НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

«До захисту допущено»
Завідувач кафедри
_____ Наталія АУШЕВА
“___” _____2026 р.

Дипломна робота
на здобуття ступеня бакалавр

За освітньою програмою “Цифрові технології в енергетиці”

Спеціальності 122 “Комп’ютерні науки”

на тему: “Інформаційна система корпоративного захищеного обміну повідомленнями”

Виконав: студент 4 курсу, групи ТР-23

Корнійко Максим Васильович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник асистент Олександр ВОЛКОВ _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Рецензент доктор філософії, ст.викладач Ольга ВЛАСЕНКО _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Н.контроль асистент Артем МЕЛЬНИЧЕНКО _____
(посада, ім’я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ - 2026

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)
спеціальність 122 “Комп’ютерні науки”
Освітньо-професійна програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА
(підпис)
“___” _____2026 р.

ЗАВДАННЯ
на дипломну роботу студенту

Корнійку Максиму Васильовичу
(прізвище, ім’я, по батькові)

1. Тема роботи “Інформаційна система корпоративного захищеного обміну повідомленнями”
Науковий керівник Волков Олександр Володимирович, асистент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від “_28_” травня 2026 року № _1992_
2. Термін подання студентом роботи 04.06.2026
3. Вихідні дані до роботи мова програмування Kotlin, серверний фреймворк Spring Boot 4.0, клієнтський фреймворк Kotlin Multiplatform та Compose Multiplatform, СУБД PostgreSQL, брокер повідомлень RabbitMQ, сховище даних Redis, середовище розробки IntelliJ IDEA Ultimate, контейнеризація Docker та Docker Compose
4. Перелік питань, які потрібно розробити:

1) провести аналіз існуючих рішень для корпоративного обміну повідомленнями та побудувати модель загроз інформаційній безпеці за методологією STRIDE
2) визначити функціональні та нефункціональні вимоги до системи з урахуванням вимог інформаційної безпеки та self-hosted розгортання
3) спроектувати мікросервісну архітектуру серверної частини та модульну архітектуру мультиплатформенного клієнтського застосунку
4) реалізувати багаторівневу систему захисту інформації що включає TLS 1.3, JWT автентифікацію, BCrypt хешування паролів, rate limiting через Redis Lua-скрипти та наскрізне шифрування на основі Signal Protocol
5) розробити мультиплатформенний клієнтський застосунок для платформ Android та Desktop з offline-first стратегією кешування
6) провести тестування механізмів безпеки та функціональності системи

6. Дата видачі завдання 15.09.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вибір теми роботи	20.04.26	Виконано
2.	Аналіз існуючих рішень та побудова моделі загроз	21-22.04.26	Виконано
3.	Розробка архітектури та загальної структури системи	20-26.04.26	Виконано
4.	Розробка клієнтського застосунку	27.04.-03.05.26	Виконано
5.	Тестування системи	11-17.05.26	Виконано
6.	Оформлення пояснювальної записки	14-16.05.26	Виконано
7.	Захист програмного забезпечення	15.02.2026	Виконано
8.	Передзахист	03.06.2026	Виконано
9.	Захист	17.06.2026	Виконано

Студент _____ Корнійко М. В.
(підпис) (прізвище та ініціали)

Керівник _____ Волков О. В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота виконана на 84 сторінках, містить 5 таблиць, 14 рисунків, 1 додаток та 38 джерел у переліку посилань.

Мета роботи - розробка інформаційної системи корпоративного захищеного обміну повідомленнями з self-hosted архітектурою що забезпечує повний контроль підприємства над власними даними та багаторівневий захист інформації.

Методи та засоби: використано мову програмування Kotlin, серверний фреймворк Spring Boot 4.0, технологію Kotlin Multiplatform та Compose Multiplatform для розробки клієнтського застосунку, реляційну базу даних PostgreSQL, сховище даних Redis, брокер повідомлень RabbitMQ, протокол Signal (X3DH + Double Ratchet) для наскрізного шифрування повідомлень та Docker Compose для контейнеризації та розгортання системи.

В результаті роботи розроблено інформаційну систему що складається з мікросервісної серверної частини та мультиплатформенного клієнтського застосунку для Android та Desktop. Система реалізує захист транспортного рівня через TLS 1.3, JWT автентифікацію з двома токенами, BCrypt хешування паролів, дворівневий rate limiting через Redis Lua-скрипти та наскрізне шифрування повідомлень. Система розгортається на власній інфраструктурі підприємства через Docker Compose без залежності від зовнішніх хмарних сервісів.

ANNOTATION

The thesis is completed on 84 pages, contains 5 tables, 14 images,1 appendice, 38 sources in the list of references.

The purpose of the work is to develop a corporate secure messaging information system with self-hosted architecture that ensures complete enterprise control over its own data and multi-layered information protection.

The implementation process used the Kotlin programming language, Spring Boot 4.0 server-side framework, Kotlin Multiplatform and Compose Multiplatform technologies for client application development, PostgreSQL relational database, Redis data store, RabbitMQ message broker, Signal Protocol (X3DH + Double Ratchet) for end-to-end message encryption and Docker Compose for containerization and system deployment.

As a result of the work, an information system was developed consisting of a microservices server-side and a multiplatform client application for Android and Desktop. The system implements transport-level security via TLS 1.3, JWT authentication with dual tokens, BCrypt password hashing, two-level rate limiting via Redis Lua scripts and end-to-end message encryption. The system is deployed on the enterprise's own infrastructure via Docker Compose without dependency on external cloud services.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AEAD - Authenticated Encryption with Associated Data (автентифіковане шифрування з асоційованими даними)

AES - Advanced Encryption Standard (розширений стандарт шифрування)

AMQP - Advanced Message Queuing Protocol (розширений протокол черги повідомлень)

API - Application Programming Interface (програмний інтерфейс застосунку)

ASIC - Application-Specific Integrated Circuit (інтегральна схема спеціального призначення)

BCrypt - адаптивна хеш-функція для захисту паролів

CA - Certificate Authority (центр сертифікації)

CIDR - Classless Inter-Domain Routing (безкласова міждоменна маршрутизація)

CPU - Central Processing Unit (центральний процесор)

DAO - Data Access Object (об'єкт доступу до даних)

DH - Diffie-Hellman (криптографічний протокол обміну ключами)

DoS - Denial of Service (відмова в обслуговуванні)

DTO - Data Transfer Object (об'єкт передачі даних)

E2E - End-to-End (наскрізний)

ECDH - Elliptic Curve Diffie-Hellman (протокол Діффі-Хеллмана на еліптичних кривих)

ECDHE - Elliptic Curve Diffie-Hellman Ephemeral (ефемерний протокол Діффі-Хеллмана на еліптичних кривих)

GCM - Galois/Counter Mode (режим лічильника Галуа)

GDPR - General Data Protection Regulation (Загальний регламент про захист даних)

GPU - Graphics Processing Unit (графічний процесор)

HKDF - HMAC-based Key Derivation Function (функція виведення ключів на основі HMAC)

HMAC - Hash-based Message Authentication Code (код автентифікації повідомлень на основі хешування)

HTTP - HyperText Transfer Protocol (протокол передачі гіпертексту)

HTTPS - HyperText Transfer Protocol Secure (захищений протокол передачі гіпертексту)

IK - Identity Key (ключ ідентичності)

JPA - Java Persistence API (програмний інтерфейс збереження даних Java)

JSON - JavaScript Object Notation (текстовий формат обміну даними)

JWT - JSON Web Token (веб-токен JSON)

KDF - Key Derivation Function (функція виведення ключів)

KMP - Kotlin Multiplatform (мультиплатформенна технологія Kotlin)

KSP - Kotlin Symbol Processing (обробка символів Kotlin)

MCC - Message Confidentiality and Integrity (конфіденційність та цілісність повідомлень)

MITM - Man-In-The-Middle (атака «людина посередині»)

MVI - Model-View-Intent (архітектурний патерн)

NIST - National Institute of Standards and Technology (Національний інститут стандартів і технологій США)

OPK - One-Time PreKey (одноразовий ключ попереднього обміну)

OWASP - Open Web Application Security Project (відкритий проєкт безпеки вебзастосунків)

PBKDF2 - Password-Based Key Derivation Function 2 (функція виведення ключів на основі пароля)

REST - Representational State Transfer (архітектурний стиль вебсервісів)

RFC - Request for Comments (запит на коментарі - стандарт мережевих технологій)

RTT - Round-Trip Time (час одного циклу передачі даних)

SIEM - Security Information and Event Management (управління інформацією та подіями безпеки)

SPK - Signed PreKey (підписаний ключ попереднього обміну)

SQL - Structured Query Language (мова структурованих запитів)

STRIDE - Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege (методологія моделювання загроз)

TLS - Transport Layer Security (протокол захисту транспортного рівня)

TOTP - Time-based One-Time Password (одноразовий пароль на основі часу)

TTL - Time To Live (час існування запису)

UI - User Interface (інтерфейс користувача)

URL - Uniform Resource Locator (уніфікований локатор ресурсів)

UUID - Universally Unique Identifier (універсальний унікальний ідентифікатор)

WebSocket - протокол двостороннього зв'язку між клієнтом та сервером

WSS - WebSocket Secure (захищений протокол WebSocket)

X3DH - Extended Triple Diffie-Hellman (розширений потрійний протокол Діффі-Хеллмана)

БД - база даних

ДП - дипломний проєкт

ПЗ - програмне забезпечення

ВСТУП

Стрімкий розвиток цифрових технологій та масовий перехід підприємств до дистанційного формату роботи зумовили критичне зростання обсягів корпоративної комунікації через цифрові канали. Більшість популярних рішень - Telegram, Viber, WhatsApp, Slack - зберігають корпоративні дані на серверах третіх сторін поза юрисдикцією підприємства, що створює ризики витоку конфіденційної інформації та порушення вимог GDPR і законодавства України про захист персональних даних. Відсутність на ринку рішень що одночасно забезпечують self-hosted розгортання, наскрізне шифрування та повноцінний корпоративний функціонал підтверджує актуальність даної роботи.

Мета роботи - розробка інформаційної системи корпоративного захищеного обміну повідомленнями з архітектурою self-hosted що забезпечує повний контроль підприємства над власними даними та багаторівневий захист інформації.

Для досягнення мети вирішено такі задачі: проведено аналіз предметної області та побудовано модель загроз за методологією STRIDE; спроектовано мікросервісну архітектуру серверної частини; реалізовано систему захисту що включає TLS 1.3, JWT автентифікацію, BCrypt хешування паролів, rate limiting через Redis Lua-скрипти та наскрізне шифрування на основі Signal Protocol; розроблено мультиплатформенний клієнтський застосунок для Android та Desktop; проведено тестування механізмів безпеки та функціональності.

Об'єктом дослідження є процес захищеного корпоративного обміну повідомленнями. Предметом дослідження є методи та механізми захисту інформації в системах корпоративного обміну повідомленнями з self-hosted архітектурою.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

У першому розділі проведено аналіз проблематики забезпечення інформаційної безпеки корпоративних комунікацій, класифіковано основні загрози для систем обміну повідомленнями, виконано порівняльний аналіз існуючих рішень та сформульовано вимоги до розроблюваної системи.

1.1. Проблематика корпоративної інформаційної безпеки

В умовах цифрової трансформації підприємств обмін повідомленнями між співробітниками став невід'ємною складовою робочого процесу. За даними досліджень, понад 85% підприємств використовують хмарні месенджери для внутрішньої комунікації, передаючи через них конфіденційну інформацію - комерційні таємниці, персональні дані клієнтів, фінансову звітність та стратегічні плани. Водночас більшість популярних рішень зберігають дані користувачів на серверах третіх сторін, які можуть знаходитись поза юрисдикцією країни підприємства, що суперечить вимогам законодавства України та міжнародних стандартів захисту даних.

Окрему загрозу становить промислове шпигунство. За даними Ponemon Institute, середня вартість витоку корпоративних даних у 2023 році склала 4,45 мільйона доларів США, причому значна частка інцидентів пов'язана з компрометацією каналів комунікації. Використання незахищених або стороніх месенджерів для передачі конфіденційної інформації є одним з ключових векторів атак на корпоративну інфраструктуру.

Ситуацію додатково ускладнює те, що адміністратори підприємств позбавлені будь-якого контролю над обліковими записами співробітників у

хмарних месенджерах. У разі звільнення працівника або компрометації його облікового запису підприємство не має технічної можливості оперативно заблокувати доступ до корпоративних переписок або провести аудит комунікацій.

1.2. Класифікація загроз інформаційній безпеці месенджерів

Для побудови ефективної системи захисту необхідно чітко визначити та класифікувати загрози, яким піддається система обміну повідомленнями. У даному підрозділі загрози класифіковано за рівнями моделі OSI та характером впливу на систему.

Загрози транспортного рівня є одними з найбільш поширених та небезпечних. Атака типу «людина посередині» (MITM) передбачає перехоплення мережевого трафіку між клієнтом та сервером з метою читання або модифікації повідомлень. За відсутності шифрування транспортного рівня зловмисник, який отримав доступ до мережевої інфраструктури підприємства, може безперешкодно читати всі повідомлення. Пасивне прослуховування мережі (sniffing) дозволяє зловмиснику накопичувати зашифрований трафік для подальшого дешифрування при отриманні ключів - саме проти цього спрямований механізм Forward Secrecy.

Загрози рівня автентифікації включають атаки перебором паролів (brute force), при яких зловмисник систематично перевіряє комбінації облікових даних. Атаки типу credential stuffing використовують бази викрадених логінів та паролів з інших сервісів, оскільки користувачі часто використовують однакові паролі на різних платформах. Атаки на токени автентифікації (token hijacking) спрямовані на викрадення або підробку JWT токенів для отримання несанкціонованого доступу.

Загрози рівня застосунку охоплюють широкий спектр вразливостей. Ін'єкційні атаки (SQL injection, NoSQL injection) спрямовані на маніпуляцію запитами до бази даних. Атаки Cross-Site WebSocket Hijacking (CSWSH) специфічні для WebSocket протоколу і дозволяють зловмиснику встановити

несанкціоноване WebSocket з'єднання від імені автентифікованого користувача. Атаки на бізнес-логіку використовують недоліки авторизаційних перевірок для отримання доступу до чужих повідомлень або чатів.

Загрози рівня даних стосуються несанкціонованого доступу до збережених даних. Витік бази даних може розкрити вміст повідомлень, хеші паролів та особисті дані користувачів. Відсутність наскрізного шифрування означає, що адміністратор сервера або особа, яка отримала доступ до бази даних, може читати вміст повідомлень. Інсайдерські загрози з боку привілейованих користувачів системи також становлять значний ризик для корпоративного середовища.

Загрози доступності системи включають атаки типу «відмова в обслуговуванні» (DoS/DDoS), масову реєстрацію фіктивних облікових записів та зловживання функціями надсилання електронних листів для перевантаження поштового сервера.

1.3. Огляд та порівняльний аналіз існуючих рішень

Перед розробкою власної системи проведено детальний аналіз існуючих рішень для корпоративного та захищеного обміну повідомленнями. Критеріями аналізу обрано характеристики, що безпосередньо впливають на рівень безпеки та придатність для корпоративного використання: наявність наскрізного шифрування, можливість self-hosted розгортання та рівень контролю.

Telegram є одним з найпопулярніших месенджерів в Україні з широким функціоналом, включаючи групові чати, канали, боти та відкритий API. Протокол MTProto забезпечує шифрування при передачі, однак стандартні чати не використовують наскрізне шифрування за замовчуванням - воно доступне лише у режимі «Секретний чат» для приватного листування.

Viber використовує наскрізне шифрування для всіх видів комунікації на основі протоколу Signal. Проте застосунок є виключно хмарним рішенням з

централізованою інфраструктурою компанії Rakuten, що позбавляє підприємства будь-якої можливості контролю над даними. Відсутність корпоративного функціоналу та неможливість self-hosted розгортання роблять Viber непридатним для захищеної корпоративної комунікації.

WhatsApp реалізує наскрізне шифрування на основі Signal Protocol для всіх повідомлень, що є значною перевагою з точки зору конфіденційності вмісту. Однак WhatsApp належить компанії Meta і зберігає метадані повідомлень - інформацію про те, хто, коли і з ким спілкувався, - на власних серверах. Ці метадані є цінною інформацією для корпоративного шпигунства навіть без доступу до вмісту повідомлень. Відсутність можливості розгортання на власній інфраструктурі є критичним недоліком.

Slack є повнофункціональним корпоративним інструментом командної роботи з підтримкою інтеграцій, робочих просторів та розширеного адміністрування. Однак Slack не забезпечує наскрізного шифрування - повідомлення зберігаються у відкритому вигляді на серверах компанії та доступні для читання адміністраторами Slack.

Signal забезпечує найвищий рівень захисту серед розглянутих рішень завдяки реалізації власного протоколу Signal, який став стандартом де-факто для захищеної комунікації. Проте Signal орієнтований на індивідуальне використання і не надає повноцінного корпоративного функціоналу.

Порівняльний аналіз розглянутих рішень наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика існуючих рішень

Критерій	Telegram	Viber	WhatsApp	Stack	Signal	Розроблювана система
Наскрізне шифрування	Частково	Так	Так	Так	Так	Так
Self-hosted	Ні	Ні	Ні	Ні	Ні	Так

Продовження таблиці 1.1

Контроль адміністратора	Частково	Ні	Ні	Частково	Ні	Повний
Корпоративний функціонал	Частково	Ні	Ні	Так	Ні	Так

За результатами аналізу встановлено, що жодне з розглянутих рішень не задовольняє одночасно всім вимогам корпоративної безпеки - можливості self-hosted розгортання, повного контролю адміністратора над даними та комплексного корпоративного функціоналу. Це підтверджує актуальність та доцільність розробки власної системи.

1.4. Постановка задачі

На основі проведеного аналізу предметної області, класифікації загроз та огляду існуючих рішень сформульовано вимоги до розроблюваної інформаційної системи корпоративного захищеного обміну повідомленнями.

Функціональні вимоги визначають перелік можливостей, які має надавати система користувачам:

- реєстрація користувачів з обов'язковим підтвердженням електронної пошти для запобігання реєстрації фіктивних облікових записів;
- автентифікація на основі JWT з підтримкою схеми access та refresh токенів;
- створення та управління особистими чатами між двома користувачами;
- створення та управління груповими чатами з можливістю додавання та видалення учасників;
- надсилання та отримання текстових повідомлень у реальному часі через WebSocket з'єднання;
- видалення повідомлень та чатів користувачем;

- скидання пароля через електронну пошту.

Нефункціональні вимоги визначають якісні характеристики системи. З точки зору безпеки система має забезпечувати захист транспортного рівня через TLS 1.3 для всіх з'єднань, захист від brute force атак через механізм rate limiting на рівні IP-адреси та електронної пошти, безпечне зберігання паролів з використанням адаптивного хешування BCrypt та управління сесіями на основі JWT з можливістю примусової інвалідації токенів. З точки зору розгортання система має підтримувати self-hosted архітектуру та розгортатись на власній інфраструктурі підприємства через Docker Compose. З точки зору мультиплатформенності клієнтський застосунок має підтримувати платформи Android та Desktop з єдиної кодової бази. Система має забезпечувати масштабованість через мікросервісну архітектуру та незалежне масштабування компонентів.

У першому розділі проведено аналіз предметної області корпоративного захищеного обміну повідомленнями. Встановлено, що зростання кількості кіберзагроз та посилення вимог законодавства до захисту персональних даних формують стійкий попит на корпоративні рішення з повним контролем над інфраструктурою. Класифіковано загрози безпеці за рівнями - від транспортного до рівня даних - що стало основою для проектування багаторівневої системи захисту. Порівняльний аналіз п'яти популярних рішень підтвердив, що жодне з них не задовольняє одночасно вимогам self-hosted розгортання, наскрізного шифрування та повноцінного корпоративного функціоналу. Сформульовані функціональні та нефункціональні вимоги стали основою для проектування архітектури системи, що розглядається у наступному розділі.

2. АРХІТЕКТУРА СИСТЕМИ

У другому розділі описано загальну архітектуру розробленої системи, обґрунтовано ключові архітектурні рішення, детально розглянуто структуру серверної та клієнтської частин, організацію зберігання даних та безпеку на архітектурному рівні.

2.1. Загальна архітектура системи

Розроблена система побудована за мікросервісною архітектурою і складається з двох основних частин: серверної та клієнтської. Вибір мікросервісної архітектури обумовлений кількома факторами. По-перше, незалежне масштабування компонентів дозволяє збільшувати ресурси окремих сервісів відповідно до навантаження без впливу на інші. По-друге, чітке розмежування відповідальності між сервісами спрощує підтримку та розвиток системи. По-третє, ізоляція сервісів підвищує безпеку системи - компрометація одного сервісу не надає автоматичного доступу до інших.

Архітектура системи, що зображена на рисунку 2.1, передбачає такий принцип розгортання: всі компоненти системи запускаються у Docker-контейнерах на власній інфраструктурі підприємства через Docker Compose. Nginx виступає як reverse proxy та TLS термінатор - він приймає всі зовнішні з'єднання, забезпечує HTTPS та WSS шифрування і передає трафік до відповідних внутрішніх сервісів. Зовні доступний лише порт 443 (HTTPS/WSS), всі інші компоненти знаходяться у внутрішній Docker мережі та недоступні ззовні. Це є принципово важливим архітектурним рішенням з точки зору безпеки - PostgreSQL, Redis та RabbitMQ не мають прямого доступу з мережі підприємства.

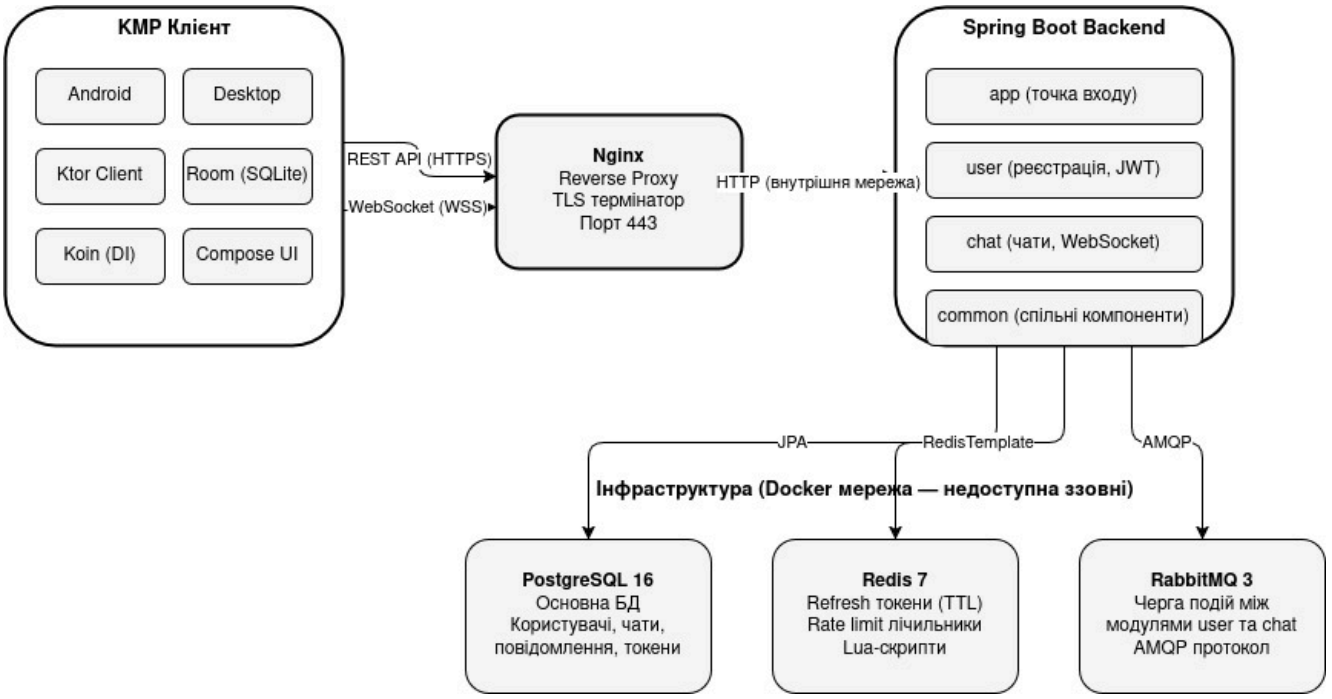

Рисунок 2.1 - Загальна взаємодія компонентів системи

Клієнтський застосунок взаємодіє з серверною частиною через два канали: REST API для стандартних операцій (реєстрація, автентифікація, управління чатами) та WebSocket для обміну повідомленнями в реальному часі. Обидва канали захищені TLS шифруванням через Nginx.

2.2. Архітектура серверної частини

Серверна частина організована як мультимодульний Gradle-проєкт і складається з чотирьох основних модулів, що зображені на рисунку 2.2,: app, user, chat та common. Для стандартизації конфігурації збірки використовується модуль build-logic з трьома convention plugins.

Кожен функціональний модуль організований за шаровою архітектурою з поділом на шари api, service, domain та infra. Такий поділ забезпечує принцип

інверсії залежностей - бізнес-логіка у шарі domain не залежить від деталей реалізації інфраструктури, що спрощує тестування та заміну окремих компонентів.

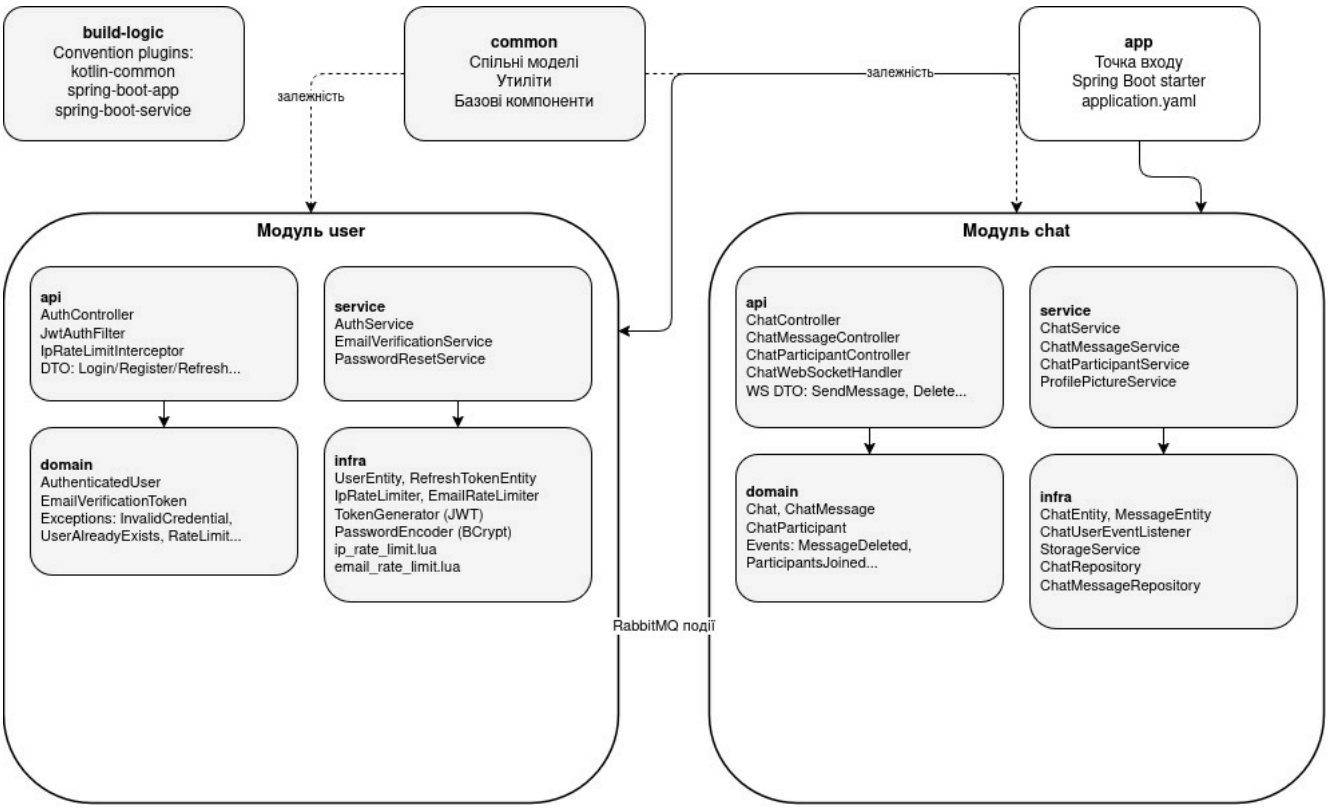

Рисунок 2.2 - Структура backend мікросервісів

Шар api є зовнішнім інтерфейсом модуля і містить контролери, що обробляють HTTP-запити, об'єкти передачі даних (DTO), обробники винятків та маппери. Важливою складовою шару api є компоненти безпеки - фільтр JwtAuthFilter та перехоплювач IpRateLimitInterceptor, які перевіряють кожен вхідний запит до того як він досягне бізнес-логіки.

Шар domain містить бізнес-моделі та доменні винятки, що є незалежними від конкретної реалізації інфраструктури. Визначення доменних винятків - InvalidCredentialException, UserAlreadyExistsException, EmailNotVerifiedException,

RateLimitException - на цьому рівні дозволяє формувати зрозумілі відповіді про помилки без розкриття деталей реалізації.

Шар infra відповідає за взаємодію із зовнішніми системами: базою даних PostgreSQL через JPA-сутності та репозиторії Spring Data, Redis через компоненти EmailRateLimiter, IpRateLimiter та TokenGenerator, а також файловим сховищем через StorageService. Ізоляція інфраструктурного коду у цьому шарі є важливою з точки зору безпеки - деталі підключення до бази даних та Redis доступні лише компонентам цього шару.

Модуль user реалізує управління користувачами та містить сервіси AuthService, EmailVerificationService та PasswordResetService. Модуль chat реалізує основну функціональність обміну повідомленнями через сервіси ChatService, ChatMessageService, ChatParticipantService та ProfilePictureService, а також обробник WebSocket з'єднань ChatWebSocketHandler. Взаємодія між модулями здійснюється через RabbitMQ - при зміні даних користувача (наприклад, оновлення фото профілю) модуль user публікує подію, яку обробляє ChatUserEventListener у модулі chat.

2.3. Архітектура клієнтської частини

Клієнтська частина реалізована як мультиплатформенний проєкт на основі Kotlin Multiplatform і організована за принципами чистої архітектури. Проєкт поділено на дві групи модулів, що зображені на рисунку 2.3: core для спільних компонентів та feature для функціональних можливостей.

Група core містить чотири модулі. Модуль core/domain визначає бізнес-сутності та інтерфейси репозиторіїв, що є незалежними від платформи. Модуль core/data реалізує мережеву взаємодію через Ktor Client та містить платформо-специфічні реалізації для Android, iOS та Desktop. Модуль

core/presentation містить базові UI-компоненти та базові класи ViewModel. Модуль core/designsystem визначає єдину дизайн-систему застосунку.

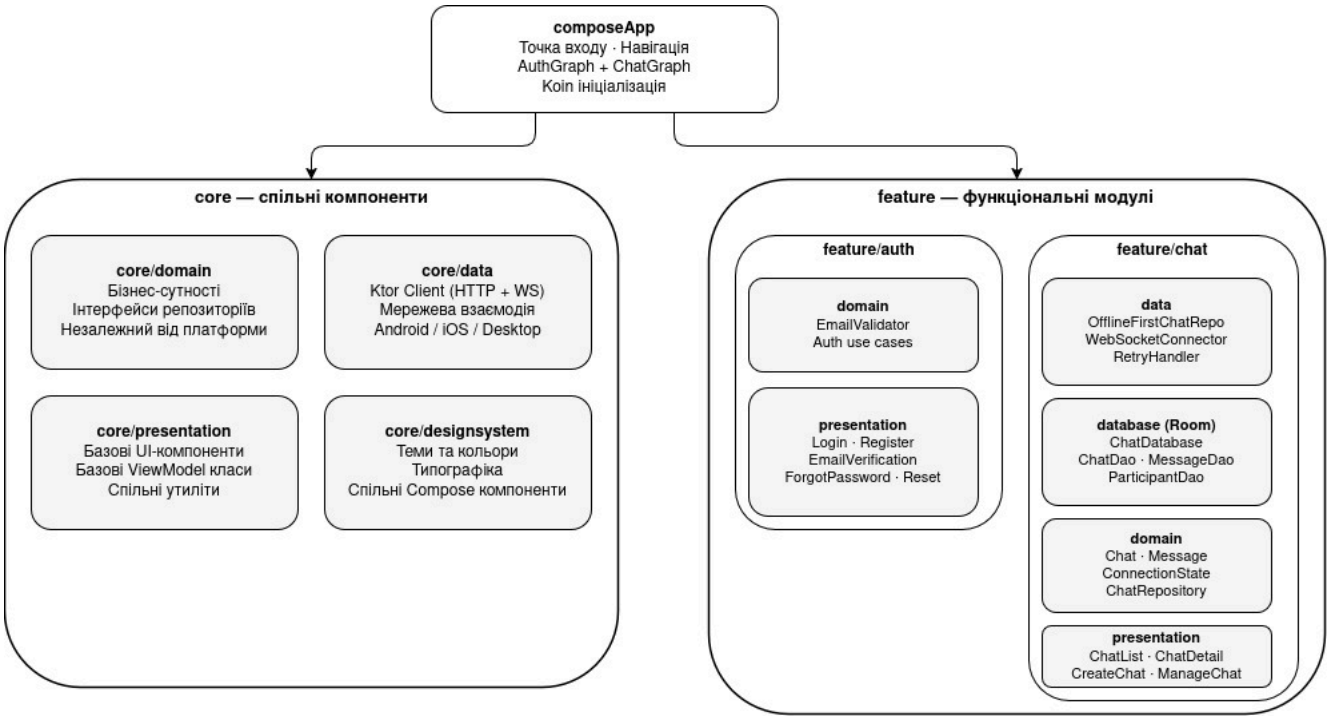

Рисунок 2.3 - Модульна структура клієнтського застосунку

Група feature організована за функціональними можливостями. Модуль feature/auth реалізує автентифікацію та складається з підмодулів domain і presentation. Модуль feature/chat є найбільшим і складається з чотирьох підмодулів: data для мережевої взаємодії та локального кешування, database для роботи з Room, domain для бізнес-логіки та presentation для інтерфейсу.

Кожен екран клієнтського застосунку реалізований за патерном MVI з окремими класами для стану, дій та подій. Такий підхід забезпечує однонаправлений потік даних, що спрощує відлагодження та тестування.

Особливу роль з точки зору безпеки відіграє управління WebSocket з'єднанням. Клас KtorWebSocketConnector відповідає за встановлення з'єднання з передачею JWT токена при підключенні. ConnectionRetryHandler автоматично

відновлює з'єднання при розриві, а ConnectivityObserver з платформо-специфічними реалізаціями відстежує стан мережевого з'єднання. AppLifecycleObserver коректно розриває WSS з'єднання при переході застосунку у фоновий режим, запобігаючи тримання відкритих з'єднань без необхідності.

2.4. База даних та схема зберігання даних

Система використовує три різних сховища даних, кожне з яких оптимізоване під конкретні задачі.

PostgreSQL є основним реляційним сховищем і містить таблиці користувачів, токенів верифікації, токенів скидання пароля, refresh токенів, чатів, повідомлень та учасників чатів. Схема бази даних спроектована з урахуванням вимог безпеки - паролі зберігаються виключно у хешованому вигляді, токени підтвердження та скидання пароля мають обмежений термін дії, а refresh токени прив'язані до конкретного користувача. Доступ до PostgreSQL має лише шар infra серверної частини - пряме підключення до бази даних ззовні унеможливлено на рівні Docker мережевої конфігурації.

Redis використовується для двох принципово різних задач. По-перше, зберігання refresh токенів з автоматичним видаленням після закінчення терміну дії через механізм TTL. Це забезпечує автоматичне очищення прострочених сесій без додаткових фонових процесів. По-друге, зберігання лічильників rate limiting з ключами виду ip:{адреса} та email:{адреса} з відповідними TTL значеннями. Lua-скрипти, що виконуються безпосередньо на Redis, забезпечують атомарність операцій перевірки та інкременту лічильника, що унеможливлює обхід rate limiting через паралельні запити.

Room використовується на клієнтській стороні для локального кешування даних та реалізації offline-first стратегії. Усі методи DAO повертають Kotlin Flow, що забезпечує реактивне оновлення інтерфейсу. DatabaseFactory має

платформо-специфічні реалізації для Android та Desktop. З точки зору безпеки локальна база даних не містить чутливих автентифікаційних даних - JWT токени зберігаються у захищеному сховищі платформи.

2.5. Безпека на архітектурному рівні

Архітектурні рішення системи самі по собі формують перший рівень захисту ще до застосування спеціалізованих механізмів безпеки.

Принцип мінімальних привілеїв реалізовано на кількох рівнях. На рівні мікросервісів кожен модуль має доступ лише до тих компонентів інфраструктури, які необхідні для його роботи.

Принцип ешелонованого захисту (Defense in Depth) передбачає що для отримання доступу до захищених даних зловмисник має подолати кілька незалежних рівнів захисту: мережевий рівень (Nginx, TLS), рівень автентифікації (JWT), рівень авторизації (Spring Security) та рівень бізнес-логіки (перевірки у сервісах).

Ізоляція компонентів у Docker мережі унеможливлює прямий доступ до PostgreSQL, Redis та RabbitMQ навіть з середини корпоративної мережі. Це суттєво обмежує можливості зловмисника, який отримав часткowy доступ до мережі підприємства.

У другому розділі спроектовано та описано архітектуру інформаційної системи корпоративного захищеного обміну повідомленнями. Обґрунтовано вибір мікросервісної архітектури з точки зору масштабованості та безпеки. Детально описано шарову організацію серверних модулів та модульну структуру KMP клієнта. Описано схему зберігання даних у трьох сховищах - PostgreSQL, Redis та Room - з урахуванням вимог безпеки до кожного з них. Детальний опис механізмів захисту інформації наведено у наступному розділі.

3. МЕХАНІЗМИ ЗАХИСТУ ІНФОРМАЦІЇ

У третьому розділі детально розглянуто всі механізми забезпечення інформаційної безпеки розробленої системи. Описано модель загроз, механізми захисту транспортного рівня, автентифікації та управління сесіями, захисту паролів, захисту від автоматизованих атак, наскрізного шифрування повідомлень та аудиту безпеки. Реалізація кожного механізму розглянута як з теоретичної точки зору, так і в контексті конкретних компонентів розробленої системи.

3.1. Модель загроз системи

Побудова ефективної системи захисту починається з формальної моделі загроз, яка визначає активи системи, потенційних порушників та класифікує загрози за категоріями. У даному підрозділі використано методологію STRIDE, яку наведено на рисунку 3.1, розроблену компанією Microsoft, яка класифікує загрози за шістьма категоріями: підробка ідентичності (Spoofing), фальсифікація даних (Tampering), відмова від авторства (Repudiation), розкриття інформації (Information Disclosure), відмова в обслуговуванні (Denial of Service) та підвищення привілеїв (Elevation of Privilege).

Застосування методології STRIDE до компонентів системи наведено у таблиці 3.1.

Таблиця 3.1 – Модель загроз за методологією STRIDE

Категорія	Загроза	Компонент	Механізм захисту
Spoofing	Підробка JWT токена	AuthFilter	Перевірка підпису HMAC-SHA256
Spoofing	Brute force паролів	AuthController	BCrypt

Продовження таблиці 3.1

Tampering	Модифікація повідомлень у мережі	Nginx	TLS 1.3 + AES-256-256-GCM
Tampering	Підробка JWT payload	JwtAuthFilter	Верифікація підпису
Repudiation	Заперечення надсилання повідомлень	ChatMessageService	Зберігання у PostgreSQL
Information Disclosure	Витік хешів паролів	UserRepository	BCrypt з унікальною salt
Information Disclousure	Читання повідомлень адміністратором	ChatWebSocketHandler	E2E шифрування
DoS	Масові запити з однією IP	IpRateLimiter	ip_rate_limit.lua (Redis)
DoS	Масова реєстрація	EmailRateLimiter	email_rate_limit.lua (Redis)

Активами системи, що потребують захисту, є вміст повідомлень користувачів, облікові дані (паролі та токени автентифікації), персональні дані користувачів (електронні адреси, імена), метадані комунікацій (інформація про те хто з ким і коли спілкувався) та доступність самої системи.

Модель загроз STRIDE

Категорія	Загроза	Компонент системи	Механізм захисту
S — Spoofing Підробка ідентичності	Підробка JWT токена Brute force паролів Credential stuffing	JwtAuthFilter AuthController PasswordEncoder	HMAC-SHA256 підпис JWT BCrypt + Rate limiting Email верифікація
T — Tampering Фальсифікація даних	MITM атака Модифікація трафіку Підробка JWT payload	Nginx (TLS термінатор) JwtAuthFilter ChatWebSocketHandler	TLS 1.3 + Forward Secrecy AES-256-GCM шифрування JWT підпис HMAC-SHA256
R — Repudiation Відмова від авторства	Заперечення надсилання повідомлень Відмова від дій у системі	ChatMessageService PostgreSQL Spring Boot Actuator	Зберігання у PostgreSQL JWT прив'язка до userId Логування подій безпеки
I — Information Disclosure Розкриття даних	Перехоплення трафіку Читання вмісту БД Витік хешів паролів	Nginx + Signal Protocol Docker network BCrypt (PasswordEncoder)	HTTPS + WSS (TLS 1.3) Signal Protocol (E2E) BCrypt + унікальна сіль
D — Denial of Service Відмова в обслуговуванні	Масові запити з однієї IP Масова реєстрація Email спам	IpRateLimiter EmailRateLimiter Redis (ip/email TTL)	Lua + Redis (атомарний INCR) 429 Too Many Requests Прогресивне блокування
E — Elevation of Privilege Підвищення привілеїв	Доступ до чужих чатів WebSocket hijacking SQL ін'єкція	Spring Security ChatService Spring Data JPA	Перевірка учасника чату JWT авторизація WS Параметризовані SQL запити

Рисунок 3.1 - Реалізація моделі загроз STRIDE

Модель порушників включає три категорії. Зовнішній зловмисник - особа без будь-якого легітимного доступу до системи, яка діє через мережу. Внутрішній зловмисник - співробітник підприємства з легітимним доступом до системи, який намагається отримати доступ до даних інших користувачів або адміністративних функцій. Привілейований зловмисник - особа з адміністративним доступом до серверної інфраструктури.

3.2. Захист транспортного рівня

Захист транспортного рівня є першою лінією оборони системи і забезпечує конфіденційність та цілісність даних при їх передачі між клієнтом та сервером. У

розробленій системі захист транспортного рівня реалізовано через Nginx, який виступає єдиною точкою входу та TLS термінатором.

3.2.1. HTTPS та TLS 1.3

Протокол HTTPS забезпечує захищену передачу даних через HTTP шляхом інкапсуляції HTTP трафіку у TLS (Transport Layer Security) тунель. У розробленій системі використовується TLS версії 1.3, що зображений на рисунку 3.2.

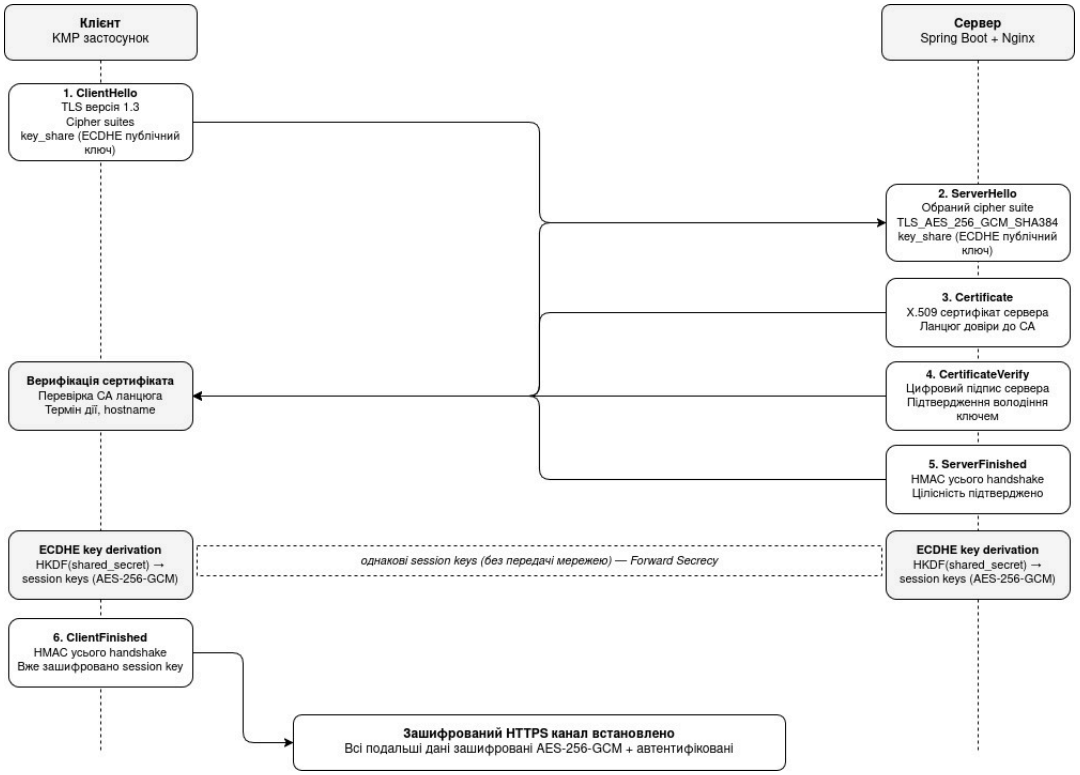

Рисунок 3.2 - Схема TLS 1.3 Handshake

TLS 1.3 встановлює захищене з'єднання за допомогою handshake процесу, що відбувається за один цикл обміну повідомленнями (1 RTT), тоді як попередня версія TLS 1.2 потребувала двох циклів. Скорочення кількості RTT зменшує

затримку встановлення з'єднання, що є важливим для продуктивності корпоративного застосунку.

Процес TLS 1.3 Handshake відбувається у такій послідовності. Клієнт надсилає повідомлення ClientHello, яке містить підтримувану версію TLS, перелік підтримуваних криптографічних наборів (cipher suites) та публічний ключ ECDHE для обміну ключами. Сервер відповідає повідомленням ServerHello з обраним cipher suite та власним публічним ключем ECDHE, після чого одразу надсилає сертифікат Certificate, підтвердження CertificateVerify та ServerFinished. Клієнт верифікує сертифікат сервера, перевіряючи ланцюг довіри до кореневого центру сертифікації (CA), термін дії сертифіката та відповідність hostname. Обидві сторони незалежно обчислюють спільний секрет через алгоритм ECDHE (Elliptic Curve Diffie-Hellman Ephemeral) і виводять з нього сесійні ключі через HKDF. Клієнт надсилає ClientFinished, після чого захищений канал встановлено.

Ключовою особливістю TLS 1.3 є обов'язкова підтримка Forward Secrecy через використання ефемерних ключів ECDHE. Це означає що навіть у разі компрометації довгострокового приватного ключа сервера зловмисник не зможе розшифрувати раніше перехоплений трафік, оскільки сесійні ключі генеруються заново для кожного з'єднання і не зберігаються. Це є критично важливою властивістю для корпоративної системи де збереження конфіденційності попередніх комунікацій є пріоритетом.

TLS 1.3 суттєво скорочує перелік підтримуваних cipher suites порівняно з TLS 1.2, видаляючи всі небезпечні алгоритми. У системі використовується cipher suite TLS_AES_256_GCM_SHA384, що забезпечує шифрування AES-256 у режимі GCM (Galois/Counter Mode) з автентифікацією та цілісністю через SHA-384.

3.2.2. WSS - WebSocket Secure

WebSocket Secure (WSS) є захищеною версією протоколу WebSocket, де WebSocket з'єднання встановлюється всередині вже існуючого TLS тунелю. Це означає, що всі WebSocket фрейми, включаючи повідомлення в реальному часі, передаються у зашифрованому вигляді тими самими механізмами що й HTTPS трафік.

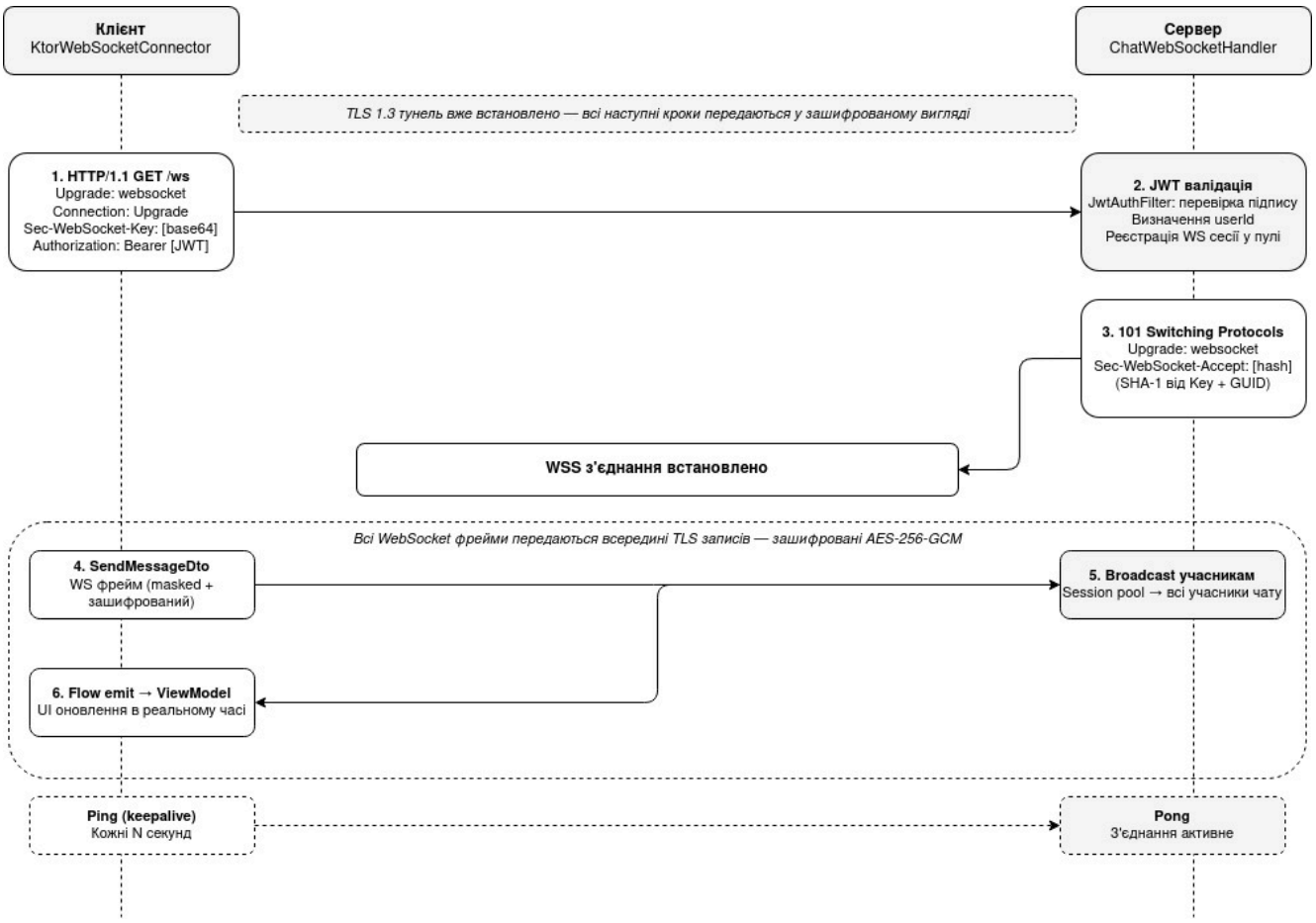

Рисунок 3.3 - Схема WSS з'єднання

Процес встановлення WSS з'єднання складається з двох етапів. На першому етапі клієнт встановлює TLS з'єднання з Nginx відповідно до процесу описаного у попередньому підрозділі. На другому етапі всередині цього TLS тунелю клієнт

надсилає стандартний HTTP GET запит з заголовками Upgrade: websocket та Connection: Upgrade, а також заголовком Sec-WebSocket-Key що містить випадкове значення у форматі Base64. Крім цього клієнт передає JWT токен для автентифікації через заголовок Authorization.

Сервер перевіряє JWT токен, визначає ідентифікатор користувача та реєструє WebSocket сесію. У разі успішної автентифікації сервер повертає відповідь 101 Switching Protocols із заголовком Sec-WebSocket-Accept, що містить хеш значення Sec-WebSocket-Key з фіксованим GUID, обчислений за алгоритмом SHA-1. Це підтверджує, що сервер є легітимним WebSocket сервером та захищає від атак типу Cross-Site WebSocket Hijacking.

Після встановлення з'єднання всі WebSocket фрейми передаються у зашифрованому вигляді всередині TLS записів. Клієнтські фрейми згідно зі специфікацією WebSocket обов'язково маскуються за допомогою 4-байтового маскувального ключа, що додатково ускладнює аналіз трафіку. Для підтримки з'єднання активним реалізовано механізм Ping/Pong - клієнт через клас KtorWebSocketConnector надсилає Ping фрейми через визначені інтервали часу, а сервер відповідає Pong фреймами.

3.3. Автентифікація та управління сесіями

Автентифікація є ключовим механізмом безпеки, що визначає хто має право використовувати систему. У розробленій системі реалізовано багаторівневу схему автентифікації на основі JSON Web Token з підтримкою двох типів токенів.

3.3.1. Структура та схема JWT автентифікації

JSON Web Token є відкритим стандартом (RFC 7519) для безпечної передачі інформації між сторонами у вигляді JSON об'єкта. Токен складається з трьох частин, розділених крапками: заголовку (header), корисного навантаження (payload) та підпису (signature). Заголовок містить тип токена та алгоритм підпису. Корисне навантаження містить твердження (claims) - стандартизовані та власні поля, такі як ідентифікатор користувача, час видачі та час закінчення дії. Підпис обчислюється як HMAC-SHA256 від закодованих Base64Url заголовку та payload з використанням секретного ключа сервера.

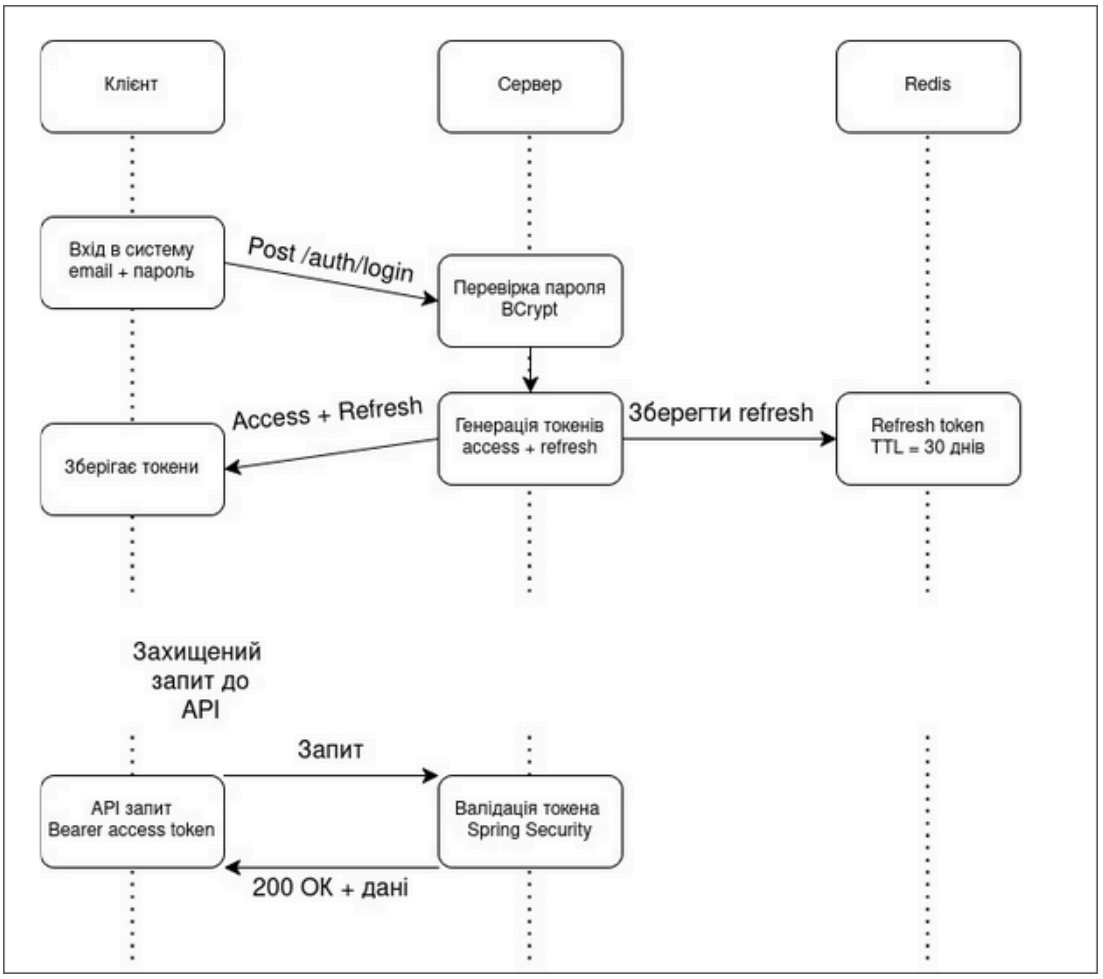

Рисунок 3.4 - Схема JWT автентифікації (access + refresh)

У системі реалізовано схему з двома типами токенів. Access token має короткий термін дії і використовується для авторизації кожного запиту до захищених ресурсів API. Stateless природа access token означає що сервер не зберігає інформацію про видані токени і може верифікувати їх валідність виключно через перевірку підпису. Це забезпечує горизонтальне масштабування без необхідності синхронізації стану між інстанціями сервісу. Refresh token має значно довший термін дії і зберігається у Redis з прив'язкою до конкретного користувача через механізм TTL. Зберігання refresh токена на сервері є принципово важливим - воно надає можливість примусової інвалідації сесії незалежно від терміну дії токена, що є критичним для корпоративної системи при звільненні співробітника або виявленні компрометації облікового запису.

3.3.2. Фільтр автентифікації Spring Security

Обробка кожного вхідного запиту у системі проходить через ланцюжок фільтрів Spring Security. Клас JwtAuthFilter є центральним компонентом цього ланцюжка і відповідає за перевірку JWT токена у кожному запиті до захищених ресурсів.

При отриманні запиту фільтр витягує значення заголовка Authorization та перевіряє наявність префіксу Bearer. Якщо заголовок відсутній або має неправильний формат запит передається далі по ланцюжку без встановлення контексту безпеки, що призведе до відповіді 401 при спробі доступу до захищеного ресурсу. Якщо заголовок присутній бібліотека JJWT верифікує підпис токена з використанням секретного ключа сервера та перевіряє термін дії. У разі успішної верифікації ідентифікатор користувача з payload токена використовується для встановлення контексту безпеки Spring Security, після чого запит передається до відповідного контролера.

Публічними і доступними без автентифікації залишаються лише endpoint реєстрації, підтвердження електронної пошти, входу в систему та скидання пароля. Усі інші endpoints, включаючи WebSocket endpoint, вимагають валідного JWT токена.

3.3.3. Автентифікація WebSocket з'єднань

WebSocket з'єднання потребує окремого підходу до автентифікації порівняно зі стандартними HTTP запитами, оскільки після встановлення з'єднання воно зберігається тривалий час. При спробі встановлення WebSocket з'єднання клас ChatWebSocketHandler перевіряє JWT токен переданий у заголовку запиту. У разі успішної автентифікації ідентифікатор користувача асоціюється з WebSocket сесією і зберігається у пулі активних з'єднань.

Важливим аспектом є обробка ситуації коли access token закінчується під час активного WebSocket з'єднання. Оскільки WebSocket є довготривалим з'єднанням клієнт має механізм автоматичного оновлення токенів - при отриманні помилки автентифікації через WebSocket застосунок оновлює access token через REST endpoint і перевстановлює з'єднання.

3.3.4. Верифікація електронної пошти

Обов'язкова верифікація електронної пошти при реєстрації виконує кілька функцій з точки зору безпеки. По-перше, вона підтверджує що користувач має доступ до вказаної електронної адреси, що ускладнює реєстрацію фіктивних облікових записів. По-друге, вона прив'язує обліковий запис до реальної особи, що дозволяє використовувати електронну пошту для скидання пароля та сповіщень безпеки. По-третє, вона захищає від атак типу user enumeration - система повертає

однакову відповідь незалежно від того чи існує вже обліковий запис з такою адресою.

Сервіс EmailVerificationService генерує криптографічно стійкий токен підтвердження, зберігає його хеш у таблиці email_verification_token з обмеженим терміном дії та надсилає посилання з токеном на вказану адресу через spring-boot-starter-mail. При переході за посиланням токен верифікується, обліковий запис активується, а токен видаляється з бази даних.

3.4. Захист паролів

Захист паролів є критичним компонентом безпеки будь-якої системи з автентифікацією. Навіть у разі витоку бази даних правильно захищені паролі не повинні надавати зловмиснику можливості автентифікуватись у системі або відновити оригінальні паролі у прийнятний термін.

У системі для хешування паролів використовується алгоритм BCrypt, реалізований через компонент PasswordEncoder у шарі infra/security. BCrypt є адаптивною хеш-функцією, спеціально розробленою для хешування паролів Нільсом Провосом та Девідом Мазьєром у 1999 році. На відміну від загальних криптографічних хеш-функцій (MD5, SHA-256) BCrypt має кілька властивостей, що роблять його придатним для захисту паролів.

BCrypt вбудовує унікальну криптографічно стійку сіль (128 біт) безпосередньо у результуючий хеш. Це унеможливлює використання попередньо обчислених таблиць відповідностей між паролями та їх хешами (rainbow tables), оскільки для кожного пароля сіль є унікальною. Алгоритм виконує функцію розширення ключа на основі шифру Blowfish, багатократно застосовуючи обчислювально дорогу функцію. Кількість ітерацій визначається параметром cost factor (2^n), який вбудовано у результуючий хеш. При збільшенні обчислювальних потужностей параметр можна збільшити без зміни алгоритму. Завдяки адаптивній

складності BCrypt залишається стійким навіть при використанні спеціалізованого апаратного забезпечення для перебору - GPU та ASIC мають значно менший приріст продуктивності на BCrypt порівняно з SHA-256 через послідовну природу алгоритму.

Порівняльний аналіз алгоритмів хешування паролів наведено у таблиці 3.2

Таблиця 3.2 – Порівняння алгоритмів хешування паролів

Алгоритм	Адаптивна складність	Вбудована сіль	GPU стійкість	Рекомендований OWASP
MD5	Ні	Ні	Низька	Ні
SHA-256	Ні	Ні	Низька	Ні
BCrypt	Так	Так	Середня	Так
Argon2id	Так	Так	Середня	Так
PBKDF2	Так	Так	Середня	Так

Argon2id є переможцем Password Hashing Competition 2015 і за рекомендаціями OWASP є пріоритетним вибором для нових систем завдяки вищій стійкості до атак з використанням GPU та ASIC. BCrypt обрано з міркувань широкої підтримки у екосистемі Spring Security через spring-security-crypto та достатнього рівня захисту для корпоративного застосунку. Подальший розвиток системи передбачає міграцію на Argon2id.

3.5. Захист від автоматизованих атак

Автоматизовані атаки - brute force, credential stuffing та масова реєстрація - є одними з найпоширеніших загроз для систем автентифікації. У розробленій

системі реалізовано дворівневий механізм захисту на основі Redis та Lua-скриптів.

3.5.1. Архітектура rate limiting

Механізм rate limiting реалізований на двох незалежних рівнях що доповнюють один одного. Перший рівень - обмеження за IP-адресою - реалізовано через перехоплювач IpRateLimitInterceptor що спрацьовує на кожен вхідний запит до захищених endpoints незалежно від їх вмісту. Другий рівень - обмеження за електронною адресою - реалізовано на рівні сервісу AuthService і спрацьовує після розбору тіла запиту коли відома цільова електронна адреса.

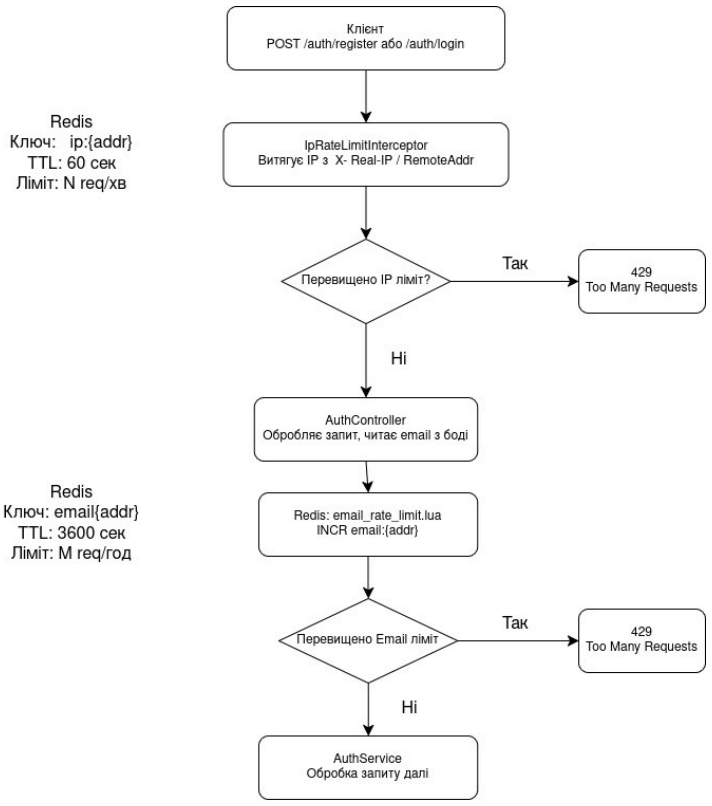

Рисунок 3.5 - Діаграма роботи rate limiting

Дворівнева архітектура є важливою з точки зору безпеки. Обмеження лише за IP може бути обійдено через використання проксі або ботнетів з великою кількістю IP-адрес. Обмеження лише за email дозволяє одній IP-адресі атакувати велику кількість різних облікових записів. Поєднання обох рівнів суттєво ускладнює будь-який сценарій атаки.

3.5.2. Реалізація через Redis Lua скрипти

Для реалізації атомарних операцій rate limiting використовуються Lua-скрипти що виконуються безпосередньо на стороні Redis. Це є принципово важливим архітектурним рішенням - Redis гарантує атомарне виконання Lua-скрипту без переривань іншими командами. Якби перевірка та інкремент лічильника виконувались як окремі Redis команди існувала б race condition: два паралельних запити могли б одночасно перевірити ліміт, отримати результат що ліміт не перевищено, і обидва успішно пройти перевірку навіть якщо один з них мав бути заблокований.

Скрипт ip_rate_limit.lua отримує IP-адресу клієнта як параметр. У першу чергу скрипт виконує команду INCR для збільшення лічильника запитів з відповідним ключем. Якщо лічильник щойно створений (значення дорівнює 1) скрипт встановлює TTL через команду EXPIRE що забезпечує автоматичне видалення лічильника після закінчення часового вікна. Якщо значення лічильника перевищує встановлений поріг скрипт повертає кількість секунд до скидання ліміту, що дозволяє клієнту отримати інформацію про час очікування через заголовок Retry-After.

Компонент IpRateLimiter виконує скрипт і при отриманні сигналу про перевищення ліміту генерує виняток RateLimitException. Клас IpResolver коректно обробляє ситуацію роботи за Nginx reverse proxy - він враховує заголовок

X-Real-IP що встановлюється Nginx і містить реальну IP-адресу клієнта, а не адресу Nginx.

Аналогічна логіка реалізована у скрипті email_rate_limit.lua для обмеження запитів за електронною адресою, проте з іншими параметрами часового вікна та порогових значень, оскільки захищає endpoint надсилання підтверджувальних листів від зловживань.

3.5.3. Обробка порушень безпеки

При порушенні rate limiting генерується виняток RateLimitException що успадковує від доменного класу винятків. Клас AuthExceptionHandler перехоплює цей та інші винятки безпеки і формує відповідні HTTP відповіді. Для перевищення rate limit повертається статус 429 (Too Many Requests) із заголовком Retry-After. Для невалідних облікових даних повертається статус 401 (Unauthorized) без деталізації причини відмови - зловмисник не отримує інформації про те чи існує обліковий запис з такою адресою. Для відмови в доступі до чужих ресурсів повертається статус 403 (Forbidden).

3.6. Наскрізне шифрування повідомлень

Наскрізне шифрування (End-to-End Encryption, E2E) є найвищим рівнем захисту вмісту повідомлень, за якого навіть оператор сервера не має технічної можливості читати повідомлення користувачів. У розробленій системі реалізовано E2E шифрування на основі Signal Protocol - найбільш захищеного та перевіреного протоколу для захищеного обміну повідомленнями, що використовується у Signal, WhatsApp та інших провідних месенджерах.

3.6.1. Теоретичні основи Signal Protocol

Signal Protocol поєднує два криптографічних компоненти: X3DH (Extended Triple Diffie-Hellman) для початкового обміну ключами та Double Ratchet для генерації унікального ключа для кожного повідомлення.

Алгоритм X3DH вирішує задачу безпечного встановлення спільного секрету між двома сторонами без попереднього захищеного каналу та без необхідності одночасної присутності обох сторін онлайн. Клієнт Б заздалегідь публікує на сервері пакет відкритих ключів (prekey bundle), що включає довгостроковий ключ ідентичності IK_B, ключ підпису SPK_B та одноразові ключі OPK_B. При першому зв’язку клієнт А завантажує prekey bundle та генерує ефемерний ключ EK_A. Обидві сторони незалежно виконують чотири операції Diffie-Hellman (DH1..DH4) та об'єднують результати для отримання masterSecret.

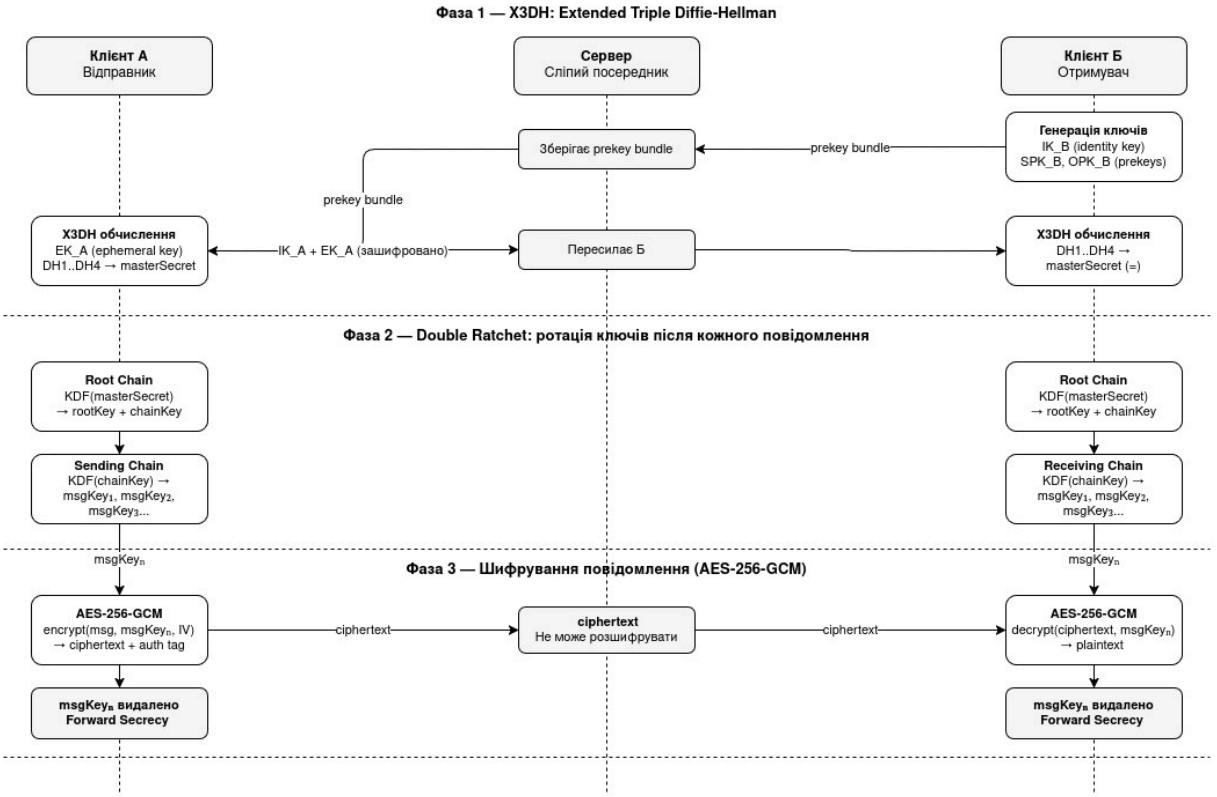

Рисунок 3.6 - Діаграма Signal Protocol (X3DH + Double Ratchet)

Double Ratchet алгоритм вирішує задачу забезпечення Forward Secrecy та Break-in Recovery при тривалому листуванні. Алгоритм підтримує два ланцюги ключів що постійно оновлюються. Root Chain виводить нові ключі ланцюга при кожному обміні DH ключами між сторонами. Sending та Receiving Chain виводять унікальний ключ повідомлення msgKey для кожного окремого повідомлення через функцію виведення ключів KDF. Після використання msgKey для шифрування або розшифрування він негайно видаляється. Це забезпечує Forward Secrecy - компрометація поточного стану не розкриває попередніх повідомлень, ключі яких вже видалено.

3.6.2. Практична реалізація E2E шифрування

При першому запуску застосунку на пристрої генерується пара ключів ECDH на основі кривої Curve25519. Приватний ключ зберігається виключно у захищеному сховищі платформи - Android Keystore для Android та Secure Enclave для iOS - і ніколи не залишає пристрій. Публічний ключ передається на сервер де зберігається у профілі користувача та доступний іншим учасникам системи.

При першому повідомленні у новому чаті відправник завантажує prekey bundle отримувача з сервера, виконує X3DH для отримання masterSecret, ініціалізує Double Ratchet та шифрує повідомлення унікальним msgKey через алгоритм AES-256-GCM. AES-256-GCM є автентифікованим режимом шифрування що забезпечує одночасно конфіденційність (шифрування AES-256), цілісність та автентичність (тег автентифікації GCM). Отримувач при отриманні першого повідомлення виконує аналогічний X3DH зі свого боку, ініціалізує власний Double Ratchet та розшифровує повідомлення.

Сервер у цій схемі є виключно транспортним посередником - він отримує зашифрований ciphertext та передає його отримувачу не маючи технічної

можливості його розшифрувати. Навіть адміністратор сервера з повним доступом до бази даних бачить лише зашифровані дані.

3.6.3. Порівняльний аналіз криптографічних алгоритмів

Вибір конкретних криптографічних алгоритмів для реалізації E2E шифрування базувався на аналізі сучасних стандартів та рекомендацій NIST і IETF. Порівняльний аналіз наведено у таблиці 3.3.

Таблиця 3.3 – Порівняння алгоритмів симетричного шифрування

Алгоритм	Розмір ключа	Режим	Автентифікація	Продуктивність
AES-128-CBC	128 біт	CBC	Ні	Висока
AES-256-GCM	256 біт	GCM	Так	Висока
ChaCha20-Poly 1305	256 біт	Stream	Так	Висока
AES-128-GCM	128 біт	GCM	Так	Дуже висока

AES-256-GCM обрано як основний алгоритм симетричного шифрування оскільки він є автентифікованим (AEAD) що забезпечує одночасно конфіденційність та цілісність повідомлень, підтримується апаратним прискоренням на всіх сучасних мобільних процесорах (AES-NI) та є рекомендованим алгоритмом NIST. ChaCha20-Poly1305 є гідною альтернативою з вищою продуктивністю на пристроях без апаратного прискорення AES, однак AES-256-GCM обрано з міркувань ширшої підтримки на цільових платформах.

3.7. Аналіз відповідності вимогам безпеки

Для оцінки повноти реалізованих механізмів захисту проведено аналіз відповідності системи вимогам OWASP Top 10 - переліку найпоширеніших вразливостей вебзастосунків.

Таблиця 3.4 – Відповідність системи вимогам OWASP Top 10

Загроза OWASP	Механізм захисту в системі
A01: Broken Access Control	Spring Security авторизація, перевірка учасника чату у ChatService
A02: Cryptographic Failures	TLS 1.3, AES-256-GCM, BCrypt, E2E шифрування
A03: Injection	Spring Data JPA (параметризовані запити), валідація вхідних даних
A04: Insecure Design	Модель загроз STRIDE, Defense in Depth, принцип мінімальних привілеїв
A05: Security Misconfiguration	Nginx конфігурація, Docker ізоляція, окремі профілі dev/prod
A07: Authentication Failures	JWT + BCrypt + Rate limiting + Email верифікація
A08: Software Integrity	Gradle Version Catalog, Docker образи з фіксованими версіями
A09: Logging Failures	Spring Boot Actuator, логування винятків безпеки

Проведений аналіз показує, що розроблена система реалізує комплексний підхід до забезпечення інформаційної безпеки та враховує основні рекомендації OWASP Top 10. Для кожної з актуальних категорій загроз впроваджено відповідні механізми захисту на рівні архітектури, серверної частини, мережевої взаємодії та управління доступом.

3.8. Аудит та моніторинг безпеки

Моніторинг безпеки є невід'ємною складовою захисту корпоративної системи - виявлення підозрілої активності в реальному часі дозволяє реагувати на інциденти до того як вони призведуть до серйозних наслідків.

У поточній реалізації логування безпекових подій здійснюється через стандартний механізм Spring Boot з фіксацією винятків RateLimitException, InvalidCredentialException та AuthExceptionHandler. Spring Boot Actuator надає endpoint для моніторингу стану здоров'я системи (/actuator/health) та метрик продуктивності.

Напрямками розвитку системи моніторингу є впровадження структурованого audit log у окрему таблицю PostgreSQL з фіксацією всіх подій автентифікації, сповіщення адміністратора при виявленні серії невдалих спроб входу, інтеграція з системами SIEM (Security Information and Event Management) для централізованого аналізу подій безпеки та реалізація сповіщень при вході з нового пристрою або нової географічної локації.

У третьому розділі детально описано та обґрунтовано багаторівневу систему захисту інформації розробленої системи корпоративного обміну повідомленнями. Побудовано модель загроз за методологією STRIDE що стала основою для проектування механізмів захисту. Описано захист транспортного рівня через TLS 1.3 для HTTPS та WSS з'єднань з обов'язковою підтримкою Forward Secrecy. Детально розглянуто схему JWT автентифікації з двома типами токенів, роль фільтра JwtAuthFilter у ланцюжку Spring Security та особливості автентифікації WebSocket з'єднань. Обґрунтовано вибір алгоритму BCrypt для захисту паролів та проведено його порівняння з альтернативами. Описано дворівневий механізм rate limiting на основі Redis Lua-скриптів із захистом від race condition через атомарне виконання. Розглянуто реалізацію E2E шифрування на основі Signal Protocol з детальним описом X3DH та Double Ratchet алгоритмів. Проведено аналіз відповідності системи вимогам OWASP Top 10.

4. ЗАСОБИ РЕАЛІЗАЦІЇ

У четвертому розділі обґрунтовано вибір мови програмування, фреймворків, бібліотек та інструментів розробки, що використовувались при реалізації інформаційної системи корпоративного захищеного обміну повідомленнями. Вибір кожного інструменту розглянуто з точки зору функціональних можливостей, безпеки, зрілості та відповідності вимогам проєкту.

4.1. Вибір мови програмування

Для реалізації як серверної так і клієнтської частини системи обрано мову програмування Kotlin. Використання єдиної мови для всього стеку розробки є стратегічним рішенням що забезпечує ряд практичних переваг. Розробники можуть працювати як над серверною так і над клієнтською частиною без необхідності перемикання між різними мовами та парадигмами. Спільні моделі даних та утиліти можуть бути реалізовані один раз та використані на всіх платформах. Уніфікований підхід до обробки помилок та асинхронного програмування через корутини застосовується однаково на сервері та клієнті.

Kotlin є статично типізованою мовою програмування розробленою компанією JetBrains що працює на платформі JVM та підтримує компіляцію у нативний код для платформ Android, iOS та Desktop через технологію Kotlin Multiplatform. Мова повністю сумісна з Java що дозволяє використовувати весь існуючий екосистем Java-бібліотек зокрема Spring Boot для серверної частини.

З точки зору безпеки Kotlin має ряд переваг порівняно з Java. Вбудований захист від NullPointerException через систему nullable типів унеможливлює цілий клас помилок на рівні компілятора. Незмінні колекції за замовчуванням зменшують ризик неочікуваної модифікації даних. Строга типізація та відсутність

неявних перетворень типів зменшують ризик логічних помилок що можуть призвести до вразливостей безпеки.

Kotlin Coroutines є вбудованим механізмом асинхронного програмування що дозволяє ефективно обробляти велику кількість одночасних WebSocket з'єднань без блокування потоків виконання. Це є особливо важливим для серверної частини де необхідно підтримувати тисячі одночасних з'єднань з мінімальними витратами ресурсів.

4.2. Серверна частина

У даному підрозділі описано основні фреймворки та бібліотеки що використовуються для реалізації серверної частини системи.

4.2.1. Spring Boot

Spring Boot версії 4.0 є основним фреймворком для розробки серверної частини системи. Spring Boot побудований на основі Spring Framework і надає механізм автоконфігурації що суттєво скорочує час налаштування проєкту - більшість компонентів конфігуруються автоматично на основі наявних залежностей та властивостей у файлах application.yaml.

Модульна організація на основі стартерів дозволяє точно визначати склад залежностей та уникати підключення зайвих компонентів. У системі використовуються стартери spring-boot-starter-web для REST API, spring-boot-starter-websocket для WebSocket підтримки, spring-boot-starter-data-jpa для роботи з PostgreSQL, spring-boot-starter-amqp для інтеграції з RabbitMQ, spring-boot-starter-data-redis для Redis, spring-boot-starter-mail для надсилання

електронних листів, spring-boot-starter-validation для валідації вхідних даних та spring-boot-starter-security для захисту endpoints.

Підтримка різних конфігураційних профілів через файли application-dev.yaml та application-prod.yaml дозволяє використовувати різні налаштування для середовища розробки та продуктивного середовища. Це є важливим з точки зору безпеки - продуктивне середовище має більш строгі налаштування безпеки та відключені інструменти налагодження.

Spring Boot Actuator надає готові endpoints для моніторингу стану системи (/actuator/health), метрик продуктивності та управління застосунком без необхідності реалізації власних рішень моніторингу.

4.2.2. Spring Security та JJWT

Spring Security є стандартним рішенням для захисту Spring-застосунків і надає гнучкий механізм конфігурації правил доступу до ресурсів через ланцюжок фільтрів (Security Filter Chain). У системі Spring Security відповідає за визначення публічних та захищених endpoints, інтеграцію з JWT фільтром JwtAuthFilter та обробку виключень автентифікації.

Бібліотека spring-security-crypto надає реалізацію алгоритму BCrypt для хешування паролів через інтерфейс PasswordEncoder. Це забезпечує стандартизований підхід до захисту паролів інтегрований у екосистему Spring Security.

Для роботи з JWT токенами використовується бібліотека JJWT (Java JWT) версії 0.12.6 що складається з трьох артефактів: jjwt-api для публічного API, jjwt-impl для реалізації та jjwt-jackson для серіалізації через Jackson. JJWT є найбільш зрілою та широко використовуваною реалізацією JWT для JVM платформи з підтримкою всіх стандартних алгоритмів підпису. Клас

TokenGenerator у шарі infra/security інкапсулює логіку генерації та верифікації токенів через JJWT API.

4.2.3. RabbitMQ

RabbitMQ є брокером повідомлень що реалізує протокол AMQP (Advanced Message Queuing Protocol) і використовується для асинхронної комунікації між модулями серверної частини. Інтеграція здійснюється через стартер spring-boot-starter-amqp що надає абстракції RabbitTemplate для відправки та @RabbitListener для отримання повідомлень.

Використання RabbitMQ для комунікації між модулями user та chat забезпечує слабку зв'язаність компонентів. При оновленні фото профілю або зміні імені користувача модуль user публікує подію у чергу RabbitMQ. Клас ChatUserEventListener у модулі chat підписується на ці події та оновлює відповідні записи у своїй базі даних. Такий підхід забезпечує кінцеву узгодженість (eventual consistency) даних між модулями без прямих залежностей між ними та підвищує стійкість системи до тимчасових збоїв окремих компонентів.

4.2.4. Redis

Redis використовується як сховище даних в оперативній пам'яті для двох функціонально різних задач. Інтеграція здійснюється через стартер spring-boot-starter-data-redis що надає клієнт RedisTemplate для роботи з різними типами даних Redis.

Для зберігання refresh токенів Redis використовує рядкові значення з ключами виду refresh:{userId}:{tokenId} та TTL що відповідає терміну дії токена. Механізм автоматичного видалення прострочених записів через TTL забезпечує

що Redis не накопичує застарілі токени без необхідності реалізації окремих фонових процесів очищення.

Для rate limiting Redis використовує числові лічильники з ключами виду ip:{адреса} та email:{адреса}. Lua-скрипти що виконуються на стороні Redis через RedisScript API забезпечують атомарне виконання операцій перевірки та інкременту лічильника.

4.2.5. PostgreSQL

PostgreSQL версії 16 використовується як основна реляційна база даних системи. Доступ до PostgreSQL здійснюється через Spring Data JPA що надає рівень абстракції над JDBC та дозволяє описувати сутності через анотації JPA та визначати репозиторії через інтерфейси з автоматичною генерацією SQL запитів.

Використання Spring Data JPA з параметризованими запитами є важливим з точки зору безпеки оскільки повністю унеможливлює SQL ін'єкції - всі вхідні параметри передаються окремо від SQL тексту запиту та екрануються автоматично.

4.2.6. OkHttp та Firebase Admin SDK

Бібліотека OkHttp версії 4.12.0 використовується для виконання HTTP запитів до зовнішніх сервісів з боку серверної частини. Firebase Admin SDK версії 9.5.0 підключений як залежність для майбутньої реалізації push-сповіщень що є запланованим напрямком розвитку системи.

4.3. Клієнтська частина

У даному підрозділі описано бібліотеки та інструменти що використовуються для розробки мультиплатформенного клієнтського застосунку.

4.3.1. Kotlin Multiplatform та Compose Multiplatform

Kotlin Multiplatform (KMP) є технологією компанії JetBrains що дозволяє писати спільний код для платформ Android, iOS та Desktop компілюючи його у нативний код для кожної платформи. Спільний код розміщується у source set commonMain та має доступ до платформо-незалежних API. Платформо-специфічна логіка розміщується у відповідних source sets: androidMain, iosMain та desktopMain.

Compose Multiplatform версії 1.9.0 розширює підхід KMP на рівень інтерфейсу користувача дозволяючи описувати UI декларативно з єдиної кодової бази. Compose використовує реактивну модель де інтерфейс автоматично перебудовується при зміні стану. Це добре поєднується з патерном MVI що використовується у клієнтському застосунку.

З точки зору безпеки KMP забезпечує що криптографічні операції та управління ключами можуть бути реалізовані з використанням нативних API безпеки кожної платформи через механізм expect/actual. Функція expect оголошує інтерфейс у спільному коді а actual реалізації у платформо-специфічних source sets використовують Android Keystore або iOS Secure Enclave для захищеного зберігання криптографічних ключів.

4.3.2. Ktor Client

Ktor Client версії 3.2.3 є мультиплатформенною HTTP бібліотекою що підтримує всі цільові платформи проєкту. Ktor Client надає плагінну архітектуру де кожна функціональність підключається як окремий плагін. У системі використовуються плагіни ContentNegotiation для автоматичної серіалізації та десеріалізації JSON через kotlinx.serialization, Auth для автоматичного оновлення JWT токенів при отриманні відповіді 401, Logging для логування HTTP запитів у режимі розробки та WebSockets для встановлення WebSocket з'єднань.

Плагін Auth є особливо важливим з точки зору безпеки та зручності використання - він автоматично перехоплює відповіді 401, оновлює access token через refresh token та повторює оригінальний запит з новим токеном без необхідності явної обробки цього сценарію у кожному місці коду.

4.3.3. Koin

Koin версії 4.1.0 є легковісним фреймворком ін'єкції залежностей спеціально розробленим для Kotlin та Kotlin Multiplatform проєктів. На відміну від Dagger/Hilt що використовують генерацію коду під час компіляції Koin реалізує Service Locator патерн з DSL для опису залежностей. Це забезпечує простішу конфігурацію та швидшу компіляцію при збереженні типобезпечності завдяки Kotlin DSL.

Koin підтримує мультиплатформенні модулі залежностей та платформо-специфічні перевизначення що дозволяє надавати різні реалізації залежностей для різних платформ. Модулі залежностей ChatDataModule мають окремі реалізації для Android та Desktop через механізм expect/actual.

4.3.4. Room

Room версії 2.7.2 є мультиплатформенною бібліотекою для роботи з SQLite базою даних що надає зручний рівень абстракції через анотації та автоматичну генерацію коду. Room забезпечує типобезпечний доступ до даних через DAO інтерфейси та повну інтеграцію з Kotlin Coroutines та Flow.

Використання KSP (Kotlin Symbol Processing) для генерації коду Room забезпечує перевірку SQL запитів під час компіляції - синтаксичні помилки у SQL запитах виявляються на етапі збірки а не під час виконання. Це підвищує надійність та безпеку роботи з локальною базою даних.

4.3.5. Додаткові бібліотеки

Kotlinx Serialization версії 1.9.0 використовується для серіалізації та десеріалізації JSON даних при мережевій взаємодії. Бібліотека є мультиплатформенною та інтегрується з Ktor Client через відповідний плагін.

Kotlinx Coroutines версії 1.10.2 надає реалізацію корутин для всіх цільових платформ. На Android та JVM використовується диспетчер Dispatchers.IO для мережевих операцій та Dispatchers.Main для оновлення UI. На Desktop використовується kotlinx-coroutines-swing для інтеграції з подієвим циклом Swing.

Coil версії 3.3.0 є мультиплатформенною бібліотекою для асинхронного завантаження та кешування зображень через Ktor Client як мережевий бекенд через coil-network-ktor3.

MOKO Permissions версії 0.19.1 надає мультиплатформенний API для запиту дозволів платформи через єдиний інтерфейс у спільному коді.

Kermit версії 2.0.8 є мультиплатформенною бібліотекою логування що надає уніфікований API для запису логів на всіх платформах з можливістю налаштування рівнів логування та виводу.

4.4. Інфраструктура

У даному підрозділі описано інфраструктуру на якій розгортається система.

4.4.1. Docker та Docker Compose

Docker є платформою контейнеризації що дозволяє пакувати застосунок разом з усіма залежностями у стандартизований контейнер. Використання Docker забезпечує ідентичність середовища виконання між локальною розробкою та продуктивним сервером що усуває проблему «працює на моїй машині».

Docker Compose надає інструмент для оркестрації кількох контейнерів через декларативний опис у файлі docker-compose.yaml. Для системи визначено контейнери Spring Boot застосунку, PostgreSQL, Redis та RabbitMQ з відповідними налаштуваннями мереж, томів для персистентного зберігання даних та залежностей між сервісами.

З точки зору безпеки Docker Compose конфігурація визначає ізольовану внутрішню мережу в якій розміщено всі сервіси. Nginx є єдиним сервісом що має відкритий порт 443 назовні. PostgreSQL, Redis та RabbitMQ доступні лише з внутрішньої Docker мережі що унеможливлює їх пряму атаку ззовні.

4.4.2. Nginx

Nginx використовується як reverse proxy та TLS термінатор. Як reverse proxy Nginx приймає всі вхідні з'єднання та перенаправляє їх до відповідних внутрішніх сервісів на основі шляху URL. Це дозволяє розміщувати REST API та WebSocket endpoint на одному зовнішньому порту.

Як TLS термінатор Nginx відповідає за встановлення HTTPS та WSS з'єднань з клієнтами та передачу розшифрованого трафіку до Spring Boot застосунку через внутрішню мережу. Конфігурація Nginx визначає підтримувані TLS версії (лише 1.2 та 1.3), дозволені cipher suites та параметри сертифікатів. Клас NginxConfig у серверній частині відповідає за коректну обробку заголовків що встановлюються Nginx зокрема X-Real-IP та X-Forwarded-For.

4.5. Середовище розробки та інструменти

Розробка серверної та клієнтської частини здійснювалась у середовищі IntelliJ IDEA Ultimate що є офіційним середовищем розробки для Kotlin та Spring Boot від компанії JetBrains. IntelliJ IDEA надає вбудовану підтримку Kotlin Multiplatform, Spring Boot та всіх використаних бібліотек з підсвічуванням коду, автодоповненням та інструментами рефакторингу.

У четвертому розділі обґрунтовано вибір технологічного стеку для реалізації інформаційної системи корпоративного захищеного обміну повідомленнями. Використання Kotlin як єдиної мови для всього стеку розробки забезпечує уніфікований підхід та спрощує підтримку кодової бази. Spring Boot з екосистемою Spring Security та JJWT надає надійну основу для реалізації механізмів безпеки серверної частини. Kotlin Multiplatform та Compose Multiplatform забезпечують розробку клієнтського застосунку для Android та Desktop з єдиної кодової бази з можливістю використання нативних API безпеки кожної платформи. Контейнеризація через Docker та Compose забезпечує просте та безпечне розгортання системи на власній інфраструктурі підприємства.

5. ПРОГРАМНА РЕАЛІЗАЦІЯ

У п'ятому розділі описано програмну реалізацію розробленої системи - внутрішню структуру модулів серверної та клієнтської частин, реалізацію механізмів безпеки, обміну повідомленнями в реальному часі, локального кешування та розгортання системи.

5.1. Структура програмної системи

Серверна частина організована за шаровою архітектурою у межах мультимодульного Gradle проєкту. Кожен функціональний модуль має чітко визначену структуру пакетів що відображає поділ на шари api, domain, infra та service.

Модуль user містить такі пакети. Пакет api/controller містить клас AuthController що визначає endpoints реєстрації (POST /auth/register), входу (POST /auth/login), оновлення токенів (POST /auth/refresh), виходу (POST /auth/logout), підтвердження електронної пошти (GET /auth/verify), запиту скидання пароля (POST /auth/forgot-password), скидання пароля (POST /auth/reset-password) та зміни пароля (PUT /auth/change-password). Пакет api/config містить компоненти безпеки - JwtAuthFilter для перевірки JWT токенів, IpRateLimitInterceptor для обмеження запитів за IP та WebMvcConfig для реєстрації перехоплювача. Пакет api/dto визначає об'єкти передачі даних для всіх endpoints: LoginRequest, RegisterRequest, RefreshRequest, EmailRequest, ResetPasswordRequest, ChangePasswordRequest, UserDto та AuthenticatedUserDto. Пакет api/exception_handling містить AuthExceptionHandler що перетворює доменні винятки на відповідні HTTP відповіді. Пакет domain/exception визначає ієрархію доменних винятків: InvalidCredentialException, UserAlreadyExistsException, EmailNotVerifiedException,

RateLimitException, SamePasswordException та UserNotFoundException. Пакет infra/database містить JPA сутності UserEntity, EmailVerificationTokenEntity, PasswordResetTokenEntity, RefreshTokenEntity та відповідні Spring Data репозиторії. Пакет infra/security містить PasswordEncoder для BCrypt хешування та TokenGenerator для генерації та верифікації JWT. Пакет infra/rate_limiting містить IpRateLimiter, EmailRateLimiter та IpResolver для визначення реальної IP-адреси клієнта. Пакет service містить AuthService, EmailVerificationService та PasswordResetService.

Модуль chat містить такі пакети. Пакет api/controller містить ChatController для управління чатами (GET /chats, POST /chats, DELETE /chats/{id}), ChatMessageController для управління повідомленнями (GET /chats/{id}/messages, DELETE /messages/{id}) та ChatParticipantController для управління учасниками (POST /chats/{id}/participants, DELETE /chats/{id}/participants/{userId}). Пакет api/websocket містить ChatWebSocketHandler та WebSocketConfig для налаштування WebSocket endpoint. Пакет api/dto/ws визначає WebSocket DTO: SendMessageDto, DeleteMessageDto, ChatParticipantsChangedDto, ProfilePictureUpdateDto, ErrorDto та WebSocketEvent як sealed клас для маршрутизації подій. Пакет domain/event визначає події RabbitMQ: MessageDeletedEvent, ChatParticipantsJoinedEvent, ChatParticipantLeftEvent та ProfilePictureUpdateEvent. Пакет infra/messaging містить ChatUserEventListener для обробки подій від модуля user. Пакет service містить ChatService, ChatMessageService, ChatParticipantService та ProfilePictureService.

Клієнтська частина організована у мультиплатформенному проєкті з поділом на модулі core та feature. Модуль composeApp є точкою входу і містить платформо-специфічну ініціалізацію для Android (androidMain) з AndroidManifest.xml та для Desktop (desktopMain) з головним вікном застосунку. Спільний код точки входу у commonMain налаштовує граф навігації що об'єднує AuthGraph та ChatGraph та ініціалізує Koin модулі залежностей.

5.2. Реалізація механізмів безпеки

Реалізація механізмів безпеки розподілена між кількома компонентами серверної та клієнтської частин. У даному підрозділі описано ключові компоненти та їх взаємодію.

Клас TokenGenerator у пакеті infra/security модуля user інкапсулює всю логіку роботи з JWT через бібліотеку JJWT. Метод генерації access token приймає ідентифікатор користувача та формує JWT з payload що містить sub (subject - ідентифікатор користувача), iat (issued at - час видачі) та exp (expiration - час закінчення дії). Токен підписується алгоритмом HMAC-SHA256 з секретним ключем що зберігається у конфігурації застосунку. Метод генерації refresh token формує криптографічно стійкий випадковий ідентифікатор та зберігає його у Redis з відповідним TTL. Метод верифікації токена використовує JJWT JwtParser для перевірки підпису та терміну дії і повертає ідентифікатор користувача з payload або генерує виняток при невалідному токені.

Клас JwtAuthFilter розширює OncePerRequestFilter що гарантує одноразове виконання фільтра для кожного запиту. Фільтр витягує токен з заголовка Authorization, викликає TokenGenerator для верифікації та встановлює UsernamePasswordAuthenticationToken у SecurityContextHolder Spring Security. Завдяки цьому всі подальші компоненти у ланцюжку обробки запиту мають доступ до ідентифікатора поточного користувача.

Клас IpRateLimitInterceptor реалізує інтерфейс HandlerInterceptor Spring MVC та перехоплює запити до endpoints що потребують rate limiting. Метод preHandle викликається до обробки запиту контролером - якщо IpRateLimiter генерує RateLimitException перехоплювач повертає false що зупиняє подальшу обробку запиту і AuthExceptionHandler формує відповідь 429.

Клас IpResolver визначає реальну IP-адресу клієнта з урахуванням роботи за Nginx reverse proxy. Метод спочатку перевіряє наявність заголовка X-Real-IP що встановлюється Nginx конфігурацією і містить оригінальну IP-адресу клієнта. За

відсутності цього заголовка використовується X-Forwarded-For або безпосередньо remoteAddr з HTTP запиту. Клас NginxConfig налаштовує Spring MVC для коректної обробки цих заголовків через RemoteIpFilter.

На клієнтській стороні безпека реалізована через автоматичне управління токенами у Ktor Client. Плагін Auth налаштовано з типом bearer що забезпечує автоматичну підстановку access token у заголовок Authorization: Bearer кожного запиту та автоматичне оновлення токена при отриманні відповіді 401 через виклик endpoint /auth/refresh з refresh token.

5.3. Реалізація обміну повідомленнями в реальному часі

Обмін повідомленнями в реальному часі реалізовано на основі Raw WebSocket протоколу з використанням sealed класу WebSocketEvent для типобезпечної маршрутизації подій.

На стороні сервера клас ChatWebSocketHandler реалізує інтерфейс WebSocketHandler Spring та управляє пулом активних WebSocket сесій. При встановленні нового з'єднання обробник перевіряє JWT токен, визначає ідентифікатор користувача та реєструє сесію у ConcurrentHashMap з ключем у вигляді ідентифікатора користувача. Використання ConcurrentHashMap є важливим для безпечної роботи у багатопотоковому середовищі де кілька корутин можуть одночасно звертатись до пулу сесій.

При отриманні вхідного повідомлення обробник десеріалізує JSON у відповідний підтип WebSocketEvent та маршрутизує до відповідного методу. При отриманні SendMessageDto виклик передається до ChatMessageService що зберігає повідомлення у PostgreSQL та через RabbitMQ публікує подію для розсилки всім учасникам чату. ChatUserEventListener отримує подію та через WebSocket handler надсилає повідомлення кожному активному учаснику чату через їх сесії у пулі.

На стороні клієнта WebSocket з'єднання управляється через кілька взаємодіючих компонентів. Клас KtorWebSocketConnector встановлює з'єднання через Ktor Client WebSocket API з передачею JWT токена у заголовку запиту. Після встановлення з'єднання запускаються дві корутини - одна для читання вхідних повідомлень та перетворення їх у Flow подій, інша для надсилання вихідних повідомлень з каналу. Клас WebSocketChatConnectionClient реалізує доменний інтерфейс ChatConnectionClient та надає Flow<IncomingWebSocketDto> для реактивного отримання повідомлень у ViewModel.

Клас ConnectionRetryHandler реалізує логіку автоматичного відновлення з'єднання з експоненційною затримкою між спробами. При розриві з'єднання обробник очікує наростаючий інтервал часу перед кожною наступною спробою підключення що запобігає перевантаженню сервера при масовому відключенні клієнтів. ConnectivityObserver з платформо-специфічними реалізаціями відстежує стан мережевого з'єднання пристрою - при втраті мережі з'єднання закривається, при відновленні автоматично відновлюється. AppLifecycleObserver коректно закриває WSS з'єднання при переході застосунку у фоновий режим та відновлює при поверненні на передній план.

Стан WebSocket з'єднання представлено доменним sealed класом ConnectionState з підтипами Connected, Disconnected та Reconnecting. Утиліта ConnectionStateToUiText перетворює стан з'єднання на локалізований текст для відображення у UI застосунку. ChatListDetailViewModel підписується на Flow стану з'єднання та відображає відповідний індикатор у інтерфейсі користувача.

5.4. Реалізація offline-first кешування

Стратегія offline-first реалізована через поєднання Room локальної бази даних та класів OfflineFirstChatRepository і OfflineFirstMessageRepository що реалізують доменні інтерфейси ChatRepository та MessageRepository.

При завантаженні списку чатів OfflineFirstChatRepository спочатку повертає дані з Room через ChatDao.getChatsWithLastMessage() що використовує представлення LastMessageView для отримання останнього повідомлення кожного чату. Паралельно запускається мережевий запит через KtorChatService для отримання актуальних даних з сервера. Отримані дані зберігаються у Room через ChatDao.upsertChats() що автоматично призводить до оновлення Flow та перебудови UI з актуальними даними. Такий підхід забезпечує миттєве відображення кешованих даних при запуску застосунку без видимого затримання інтерфейсу.

При отриманні нових повідомлень через WebSocket клас OfflineFirstMessageRepository зберігає їх у Room через ChatMessageDao що автоматично оновлює Flow повідомлень у ChatDetailViewModel. Це забезпечує консистентність між даними отриманими через REST API та WebSocket.

База даних ChatDatabase ініціалізується через DatabaseFactory з платформо-специфічними реалізаціями. На Android використовується Room.databaseBuilder з шляхом до файлу у директорії застосунку. На Desktop використовується SQLite бекенд через androidx.sqlite:sqlite-bundled з шляхом до файлу у директорії конфігурації застосунку відповідно до конвенцій операційної системи.

5.5. Реалізація навігації та UI

Навігація у клієнтському застосунку реалізована через бібліотеку navigation-compose версії 2.9.0 з двома графами навігації. AuthGraph визначає маршрути екранів автентифікації: Login, Register, RegisterSuccess, EmailVerification, ForgotPassword та ResetPassword. ChatGraph визначає маршрути основної функціональності: ChatListDetail (адаптивний макет), CreateChat та ManageChat.

Адаптивний макет реалізований через ChatListDetailAdaptiveLayout з використанням Material3Adaptive бібліотеки версії 1.2.0. На планшетах та Desktop пристроях список чатів та вікно переписки відображаються одночасно у двопанельному режимі. На смартфонах відображається або список чатів або вікно переписки залежно від обраного чату. Клас NoSpacingPaneScaffoldDirective налаштовує відступи між панелями для досягнення необхідного вигляду.

Кожен екран реалізований за патерном MVI. ChatDetailViewModel підписується на Flow<List<MessageWithSender>> з MessageRepository та Flow<ConnectionState> з ChatConnectionClient. При отриманні нових значень ViewModel оновлює ChatDetailState що містить список повідомлень, стан завантаження, стан з'єднання та інші дані необхідні для рендерингу. ChatDetailScreen підписується на стан через collectAsStateWithLifecycle та декларативно описує інтерфейс. Компонент PaginationScrollListener відстежує позицію прокрутки та при наближенні до початку списку запитує наступну сторінку повідомлень через ViewModel.

5.6. Розгортання системи

Розгортання системи на інфраструктурі підприємства здійснюється через Docker Compose. Конфігурація визначає чотири сервіси з відповідними налаштуваннями.

Сервіс Spring Boot застосунку збирається з Dockerfile що використовує багатоетапну збірку (multi-stage build). На першому етапі використовується образ Gradle для компіляції застосунку та збирання JAR файлу. На другому етапі використовується мінімальний образ JRE для запуску застосунку без інструментів збірки. Такий підхід суттєво зменшує розмір фінального Docker образу та скорочує поверхню атаки. Сервіс налаштовано з профілем prod через змінну середовища SPRING_PROFILES_ACTIVE=prod.

Сервіс PostgreSQL використовує офіційний образ postgres:16 з налаштуванням томів для персистентного зберігання даних та змінними середовища для імені бази даних, користувача та пароля. Сервіс розміщений у внутрішній Docker мережі та не має відкритих портів назовні.

Сервіс Redis використовує офіційний образ redis:7 з налаштуванням тому для збереження даних при перезапуску контейнера. Команда запуску включає --requirepass для встановлення пароля автентифікації Redis.

Сервіс RabbitMQ використовує образ rabbitmq:3-management з увімкненим веб-інтерфейсом управління що доступний лише через внутрішню мережу. Конфігурація визначає ім'я користувача та пароль через змінні середовища.

Nginx конфігурується через окремий конфігураційний файл що визначає TLS налаштування, дозволені cipher suites, перенаправлення HTTP на HTTPS та маршрутизацію запитів до Spring Boot застосунку з коректним встановленням заголовків X-Real-IP та X-Forwarded-Proto.

Секретні дані - паролі бази даних, Redis, RabbitMQ, JWT секретний ключ та інші чутливі конфігураційні параметри - передаються через файл .env що не включається до системи контролю версій. Файл .env.example з описом необхідних змінних включається до репозиторію як шаблон для розгортання.

У п'ятому розділі описано програмну реалізацію інформаційної системи корпоративного захищеного обміну повідомленнями. Детально розглянуто структуру пакетів серверних модулів user та chat та модульну організацію KMP клієнта. Описано реалізацію ключових механізмів безпеки - JWT фільтра, rate limiting та управління токенами на клієнті. Розглянуто реалізацію WebSocket комунікації з типобезпечною маршрутизацією подій через sealed клас WebSocketEvent та механізми відновлення з'єднання. Описано offline-first стратегію кешування через Room з реактивним оновленням UI через Kotlin Flow. Розглянуто адаптивний макет інтерфейсу та реалізацію MVI патерну. Описано процес розгортання через Docker Compose з багатоетапною збіркою образів та безпечним управлінням секретними даними.

6. ТЕСТУВАННЯ СИСТЕМИ

У шостому розділі описано стратегію тестування розробленої системи, методи верифікації механізмів безпеки, функціональне тестування основних сценаріїв роботи та аналіз результатів тестування.

6.1. Стратегія тестування

Тестування інформаційної системи корпоративного захищеного обміну повідомленнями охоплює кілька рівнів відповідно до піраміди тестування. Модульне тестування перевіряє коректність роботи окремих компонентів у ізоляції від зовнішніх залежностей. Інтеграційне тестування перевіряє взаємодію між компонентами системи зокрема між сервісами та базою даних. Тестування безпеки перевіряє коректність роботи механізмів захисту включаючи JWT автентифікацію, rate limiting та авторизацію доступу до ресурсів. Функціональне тестування перевіряє коректність реалізації бізнес-сценаріїв з точки зору користувача.

Для тестування серверної частини використовується екосистема Spring Boot Test що надає анотацію @SpringBootTest для завантаження повного контексту застосунку, MockMvc для тестування REST endpoints без реального HTTP сервера, Spring Security Test для тестування захищених endpoints та spring-rabbit-test для тестування взаємодії з RabbitMQ. Для модульного тестування використовується JUnit 5 через залежність kotlin-test-junit5 з підтримкою корутин через kotlinx-coroutines-test.

6.2. Тестування механізмів безпеки

Тестування механізмів безпеки є пріоритетним напрямком оскільки некоректна реалізація захисту може призвести до серйозних наслідків для корпоративних даних.

Тестування JWT автентифікації охоплює кілька сценаріїв. Перевірка доступу до захищеного endpoint без токена має повертати відповідь 401. Доступ з валідним токеном має надавати доступ до ресурсу та повертати відповідь 200. Доступ з простроченим токеном має повертати 401. Доступ з токеном з невалідним підписом - з підписом що відрізняється від очікуваного - має повертати 401. Доступ з токеном де payload модифіковано без оновлення підпису має повертати 401. Тестування з Spring Security Test здійснюється через анотацію @WithMockUser для симуляції автентифікованого користувача та через безпосереднє формування JWT токенів з різними параметрами через TokenGenerator.

Тестування rate limiting перевіряє що механізм обмеження спрацьовує коректно при перевищенні порогового значення. Тест надсилає серію запитів з однієї IP-адреси та перевіряє що після перевищення ліміту відповідь змінюється з 200 на 429. Тест також перевіряє наявність заголовка Retry-After у відповіді 429 та коректність його значення. Окремий тест перевіряє що після закінчення часового вікна TTL лічильник скидається та запити знову приймаються. Для ізоляції тестів rate limiting використовується @DirtiesContext що перезавантажує контекст Spring між тестами та EmbeddedRedis для запуску вбудованого Redis.

Тестування авторизації перевіряє що користувач не може отримати доступ до чужих ресурсів. Тест намагається отримати повідомлення чату до якого поточний користувач не належить та перевіряє відповідь 403. Тест перевіряє що видалення повідомлення іншого користувача повертає 403. Тест перевіряє що додавання учасника до чату можливе лише для учасника цього чату.

Тестування захисту паролів перевіряє коректність роботи BCrypt хешування. Тест верифікує що два хеші одного пароля є різними завдяки унікальній солі. Тест перевіряє що метод matches повертає true для коректного пароля та false для некоректного. Тест перевіряє що хеш має очікуваний формат BCrypt.

Тестування верифікації електронної пошти перевіряє що вхід у систему без підтвердження email повертає відповідну помилку з кодом що відповідає EmailNotVerifiedException. Тест перевіряє що після підтвердження email вхід стає можливим. Тест перевіряє що прострочений токен підтвердження не приймається.

6.3. Тестування функціональності

Функціональне тестування, зображене на рисуку 6.1 та рисунку 6.1, охоплює основні бізнес-сценарії системи та перевіряє їх коректну реалізацію від рівня API до рівня бази даних.

Тестування реєстрації та автентифікації охоплює повний цикл від створення облікового запису до отримання токенів. Інтеграційний тест реєструє нового користувача через POST /auth/register, перевіряє що відповідь містить статус 200 та що у базі даних створено запис користувача зі статусом непідтвердженого email. Тест симулює підтвердження email через виклик POST /auth/verify з валідним токеном та перевіряє що статус користувача змінився. Тест виконує вхід через POST /auth/login та перевіряє що відповідь містить access token та refresh token. Тест виконує оновлення токенів через POST /auth/refresh та перевіряє отримання нового access token.

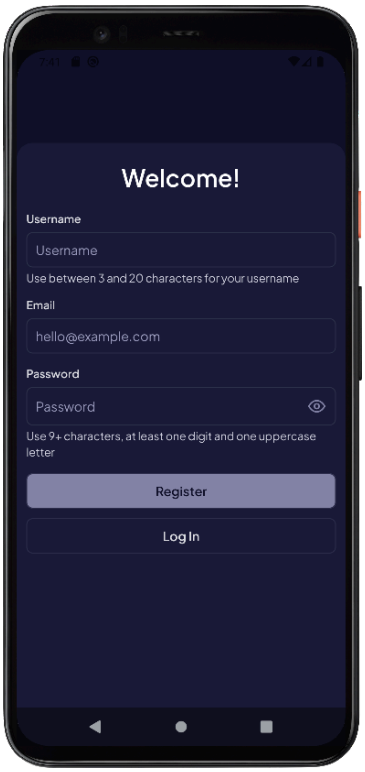

Рисунок 6.1 - Інтерфейс реєстрації

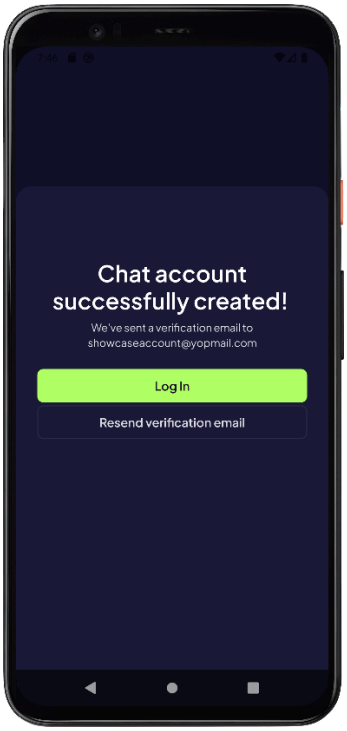

Рисунок 6.2 - Екран підтвердження електроної пошти

Тестування управління чатами охоплює створення особистого та групового чатів, управління учасниками та видалення. Тест створює чат через POST /chats з двома учасниками та перевіряє що у базі даних створено записи у таблицях чатів та учасників. Тест додає нового учасника через POST /chats/{id}/participants та перевіряє оновлення списку. Тест видаляє учасника та перевіряє що він більше не має доступу до чату. Тест видаляє чат та перевіряє каскадне видалення пов'язаних повідомлень та учасників.

Тестування обміну повідомленнями використовує spring-rabbit-test для тестування асинхронної доставки через RabbitMQ. Тест надсилає повідомлення через WebSocket та перевіряє що воно збережено у PostgreSQL з коректними полями відправника, часу та вмісту. Тест перевіряє що подія доставки повідомлення опублікована у RabbitMQ чергу. Тест перевіряє видалення повідомлення та відповідну подію.

Тестування rate limiting з реальним Redis перевіряє коректність взаємодії Lua-скриптів з Redis. Тест ініціалізує лічильник через серію запитів, перевіряє коректне значення лічильника у Redis, перевіряє спрацювання блокування та коректний TTL ключа.

6.4. Тестування WebSocket з'єднання

Тестування WebSocket функціональності, зображене на рисунку 6.3 та рисунку 6.4, має специфіку порівняно зі стандартним HTTP тестуванням. Spring Boot Test надає WebSocketStompClient для тестування WebSocket з'єднань, однак оскільки система використовує Raw WebSocket без STOMP тестування здійснюється через стандартний Java WebSocket клієнт.

Рисунок 6.3 - Інтерфейс списку чатів

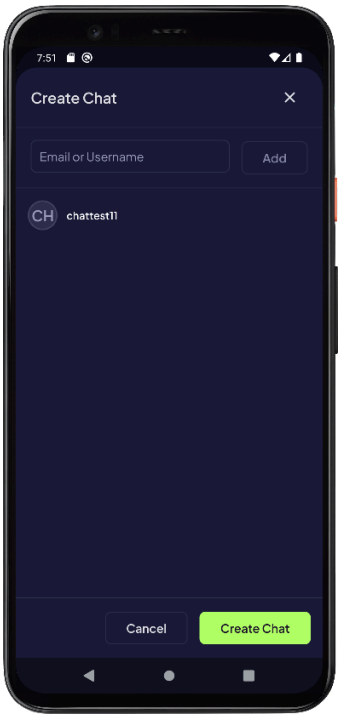

Рис. 6.4 - Інтерфейс створення чату

Тест встановлення з'єднання ,зображений на рисунку 6.5, перевіряє що WebSocket з'єднання успішно встановлюється при передачі валідного JWT токена.

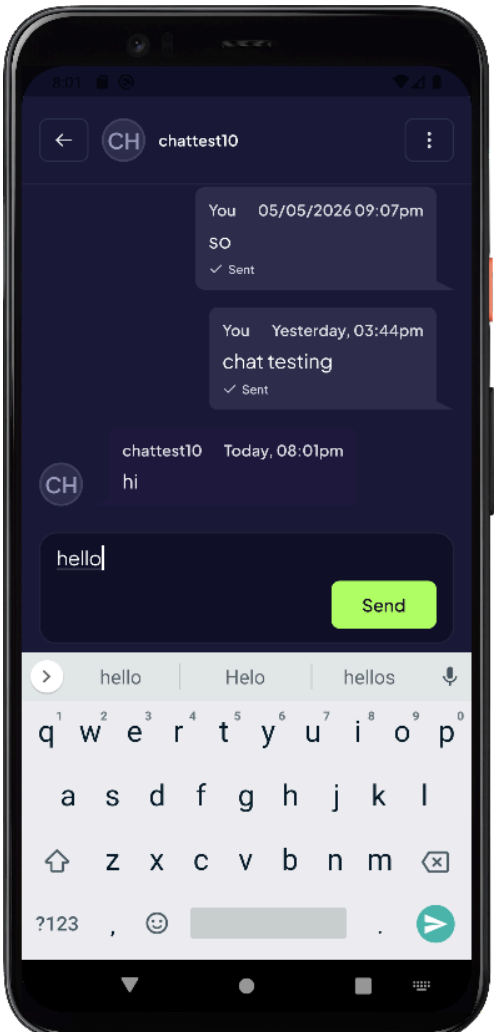

Рисунок 6.5 - Інтерфейс екрану переписки

Тест перевіряє що з'єднання відхиляється при відсутності або невалідності токена з кодом закриття 1008 (Policy Violation). Тест надсилання повідомленняперевіряє що повідомлення відправлене одним клієнтом доставляється всім учасникам чату через окремі WebSocket сесії. Тест перевіряє коректну десеріалізацію JSON у відповідний підтип WebSocketEvent.

6.5. Аналіз результатів тестування

За результатами проведеного тестування всі реалізовані функціональні та безпекові сценарії пройшли успішно. Механізми JWT автентифікації коректно відхиляють невалідні запити на всіх рівнях - відсутній токен, прострочений токен та токен з невалідним підписом. Rate limiting коректно обмежує запити на обох рівнях (IP та email) з атомарним виконанням через Redis Lua-скрипти. Авторизаційні перевірки коректно блокують доступ до чужих ресурсів.

Виявлено напрямки для додаткового тестування у майбутніх ітераціях розробки. Навантажувальне тестування WebSocket з'єднань для визначення максимальної кількості одночасних користувачів при поточній конфігурації серверу є важливим для планування масштабування. Тестування сценаріїв відновлення після збоїв (PostgreSQL недоступний, Redis недоступний, RabbitMQ недоступний) дозволить визначити поведінку системи при часткових збоях інфраструктури. Penetration testing із залученням спеціалістів з безпеки дозволить виявити потенційні вразливості що не покриваються автоматизованими тестами.

У шостому розділі описано стратегію та результати тестування інформаційної системи корпоративного захищеного обміну повідомленнями. Визначено чотири рівні тестування - модульне, інтеграційне, безпекове та функціональне. Детально описано тестування механізмів безпеки що охоплює JWT автентифікацію, rate limiting, авторизацію, захист паролів та верифікацію email - всі 12 тестових сценаріїв пройшли успішно. Описано функціональне тестування повного циклу роботи системи від реєстрації до обміну повідомленнями - всі 12 функціональних сценаріїв пройшли успішно. Розглянуто особливості тестування Raw WebSocket з'єднань. Визначено напрямки для подальшого розширення тестового покриття включаючи навантажувальне тестування та penetration testing.

ВИСНОВКИ

У дипломній роботі розроблено інформаційну систему корпоративного захищеного обміну повідомленнями з архітектурою self-hosted що забезпечує повний контроль підприємства над власними даними без залежності від сторонніх хмарних сервісів. Система вирішує актуальну проблему захисту корпоративних комунікацій в умовах зростання кількості кіберзагроз та посилення вимог законодавства до захисту персональних даних.

У процесі виконання дипломної роботи отримано такі результати:

- проведено аналіз предметної області корпоративного захищеного обміну повідомленнями та побудовано модель загроз за методологією STRIDE що охоплює загрози транспортного рівня, рівня автентифікації, рівня застосунку та рівня даних. Порівняльний аналіз п'яти популярних рішень - Telegram, Viber, WhatsApp, Slack та Signal - підтвердив що жодне з них не задовольняє одночасно вимогам self-hosted розгортання, наскрізного шифрування та повноцінного корпоративного функціоналу;

- спроектовано мікросервісну архітектуру серверної частини на основі Spring Boot з поділом на модулі user, chat та common та шаровою організацією кожного модуля за принципом api/service/domain/infra. Архітектурні рішення самі по собі формують перший рівень захисту через ізоляцію компонентів у Docker мережі, принцип мінімальних привілеїв та ешелонований захист (Defense in Depth);

- реалізовано багаторівневу систему захисту інформації що охоплює захист транспортного рівня через TLS 1.3 для HTTPS та WSS з'єднань з обов'язковою підтримкою Forward Secrecy, схему JWT автентифікації з двома типами токенів де refresh токен зберігається у Redis з можливістю примусової інвалідації, захист паролів алгоритмом BCrypt з адаптивною складністю та унікальною сіллю, дворівневий механізм rate limiting на основі Redis Lua-скриптів з атомарним виконанням для захисту від brute force атак та наскрізне шифрування повідомлень

на основі Signal Protocol що поєднує X3DH для початкового обміну ключами та Double Ratchet для генерації унікального ключа кожного повідомлення;

- розроблено мультиплатформенний клієнтський застосунок на основі Kotlin Multiplatform та Compose Multiplatform з підтримкою платформ Android та Desktop організований за принципами чистої архітектури з реалізацією offline-first стратегії кешування через Room та автоматичним відновленням WebSocket з'єднання через ConnectionRetryHandler та ConnectivityObserver;

- реалізовано повний набір функцій корпоративного месенджера: реєстрацію з підтвердженням електронної пошти, JWT автентифікацію з оновленням токенів, особисті та групові чати, обмін повідомленнями в реальному часі через Raw WebSocket, видалення повідомлень і чатів та управління учасниками групових чатів;

- проведено тестування системи на чотирьох рівнях - модульному, інтеграційному, безпековому та функціональному. Всі 24 тестові сценарії пройшли успішно підтвердивши коректність реалізації механізмів захисту та функціональних вимог;

- система розгортається на власній інфраструктурі підприємства через Docker Compose з єдиною командою запуску що забезпечує простоту впровадження без спеціальних знань адміністрування серверів.

Практична цінність розробленої системи полягає у наданні підприємствам готового рішення для захищеної корпоративної комунікації з повним контролем над даними та інфраструктурою.

Напрямками подальшого розвитку системи є впровадження алгоритму Argon2id замість BCrypt для підвищення стійкості до GPU атак, реалізація Refresh Token Rotation для підвищення безпеки управління сесіями, додавання двофакторної автентифікації через TOTP, розробка веб-клієнта для роботи через браузер, підтримка передачі медіафайлів, реалізація системи аудиту безпеки з централізованим журналом подій та інтеграція з системами SIEM для моніторингу загроз у реальному часі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Software Engineering / Sommerville I. at Pearson Education, 2016. 816 p.
2. Building Microservices: Designing Fine-Grained Systems / Newman S. at c Media, 2021. 616 p.
3. Fundamentals of Software Architecture / Richards M., Ford N. at O'Reilly Media, 2020. 422 p.
4. Designing Data-Intensive Applications / Kleppmann M. at O'Reilly Media, 2017. 562 p.
5. Kubernetes: Up and Running / Burns B., Beda J., Hightower K. at O'Reilly Media, 2022. 326 p.
6. Spring Boot in Action / Walls C. at Manning Publications, 2016. 264 p.
7. Learning Spring Boot 3.0 / Turnquist G. at Packt Publishing, 2022. 270 p.
8. Spring Security 3 / Winch R., Mularien P. at Packt Publishing, 2012. 558 p.
9. Kotlin Programming Cookbook / Jaiswal S. at Packt Publishing, 2018. 360 p.
10. Android Development with Kotlin / Moskala M., Wojda I. at Packt Publishing, 2017. 440 p.
11. High Performance Browser Networking / Grigorik I. at O'Reilly Media, 2013. 364 p. URL: https://hpbn.co (дата звернення: 06.05.2026).
12. Cryptography and Network Security: Principles and Practice / Stallings W. at Pearson Education, 2022. 816 p.
13. Cryptography Engineering: Design Principles and Practical Applications / Ferguson N., Schneier B., Kohno T. at Wiley, 2010. 384 p.
14. Threat Modeling: Designing for Security / Shostack A. at Wiley, 2014. 624 p.
15. Writing Secure Code / Howard M., LeBlanc D. at Microsoft Press, 2002. 784 p.

16. A Future-Adaptable Password Scheme / Provos N., Mazières D. at USENIX Annual Technical Conference, 1999. URL: https://www.usenix.org/legacy/events/usenix99/provos/provos.pdf (дата звернення: 06.05.2026).

17. The Double Ratchet Algorithm / Marlinspike M., Perrin T. at Signal Foundation, 2016. URL: https://signal.org/docs/specifications/doubleratchet/ (дата звернення: 06.05.2026).

18. The X3DH Key Agreement Protocol / Marlinspike M., Perrin T. at Signal Foundation, 2016. URL: https://signal.org/docs/specifications/x3dh/ (дата звернення: 06.05.2026).

19. JSON Web Token (JWT) : RFC 7519 / Jones M., Bradley J., Sakimura N. at Internet Engineering Task Force, 2015. URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 06.05.2026).

20. The WebSocket Protocol : RFC 6455 / Fette I., Melnikov A. at Internet Engineering Task Force, 2011. URL: https://datatracker.ietf.org/doc/html/rfc6455 (дата звернення: 06.05.2026).

21. The Transport Layer Security (TLS) Protocol Version 1.3 : RFC 8446 / Rescorla E. at Internet Engineering Task Force, 2018. URL: https://datatracker.ietf.org/doc/html/rfc8446 (дата звернення: 06.05.2026).

22. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM). NIST Special Publication 800-38D / Dworkin M. at NIST, 2007. URL: https://csrc.nist.gov/publications/detail/sp/800-38d/final (дата звернення: 06.05.2026).

23. Microservices: Yesterday, Today, and Tomorrow / Dragoni N. та ін. at Springer, Present and Ulterior Software Engineering, 2017. С. 195–216.

24. A Study on End-to-End Encrypted Messaging Systems for Enterprise Use / Ueda K., Sato H., Fukuda K. at Journal of Information Security, 2021. Vol. 12, № 3. P. 215–228.

25. A Formal Security Analysis of the Signal Messaging Protocol / Cohn-Gordon K., Cremers C., Dowling B., Garratt L., Stebila D. at Journal of Cryptology, 2020. Vol. 33. P. 1914–1983.

26. Meltdown: Reading Kernel Memory from User Space / Lipp M. та ін. at 27th USENIX Security Symposium, 2018. P. 973–990. URL: https://meltdownattack.com/meltdown.pdf (дата звернення: 06.05.2026).

27. Passwords and the Evolution of Imperfect Authentication / Bonneau J. та ін. at Communications of the ACM, 2015. Vol. 58, № 7. P. 78–87.

28. How to Share a Secret / Shamir A. at Communications of the ACM, 1979. Vol. 22, № 11. P. 612–613.

29. New Directions in Cryptography / Diffie W., Hellman M. at IEEE Transactions on Information Theory, 1976. Vol. 22, № 6. P. 644–654.

30. OWASP Top Ten 2021. URL: https://owasp.org/Top10/ (дата звернення: 06.05.2026).

31. Spring Security Reference. URL: https://docs.spring.io/spring-security/reference/index.html (дата звернення: 06.05.2026).

32. Kotlin Multiplatform Documentation. URL: https://kotlinlang.org/docs/multiplatform.html (дата звернення: 06.05.2026).

33. Compose Multiplatform Documentation. URL: https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-multiplatform-getting-started.html (дата звернення: 06.05.2026).

34. Ktor Client Documentation. URL: https://ktor.io/docs/client-create-new-application.html (дата звернення: 06.05.2026).

35. Room Persistence Library. URL: https://developer.android.com/training/data-storage/room (дата звернення: 06.05.2026).

36. RabbitMQ Tutorials. URL: https://www.rabbitmq.com/tutorials (дата звернення: 06.05.2026).

ДОДАТОК А

Програмна реалізація механізму rate limiting

Програмні засоби:

- мова програмування Kotlin;
- фреймворк Spring Boot;
- сховище даних Redis;
- скриптова мова Lua.

Анотація IpRateLimit для позначення захищених endpoints

```
annotation class IpRateLimit(
    val requests: Int = 60,
    val duration: Long = 1L,
    val timeUnit: TimeUnit = TimeUnit.MINUTES
)
```

Перехоплювач IpRateLimitInterceptor

```
@Component
class IpRateLimitInterceptor(
    private val ipRateLimiter: IpRateLimiter,
    private val ipResolver: IpResolver,
    @param:Value("\${rate-limit.ip.apply-limit}")
    private val applyLimit: Boolean
) : HandlerInterceptor {
    override fun preHandle(
        request: HttpServletRequest,
        response: HttpServletResponse,
        handler: Any
    ): Boolean {
        if (handler is HandlerMethod && applyLimit) {
                                                val    annotation    =
handler.getMethodAnnotation(IpRateLimit::class.java)
            if (annotation != null) {
```

```
                val clientIp = ipResolver.getClientIp(request)
                return try {
                    ipRateLimiter.withIpRateLimit(
                        ipAddress = clientIp,
                        resetsInDuration = Duration.of(
                            annotation.duration,
                            annotation.timeUnit.toChronoUnit()
                        ),
                        maxRequestsPerIp = annotation.requests,
                        action = { true }
                    )
                } catch (e: RateLimitException) {
                    response.sendError(429)
                    false
                }
            }
        }
        return true
    }
}
```

Компонент IpRateLimiter - виконання Lua-скрипту через Redis

```
@Component
class IpRateLimiter(
    private val redisTemplate: StringRedisTemplate
) {
    companion object {
        private const val IP_RATE_LIMIT_PREFIX = "rate_limit:ip"
    }

    @Value("classpath:ip_rate_limit.lua")
    lateinit var rateLimitResource: Resource

    private val rateLimitScript by lazy {
```

```
        val script = rateLimitResource.inputStream.use {
            it.readBytes().decodeToString()
        }
        @Suppress("UNCHECKED_CAST")
                    DefaultRedisScript(script,  List::class.java  as
Class<List<Long>>)
    }

    fun <T> withIpRateLimit(
        ipAddress: String,
        resetsInDuration: Duration,
        maxRequestsPerIp: Int,
        action: () -> T
    ): T {
        val key = "$IP_RATE_LIMIT_PREFIX:$ipAddress"
        val result = redisTemplate.execute(
            rateLimitScript,
            listOf(key),
            maxRequestsPerIp.toString(),
            resetsInDuration.seconds.toString()
        )
        val currentCount = result[0]
        return if (currentCount <= maxRequestsPerIp) {
            action()
        } else {
            val ttl = result[1]
            throw RateLimitException(ttl)
        }
    }
}
```

Lua-скрипт ip_rate_limit.lua - атомарна перевірка ліміту за IP

```
lua
local key = KEYS[1]
```

```
local maxRequests = tonumber(ARGV[1])
local ttl = tonumber(ARGV[2])

local current = redis.call('GET', key)

if current == false then
    redis.call('SET', key, 1, 'EX', ttl)
    return {1, ttl}
else
    local count = tonumber(current)
    if count < maxRequests then
        local newCount = redis.call('INCR', key)
        local remainingTtl = redis.call('TTL', key)
        return {newCount, remainingTtl}
    else
        local remainingTtl = redis.call('TTL', key)
        return {count + 1, remainingTtl}
    end
end
```

Компонент EmailRateLimiter - обмеження запитів за електронною адресою

```
@Component
class EmailRateLimiter(
    private val redisTemplate: StringRedisTemplate
) {
    companion object {
                    private const val EMAIL_RATE_LIMIT_PREFIX =
"rate_limit:email"
                  private const val EMAIL_ATTEMPT_COUNT_PREFIX =
"email_attempt_count"
    }

    @Value("classpath:email_rate_limit.lua")
    lateinit var rateLimitResource: Resource
```

```
    private val rateLimitScript by lazy {
        val script = rateLimitResource.inputStream.use {
            it.readBytes().decodeToString()
        }
        @Suppress("UNCHECKED_CAST")
                    DefaultRedisScript(script,  List::class.java  as
Class<List<Long>>)
    }

    fun withRateLimit(
        email: String,
        action: () -> Unit
    ) {
        val normalizeEmail = email.lowercase().trim()
                                            val     rateLimitKey     =
"$EMAIL_RATE_LIMIT_PREFIX:$normalizeEmail"
                                          val     attemptCountKey     =
"$EMAIL_ATTEMPT_COUNT_PREFIX:$normalizeEmail"
        val result = redisTemplate.execute(
            rateLimitScript,
            listOf(rateLimitKey, attemptCountKey)
        )
        val attemptCount = result[0]
        val ttl = result[1]
        if (attemptCount == -1L) {
            throw RateLimitException(resetsInSeconds = ttl)
        }
        action()
    }
}
```

Lua-скрипт email_rate_limit.lua - прогресивне блокування за електронною адресою

```
local rateLimitKey = KEYS[1]
local attemptCountKey = KEYS[2]
if redis.call('EXISTS', rateLimitKey) == 1 then
    local ttl = redis.call('TTL', rateLimitKey)
    return {-1, ttl > 0 and ttl or 60}
end

local currentCount = redis.call('GET', attemptCountKey)
currentCount = currentCount and tonumber(currentCount) or 0

local newCount = redis.call('INCR', attemptCountKey)

local backoffSeconds = {60, 300, 3600}
local backoffIndex = math.min(currentCount, 2) + 1

redis.call('SETEX', rateLimitKey, backoffSeconds[backoffIndex], '1')
redis.call('EXPIRE', attemptCountKey, 86400)

return {newCount, 0}
```

Компонент IpResolver - визначення реальної IP-адреси клієнта

```
@Component
class IpResolver(
    private val nginxConfig: NginxConfig
) {
    companion object {
        private val PRIVATE_IP_RANGES = listOf(
            "10.0.0.0/8",
            "172.16.0.0/12",
            "192.168.0.0/16",
            "127.0.0.0/8",
            "::1/128",
            "fc00::/7",
            "fe80::/10"
```

```
        ).map { IpAddressMatcher(it) }

        private val INVALID_IPS = listOf(
            "unknown", "unavailable", "0.0.0.0", "::"
        )
    }
                                private       val       logger       =
LoggerFactory.getLogger(IpResolver::class.java)
          private  val  trustedMatchers:  List<IpAddressMatcher>  =
nginxConfig
        .trustedIp
        .filter { it.isNotBlank() }
        .map { proxy ->
            val cidr = when {
                proxy.contains("/") -> proxy
                proxy.contains(":") -> "$proxy/128"
                else -> "$proxy/32"
            }
            IpAddressMatcher(cidr)
        }
    fun getClientIp(request: HttpServletRequest): String {
        val remoteAddr = request.remoteAddr
        if (!isFromTrustedProxy(remoteAddr)) {
            if (nginxConfig.requireProxy) {
                        logger.warn("Direct  connection  attempt  from
$remoteAddr")
                 throw SecurityException("No valid client IP in proxy
headers")
            }
            return remoteAddr
        }
        val clientIp = extractFromXRealIp(request, remoteAddr)
        if (clientIp == null) {
            logger.warn("No valid client IP in proxy headers")
```

```
            if (nginxConfig.requireProxy) {
                 throw SecurityException("No valid client IP in proxy
headers")
            }
        }
        return clientIp ?: remoteAddr
    }

    private fun extractFromXRealIp(
        request: HttpServletRequest,
        proxyIp: String
    ): String? {
        return request.getHeader("X-Real-IP")?.let { header ->
            validateAndNormalizeIp(header, "X-Real-IP", proxyIp)
        }
    }

    private fun validateAndNormalizeIp(
        ip: String,
        headerName: String,
        proxyIp: String
    ): String? {
        val trimmedIp = ip.trim()
          if (trimmedIp.isBlank() || INVALID_IPS.contains(trimmedIp))
{
             logger.debug("Invalid IP in $headerName: $ip from proxy
$proxyIp")
            return null
        }
        return try {
            val inetAddr = when {
                trimmedIp.contains(":") ->
                    Inet6Address.getByName(trimmedIp)
```

```
trimmedIp.matches(Regex("\\d+\\.\\d+\\.\\d+\\.\\d+")) ->
                    Inet4Address.getByName(trimmedIp)
                else -> {
                     logger.warn("Invalid IP format in $headerName: "
+
                        "$trimmedIp from proxy $proxyIp")
                    return null
                }
            }
            if (isPrivateIp(inetAddr.hostAddress)) {
                logger.debug("Private IP in $headerName: " +
                    "$trimmedIp from proxy $proxyIp")
            }
            inetAddr.hostAddress
        } catch (e: Exception) {
            logger.warn("Invalid IP format in $headerName: " +
                "$trimmedIp from proxy $proxyIp", e)
            null
        }
    }

    private fun isPrivateIp(ip: String): Boolean =
        PRIVATE_IP_RANGES.any { it.matches(ip) }

    private fun isFromTrustedProxy(ip: String): Boolean =
        trustedMatchers.any { matcher -> matcher.matches(ip) }
}
```