Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Мобільна система обміну миттєвими повідомленнями з наскрізним шифруванням на платформі Android

Назва теми

Рівень вищої освіти Перший (бакалаврський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр КвРІПЗ. 220199.01.04.ПЗ

Виконав студент IV курсу, група ІПЗ-22-1 _____ Дарина ГАВРИЛЮК
Підпис Ім'я, ПРІЗВИЩЕ

Керівник асистентка _____ Анастасія ДЬОМІНА
Науковий ступінь, вчене звання Підпис Ім'я, ПРІЗВИЩЕ

Нормоконтролер канд. техн. наук, доцент _____ Юрій ФОРКУН
Підпис Ім'я, ПРІЗВИЩЕ

До захисту допускаю:
Завідувач кафедри інженерії програмного забезпечення _____ Леонід БЕДРАТЮК
Підпис Ім'я, ПРІЗВИЩЕ

1 червня 2026 р.

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Інженерії програмного забезпечення

Рівень вищої освіти Перший (бакалаврський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
_____ Л. П. Бедратюк
02 01 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Гаврилюк Дарині Олегівні

Прізвище, ім'я, по батькові студента

1. Тема роботи «Мобільна система обміну миттєвими повідомленнями з наскрізним шифруванням на платформі Android»

Керівник роботи Дьоміна Анастасія Іванівна, асистентка

Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 20.01.2026 р. № 7

2. Строк подання студентом роботи на кафедру 01.06.2026 р.

3. Вихідні дані до роботи Матеріали переддипломної практики

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити): дослідження предметної області, проєктування програмного забезпечення, програмна реалізація та тестування програмного забезпечення,

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень

1. Діаграма діяльності
2. Діаграма станів сесії користувача
3. Діаграма станів повідомлення
4. Блок-схема алгоритму шифрування RSA

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Форкун Ю. В., канд. техн. наук, доцент	05.05.26	06.05.26
Антиплагіат	Форкун Ю. В., канд. техн. наук, доцент	05.05.26	06.05.26

7. Дата видачі завдання «_02_» __січня_ 2026 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1 Ознайомлення з тематикою дипломного проєктування, визначення та узгодження індивідуальних тем кваліфікаційних робіт	01.12 – 31.12.2025	
2 Збір матеріалу за тематикою КвР; дослідження предметної області, в якій планується використання програмного забезпечення (ПЗ), визначення задач та вимог, розробка технічного завдання	01.01 – 20.02.2026	
3 Проєктування програмного забезпечення	21.02 – 20.03.2026	
4 Програмна реалізація з використанням відповідних засобів розробки. Тестування ПЗ	21.03 – 30.04.2026	
5 Написання вступу, загальних висновків, оформлення переліку джерел посилання та додатків. Оформлення пояснювальної записки КвР згідно вимог	01.05 – 25.05.2026	
6 Попередній захист КвР	Травень 2026	Згідно графіка
7 Перевірка КвР на плагіат, нормоконтроль, отримання відгуків, рецензій та інших супровідних документів. Брошурування (зшиття) пояснювальної записки	26.05 – 30.06.2026	
8 Здача КвР на кафедру; підготовка КвР для розміщення у репозитарії ХНУ; підготовка до захисту та захист КвР	З 01.06.2026	

Студент _____ Дарина ГАВРИЛЮК
Підпис Ім'я, ПРІЗВИЩЕ

Керівник роботи _____ Анастасія ДЬОМІНА
Підпис Ім'я, ПРІЗВИЩЕ

АНОТАЦІЯ

Тема кваліфікаційної роботи «Мобільна система обміну миттєвими повідомленнями з наскрізним шифруванням на платформі Android».

Авторка роботи: Гаврилюк Дарина Олегівна.

Керівник роботи: Дьоміна Анастасія Іванівна.

Пояснювальна записка: 74 с., 22 рис., 5 табл., 4 дод., 41 джерел.

Графічна частина: 4 креслення ф. А3, 14 слайдів.

ANDROID, МЕСЕНДЖЕР, КІБЕРБЕЗПЕКА, НАСКРІЗНЕ ШИФРУВАННЯ, E2EE, КРИПТОГРАФІЯ, ZERO TRUST, ПОВІДОМЛЕННЯ, ІНТЕРФЕЙС, ЗАЛЕЖНОСТІ.

Мета кваліфікаційної роботи: проєктування мобільної системи миттєвого обміну повідомленнями на платформі Android з наскрізним шифруванням.

У кваліфікаційній роботі досліджено проблему захисту конфіденційних даних у мобільних мережах, проведено аналіз предметної області та її інформаційного забезпечення, визначені функціональні та нефункціональні вимоги до програмної системи, розроблена загальна архітектура та структура застосунку з мінімізацією збереження метаданих на сервері.

Для реалізації програмного продукту використано мову програмування Kotlin, фреймворки Jetpack Compose та Room Database.

Практичне значення результатів роботи полягає в тому, що використання мобільного застосунку дозволяє впровадити надійний інструмент для конфіденційного обміну даними, раціоналізувати систему захисту інформації шляхом апаратного збереження криптографічних ключів та оптимізувати безпеку комунікацій у приватному, бізнесовому та державному секторах.

29/05/2026
Дата

Підпис

ВІДОМІСТЬ ДОКУМЕНТІВ

№ рядка	Формат	Позначення документа	Найменування документа	К-сть аркушів	№ екз.	Примітка
			Текстові документи			
1	А4	КвРІПЗ. 220199.01.04.ПЗ	Пояснювальна записка	74		
2	А4		Завдання на кваліфіка-	1		
			ційну роботу			
3	А4		Анотація	1		
			Графічні документи			
4	А3	КвРІПЗ. 220199.01.04.Е8	Діаграма діяльності	1		
5	А3	КвРІПЗ. 220199.01.04.Е8	Діаграма станів сесії	1		
			користувача			
6	А3	КвРІПЗ. 220199.01.04.Е8	Діаграма станів	1		
			повідомлення			
7	А3	КвРІПЗ. 220199.01.04.Е8	Блок-схема алгоритму	1		

					КвРІПЗ.220199.01.04.ПЗ		
Змін.	Арк.	№ докум.	Підпис	Дат			
Виконав		Гаврилюк Д. О.		29/05/2026	*Мобільна система обміну миттєвими повідомленнями з наскрізним шифруванням на платформі Android*	Літ.	Арк. 1 / Аркушів 1
Керівник		Дьоміна А. І.		29/05/2026			
Рецензент						*ХНУ, ІПЗ-22-1*	
Н. контр.		Форкун Ю.В.		29.05			
Зав. каф.		Бевратюк Л. П.		29/05/2026			

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

DI	–	Dependency Injection
E2EE	–	End-to-end encryption
UI	–	User Interface
UML	–	Unified Modeling Language
ПЗ	–	Програмне забезпечення
ТЗ	–	Технічне завдання

ВСТУП

В Україні приблизно 82,4% населення користуються інтернетом [2]. Це свідчить про те, що величезні обсяги інформації передаються мережею, причому ці дані не завжди захищені надійними протоколами. Важливо зауважити, що 91% українців споживають контент саме через смартфони, водночас 67% - це пристрої на базі ОС Android [3].

Традиційні методи захисту на рівні транспортного протоколу часто виявляються недостатніми та недосконалими, оскільки залишають дані вразливими на боці сервера провайдера послуг. Це підтверджується статистикою, згідно з якою кібератаки на сервери стали причиною 85% усіх витоків даних. Це доводить, що сервери є «слабкою ланкою» у сучасних системах захисту [4].

Отже, проблема полягає у відсутності у користувачів повного контролю над власними даними при використанні централізованих месенджерів. Більшість популярних платформ зберігають ключі шифрування або метадані на своїх серверах, що створює ризик несанкціонованого доступу з боку адміністраторів сервісу або зловмисників у разі зламів.

Актуальність теми кваліфікаційної роботи визначається об’єктивною потребою у створенні програмного забезпечення, яке реалізує архітектуру «нульової довіри». Впровадження наскрізного шифрування є критично важливим для забезпечення конституційного права громадян на таємницю листування та захисту комерційної таємниці в умовах нестабільного кіберпростору.

В сучасних реаліях реалізація наскрізного шифрування повідомлень для пристроїв на базі Android вимагає комплексного підходу, адже потрібно враховувати особливості платформи, рівень її захисту та апаратні компоненти. Цей виклик характеризується необхідністю балансування між високою криптографічною стійкістю та обмеженими ресурсними можливостями мобільних пристроїв. Попри наявність комерційних месенджерів, існує дефіцит

відкритих, прозорих та спеціалізованих систем, які дозволяють зрозуміти, які алгоритми захисту конфіденційних даних було використано та дозволять уникнути проблем витоку критично важливих даних, що в деяких галузях мають вирішальну роль та можуть спричинити великі проблеми.

Обґрунтуванням необхідності вирішення завдання є створення власної системи обміну повідомленнями з наскрізним шифруванням, що дозволить усунути залежність від закордонних провайдерів та забезпечити повну прозорість алгоритмів захисту. Це важливо для державного сектору, військових структур та бізнес-організацій, де вимоги до конфіденційності є найвищими.

Розроблювана система призначена для забезпечення безпечного текстового та мультимедійного обміну даними між користувачами мобільної платформи Android у різних сферах: від приватного використання до державних та військових установ, що в сучасних реаліях України відіграє вирішальну роль.

Метою роботи є проєктування та розробка мобільної системи миттєвого обміну повідомленнями на платформі Android, яка забезпечує гарантовану конфіденційність та цілісність даних користувачів шляхом впровадження протоколів наскрізного шифрування та захисту криптографічних ключів.

Об’єктом дослідження є процес обміну миттєвими повідомленнями та забезпечення конфіденційності даних у мобільних мережах передачі інформації.

Предмет дослідження - криптографічні протоколи, їх реалізація на платформі Android та методи апаратного захисту криптографічних ключів.

Основні завдання моєї проєктної роботи: аналіз сучасних методів та протоколів наскрізного шифрування в мобільних системах, особливості предметної області та технічні обмеження платформи Android щодо реалізації безпечного обміну даними, архітектура мобільної системи, включаючи клієнт-серверну взаємодію з мінімізацією збереження метаданих на сервері та проведення тестування розробленого ПЗ на предмет відповідності вимогам безпеки та оцінити швидкодію системи при роботі з різними типами даних.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Змістовий аналіз предметної області, її структурних та функціональних особливостей

Сучасний стан обраної мною предметної області суттєво змінюється під впливом новітніх технологій та алгоритмів шифрування. Дослідження, проведене у 2023 році та опубліковане в журналі MDPI Technologies [5], демонструє, що більшість сучасних систем безпечного обміну повідомленнями базуються на комбінації алгоритмів Double Ratchet та X3DH. Автори підкреслюють, що ключовим викликом для мобільних платформ залишається забезпечення цілісності метаданих та стійкість до атак на централізовану інфраструктуру, що підтверджує доцільність розробки систем з архітектурою «нульової довіри».

Проте, попри появу нових математичних моделей, фундаментальним завданням залишається забезпечення конфіденційності в мережах загального користування, де основним вектором атак є не стільки перебір ключів, скільки компрометація проміжних вузлів. У цьому контексті криптографія еволюціонувала від простого шифрування каналу зв’язку до концепції наскрізного шифрування, де зміст повідомлення інформації залишається недоступним для зловмисників.

Адже в традиційній архітектурі, де сервер виступає «довіреним джерелом», це створює критичну вразливість до несанкціонованого вторгнення і використання ключів шифрування (Рисунок 1.1).

Окрему увагу слід приділити специфіці апаратного забезпечення цільових платформ. Оскільки в обраній предметній області йдеться про функціонування системи на мобільних пристроях під керуванням ОС Android, які мають обмежені обчислювальні ресурси та лімітовану місткість акумулятора, виникає потреба в суворому балансуванні.

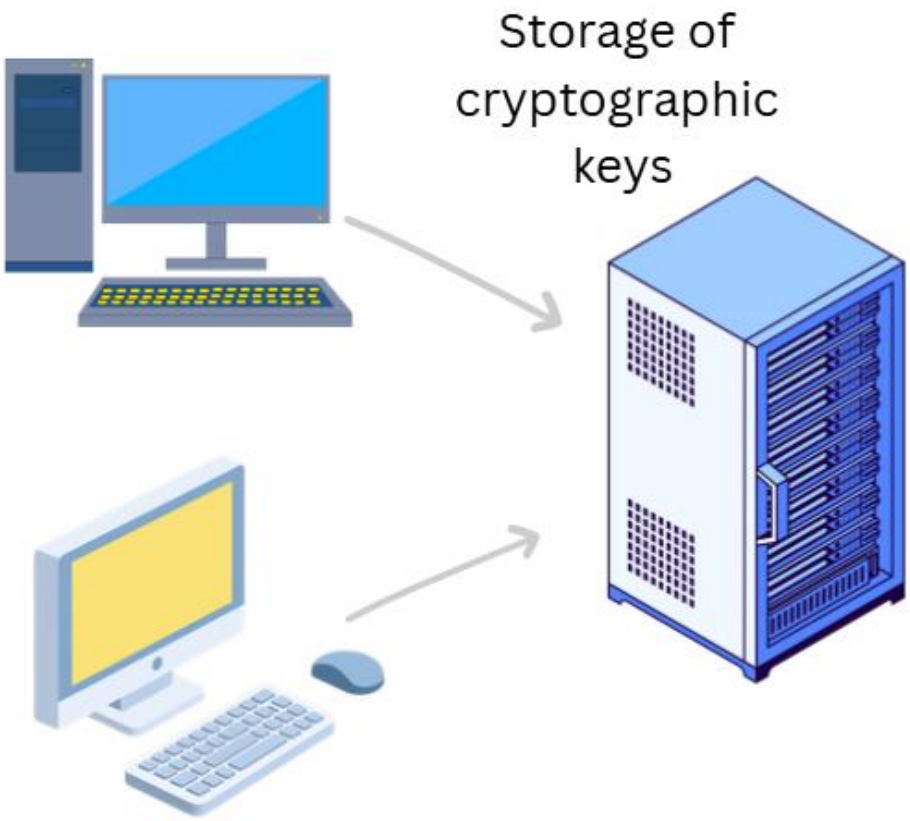


Рисунок 1.1 – «Архітектура «довіреного джерела»»

Особливостями обраної предметної області є те, що її функціональність базується на забезпеченні трьох критичних властивостей: конфіденційності, цілісності та автентичності. Для того щоб краще зрозуміти логіку роботи з такими системами, важливо розглянути життєвий цикл інформаційної взаємодії, який складається з наступних етапів: реєстрація ключів, встановлення сесії.

Процес реєстрації ключів – це процес створення приватних та публічних ключів, які будуть використані для криптографічних операцій [6]. Такі ключі мають під собою чисту математику і часто становлять собою незворотні функції, для обчислення яких у реверсивному порядку знадобиться така кількість часу, що не вистачить всього часу існування Всесвіту, щоб це здійснити.

Приватний ключ зберігається виключно в захищеному сховищі пристрою користувача. Він ніколи не передається мережею. Його функція – дешифрування вхідних повідомлень та створення цифрових підписів.

Публічний ключ відкритий для всіх. Він завантажується на сервер і використовується іншими користувачами для зашифрування повідомлень, які зможе прочитати лише власник відповідного приватного ключа.

Для забезпечення асинхронного встановлення спільного секрету в комерційних проєктах використовується протокол X3DH, який базується на механізмі попередньо опублікованих ключів і таким чином забезпечує встановлення сесії [7]. У месенджерах цей процес складніший, ніж у звичайному вебзастосунку, оскільки він має бути асинхронним: сесія повинна успішно створитися, навіть якщо отримувач у цей момент офлайн.

Перед встановленням сесії кожен з учасників має створити по парі приватного та публічного ключів, адже без них сесія не може бути встановлена. Далі користувачі мають обмінятись публічними ключами та надалі використовувати їх для шифрування повідомлень один одному. І в такому випадку тепер обидва девайси мають спільний ключ K, який жодного разу не проходив через сервери – сесія вважається відкритою.

До того ж аналіз був би неповним без врахування специфіки ОС Android. Платформа висуває жорсткі вимоги до енергоефективності та управління фоновими процесами, що впливає на доставлення push-повідомлень у реальному часі та їхню вчасну обробку.

З точки зору безпеки, предметна область також включає вивчення сховища ключів Android. Це апаратно-програмний комплекс, який дозволяє виконувати криптографічні операції всередині «пісочниці». Таким чином, ключі ніколи не потрапляють у простір оперативної пам'яті застосунку в чистому вигляді, що нівелює загрозу від програм-шпигунів.

Таким чином, сучасний ринок пропонує широкий спектр інструментів та методологій для захисту інформації в мережах загального користування. Проте ефективність їхнього застосування критично залежить від специфіки конкретної програмно-апаратної платформи. З огляду на це, виникає необхідність детального аналізу існуючих рішень та практичних кейсів їх впровадження.

1.2 Аналіз наявного програмно-технічного забезпечення предметної області

В цьому розділі буде розглянуто готові рішення в предметній області для глибшого розуміння технологій, які будуть використовуватись при проєктуванні застосунку. Адже згідно з дослідженням 35% проєктів провалюються, бо створюють продукт, який «не потрібен ринку» [8].

Головний конкурент на ринку захищених месенджерів – це «Signal». Його ключовою характеристикою є те, що це кросплатформний месенджер з відкритим вихідним кодом, розроблений організацією «Signal Technology Foundation». Його головним призначенням є забезпечення максимально приватного спілкування, де навіть розробник не має доступу до даних користувачів.

Користувацький інтерфейс Signal відзначається мінімалізмом. Дизайн використовує стриману палітру кольорів та плавні анімації, які не навантажують процесор. Головне вікно містить список чатів без зайвих елементів. Вікно діалогу надає стандартні функції: обмін текстом, голосом, медіафайлами (Рисунок 1.2).

Ключовими перевагами месенджера «Signal» є те, що він реалізовує еталонну безпеку за допомогою «Signal Protocol», який є стандартом де-факто в індустрії.

Також його перевагою є те, що клієнтська та серверна частини є відкритими, що дозволяє незалежний аудит. До того ж сервер зберігає лише дату реєстрації та дату останнього входу користувача, що дозволяє мінімізувати метадані, які зберігаються на серверах.

Аналіз архітектури застосунку дозволяє виявити кілька суттєвих деструктивних чинників, першим з яких є використання номера телефону як єдиного ідентифікатора клієнта.

Крім того, у системі не реалізовано хмарний бекап чатів, що зумовлено специфікою розподілу ключів шифрування.

Усі криптографічні операції відбуваються локально, тому серверна інфраструктура не має доступу до закритих ключів користувачів і не може дешифрувати їхні архіви.

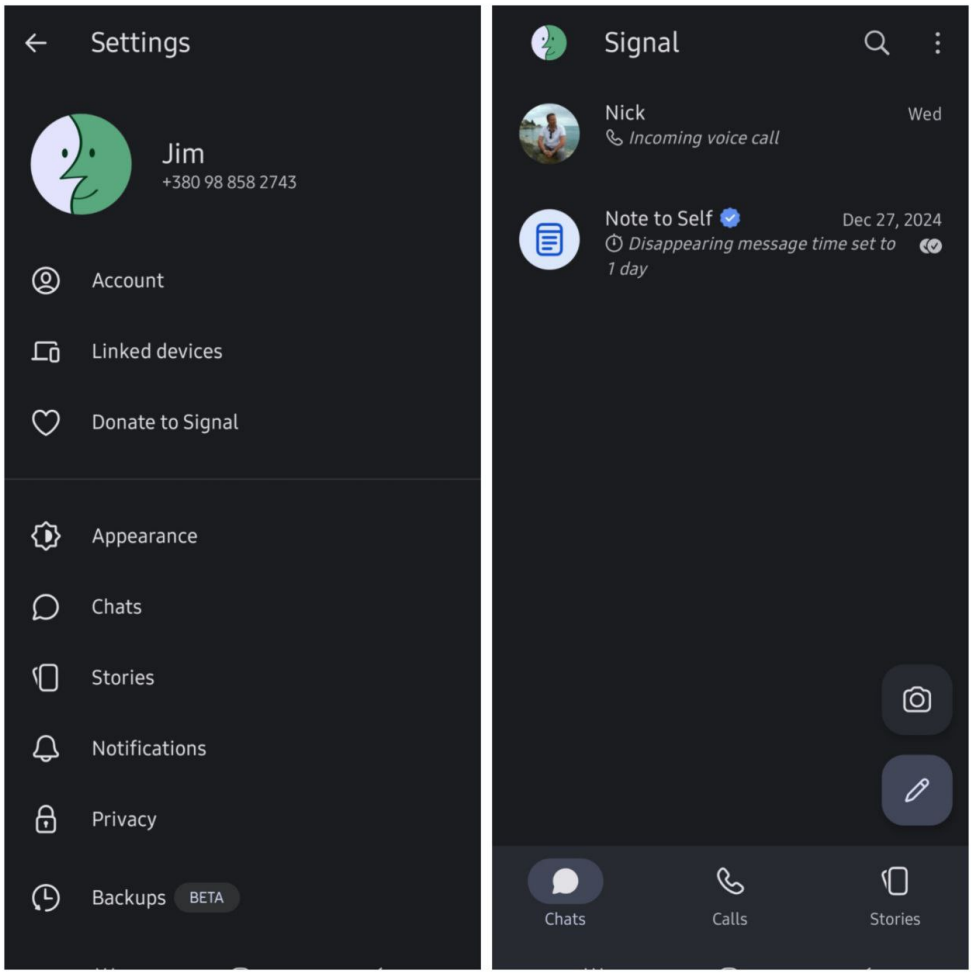


Рисунок 1.2 – «Інтерфейс застосунку «Signal»»

На ринку месенджерів існує ще один відомий застосунок, який має назву «WhatsApp». Згідно з даними, «WhatsApp» входить до трійки найпопулярніших месенджерів, проте поступається Telegram за динамікою росту аудиторії під час повномасштабного вторгнення [9].

Найпопулярніший месенджер у світі, розроблений компанією «Meta Platforms». Призначений для масового використання та дзвінків, а також бізнес комунікацій та обміну мультимедійними файлами різного формату.

Інтерфейс поділений на вкладки «Чати», «Статус» та «Дзвінки». Дизайн орієнтований на простоту та доступність для користувачів. В палітрі кольорів переважає зелений колір, що створює в користувача відчуття довіри (Рисунок 1.3).

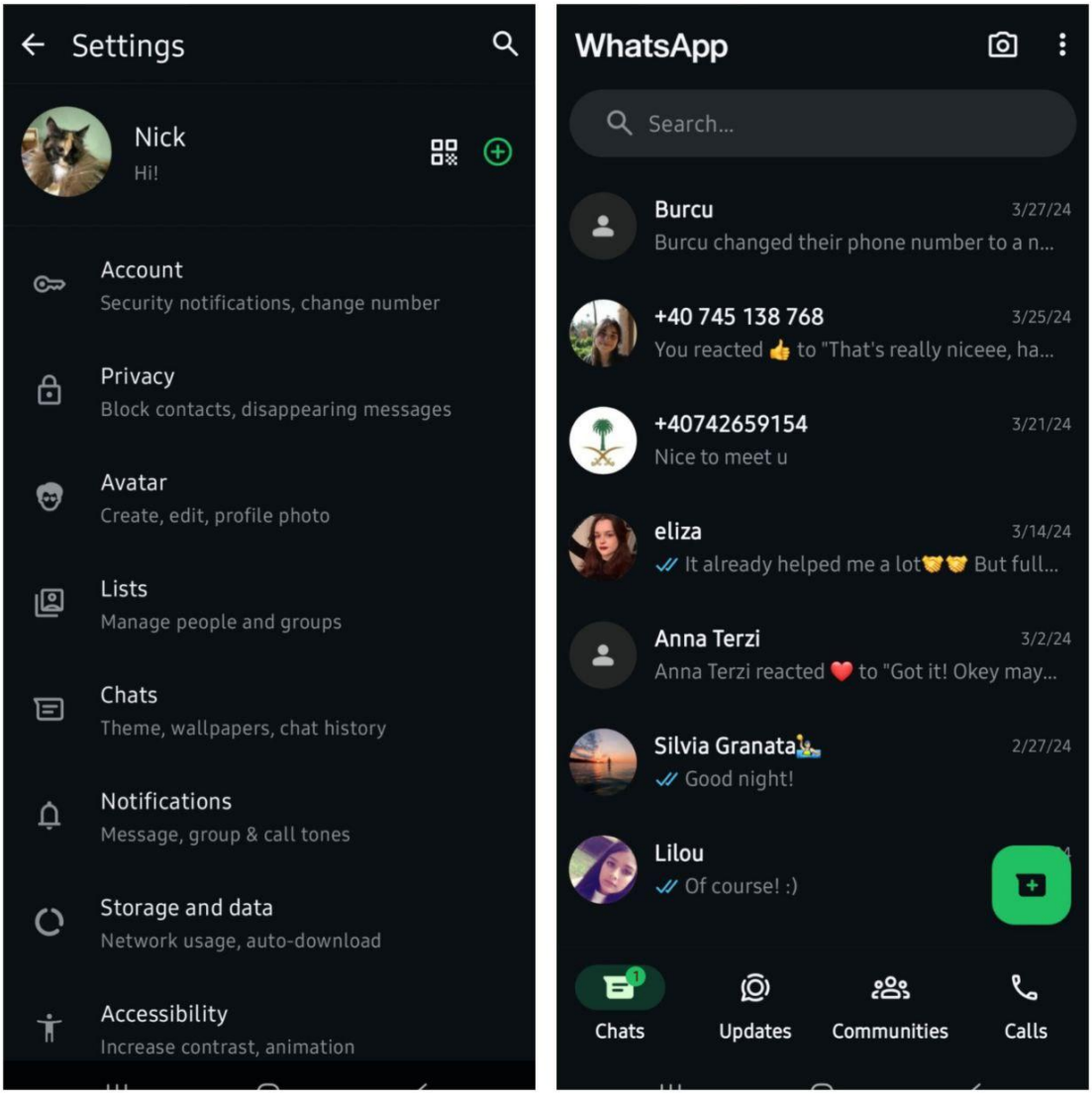


Рисунок 1.3 – «Інтерфейс застосунку «WhatsApp»»

Перевагою цього застосунку є те, що тут використовується імплементація «Signal Protocol» для всіх чатів за замовчуванням та він є стандартом комунікації в багатьох країнах.

Недоліком слід вважати, що «Meta» збирає величезну кількість метаданих про місцеперебування та комунікацію користувачів для рекламного

профілювання. Ба більше мінусом є те, що хмарні копії часто зберігаються без E2EE, якщо користувач сам не обрав варіант шифрувати його дані.

Також окремої уваги заслуговує месенджер «Telegram». Аналіз статистичних даних вказує, що «Telegram» є безальтернативним джерелом оперативної інформації для 51% населення України [10].

Разом з тим цей застосунок використовується понад 90% українців, що робить його найпопулярнішим месенджером в Україні [11].

Хмарний месенджер, розроблений компанією «Telegram FZ-LLC». Фокусується на швидкості, синхронізації між пристроями та соціальних функціях: канали, боти та так далі.

Має найбільш гнучкий інтерфейс серед аналогів: підтримка папок для чатів, кастомізація тем, анімовані наліпки. Це дозволяє користувачу використовувати ці функції для його потреб (Рисунок 1.4).

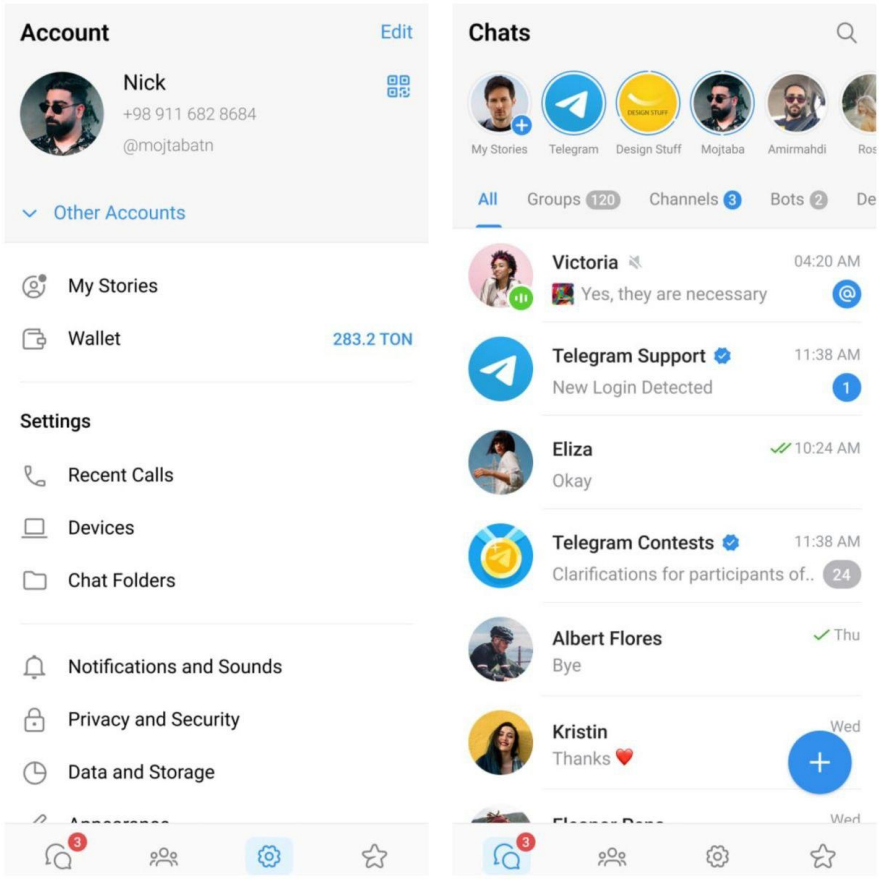


Рисунок 1.4 – «Інтерфейс застосунку «Telegram»»

Перевагами цього застосунку є те, що протокол MTProto оптимізований для роботи в умовах слабкого зв'язку. У періоди блекаутів або перевантаження мереж «Telegram» часто залишається єдиним месенджером, який здатен відправити текстове повідомлення. Водночас він має функціонал миттєвої синхронізації та передачі великих файлів.

Недоліком «Telegram» є те, що звичайні чати зберігаються на серверах компанії та наскрізне шифрування доступне лише в режимі «Секретних чатів».

Поряд з цим попри популярність Telegram, незалежні аудити безпеки вказують на суттєві криптографічні недоліки його власного протоколу MTProto 2.0. Зокрема, у дослідженні доведено, що протокол наскрізного шифрування Telegram є вразливим до атак підміни алгоритмів. Дослідники підкреслюють, що через особливості обробки випадкового заповнення, зловмисники або державні структури можуть здійснювати масове спостереження за приватними комунікаціями, особливо при використанні неофіційних сторонніх клієнтів. Це ще раз підтверджує ризикованість використання "саморобних" криптографічних рішень замість загальновизнаних стандартів [12].

Наступним кандидатом для аналізу готових рішень є месенджер «Viber» розроблений ізраїльсько-білоруською командою і, який зараз належить японській корпорації Rakuten. У контексті українського ринку Viber є критично важливою інфраструктурою, оскільки він інтегрований у державні сервіси, освіту та бізнес-комунікації. Цей застосунок є один із найпопулярніших месенджерів у Східній Європі та Україні зокрема.

Користувацький інтерфейс Viber спроєктований за принципом "все в одному", через що його часто критикують за надмірну перенасиченість елементами керування. Окрім класичних чатів, месенджер містить інтегровані блоки реклами, магазини наліпок, новинні канали та платіжні сервіси, що суттєво перевантажує екранний простір. Такий підхід не лише ускладнює ергономіку застосунку, а й може збивати з пантелику пересічного користувача, відволікаючи його від основної функції – безпечного обміну повідомленнями (Рисунок 1.5).

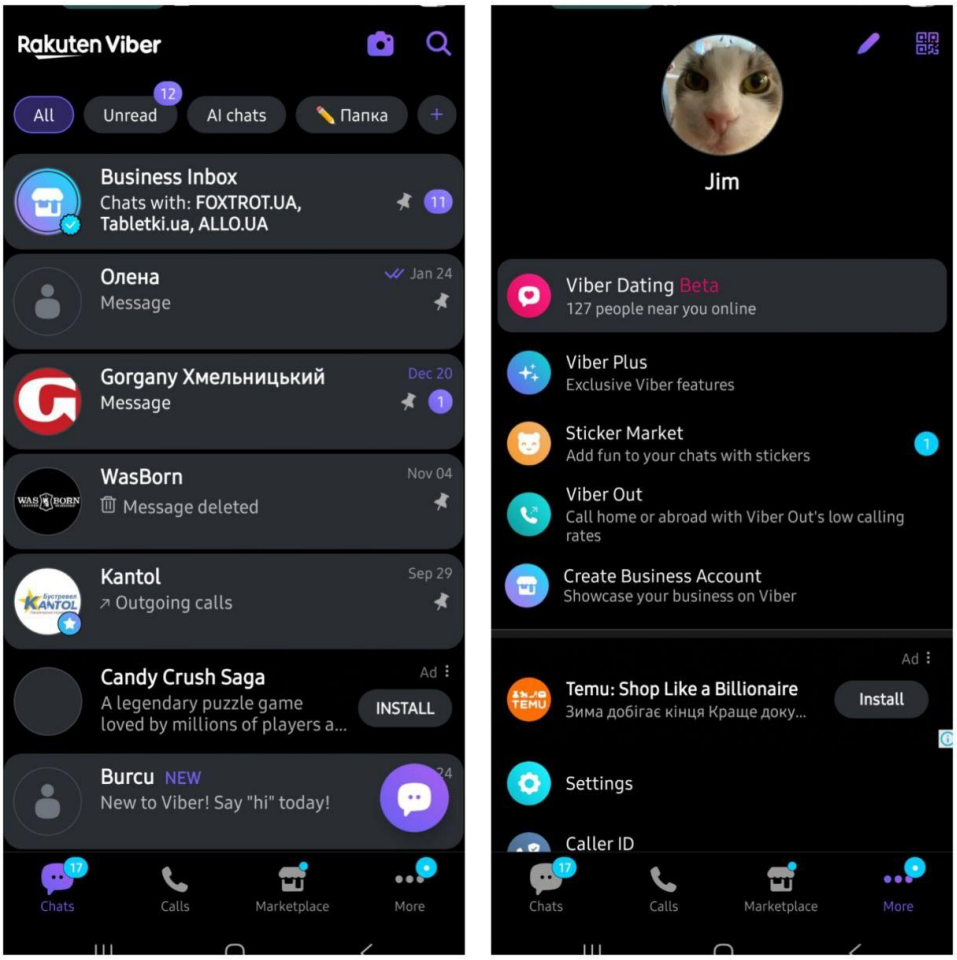


Рисунок 1.5 – «Інтерфейс застосунку «Viber»»

Перевага застосунку «Viber», полягає в тому, що він встановлений на 98% смартфонів українців, що робить його стандартом для побутових комунікацій [13]. Також слід зауважити, що «Viber» використовує власний протокол, який базується на концепціях «Signal Protocol», для захисту особистих чатів та дзвінків.

Натомість попри використання архітектури «Open Whisper Signal», пропрієтарні протоколи масових месенджерів, таких як «Viber», залишаються вразливими до методів аналізу трафіку. Згідно з дослідженнями, специфічні мережеві підписи дозволяють з високою точністю ідентифікувати тип комунікації: аудіо, відео, дзвінки, передача файлів. В результаті чого відбувається деанонімізування учасників обміну [14].

Наступним недоліком є те, що Viber збирає значну кількість метаданих для таргетування реклами, що знижує загальний рівень приватності.

Наступним готовим рішенням для аналізу є швейцарський месенджер «Threema», який був розроблений компанією «Threema GmbH». Його основна філософія - повна анонімність. Він не вимагає прив’язки до номера телефону чи електронної пошти.

Його інтерфейс можна описати двома словами лаконічний та функціональний. Основна особливість – наявність індикатора рівня довіри у вигляді трьох крапок, який показує, як саме був верифікований контакт, у такому випадку сканування QR-коду при зустрічі дає найвищий рівень (Рисунок 1.6).

Threema забезпечує конфіденційність через два незалежні шари шифрування: транспортний та наскрізний між користувачами, де ключовим елементом є протокол Ibex, що пройшов формальну верифікацію [15].

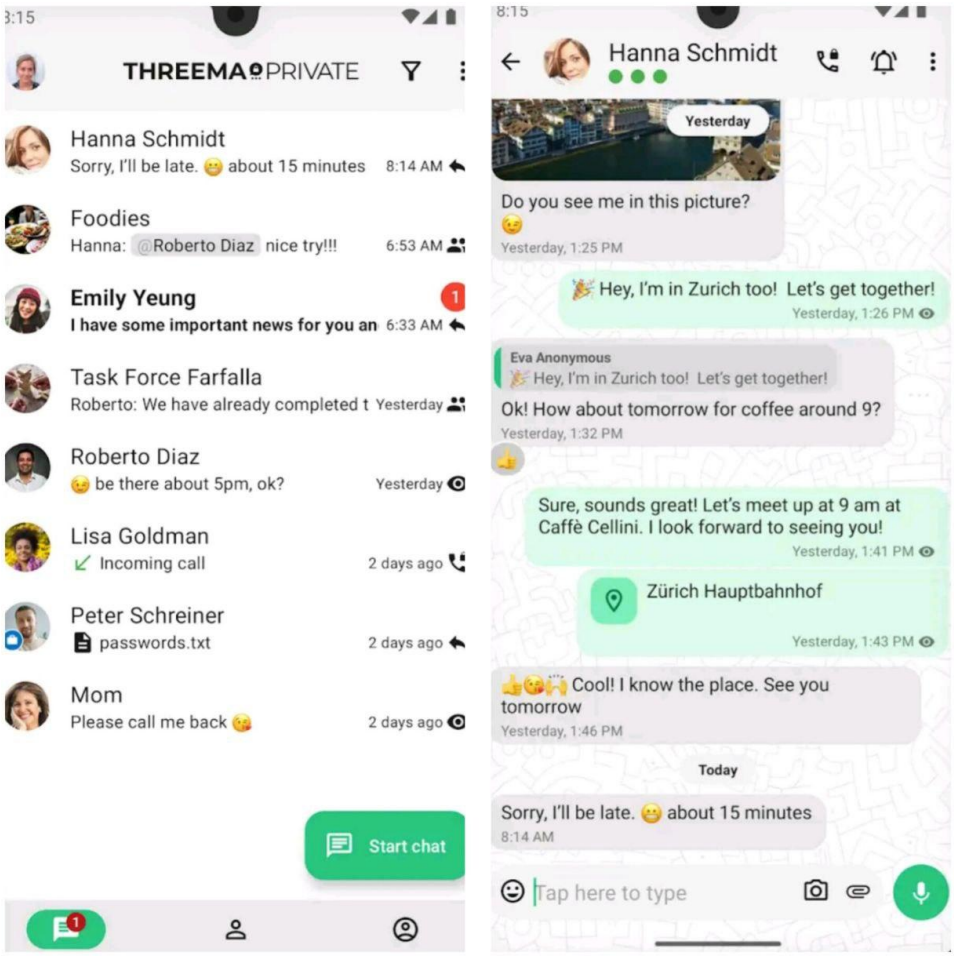


Рисунок 1.6 – «Інтерфейс застосунку «Threema»»

Перевагами месенджера «Threema» є те, що обліковий запис не прив’язується до номера телефону, замість того користувач отримує випадковий 8-значний Threema ID. Ба більше, сервери знаходяться у Швейцарії, що забезпечує високий рівень законодавчого захисту даних. І в решті решт важливим аспектом є те, що на серверах не зберігається нічого, крім зашифрованих пакетів, які видаляються одразу після доставлення.

На противагу цьому важливо зазначити, що застосунок є платним, що суттєво обмежує коло користувачі та до недавнього часу медіафайли шифрувалися інакше, ніж текстові повідомлення, хоча це виправляється в нових версіях протоколу Ibex.

Відповідно до аналізу конкурентів проведено комплексний аналіз сучасних месенджерів на предмет дотримання стандартів наскрізного шифрування.

Внаслідок чого було сформовано розуміння стандартів шифрування в сучасних застосунках, стандарти яких балансують між бізнес вимогами та безпекою. До такого ж висновку дійшли автори дослідження «Review of End-to-End Encryption for Social Media». Вони зазначають, що попри високу стійкість протоколу Signal, значна кількість комерційних месенджерів жертвує приватністю заради зручності. Це підтверджує необхідність розробки спеціалізованого ПЗ, де пріоритетом є повна ізоляція ключів на боці клієнта [16].

Враховуючи виявлені недоліки прямих конкурентів та вимоги до конфіденційності, було прийнято рішення про розробку власної системи обміну повідомленнями. Ключові архітектурні принципи проєктованої системи:

- використання наскрізного шифрування;
- мінімізація ролі сервера;
- відмова від збереження метаданих;

Такий підхід дозволить усунути «довіру до сервера» як слабку ланку в ланцюгу безпеки та забезпечити вищий рівень захисту в порівнянні з розглянутими аналогами.

1.3 Визначення функціональних та нефункціональних вимог до програмного забезпечення

На основі проведеного аналізу предметної області та наявних рішень було сформульовано вимоги до розроблюваної системи захищеного обміну повідомленнями. Головна мета системи – забезпечення конфіденційності листування шляхом наскрізного шифрування без збереження метаданих на сервері.

Процес проєктування будь-якої складної програмної системи починається з детального аналізу та формалізації вимог. Це дозволяє чітко окреслити межі проєкту, визначити очікувану поведінку системи та встановити критерії якості, яким має відповідати кінцевий продукт.

У межах даної розробки вимоги поділяються на дві основні категорії:

– функціональні вимоги, які описують конкретні сервіси та можливості, що надаються користувачеві;

– нефункціональні вимоги, які визначають атрибути якості, обмеження середовища та технічні стандарти;

Отже, для розроблюваної системи необхідно реалізувати такі функціональні вимоги:

– при першому запуску додаток має генерувати пару ключів за допомогою еліптичної криптографії;

– можливість встановлення PIN-коду для входу в застосунок;

– пошук та додавання співрозмовників за їхнім унікальним ID;

– можливість ігнорувати повідомлення від конкретних ідентифікаторів;

– шифрування кожного текстового повідомлення на пристрої відправника перед передачею на транспортний вузол;

– система повинна упаковувати зашифроване повідомлення у пакет, що містить лише ID отримувача;

– відображення статусів, реалізоване з мінімальним логуванням на сервері;

– функція повного очищення локальної бази даних та кешу медіафайлів одним натисканням;

– система не повинна завантажувати історію повідомлень у незашифровані хмарні сервіси;

– користувачу має бути надана можливість налаштовувати свій профіль користувача;

– реєстрація користувача за допомогою електронної пошти;

– верифікація електронної пошти методом надсилання листа верифікації;

– можливість встановлення фотографії користувача для свого облікового запису;

– можливість редагування та видалення повідомлень;

– перегляд чату з надісланими повідомленнями;

Наступним етапом є опис нефункціональних вимог до застосунку, вони включають в себе:

– використання сучасних стандартів шифрування;

– сервер не повинен зберігати жодної інформації в незашифрованому вигляді, включаючи логи зв'язків та IP-адреси користувачів після завершення сесії;

– час між натисканням кнопки «Надіслати» та надходженням повідомлення на сервер не повинен перевищувати 1 секунду при стабільному інтернет-з'єднанні;

– робота додатка у фоновому режимі повинна споживати не більше 3-5% заряду акумулятора на добу;

– транспортний сервер повинен забезпечувати коефіцієнт доступності не менше 99.9%;

– у разі розриву з'єднання система повинна автоматично відновлювати зв'язок та завершувати передачу черги повідомлень без втручання користувача;

– локальне сховище на пристрої повинно бути захищеним від пошкодження даних у разі раптового вимкнення пристрою;

– застосунок повинен коректно працювати на актуальних версіях ОС;

– архітектура сервера повинна підтримувати горизонтальне масштабування для обслуговування одночасно до 10,000+ активних користувачів без деградації продуктивності;

– користувач повинен мати можливість розпочати спілкування не більше ніж за 3-4 кроки;

– інтерфейс повинен підтримувати українську та англійську мови як базові;

Для візуалізації функціональних вимог та визначення меж системи було побудовано діаграму варіантів використання мовою UML (Рисунок 1.7). Вона відображає взаємодію основного актора «Користувач» з ключовими процесами додатка. На діаграмі також показано залежності між прецедентами, зокрема обов'язкові етапи та опціональні розширення функціонала.

До того ж було створено діаграму послідовностей для варіанту використання «Надсилання повідомлення» (Рисунок 1.8).

Система матиме такі варіанти використання:

- реєстрація;
  - генерація ключів;
- надсилання повідомлення;
  - шифрування даних;
  - прикріплення медіафайлу;
  - редагування повідомлення;
  - видалення повідомлення;
- отримання повідомлення;
  - дешифрування даних;
- додавання контакту;
  - пошук користувача за ID;

- видалення контакту;

– створення приватного чату;

- Видалення чату;

– зміна налаштувань профілю;

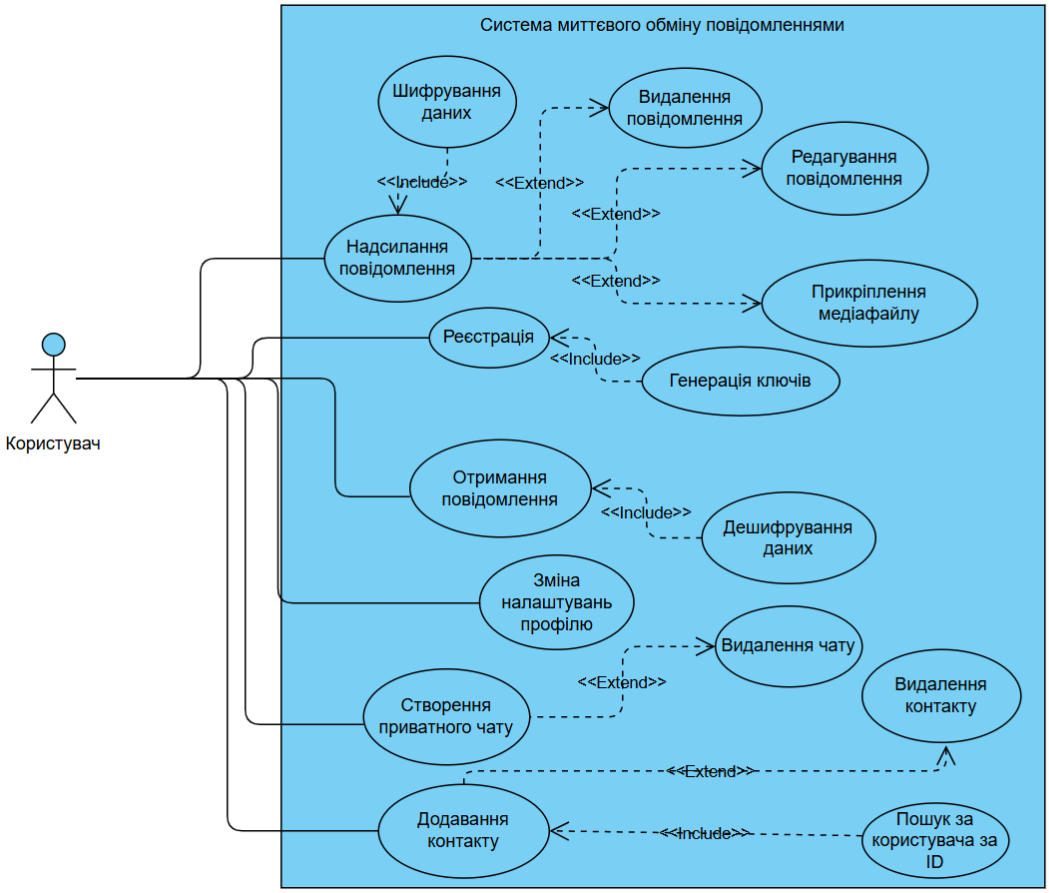


Рисунок 1.7 – «Діаграма Варіантів Використання»

Підсумовуючи викладене, у даному підрозділі було проведено детальний аналіз та формалізацію вимог до системи захищеного обміну повідомленнями.

По-перше, чітко розмежовано функціональні вимоги, які визначають набір можливостей користувача, та нефункціональні вимоги, що встановлюють стандарти якості системи, зокрема високий рівень криптографічного захисту, швидкодію та надійність.

По-друге, за допомогою уніфікованої мови моделювання розроблено діаграму варіантів використання. Це дозволило візуалізувати межі системи, ідентифікувати ключових акторів та визначити сценарії їхньої взаємодії з програмним продуктом.

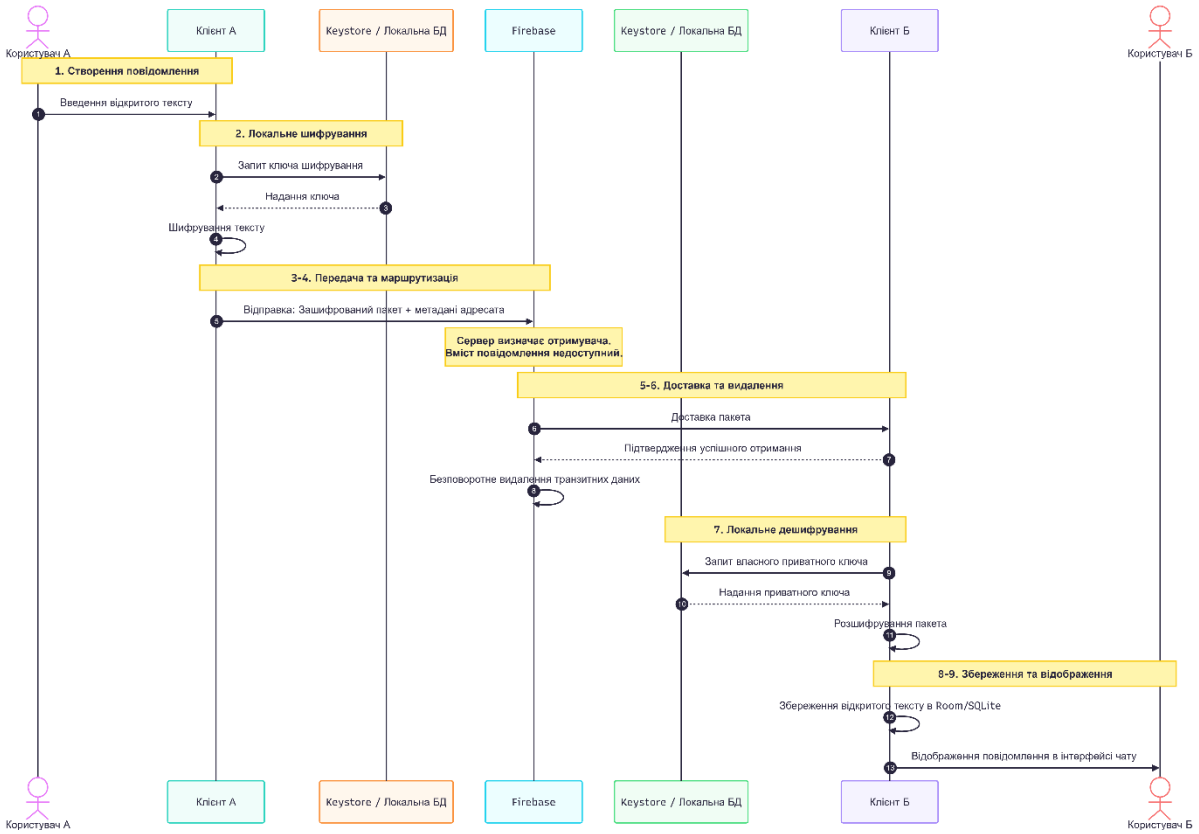


Рисунок 1.8 – Діаграма послідовності для варіанту використання «Надсилання повідомлення»

1.4 Висновки. Постановка задачі

У цьому розділі було розглянуто предметну область створення мобільних систем обміну миттєвими повідомленнями в загальному і зокрема систем із наскрізним шифруванням. Було проаналізовано основні процеси і етапи криптографічного захисту даних, яку задачу вони вирішують і яка мета в створенні ще одного месенджера з архітектурою «нульової довіри». Для

визначення вимог до застосунку було проаналізовано особливості захисту інформації в мобільних мережах, розглянуто популярні підходи до управління ключами та обрано механізми мінімізації метаданих для мого проєкту. Також, для повного розуміння, було досліджено статистичні дані, які описують обсяги використання месенджерів та ризики кібератак і визначають, що саме цінять користувачі захищених програмних рішень.

З метою виокремити безпечні механіки, які пропонують наявні застосунки, було розглянуто найпопулярніших представників на ринку мобільних комунікацій.

Отримавши вичерпну функціональну і технічну специфікацію під час роботи над розділом, можна поставити задачі, які мають бути реалізовуватись надалі під час виконання кваліфікаційної роботи.

Перелік поставлених задач:

– провести аналіз наявних архітектурних підходів і патернів проєктування, які зазвичай використовуються, та на основі цього визначитись з тими, які будуть використані для мого застосунку;

– описати декомпозицію проєкту, зв’язки між варіантами декомпозиції та інтерфейси модулів;

– розробити макетне представлення інтерфейсу користувача;

– провести аналіз та вибрати технології для реалізації спроєктованої архітектури, які будуть найефективнішим варіантом для реалізації;

– розробити основні модулі застосунку;

– реалізувати логіку генерації ключів та наскрізного шифрування повідомлень;

– створити та запрограмувати інтерфейс користувача;

– визначитись з методами тестування та реалізувати їх;

– провести тестування застосунку та проаналізувати результати;

– зробити висновки на основі результатів кваліфікаційної роботи.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Вибір типу архітектури та зразків проєктування

Наразі існує безліч різноманітних архітектурних шаблонів, які активно використовуються в індустрії та впроваджуються в стандарти розробки, але важливо правильно обрати архітектуру, яка буде ефективно виконувати завдання, поставлені перед застосунком, який проєктується. Тому до розгляду буде взято кілька найбільш поширених, відомих і надійних архітектур для обґрунтованого вибору однієї з них.

Першим до розгляду було обрано MVC. Це класичний архітектурний шаблон, який став одним із перших стандартів у розробці програмного забезпечення. Його основна мета – розділити відповідальність між даними та інтерфейсом, щоб зробити код зрозумілішим [17].

Він складається з таких елементів:

– Model, що керує даними застосунку, бізнес-правилами та логікою доступу до інформації;

– View, що відповідає за візуальне представлення даних користувачеві;

– Controller, що виступає посередником. Він отримує вхідні дані, обробляє їх і вирішує, як мають змінитися «Модель» або «Відображення».

В Android-розробці це призводить до перевантаження класів Activity, що унеможливлює якісне модульне тестування та підтримку великих проєктів.

Другим для розгляду було взято архітектурний шаблон MVP. Архітектурний шаблон MVP з’явився як логічний етап еволюції після MVC з метою усунути проблему надмірної зв'язності компонентів та полегшити модульне тестування. Головна зміна полягала в повній ізоляції Відображення від Моделі, де роль координатора перейшла до Презентера [18].

Його складові елементи**:**

– Model, як і в інших підходах, відповідає за дані та бізнес-логіку;

– View, яким в Android є Activity, Fragment або Custom View. Воно є максимально пасивним і лише відображає дані, які передає Презентер через інтерфейс;

– Presenter, що містить логіку поведінки інтерфейсу. Він отримує дані з Моделі, опрацьовує їх і безпосередньо вказує Відображенню, що саме потрібно показати.

Недолік цього шаблону, що він створює жорстку залежність «один до одного» між View та Presenter, що веде до надмірної кількості інтерфейсів та ускладнює обробку життєвого циклу Android-компонентів [19].

Та останній архітектурний шаблон до розгляду – це MVVM. Архітектурний шаблон MVVM є сучасним стандартом розробки для платформи Android, рекомендованим компанією «Google».

Він складається з таких елементів: модель, відображення, модель відображення.

Модель або Model – це фундаментальний шар абстракції, який відповідає за бізнес-логіку застосунку, структуру даних та безпосередній доступ до них. Важливо зазначити, що Модель повністю ізольована від користувацького інтерфейсу і не знає про його існування [20].

Відображення або View є графічним інтерфейсом користувача і відповідає за відображення візуальних елементів та захоплення дій користувача [21].

Модель відображення або ViewModel виконує роль інтелектуального посередника між графічним інтерфейсом та бізнес-логікою. Вона трансформує необроблені дані з моделі у зручний для відображення формат [22].

Для кращого розуміння архітектурної логіки на Рисунку 2.1 зображено взаємодію ключових компонентів патерна MVVM між собою. Представлена схема демонструє чіткий розподіл обов'язків: шар ViewModel взаємодіє з Model для запиту та модифікації бізнес-логіки й даних. Водночас View підписується на оновлення ViewModel за допомогою механізмів спостереження.

Рисунок 2.1 – Шаблон проєктування «MVVM»

Отже, для забезпечення надійності та узгодження зі стандартами індустрії для обраного застосунку було обрано архітектурний шаблон MVVM, який у повному вигляді має назву – Model-View-ViewModel.

Згідно з дослідженнями, архітектура MVVM демонструє кращі показники використання центрального процесора та часу виконання порівняно з іншими поширеними шаблонами, такими як MVP або MVC. Хоча використання додаткових бібліотек в MVVM може дещо збільшити споживання оперативної пам'яті, це компенсується значним підвищенням швидкості реакції системи [23].

Також для забезпечення гнучкості та відповідності принципам чистої архітектури в розробленій системі було впроваджено низку шаблонів проєктування, кожен з яких вирішує конкретне завдання з управління даними, життєвим циклом об'єктів або станом інтерфейсу.

Базовим архітектурним рішенням для роботи з інформацією став шаблон «Репозиторій». Він виступає центральним вузлом управління даними, ізолюючи бізнес-логіку від особливостей реалізації джерел даних.

Реактивність інтерфейсу забезпечується завдяки використанню шаблону «Спостерігач». Це фундаментальний концепт для сучасних Android-систем, який дозволяє об'єктам реагувати на зміни стану інших об'єктів.

Контроль за унікальністю критичних ресурсів покладено на патерн «Одинак». Цей шаблон гарантує, що певний клас матиме лише один примірник у межах всього життєвого циклу застосунку.

Специфічні завдання зі створення екземплярів класів вирішує шаблон «Фабрика». Він використовується для створення об'єктів, коли процес їх ініціалізації є складним або залежить від зовнішніх умов.

Завершує цей комплекс архітектурних рішень патерн «Стан», який дозволяє об'єкту змінювати свою поведінку залежно від його внутрішнього стану. Він зазвичай використовується для управління процесами авторизації та навігації.

Отже, підсумовуючи результати дослідження, проведеного у даному підрозділі, можна стверджувати, що вибір архітектурного шаблону MVVM є найбільш оптимальним та науково обґрунтованим рішенням для розробки безпечної мобільної системи обміну повідомленнями.

Також впровадження комплексу патернів проєктування: «Репозиторій», «Впровадження залежностей», «Спостерігач», «Одинак», «Фабричний метод» та «Стан», дозволило вирішити ключові інженерні завдання. Завдяки цим зразкам система отримала слабку зв'язність компонентів, високий рівень придатності до тестування, здатність ефективно працювати в офлайн-режимі та миттєво реагувати на зміни даних через реактивний інтерфейс без блокування головного потоку.

2.2 Опис декомпозиції

Декомпозиція програмної системи є фундаментальним етапом проєктування, який полягає у розбитті складної монолітної структури на менші, слабкозв’язані та логічно завершені частини.

Проєктована система базується на багаторівневому підході до декомпозиції, який охоплює архітектурний, функціональний, компонентний та інформаційний аспекти, що дозволяє всебічно розглянути її структуру.

Спочатку потрібно провести загальну декомпозицію системи, яка передбачає горизонтальний поділ програмного забезпечення на незалежні шари за принципами шаблону MVVM.

Це дозволяє ізолювати бізнес-логіку від деталей реалізації та забезпечити слабку зв'язність компонентів.

Система структурована за такими рівнями:

– шар відображення або Presentation Layer. Цей рівень охоплює пакети екранів та навігацію. Головна роль зазначеного розділу полягає у декларативній обробці інтерфейсу;

– шар управління станом, відомий як ViewModel Layer. Дана складова архітектури зосереджена навколо ViewModel. Він функціонує як інтелектуальний посередник, що забезпечує трансформацію даних із репозиторіїв у реактивні стани;

– шар предметної області чи Domain Layer. Цей шар є ядром системи, він містить у собі виключно чисті бізнес-моделі та набір абстрактних інтерфейсів, яскравим прикладом яких є мережеві інтерфейси;

– шар даних або Data Layer. Зазначений рівень бере на себе відповідальність за надійне збереження та синхронізацію потоків інформації.

Наступною декомпозицією є модульна декомпозиція системи, яка передбачає вертикальне розділення програмного забезпечення на незалежні функціональні блоки та інфраструктурні модулі. Такий підхід дозволяє ізолювати окремі функції застосунку, забезпечуючи їх автономність та спрощуючи процес паралельної розробки та тестування. У месенджері реалізовано наступні ключові модулі:

– модуль автентифікації та управління профілем. Основне завдання модуля полягає в управлінні життєвим циклом сесії користувача;

– модуль чатів та повідомлень. Зазначений сегмент є функціональним ядром системи, що базується на екранах списку діалогів та вікні безпосереднього спілкування;

– модуль управління контактами. Ця складова відповідає за реалізацію функцій пошуку інших учасників системи за їхніми унікальними ідентифікаторами та формування персональної книги контактів;

– інфраструктурний модуль впровадження залежностей. Він базується на використанні фреймворку Dagger 2. Головна роль шаблона впровадження залежностей полягає в автоматизації управління життєвим циклом об'єктів;

– модуль криптографії та безпеки. Він базується на сучасній криптографічній бібліотеці Google Tink. Він повністю ізольований від мережевого шару та інтерфейсу, надаючи виключно методи для реалізації наскрізного шифрування;

Отже, застосований комплексний підхід до декомпозиції дозволив трансформувати складну систему мобільного месенджера у набір слабкозв’язаних та функціонально завершених модулів. Завдяки горизонтальному поділу на архітектурні шари та вертикальному розбиттю за функціональними ознаками, було досягнуто високого рівня ізоляції бізнес-логіки від особливостей програмної платформи та мережевої інфраструктури.

2.3 Опис залежностей

Опис залежностей є важливим етапом проєктування програмного забезпечення. Він дозволяє сформувати чітке розуміння того, як саме взаємодіють між собою різноманітні компоненти системи.

Архітектура розробленого програмного застосунку повністю базується на шаблоні проєктування MVVM. Це забезпечує суворий розподіл відповідальності між компонентами та односпрямований потік даних.

Шар користувацького інтерфейсу у розробленому застосунку реалізований за допомогою декларативного фреймворку Jetpack Compose. Цей шар є абсолютно пасивним і не містить жодних елементів прийняття рішень чи прямого доступу до

даних. Відповідно, екрани чату, сторінки авторизації чи списки контактів мають жорстку залежність виключно від відповідних класів управління станом.

Своєю чергою класи управління станом виступають інтелектуальним мостом між інтерфейсом та даними. Вони абстраговані від безпосередньої роботи з джерелами інформації та залежать лише від абстрактних інтерфейсів предметної області. Завдяки такій структурі досягається низька зв'язність коду.

Ядром системи виступає шар предметної області. Цей рівень не має жодних зовнішніх залежностей від бібліотек операційної системи Android або сторонніх фреймворків. Він містить у собі чисті бізнес-моделі та визначає контракти у вигляді інтерфейсів. Саме ці контракти диктують правила того, як інші модулі повинні зберігати чи передавати інформацію.

Шар даних у застосунку структурно розділений на локальне та віддалене сховища, які реалізують інтерфейси предметної області. За комунікацію з хмарними сервісами відповідає окремий репозиторій, який має залежності від бібліотек Firebase Auth та Firestore.

Окремої уваги заслуговує інтеграція криптографічного модуля, який є основою безпеки месенджера. Цей модуль має залежності від вбудованого сховища ключів Android та алгоритмів бібліотеки Google Tink. Процес перетворення даних ініціюється виключно на рівні управління станом.

Для автоматизації процесу створення об'єктів та управління їхнім життєвим циклом у системі застосовано шаблон інверсії управління.

Завдяки цьому впровадженню система була поділена на ізольовані модулі постачання залежностей. Ці модулі формують загальний граф об'єктів та визначають правила їх створення. В архітектурі месенджера можна виділити наступні ключові напрямки постачання залежностей:

– системний модуль відповідає за надання глобальних об'єктів, які існують в єдиному екземплярі протягом усього часу роботи програми;

– мережевий модуль керує створенням клієнтів Firebase та надає їхні екземпляри тим репозиторіям;

– криптографічний модуль забезпечує ініціалізацію інструментів Google Tink та постачає готові до використання класи шифрування безпосередньо у відповідні рівні управління станом;

– модуль репозиторіїв зв'язує абстрактні інтерфейси предметної області з їхніми конкретними реалізаціями на рівні даних.

Отже, детальний розгляд та аналіз залежностей забезпечили глибше розуміння інфраструктури проєктованої системи. Як зазначається у дослідженні «Choosing a Global Architecture for Mobile Applications», інтеграція патернів управління залежностями з глобальною архітектурою дозволяє вирішити ключові виклики мобільної розробки. Вона забезпечує дві критично важливі властивості: повну незалежність бізнес-логіки від зовнішніх фреймворків та високу придатність системи до модульного тестування [24].

Завдяки такому підходу, розроблений застосунок є стійким до змін інфраструктури та готовим до подальшого масштабування без ризику порушення цілісності конфіденційних даних користувачів.

2.4 Опис інтерфейсу користувача

Успіх застосунку безпосередньо залежить від детально опрацьованого інтерфейсу, оскільки саме він формує перше враження та визначає ергономіку взаємодії з продуктом. У цій роботі акцент зроблено на балансі між технічною складністю криптографічних протоколів та інтуїтивністю їх візуалізації. Процес розробки розпочався з проєктування логічної структури екранів та створення прототипів для подальшої програмної реалізації на Android.

Основою візуальної мови застосунку стала специфікація Material Design 3, яка забезпечує уніфікований вигляд компонентів, сучасну динамічну систему кольорів та адаптивність інтерфейсу до різних конфігурацій мобільних пристроїв. Це дозволяє гарантувати не лише візуальну цілісність усіх екранів, а й високу

ергономіку інтерфейсу, незалежно від діагоналі екрана чи щільності пікселів цільового смартфона.

Аналізуючи функціональні вимоги до застосунку, було визначено оптимальну кількість функціональних груп екранів, які охоплюють повний життєвий цикл взаємодії користувача з месенджером:

Перша група охоплює екрани автентифікації та початкового налаштування безпеки, які є вхідними точками в систему та забезпечують цілісність користувацької сесії. Сюди входить екран входу з перевіркою облікових даних та екран реєстрації, який ініціює створення записів у хмарному сховищі.

Друга група являє собою основний робочий простір застосунку та керується через централізовану систему нижньої навігації, винесену в окремий програмний модуль. Головним вузлом цієї групи є екран списку чатів, де реалізовано механізм динамічного відстеження оновлень у локальній базі даних Room.

Третя група екранів присвячена безпосередньо процесу конфіденційного спілкування та є найбільш технологічно насиченою частиною інтерфейсу.

Розуміючи структуру та визначившись із переліком необхідних елементів, було створено детальні макети ключових екранів програми, які зображені на рисунку А.1 у додатку А.

Процес макетування дозволив проаналізувати ергономіку розміщення компонентів та адаптувати їх під потреби користувачів, які цінують приватність та швидкість роботи.

2.5 Детальне проєктування модулів

У цьому розділі будуть детально спроєктовані модулі для глибшого розуміння системи. Особливу увагу приділено моделюванню інформаційних потоків, алгоритмам обробки даних та визначенню життєвого циклу ключових об'єктів системи. Результатом цього етапу проєктування є формування статичної

структури програмного коду, яка враховує специфіку платформи Android та слугує безпосередньою основою для подальшої програмної реалізації застосунку.

Архітектура месенджера структурована за багаторівневим принципом і складається з трьох основних шарів, які були згадані у розділі 2.2.

Такий поділ забезпечує реалізацію принципу розділення відповідальності, де кожен рівень виконує суворо визначені функції. Рівень представлення відповідає за взаємодію з користувачем, рівень предметної області містить ядро бізнес-логіки та криптографічні правила, а рівень даних реалізує шаблон «єдиного джерела істини», поєднуючи локальну базу даних Room та хмарні сервіси Firebase.

Взаємодія між цими компонентами дозволяє системі стабільно функціонувати в умовах нестабільного інтернет-з'єднання, гарантуючи при цьому цілісність та конфіденційність інформації.

Для наочного представлення взаємозв'язків між програмними модулями нижче наведено діаграму компонентів на рисунку 2.2.

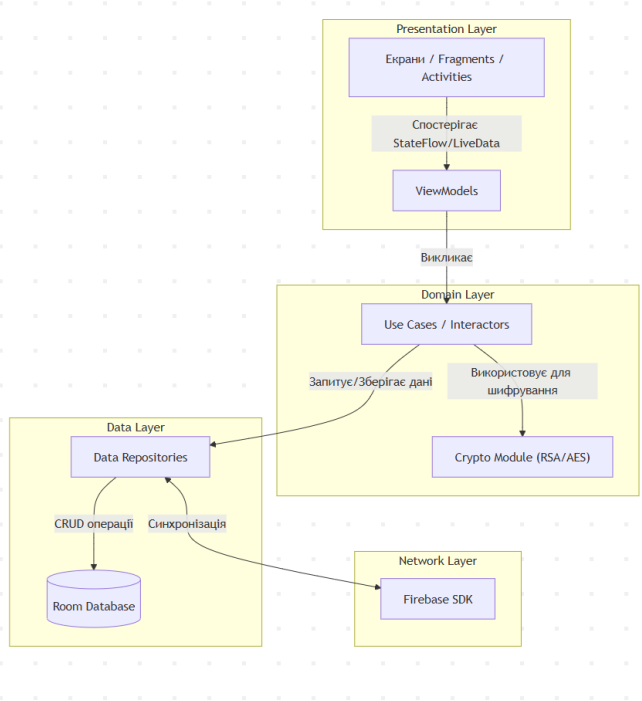


Рисунок 2.2 – Діаграма компонентів

Як видно з наведеної діаграми, спрямовані стрілки позначають вектор залежностей між архітектурними шарами системи.

Наприклад, рівень представлення безпосередньо залежить від рівня бізнес-логіки, проте зворотна залежність суворо відсутня, що гарантує ізольованість ядра застосунку та криптографічних алгоритмів від будь-яких змін у користувацькому інтерфейсі.

Для моделювання процесів перетворення та переміщення інформації використано діаграми потоків даних. Оскільки базовою вимогою є конфіденційність, ключовим аспектом обробки є наскрізне шифрування, яке забезпечує надійний захист даних від стороннього доступу.

Згідно з останніми дослідженнями у сфері кібербезпеки мобільних систем, сучасна імплементація E2EE з використанням комбінації криптографічних алгоритмів гарантує так звану «нульову довіру». Це означає, що навіть провайдери хмарних послуг чи власники платформи технічно не здатні отримати доступ до відкритого тексту, оскільки дешифрування відбувається суворо ізольовано на пристрої кінцевого користувача [25]. Саме тому потоки даних спроєктовано так, щоб відкритий текст ніколи не потрапляв до мережевого шару.

Для розуміння цих процесів загальний потік даних розділено на два основні напрямки: процес відправлення повідомлення та процес його отримання.

На першому етапі система звертається до локальної бази даних для отримання публічного ключа отримувача. Далі текст передається до криптографічного модуля, де за допомогою алгоритму асиметричного шифрування перетворюється на шифротекст для подальшого надсилання адресату.

Важливою архітектурною особливістю є те, що зашифроване повідомлення спочатку зберігається у локальній базі даних зі статусом «Очікує надсилання», і лише після цього передається до мережевого модуля для синхронізації з хмарною базою даних. Такий підхід гарантує збереження даних тільки на локальних

пристроях і унеможливлює крадіжку конфіденційних даних з чату методом втручання у серверну частину.

На рисунку 2.3 наведено візуалізацію потоку даних при відправленні повідомлення адресату.

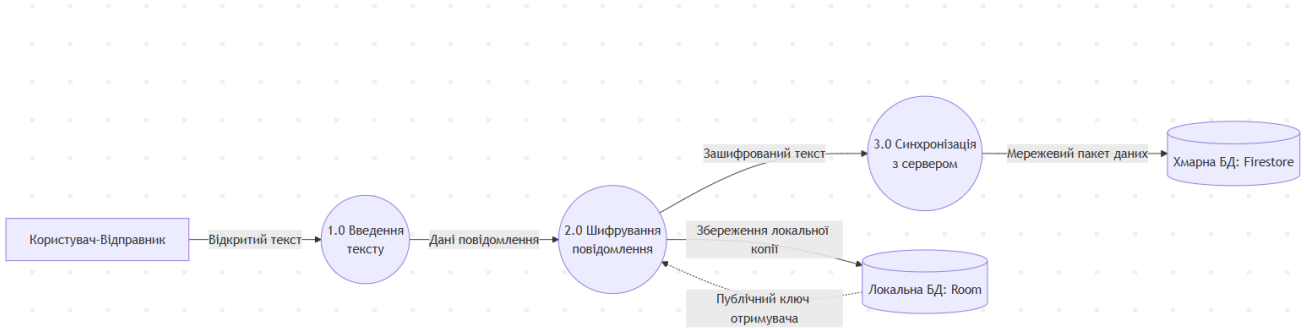


Рисунок 2.3 – Діаграма потоку даних при відправленні повідомлення

Процес отримання ініціюється зовнішньою подією – надходженням пакета даних від Firebase. Отриманий пакет містить зашифрований текст та метадані.

Для прочитання повідомлення система вилучає зашифрований текст і звертається до захищеного локального сховища за власним приватним ключем користувача. Після успішного дешифрування у криптографічному модулі, відновлений відкритий текст зберігається у локальній базі даних Room.

На рисунку 2.4 наведено візуалізацію потоку даних при надходженні повідомлення на стороні отримувача.

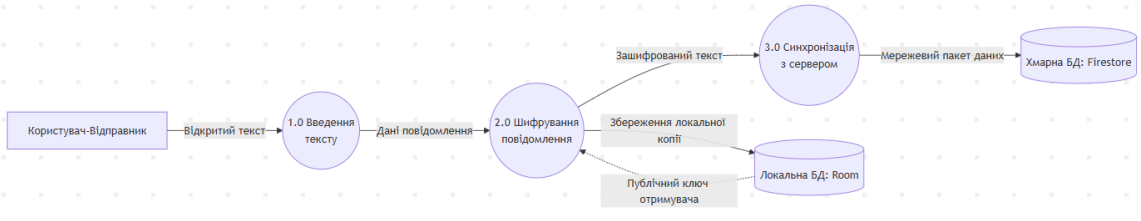


Рисунок 2.4 – Діаграма потоку даних при отриманні повідомлення

Для моделювання динаміки поведінки системи та візуалізації процесів обміну повідомленнями між об'єктами у часі було використано діаграми послідовності мови UML.

Даний тип діаграм дозволяє найточніше описати хронологію викликів методів та передачу даних між клієнтським застосунком, локальною базою даних та хмарним сервером. У межах цього підрозділу деталізовано два критично важливі процеси: ініціалізацію криптографічного захисту та протокол наскрізного шифрування при обміні повідомленнями.

Ініціалізація захисту є першим і найважливішим етапом у життєвому циклі сесії користувача, який відбувається безпосередньо під час створення облікового запису. Після успішної автентифікації клієнта через сервіс Firebase Auth, рівень бізнес-логіки застосунку ініціює локальну генерацію асиметричної пари криптографічних ключів.

Згідно з вимогами безпеки, приватний ключ зберігається виключно у локальному сховищі пристрою і ніколи не передається через мережу. Хронологію цього процесу наведено на рисунку 2.5.

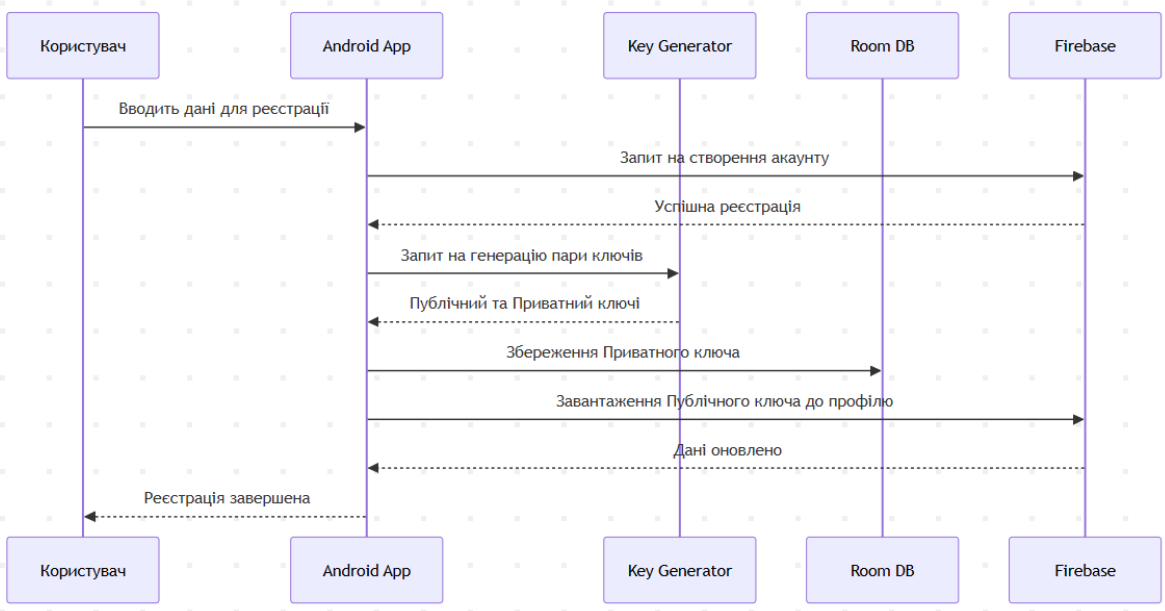


Рисунок 2.5 – Діаграма послідовності «Реєстрація користувача та ініціалізація ключів»

Основний бізнес-процес месенджера реалізує логіку наскрізного шифрування. Процес відправлення починається з того, що клієнт-відправник запитує з сервера публічний ключ отримувача. Використовуючи цей ключ, система локально шифрує текст повідомлення. До мережі та на сервери Firebase потрапляє виключно нечитабельний шифротекст.

На стороні отримувача процес ініціюється автоматично при надходженні нового пакета даних. Застосунок отримувача вилучає шифротекст, звертається до локальної бази даних Android Keystore за власним приватним ключем та виконує дешифрування. Зашифрований текст зберігається локально і передається на рівень інтерфейсу для відображення. Даний сценарій деталізовано на рисунку 2.6.

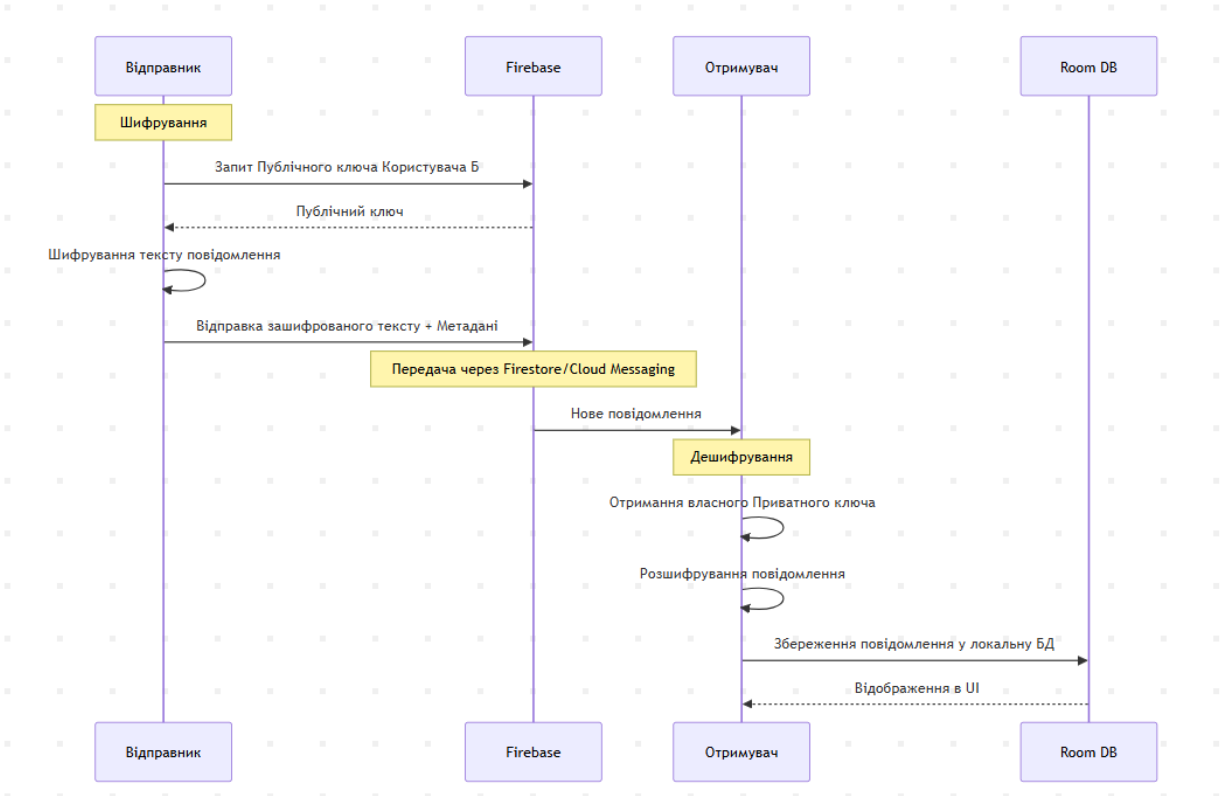


Рисунок 2.6 – Діаграма послідовності «Надсилання та отримання зашифрованого повідомлення»

Аналіз наведених динамічних моделей підтверджує, що архітектура застосунку унеможливлює перехоплення відкритого тексту на етапі передачі даних, оскільки сервер виконує виключно транзитну функцію і не має доступу до приватних ключів користувачі.

Наступним етапом є опис внутрішньої логіки прийняття рішень та алгоритмів обробки даних у системі використано діаграми діяльності. Одним із найскладніших та найважливіших алгоритмів у розробленому месенджері є процес відправлення зашифрованого повідомлення. Цей процес вимагає врахування багатьох факторів: стану мережі, наявності криптографічних ключів та результатів взаємодії з хмарним сервером.

Алгоритм відправлення побудований за принципом «Offline-first», що забезпечує безперебійну роботу інтерфейсу користувача навіть за умови відсутності інтернет-з'єднання.

Покроковий опис алгоритму:

1. процес розпочинається після того, як користувач ввів текст і натиснув кнопку «Надіслати»;
2. система опитує мережеві служби Android щодо наявності активного підключення до інтернету;

– якщо підключення відсутнє, то зашифрований текст повідомлення негайно зберігається у локальну базу даних зі статусом «Очікує». На цьому поточна активність завершується, а фоновий планувальник бере на себе завдання відправити повідомлення при появі мережі;

– якщо підключення наявне, то алгоритм переходить до підготовки криптографічних даних;

3. для шифрування необхідний публічний ключ отримувача. Система перевіряє локальний кеш на його наявність;

– якщо ключа немає, то виконується асинхронний запит до Firebase Firestore для його завантаження;

– якщо ключ знайдено, то система переходить безпосередньо до шифрування, мінімізуючи мережеві затримки;

4. відкритий текст шифрується, після чого зашифрований пакет даних зберігається у локальну БД для відображення в UI;

5. зашифрований пакет відправляється на сервери Firebase;

– у разі успішної відповіді статус повідомлення в локальній базі даних оновлюється на «Надіслано»;

– у разі помилки статус оновлюється на «Помилка», що дозволяє користувачу здійснити повторну спробу пізніше.

В актуальних оглядах архітектур, використання NoSQL-рішень на кшталт Cloud Firestore дозволяє ефективно реалізувати механізм глибокої офлайн-синхронізації. Завдяки цьому застосунок здатний оптимістично оновлювати стан інтерфейсу користувача миттєво після натискання кнопки "Надіслати", зберігаючи дані локально [26].

На рисунку А.2 у додатку А наведено діаграму діяльності, яка візуалізує описаний алгоритм.

Крім того, згідно із сучасними галузевими стандартами розробки під ОС Android, реалізація Clean Architecture вимагає глибокої інтеграції з Kotlin Coroutines та реактивними потоками. Тому всі сценарії обробки вводу реалізуються як асинхронні функції. Це дозволяє виконувати важкі криптографічні та мережеві операції у фонових потоках, не блокуючи головний потік інтерфейсу [27].

Також, в контексті обміну повідомленнями об’єкт повідомлення має складний життєвий цикл, що обумовлено використанням наскрізного шифрування та необхідністю синхронізації між пристроями. Кожне повідомлення проходить шлях від чернетки до підтвердження прочитання отримувачем.

Критичним етапом є перехід від стану «Draft» до «SavedLocally». Саме на цьому етапі відбувається шифрування контенту. Стан «Pending» у локальній базі даних Room вказує на те, що повідомлення вже захищене і готове до передачі, що

дозволяє системі автоматично надіслати його при відновленні зв'язку. Кінцевим станом у циклі відправника є «Read», який підтверджує, що отримувач успішно дешифрував та відкрив повідомлення. На рисунку А.4 в додатку А наведено діаграму станів повідомлення.

Аналіз станів дозволяє уникнути невизначеності в роботі застосунку. Зокрема, чітке розмежування між станами «Sent» та «Delivered» дає користувачеві зворотний зв’язок про успішність проходження даних через транзитні сервери Firebase, а стан «Encrypting» гарантує, що жоден пакет не покине межі пристрою у відкритому вигляді.

Завершальним етапом детального проєктування системи є формування її статичної структури, яка слугуватиме безпосереднім «кресленням» для написання програмного коду. На основі розроблених раніше динамічних моделей, алгоритмів та архітектурних рішень було побудовано діаграму класів. Вона формалізує ключові сутності застосунку, їхні атрибути, методи та взаємозв'язки.

На рисунку 2.7 наведено UML-діаграму класів спроєктованого застосунку.

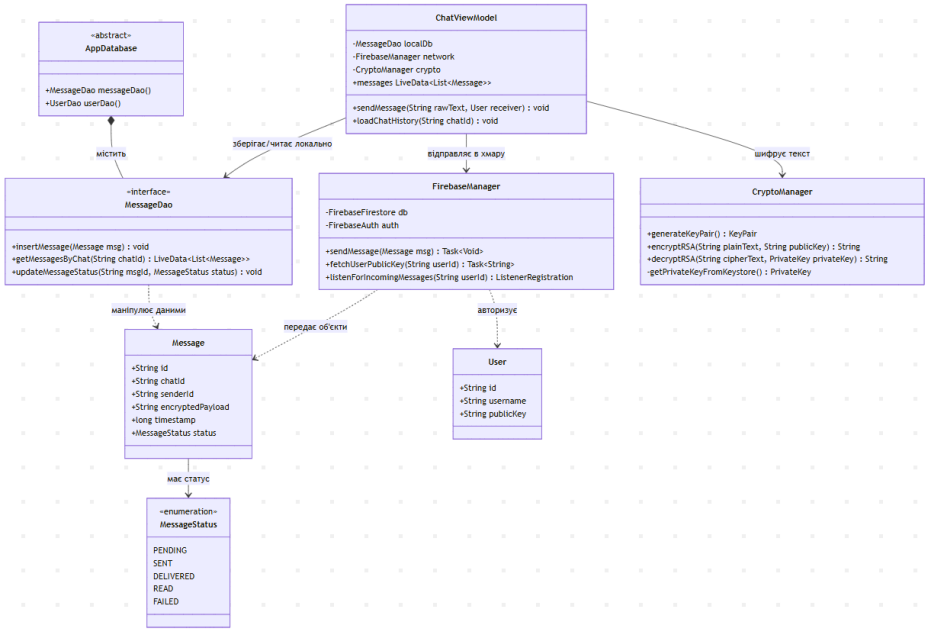


Рисунок 2.7 – Діаграма класів

У ході детального проєктування модулів було сформовано цілісну технічну базу для реалізації мобільного месенджера. Завдяки комплексному використанню статичних та динамічних моделей UML, а також діаграм потоків даних, вдалося детально формалізувати процеси наскрізного шифрування, визначити логіку обробки повідомлень у різних мережевих станах та структурувати взаємодію між компонентами системи.

2.6 Аналіз та вибір технологій і методів реалізації застосунку

Вибір технологічного стека для розробки сучасної системи обміну повідомленнями є багатогранним процесом, що вимагає аналізу системних вимог, апаратних обмежень мобільних пристроїв та суворих стандартів безпеки.

Фундаментальним інженерним рішенням став вибір мови програмування Kotlin замість традиційної Java. Як зазначається в аналізі сучасних мобільних кодових баз, сьогоднішня архітектура Android-додатків вимагає обробки великої кількості комплексних станів та інтенсивної фонової роботи. Використання Kotlin значно краще відповідає цим реаліям, оскільки його синтаксис дозволяє суттєво зменшити кількість надлишкового програмного «шуму» навколо бізнес-логіки. Якщо в Java надійність програми часто досягається за рахунок складної структурної церемоніальності, то Kotlin забезпечує безпеку на рівні самого наміру розробника, роблячи код значно зрозумілішим та легшим у супроводі при масштабуванні проєкту [28].

Наступним стратегічним кроком став вибір Jetpack Compose як основного інструментарію для розробки користувацького інтерфейсу. Традиційна система побудови UI на основі XML-розмітки є імперативною та вимагає від розробника ручного управління станом кожного елемента, що часто призводить до помилок синхронізації. Як зазначається у детальному порівняльному аналізі підходів до створення Android-інтерфейсів, відмова від XML на користь Jetpack Compose дозволяє вирішити ключову проблему розриву між логікою програми та її

візуальним представленням. Фреймворк усуває необхідність написання великих обсягів шаблонного коду і самостійно здійснює інтелектуальну рекомпозицію, перемальовуючи виключно ті UI-компоненти, стан яких змінився [29].

Окрему увагу було приділено вибору методів асинхронного програмування та управління потоками даних у реальному часі. Як зазначається в офіційній документації мови Kotlin, використання багатопотокових функцій пропонує принципово новий підхід до конкурентності порівняно з традиційними потоками операційної системи. Такі функції функціонують як легковагові потоки, які можна призупиняти та відновлювати без фактичного блокування основного робочого потоку застосунку [30].

Для реалізації шару збереження даних було обрано бібліотеку Room, яка є важливою частиною Android Jetpack. Як зазначається в огляді архітектурних рішень для персистентності даних, Room забезпечує вищий рівень абстракції порівняно з чистим SQLite, суттєво полегшуючи навантаження на розробників [31].

Вибір серверної інфраструктури базувався на необхідності забезпечення передачі даних у реальному часі та високої доступності сервісу. Платформа Firebase була обрана завдяки її масштабованості та цілісності екосистеми інструментів. В огляді еволюції цієї платформи, Firebase функціонує як комплексна система бекенду, що надає розробникам набір інтегрованих сервісів для вирішення інфраструктурних завдань без необхідності управління власними серверами [32].

Використання такої інфраструктури гарантує стабільну роботу месенджера при будь-яких навантаженнях, оскільки платформа автоматично масштабується відповідно до кількості активних користувачів.

Ключовим компонентом внутрішньої архітектури є механізм управління залежностями, реалізований за допомогою фреймворку Dagger 2. Зазначається, що Dagger є повністю статичним фреймворком, що працює виключно на етапі компіляції [33]. Використання цього інструменту дозволяє автоматизувати

створення об'єктів та управління їхнім життєвим циклом через формування чітко структурованого графа залежностей.

Для забезпечення найвищого рівня захисту конфіденційних даних у системі було обрано криптографічну бібліотеку Google Tink. В офіційній документації розробників Google, ця бібліотека спроєктована з фокусом на стійкість до типових помилок використання криптографії, що робить її надійним інструментом для реалізації наскрізного шифрування у мобільних застосунках [34].

Це гарантує, що конфіденційні ключі ніколи не покидають межі безпечного сховища пристрою, а всі операції шифрування виконуються на рівні апаратних засобів захисту. Використання глобальних архітектурних шаблонів у поєднанні з перевіреними інструментами є фундаментальною вимогою для створення безпечних застосунків. Суворе розділення зон відповідальності та незалежність від зовнішніх фреймворків значно підвищують рівень придатності коду до тестування та довгострокового супроводу.

Для роботи з мультимедійним контентом та асинхронного завантаження графічних ресурсів було обрано спеціалізовану бібліотеку Coil. В оглядах сучасних інструментів для екосистеми Jetpack Compose, Coil є першою бібліотекою, яка повністю побудована на базі Kotlin Coroutines, що робить її нативним рішенням для сучасного стека розробки [35].

Отже, проведений детальний аналіз інструментальних засобів, сучасних фреймворків та криптографічних бібліотек переконливо підтверджує, що обраний технологічний стек повністю відповідає поставленим архітектурним, функціональним та безпековим вимогам до сучасного месенджера. Кожен елемент програмної екосистеми – від засобів розробки серверної частини до компонентів побудови клієнтського інтерфейсу – був підібраний із урахуванням жорстких критеріїв кросплатформеної сумісності, високої швидкодії та мінімального споживання ресурсів.

Такий комплексний, системний та науково обґрунтований підхід дозволяє створити не просто базовий мобільний застосунок, а гнучку, масштабовану,

оптимізовану та стійку до зовнішніх вразливостей програмну систему. Інтеграція перевірених часом криптографічних примітивів та сучасних патернів проєктування мінімізує технологічні ризики, запобігає виникненню "вузьких місць" під час пікових навантажень і закладає надійний фундамент для майбутнього розширення функціоналу.

З огляду на це, теоретичний етап моделювання та аналізу можна вважати успішно завершеним, а розроблену архітектурну модель – повністю готовою до переходу на етап безпосередньої практичної реалізації, написання коду та розгортання системи.

2.7 Висновки

У межах завершеного етапу дослідження було виконано комплексне, системне, всебічне та глибоке інженерне проєктування архітектурного каркаса програмної системи, призначеної для організації високозахищеного обміну текстовою та мультимедійною інформацією у реальному часі. На основі ретельного передпроєктного аналізу суворих специфікацій сучасної індустрії кібербезпеки та криптографії було сформовано концептуальну та логічну модель мобільного застосунку, яка у повній мірі розв'язує класичну дилему проектування, забезпечуючи безкомпромісний рівень конфіденційності даних без шкоди для загальної продуктивності, швидкодії інтерфейсу та чутливості програмного

Проєктування підсистеми збереження даних спиралося на використання сучасного об'єктно-реляційного відображення на базі бібліотеки Room, схема якої була оптимізована на рівні індексів та запитів для швидкого виконання транзакцій в умовах обмеженого обсягу оперативної пам'яті смартфона. Інтеграція локального сховища з хмарною інфраструктурою Firebase дозволила побудувати реактивний, асинхронний та відмовостійкий механізм синхронізації та реплікації повідомлень. Застосування шаблона єдиного джерела істини, де локальна база даних виступає монопольним посередником між мережею та інтерфейсом,

гарантує безперебійний доступ користувача до історії чатів в офлайн-режимі та забезпечує миттєве відновлення консистентності даних після відновлення зв'язку з бекендом.

Компонентна модель Jetpack Compose забезпечує інтелектуальну рекомпозицію лише тих елементів екрана, стан яких фактично змінився, що мінімізує навантаження на графічний та центральний процесори пристрою під час обробки високоінтенсивних потоків вхідних даних, включаючи швидке прокручування списків чи обробку медіафайлів.

Принципове значення для успішної реалізації проєкту мав проведений аналіз та фінальний вибір компонентів технологічного стека, де використання мови програмування Kotlin забезпечило безпеку щодо нульових посилань та високу лаконічність коду завдяки механізму асинхронних потоків StateFlow для обробки подій у реальному часі. Впровадження фреймворку інжекції залежностей Dagger 2 дозволило автоматизувати керування життєвим циклом об'єктів у пам'яті, забезпечило гнучке налаштування модулів та усунуло проблему жорсткого кодування залежностей, що є критичним для безпеки системи. Для реалізації ядра криптографічного захисту було обрано високорівневу бібліотеку Google Tink.

У поєднанні з апаратними засобами захисту ключів на базі ізольованого середовища Android KeyStore, яке використовує захищені чіпи на рівні Trusted Execution Environment, проєктована архітектура гарантує, що закриті криптографічні ключі користувача ніколи не залишать меж його фізичного пристрою у відкритому вигляді.

Отримані архітектурні рішення, деталізовані схеми потоків даних, діаграми компонентів та розроблена логіка взаємодії шарів системи складають завершений теоретичний і практичний базис. Спроектована модель є повністю верифікованою, збалансованою та готовою до безпосереднього переходу на етап практичного кодування, розгортання серверної інфраструктури та компіляції фінального клієнтського дистрибутива мобільного месенджера.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Особливості програмної реалізації криптографічних алгоритмів

Центральною ланкою розробленого месенджера є система наскрізного шифрування, яка забезпечує конфіденційність та цілісність інформації на всьому шляху її передачі від відправника до отримувача. Програмна реалізація цієї підсистеми базується на використанні асиметричного алгоритму RSA, математична стійкість якого ґрунтується на обчислювальній складності задачі факторизації великих цілих чисел. Вибір даного алгоритму зумовлений потребою в ефективному управлінні ключами, де кожен учасник системи оперує парою криптографічних об'єктів, математично пов'язаних між собою через властивості простих чисел та модулярної арифметики.

Процес генерації ключів ініціюється під час створення профілю користувача в межах класу SignViewModel. На першому етапі алгоритм обирає два великих випадкових простих числа p та q, після чого обчислюється модуль системи n як їхній добуток. Як доводить дослідження особливостей програмної реалізації та криптоаналізу алгоритму RSA, загальна безпека цієї криптосистеми критично залежить від параметрів згенерованих експонент. Аналіз математичних вразливостей показує, що якщо довжина закритого приватного ключа становить приблизно чверть або менше від кількості бітів модуля n, алгоритм стає вразливим до специфічних атак, які дозволяють зловмисникам аналітично відновити секретні параметри без необхідності повного перебору [36]. Саме тому для забезпечення стійкості до сучасних методів криптоаналізу у розробленому застосунку було встановлено довжину ключа на рівні 2048 біт.

Наступним кроком є обчислення значення функції Ейлера за формулою $\varphi(n) = (p-1)(q-1)$. Відкрита експонента e обирається як ціле число, що задовольняє умові взаємної простоти з $\varphi(n)$, тобто їхній найбільший спільний дільник має дорівнювати одиниці. Закрита експонента d розраховується як мультиплікативне

обернене до e за модулем $\varphi(n)$, що фактично означає виконання рівності $(d \times e) \pmod{\varphi(n)} = 1$. Таким чином формується публічний ключ у складі пари $\{e, n\}$ та приватний ключ $\{d, n\}$.

Важливим інженерним рішенням у контексті безпеки є інтеграція процесу генерації з апаратним сховищем AndroidKeyStore. Це дозволяє створювати криптографічні об'єкти в захищеному середовищі Trusted Execution Environment, де приватний ключ d ізольований на рівні апаратного забезпечення. Після завершення генерації публічний ключ піддається трансформації у формат Base64 згідно зі стандартом X.509 для подальшої передачі у хмарну базу даних Firebase.

Безпосередній процес шифрування текстових повідомлень реалізовано в межах компонента MessagesViewModel. Коли ініціюється відправка даних, текстовий блок M перетворюється у числове представлення m, яке повинно бути меншим за модуль n. Процес шифрування виконується шляхом піднесення числа m до степеня e за модулем n за формулою $c = m^e \pmod{n}$. Отримана криптограма c кодується у формат Base64, що гарантує її цілісну.

Операція розшифрування відбувається виключно на стороні отримувача та вимагає звернення до захищеного приватного ключа d. Відновлення початкового повідомлення здійснюється шляхом піднесення отриманої криптограми c до степеня d за модулем n за формулою $m = c^d$. Завдяки теоремі Ейлера та правильно обраним параметрам ключової пари результатом цієї математичної операції є ідентичне числове значення m, яке згодом конвертується назад у текстовий рядок у кодуванні UTF-8.

З архітектурної точки зору уся криптографічна логіка повністю відокремлена від бізнес-шару застосунку та інкапсульована в класі Encryption. Використання інтерфейсу I_Encryption та фреймворку Dagger 2 дозволяє впроваджувати ці обчислювальні механізми у ViewModel, зберігаючи при цьому гнучкість системи. Такий підхід дозволяє легко масштабувати рішення, наприклад, інтегруючи гібридні схеми шифрування через бібліотеку Google Tink, де алгоритм RSA може використовуватися для захищеної передачі сесійних

ключів симетричного шифрування. Це забезпечує високу швидкість обробки великих масивів даних при збереженні математичної надійності асиметричної криптографії.

Для графічного представлення алгоритму шифрування було створено блок-схему на рисунку А.5.

3.2 Програмна реалізація модулів

Процес розробки було організовано у порядку від нижчих рівнів абстракції до вищих. Це дозволило спочатку сформувати надійний фундамент для роботи з інформацією, а згодом нарощувати функціональні можливості користувацького інтерфейсу та управління станами.

Першим фундаментальним етапом реалізації стало створення шару локального зберігання даних на основі бібліотеки Room. Головним інструментом для забезпечення доступу до історії переписки в офлайн-режимі виступає сутність MessageEntity, яка відображає структуру таблиці повідомлень у локальній базі даних SQLite. Цей клас інкапсулює унікальний локальний ідентифікатор із функцією автоматичної генерації, посилання на конкретну кімнату чату та ідентифікатор відправника, приклад цього класу зображено на рисунку 3.1.

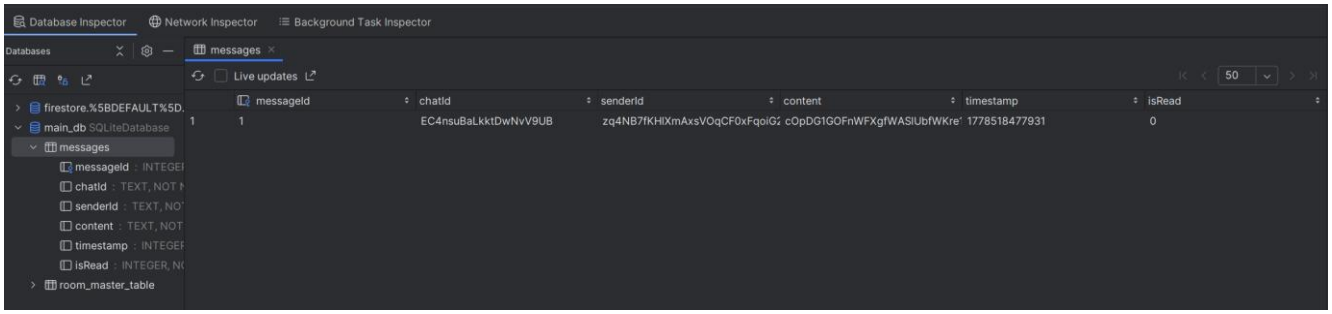


Рисунок 3.1 – Приклад об’єкту класу «MessageEntity»

Структуру сутностей та їхні взаємозв'язки на рівні локальної бази даних детально відображено на ER-діаграмі, яка зображена на рисунку 3.2.

Крім того, структура локальної бази передбачає зберігання часової мітки та логічного прапорця статусу прочитання повідомлення. Взаємодія з цими даними здійснюється через спеціалізований інтерфейс об'єктів доступу до даних MessagesDAO. Він надає методи для збереження нових записів, видалення історії та реактивного отримання списку повідомлень за допомогою механізму асинхронних потоків Flow. Використання реактивних потоків дозволяє інтерфейсу автоматично оновлюватися при надходженні нових повідомлень без необхідності циклічного опитування бази даних, що суттєво економить системні ресурси.

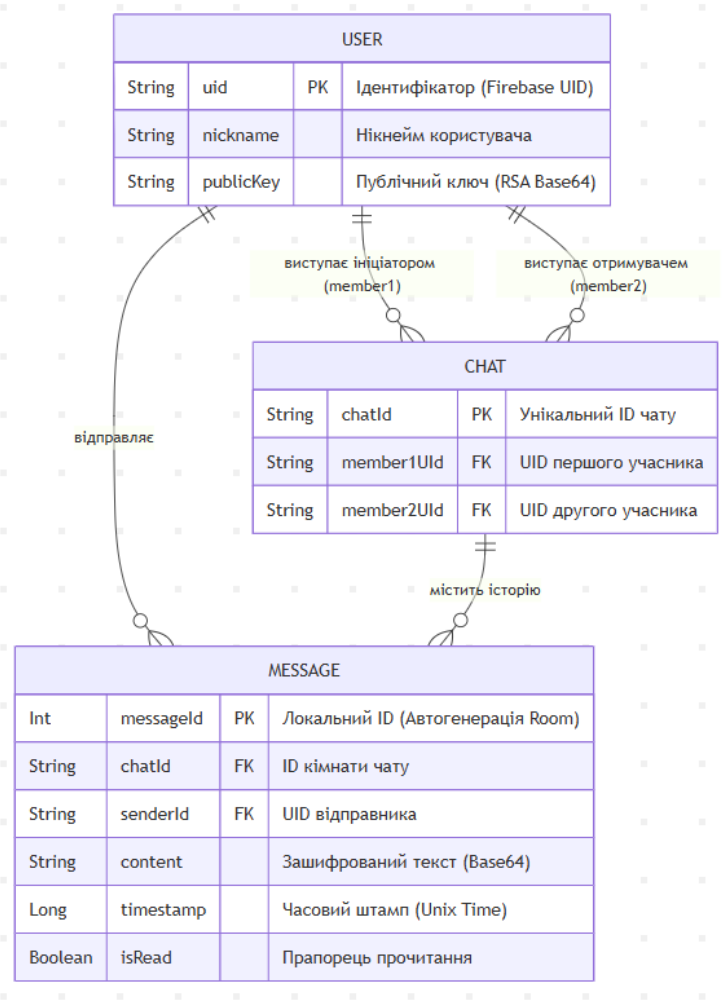


Рисунок 3.2 – ER-діаграма

Наступним кроком була програмна реалізація мережевої взаємодії за допомогою хмарної платформи Firebase. Як основне транзитне сховище для динамічних даних було обрано документо-орієнтовану базу Cloud Firestore, структура якої зображено на рисунку 3.3.

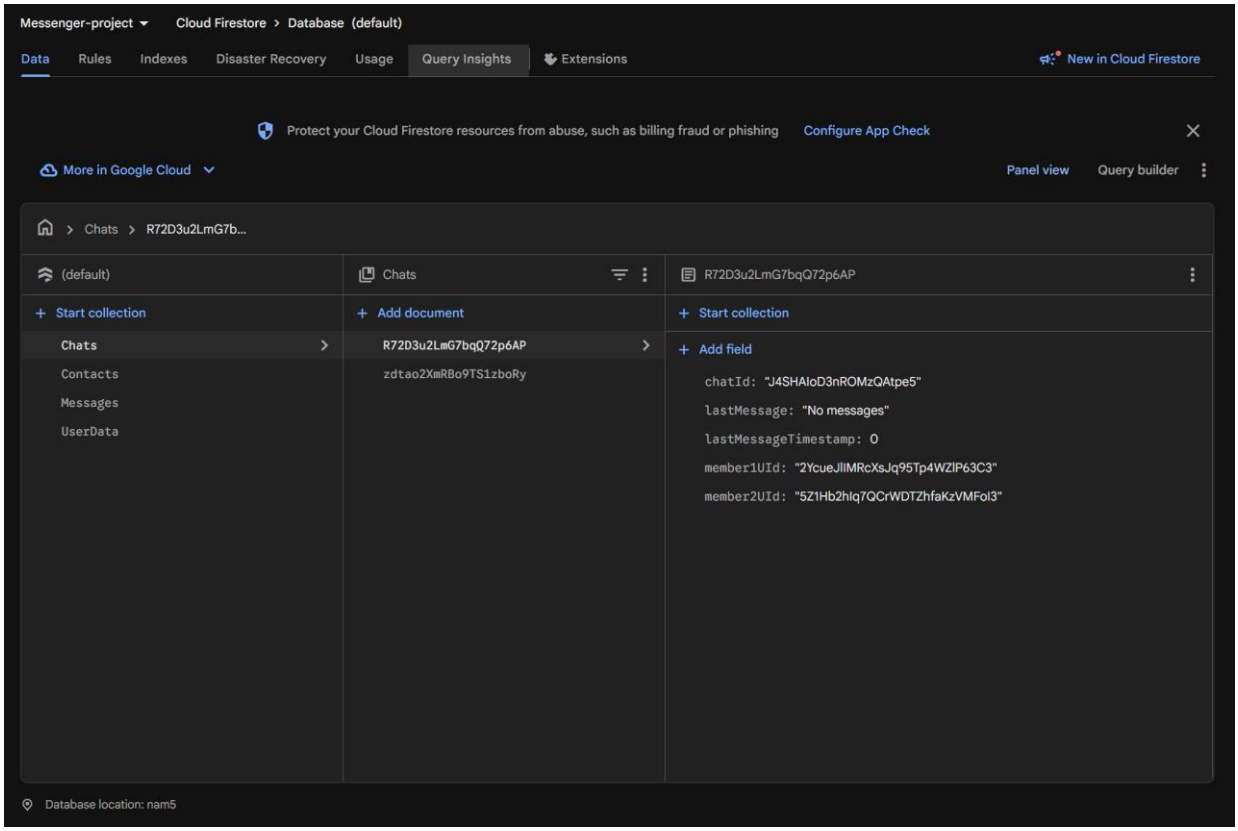


Рисунок 3.3 – Структура документо-орієнтованої бази Cloud Firestore

Особливістю програмної реалізації класу FirebaseRepository є використання методу прослуховування знімків бази даних, який інтегровано у сучасний корутинний механізм мови Kotlin через функцію callbackFlow. Таке інженерне рішення дозволяє трансформувати асинхронні події сервера у стабільний потік даних, що забезпечує миттєве доставлення повідомлень у реальному часі зі збереженням високої енергоефективності пристрою.

Для забезпечення модульності, зручності масштабування та придатності системи до тестування було впроваджено механізм інверсії управління за

допомогою фреймворку Dagger 2. Логіка постачання об'єктів жорстко розділена на спеціалізовані модулі.

Системний модуль відповідає за надання доступу до контексту застосунку. Модуль мережевих дій та модуль репозиторію повідомлень забезпечують ініціалізацію клієнтів Firebase та об'єктів бази даних Room відповідно.

Окремо виділено модуль шифрування, який постачає класи криптографічних перетворень у місця їхнього безпосереднього використання. Загальну архітектуру впровадження залежностей та ієрархію зв’язків між інфраструктурними модулями представлено на діаграмі графа залежностей на рисунку 3.4.

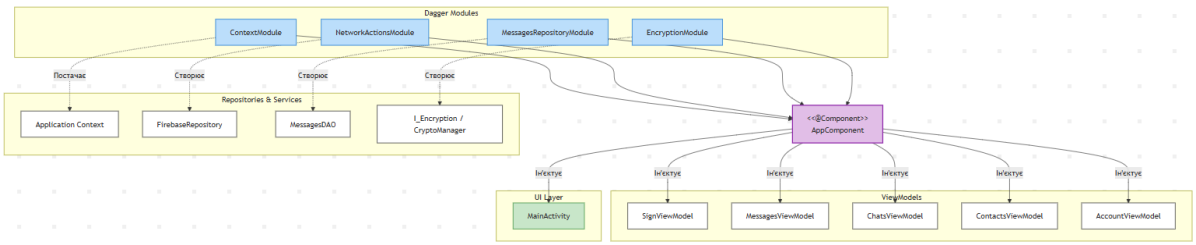


Рисунок 3.4 – Діаграма графа залежностей

Центральним вузлом управління всім графом залежностей виступає глобальний компонент застосунку. Він виконує автоматичну ін'єкцію необхідних об'єктів у життєвий цикл головних екранів та моделей відображення, позбавляючи класи необхідності самостійно створювати екземпляри складних сервісів.

Шар управління станом системи представлений набором із п'яти спеціалізованих класів ViewModel, де кожен об'єкт відповідає за ізольовану функціональну область. Модель авторизації SignViewModel керує процесами створення облікового запису та ініціалізує генерацію криптографічних пар ключів при першому вході в систему. Моделі чатів, контактів та профілю користувача обробляють події інтерфейсу та взаємодіють із глобальними списками Firebase.

Ядром комунікаційної системи виступає модель відображення повідомлень MessagesViewModel, яка координує найскладніший процес обміну інформацією. Логіка відправлення повідомлення є комбінованим транзакційним процесом. При ініціації відправлення модуль спочатку викликає метод шифрування, який використовує відкритий ключ співрозмовника для математичного перетворення тексту алгоритмом RSA. Після цього зашифроване повідомлення відправляється у колекцію Firestore через асинхронну корутину і паралельно зберігається у локальну базу Room для миттєвого відображення на екрані та забезпечення офлайн-доступу та захищеного збереження повідомлень.

Зворотний процес отримання ініціюється автоматично при спрацьовуванні мережевого слухача. Система перехоплює вхідний пакет, викликає метод дешифрування, вилучає локальний приватний ключ зі сховища AndroidKeyStore та відновлює оригінальний текст перед його передачею на рівень візуалізації.

Такий суворий розподіл функцій між архітектурними модулями дозволяє досягти високої швидкодії, унеможливлює витоки не шифрованих даних у мережу та забезпечує стабільність роботи користувацького інтерфейсу. Програмну реалізацію описаних вище методів шифрування та логіки взаємодії з репозиторіями наведено у Додатку Б.

3.3 Програмна реалізація інтерфейсу мобільного застосунку

Процес розробки інтерфейсу користувача базувався на використанні сучасного декларативного інструментарію Jetpack Compose.

Для створення уніфікованого та сучасного вигляду всіх елементів було впроваджено бібліотеку Material Design 3. Це забезпечило доступ до широкого набору готових компонентів, таких як кнопки, картки та текстові поля.

Фундаментом візуальної ідентифікації програми стала кастомна тема, реалізована у файлах конфігурації палітри кольорів. У межах цієї теми було перевизначено стандартні колірні схеми для світлого та темного режимів роботи.

Архітектура інтерфейсу побудована за принципом Single Activity Architecture, де єдина активність MainActivity виступає контейнером для всіх екранів. Навігація між різними станами застосунку, такими як екрани авторизації, списку чатів та безпосередньо вікна листування, реалізована за допомогою компонента Jetpack Compose Navigation.

Процес реєстрації та первинного налаштування профілю користувача розділено на кілька логічних етапів. Екрани SignInScreen та SignUpScreen забезпечують автентифікацію, тоді як UserInfoScreen дозволяє заповнити основні дані профілю та завантажити фотографію.

Результат реалізації екрану реєстрації на рисунку 3.5.

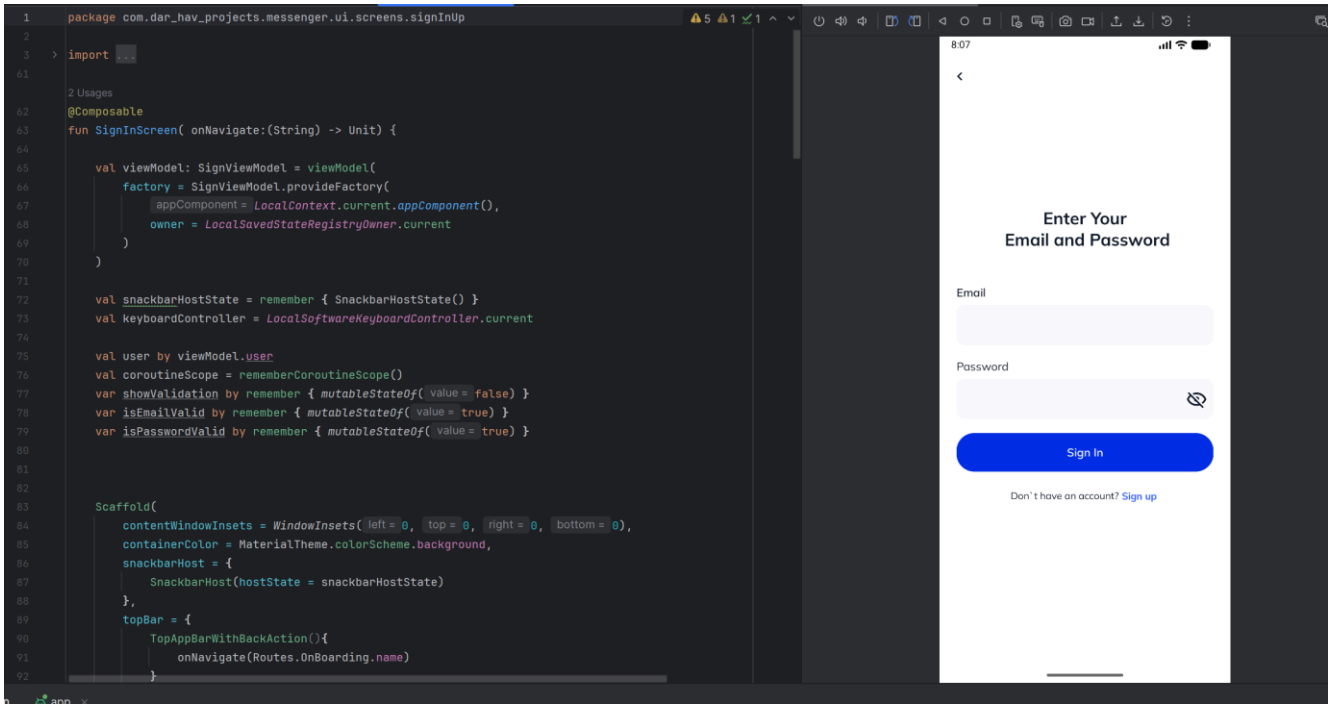


Рисунок 3.5 – Реалізація екрану реєстрації

Головний екран додатка зі списком активних чатів базується на використанні компонента LazyColumn. Даний інструмент забезпечує ефективний відображення елементів списку, що дозволяє плавно відображати велику кількість діалогів. Візуальне представлення списку чатів наведено на рисунку 3.6.

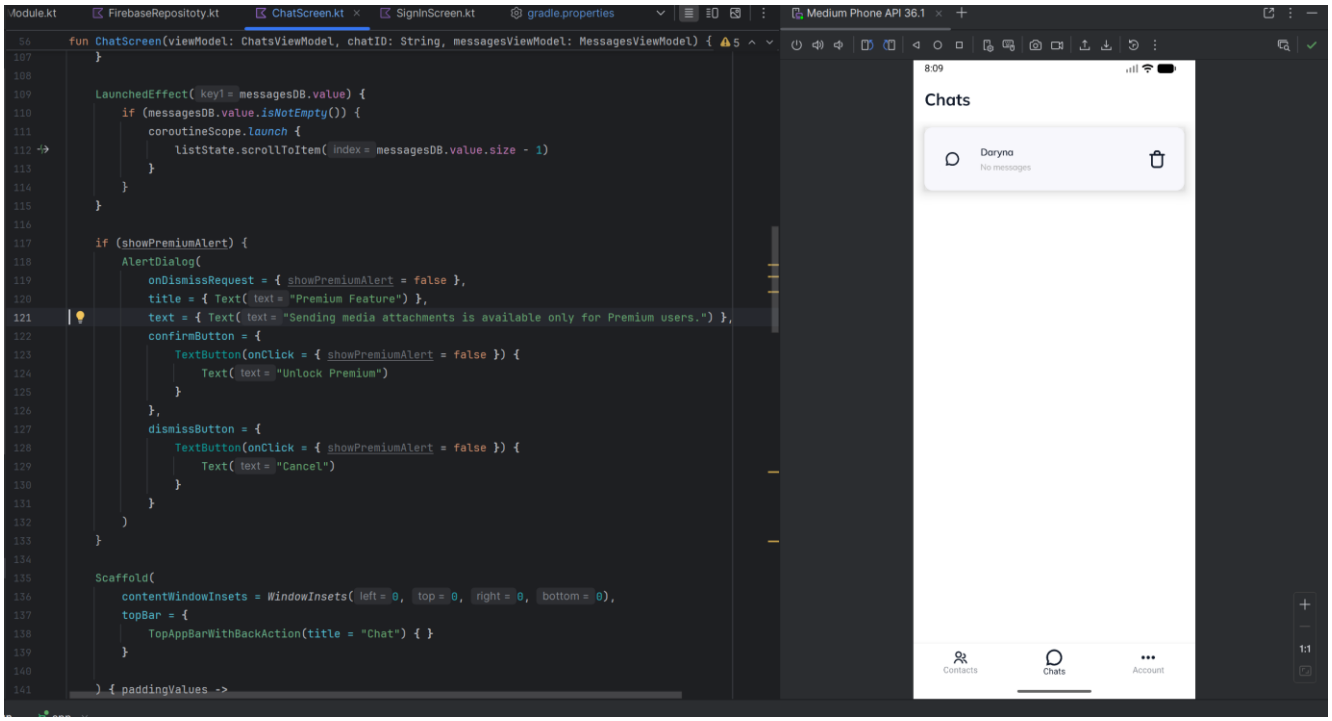


Рисунок 3.6 – Реалізація екрану списку чатів

Найбільш складним компонентом інтерфейсу є екран повідомлень ChatScreen, побудований на базі структури Scaffold. Історія листування відображається через вертикальний список, який автоматично прокручується до останнього повідомлення при отриманні нових даних.

Історія листування відображається за допомогою вертикального списку, що автоматично прокручується до найновішого елемента під час надходження нових повідомлень у реальному часі. Візуалізація кожного окремого повідомлення реалізована через кастомний компонент MessageCard.

Цей компонент підтримує логіку динамічного відображення та змінює своє позиціонування, колірну палітру і форму контейнера залежно від того, є користувач відправником чи отримувачем.

Нижня панель введення тексту містить заокруглене поле TextField для введення символів, а також інтегровані кнопки для прикріплення мультимедійних файлів і безпосереднього відправлення даних, як це продемонстровано на схемі інтерфейсу екрану повідомлень (Рисунок 3.7).

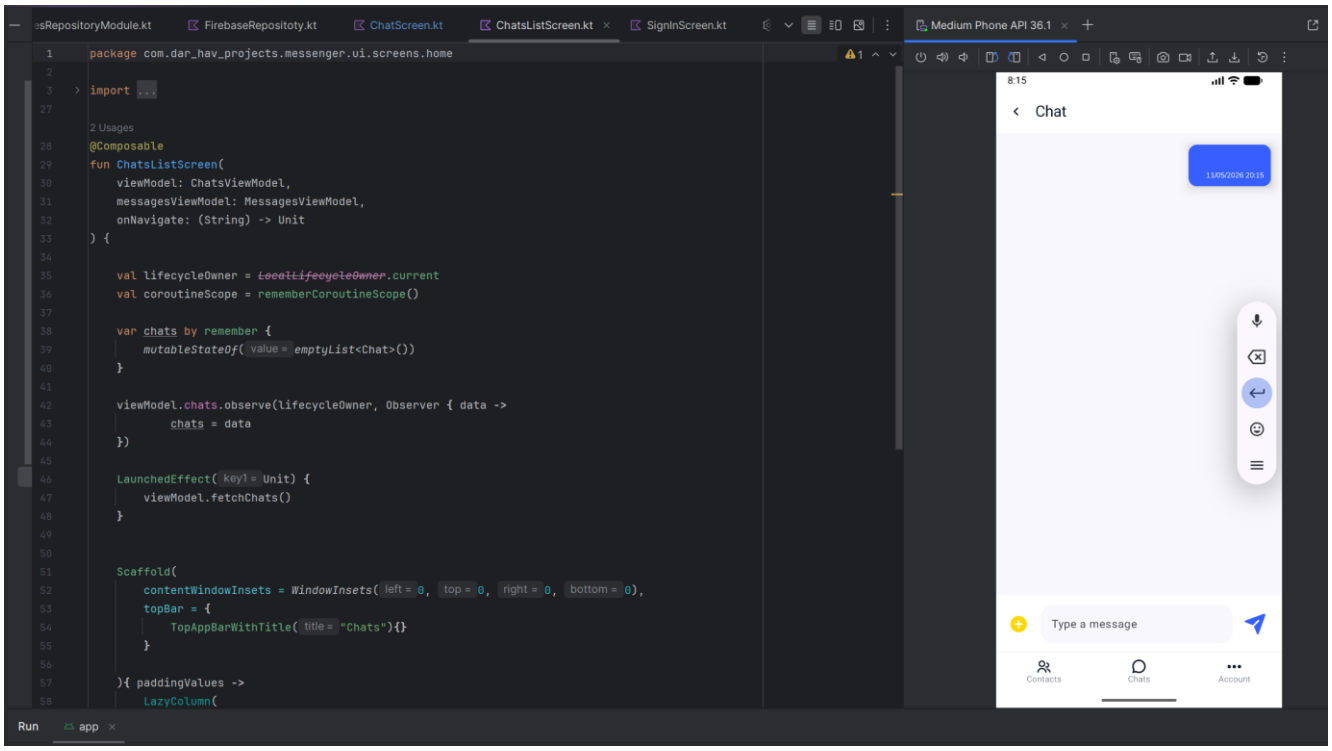


Рисунок 3.7 – Реалізація екрану чату

Реактивність інтерфейсу забезпечується через механізм спостереження за станами у ViewModel. Дані з локальної бази Room прослуховуються через потоки Flow за допомогою методу collectAsState, а мережеві події з Firebase відстежуються через LiveData. Такий підхід гарантує миттєве оновлення лише необхідних частин графічного інтерфейсу при зміні станів під капотом застосунку.

3.4 Вимоги до апаратного забезпечення

Стабільна робота мобільного месенджера та можливість його подальшої компіляції вимагають дотримання визначених апаратних і програмних специфікацій. Оскільки архітектура застосунку передбачає інтенсивні криптографічні обчислення, а також використання апаратного сховища Android Keystore, до мобільних пристроїв висуваються жорсткі вимоги щодо мінімального рівня операційної системи та наявності ізольованого середовища виконання

Trusted Execution Environment. Зокрема, для забезпечення коректної інжекції ключів та стабільної рекомпозиції інтерфейсу необхідна платформа на базі Android 8.0 або вище, а також процесор із підтримкою 64-бітних обчислень.

Зведені ключові характеристики для цільового пристрою користувача та робочої станції, яка буде використовуватись для завантаження застосунку наведено у таблиці 3.1.

Таблиця 3.1 – Мінімальні та рекомендовані вимоги до апаратного і програмного забезпечення

Характеристика	Мінімальні вимоги	Рекомендовані вимоги
Операційна система	Android 7.0 - API 24	Android 14 - API 34
Оперативна пам'ять	2 ГБ	3 ГБ та більше
Внутрішня пам'ять	50 МБ вільного простору	Залежить від локального кешу Room
Процесор	Наявність TEE / HSM модуля	Багатоядерний з частотою від 1.4 ГГц
Мережеве з'єднання	1 Мбіт/с	Стабільне широкосмугове з'єднання
Дисплей	Роздільна здатність 360 × 640 dp	Підтримка Edge-to-Edge архітектури

3.5 Тестування застосунку

3.5.1 Вибір та обґрунтування методів тестування застосунку

Вибір конкретних видів перевірки був продиктований архітектурними особливостями застосунку та його специфічними вимогами, такими як наскрізне шифрування та локальне зберігання даних. Загалом для проєкту було реалізовано чотири рівні тестування, кожен з яких покриває певну ланку архітектури системи.

Базовим рівнем перевірки було обрано модульне тестування завдяки його швидкості та здатності тестувати бізнес-логіку ізольовано від фреймворку Android.

Модульне тестування – це метод розробки через тестування для оцінки програмного забезпечення, який приділяє особливу увагу окремому компоненту або одиниці коду – найменшому можливому інкременту. Цей метод передбачає ізоляцію модулів, щоб їхню функціональність можна було підтвердити до того, як вони будуть інтегровані з іншими частинами застосунку [37].

Оскільки месенджер містить велику кількість суворих правил валідації та логіки обробки даних користувача, необхідно гарантувати їх безпомилкову роботу на найнижчому рівні.

Наступним етапом стало інтеграційне тестування. Відповідно до сучасних практик, головною метою інтеграційного тестування є перевірка коректності взаємодії та безперебійного обміну даними між різними модулями, компонентами або підсистемами після їхнього об'єднання в єдину групу, що дозволяє виявити дефекти на стиках інтерфейсів [38].

Месенджер активно використовує локальну базу даних Room для збереження історії листування та забезпечення офлайн-доступу, тому виникла критична необхідність перевірити коректність взаємодії між об'єктами доступу до даних та самою базою.

Окрему увагу було приділено тестуванню безпеки та криптографії. Для сучасного комунікаційного застосунку конфіденційність переданих даних є

найвищим пріоритетом, тому стандартних видів тестування виявилося недостатньо. Як зазначається в сучасних дослідженнях, алгоритми асиметричного шифрування, зокрема RSA, постійно піддаються загрозам з боку складних векторів атак на основі аналізу часу виконання або апаратних збоїв. Це робить критично важливою імплементацію ретельної перевірки механізмів математичної генерації та ізольованого зберігання ключів [39].

Останнім рівнем перевірки стало тестування інтерфейсу користувача. Використання сучасного декларативного UI-фреймворку Jetpack Compose вимагало специфічного підходу до перевірки реакції компонентів на зміну стану та дії користувача.

Відповідно до сучасних підходів забезпечення якості, тестування інтерфейсу має гарантувати не лише коректність візуального відображення ізольованих елементів, а й перевірку того, як користувацькі дії взаємодіють із бізнес-логікою та як система реагує на крайні випадки чи помилки обробки даних, забезпечуючи безперебійний користувацький досвід [40].

Відповідно до сучасних досліджень, саме такий багатокомпонентний підхід є вирішальним для забезпечення найвищої якості та надійності програмного продукту [41]. Тому для застосунку була використана комплексна стратегія тестування, яка допомагає повністю перевірити увесь функціонал застосунку.

3.5.2 Валідація та верифікація програмного забезпечення

Верифікація програмного забезпечення здійснювалася на кількох рівнях архітектури. Для підтвердження вимог ТЗ щодо безпеки даних та конфіденційності листування було реалізовано модульні тести криптографічного ядра.

Ці тести верифікують коректність математичних перетворень алгоритмом RSA та підтверджують неможливість доступу до приватних ключів в обхід механізмів Android Key Store.

Вимоги ТЗ щодо збереження історії повідомлень та забезпечення офлайн-доступу верифікувалися за допомогою інтеграційних тестів. Вони підтверджують правильність структури локальної бази даних та здатність системи швидко обробляти запити на збереження і вилучення зашифрованих пакетів даних.

Валідація готового продукту, тобто перевірка його відповідності очікуванням кінцевого користувача, здійснювалася шляхом тестування інтерфейсу. Використовуючи інструментарій ComposeTestRule, перевіряє чи застосунок реагує на дії користувача та своєчасно відображає зміни стану системи.

Таким чином, комплекс проведених перевірок підтвердив, що розроблений мобільний месенджер відповідає заявленим у ТЗ функціональним та нефункціональним вимогам.

3.5.3 Аналіз результатів тестування

Процес аналізу дозволяє не просто констатувати факт успішного виконання створених тестових сценаріїв, а й здійснити комплексну оцінку якості, стабільності та надійності мобільного застосунку. Детальний розгляд зафіксованих показників дає змогу підтвердити ефективність обраних архітектурних рішень, переконатися у високій стійкості реалізованих криптографічних алгоритмів та гарантувати повну відповідність готового продукту всім критеріям технічного завдання.

Детальні результати перевірки ізольованих компонентів системи та доменних моделей зведено у таблиці 3.2.

Проведене модульне тестування підтвердило коректність роботи алгоритмів валідації користувацьких даних та внутрішньої бізнес-логіки застосунку. Усі розроблені сценарії завершилися успішно, що гарантує надійність базового рівня архітектури перед переходом до етапів інтеграції.

Таблиця 3.2 – Результати модульного тестування

Категорія тестування	Опис сценарію	Очікувана реакція програми	Результат
1. Модульне тестування	Введення пароля коротшого за 8 символів	Система розпізнає пароль як невалідний	Успішно
	Введення пароля без спецсимволів	Система розпізнає пароль як невалідний	Успішно
	Введення валідного пароля	Пароль проходить перевірку успішно	Успішно
	Перевірка email без символу @	Система розпізнає email як невалідний	Успішно
	Маппінг полів користувача з Firebase	Усі поля переносяться коректно в об'єкт	Успішно
	Спроба перейменування чату без преміум-статусу	Система блокує операцію та повертає false	Успішно

Наступним етапом перевірки став аналіз взаємодії між ключовими архітектурними компонентами застосунку, результати якого наведено у таблиці 3.3. Проведене інтеграційне тестування дозволило верифікувати коректність обміну даними між логічними модулями та локальним сховищем Room.

Успішне виконання всіх сценаріїв підтвердило надійність інтеграційних інтерфейсів та гарантувало стабільну роботу системи при виконанні складних операцій із базою даних.

Таблиця 3.3 – Результати інтеграційного тестування

2. Інтеграційне тестування	Збереження повідомлення для чату	Повідомлення успішно зберігається та вилучається	Успішно
	Отримання повідомлень конкретного чату	Повертаються дані суворо для вказаного ID чату	Успішно
	Сортування повідомлень	Повідомлення автоматично відсортовані за часом	Успішно
	Видалення історії чату	Історія повністю очищається	Успішно
	Вставка повідомлення з існуючим ID	Старий запис коректно замінюється новим	Успішно

Особливе місце у загальній структурі перевірок посідає оцінка захищеності даних, результати якої відображено у таблиці 3.4. Проведене тестування безпеки підтвердило високу стійкість криптографічного ядра при виконанні операцій асиметричного шифрування алгоритмом RSA. Окрім цього, було додатково

проведено тестування в іншому незалежному ресурсі, результати якого наочно наведені на рисунку А.6.

Таблиця 3.4 – Результати тестування безпеки застосунку

3. Тестування безпеки	Генерація пари ключів	Ключова пара RSA успішно генерується	Успішно
	Збереження ключа	Ключ успішно вилучається зі сховища	Успішно
	Шифрування повідомлення	Зашифрований текст повністю відрізняється від оригіналу	Успішно
	Безпека шифрування	Однакові дані генерують різний	Успішно
	Дешифрування тексту	Оригінальний текст успішно відновлюється	Успішно
	Спроба дешифрування чужим ключем	Система генерує помилку доступу	Успішно

Заключним етапом комплексної перевірки став аналіз поведінки графічного інтерфейсу та його безпосередньої взаємодії з користувачем, підсумки якого представлено у таблиці 3.5. Успішне виконання всіх графічних сценаріїв довело коректність зміни станів екранів, надійність обробки подій натискання та повну відповідність візуальної реалізації месенджера затвердженим макетам дизайну.

Таблиця 3.5 – Результати тестування інтерфейсу

4. Тестування інтерфейсу	Відображення тексту в полі email	Введений текст коректно відображається на екрані	Успішно
	Валідація інтерфейсу	З'являється повідомлення про помилку при невалідних даних	Успішно
	Клік на кнопку	Система коректно викликає подію onClick	Успішно
	Валідація форми авторизації	Форма показує помилку, якщо натиснути відправку з порожнім полем	Успішно

Узагальнюючи результати проведених випробувань, можна зробити висновок про високу надійність розробленого мобільного застосунку. Успішне виконання всіх чотирьох рівнів тестових сценаріїв підтвердило стабільність функціонування кожної окремої ланки архітектури та системи в цілому.

Комплексна верифікація дозволила підтвердити коректність ізольованих криптографічних обчислень, стабільність реактивного інтерфейсу та надійність синхронізації даних із хмарною інфраструктурою Firebase. Отримані метрики успішності тестів доводять стійкість месенджера до критичних навантажень та його повну готовність до реальної експлуатації.

3.6 Висновки

У цьому розділі було успішно здійснено програмну реалізацію, розгортання та комплексне автоматизоване тестування безпечного мобільного месенджера на платформі Android.

По-перше, реалізовано криптографічний модуль на основі асиметричного алгоритму RSA. Завдяки успішній інтеграції з апаратним сховищем Android KeyStore забезпечено генерацію та ізольоване зберігання приватних ключів у захищеному середовищі Trusted Execution Environment, що повністю виключає ризик їхнього витоку чи перехоплення на рівні операційної системи.

По-друге, розроблено та впроваджено архітектурні модулі застосунку відповідно до шаблону MVVM. Організацію локального зберігання зашифрованої історії чатів забезпечено за допомогою бази даних Room. Транзитну маршрутизацію та синхронізацію інформаційних пакетів у реальному часі реалізовано через інтеграцію з хмарною системою Cloud Firestore.

По-третє, побудовано сучасний та енергоефективний графічний інтерфейс користувача на основі декларативного фреймворку Jetpack Compose та дизайну Material Design 3.

По-четверте, проведено повну верифікацію та валідацію створеного програмного забезпечення за допомогою чотирирівневої піраміди тестування. Розроблені модульні, інтеграційні, безпекові сценарії та графічні тести інтерфейсу завершилися зі стовідсотковим успіхом. Це дозволило експериментально підтвердити безпомилковість бізнес-логіки, стабільність транзакцій бази даних, стійкість шифрування та правильність динамічного відображення елементів. Результати випробувань довели відповідність розробленого програмного продукту всім критеріям висунутого технічного завдання та підтвердили його готовність до безпечної практичної експлуатації.

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальне науково-практичне завдання, що полягає у проєктуванні та розробці мобільної системи обміну миттєвими повідомленнями на платформі Android із впровадженням концепції наскрізного шифрування та апаратного захисту криптографічних ключів. На основі проведених досліджень, розроблених архітектурних рішень та результатів автоматизованого випробування було сформульовано узагальнені підсумки виконаної роботи.

На першому етапі дослідження предметної області було здійснено глибокий змістовий аналіз сучасного стану систем мобільної комунікації та виявлено критичні вразливості наявних централізованих архітектур. На основі цих висновків було сформульовано функціональні та нефункціональні вимоги до месенджера, а також побудовано діаграми варіантів використання та послідовності мовою UML.

Під час проєктування програмного забезпечення було обрано та науково обґрунтовано використання сучасного архітектурного шаблону MVVM у поєднанні з принципами чистої архітектури.

На етапі безпосередньої розробки та тестування було повністю перенесено спроєктовані моделі у готовий програмний продукт. Центральною ланкою системи безпеки стала програмна реалізація асиметричного алгоритму RSA, що гарантує високу стійкість до сучасних методів криптоаналізу.

Комплексна верифікація і валідація розробленого програмного забезпечення здійснювалася за допомогою побудови чотирирівневої піраміди тестування.

Впровадження розробленого програмного забезпечення надає користувачам суттєві переваги в порівнянні з масовими комерційними аналогами. Застосунок дозволяє повністю ліквідувати ризики витоку критично важливих даних через злам серверної інфраструктури, оскільки інформація залишає пристрій

відправника виключно в зашифрованому вигляді. Це дозволяє суттєво заощадити фінансові та людські ресурси на побудову складних систем захисту периметра мережі, адже безпека гарантується на рівні кінцевих точок. Завдяки реалізації концепції локального збереження повідомлень користувачі отримують можливість безперешкодно працювати з історією чатів за умов повної відсутності зв'язку, а оптимізований реактивний інтерфейс скорочує затрати часу на відправку та відображення нових пакетів даних, підвищуючи якість та швидкість комунікації.

Практична цінність розробленої системи дозволяє успішно використовувати її не лише для приватного конфіденційного спілкування громадян, але й у багатьох корпоративних та спеціалізованих галузях. Програмний продукт є ефективним інструментом для бізнес-організацій та комерційного сектору з метою захисту інтелектуальної власності, обговорення фінансових стратегій та запобігання промисловому шпигунству. Високий рівень захисту та повна прозорість використаних алгоритмів роблять застосунок придатним для впровадження у державній службі, дипломатичних установах та структурах оборонного комплексу, де вимоги до таємниці зв'язку є найвищими та мають вирішальне значення для національної безпеки.

Перспективами подальшого розвитку та можливими напрямками продовження роботи є інтеграція гібридних схем шифрування, де алгоритм RSA застосовуватиметься для захищеного обміну сесійними ключами, а безпосереднє шифрування великих обсягів мультимедійних даних виконуватиметься високошвидкісним симетричним алгоритмом AES-256.

Також важливим вектором модернізації є дослідження та впровадження постквантових криптографічних протоколів, стійких до атак із використанням квантових обчислювальних потужностей. Окрім цього, планується розширити функціонал застосунку шляхом додавання механізмів захищених голосових та відеодзвінків, а також реалізувати перехід від транзитного сервера до повністю децентралізованих P2P протоколів передачі пакетів даних.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Бедратюк Л. П., Радельчук Г. І. Кваліфікаційна робота: Методичні настанови для здобувачів першого (бакалаврського) рівня вищої освіти спеціальності 121 «Інженерія програмного забезпечення». Хмельницький: ХНУ, 2023. 60 с. (дата звернення: 05.02.2026).
2. Share of the population using the Internet. *Our World in Data* : вебсайт. URL: https://ourworldindata.org/grapher/share-of-individuals-using-the-internet (дата звернення: 05.02.2026).
3. Digital 2025: Ukraine. *DataReportal* : вебсайт. URL: https://datareportal.com/reports/digital-2025-ukraine (дата звернення: 20.02.2026).
4. 2025 ITRC Annual Data Breach Report. *Identity Theft Resource Center* : вебсайт. URL: https://idtheftcenter.org/wp-content/uploads/2026/01/2025-ITRC-Annual-Data-Breach-Report.pdf (дата звернення: 05.02.2026).
5. A Survey of Post-Quantum Cryptography: Start of a New Race / D.-T. Dam et al. // Cryptography. 2023. Vol. 7, No. 3. Art. No. 40. DOI: 10.3390/cryptography7030040. URL: https://www.mdpi.com/2410-387X/7/3/40 (дата звернення: 10.02.2026).
6. What Is Key Management? IBM : вебсайт. URL: https://www.ibm.com/think/topics/key-management (дата звернення: 10.02.2026).
7. Marlinspike M., Perrin T. The X3DH Key Agreement Protocol. *Signal* : вебсайт. URL: https://signal.org/docs/specifications/x3dh/ (дата звернення: 20.02.2026).
8. The Top 20 Reasons Startups Fail. *CB Insights* : вебсайт. URL: https://www.cbinsights.com/research/report/startup-failure-reasons-top/ (дата звернення: 12.02.2026).
9. Changes in media consumption in Ukraine in 2024. *Gradus Research* : вебсайт. URL: https://gradus.app/en/open-reports/changes-media-consumption-ukraine-2024/ (дата звернення: 14.02.2026).

10. Ставлення до медіа та споживання контенту (серпень 2025). *Соціологічна група «Рейтинг»* : вебсайт. URL: https://www.ratinggroup.ua/news/media-aug2025 (дата звернення: 14.02.2026).

11. Як змінилося медіаспоживання в Україні у 2024-му. *Детектор медіа* : вебсайт. URL: https://ms.detector.media/withoutsection/post/36734/2024-11-15-yak-zminylosya-mediaspozhyvannya-v-ukraini-u-2024-mu-doslidzhennya-gradus-research/ (дата звернення: 14.02.2026).

12. Cogliati B., Ethan J., Jha A. Subverting Telegram's End-to-End Encryption // IACR Transactions on Symmetric Cryptology. 2023. Vol. 2023, No. 1. P. 5–40. DOI: 10.46586/tosc.v2023.i1.5-40. URL: https://doi.org/10.46586/tosc.v2023.i1.5-40 (дата звернення: 14.02.2026).

13. Рейтинг мобільних додатків (квітень 2022). *Kantar Ukraine* : вебсайт. URL: https://www.kantar.com/ua/inspiration/advertising-media/mobile-app-ranking_april-2022 (дата звернення: 20.02.2026).

14. Sudozai M., Habib N., Saleem S., Khan A. Signatures of Viber Security Traffic // Journal of Digital Forensics, Security and Law. 2017. Vol. 12, No. 3. P. 109–120. DOI: 10.15394/jdfsl.2017.1477. URL: https://doi.org/10.15394/jdfsl.2017.1477 (дата звернення: 20.02.2026).

15. Paterson K., Scarlata M., Truong K. Three Lessons From Threema: Analysis of a Secure Messenger // USENIX Security Symposium. 2023. P. 1289–1306. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/paterson (дата звернення: 20.02.2026).

16. Bhuse V. Review of End-to-End Encryption for Social Media // Proceedings of the 18th International Conference on Cyber Warfare and Security (ICCWS 2023). 2023. Vol. 18, No. 1. P. 35–37. DOI: 10.34190/iccws.18.1.1017. URL: https://www.researchgate.net/publication/368905447_Review_of_End-to-End_Encryption_for_Social_Media (дата звернення: 20.02.2026).

17. MVC Architecture: Model, View, and Controller. Codecademy : вебсайт. URL: https://www.codecademy.com/article/mvc-architecture-model-view-controller (дата звернення: 17.04.2026).

18. Architectural Pattern - Model View Presenter (MVP). DEV Community : вебсайт. URL: https://dev.to/binoy123/architectural-pattern-model-view-presenter-mvp-28hl (дата звернення: 17.04.2026).

19. Wisnuadhi B., Munawar G., Wahyu U. Performance Comparison of Native Android Application on MVP and MVVM // Advances in Engineering Research. 2020. Vol. 198. P. 276–282. DOI: 10.2991/aer.k.201221.047. URL: https://www.atlantis-press.com/article/125949769.pdf (дата звернення: 17.04.2026).

20. Introduction to MVVM Architecture. Medium : вебсайт. URL: https://medium.com/@onurcem.isik/introduction-to-mvvm-architecture-5c5558c3679 (дата звернення: 17.04.2026).

21. MVVM (Model View ViewModel) Architecture Pattern in Android. GeeksforGeeks : вебсайт. URL: https://www.geeksforgeeks.org/android/mvvm-model-view-viewmodel-architecture-pattern-in-android/ (дата звернення: 17.04.2026).

22. MVVM Architecture in Android. Outcome School : вебсайт. URL: https://outcomeschool.com/blog/mvvm-architecture-android (дата звернення: 17.04.2026).

23. Pratama A. N., Ardiani F. Optimization of Model-View-ViewModel (MVVM) Architecture Pattern and RESTfull API on Android-based E-Learning Application // International Journal of Computer Applications. 2023. Vol. 185, No. 45. P. 4–11. DOI: 10.5120/ijca2023923261. URL: https://www.ijcaonline.org/archives/volume185/number45/pratama-2023-ijca-923261.pdf (дата звернення: 17.04.2026).

24. Nunkesser R. Choosing a Global Architecture for Mobile Applications. *TechRxiv* : вебсайт. 2021. DOI: 10.36227/techrxiv.14212571. URL: https://www.techrxiv.org/doi/pdf/10.36227/techrxiv.14212571 (дата звернення: 20.04.2026).

25. Implementation of Secure End-to-End Encrypted Chat Application Using Diffie–Hellman Key Exchange and AES-256 in a Microservice Architecture. MDPI : веб-сайт. 2025. URL: https://www.mdpi.com/2673-4591/107/1/98 (дата звернення: 20.04.2026).

26. What is Firebase Realtime: A Review of Serverless Database Features. Bejamas : веб-сайт. 2025. URL: https://bejamas.com/hub/serverless-database/firebase-realtime-database (дата звернення: 20.04.2026).

27. Build Better Android Apps with MVVM & Clean Architecture. iFlair : веб-сайт. 2025. URL: https://www.iflair.com/building-better-android-apps-with-mvvm-and-clean-architecture/ (дата звернення: 20.04.2026).

28. Smith R. Kotlin vs Java for Android App Development in Modern Codebases. *DEV Community* : вебсайт. 2025. URL: https://dev.to/raulsmithus/kotlin-vs-java-for-android-app-development-in-modern-codebases-3i5j (дата звернення: 24.04.2026).

29. Komilov Y. Jetpack Compose vs XML in Android: A Detailed Comparison. *Medium* : вебсайт. URL: https://medium.com/@YodgorbekKomilo/jetpack-compose-vs-xml-in-android-a-detailed-comparison-aa411f5391d9 (дата звернення: 24.04.2026).

30. Coroutines guide. *Kotlin Documentation* : вебсайт. URL: https://kotlinlang.org/docs/coroutines-overview.html (дата звернення: 24.04.2026).

31. Why Choose Room Database. *Medium* : вебсайт. URL: https://medium.com/@myofficework000/why-choose-room-database-42ab458334b5 (дата звернення: 24.04.2026).

32. Stevenson D. What is Firebase? The Complete Story, Abridged. *Firebase Developers* : вебсайт. 2020. URL: https://medium.com/firebase-developers/what-is-firebase-the-complete-story-abridged-bcc730c5f2c0 (дата звернення: 24.04.2026).

33. Dependency Injection with Dagger. *Android Developers* : вебсайт. URL: https://developer.android.com/training/dependency-injection/dagger-android (дата звернення: 24.04.2026).

34. Google Tink. *Google Developers* : вебсайт. URL: https://developers.google.com/tink (дата звернення: 24.04.2026).

35. Coil: My favorite Image Loading library for Jetpack Compose. *ProAndroidDev* : вебсайт. URL: https://proandroiddev.com/coil-my-favorite-image-loading-library-for-jetpack-compose-877fa0b818fe (дата звернення: 24.04.2026).

36. Kota C. M., Aissi C. Implementation of the RSA algorithm and its cryptanalysis. Proceedings of the 2002 ASEE Gulf-Southwest Annual Conference. 2002. URL: https://peer.asee.org/implementation-of-the-rsa-algorithm-and-its-cryptanalysis (дата звернення: 09.05.2026).

37. Powell P., Smalley I. What is unit testing? IBM Think : вебсайт. URL: https://www.ibm.com/think/topics/unit-testing (дата звернення: 09.05.2026).

38. Zaidi S. Integration Testing: A Comprehensive guide with best practices. Opkey : вебсайт. 2024. URL: https://www.opkey.com/blog/integration-testing-a-comprehensive-guide-with-best-practices (дата звернення: 09.05.2026).

39. Liu Z. Evaluating RSA encryption: Primality testing, pollard's algorithms, and security challenges. Proceedings of the 3rd International Conference on Computing Innovation and Applied Physics. 2024. URL: https://tns.ewapub.com/article/view/7932 (дата звернення: 09.05.2026).

40. Kumar R. What is Interface Testing and How to Test it End to End. Virtuoso QA. 2025. URL: https://www.virtuosoqa.com/post/interface-testing (дата звернення: 09.05.2026).

41. Bajjouk M., Rana M. E., Ramachandiran C. R., Chelliah S. Software testing for reliability and quality improvement. *Journal of Applied Technology and Innovation*. 2021. Vol. 5, No. 2. P. 40–46. URL: https://jati.apu.edu.my/index.php/JATI/article/view/216 (дата звернення: 09.05.2026).

ДОДАТОК А
(обов’язковий)
ГРАФІЧНІ МАТЕРІАЛИ

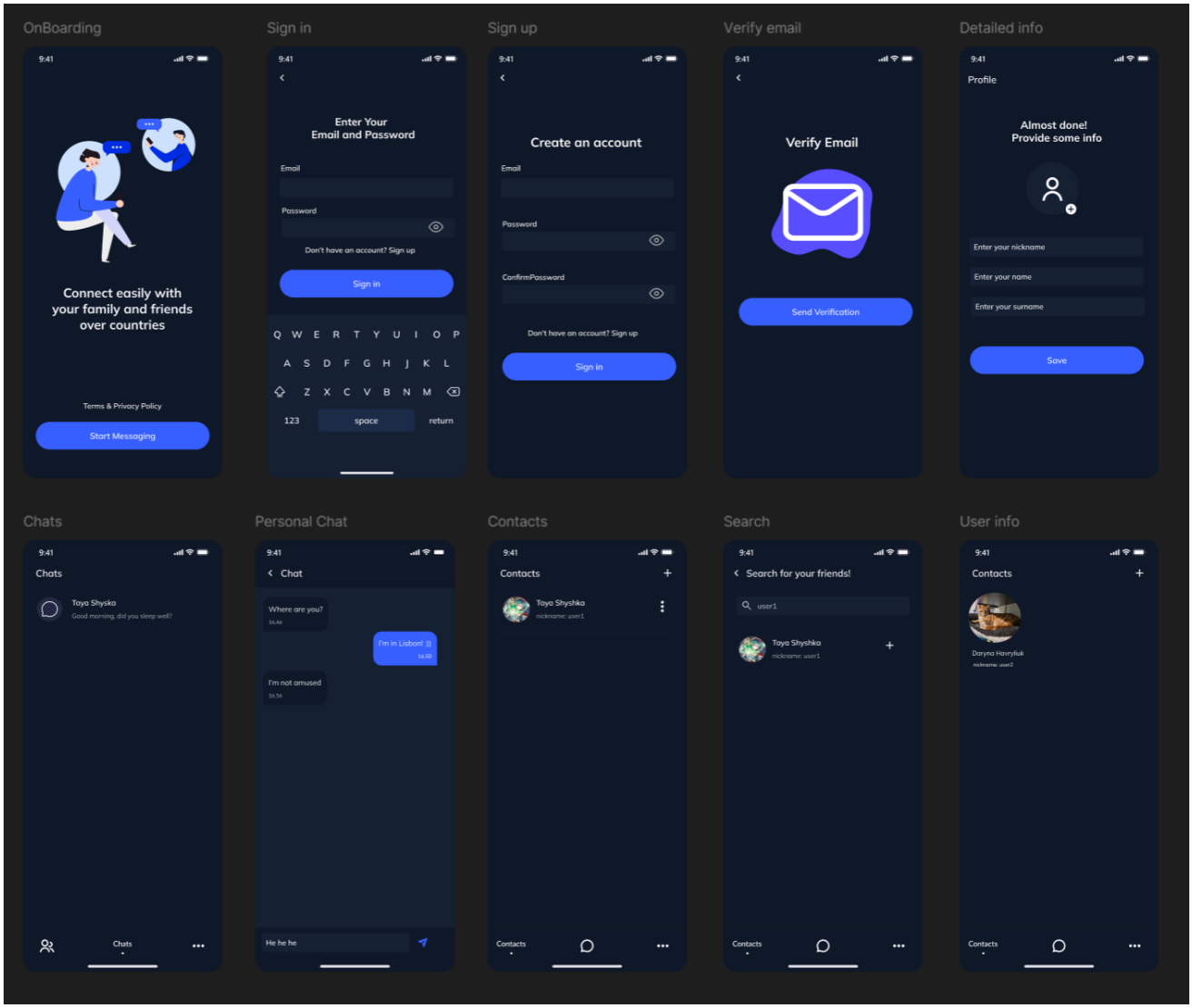


Рисунок А.1 – Макет ключових вікон застосунку

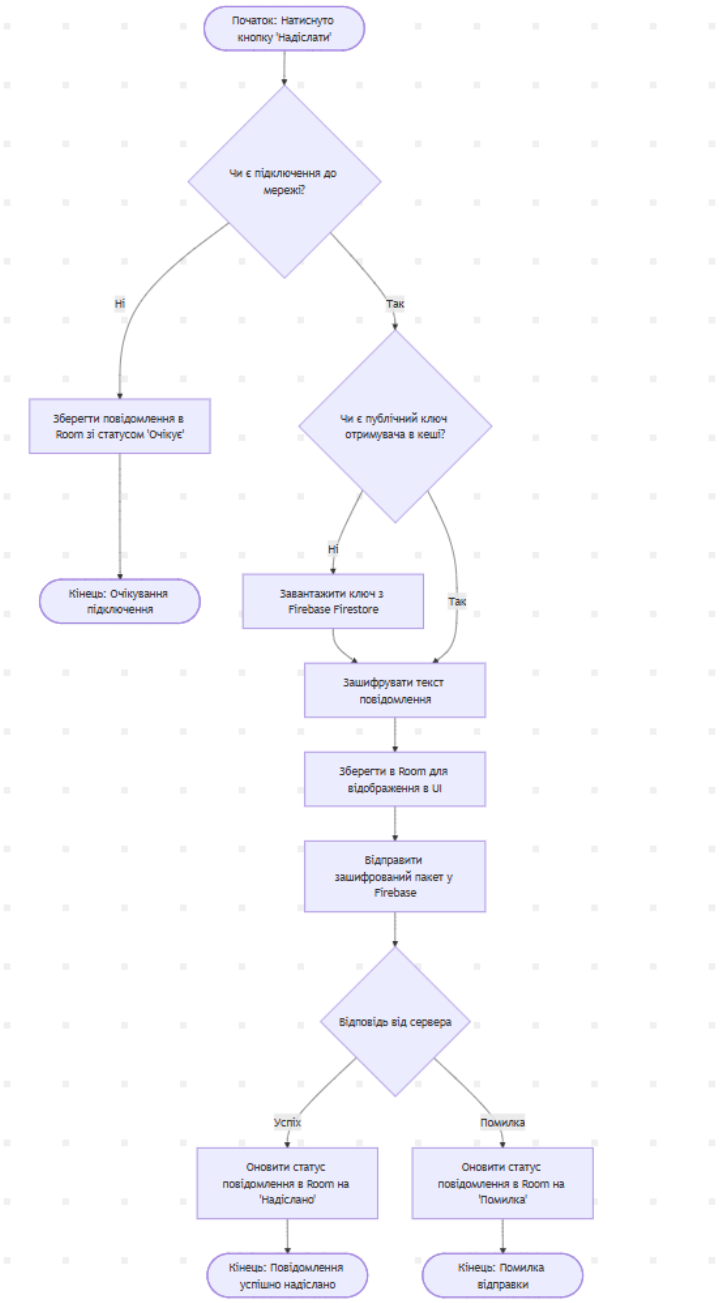


Рисунок А.2 – Діаграма діяльності

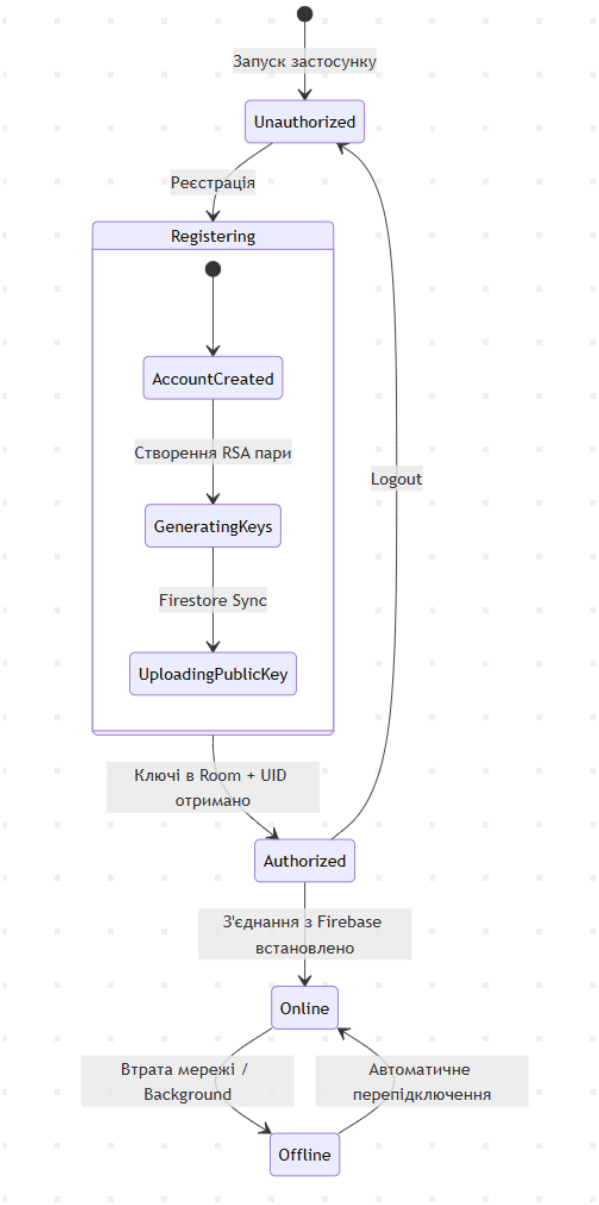


Рисунок А.3 – Діаграма станів сесії користувача

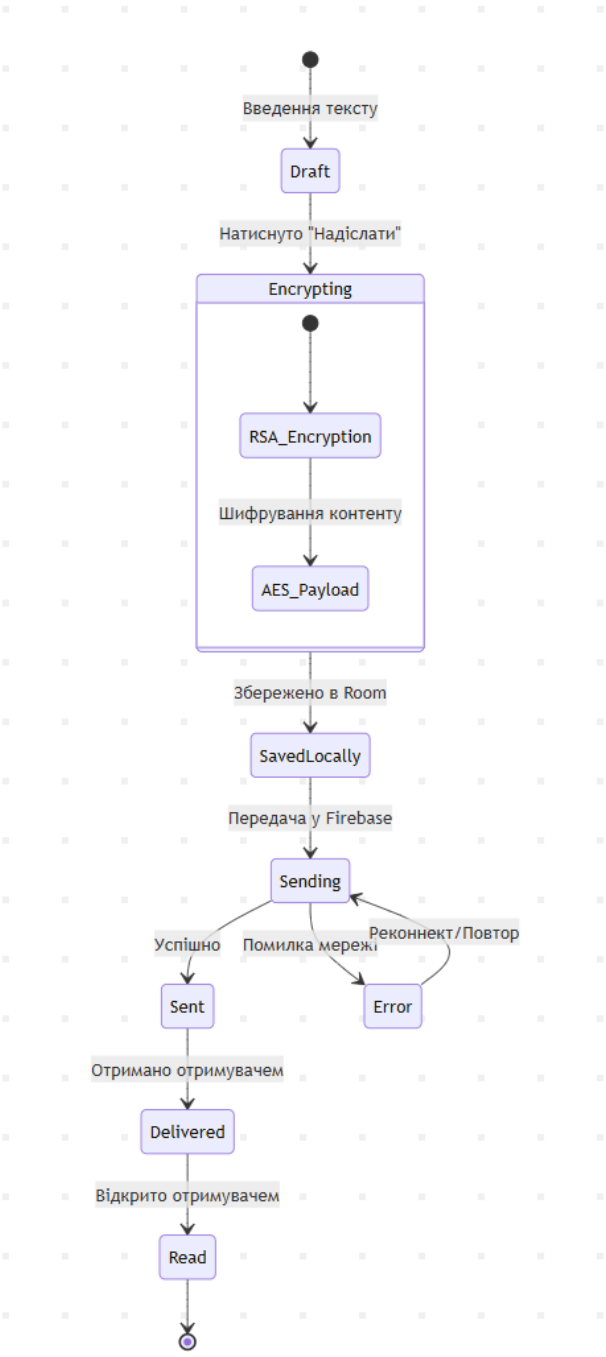


Рисунок А.4 – Діаграма станів повідомлення

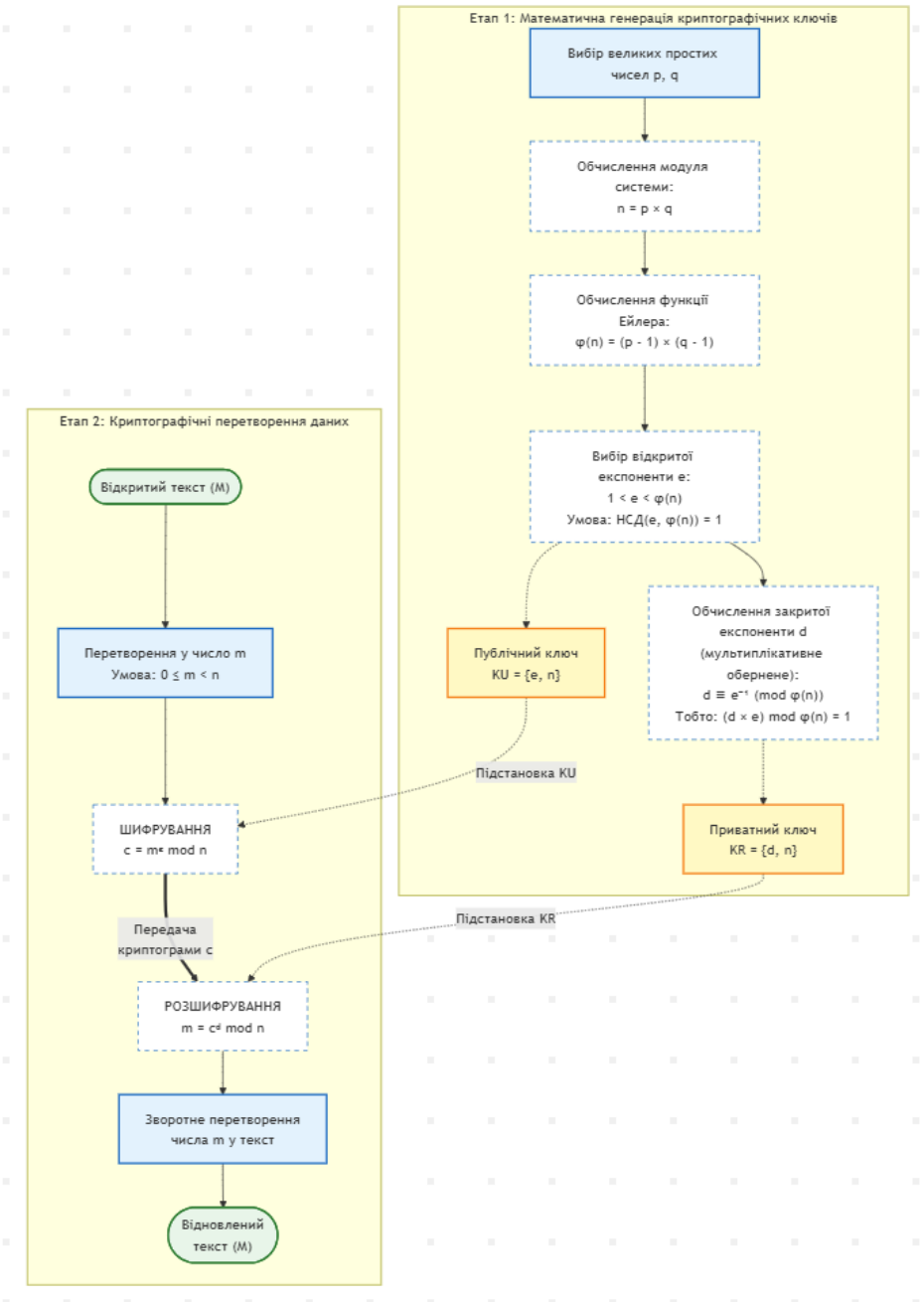


Рисунок А.5 – Блок-схема алгоритму шифрування RSA

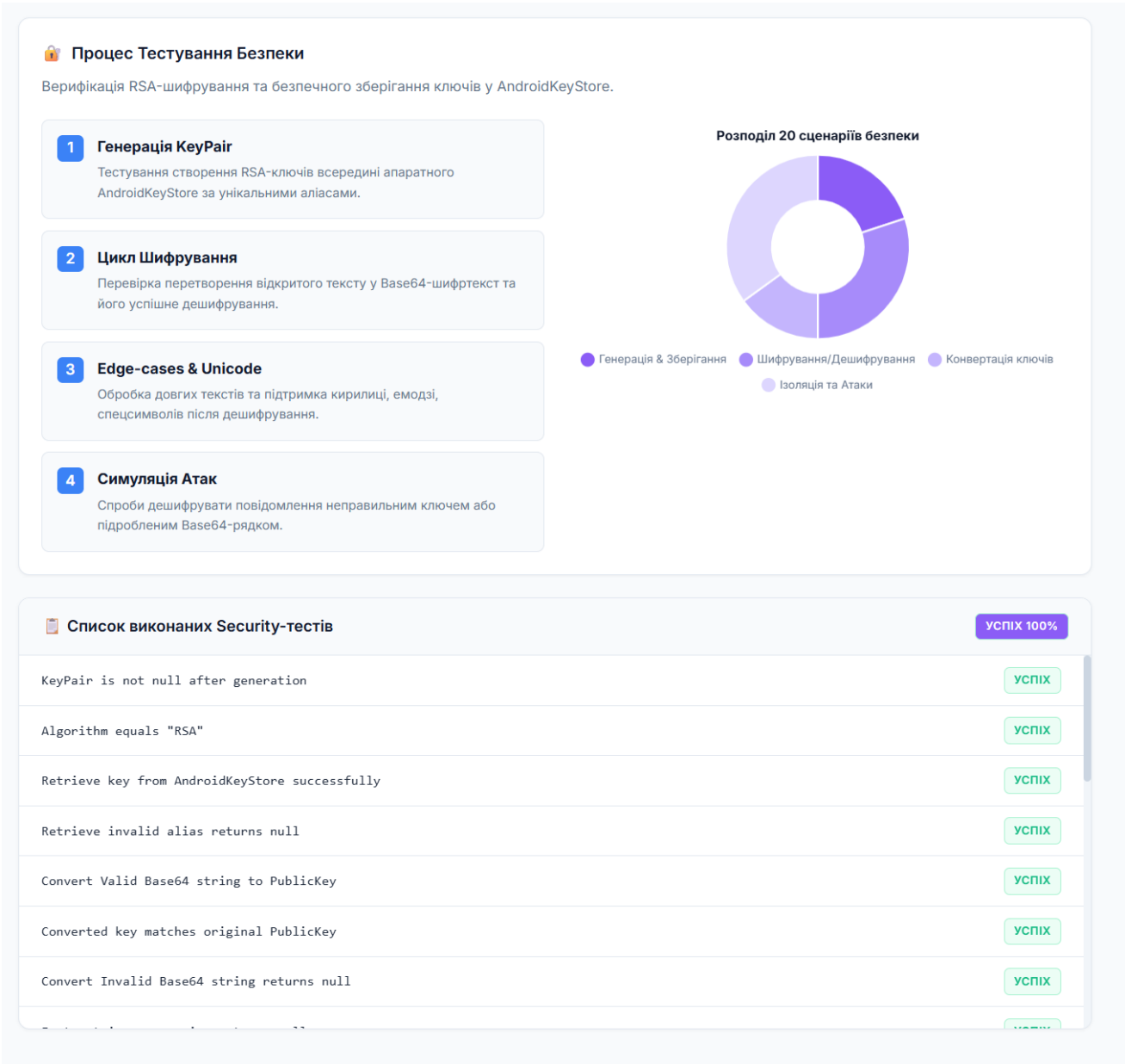


Рисунок А.6 – Результати тестування безпеки застосунку

ДОДАТОК Б

(обов'язковий)

ПРОГРАМНИЙ КОД ОСНОВНИХ МОДУЛІВ

Б.1 Код модулю шифрування

```
import android.content.Context
import android.security.keystore.KeyGenParameterSpec
import android.security.keystore.KeyProperties
import android.util.Log
import com.google.crypto.tink.Aead
import com.google.crypto.tink.HybridDecrypt
import com.google.crypto.tink.HybridEncrypt
import com.google.crypto.tink.KeyTemplates
import com.google.crypto.tink.KeysetHandle
import com.google.crypto.tink.aead.AeadKeyTemplates
import com.google.crypto.tink.integration.android.AndroidKeysetManager
import com.google.crypto.tink.subtle.Base64
import com.google.crypto.tink.subtle.Ed25519Sign
import java.io.ByteArrayOutputStream
import java.security.KeyFactory
import java.security.KeyPair
import java.security.KeyPairGenerator
import java.security.KeyStore
import java.security.PrivateKey
import java.security.PublicKey
import java.security.spec.X509EncodedKeySpec
import javax.crypto.Cipher
import javax.inject.Inject

class Encryption @Inject constructor(
    private val context: Context
) : I_Encryprion {

   override fun generateAndSaveKeyPair(alias: String): KeyPair? {
        return try {
            val keyPairGenerator = KeyPairGenerator.getInstance(
                KeyProperties.KEY_ALGORITHM_RSA, "AndroidKeyStore"
            )

            val keyGenParameterSpec = KeyGenParameterSpec.Builder(
                alias,
                KeyProperties.PURPOSE_ENCRYPT or KeyProperties.PURPOSE_DECRYPT
```

```
            ).apply {
                setKeySize(2048)
                setDigests(KeyProperties.DIGEST_SHA256, KeyProperties.DIGEST_SHA512)
                setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_RSA_PKCS1)
                setUserAuthenticationRequired(false)
            }.build()

            keyPairGenerator.initialize(keyGenParameterSpec)
            val keyPair = keyPairGenerator.generateKeyPair()
            keyPair
        } catch (e: Exception) {
            e.printStackTrace()
            null
        }
    }

    override fun getKeyPair(alias: String): KeyPair? {
        return try {
            val keyStore = KeyStore.getInstance("AndroidKeyStore")
            keyStore.load(null)

            val privateKey = keyStore.getKey(alias, null) as? PrivateKey
            val publicKey = keyStore.getCertificate(alias)?.publicKey

            if (privateKey != null && publicKey != null) {
                KeyPair(publicKey, privateKey)
            } else {
                null
            }
        } catch (e: Exception) {
            e.printStackTrace()
            null
        }
    }


    override fun convertStringToPublicKey(publicKeyString: String): PublicKey? {
        return try {
            val keyBytes = Base64.decode(publicKeyString, Base64.DEFAULT)

            val keySpec = X509EncodedKeySpec(keyBytes)

            val keyFactory = KeyFactory.getInstance("RSA")

            keyFactory.generatePublic(keySpec)
```

```
        } catch (e: Exception) {
            e.printStackTrace()
            null
        }
    }


    override fun encryptMessage(message: String, publicKey: PublicKey): String {
        Log.d("MyLogPUBLIC", "encryptMessage $$message $publicKey ")
        val cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding")
        cipher.init(Cipher.ENCRYPT_MODE, publicKey)
        val encryptedBytes = cipher.doFinal(message.toByteArray())
        return Base64.encodeToString(encryptedBytes, Base64.DEFAULT)
    }


    override fun decryptMessage(encryptedMessage: ByteArray, privateKey: PrivateKey): String
{
        Log.d("MyLogPUBLIC", "decryptMessage $$encryptedMessage $privateKey ")
        val cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding")
        cipher.init(Cipher.DECRYPT_MODE, privateKey)
        return String(cipher.doFinal(encryptedMessage), Charsets.UTF_8)
    }
}
```

Б.2 Код Message View Module

```
import android.os.Bundle
import android.util.Log
import androidx.compose.runtime.mutableStateOf
import androidx.lifecycle.AbstractSavedStateViewModelFactory
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.SavedStateHandle
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.savedstate.SavedStateRegistryOwner
import com.dar_hav_projects.messenger.db.MessageEntity
import com.dar_hav_projects.messenger.db.MessagesRepository
import com.dar_hav_projects.messenger.di.AppComponent
import com.dar_hav_projects.messenger.domens.actions.I_NetworkActions
import com.dar_hav_projects.messenger.domens.models.Chat
import com.dar_hav_projects.messenger.domens.models.Message
import com.dar_hav_projects.messenger.encryption.I_Encryprion
import com.google.crypto.tink.subtle.Base64
```

```
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.flowOf
import kotlinx.coroutines.launch
import kotlinx.coroutines.withContext
import java.security.PrivateKey
import java.security.PublicKey
import javax.inject.Inject

class MessagesViewModel(
    appComponent: AppComponent
) : ViewModel() {

    @Inject
    lateinit var networkActions: I_NetworkActions

    @Inject
    lateinit var messagesRepo: MessagesRepository

    @Inject
    lateinit var encryption: I_Encryprion

    val chatId = mutableStateOf("")

    private val _messages = MutableLiveData<List<Message>>()
    val messages: LiveData<List<Message>> = _messages

    private val _publicKey = MutableLiveData<PublicKey>()

    var messagesDB: Flow<List<MessageEntity>> = flowOf()

    init {
        appComponent.inject(this)
    }

    fun setChatId(newChatId: String) {
        chatId.value = newChatId
        messagesDB = messagesRepo.getMessagesForChat(newChatId)
    }

    suspend fun createMessage(message: Message) {
        networkActions.createMessage(message)
    }

    suspend fun deleteMessage(message: Message) {
```

```
        networkActions.deleteMessage(message)
    }

    fun listenForMessages(chatId: String) {
        viewModelScope.launch {
            networkActions.listenForMessages(chatId).collect { newMessages ->
                _messages.postValue(newMessages)
            }
        }
    }


    fun encryptMessage(message: String): String? {


        Log.d("MyLog", "encryptMessage publicKey: ${_publicKey.value} ")
        return _publicKey.value?.let { encryption.encryptMessage(message, it) }
    }


    fun encryptMessageSender(message: String): String? {
        val publicKey = encryption.getKeyPair(networkActions.getAlias())?.public
        return publicKey?.let { encryption.encryptMessage(message, it) }
    }


    fun  decryptMessage(encryptedMessage: String): String? {
        val byteArray = Base64.decode(encryptedMessage, Base64.DEFAULT)
        val privateKey = encryption.getKeyPair(networkActions.getAlias())?.private
        return privateKey?.let { encryption.decryptMessage(byteArray, it) }
    }


    fun checkSender(userUID: String): Boolean {
        return networkActions.checkSender(userUID)
    }

    private fun getSender(): String{
        return networkActions.getAlias()
    }

    suspend fun createMessageDB(message: Message) {
        if(!networkActions.checkSender(message.senderId)){
            messagesRepo.insertMessage(
                MessageEntity(
```

```
                null,
                message.chatId,
                message.senderId,
                message.content,
                message.timestamp,
                message.isRead
            )
        )
    }
}

suspend fun createMessageDBSender(message: Message) {
        messagesRepo.insertMessage(
            MessageEntity(
               null,
                message.chatId,
                getSender(),
                message.content,
                message.timestamp,
                message.isRead
            )
        )
}

suspend fun deleteMessageByIdDB(message: MessageEntity){
    messagesRepo.deleteMessageByID(message)
}

suspend fun deleteMessagesForChat(chatId: String){
    messagesRepo.deleteMessagesForChat(chatId)
}


suspend fun getPublicKey(chatId: String) {
  networkActions.getPublicKey(chatId)
      .onSuccess {
          _publicKey.value = encryption.convertStringToPublicKey(it)
  }

}

companion object {
    fun provideFactory(
        appComponent: AppComponent,
        owner: SavedStateRegistryOwner,
```

```
            defaultArgs: Bundle? = null,
        ): AbstractSavedStateViewModelFactory =
            object : AbstractSavedStateViewModelFactory(owner, defaultArgs) {
                @Suppress("UNCHECKED_CAST")
                override fun <T : ViewModel> create(
                    key: String,
                    modelClass: Class<T>,
                    handle: SavedStateHandle
                ): T {
                    return MessagesViewModel(appComponent) as T
                }
            }
    }
}
```

Б.3 Код Firebase Repository

```
import android.content.Context
import android.net.Uri
import android.util.Log
import com.dar_hav_projects.messenger.domens.models.Chat
import com.dar_hav_projects.messenger.domens.models.Contact
import com.dar_hav_projects.messenger.domens.models.IsSignedEnum
import com.dar_hav_projects.messenger.domens.models.Message
import com.dar_hav_projects.messenger.domens.models.UserData
import com.google.crypto.tink.subtle.Base64
import com.google.firebase.Firebase
import com.google.firebase.auth.FirebaseAuthInvalidUserException
import com.google.firebase.auth.auth
import com.google.firebase.auth.userProfileChangeRequest
import com.google.firebase.firestore.SetOptions
import com.google.firebase.firestore.firestore
import com.google.firebase.firestore.toObjects
import com.google.firebase.storage.FirebaseStorage
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.channels.awaitClose
import kotlinx.coroutines.coroutineScope
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.callbackFlow
import kotlinx.coroutines.launch
import kotlinx.coroutines.suspendCancellableCoroutine
import kotlinx.coroutines.tasks.await
import kotlinx.coroutines.withContext
import java.security.PublicKey
```

```
import javax.inject.Inject
import kotlin.coroutines.resume
import kotlin.coroutines.suspendCoroutine


class FirebaseRepository@Inject constructor(
    private val context: Context
): I_NetworkActions {

    private val auth = Firebase.auth
    private val firestore = Firebase.firestore


    override suspend fun signUp(email: String, pass: String): Result<Boolean> =
        suspendCoroutine { res ->
            CoroutineScope(Dispatchers.Default).launch {
                auth.createUserWithEmailAndPassword(email, pass)
                    .addOnCompleteListener { task ->
                        if (task.isSuccessful) {
                            res.resume(Result.success(true))
                        } else {
                            res.resume(
                                Result.failure(
                                    task.exception ?: Throwable("Sign-up failed")
                                )
                            )
                        }
                    }
            }
        }

    override suspend fun verifyEmail(): Result<Boolean> =
        suspendCoroutine { res ->
            CoroutineScope(Dispatchers.Default).launch {
                val user = auth.currentUser
                user?.let {
                    it.sendEmailVerification()
                        .addOnCompleteListener { verifyTask ->
                            if (verifyTask.isSuccessful) {
                                CoroutineScope(Dispatchers.Default).launch {
                                    it.reload().addOnCompleteListener { reloadTask ->
                                        if (reloadTask.isSuccessful && it.isEmailVerified)
{
                                            res.resume(Result.success(true))
                                        } else {
```

```
                                        res.resume(Result.failure(Throwable("Email not
verified yet")))
                                    }
                                }
                            }
                        } else {
                            res.resume(
                                Result.failure(
                                    verifyTask.exception
                                        ?: Throwable("Error sending verification
email")
                                )
                            )
                        }
                    }
                }
            }
        }


    override suspend fun signIn(email: String, pass: String): Result<IsSignedEnum> =
        suspendCoroutine { res ->
            auth.signInWithEmailAndPassword(email, pass)
                .addOnCompleteListener { task ->
                    if (task.isSuccessful) {
                        val user = auth.currentUser
                        if (user != null) {
                            if (user.isEmailVerified) {
                                if (user.displayName.isNullOrEmpty()) {

res.resume(Result.success(IsSignedEnum.DoNotHaveNickname))
                                } else {
                                    res.resume(Result.success(IsSignedEnum.IsSigned))
                                }
                            } else {
                                res.resume(Result.success(IsSignedEnum.DoNotVerifyEmail))
                            }
                        } else {
                            res.resume(Result.success(IsSignedEnum.IsNotSigned))
                        }
                    } else {
                        res.resume(
                            Result.failure(
                                task.exception ?: Throwable("An error occurred, please try
again")
```

```
                        )
                    )
                }
            }
        }


override suspend fun isSignedAsync(): IsSignedEnum = withContext(Dispatchers.IO) {
    val user = auth.currentUser
    var isSigned: IsSignedEnum
    try {
        if (user != null) {
            user.reload().await()
            if (user.isEmailVerified) {
                isSigned = if (!user.displayName.isNullOrEmpty()) {
                    IsSignedEnum.IsSigned
                } else {
                    IsSignedEnum.DoNotHaveNickname
                }
            } else {
                isSigned = IsSignedEnum.DoNotVerifyEmail
            }
        } else {
            isSigned = IsSignedEnum.IsNotSigned
        }
    } catch (e: FirebaseAuthInvalidUserException) {
        Log.e("MyLog", "User not found or deleted", e)
        auth.signOut()
        isSigned = IsSignedEnum.IsNotSigned
    } catch (e: Exception) {
        Log.e("MyLog", "An unexpected error occurred", e)
        isSigned = IsSignedEnum.IsNotSigned
    }

    isSigned
}

override suspend fun saveUserData(
    nickname: String,
    name: String,
    surname: String,
    imageUri: Uri?,
    publicKey: PublicKey?
): Result<Any> = suspendCoroutine { continuation ->
    val userId = auth.currentUser?.uid
```

```
if (userId == null) {
    continuation.resume(Result.failure(Throwable("User not authenticated")))
    return@suspendCoroutine
}

val storage = FirebaseStorage.getInstance()
val storageRef = storage.reference
val profileImageRef = storageRef.child("profileImages/$userId.jpg")

val publicKeyString = publicKey?.encoded?.let {
    Base64.encodeToString(it, Base64.DEFAULT)
} ?: ""

val saveUserDataToFirestore: (String) -> Unit = { downloadUrl ->
    firestore.collection(COLLECTION_USERDATA)
        .document(userId)
        .set(
            hashMapOf(
                COLLECTION_USERDATA_USER_UID to userId,
                COLLECTION_USERDATA_NICKNAME to nickname,
                COLLECTION_USERDATA_NAME to name,
                COLLECTION_USERDATA_SURNAME to surname,
                COLLECTION_USERDATA_URL to downloadUrl,
                COLLECTION_USERDATA_PUBLIC_KEY to publicKeyString
            )
        )
        .addOnSuccessListener {
            val profileUpdates = userProfileChangeRequest {
                displayName = nickname
            }
            auth.currentUser?.updateProfile(profileUpdates)
                ?.addOnSuccessListener {
                    continuation.resume(Result.success(true))
                }
                ?.addOnFailureListener { ex ->
                    Log.e("FirebaseAuth", "Error updating display name", ex)
                    continuation.resume(Result.failure(ex))
                }
        }
        .addOnFailureListener { ex ->
            Log.e("Firestore", "Error saving user data", ex)
            continuation.resume(Result.failure(ex))
        }
}
```

```
    if (imageUri != null) {
        val uploadTask = profileImageRef.putFile(imageUri)
        uploadTask.addOnSuccessListener {
            profileImageRef.downloadUrl
                .addOnSuccessListener { downloadUrl ->
                    saveUserDataToFirestore(downloadUrl.toString())
                }
                .addOnFailureListener { e ->
                    Log.e("FirebaseStorage", "Error getting download URL", e)
                    continuation.resume(Result.failure(e))
                }
        }.addOnFailureListener { e ->
            Log.e("Upload", "File upload failed", e)
            continuation.resume(Result.failure(e))
        }
    } else {
        saveUserDataToFirestore("")
    }
}

override suspend fun fetchUserData(): Result<UserData?> {
    val userId = auth.currentUser?.uid
    return try {
        val document = userId?.let {
            firestore.collection(COLLECTION_USERDATA)
                .document(it)
                .get()
                .await()
        }

        val userData = document?.toObject(UserData::class.java)
        Result.success(userData)
    } catch (ex: Exception) {
        Result.failure(ex)
    }
}


override suspend fun fetchUserDataByID(userId: String): Result<UserData?> {
    return try {
        val document = userId?.let {
            firestore.collection(COLLECTION_USERDATA)
                .document(it)
                .get()
```

```
                .await()
            }

            val userData = document?.toObject(UserData::class.java)
            Result.success(userData)
        } catch (ex: Exception) {
            Result.failure(ex)
        }
    }

    override suspend fun fetchAccountData(): Result<UserData?> {
        val userId = auth.currentUser?.uid
        return try {
            val document = userId?.let {
                firestore.collection(COLLECTION_USERDATA)
                    .document(it)
                    .get()
                    .await()
            }

            val userData = document?.toObject(UserData::class.java)
            Result.success(userData)
        } catch (ex: Exception) {
            Result.failure(ex)
        }
    }




    override suspend fun fetchChats(): Result<List<Chat>> = suspendCancellableCoroutine {
res ->
        val userUid = auth.currentUser?.uid.toString()

        val chats = mutableListOf<Chat>()
        val firestoreCollection = firestore.collection(COLLECTION_CHATS)

        firestoreCollection
            .whereEqualTo(COLLECTION_CHATS_MEMBER1, userUid)
            .get()
            .addOnSuccessListener { result ->
                chats.addAll(result.toObjects(Chat::class.java))

                firestoreCollection
```

```
                    .whereEqualTo(COLLECTION_CHATS_MEMBER2, userUid)
                    .get()
                    .addOnSuccessListener { result2 ->
                        chats.addAll(result2.toObjects(Chat::class.java))
                        res.resume(Result.success(chats))
                    }
                    .addOnFailureListener { ex ->
                        res.resume(Result.failure(ex))
                    }
            }
            .addOnFailureListener { ex ->
                Log.d("MyLog", "Failure in member1UId query: ${ex.message}")
                res.resume(Result.failure(ex))
            }
    }

    override suspend fun createChat(member2: String): Result<Boolean> = suspendCoroutine {
res ->
        val userId = auth.currentUser?.uid.toString()

        val newChatRef = firestore.collection(COLLECTION_CHATS).document()
        val chatId = newChatRef.id

        val chatData = hashMapOf(
            COLLECTION_CHAT_ID to chatId,
            COLLECTION_CHATS_MEMBER1 to userId,
            COLLECTION_CHATS_MEMBER2 to member2,
            COLLECTION_CHATS_LAST_MESSAGE to "No messages",
            COLLECTION_CHATS_LAST_MESSAGE_TIMESTAMP to 0
        )

        firestore
            .collection(COLLECTION_CHATS)
            .add(chatData)
            .addOnCompleteListener { result ->
                if (result.isSuccessful) {
                    res.resume(Result.success(true))
                } else {
                    res.resume(Result.failure(Throwable("Can't connect to server")))
                }
            }
            .addOnFailureListener { ex ->
                res.resume(Result.failure(ex))
            }
    }
```

```
    override suspend fun getPublicKey(chatID: String): Result<String> = suspendCoroutine {
res ->
        val userId = auth.currentUser?.uid.toString()
        val firestoreCollection = firestore.collection(COLLECTION_CHATS)

        firestoreCollection
            .whereEqualTo(COLLECTION_CHAT_ID, chatID)
            .get()
            .addOnSuccessListener { result ->
                val item = result.toObjects(Chat::class.java).lastOrNull()

                if (item != null) {

                    if (userId == item.member1UId) {

                        CoroutineScope(Dispatchers.IO).launch {
                            fetchUserDataByID(item.member2UId)
                                .onSuccess { user ->
                                    if (user != null) {
                                        res.resume(Result.success(user.publicKey))
                                    }
                                }.onFailure {
                                    res.resume(Result.success(""))
                                }
                        }
                    } else {

                        CoroutineScope(Dispatchers.IO).launch {
                            fetchUserDataByID(item.member1UId)
                                .onSuccess { user ->
                                    if (user != null) {
                                        res.resume(Result.success(user.publicKey))
                                    }
                                }.onFailure {
                                    res.resume(Result.success(""))
                                }
                        }
                    }
                }

            } .addOnFailureListener { ex ->
                res.resume(Result.failure(ex))
            }
```

```
    }


    override suspend fun getChatName(item: Chat): Result<String> = suspendCoroutine { res -
>
        val userId = auth.currentUser?.uid.toString()

        if (userId == item.member1UId) {
            CoroutineScope(Dispatchers.IO).launch {
               fetchUserDataByID(item.member2UId)
                   .onSuccess { user ->
                   if (user != null) {
                       res.resume(Result.success(user.name))
                   }
                }.onFailure {
                       res.resume(Result.success(""))
                   }
            }
        }else{
            CoroutineScope(Dispatchers.IO).launch {
                fetchUserDataByID(item.member1UId)
                    .onSuccess { user ->
                        if (user != null) {
                            res.resume(Result.success(user.name))
                        }
                    }.onFailure {
                        res.resume(Result.success(""))
                    }
            }
        }

    }

    override    suspend    fun    deleteChat(chatId:    String):    Result<Boolean>    =
suspendCancellableCoroutine { continuation ->
        firestore.collection(COLLECTION_CHATS)
            .whereEqualTo(COLLECTION_CHAT_ID, chatId)
            .get()
            .addOnSuccessListener { querySnapshot ->
                if (!querySnapshot.isEmpty) {
                    val batch = firestore.batch()
                    for (document in querySnapshot.documents) {
                        batch.delete(document.reference)
                    }
```

```
                    batch.commit()
                        .addOnSuccessListener {
                            continuation.resume(Result.success(true), null)
                        }
                        .addOnFailureListener { ex ->
                            continuation.resume(Result.failure(ex), null)
                        }
                } else {
                    continuation.resume(Result.failure(NoSuchElementException("No message found with the given ID")), null)
                }
            }
            .addOnFailureListener { ex ->
                continuation.resume(Result.failure(ex), null)
            }
    }

    override suspend fun listenForMessages(chatID: String): Flow<List<Message>> = callbackFlow {
        val listenerRegistration = firestore.collection(COLLECTION_MESSAGES)
            .whereEqualTo(COLLECTION_MESSAGES_CHAT_ID, chatID)
            .addSnapshotListener { snapshot, error ->
                if (error != null) {
                    Log.e("FirestoreError", "Listening failed: ", error)
                    close(error)
                    return@addSnapshotListener
                }

                snapshot?.let {
                    val items = it.documents.mapNotNull { doc -> doc.toObject(Message::class.java) }
                    Log.d("MyLog", "listener $items")
                    trySend(items).isSuccess
                }
            }

        awaitClose { listenerRegistration.remove() }
    }

    override suspend fun createMessage(message: Message): Result<Boolean> = suspendCancellableCoroutine { continuation ->
        val userId = auth.currentUser?.uid.toString()

        val newMessageRef = firestore.collection(COLLECTION_MESSAGES).document()
        val messageId = newMessageRef.id
```

```
        val chatData = hashMapOf(
            COLLECTION_MESSAGE_ID to messageId,
            COLLECTION_MESSAGES_CHAT_ID to message.chatId,
            COLLECTION_MESSAGES_SENDER_ID to userId,
            COLLECTION_MESSAGES_TIMESTAMP to message.timestamp,
            COLLECTION_MESSAGES_CONTENT to message.content,
            COLLECTION_MESSAGES_IS_READ to message.isRead
        )

        firestore.collection(COLLECTION_MESSAGES)
            .add(chatData)
            .addOnCompleteListener { result ->
                if (continuation.isActive) {
                    if (result.isSuccessful) {
                        continuation.resume(Result.success(true), null)
                    } else {
                        continuation.resume(Result.failure(Throwable("Can't connect to
server")), null)
                    }
                }
            }
            .addOnFailureListener { ex ->
                if (continuation.isActive) {
                    continuation.resume(Result.failure(ex), null)
                }
            }
    }


    override suspend fun deleteMessage(message: Message): Result<Boolean> =
suspendCancellableCoroutine { continuation ->
        val userId = auth.currentUser?.uid.toString()

       if(message.senderId != userId){
           firestore.collection(COLLECTION_MESSAGES)
               .whereEqualTo(COLLECTION_MESSAGE_ID, message.messageId)
               .get()
               .addOnSuccessListener { querySnapshot ->
                   if (!querySnapshot.isEmpty) {
                       val batch = firestore.batch()
                       for (document in querySnapshot.documents) {
                           batch.delete(document.reference)
                       }
```

```
                        batch.commit()
                            .addOnSuccessListener {
                                continuation.resume(Result.success(true), null)
                            }
                            .addOnFailureListener { ex ->
                                continuation.resume(Result.failure(ex), null)
                            }
                    } else {
                        continuation.resume(Result.failure(NoSuchElementException("No
message found with the given ID")), null)
                    }
                }
                .addOnFailureListener { ex ->
                    continuation.resume(Result.failure(ex), null)
                }
        }

    }

    override fun checkSender(userUID: String): Boolean {
        val userId = auth.currentUser?.uid.toString()
        return userId == userUID
    }

    override fun getAlias(): String {
        return  auth.currentUser?.uid.toString()
    }


    override suspend fun fetchContacts(): Result<List<Contact>> = suspendCoroutine { res -
>
        val userUid = auth.currentUser?.uid.toString()

        firestore.collection(COLLECTION_CONTACTS)
            .whereEqualTo(COLLECTION_CONTACTS_CONTACT_WITH, userUid)
            .get()
            .addOnSuccessListener { result ->
                val items = result.toObjects<Contact>()
                res.resume(Result.success(items))
                Log.d("MyLog", "contacts 1: ${items}")
            }
            .addOnFailureListener { ex ->
                Log.d("MyLog", "contacts 1: failure $ex")
                res.resume(Result.success(emptyList()))
            }
```

```
    }

    override suspend fun addContact(item: UserData): Result<Boolean> = suspendCoroutine {
res ->
        val userId = auth.currentUser?.uid.toString()
        firestore
            .collection(COLLECTION_CONTACTS)
            .document()
            .set(
                hashMapOf(
                    COLLECTION_CONTACTS_CONTACT_WITH to userId,
                    COLLECTION_CONTACTS_NAME to item.name,
                    COLLECTION_CONTACTS_SURNAME to item.surname,
                    COLLECTION_CONTACTS_NICKNAME to item.nickname,
                    COLLECTION_CONTACTS_PROFILE_PHOTO_URL to item.url,
                    COLLECTION_CONTACTS_PROFILE_UID to item.userUID
                ),
                SetOptions.merge()
            )
            .addOnCompleteListener { result ->
                if(result.isSuccessful){
                    res.resume(Result.success(true))
                } else {
                    res.resume(Result.failure(Throwable("Can`t connect to server")))
                }
            }
            .addOnFailureListener { ex ->
                res.resume(Result.failure(ex))
            }
    }

    override suspend fun addContactForFriend(item: UserData): Result<Boolean> {
        val userId = auth.currentUser?.uid

        if (userId == null) {
            return Result.failure(Throwable("User not authenticated"))
        }

        val userdataResult = fetchUserDataByID(userId)

        return if (userdataResult.isSuccess && userdataResult.getOrNull() != null) {
            val userdata = userdataResult.getOrNull()
            if (userdata != null) {
                firestore
                    .collection(COLLECTION_CONTACTS)
```

```
                    .document()
                    .set(
                        hashMapOf(
                            COLLECTION_CONTACTS_CONTACT_WITH to item.userUID,
                            COLLECTION_CONTACTS_NAME to userdata.name,
                            COLLECTION_CONTACTS_SURNAME to userdata.surname,
                            COLLECTION_CONTACTS_NICKNAME to userdata.nickname,
                            COLLECTION_CONTACTS_PROFILE_PHOTO_URL to userdata.url,
                            COLLECTION_CONTACTS_PROFILE_UID to userdata.userUID
                        ),
                        SetOptions.merge()
                    )
                    .await()
            }

            Result.success(true)
        } else {
            Result.failure(Throwable("User data not found or fetchUserDataByID failed"))
        }
    }


    override suspend fun searchContact(nickname: String): Result<List<UserData>> =
suspendCoroutine { res ->
        val userId = auth.currentUser?.uid.toString()
        Log.d("MyLog", "userId 1: ${userId}")
        firestore.collection(COLLECTION_USERDATA)
            .whereEqualTo(COLLECTION_USERDATA_NICKNAME, nickname)
            .get()
            .addOnSuccessListener { result ->
                val items = result.toObjects<UserData>().filter { it.userUID != userId }

                res.resume(Result.success(items))
                Log.d("MyLog", "contacts 1: ${items}")
            }
            .addOnFailureListener { ex ->
                Log.d("MyLog", "contacts 1: failure $ex")
                res.resume(Result.success(emptyList()))
            }
    }

    override suspend fun deleteContact(contactId: String): Result<Boolean> =
suspendCancellableCoroutine { continuation ->

        firestore.collection(COLLECTION_CONTACTS)
```

```
            .whereEqualTo(COLLECTION_CONTACTS_PROFILE_UID, contactId)
            .get()
            .addOnSuccessListener { querySnapshot ->
                if (!querySnapshot.isEmpty) {
                    val batch = firestore.batch()
                    for (document in querySnapshot.documents) {
                        batch.delete(document.reference)
                    }

                    batch.commit()
                        .addOnSuccessListener {
                            continuation.resume(Result.success(true), null)
                        }
                        .addOnFailureListener { ex ->
                            continuation.resume(Result.failure(ex), null)
                        }
                } else {
                    continuation.resume(Result.failure(NoSuchElementException("No   message
found with the given ID")), null)
                }
            }
            .addOnFailureListener { ex ->
                continuation.resume(Result.failure(ex), null)
            }
    }


    companion object{

        private const val COLLECTION_USERDATA = "UserData"
        private const val COLLECTION_USERDATA_USER_UID = "userUID"
        private const val COLLECTION_USERDATA_NICKNAME = "nickname"
        private const val COLLECTION_USERDATA_NAME = "name"
        private const val COLLECTION_USERDATA_SURNAME = "surname"
        private const val COLLECTION_USERDATA_URL = "url"
        private const val COLLECTION_USERDATA_PUBLIC_KEY= "publicKey"

        private const val COLLECTION_CHATS = "Chats"
        private const val COLLECTION_CHAT_NAME = "chatName"
        private const val COLLECTION_CHAT_ID = "chatId"
        private const val COLLECTION_CHATS_LAST_MESSAGE = "lastMessage"
        private const val COLLECTION_CHATS_LAST_MESSAGE_TIMESTAMP = "lastMessageTimestamp"
        private const val COLLECTION_CHATS_MEMBER1 = "member1UId"
        private const val COLLECTION_CHATS_MEMBER2 = "member2UId"
```

```
        private const val COLLECTION_CONTACTS = "Contacts"
        private const val COLLECTION_CONTACTS_CONTACT_WITH = "contactWith"
        private const val COLLECTION_CONTACTS_NAME = "name"
        private const val COLLECTION_CONTACTS_SURNAME = "surname"
        private const val COLLECTION_CONTACTS_NICKNAME = "nickname"
        private const val COLLECTION_CONTACTS_PROFILE_PHOTO_URL = "profileImageUrl"
        private const val COLLECTION_CONTACTS_PROFILE_UID = "userId"

        private const val COLLECTION_MESSAGES = "Messages"
        private const val COLLECTION_MESSAGE_ID = "messageId"
        private const val COLLECTION_MESSAGES_CHAT_ID = "chatId"
        private const val COLLECTION_MESSAGES_SENDER_ID = "senderId"
        private const val COLLECTION_MESSAGES_TIMESTAMP = "timestamp"
        private const val COLLECTION_MESSAGES_CONTENT = "content"
        private const val COLLECTION_MESSAGES_IS_READ = "isRead"
    }


}
```

Б.4 Код Room Repository

```
import com.dar_hav_projects.messenger.domens.models.Message
import kotlinx.coroutines.flow.Flow

class MessagesRepoImpl(
    private val dao: MessagesDAO
): MessagesRepository {
    override suspend fun insertMessage(message: MessageEntity) {
        dao.insertMessage(message)
    }

    override  fun getMessagesForChat(chatId: String): Flow<List<MessageEntity>> {
        return dao.getMessagesForChat(chatId)
    }

    override suspend fun deleteMessageByID(message: MessageEntity) {
        return dao.deleteMessageByID(message)
    }

    override suspend fun deleteMessagesForChat(chatId: String) {
        return dao.deleteMessagesForChat(chatId)
    }
}
```

Б.5 Код Message Data Class

```
data class Message(
    val messageId: String = "",
    val chatId: String = "",
    val senderId: String = "",
    val timestamp: Long = 0L,
    val content: String = "",
    val isRead: Boolean = false,
)
```

Б.6 Код Dependency Injection

```
import com.dar_hav_projects.messenger.MainActivity
import com.dar_hav_projects.messenger.domens.actions.FirebaseRepository
import com.dar_hav_projects.messenger.encryption.Encryption
import com.dar_hav_projects.messenger.view_models.AccountViewModel
import com.dar_hav_projects.messenger.view_models.ChatsViewModel
import com.dar_hav_projects.messenger.view_models.ContactsViewModel
import com.dar_hav_projects.messenger.view_models.MessagesViewModel
import com.dar_hav_projects.messenger.view_models.SignViewModel
import dagger.Component
import javax.inject.Singleton

@Singleton
@Component(modules = [
    NetworkActionsModule::class,
    ContextModule::class,
    EncryptionModule::class,
    MessagesRepositoryModule::class
])
interface AppComponent {
    fun inject(signViewModel: SignViewModel)
    fun inject(chatsViewModel: ChatsViewModel)
    fun inject(accountViewModel: AccountViewModel)
    fun inject(encryptionModule: Encryption)
    fun inject(messagesViewModel: MessagesViewModel)
    fun inject(contactsViewModel: ContactsViewModel)
    fun inject(mainActivity: MainActivity)
    fun inject(firebaseRepository: FirebaseRepository)

}
```

ДОДАТОК В

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

Рисунок В.1 – Титульний слайд

Рисунок В.2 – Слайд «Актуальність теми»

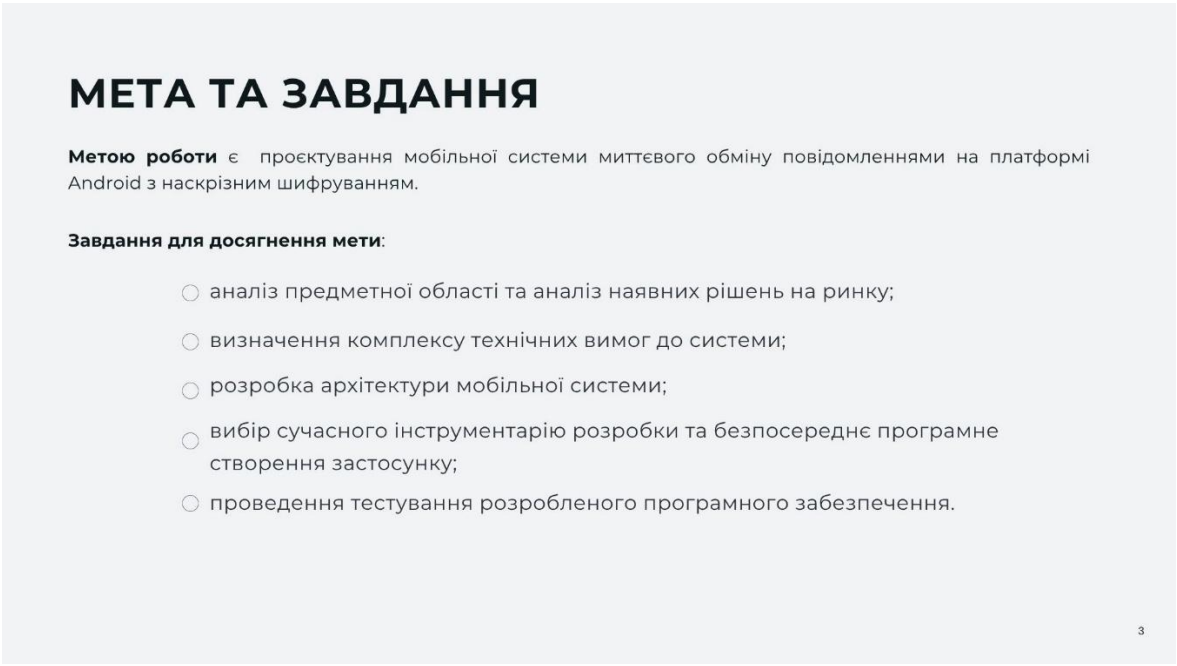


Рисунок В.3 – Слайд «Мета та завдання»

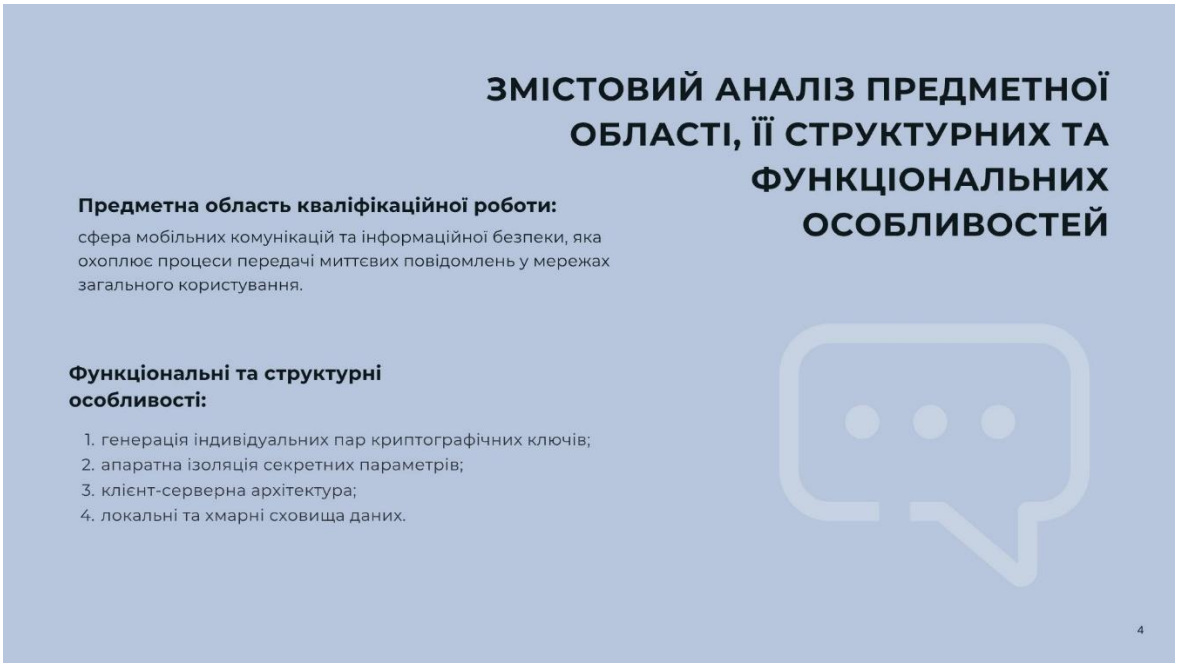


Рисунок В.4 – Слайд «Змістовний аналіз предметної області, її структурних та функціональних особливостей»

Рисунок В.5 – Слайд «Аналіз наявного програмно-технічного забезпечення»

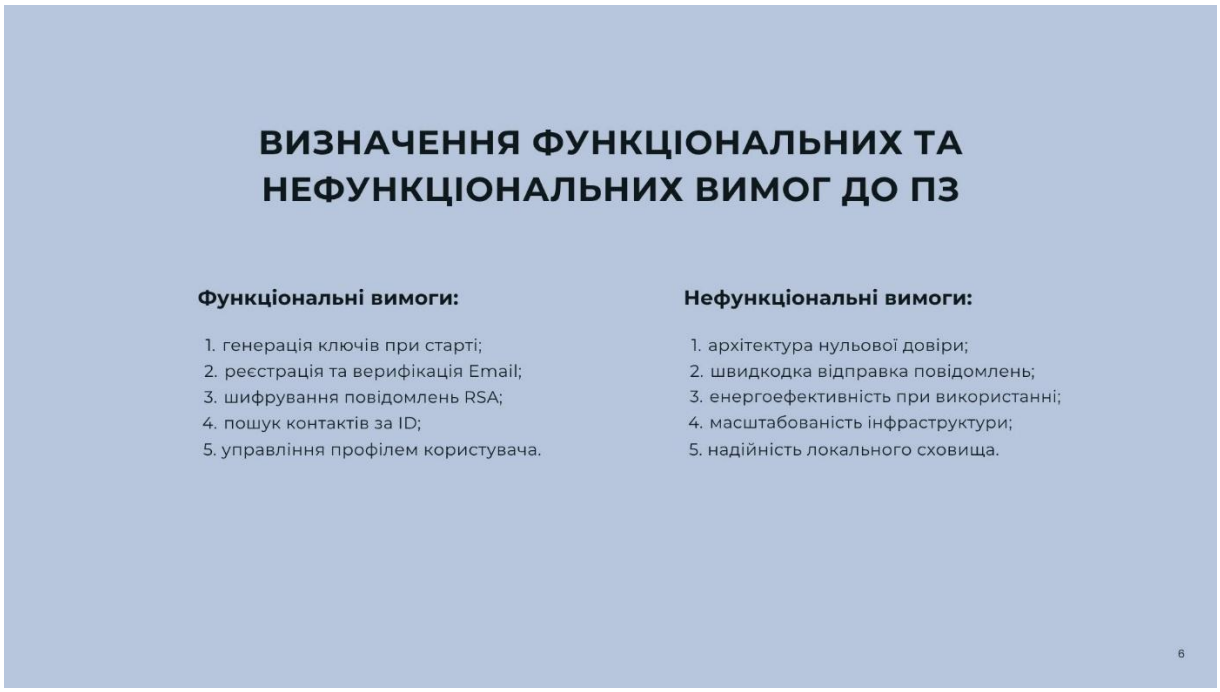


Рисунок В.6 – Слайд «Визначення функціональних та нефункціональних вимог до ПЗ»

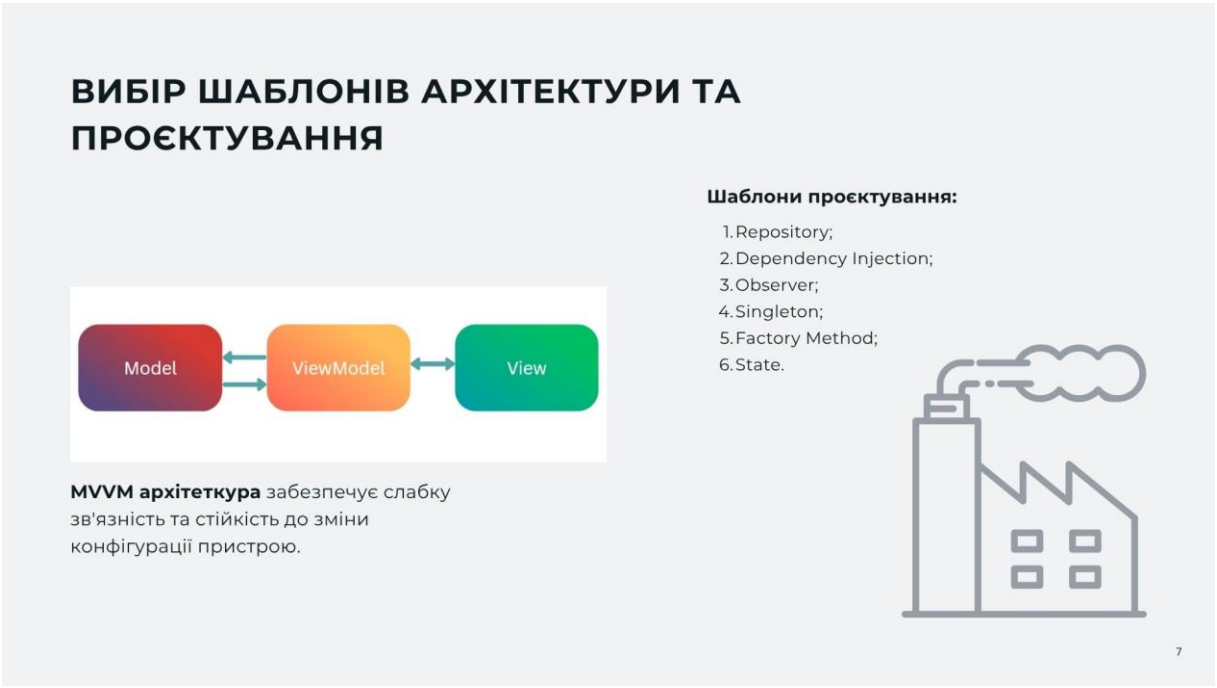


Рисунок В.7 – Слайд «Вибір шаблонів архітектури та проєктування»

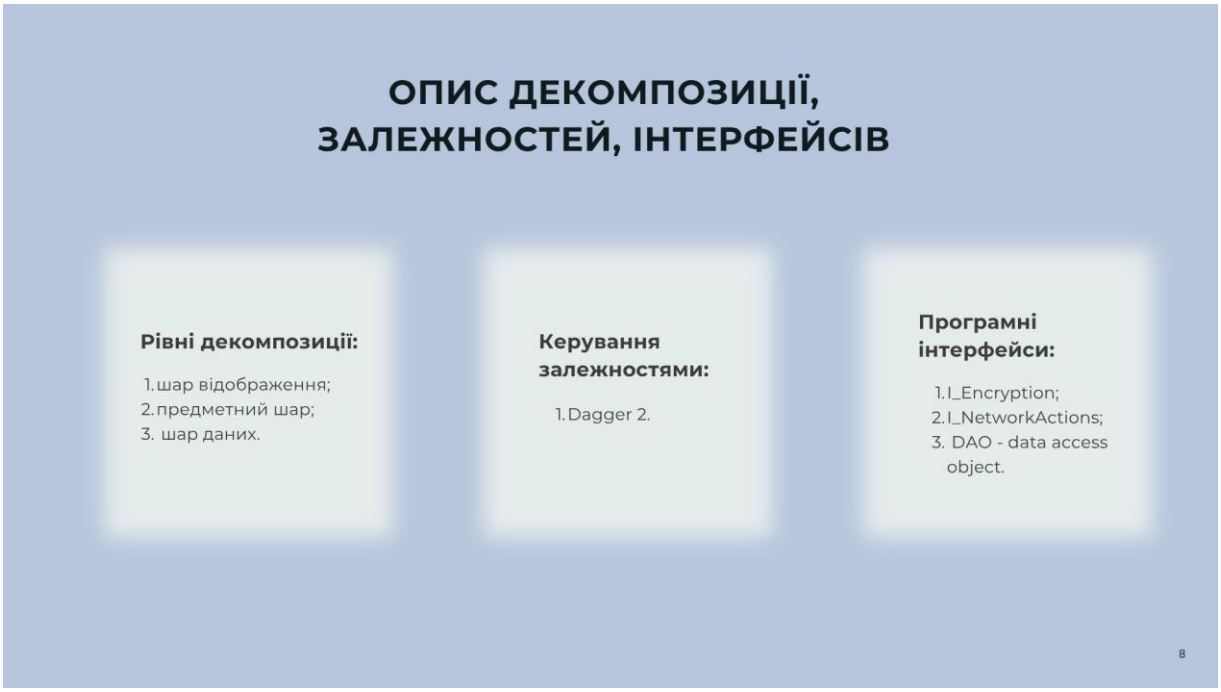


Рисунок В.8 – Слайд «Опис декомпозиції, залежностей, інтерфейсів»

Рисунок В.9 – Слайд «Проєктування модулів і даних»

Рисунок В.10 – Слайд «Аналіз та вибір технологій»

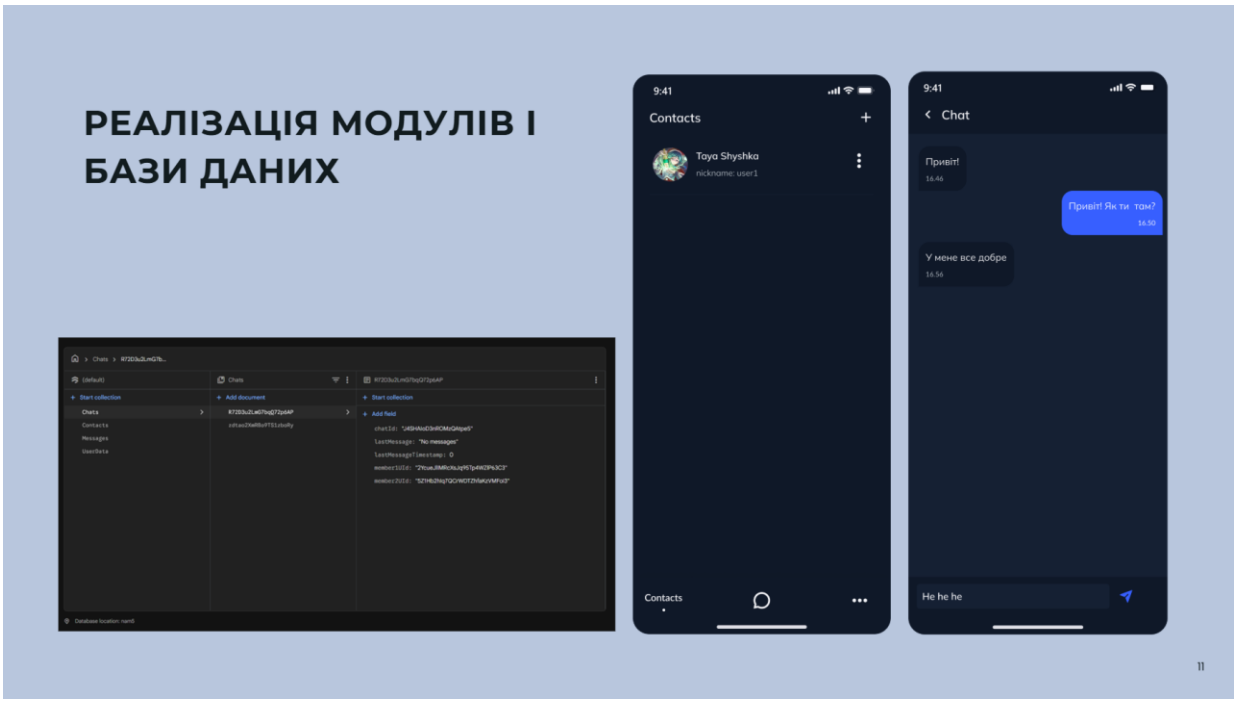


Рисунок В.11 – Слайд «Реалізація модулів і бази даних»

Рисунок В.12 – Слайд «Тестування ПЗ»

Рисунок В.13 – Слайд «Висновки»

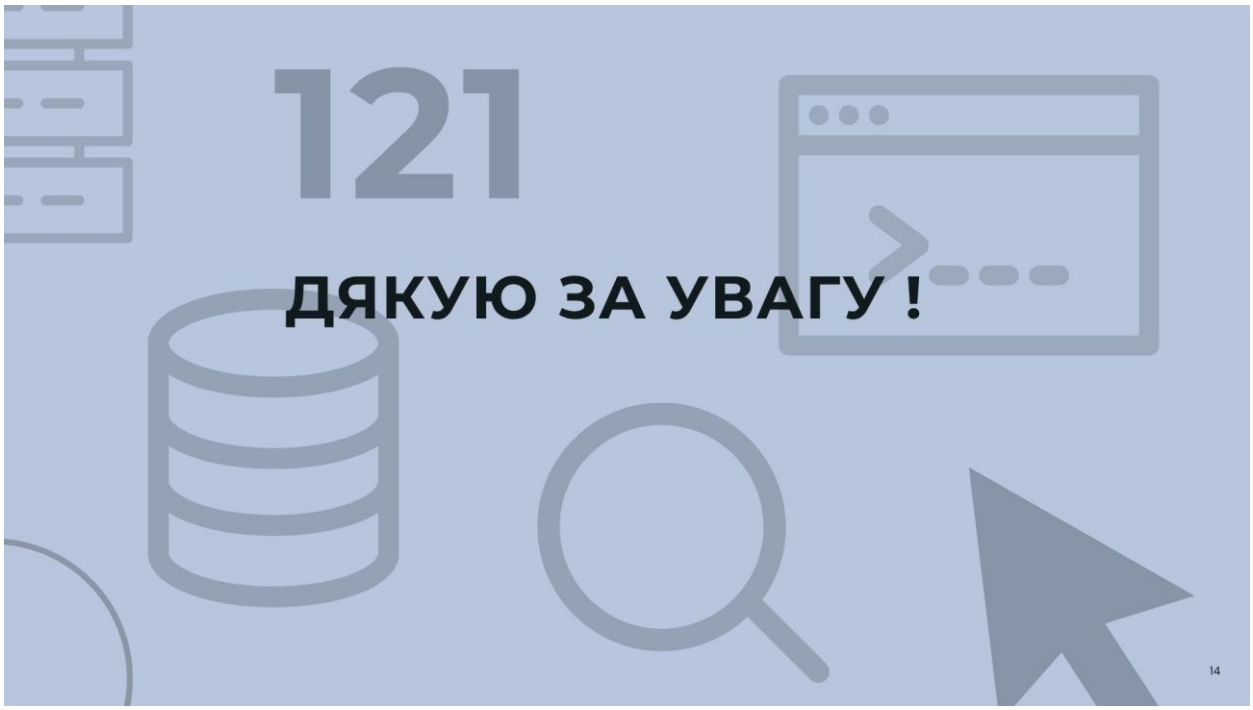
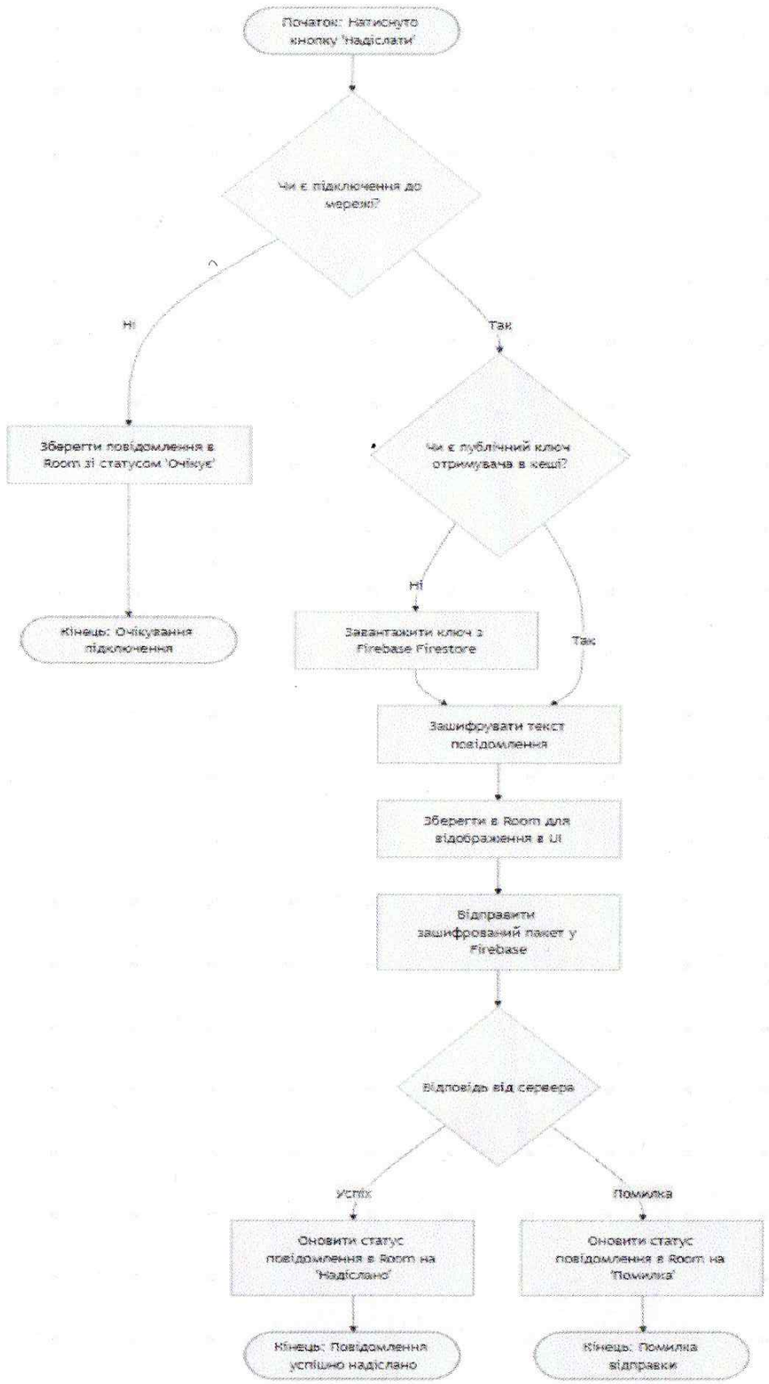
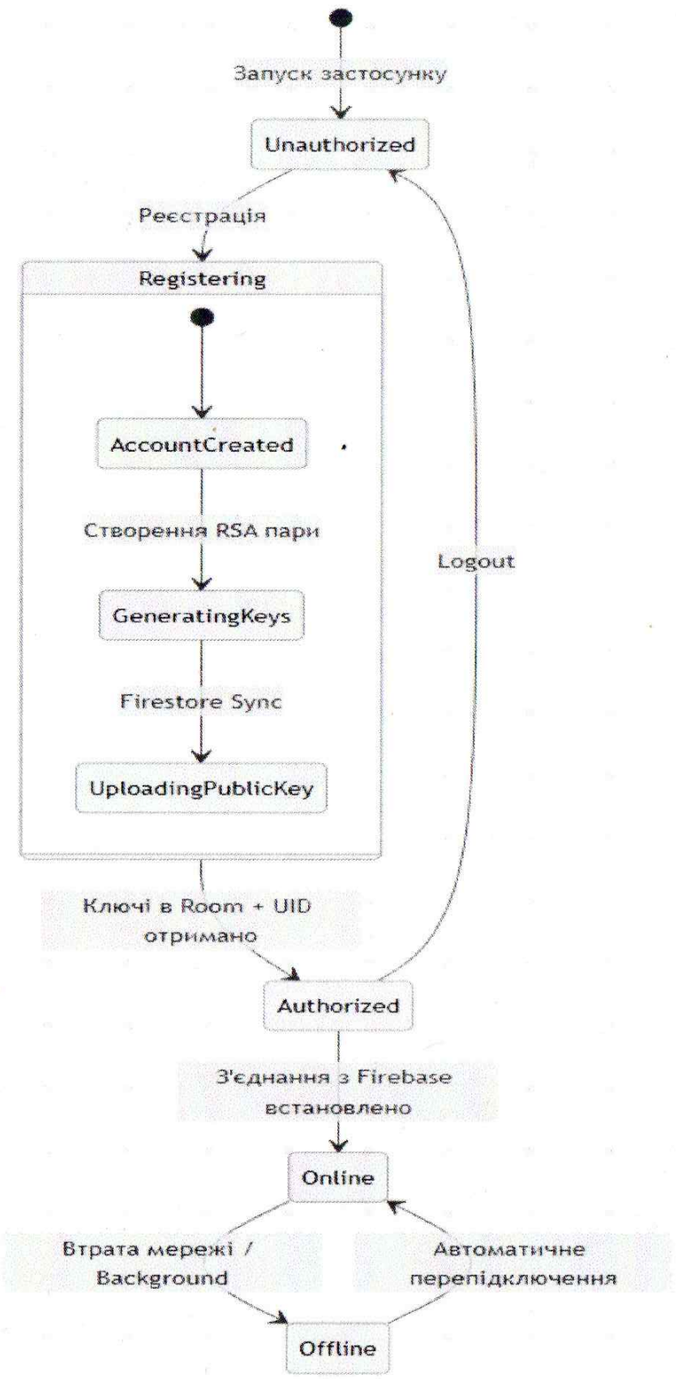


Рисунок В.14 – Слайд «Дякую за увагу»

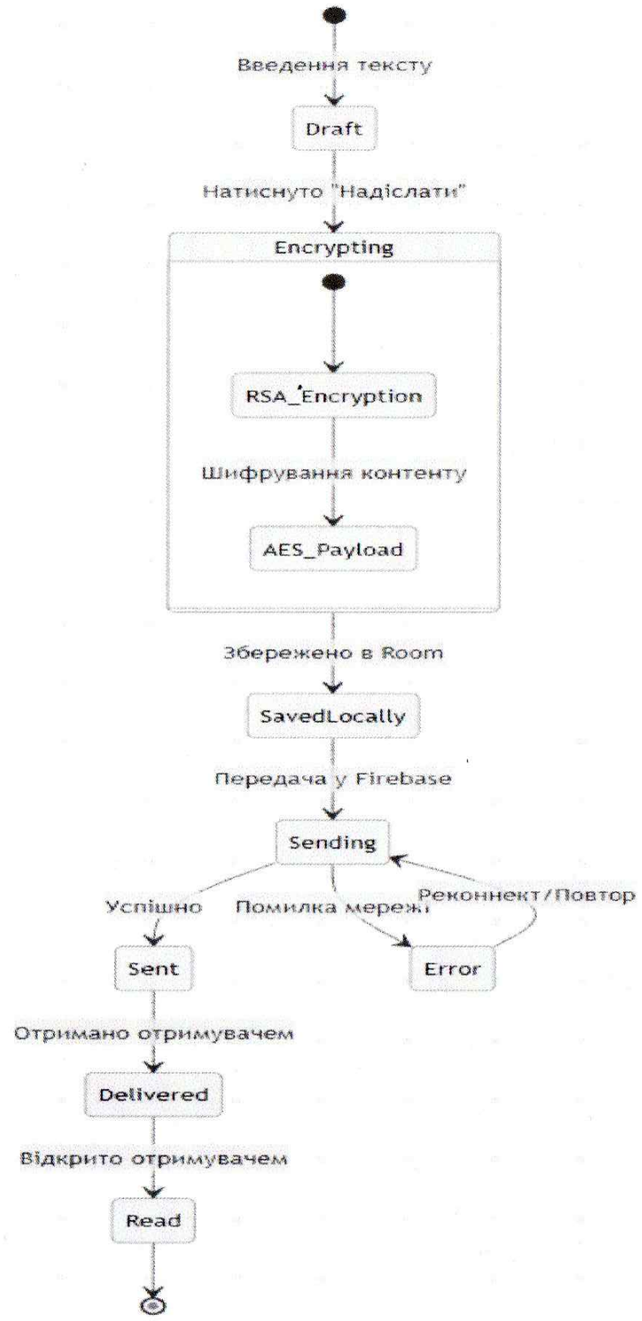
ГРАФІЧНА ЧАСТИНА



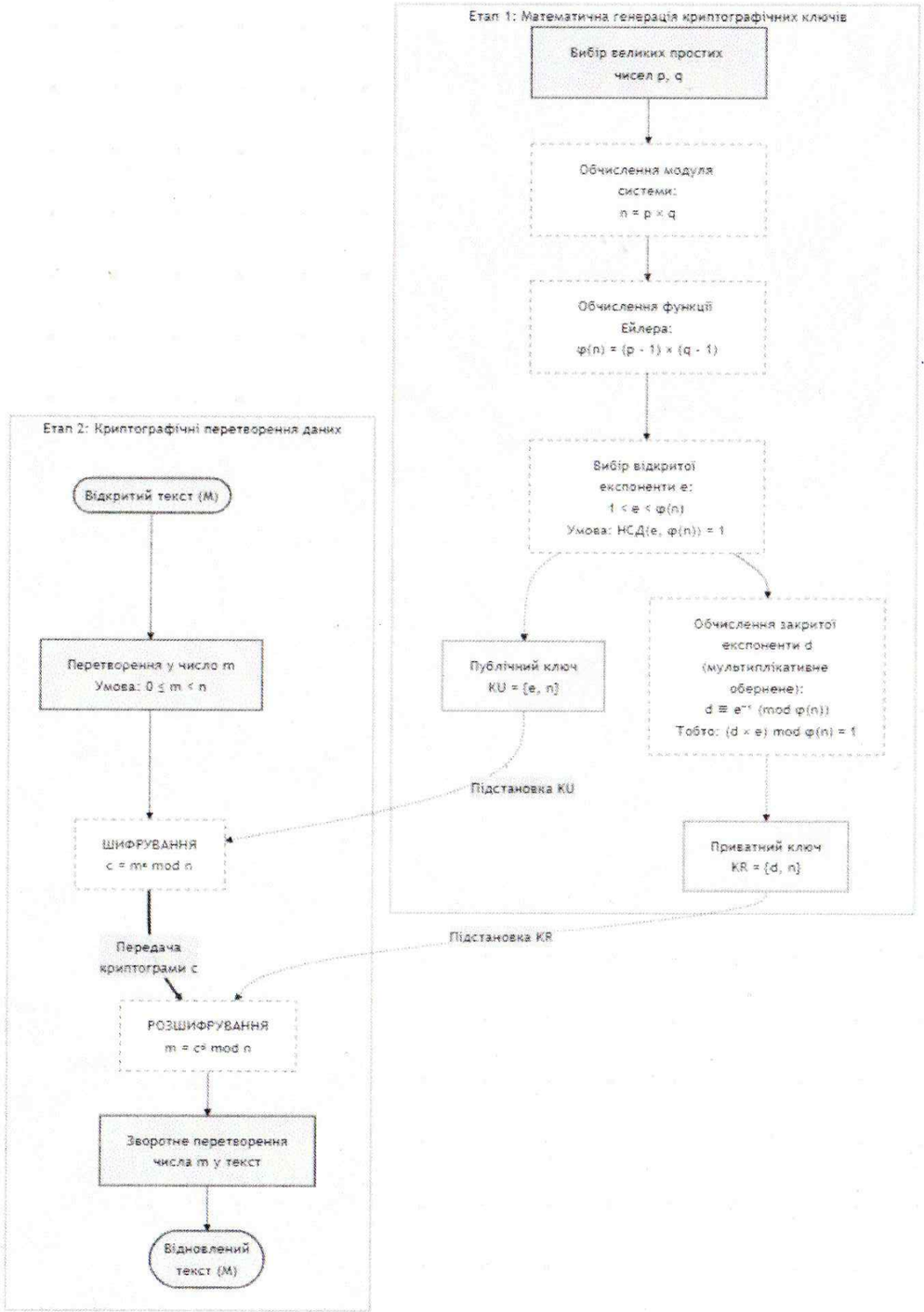
					КвРІПЗ. 220199.01.04.Е8			
Змн.	Арк	№ докум.	Підпис	Дата	Діаграма діяльності	Літ.	Маса	Масштаб
Розробив		Гаврилюк Д.О.		26.05.2026				
Керівник		Дьоміна А.І.		29.05.2026		Аркуш 1	Аркушів 4	
Н. Контр.		Форкун Ю. В.		29.05.2026	ХНУ,ІПЗ-22-1			
Затверд.		Бедратюк Л. П.		29.05.2026				



					КвРІПЗ. 220199.01.04.Е8	
Змн.	Арк	№ докум.	Підпис	Дата		
Розробив		Гаврилюк Д.О.		28.05.2026	Діаграма станів сесії користувача	*Літ.* / *Маса* / *Масштаб*
Керівник		Дьоміна А.І.		29.05.2026		Аркуш 2 / Аркушів 4
Н. Контр.		Форкун Ю. В.		29.05.26	ХНУ,ІПЗ-22-1	
Затверд.		Бедратюк Л. П.		29.05.2026		



					КвРІПЗ. 220199.01.04.Е8		
Змн.	Арк	№ докум.	Підпис	Дата			
Розробив		Гаврилюк Д.О.			Діаграма станів повідомлення	*Літ.*	*Маса* / *Масштаб*
Керівник		Дьоміна А.І.				Аркуш 3	Аркушів 4
Н. Контр.		Форкун Ю. В.				ХНУ,ІПЗ-22-1	
Затверд.		Бедратюк Л. П.					



					КвРІПЗ. 220199.01.04.Е8		
Змн.	Арк	№ докум.	Підпис	Дата	Блок-схема алгоритму шифрування RSA	Літ.	Маса / Масштаб
Розробив		Гаврилюк Д.О.		29.05.2026			
Керівник		Дьоміна А.І.		29.05.2026		Аркуш 4	Аркушів 4
Н. Контр.		Форкун Ю. В.		29.05.2026		ХНУ,ІПЗ-22-1	
Затверд.		Бедратюк Л. П.		29/05/2026			

СУПРОВІДНІ ДОКУМЕНТИ

Завідувачу кафедри інженерії програмного
забезпечення проф. Леоніду БЕДРАТЮКУ
здобувача вищої освіти
Гаврилюк Дарина Олегівна
факультет ІТ, IVкурс, група ІПЗ-22-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності в Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності, ознайомлена. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщена та надаю свою згоду на обробку й збереження університетом моєї роботи в інституційному репозитарії Хмельницького національного університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-обчислювального комплексу StrikePlagiarism та/або програмно-технічного засобу AntiPlagiarism і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення текстових збігів у роботах.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

29/05/2026
дата

підпис

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Дарина ГАВРИЛЮК

Співавтор:

Назва: Мобільна система обміну миттєвими повідомленнями з наскрізним шифруванням на платформі Android

Науковий керівник: Анастасія ДЬОМІНА

Підрозділ: Кафедра інженерії програмного забезпечення

Коефіцієнт подібності 1:5.93%

Коефіцієнт подібності 2:1.32%

Мікропробіли: 0

Заміна букв: 0

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2026-05-28 22:30:53.0

Після аналізу Звіту подібності констатую наступне:

☑ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

Дата 25.05.26

експерт

Anti-Plagiarism (http://ap.km.ua) v-16.718

Максимальне співпадіння з одним документом 2.0%

Словники перевірки: UA, US, RU. **Помилок в документах: 20%**

	Документ		Сумарний збіг по Базі Даних	
ID: 272700 Назва: БКР_Мобільна система обміну миттєвими повідомленнями з наскрізним шифруванням на платформі Android Додано в БД: 2026-05-28 Автора: Дарина ГАВРИЛЮК Керівники: Анастасія ДЬОМІНА Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	112261	962	4844 (4%)	61 (6%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ
освітнього ступеня «Бакалавр»

Дипломник Гаврилюк Дарина Олегівна

Тема Мобільна система обміну миттєвими повідомленнями з наскрізним шифруванням на платформі Android

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг кваліфікаційної роботи:

Кількість листів креслень 4; кількість сторінок записки 74

1.Короткий зміст пояснювальної записки та прийнятих рішень У кваліфікаційній роботі досліджено і проаналізовано предметну область, визначено усі функціональні та нефункціональні вимоги. Був проведений аналіз існуючих програм на ринку, розглянуто їх переваги і недоліки, та доведено актуальність розробки нового програмного забезпечення. Розглянуто інструменти для реалізації спроектованих рішень, в результаті чого створено програмне забезпечення. Також було проведено тестування програми, за результатами якого доведено, що розроблене програмне забезпечення працює коректно та готове до експуатації.

2. Висновок про відповідність роботи поставленому завданню Кваліфікаційна робота виконана відповідно до поставленого завдання та з дотриманням всіх вимог.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки та передових методів роботи У вступі доведено актуальність теми, визначено мету та завдання дипломного проектування. У першому розділі проведено аналіз предметної області, розглянуто існуючі рішення та визначені функціональні і нефункціональні вимоги до розроблюваного програмного забезпечення. У другому розділі проведено аналіз сучасних архітектур, розглянуто їх переваги і недоліки та визначено, що система буде відповідати монолітній архітектурі та моделі клієнт-сервер. У третьому розділі підготовлено всі залежності для написання коду та виконано практичну розробку програмних модулів і описано їх особливості, в результаті чого створено програмний продукт. Також у цьому розділі виконано модульне тестування системи та проведено його у відповідності до функціональних вимог, в результаті було підтверджено коректну роботу програми.

4. Позитивні сторони роботи Тематика кваліфікаційної роботи є актуальною, оскільки на сьогодні існуючі мобільні месенджери не є достатньо захищеними від фізичного зламу пристроїв та не мають повної апаратної ізоляції криптографічних ключів. Також було застосовано новітні технології для побудови програмного продукту та актуальні архітектурні рішення.

5. Негативні сторони роботи У роботі захищений обмін даними був реалізований лише для текстових повідомлень – було б доцільно додати можливість безпечного пересилання мультимедійних файлів та документів. Також краще реалізувати механізм верифікації співрозмовників за допомогою сканування QR-кодів, коли користувачі починають нове листування.

6. Оцінка графічного оформлення та пояснювальної записки Графічне оформлення виконано відповідно до теми кваліфікаційної роботи та подано у вигляді діаграм. Пояснювальна записка оформлена згідно вимог чинних стандартів.

7. Відгук про кваліфікаційну роботу в цілому Кваліфікаційна робота заслуговує позитивної оцінки. Матеріал пояснювальної записки структурований, послідовний, чіткий та простий, що дозволяє чітко зрозуміти викладений матеріал у рамках тематики проєктування. Графічний матеріал дає можливість наочно побачити деталі проектування системи.

8. Інші зауваження

9. Оцінка кваліфікаційної роботи Кваліфікаційна робота виконана у повному обсязі, відповідає поставленій задачі та заслуговує на оцінку «відмінно».

РЕЦЕНЗЕНТ Нічепорук Андрій Олександрович, кандидат технічних наук, доцент кафедри комп'ютерної інженерії та інформаційних систем

"29" травня 2026 р. _____
(підпис)

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатами звіту/звітів перевірки роботи, продукованими програмно-технічним засобом (ами), на наявність текстових збігів.
Назва кваліфікаційної роботи: «Мобільна система обміну миттєвими повідомленнями з наскрізним шифруванням на платформі Android »
Автор: Гаврилюк Дарина Олегівна
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Спеціальність: 121 – Інженерія програмного забезпечення
Науковий керівник: Дьоміна Анастасія Іванівна, асистентка
Після аналізу звіту/звітів зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається до захисту.	**відповідає**
2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована.	
3	Виявлені запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Виявлені запозичення частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнуті. Робота може бути допущена до захисту після того, як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:
Запозичення, виявлені у роботі, є законними і не є плагіатом, оскільки:
1) у тексті кваліфікаційної роботи системою перевірки на плагіат Anti-Plagiarism виявлено схожість з деякими документами у частині загальновживаних обов'язкових словосполучень у стандартних бланках, у структурі змісту, назвах розділів/підрозділів, рамках форм, у назвах та URL-адресах публікацій переліку джерел посилання;
2) запозичення, виявлені у тексті роботи, є фрагментарними.
Максимальний обсяг запозичень, визнаний системою Anti-Plagiarism, складає 2.0% з одного джерела. Загальна сумарна подібність у базі даних складає 4% за символами та 6% за лексемами. Крім того, за результатами додаткового аналізу коефіцієнт подібності 1 становить 5.93% коефіцієнт подібності 2 – 1.32%. Виявлено 0 мікропробілів, проте не виявлено зайвих білих знаків або маніпуляцій з інтервалами. З урахуванням наведених обґрунтувань, відповідає характеру теми і свідчить на користь кваліфікаційної роботи.

Дата 1.06.2026

Завідувач кафедри _____ Леонід БЕДРАТЮК
Гарант освітньої програми _____ Леонід БЕДРАТЮК
Керівник кваліфікаційної роботи _____ Анастасія ДЬОМІНА