DeleteMessageAsync(messageId);
        if (!deleteResult.Success)
            throw new HubException(deleteResult.Message ?? "Hard delete failed");

        await NotifyUsersAsync(sender, messageOwner.Data.Recipient, "MessageDeleted", new
        {
            messageId,
            deletedAt,
            type = "hard"
        });
    }
  }
  public async Task<List<object>> LoadChatHistoryAsync(string withUser, int take, int skip)
  {
      var currentUser = CurrentUserNickname;

      var messages = await messageOrchestrator.LoadChatHistory(currentUser, withUser, skip, take);

return messages.Select(m => new
      {
          messageId = m.Id,
          sender = m.Sender,
          content = m.Content,
          sentAt = m.SendTime,
          isDeleted = m.IsDeleted,
          isEdited = m.IsEdited,
          editedAt = m.EditDate,
          replyFor = m.ReplyFor
      }).Cast<object>().ToList();
  }
  public async Task<List<ChatDto>?> GetChatsAsync()
  {
      var nickname = CurrentUserNickname;
      return await chatOrchestrator.GetChatsAsync(nickname);
  }
  public async Task<object> ReplyToMessageAsync(string recipientNickname, string message, Guid
replyToMessageId)
  {
      var sender = CurrentUserNickname;

      logger.LogInformation("User {Sender} replying to message {ReplyToId} for {Recipient}", sender,
replyToMessageId, recipientNickname);
```

```
    var sendResult = await messageOrchestrator.SendMessageAsync(new MessagesDto(sender,
recipientNickname, message));
        if (!sendResult.Success || sendResult.Data == null)
            throw new HubException(sendResult.Message ?? "Failed to send message");

        var replyResult = await messageOrchestrator.ReplyForMessageAsync(sendResult.Data.MessageId,
replyToMessageId);
        if (!replyResult.Success || replyResult.Data == null)
            throw new HubException(replyResult.Message ?? "Failed to attach reply");

        var resultData = new
        {
            messageId = replyResult.Data.Id,
            sender,
            content = replyResult.Data.Content,
            sentAt = replyResult.Data.SendTime,
            replyTo = replyToMessageId
        };

        await NotifyUsersAsync(sender, recipientNickname, "ReplyToMessageAsync", resultData);
        return resultData;
    }
    public async Task<List<object>> FindMessagesAsync(string? recipient, DateTime? time, string? sender)
    {
        var filter = new MessageFilter
        {
            Sender = sender,
            Date = time
        };

        var messages = await messageOrchestrator.FindMessagesAsync(filter);
        if (messages == null || !messages.Any())
            return [];

        return messages.Select(m => new
        {
            messageId = m.Id,
            sender = m.Sender,
            content = m.Content,
            sentAt = m.SendTime,
            isEdited = m.IsEdited,
            replyFor = m.ReplyFor
        }).Cast<object>().ToList();
    }
    private Task NotifyUsersAsync(string user1, string? user2, string method, object data)
    {
        var tasks = new List<Task> { Clients.Group(user1).SendAsync(method, data) };
        if (!string.IsNullOrEmpty(user2) && user2 != user1)
            tasks.Add(Clients.Group(user2).SendAsync(method, data));

        return Task.WhenAll(tasks);
    }
}
```

GroupChatHub.cs

```
using System.Security.Claims;
using AutoMapper;
using MessagingSystem.Services.Messaging.Application.Group.GroupMessages;
using MessagingSystem.Services.Messaging.Application.Group.GroupMessages.Dto;
using MessagingSystem.Services.Messaging.Application.Group.GroupsInformation;
using MessagingSystem.Services.Messaging.Application.Group.GroupsInformation.Dto;
using MessagingSystem.Services.Messaging.Application.MessageDto;
using MessagingSystem.Services.Messaging.Core;
using MessagingSystem.Services.Messaging.Core.Groups.GroupMessages;
using MessagingSystem.Services.Messaging.Infrastructure.Hasher;
using MessagingSystem.Services.Messaging.WebApi.Group.Info.Contracts.GroupInfo;
using MessagingSystem.Services.Messaging.WebApi.Group.Info.Contracts.Members;
using MessagingSystem.Services.Messaging.WebApi.Group.Messages.Contracts;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.SignalR;

namespace MessagingSystem.Services.Messaging.Infrastructure.ChatsHubs.Group;

[Authorize]
public class GroupChatHub(
    ILogger<GroupChatHub> logger,
    IGroupInfoOrchestrator groupInfoOrchestrator,
    IGroupMessagesOrchestrator groupMessagesOrchestrator,
    IMapper mapper,
    IHasher hasher
) : Hub
{
    private string CurrentUserNickname =>
        Context.User?.Claims.FirstOrDefault(c => c.Type == ClaimTypes.UserData)?.Value
        ?? throw new HubException("Unauthorized");
    public override async Task OnConnectedAsync()
    {
        var nickname = CurrentUserNickname;
        logger.LogInformation("User {Nickname} connected to group chat hub", nickname);
        await Groups.AddToGroupAsync(Context.ConnectionId, nickname);
        await base.OnConnectedAsync();
    }
public override async Task OnDisconnectedAsync(Exception? exception)
    {
        var nickname = CurrentUserNickname;
        await Groups.RemoveFromGroupAsync(Context.ConnectionId, nickname);
        await base.OnDisconnectedAsync(exception);
    }
    public async Task<List<GroupDto>> GetAllGroupForUserAsync()
    {
        var nickname = CurrentUserNickname;
        var groups = await groupInfoOrchestrator.GetGroupsForUserAsync(nickname);
        return groups is { Success: true, Data: not null }
            ? groups.Data
            : [];
    }
```

```
public async Task<GroupDto> CreateGroupAsync(CreateGroup groupInfo)
{
    groupInfo.Admin = CurrentUserNickname;

    var dto = mapper.Map<GroupDto>(groupInfo);
    var result = await groupInfoOrchestrator.CreateGroupAsync(dto);

    if (!result.Success || result.Data == null)
        throw new HubException(result.Message ?? "Failed to create group");

    await Groups.AddToGroupAsync(Context.ConnectionId, result.Data.GroupName);

    return result.Data;
}
public async Task<GroupDto> EditGroupAsync(Guid groupId, EditGroup groupInfo)
{
    var dto = mapper.Map<EditGroupDto>(groupInfo);

    var result = await groupInfoOrchestrator.EditGroupInfoAsync(groupId, dto);

    if (!result.Success || result.Data == null)
        throw new HubException(result.Message ?? "Failed to edit group");

    return result.Data;
}
public async Task<string> DeleteGroupAsync(Guid groupId)
{
    var result = await groupInfoOrchestrator.DeleteGroupInfoAsync(groupId, CurrentUserNickname);

    if (!result.Success || result.Data == null)
        throw new HubException(result.Message ?? "Failed to delete group");

    await Clients.All.SendAsync("DeleteGroupAsync", groupId.ToString());

    return result.Data;
}
public async Task<GroupDto> AddMembersToGroupAsync(Guid groupId, AddMembers members)
{
    var dto = mapper.Map<GroupMembersDto>(members);
    var adminHash = hasher.Hash(CurrentUserNickname);

    var result = await groupInfoOrchestrator.AddMembersToGroupAsync(groupId, dto, adminHash);

 if (!result.Success || result.Data == null)
        throw new HubException(result.Message ?? "Failed to add members");

    await NotifyUsersInGroupAsync(groupId.ToString(), "GroupMembersAdded", result.Data);

    return result.Data;
}
public async Task<GroupDto> RemoveMembersFromGroupAsync(Guid groupId, AddMembers members)
{
    var dto = mapper.Map<GroupMembersDto>(members);
    var adminHash = hasher.Hash(CurrentUserNickname);

    var result = await groupInfoOrchestrator.DeleteMembersFromGroupAsync(groupId, dto, adminHash);
```

```
    if (!result.Success || result.Data == null)
        throw new HubException(result.Message ?? "Failed to remove members");
      await NotifyUsersInGroupAsync(groupId.ToString(), "GroupMembersRemoved", result.Data);

      await NotifyRemovedUsersAsync(members.Users, groupId, result.Data);

      return result.Data;
   }
   public Task JoinGroupAsync(Guid groupId)
   {
      return Groups.AddToGroupAsync(Context.ConnectionId, groupId.ToString());
   }
   public Task LeaveGroupAsync(Guid groupId)
   {
      return Groups.RemoveFromGroupAsync(Context.ConnectionId, groupId.ToString());
   }
   public async Task<List<GroupMessage>> LoadChatHistoryAsync(Guid groupId, int skip, int take)
   {
      var messages = await groupMessagesOrchestrator.LoadChatHistory(groupId, skip, take);
      return messages;
   }
   public async Task<object> SendMessageAsync(string content, Guid groupId)
   {
      var nickname = CurrentUserNickname;

      var message = mapper.Map<GroupMessageDto>(new CreateGroupMessage(nickname, content, groupId));

      var result = await groupMessagesOrchestrator.SendMessageAsync(message);

      if (result.Data == null)
         throw new HubException("Failed to send message");


      var messageResult = new
      {
         Id = result.Data.MessageId,
         GroupId = groupId,
         Sender = nickname,
         Content = content,
         SendTime = result.Data.SentTime,
      };
      await NotifyUsersInGroupAsync(message.GroupId.ToString(), "ReceiveMessage", messageResult);
      return result.Data;
   }

   public async Task EditMessageAsync(Guid messageId, string content, Guid groupId)
   {
      var result = await groupMessagesOrchestrator.EditMessageAsync(messageId, new
EditMessageDto(content));

      if (!result.Success)
         throw new HubException(result.Message ?? "Failed to edit message");
```

```
    var editInfo = new
        {
            messageId,
            newContent = content,
            editedAt = DateTime.UtcNow,
            isEdited = true
        };

        await NotifyUsersInGroupAsync(groupId.ToString(), "MessageEdited", editInfo);
    }
    public async Task SoftDeleteMessageAsync(Guid messageId, Guid groupId)
    {
        var result = await groupMessagesOrchestrator.SoftDeleteMessageAsync(messageId);

        if (!result.Success)
            throw new HubException(result.Message ?? "Failed to soft delete message");

        var message = await groupMessagesOrchestrator.FindMessageByIdAsync(messageId);
        if(!message.Success)
            throw new HubException("Message not found");

        var deleteInfo = new
        {
            MessageId = messageId,
            GroupId = groupId,
            isDeleted = true
        };

        await NotifyUsersInGroupAsync(groupId.ToString(), "MessageSoftDeleted", deleteInfo);
    }
    public async Task DeleteMessageAsync(Guid messageId, Guid groupId)
    {
        var result = await groupMessagesOrchestrator.DeleteMessageAsync(messageId);

        if (!result.Success)
            throw new HubException(result.Message ?? "Failed to delete message");

        var deleteInfo = new
        {
            MessageId = messageId,
            GroupId = groupId
        };

        await NotifyUsersInGroupAsync(groupId.ToString(), "MessageDeleted", deleteInfo);
    }
    public async Task<object> ReplyForMessageAsync(Guid messageId, string message, Guid groupId)
    {
        var sender = CurrentUserNickname;
            var sendResult = await groupMessagesOrchestrator.SendMessageAsync(new
GroupMessageDto(sender, message, groupId));
        if (!sendResult.Success || sendResult.Data == null)
            throw new HubException(sendResult.Message ?? "Failed to send message");

        var replyResult = await groupMessagesOrchestrator.ReplyForMessageAsync(sendResult.Data.MessageId,
messageId);
        if (!replyResult.Success || replyResult.Data == null)
            throw new HubException(replyResult.Message ?? "Failed to attach reply");
```

```
    var resultData = new
    {
        messageId = replyResult.Data.Id,
        GroupId = groupId,
        sender,
        content = replyResult.Data.Content,
        sentAt = replyResult.Data.SendTime,
        replyTo = messageId
    };

    await NotifyUsersInGroupAsync(groupId.ToString(), "MessageReplied", resultData);
    return resultData;
}
public async Task<List<object>> FindMessageByFilterAsync(string? recipient, DateTime? time, string? sender)
{
    var filter = new MessageFilter()
    {
        Sender = sender,
        Recipient = recipient,
        Date = time
    };

    var messages = await groupMessagesOrchestrator.FindMessagesAsync(filter);

    if (messages == null || !messages.Any())
        return [];

    return messages.Select(m => new
    {
        messageId = m.Id,
        sender = m.Sender,
        content = m.Content,
        sentAt = m.SendTime,
        isEdited = m.IsEdited,
        replyFor = m.ReplyFor
    }).Cast<object>().ToList();
}
private async Task NotifyUsersInGroupAsync(string groupIdOrGuid, string method, object data)
{
    if (Guid.TryParse(groupIdOrGuid, out var groupId))
    {
        var groupResult = await groupInfoOrchestrator.FindGroupByIdAsync(groupId);
        var members = groupResult.Data?.Users;
      if (members is { Count: > 0 })
        {
            var tasks = members.Select(nick => Clients.Group(nick).SendAsync(method, data));
            await Task.WhenAll(tasks);
            return;
        }
    }

    Console.WriteLine($"[Hub] Fallback - sending to group: {groupIdOrGuid}");
    await Clients.Group(groupIdOrGuid).SendAsync(method, data);
}
```

```
    private async Task NotifyRemovedUsersAsync(List<string> removedUsers, Guid groupId, GroupDto updatedGroup)
    {
        if (removedUsers is { Count: > 0 })
        {
            var removalNotification = new
            {
                GroupId = groupId,
                groupName = updatedGroup.GroupName,
                removedFromGroup = true,
                message = "You have been removed from the group"
            };

            var tasks = removedUsers.Select(userName => Clients.Group(userName).SendAsync("UserRemovedFromGroup", removalNotification));

            await Task.WhenAll(tasks);
        }
    }
}
```

e2ee.ts

```
import { Injectable } from '@angular/core';
import * as bip39 from 'bip39';
import sodium from 'libsodium-wrappers-sumo';

interface StoredKeys {
  edPrivateKey: Uint8Array;
  xPrivateKey: Uint8Array;
  xPublicKey: Uint8Array;
  mnemonic: string;
  timestamp: number;
}

interface RatchetState {
  rootKey: Uint8Array;
  sendingChainKey: Uint8Array;
  receivingChainKey: Uint8Array;
  sendMessageNumber: number;
  recvMessageNumber: number;
  dhRatchetPrivate: Uint8Array;
  dhRatchetPublic: Uint8Array;
  remotePublicKey?: Uint8Array;
  ratchetId?: string;
}

interface SenderKeyState {
  chainKey: Uint8Array;
  generationId: number;
  chainIndex: number;
  senderKeyId: string;
}

interface MessageKeyData {
  messageKey: Uint8Array;
  chainKeySnapshot: Uint8Array;
  keyIndex: number;
}
```

```
@Injectable({ providedIn: 'root' })
export class E2eeService {
  private ready = false;
  private keys: StoredKeys | null = null;
  private ratchetStates = new Map<string, RatchetState>();
  public senderKeys = new Map<string, SenderKeyState>();
  public receivedSenderKeys = new Map<string, Map<string, SenderKeyState>>();
  public messageKeysCache = new Map<string, MessageKeyData[]>();

  private keysLoaded = false;


  constructor() {
    sodium.ready.then(() => {
      this.ready = true;
    });
  }

  private ensureReady() {
    if (!this.ready) throw new Error('libsodium not ready yet');
  }

  private generateRatchetId(): string {
    return `ratchet_${Date.now()}_${Math.random().toString(36).substr(2, 9)}`;
  }

  async saveMessageKeyForHistory(
    contactId: string,
    messageNumber: number,
    messageKey: Uint8Array,
    chainKeySnapshot: Uint8Array
  ): Promise<MessageKeyData> {
    const keyData: MessageKeyData = {
      messageKey,
      chainKeySnapshot,
      keyIndex: messageNumber
    };

    const cacheKey = `${contactId}`;
    if (!this.messageKeysCache.has(cacheKey)) {
      this.messageKeysCache.set(cacheKey, []);
    }
    this.messageKeysCache.get(cacheKey)!.push(keyData);

return keyData;
  }

exportMessageKeyForGroupMember(
  groupId: string,
  memberPublicKey: Uint8Array,
  chainIndex: number
): {
  encryptedKey: string;
  ephemeralPublicKey: string;
  chainKeySnapshot: string;
  chainIndex: number;
} | null {
  this.ensureReady();
```

```
  const senderKeyState = this.senderKeys.get(groupId);
  if (!senderKeyState) {
    return null;
  }

  const messageKey = sodium.crypto_generichash(
    32,
    new Uint8Array([...senderKeyState.chainKey, chainIndex])
  );

  const ephemeralKeyPair = sodium.crypto_box_keypair();

  const sharedSecret = sodium.crypto_scalarmult(
    ephemeralKeyPair.privateKey,
    memberPublicKey
  );

  const encryptionKey = sodium.crypto_generichash(32, sharedSecret);

  const nonce = sodium.randombytes_buf(sodium.crypto_secretbox_NONCEBYTES);
  const encrypted = sodium.crypto_secretbox_easy(
    messageKey,
    nonce,
    encryptionKey
  );

  return {
    encryptedKey: this.toBase64(new Uint8Array([...nonce, ...encrypted])),
    ephemeralPublicKey: this.toBase64(ephemeralKeyPair.publicKey),
    chainKeySnapshot: this.toBase64(senderKeyState.chainKey),
    chainIndex: chainIndex
  };
}

  async getMessageKeyForHistory(
    contactId: string,
    messageNumber: number
  ): Promise<Uint8Array | null> {
    const cacheKey = `${contactId}`;
    const keys = this.messageKeysCache.get(cacheKey);

    if (!keys) return null;

const keyData = keys.find(k => k.keyIndex === messageNumber);
    return keyData ? keyData.messageKey : null;
  }

  async encryptMessageWithHistory(contactId: string, plaintext: string): Promise<{
    ciphertext: string;
    ephemeralKey: string;
    nonce: string;
    messageNumber: number;
    previousChainN: number;
    ratchetId: string;
    messageKeyData: MessageKeyData;
  }> {
    this.ensureReady();
```

```
  const state = this.ratchetStates.get(contactId);
  if (!state) {
    throw new Error('Ratchet state not initialized');
  }

  if (!state.ratchetId) {
    state.ratchetId = this.generateRatchetId();
    this.saveRatchetStateToSession(contactId);
  }

  const currentMessageNumber = state.sendMessageNumber;
  const previousChainN = state.recvMessageNumber;
  const currentEphemeralKey = this.toBase64(state.dhRatchetPublic);
  const currentRatchetId = state.ratchetId;

  const chainKeySnapshot = new Uint8Array(state.sendingChainKey);

  const messageKey = sodium.crypto_generichash(32,
    new Uint8Array([...state.sendingChainKey, currentMessageNumber])
  );

  const messageKeyData = await this.saveMessageKeyForHistory(
    contactId,
    currentMessageNumber,
    messageKey,
    chainKeySnapshot
  );

  state.sendingChainKey = sodium.crypto_generichash(32, state.sendingChainKey);

  const nonce = sodium.randombytes_buf(sodium.crypto_secretbox_NONCEBYTES);
  const cipher = sodium.crypto_secretbox_easy(plaintext, nonce, messageKey);

  state.sendMessageNumber++;
  if (state.sendMessageNumber > 0 && state.sendMessageNumber % 50 === 0) {
    const newDHKeyPair = sodium.crypto_box_keypair();

    const dhOutput = sodium.crypto_scalarmult(
      newDHKeyPair.privateKey,
      state.remotePublicKey!
    );

state.rootKey = sodium.crypto_generichash(32,
      new Uint8Array([...state.rootKey, ...dhOutput, 1])
    );

    state.sendingChainKey = sodium.crypto_generichash(32,
      new Uint8Array([...state.rootKey, 3])
    );

    state.dhRatchetPrivate = newDHKeyPair.privateKey;
    state.dhRatchetPublic = newDHKeyPair.publicKey;
  }

  this.ratchetStates.set(contactId, state);
  this.saveRatchetStateToSession(contactId);
```

```
      return {
      ciphertext: this.toBase64(cipher),
      ephemeralKey: currentEphemeralKey,
      nonce: this.toBase64(nonce),
      messageNumber: currentMessageNumber,
      previousChainN: previousChainN,
      ratchetId: currentRatchetId,
      messageKeyData
    };
  }

  async decryptHistoricalMessage(
    contactId: string,
    ciphertext: string,
    nonce: string,
    messageNumber: number
  ): Promise<string> {
    this.ensureReady();

    const messageKey = await this.getMessageKeyForHistory(contactId, messageNumber);
    if (!messageKey) {
      throw new Error('Message key not found in history');
    }

    const cipher = this.fromBase64(ciphertext);
    const nonceBytes = this.fromBase64(nonce);
    const decrypted = sodium.crypto_secretbox_open_easy(cipher, nonceBytes, messageKey);

    if (!decrypted) throw new Error('Decryption failed');

    return new TextDecoder().decode(decrypted);
  }

  async decryptMessage(
    contactId: string,
    ciphertext: string,
    ephemeralKey: string,
    nonce: string,
    messageNumber: number,
    ratchetId?: string
  ): Promise<string> {
    this.ensureReady();
let state = this.ratchetStates.get(contactId);
    if (!state) {
      throw new Error('Ratchet state not initialized');
    }

    const remoteEphemeralKey = this.fromBase64(ephemeralKey);
    const isNewRatchet = ratchetId && state.ratchetId && ratchetId !== state.ratchetId;
    const isKeyChanged = !state.remotePublicKey || !this.arraysEqual(state.remotePublicKey,
remoteEphemeralKey);

    if (isNewRatchet) {
      const dhOutput = sodium.crypto_scalarmult(
        state.dhRatchetPrivate,
        remoteEphemeralKey
      );
```

```
  state.rootKey = sodium.crypto_generichash(32, dhOutput);
  state.receivingChainKey = sodium.crypto_generichash(32,
    new Uint8Array([...state.rootKey, 2])
  );
  state.remotePublicKey = remoteEphemeralKey;
  state.recvMessageNumber = 0;
  state.ratchetId = ratchetId;

  const newDHKeyPair = sodium.crypto_box_keypair();
  state.dhRatchetPrivate = newDHKeyPair.privateKey;
  state.dhRatchetPublic = newDHKeyPair.publicKey;
  state.sendMessageNumber = 0;

  this.messageKeysCache.delete(contactId);
}
else if (isKeyChanged) {
  const dhOutput1 = sodium.crypto_scalarmult(
    state.dhRatchetPrivate,
    remoteEphemeralKey
  );

  state.rootKey = sodium.crypto_generichash(32,
    new Uint8Array([...state.rootKey, ...dhOutput1, 1])
  );

  state.receivingChainKey = sodium.crypto_generichash(32,
    new Uint8Array([...state.rootKey, 2])
  );

  state.remotePublicKey = remoteEphemeralKey;
  state.recvMessageNumber = 0;

  const newDHKeyPair = sodium.crypto_box_keypair();
  const dhOutput2 = sodium.crypto_scalarmult(
    newDHKeyPair.privateKey,
    remoteEphemeralKey
  );

  state.rootKey = sodium.crypto_generichash(32,
    new Uint8Array([...state.rootKey, ...dhOutput2, 1])
  );


   state.sendingChainKey = sodium.crypto_generichash(32,
    new Uint8Array([...state.rootKey, 3])
  );

  state.dhRatchetPrivate = newDHKeyPair.privateKey;
  state.dhRatchetPublic = newDHKeyPair.publicKey;
  state.sendMessageNumber = 0;
}

if (messageNumber < state.recvMessageNumber) {
  const cachedKey = await this.getMessageKeyForHistory(contactId, messageNumber);
  if (!cachedKey) {
    throw new Error(`Missing key for message ${messageNumber}`);
  }
```

```
    const cipher = this.fromBase64(ciphertext);
    const nonceBytes = this.fromBase64(nonce);
    const decrypted = sodium.crypto_secretbox_open_easy(cipher, nonceBytes, cachedKey);

    if (!decrypted) throw new Error('Decryption failed');
    return new TextDecoder().decode(decrypted);
   }

   while (state.recvMessageNumber < messageNumber) {
    const chainKeySnapshot = new Uint8Array(state.receivingChainKey);
    const skippedMessageKey = sodium.crypto_generichash(32,
     new Uint8Array([...state.receivingChainKey, state.recvMessageNumber])
    );

    await this.saveMessageKeyForHistory(
     contactId,
     state.recvMessageNumber,
     skippedMessageKey,
     chainKeySnapshot
    );

    state.receivingChainKey = sodium.crypto_generichash(32, state.receivingChainKey);
    state.recvMessageNumber++;
   }

   const chainKeySnapshot = new Uint8Array(state.receivingChainKey);
   const messageKey = sodium.crypto_generichash(32,
    new Uint8Array([...state.receivingChainKey, messageNumber])
   );

   await this.saveMessageKeyForHistory(
    contactId,
    messageNumber,
    messageKey,
    chainKeySnapshot
   );

   state.receivingChainKey = sodium.crypto_generichash(32, state.receivingChainKey);
   state.recvMessageNumber++;

try {
    const cipher = this.fromBase64(ciphertext);
    const nonceBytes = this.fromBase64(nonce);
    const decrypted = sodium.crypto_secretbox_open_easy(cipher, nonceBytes, messageKey);

    if (!decrypted) throw new Error('Decryption failed');

    this.ratchetStates.set(contactId, state);
    this.saveRatchetStateToSession(contactId);

    return new TextDecoder().decode(decrypted);
   } catch (error) {
    throw error;
   }
  }
```

```
 async cleanupOldMessageKeys(contactId: string, keepLast: number = 100): Promise<void> {
   const keys = this.messageKeysCache.get(contactId);
   if (!keys || keys.length <= keepLast) return;

   keys.sort((a, b) => a.keyIndex - b.keyIndex);
   const toKeep = keys.slice(-keepLast);
   this.messageKeysCache.set(contactId, toKeep);
 }

 async importMessageKeyFromServer(
   contactId: string,
   userId: string,
   encryptedKey: string,
   chainKeySnapshot: string,
   messageNumber: number
 ): Promise<void> {
   this.ensureReady();

   const userKeys = this.getKeys();
   if (!userKeys) throw new Error('No user keys available');

   const derivedKey = sodium.crypto_generichash(32,
     new Uint8Array([...userKeys.xPrivateKey, ...new TextEncoder().encode(userId)])
   );

   const encryptedData = this.fromBase64(encryptedKey);
   const nonce = encryptedData.slice(0, sodium.crypto_secretbox_NONCEBYTES);
   const cipher = encryptedData.slice(sodium.crypto_secretbox_NONCEBYTES);

   const messageKey = sodium.crypto_secretbox_open_easy(cipher, nonce, derivedKey);
   if (!messageKey) throw new Error('Failed to decrypt message key');

   await this.saveMessageKeyForHistory(
     contactId,
     messageNumber,
     messageKey,
     this.fromBase64(chainKeySnapshot)
   );
 }
exportMessageKeysForServer(contactId: string, userId: string): {
   messageNumber: number;
   encryptedKey: string;
   chainKeySnapshot: string;
 } | null {
   const keys = this.messageKeysCache.get(contactId);
   if (!keys || keys.length === 0) {
     return null;
   }

   const userKeys = this.getKeys();
   if (!userKeys) throw new Error('No user keys available');

   const lastKey = keys[keys.length - 1];

   const derivedKey = sodium.crypto_generichash(32,
     new Uint8Array([...userKeys.xPrivateKey, ...new TextEncoder().encode(userId)])
   );
```

```
    const nonce = sodium.randombytes_buf(sodium.crypto_secretbox_NONCEBYTES);
    const encrypted = sodium.crypto_secretbox_easy(
      lastKey.messageKey,
      nonce,
      derivedKey
    );

    return {
      messageNumber: lastKey.keyIndex,
      encryptedKey: this.toBase64(new Uint8Array([...nonce, ...encrypted])),
      chainKeySnapshot: this.toBase64(lastKey.chainKeySnapshot)
    };
  }

  storeKeys(keys: Omit<StoredKeys, 'timestamp'>) {
    this.keys = {
      ...keys,
      timestamp: Date.now()
    };
    this.keysLoaded = true;
  }

  hasKeysLoaded(): boolean {
    return this.keysLoaded;
  }

  private saveRatchetStateToSession(contactId: string) {
    const stateJson = this.exportRatchetState(contactId);
    if (stateJson) {
      sessionStorage.setItem(`ratchet_${contactId}`, stateJson);
    }
  }

loadRatchetStateFromSession(contactId: string): boolean {
    const stateJson = sessionStorage.getItem(`ratchet_${contactId}`);
    if (stateJson) {
      try {
        this.importRatchetState(contactId, stateJson);
        return true;
      } catch (error) {
        return false;
      }
    }
    return false;
  }

  getKeys(): StoredKeys | null {
    return this.keys;
  }

  clearKeys() {
    this.keys = null;
    this.keysLoaded = false;
    this.ratchetStates.clear();
    this.senderKeys.clear();
    this.receivedSenderKeys.clear();
    this.messageKeysCache.clear();
```

```
    sessionStorage.removeItem('e2ee_keys_password');
    Object.keys(sessionStorage).forEach(key => {
      if (key.startsWith('ratchet_')) {
        sessionStorage.removeItem(key);
      }
    });
  }

  hasKeys(): boolean {
    return this.keys !== null;
  }

  getPrivateKey(): Uint8Array | null {
    return this.keys?.xPrivateKey || null;
  }

  getPublicKey(): Uint8Array | null {
    return this.keys?.xPublicKey || null;
  }

  generateMnemonic(strength = 256): string {
    return bip39.generateMnemonic(strength);
  }

  async mnemonicToSeed32(mnemonic: string, passphrase = ''): Promise<Uint8Array> {
    this.ensureReady();
    const seed = await bip39.mnemonicToSeed(mnemonic, passphrase);
    return new Uint8Array(seed.slice(0, 32));
  }

async keypairFromSeed32(seed32: Uint8Array) {
    await sodium.ready;
    const kp = sodium.crypto_sign_seed_keypair(seed32);
    return {
      edPublicKey: new Uint8Array(kp.publicKey),
      edPrivateKey: new Uint8Array(kp.privateKey),
    };
  }

  async convertEdToX(edPublicKey: Uint8Array, edPrivateKey: Uint8Array) {
    this.ensureReady();
    const xPublic = sodium.crypto_sign_ed25519_pk_to_curve25519(edPublicKey);
    const xPrivate = sodium.crypto_sign_ed25519_sk_to_curve25519(edPrivateKey);
    return {
      xPublicKey: new Uint8Array(xPublic),
      xPrivateKey: new Uint8Array(xPrivate),
    };
  }

  toBase64(u8: Uint8Array): string {
    return sodium.to_base64(u8, sodium.base64_variants.ORIGINAL);
  }

  fromBase64(b64: string): Uint8Array {
    return sodium.from_base64(b64, sodium.base64_variants.ORIGINAL);
  }
```

```
async createEncryptedBackupAndDownload(
  edPrivateKey: Uint8Array,
  mnemonic: string,
  password: string | null
) {
  this.ensureReady();

  const payload = {
    version: 1,
    createdAt: new Date().toISOString(),
    mnemonic,
    edPrivateKey: this.toBase64(edPrivateKey)
  };

  const payloadStr = JSON.stringify(payload);

  const pw = password || '';
  const salt = sodium.randombytes_buf(sodium.crypto_pwhash_SALTBYTES);
  const opslimit = sodium.crypto_pwhash_OPSLIMIT_INTERACTIVE;
  const memlimit = sodium.crypto_pwhash_MEMLIMIT_INTERACTIVE;

  const key = sodium.crypto_pwhash(
    sodium.crypto_secretbox_KEYBYTES,
    pw,
    salt,
    opslimit,
    memlimit,
    sodium.crypto_pwhash_ALG_DEFAULT
  );
  const nonce = sodium.randombytes_buf(sodium.crypto_secretbox_NONCEBYTES);
  const cipher = sodium.crypto_secretbox_easy(payloadStr, nonce, key);

  const backupJson = {
    version: 1,
    salt: this.toBase64(salt),
    nonce: this.toBase64(nonce),
    opslimit,
    memlimit,
    ciphertext: this.toBase64(cipher),
  };

  const blob = new Blob([JSON.stringify(backupJson)], { type: 'application/json' });
  const url = URL.createObjectURL(blob);
  const a = document.createElement('a');
  a.href = url;
  a.download = `e2ee-backup-${new Date().toISOString()}.json`;
  document.body.appendChild(a);
  a.click();
  a.remove();
  URL.revokeObjectURL(url);
}
```

```
async decryptBackup(backupJson: any, password: string) {
  this.ensureReady();
  const salt = this.fromBase64(backupJson.salt);
  const nonce = this.fromBase64(backupJson.nonce);
  const cipher = this.fromBase64(backupJson.ciphertext);
  const opslimit = backupJson.opslimit;
  const memlimit = backupJson.memlimit;

  const key = sodium.crypto_pwhash(
    sodium.crypto_secretbox_KEYBYTES,
    password,
    salt,
    opslimit,
    memlimit,
    sodium.crypto_pwhash_ALG_DEFAULT
  );

  const decrypted = sodium.crypto_secretbox_open_easy(cipher, nonce, key);
  if (!decrypted) throw new Error('Invalid password or corrupted backup');

  const decoder = new TextDecoder();
  const payloadStr = decoder.decode(decrypted);
  return JSON.parse(payloadStr);
}

async restoreKeysFromBackup(backupJson: any, password: string) {
  try {
    const decrypted = await this.decryptBackup(backupJson, password);

    const mnemonic = decrypted.mnemonic;
    const seed32 = await this.mnemonicToSeed32(mnemonic);
    const ed = await this.keypairFromSeed32(seed32);
    const x = await this.convertEdToX(ed.edPublicKey, ed.edPrivateKey);

   this.storeKeys({
      edPrivateKey: ed.edPrivateKey,
      xPrivateKey: x.xPrivateKey,
      xPublicKey: x.xPublicKey,
      mnemonic: mnemonic
    });

  } catch (error) {
    throw error;
  }
}

async generateSignedPreKey(edPrivateKey: Uint8Array): Promise<{
  publicKey: string;
  privateKey: Uint8Array;
  signature: string;
}> {
  this.ensureReady();
  const keyPair = sodium.crypto_box_keypair();
  const signature = sodium.crypto_sign_detached(keyPair.publicKey, edPrivateKey);
```

```
  return {
    publicKey: this.toBase64(keyPair.publicKey),
    privateKey: keyPair.privateKey,
    signature: this.toBase64(signature),
  };
}

async generateOneTimePreKeys(count: number): Promise<Array<{
  publicKey: string;
  privateKey: Uint8Array;
}>> {
  this.ensureReady();
  const keys: Array<{ publicKey: string; privateKey: Uint8Array }> = [];
  for (let i = 0; i < count; i++) {
    const keyPair = sodium.crypto_box_keypair();
    keys.push({
      publicKey: this.toBase64(keyPair.publicKey),
      privateKey: keyPair.privateKey,
    });
  }
  return keys;
}

async initRatchetAsSender(
  contactId: string,
  myXPrivate: Uint8Array,
  theirXPublic: Uint8Array,
  theirSignedPreKey: Uint8Array
): Promise<Uint8Array> {
  this.ensureReady();

  const ephemeralKeyPair = sodium.crypto_box_keypair();

  const dh1 = sodium.crypto_scalarmult(myXPrivate, theirSignedPreKey);
  const dh2 = sodium.crypto_scalarmult(ephemeralKeyPair.privateKey, theirXPublic);
  const dh3 = sodium.crypto_scalarmult(ephemeralKeyPair.privateKey, theirSignedPreKey);

  const info = new Uint8Array([...dh1, ...dh2, ...dh3]);
  const rootKey = sodium.crypto_generichash(32, info);

  const sendingChainKey = sodium.crypto_generichash(32, new Uint8Array([...rootKey, 1]));

  const ratchetId = this.generateRatchetId();

  this.ratchetStates.set(contactId, {
    rootKey,
    sendingChainKey,
    receivingChainKey: new Uint8Array(32),
    sendMessageNumber: 0,
    recvMessageNumber: 0,
    dhRatchetPrivate: ephemeralKeyPair.privateKey,
    dhRatchetPublic: ephemeralKeyPair.publicKey,
    remotePublicKey: theirXPublic,
    ratchetId,
  });
```

```
  this.saveRatchetStateToSession(contactId);
  return ephemeralKeyPair.publicKey;
}

async initRatchetAsReceiver(
  contactId: string,
  myXPrivate: Uint8Array,
  theirXPublic: Uint8Array,
  theirEphemeralKey: Uint8Array,
  mySignedPreKeyPrivate: Uint8Array
): Promise<void> {
  this.ensureReady();

  if (this.ratchetStates.has(contactId)) {
    return;
  }

  const dh1 = sodium.crypto_scalarmult(mySignedPreKeyPrivate, theirXPublic);
  const dh2 = sodium.crypto_scalarmult(myXPrivate, theirEphemeralKey);
  const dh3 = sodium.crypto_scalarmult(mySignedPreKeyPrivate, theirEphemeralKey);
  const info = new Uint8Array([...dh1, ...dh2, ...dh3]);
  const rootKey = sodium.crypto_generichash(32, info);
  const receivingChainKey = sodium.crypto_generichash(32, new Uint8Array([...rootKey, 1]));
  const newDHKeyPair = sodium.crypto_box_keypair();
  const ratchetId = this.generateRatchetId();

  this.ratchetStates.set(contactId, {
    rootKey,
    sendingChainKey: new Uint8Array(32),
    receivingChainKey,
    sendMessageNumber: 0,
    recvMessageNumber: 0,
    dhRatchetPrivate: newDHKeyPair.privateKey,
    dhRatchetPublic: newDHKeyPair.publicKey,
    remotePublicKey: theirEphemeralKey,
    ratchetId,
  });

  this.saveRatchetStateToSession(contactId);
}

async generateSenderKey(groupId: string): Promise<{
  chainKey: Uint8Array;
  senderKeyId: string;
  generationId: number;
}> {
  this.ensureReady();

  const chainKey = sodium.randombytes_buf(32);
  const senderKeyId = this.generateUUID();
  const generationId = 0;

  const state: SenderKeyState = {
    chainKey,
    generationId,
    chainIndex: 0,
    senderKeyId,
  };
```

```
  this.senderKeys.set(groupId, state);
  return { chainKey, senderKeyId, generationId };
}

async encryptSenderKeyForMember(
  senderKeyBundle: {
    chainKey: Uint8Array;
    senderKeyId: string;
    generationId: number;
  },
  memberPublicKey: Uint8Array
): Promise<{
  ephemeralKey: string;
  encryptedData: string;
  nonce: string;
}> {
  this.ensureReady();

  const keys = this.getKeys();
  if (!keys) throw new Error('No keys available');

  const ephemeralKeyPair = sodium.crypto_box_keypair();

  const sharedSecret = sodium.crypto_scalarmult(
    ephemeralKeyPair.privateKey,
    memberPublicKey
  );

  const encKey = sodium.crypto_generichash(32, sharedSecret);

  const bundle = JSON.stringify({
    chainKey: this.toBase64(senderKeyBundle.chainKey),
    senderKeyId: senderKeyBundle.senderKeyId,
    generationId: senderKeyBundle.generationId,
  });

  const nonce = sodium.randombytes_buf(sodium.crypto_secretbox_NONCEBYTES);
  const cipher = sodium.crypto_secretbox_easy(bundle, nonce, encKey);

  return {
    ephemeralKey: this.toBase64(ephemeralKeyPair.publicKey),
    encryptedData: this.toBase64(cipher),
    nonce: this.toBase64(nonce),
  };
}

async decryptSenderKeyFromMember(
  ephemeralKey: string,
  encryptedData: string,
  nonce: string,
  groupId: string,
  senderId: string
): Promise<void> {
  this.ensureReady();

  const keys = this.getKeys();
  if (!keys) throw new Error('No keys available');
```

```
  const ephemeralPubKey = this.fromBase64(ephemeralKey);

  const sharedSecret = sodium.crypto_scalarmult(
    keys.xPrivateKey,
    ephemeralPubKey
  );

  const encKey = sodium.crypto_generichash(32, sharedSecret);
  const cipher = this.fromBase64(encryptedData);
  const nonceBytes = this.fromBase64(nonce);
  const decrypted = sodium.crypto_secretbox_open_easy(cipher, nonceBytes, encKey);

  if (!decrypted) throw new Error('Failed to decrypt sender key');

  const bundle = JSON.parse(new TextDecoder().decode(decrypted));

  if (!this.receivedSenderKeys.has(groupId)) {
    this.receivedSenderKeys.set(groupId, new Map());
  }

  this.receivedSenderKeys.get(groupId)!.set(senderId, {
    chainKey: this.fromBase64(bundle.chainKey),
    generationId: bundle.generationId,
    chainIndex: 0,
    senderKeyId: bundle.senderKeyId,
  });
}

async encryptGroupMessage(groupId: string, plaintext: string): Promise<{
  ciphertext: string;
  nonce: string;
  senderKeyId: string;
  chainIndex: number;
}> {
  this.ensureReady();

  const state = this.senderKeys.get(groupId);
  if (!state) throw new Error('Sender key not initialized for group');

  const messageKey = sodium.crypto_generichash(
    32,
    new Uint8Array([...state.chainKey, state.chainIndex])
  );

  state.chainKey = sodium.crypto_generichash(32, state.chainKey);
  const currentIndex = state.chainIndex;
  state.chainIndex++;

  const nonce = sodium.randombytes_buf(sodium.crypto_secretbox_NONCEBYTES);
  const cipher = sodium.crypto_secretbox_easy(plaintext, nonce, messageKey);

  this.senderKeys.set(groupId, state);

  return {
    ciphertext: this.toBase64(cipher),
    nonce: this.toBase64(nonce),
    senderKeyId: state.senderKeyId,
    chainIndex: currentIndex,
```

```
  };
}

async decryptGroupMessage(
  groupId: string,
  senderId: string,
  ciphertext: string,
  nonce: string,
  senderKeyId: string,
  chainIndex: number
): Promise<string> {
  this.ensureReady();

  const groupKeys = this.receivedSenderKeys.get(groupId);
  if (!groupKeys) throw new Error('No keys for this group');

  const senderState = groupKeys.get(senderId);
  if (!senderState) throw new Error('No sender key from this user');

  if (senderState.senderKeyId !== senderKeyId) {
    throw new Error('Sender key ID mismatch - need to fetch new key');
  }

  const messageKey = sodium.crypto_generichash(
    32,
    new Uint8Array([...senderState.chainKey, chainIndex])
  );

  while (senderState.chainIndex <= chainIndex) {
    senderState.chainKey = sodium.crypto_generichash(32, senderState.chainKey);
    senderState.chainIndex++;
  }

  const cipher = this.fromBase64(ciphertext);
  const nonceBytes = this.fromBase64(nonce);
  const decrypted = sodium.crypto_secretbox_open_easy(cipher, nonceBytes, messageKey);

  if (!decrypted) throw new Error('Decryption failed');

  groupKeys.set(senderId, senderState);

  return new TextDecoder().decode(decrypted);
}

async rotateSenderKey(groupId: string): Promise<{
  chainKey: Uint8Array;
  senderKeyId: string;
  generationId: number;
}> {
  const oldState = this.senderKeys.get(groupId);
  const newGenerationId = oldState ? oldState.generationId + 1 : 0;

  const chainKey = sodium.randombytes_buf(32);
  const senderKeyId = this.generateUUID();
```

```
const state: SenderKeyState = {
      chainKey,
      generationId: newGenerationId,
      chainIndex: 0,
      senderKeyId,
    };

    this.senderKeys.set(groupId, state);

    return { chainKey, senderKeyId, generationId: newGenerationId };
  }

  exportRatchetState(contactId: string): string | null {
    const state = this.ratchetStates.get(contactId);
    if (!state) return null;

    return JSON.stringify({
      rootKey: this.toBase64(state.rootKey),
      sendingChainKey: this.toBase64(state.sendingChainKey),
      receivingChainKey: this.toBase64(state.receivingChainKey),
      sendMessageNumber: state.sendMessageNumber,
      recvMessageNumber: state.recvMessageNumber,
      dhRatchetPrivate: this.toBase64(state.dhRatchetPrivate),
      dhRatchetPublic: this.toBase64(state.dhRatchetPublic),
      remotePublicKey: state.remotePublicKey ? this.toBase64(state.remotePublicKey) : null,
      ratchetId: state.ratchetId || null,
    });
  }

  importRatchetState(contactId: string, stateJson: string): void {
    const data = JSON.parse(stateJson);
    this.ratchetStates.set(contactId, {
      rootKey: this.fromBase64(data.rootKey),
      sendingChainKey: this.fromBase64(data.sendingChainKey),
      receivingChainKey: this.fromBase64(data.receivingChainKey),
      sendMessageNumber: data.sendMessageNumber,
      recvMessageNumber: data.recvMessageNumber,
      dhRatchetPrivate: this.fromBase64(data.dhRatchetPrivate),
      dhRatchetPublic: this.fromBase64(data.dhRatchetPublic),
      remotePublicKey: data.remotePublicKey ? this.fromBase64(data.remotePublicKey) : undefined,
      ratchetId: data.ratchetId || undefined,
    });
  }

  exportSenderKey(groupId: string): string | null {
    const state = this.senderKeys.get(groupId);
    if (!state) return null;

    return JSON.stringify({
      chainKey: this.toBase64(state.chainKey),
      generationId: state.generationId,
      chainIndex: state.chainIndex,
      senderKeyId: state.senderKeyId,
    });
  }
```

```
importSenderKey(groupId: string, stateJson: string): void {
  const data = JSON.parse(stateJson);
  this.senderKeys.set(groupId, {
    chainKey: this.fromBase64(data.chainKey),
    generationId: data.generationId,
    chainIndex: data.chainIndex,
    senderKeyId: data.senderKeyId,
  });
}

private arraysEqual(a: Uint8Array, b: Uint8Array): boolean {
  if (a.length !== b.length) return false;
  for (let i = 0; i < a.length; i++) {
    if (a[i] !== b[i]) return false;
  }
  return true;
}

private generateUUID(): string {
  return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, (c) => {
    const r = Math.random() * 16 | 0;
    const v = c === 'x' ? r : (r & 0x3 | 0x8);
    return v.toString(16);
  });
}

async deriveSharedSecret(myXPrivate: Uint8Array, theirXPublic: Uint8Array) {
  this.ensureReady();
  const shared = sodium.crypto_scalarmult(myXPrivate, theirXPublic);
  return shared;
}

exportMessageKeyForUser(
  contactId: string,
  userPublicKey: Uint8Array
): {
  messageNumber: number;
  encryptedKey: string;
  chainKeySnapshot: string;
  ephemeralPublicKey: string;
} | null {

  const keys = this.messageKeysCache.get(contactId);

  if (!keys || keys.length === 0) {
    return null;
  }

  const lastKey = keys[keys.length - 1];
  const ephemeralKeyPair = sodium.crypto_box_keypair();

  const sharedSecret = sodium.crypto_scalarmult(
    ephemeralKeyPair.privateKey,
    userPublicKey
  );
```

```
  const encryptionKey = sodium.crypto_generichash(32, sharedSecret);
  const nonce = sodium.randombytes_buf(sodium.crypto_secretbox_NONCEBYTES);
  const encrypted = sodium.crypto_secretbox_easy(
    lastKey.messageKey,
    nonce,
    encryptionKey
  );


  const result = {
    messageNumber: lastKey.keyIndex,
    encryptedKey: this.toBase64(new Uint8Array([...nonce, ...encrypted])),
    chainKeySnapshot: this.toBase64(lastKey.chainKeySnapshot),
    ephemeralPublicKey: this.toBase64(ephemeralKeyPair.publicKey)
  };

  return result;
}

async importMessageKeyForUser(
  encryptedKeyData: string,
  ephemeralPublicKey: string,
  myPrivateKey: Uint8Array
): Promise<Uint8Array> {
  this.ensureReady();

  const ephemeralPubKey = this.fromBase64(ephemeralPublicKey);

  const sharedSecret = sodium.crypto_scalarmult(
    myPrivateKey,
    ephemeralPubKey
  );

  const encryptionKey = sodium.crypto_generichash(32, sharedSecret);
  const encryptedData = this.fromBase64(encryptedKeyData);

  const nonce = encryptedData.slice(0, sodium.crypto_secretbox_NONCEBYTES);
  const cipher = encryptedData.slice(sodium.crypto_secretbox_NONCEBYTES);

  const messageKey = sodium.crypto_secretbox_open_easy(cipher, nonce, encryptionKey);

  if (!messageKey) throw new Error('Failed to decrypt message key');

  return messageKey;
}

async decryptWithKey(
  cipher: Uint8Array,
  nonce: Uint8Array,
  messageKey: Uint8Array
): Promise<string> {
  this.ensureReady();

  const decrypted = sodium.crypto_secretbox_open_easy(cipher, nonce, messageKey);

  if (!decrypted) throw new Error('Decryption failed');
```

```
    return new TextDecoder().decode(decrypted);
  }
}
```

AppDbContext.cs

```
using Microsoft.EntityFrameworkCore;

namespace MessagingSystem.Services.User.Persistence;

public class AppDbContext(DbContextOptions<AppDbContext> options) : DbContext(options)
{
    public DbSet<Core.User.User> Users { get; init; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);
        modelBuilder.Entity<Core.User.User>(builder =>
        {
            builder.Property(u => u.Login)
                .HasMaxLength(80)
                .IsRequired();

            builder.Property(u => u.NickName)
                .HasMaxLength(80)
                .IsRequired();

            builder.Property(u => u.HashLogin)
                .HasMaxLength(120)
                .IsRequired();

            builder.Property(u => u.HashEmail)
                .HasMaxLength(120)
                .IsRequired();

            builder.Property(u => u.HashNickName)
                .HasMaxLength(120)
                .IsRequired();
            builder.Property(u => u.Image)
                .HasMaxLength(500);
        });
        modelBuilder.Entity<Core.User.User>()
            .HasIndex(u => u.HashLogin)
            .IsUnique();
        modelBuilder.Entity<Core.User.User>()
            .HasIndex(r => r.HashEmail)
            .IsUnique();
        modelBuilder.Entity<Core.User.User>()
            .HasIndex(r => r.HashNickName)
            .IsUnique();
    }
}
```

GroupAppDbContext.cs

```
using MessagingSystem.Services.Messaging.Core.Groups.Group;
using MessagingSystem.Services.Messaging.Core.Groups.GroupMember;
using MessagingSystem.Services.Messaging.Core.Groups.GroupMessages;
using Microsoft.EntityFrameworkCore;

namespace MessagingSystem.Services.Messaging.Persistence.Group;

public class GroupAppDbContext(DbContextOptions<GroupAppDbContext> options) : DbContext(options)
{
    public DbSet<GroupInfo> GroupInfos { get; init; }
    public DbSet<GroupMessage> GroupMessages { get; init; }
    public DbSet<GroupMembers> GroupMembers { get; init; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<GroupMessage>(builder =>
        {
            builder.Property(u => u.Sender)
                .HasMaxLength(120)
                .IsRequired();
            builder.Property(u => u.Content)
                .HasMaxLength(20000)
                .IsRequired();
            builder.Property(u => u.SenderHash)
                .HasMaxLength(120)
                .IsRequired();
        });

        modelBuilder.Entity<GroupInfo>(builder =>
        {
            builder.Property(u => u.GroupName)
                .HasMaxLength(500)
                .IsRequired();
            builder.Property(u => u.Image)
                .HasMaxLength(900);
            builder.Property(u => u.Description)
                .HasMaxLength(600);
            builder.Property(u => u.Admin)
                .HasMaxLength(80)
                .IsRequired();
            builder.Property(u => u.AdminHash)
                .HasMaxLength(120)
                .IsRequired();
            builder.Property(u => u.GroupNameHash)
                .HasMaxLength(200)
                .IsRequired();
        });
```

```
        modelBuilder.Entity<GroupMembers>(builder =>
        {
            builder.Property(u => u.UserNickName)
                .HasMaxLength(120)
                .IsRequired();
            builder.Property(u => u.UserNickNameHash)
                .HasMaxLength(120)
                .IsRequired();
            builder.Property(u => u.Image)
                .HasMaxLength(600);
        });

        modelBuilder.Entity<GroupMessage>()
            .HasOne(gm => gm.Group)
            .WithMany()
            .HasForeignKey(gm => gm.GroupId);

        modelBuilder.Entity<GroupInfo>()
            .HasIndex(u => u.Id)
            .IsUnique();
        modelBuilder.Entity<GroupInfo>()
            .HasIndex(u => u.GroupNameHash)
            .IsUnique();
        modelBuilder.Entity<GroupInfo>()
            .Property(g => g.RowVersion)
            .IsRowVersion()
            .IsConcurrencyToken();

        modelBuilder.Entity<GroupInfo>()
            .HasMany(g => g.Members)
            .WithOne(m => m.Group)
            .HasForeignKey(m => m.GroupId)
            .OnDelete(DeleteBehavior.Cascade);
    }
}
```

OtoAppDbContext.cs

```
using MessagingSystem.Services.Messaging.Core.Oto.OtoMessages;
using MessagingSystem.Services.Messaging.Core.Oto.Users;
using Microsoft.EntityFrameworkCore;

namespace MessagingSystem.Services.Messaging.Persistence.Oto;

public class OtoAppDbContext(DbContextOptions<OtoAppDbContext> options) : DbContext(options)
{
    public DbSet<Message> UsersMessages { get; init; }
    public DbSet<UserImage> Images { get; init; }

  protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);
```

```
        modelBuilder.Entity<Message>(builder =>
        {
            builder.Property(u => u.Sender)
                .HasMaxLength(120)
                .IsRequired();
            builder.Property(u => u.Recipient)
                .HasMaxLength(120)
                .IsRequired();
            builder.Property(u => u.Content)
                .HasMaxLength(20000)
                .IsRequired();
            builder.Property(u => u.RecipientHash)
                .HasMaxLength(120)
                .IsRequired();
            builder.Property(u => u.SenderHash)
                .HasMaxLength(120)
                .IsRequired();
        });

        modelBuilder.Entity<Message>()
            .HasIndex(u => u.Id);
        modelBuilder.Entity<Message>()
            .HasIndex(u => u.RecipientHash);
        modelBuilder.Entity<Message>()
            .HasIndex(u => u.SenderHash);

        modelBuilder.Entity<UserImage>(builder =>
        {
            builder.Property(u => u.NickNameHash)
                .HasMaxLength(120)
                .IsRequired();
            builder.Property(u => u.Image)
                .HasMaxLength(500)
                .IsRequired();
        });

        modelBuilder.Entity<UserImage>()
            .HasIndex(u => u.Id);
        modelBuilder.Entity<UserImage>()
            .HasIndex(u => u.NickNameHash)
            .IsUnique();
    }
}
```