Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Мобільний застосунок для створення й збереження нотаток у формат

Назва теми

особистого чату на платформі Android

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. 2201114.01.19.ПЗ

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

Керівник д-р фіз.-мат. наук, професор. _____ Леонід БЕДРАТЮК
Науковий ступінь, вчене звання Підпис Ім'я, ПРІЗВИЩЕ

Нормоконтролер асистент _____ В'ячеслав БОЙКО
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

Хотича Миколи Володимировича

Прізвище, ім'я, по батькові студента

1. Тема роботи Мобільний застосунок для створення й збереження нотаток у форматі особистого чату на платформі Android

Керівник роботи Бедратюк Леонід Петрович, д-р фіз.-мат. наук, професор.

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 та постановка задачі, Проєктування мобільного застосунку, Програмна реалізація та тестування мобільного застосунку

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

Три креслення:

1. Діаграма варіантів використання
2. Логічна модель бази даних
3. Шарова архітектура застосунку

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Бойко В. О., асистент	01.06.2026	01.06.2026
Антиплагіат	Форкун Ю. В., доцент	25.05.26	25.05.26

7. Дата видачі завдання «_02_»__січня__2026 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1. Ознайомлення з тематикою, визначення та узгодження тем КвР	01.12–31.12.2025	
2. Збір матеріалу, дослідження предметної області, розробка ТЗ	01.01–20.02.2026	
3. Проектування програмного забезпечення	21.02–20.03.2026	
4. Програмна реалізація та тестування	21.03–30.04.2026	
5. Написання вступу, висновків, оформлення списку джерел та додатків	01.05–25.05.2026	
6. Попередній захист КвР	Травень 2026	Згідно з графіком
7. Перевірка на плагіат, нормоконтроль, отримання відгуків, брошурування	26.05–30.06.2026	
8. Здача КвР на кафедру, підготовка до репозитарію та захист	З 01.06.2026	

Студент _____ Микола ХОТИЧ
Підпис Ім'я, ПРІЗВИЩЕ

Керівник роботи _____ Леонід БЕДРАТЮК
Підпис Ім'я, ПРІЗВИЩЕ

ВІДОМІСТЬ ДОКУМЕНТІВ

№ рядка	Формат	Позначення документа	Найменування документа	К-сть аркушів	№ екз.	Примітка
			Текстові документи			
1	А4	КвРІПЗ. 2201114.01.19.ПЗ	Пояснювальна записка	74		
2	А4		Завдання на кваліфіка-	1		
			ційну роботу			
3	А4		Анотація	1		
			Графічні документи			
4	А3	КвРІПЗ. 2201114.01.19.Е2	Діаграма варіантів	1		
			використання			
5	А3	КвРІПЗ. 2201114.01.19.Е1	Логічна модель	1		
			бази даних			
6	А3	КвРІПЗ. 2201114.01.19.Е1	Шарова архітектура	1		
			застосунку			

Змн.	Арк.	№ докум.	Підпис	Дата	КвРІПЗ. 2201114.01.19.ПЗ		
Виконав		Хотич М.В.		22.05.2026	Мобільний застосунок для створення й збереження нотаток у форматі особистого чату на платформі Android Відомість документів	Літ.	Арк. 4 / Аркушів 74
Керівник		Бедратюк Л.П.		1.06.2026			
Н. контр.		Бойко В.О.		1.06.2026		ХНУ, ІПЗ-22-1	
Зав. каф.		Бедратюк Л.П.		1.06.2026			

АНОТАЦІЯ

Тема кваліфікаційної роботи «Мобільний застосунок для створення й збереження нотаток у форматі особистого чату на платформі Android».

Автор роботи: Хотич Микола Володимирович.

Керівник роботи: Бедратюк Леонід Петрович.

Пояснювальна записка: 74 с., 16 рис., 15 табл., 3 дод., 35 джерел.

Графічна частина: 3 креслення ф. А3.

ЛОКАЛЬНЕ ЗБЕРІГАННЯ, МОБІЛЬНИЙ ЗАСТОСУНОК, НОТАТКА, ОСОБИСТИЙ ЧАТ, ANDROID, JETPACK COMPOSE, KOTLIN.

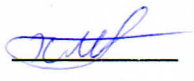
Мета кваліфікаційної роботи: розроблення локального мобільного застосунку для платформи Android, що дозволяє створювати та зберігати нотатки у форматі тематичних чатів із гарантованою конфіденційністю даних.

У кваліфікаційній роботі проаналізовано предметну область та наявні рішення, визначено вимоги до програмного забезпечення, обґрунтовано архітектуру на основі Clean Architecture з патерном MVI, спроєктовано локальну базу даних, інтерфейс користувача та алгоритми ключових процесів. Для реалізації використано Kotlin, Jetpack Compose, Room та Hilt.

Практичне значення результатів роботи полягає в тому, що розроблений застосунок дозволяє оперативно фіксувати персональну інформацію у звичному діалоговому форматі, структурувати її засобами тематичних чатів і тегів та гарантує конфіденційність завдяки локальному зберіганню даних.

28.05.2026

Дата



Підпис

ЗМІСТ

ВСТУП

У сучасному інформаційному суспільстві ефективна організація та оперативна фіксація персональної інформації набуває дедалі більшої актуальності. Щоденне зростання обсягів даних, з якими взаємодіє людина, призводить до значного підвищення когнітивного навантаження, що ставить нові вимоги до програмних засобів управління особистими записами. Особливої актуальності ця проблема набуває в контексті мобільних платформ, де користувачі очікують миттєвого доступу до інструментів фіксації інформації в будь-який момент.

Традиційні застосунки для ведення нотаток здебільшого пропонують складні ієрархічні структури з розгалуженими системами налаштувань, що вимагає від користувача додаткових зусиль і часу навіть для збереження лаконічних ідей. Альтернативним підходом стало використання месенджерів — їхній діалоговий інтерфейс забезпечує високу швидкість фіксації, однак не пропонує інструментів категоризації, що ускладнює подальший пошук і систематизацію збережених даних. Таким чином, жоден із поширених підходів не забезпечує одночасно швидкість фіксації, зручну категоризацію та конфіденційність, що свідчить про наявність незакритої потреби серед існуючих програмних рішень.

Не менш важливою проблемою є конфіденційність особистих записів. Зберігання даних на сторонніх хмарних серверах створює ризики компрометації інформації та робить роботу залежною від стабільного підключення до Інтернету. Це актуалізує доцільність розроблення програмних рішень, що ґрунтуються на концепції виключно локального збереження даних. Відтак актуальність цього дослідження зумовлена необхідністю створення програмного інструменту, який поєднує оперативність та ергономічність формату чату з розширеними можливостями структурування інформації та гарантованим рівнем безпеки.

Мета кваліфікаційної роботи полягає у розробленні локального мобільного застосунку для операційної системи Android, що надаватиме користувачам

можливість створювати нотатки у форматі тематичних чатів, забезпечуючи високу швидкість фіксації інформації, ефективну систему категоризації та гарантовану конфіденційність завдяки локальному збереженню даних.

Для досягнення поставленої мети передбачено виконання таких завдань:

– проаналізувати предметну область та виявити її структурні і функціональні особливості;

– виконати порівняльний аналіз наявних на ринку програмних рішень, визначити їхні переваги, недоліки та набір функціональних можливостей;

– сформулювати функціональні та нефункціональні вимоги до мобільного застосунку;

– спроєктувати архітектуру мобільного застосунку та обґрунтувати вибір архітектурного рішення;

– розробити концепцію інтерфейсу користувача та візуального дизайну;

– розробити алгоритми ключових процесів функціонування застосунку;

– обґрунтувати вибір технологічного стеку та інструментів;

– виконати програмну реалізацію мобільного застосунку та провести його тестування для перевірки відповідності визначеним вимогам.

Практичне значення одержаних результатів полягає у створенні функціонально завершеного програмного продукту, який забезпечить ключову цінності для користувача за мінімального обсягу функціонала. Розроблений застосунок оптимізує процес фіксації думок у звичному форматі повідомлень із гарантуванням високого рівня конфіденційності та орієнтований на користувачів, які потребують оперативного, структурованого та безпечного інструменту для ведення особистих записів без Інтернет-з'єднання.

Об'єктом дослідження є процес управління персональними записами користувача на мобільній платформі Android. Предметом дослідження є методи та засоби розроблення мобільного застосунку з діалоговим інтерфейсом і локальним збереженням даних.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Змістовий аналіз предметної області, її структурних та функціональних особливостей

Ведення нотаток є фундаментальною складовою навчання, професійної діяльності та повсякденного життя людини [1]. Історично цей процес еволюціонував від використання фізичних носіїв — паперу та блокнотів — до складних цифрових систем управління знаннями. Перехід до цифрових засобів зумовлений потребою у швидкому доступі до інформації, її пошуку та можливості редагування записів незалежно від місця перебування користувача.

Стрімкий розвиток цього сегменту програмного забезпечення підтверджується актуальними аналітичними даними. Згідно з дослідженням аналітичної компанії «Research and Markets», станом на 2025 рік обсяг глобального ринку мобільних застосунків для нотаток оцінюється у 11,11 млрд доларів США, а до 2029 року прогнозується його зростання до 23,79 млрд доларів із середньорічним темпом 21,0% [2]. Такі показники свідчать про стійкий попит на цифрові інструменти ведення записів та високу конкуренцію серед розробників.

Сучасні дослідження когнітивного розвантаження (cognitive offloading) — делегування мнемічних і обчислювальних завдань зовнішнім засобам — показують, що будь-які додаткові дії інтерфейсу при фіксації думки створюють надлишкове когнітивне навантаження та знижують ефективність засвоєння знань [3]. Інтерфейс і модель взаємодії мобільного пристрою безпосередньо впливають на готовність користувача переносити робочі процеси пам'яті у зовнішнє середовище [4], а надмірно тривалий процес фіксації призводить до погіршення внутрішнього запам'ятовування інформації [5].

Традиційні застосунки для продуктивності часто змушують виконувати складну послідовність дій для створення одного запису: придумування заголовка, вибір папки в ієрархічній структурі або призначення тегів ще до введення тексту

виступають штучними бар'єрами. Унаслідок цього користувачі витрачають більше часу та ментальної енергії на організацію цифрового простору, ніж на безпосередню фіксацію знань.

Систематичний огляд досліджень із делегування намірів зовнішнім засобам «intention offloading» показує, що користувачі систематично надають перевагу зовнішнім нагадуванням над внутрішнім запам'ятовуванням навіть тоді, коли це не є оптимальним з огляду на витрати на створення такого нагадування [6]. Це додатково обґрунтовує доцільність розроблення інструментів, які мінімізують зусилля на створення зовнішнього запису. Найбільш повним практичним втіленням цього принципу став діалоговий інтерфейс, який стрімко розвинувся протягом останнього десятиліття. На відміну від класичних графічних інтерфейсів, що покладаються на складні меню, форми для заповнення та ієрархічні дерева, діалоговий інтерфейс імітує природне людське спілкування, що робить його особливо ефективним для швидкої фіксації думок.

Використання формату чату для ведення нотаток має низку переваг. По-перше, він відповідає усталеним моторним та когнітивним звичкам користувачів мобільних пристроїв: введення тексту в поле внизу екрана та натискання кнопки «надіслати» є звичною дією з месенджерів і не потребує додаткового навчання. Повторне використання усталених патернів інтерфейсу істотно знижує час адаптації до нового продукту [7].

Друга перевага — хронологічна організація інформації за замовчуванням. У традиційних блокнотах нові записи часто сортуються за алфавітом або вимагають ручного переміщення, тоді як у чаті кожне нове повідомлення автоматично додається в кінець стрічки з точною фіксацією часу створення. Такий підхід відповідає природним механізмам людської пам'яті, оскільки контекст створення нотатки часто згадується через її часову прив'язку.

Третя перевага — модульність та атомарність збережених даних. У звичайному документі текст є єдиним суцільним масивом, тому видалення однієї ідеї або її переміщення вимагає використання інструментів виділення, вирізання

та вставлення тексту, що на сенсорних екранах мобільних пристроїв буває незручно. У форматі чату кожен запис є окремим ізольованим повідомленням, що дозволяє легко взаємодіяти з ним: копіювати, видаляти, прикріплювати метадані або швидко ділитися з іншими застосунками.

Окрім перелічених переваг, формат чату стимулює регулярне ведення записів. Відсутність великого порожнього простору, який психологічно вимагає заповнення структурованим та вичитаним текстом, знімає бар'єр перфекціонізму. Користувач без вагань фіксує коротку фразу, окреме слово або посилання, що відкриває шляхи до подальшої інтеграції інструментів аналізу повідомлень — напряму, який вже стає трендом на ринку систем управління знаннями.

Поряд із форматом взаємодії важливим аспектом є місце збереження даних. Перехід індустрії на хмарні обчислення є зручним для спільної роботи команд, проте виявився проблематичним для персональних інструментів, що зумовило формування нового напряму — локального програмного забезпечення.

Основний принцип цієї парадигми полягає у тому, що первинним і головним місцем збереження даних користувача є його власний пристрій. Такий підхід дозволяє поєднати зручність сучасних хмарних застосунків з абсолютним правом власності на дані, притаманним класичному офлайн-забезпеченню [8]. У межах цієї концепції виділяють сім ключових ідеалів локальної архітектури, які визначають її переваги над централізованими хмарними рішеннями. Ідеали та їх практичне значення для користувача представлені в таблиці 1.1.

Таблиця 1.1 – Сім ідеалів локального програмного забезпечення

Ідеал	Практичне значення для користувача
Миттєвий доступ	Відсутність індикаторів завантаження; дані завжди під рукою та доступні миттєво з пристрою.

Кінець таблиці 1.1

Мультиплатформність	Дані не ізольовані на одному пристрої; архітектура підтримує синхронізацію між платформами.
Мережа є опціональною	Повноцінна робота (запис та читання нотаток) гарантується навіть за повної відсутності Інтернету.
Безшовна співпраця	За наявності підключення можлива надійна синхронізація без конфліктів.
Довговічність	Інформація продовжує жити на пристрої користувача навіть якщо розробник закриє сервери компанії.
Приватність за замовчуванням	Гарантована конфіденційність, адже дані фізично зберігаються на пристрої, а не в корпоративній хмарі.
Повний контроль користувача	Абсолютне та безумовне право власності на створену інформацію без диктату хмарних провайдерів.

Окрім перелічених у таблиці характеристик, важливим стимулом переходу до локальної архітектури є зростаюча необхідність захисту конфіденційності. Зберігання особистих нотаток, думок чи фінансових записів на сторонніх серверах створює значні ризики: компанії можуть змінювати умови надання послуг, використовувати тексти для навчання нейронних мереж або надавати доступ до даних третім особам. Відмова від постійної синхронізації з хмарою гарантує, що інформація не покине пристрій без прямої вказівки власника. Додатковою перевагою локальної архітектури для мобільних пристроїв є оптимізація споживання ресурсів акумулятора та мобільного трафіку. Таким чином, локальна архітектура забезпечує не лише технічні переваги, а й формує якісно вищий рівень довіри користувача до програмного засобу.

1.2 Аналіз наявного програмно-технічного забезпечення предметної області

Більшість наявних на ринку рішень для ведення нотаток на мобільних платформах вимагає компромісу між зручністю, швидкістю та безпекою. Для обґрунтування необхідності розроблення нового продукту проаналізовано три популярні аналоги, що представляють різні підходи: месенджер у ролі нотатника, класичний хмарний нотатник та локальну систему управління знаннями.

Першим розглянутим аналогом є функція збережених повідомлень «Saved Messages» у месенджері Telegram. Цей інструмент базується на веденні персонального чату користувача із самим собою та став одним із найпопулярніших нестандартних підходів до фіксації нотаток. Інтерфейс збережених повідомлень месенджера подано на рисунку 1.1.

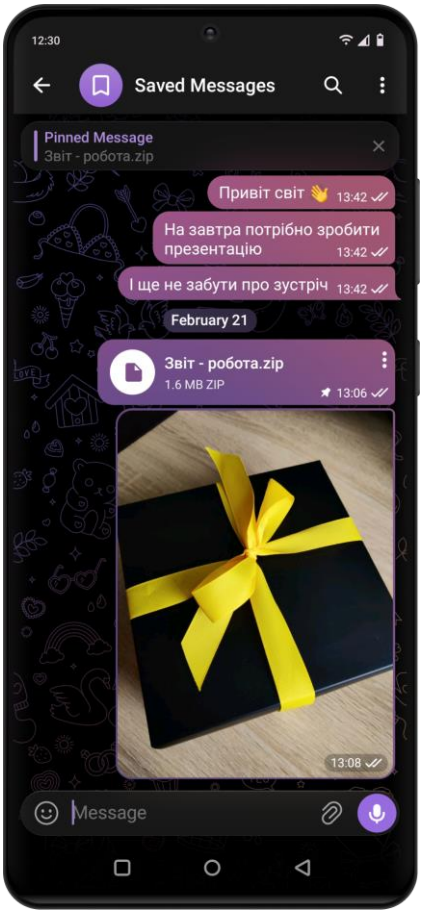


Рисунок 1.1 – Головний екран збережених повідомлень Telegram

Головною перевагою такого підходу є висока швидкість фіксації думок: діалоговий інтерфейс вимагає значно менше часу та зусиль для створення запису порівняно з графічними інтерфейсами традиційних застосунків. Окрім того, сервіс дозволяє зберігати медіафайли будь-якого типу без втрати якості та забезпечує синхронізацію між усіма підключеними пристроями користувача.

Водночас цей підхід має суттєві недоліки. Найбільший із них — повна залежність від серверної інфраструктури розробника та постійної наявності підключення до Інтернету. Окрім того, відсутня можливість створення окремих тематичних чатів, що з часом ускладнює пошук потрібної інформації, оскільки всі записи накопичуються в єдиній стрічці.

Другим аналогом, обраним для аналізу, є Google Keep — класичний представник хмарних систем для ведення нотаток. Цей застосунок розроблявся як цифровий аналог дошки з наліпками та орієнтований на швидке створення коротких списків і нагадувань. Головний екран застосунку подано на рисунку 1.2.

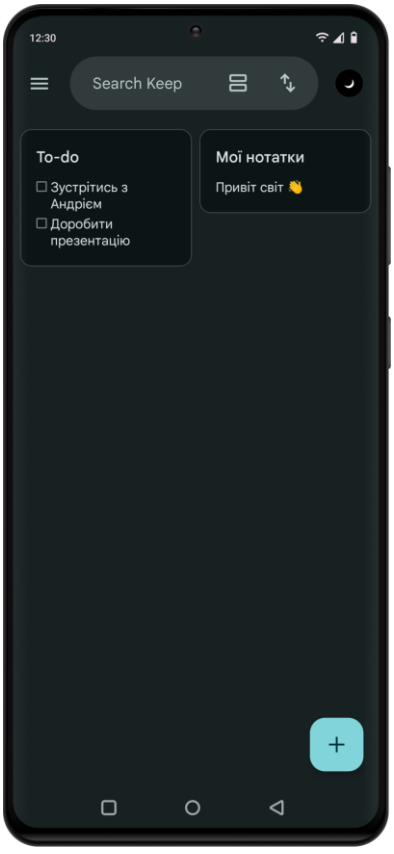


Рисунок 1.2 – Головний екран застосунку Google Keep

До переваг Google Keep належать простота використання, можливість кольорового маркування нотаток для візуального розділення, а також глибока інтеграція з іншими сервісами екосистеми Google. Однак зі збільшенням кількості записів інтерфейс у вигляді карток швидко стає хаотичним та візуально перевантаженим. Застосунок не підтримує побудови глибокої ієрархії або складної структури матеріалів, що обмежує його використання як повноцінної системи управління знаннями. Окрім того, тісна прив'язка даних до єдиного хмарного облікового запису корпорації суперечить принципу повного контролю користувача над власною інформацією.

Третім розглянутим аналогом є Obsidian — система управління знаннями, що реалізує підхід, протилежний до двох попередніх. Цей мобільний застосунок фокусується на локальному збереженні текстових файлів у форматі Markdown та надає користувачеві інструменти для побудови складних взаємозв'язків між ними. Інтерфейс мобільної версії системи зображено на рисунку 1.3.

Рисунок 1.3 – Головний екран мобільного застосунку Obsidian

Беззаперечною перевагою Obsidian є високий рівень безпеки та приватності, який забезпечується парадигмою «local-first». Уся база даних фізично знаходиться на мобільному пристрої, завдяки чому гарантується автономність роботи та повний контроль користувача над власною інформацією. Застосунок пропонує гнучкі можливості структурування за допомогою системи вкладених папок, тегів та графу зв'язків. Однак ці потужні функціональні можливості зумовлюють високий поріг входу та перевантаженість інтерфейсу. Процес збереження ідеї стає складним, оскільки вимагає створення нового файлу, придумування для нього заголовка та визначення місця збереження, що суттєво підвищує когнітивне навантаження порівняно з простими месенджерами.

Для систематизації результатів дослідження сформовано порівняльну характеристику проаналізованих програмних рішень, подану в таблиці 1.2.

Таблиця 1.2 – Порівняльна характеристика наявних програмних рішень

Критерій	Telegram (Saved Messages)	Google Keep	Obsidian
Формат інтерфейсу	Чат	Картки та списки	Текстовий редактор
Швидкість створення запису	Висока	Середня	Низька
Наявність категорій	Відсутня	Наявна (теги, кольори)	Наявна (папки, теги)
Архітектура збереження даних	Хмарна	Хмарна	Локальна
Складність інтерфейсу	Низька	Середня	Висока

Проведений аналіз показує, що на сучасному ринку відсутнє комплексне рішення, яке одночасно задовольняло б потреби у швидкості, зручності та безпеці. Месенджери пропонують найкращий користувацький досвід щодо швидкості введення даних, проте їхня хмарна природа та відсутність структурування роблять їх непридатними для серйозного управління інформацією. Локальні системи

натомість забезпечують надійний захист і контроль, але їхній складний інтерфейс перешкоджає швидкому запису ідей. Класичні хмарні нотатники, своєю чергою, порушують принцип приватності та характеризуються візуальним перевантаженням інтерфейсу.

З огляду на ці висновки розроблення нового мобільного застосунку є обґрунтованим та доцільним. Проєктований програмний продукт має синтезувати найкращі практики проаналізованих рішень. Від месенджерів буде запозичено концепцію діалогового інтерфейсу для мінімізації когнітивного навантаження, доповнену можливістю створювати декілька тематичних чатів для зручного розподілу нотаток. Від системи Obsidian буде інтегровано парадигму локального збереження даних безпосередньо на пристрої, що гарантуватиме конфіденційність, незалежність від якості Інтернет-з'єднання та миттєвий відгук інтерфейсу. Такий гібридний підхід дозволить створити інструмент, який поєднує сильні сторони аналогів за показниками швидкодії, організації та безпеки.

1.3 Визначення функціональних та нефункціональних вимог до програмного забезпечення

На основі проведеного аналізу предметної області та розглянутих аналогів сформовано вимоги до розроблюваного програмного забезпечення. Для формалізації функціональних вимог у межах об'єктно-орієнтованої парадигми використано мову Unified Modeling Language (UML). Як підтверджують сучасні дослідження в галузі програмної інженерії, побудова діаграм варіантів використання (ВВ) є одним із найефективніших та найпоширеніших методів моделювання і документування функціональних вимог системи, оскільки наочно відображає сценарії взаємодії між користувачами та системою під час виконання конкретних завдань [9]. Такий підхід дозволяє абстрагуватись від технічних деталей реалізації та зосередитись на функціональних очікуваннях користувача як основному критерії якості майбутнього застосунку.

Оскільки проєктований застосунок ґрунтується на парадигмі локального ПЗ та призначений для персонального ведення нотаток на мобільному пристрої, у системі виділяється лише один основний актор. Його опис подано в таблиці 1.3.

Таблиця 1.3 – Опис головного актора

Актор	Короткий опис
Користувач	Особа, яка безпосередньо взаємодіє із застосунком на своєму смартфоні. Цей актор має повний та ексклюзивний доступ до всього функціоналу системи, включаючи створення даних, їх структурування, управління мультимедійними файлами та експорт локальної бази даних.

Для виділеного актора визначено базові варіанти використання та сформовано їх короткі специфікації, подані в таблиці 1.4.

Таблиця 1.4 – Опис варіантів використання головного актора

Назва ВВ	Опис ВВ
Створення тематичного чату	Користувач ініціює створення нової категорії у вигляді чату, задаючи їй унікальну назву. Цей процес є основою для подальшої систематизації нотаток.
Налаштування візуального оформлення чату	Для кращої візуальної категоризації та швидкої навігації користувач може обрати для кожного чату унікальний колір, відповідне емоджі або встановити власне фонове зображення з галереї.
Відправлення текстової нотатки	Базовий сценарій, під час якого користувач вводить текст у поле введення та натискає кнопку відправлення. Нотатка миттєво фіксується у вибраному чаті із збереженням точного часу створення.

Кінець таблиці 1.4

Прикріплення файлів та медіа	Користувач може доповнювати свої записи різноманітними файлами. Сценарій передбачає можливість вибору документів із файлової системи, додавання зображень чи відео з галереї пристрою, а також безпосереднє створення нових фото чи відеоматеріалів через інтерфейс камери.
Створення аудіоповідомлень	За аналогією до популярних месенджерів, актор має можливість швидко записати голосову нотатку, яка зберігається як аудіофайл і відображається у стрічці чату з можливістю подальшого відтворення.
Отримання даних через системні механізми «Intent»	Користувач може знаходитися в іншій програмі і через системне меню «Поділитися» переслати текст, посилання або файл безпосередньо у розроблюваний застосунок, обравши потрібний цільовий чат.
Збереження даних з буфера обміну	При вході в чат система може автоматично пропонувати зберегти текстову інформацію чи файли, які були скопійовані актором раніше.
Пошук інформації	Користувач вводить пошуковий запит для знаходження конкретних текстових повідомлень у межах вибраного чату. Система шукає відповідні повідомлення та підсвічує знайдені збіги.
Експорт даних	Для забезпечення повного контролю над інформацією користувач ініціює процес створення резервної копії. У результаті виконання цього варіанту використання система формує єдиний ZIP-архів, який містить локальну базу даних, усі прикріплені медіафайли та налаштування.
Імпорт даних	Користувач завантажує раніше створений ZIP-архів у систему для відновлення всіх своїх записів, чатів та файлів на поточному чи новому пристрої.

У процесі моделювання також враховано альтернативні та заборонені шляхи виконання зазначених сценаріїв. До альтернативних шляхів належать ситуації, коли під час імпорту або експорту великого ZIP-архіву на пристрої користувача

не вистачає вільного місця: у такому випадку система виводить повідомлення про помилку, не пошкоджуючи наявні дані. До заборонених шляхів належать спроби створення чату без указування назви або відправлення повідомлення без контенту — такі дії блокуються на рівні інтерфейсу шляхом деактивації відповідних кнопок.

Окрему увагу під час аналізу приділено вимогам до інтерфейсу користувача. Для забезпечення максимальної швидкості створення записів та мінімального когнітивного навантаження форма взаємодії повинна спиратися на звичні патерни месенджерів. Головним екраном застосунку, що відображається одразу при запуску, має бути останній відкритий користувачем чат — це дозволить миттєво вводити нотатки без зайвої навігації.

Узагальнену модель взаємодії користувача із застосунком подано на діаграмі варіантів використання (рисунок 1.4).

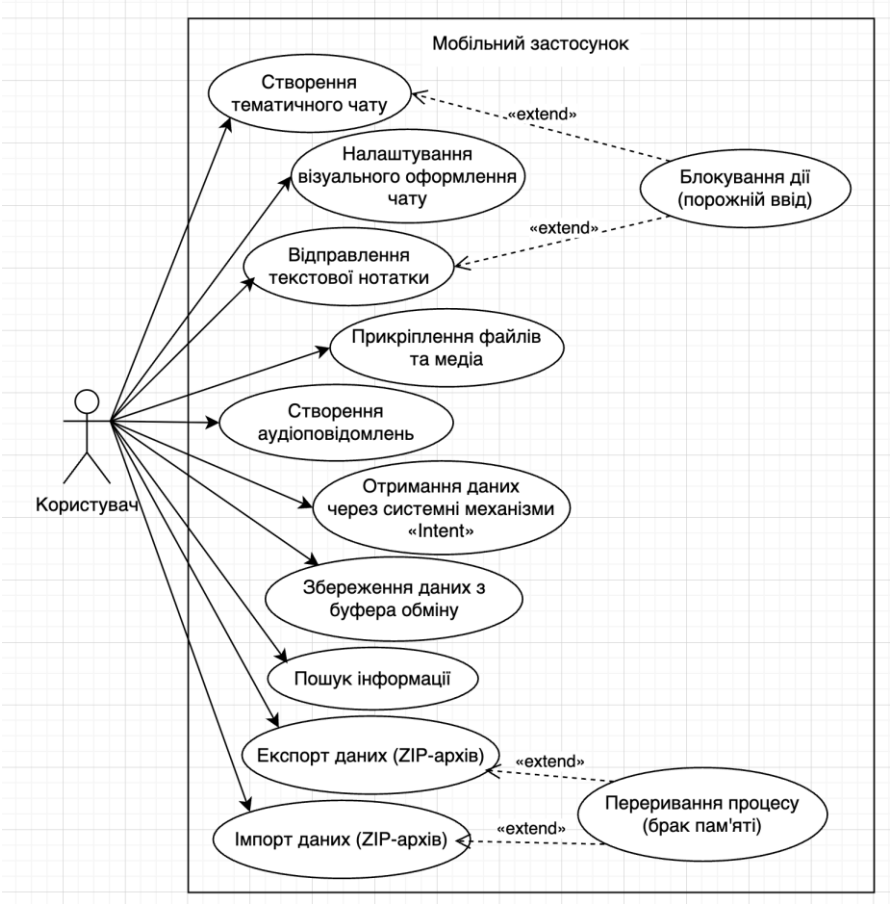


Рисунок 1.4 – Діаграма варіантів використання

На основі описаних варіантів використання сформульовано функціональні та нефункціональні вимоги до системи. Функціональні вимоги визначають конкретні можливості системи та поведінку, яку вона повинна забезпечувати для задоволення потреб користувача [10]. Вони є основою для визначення меж системи та слугують головним критерієм перевірки відповідності готового продукту очікуванням замовника. Нефункціональні вимоги визначають якісні характеристики системи — продуктивність, надійність, безпеку та зручність використання, що формують загальний користувацький досвід [11].

Визначення вимог виконувалось з урахуванням результатів порівняльного аналізу аналогів, проведеного у попередньому підрозділі. Це дозволило сформувати збалансований перелік можливостей, що усуває виявлені недоліки існуючих рішень.

Функціональні вимоги до застосунку включають:

– створення, редагування та видалення тематичних чатів;

– функціонування екрана останнього відкритого чату як головного екрана застосунку;

– відправлення, редагування та видалення текстових повідомлень із підтримкою базового форматування Markdown;

– відповідь на повідомлення для створення зв'язків між записами;

– закріплення важливих повідомлень для швидкого доступу до них;

– встановлення нагадувань на конкретні повідомлення з можливістю отримувати сповіщення у визначений час;

– позначення повідомлень як виконаних із візуальним відображенням;

– додавання мультимедійного контенту (зображень, відео, аудіо, файлів) для збереження додаткових матеріалів;

– перегляд мультимедійного контенту популярних форматів безпосередньо в застосунку;

– присвоєння повідомленням користувацьких тегів для категоризації;

– пошук записів у межах обраного чату та фільтрація за тегами;

– отримання контенту зі сторонніх застосунків через системний механізм Intent;

– пересилання повідомлень із передачею тексту та мультимедіа у сторонні застосунки;

– формування резервних копій усіх даних застосунку у форматі ZIP-архіву та їх відновлення.

Нефункціональні вимоги до застосунку включають:

– коректне функціонування під управлінням операційної системи Android версії 8.0 та новіших версій;

– захист від зворотної інженерії та оптимізація програмного коду шляхом обфускації засобами R8 та ProGuard;

– візуальне оформлення на принципах сучасного мінімалізму відповідно до специфікацій Material Design 3;

– час холодного запуску застосунку та ініціалізації останнього відкритого чату — не більше двох секунд;

– виключно локальне зберігання текстових записів, метаданих та мультимедійних файлів у внутрішній пам'яті пристрою;

– адаптивність інтерфейсу під різні розміри екранів;

– використання знайомих патернів інтерфейсу та плавних анімацій для забезпечення інтуїтивно зрозумілої взаємодії.

На основі визначених функціональних та нефункціональних вимог, з урахуванням особливостей предметної області, розроблено деталізоване технічне завдання (ТЗ) на створення програмного продукту. Цей документ виступає основним регламентом, що фіксує етапи розроблення, архітектурні обмеження та критерії приймання готового застосунку. Наявність чітко сформульованого технічного завдання мінімізує ризик відхилення від поставленої мети в процесі реалізації. Повний текст технічного завдання наведено в додатку А.

1.4 Висновки. Постановка задачі

Проведене у першому розділі дослідження предметної області дозволило сформувати цілісне уявлення про сучасний стан інструментів для ведення персональних нотаток та виявити ключові обмеження наявних на ринку рішень.

Аналіз показав, що сегмент мобільних застосунків для нотаток демонструє стрімке зростання, проте більшість представлених на ринку рішень не задовольняє одночасно трьох критичних потреб користувача: швидкості фіксації, можливостей структурування та конфіденційності даних. Розглянуті аналоги — функція збережених повідомлень Telegram, застосунок Google Keep та система Obsidian — представляють різні підходи до організації інформації, але кожен із них реалізує лише частину необхідного функціоналу. Месенджери забезпечують високу швидкість введення даних завдяки діалоговому інтерфейсу, проте зберігають інформацію у хмарі та не пропонують засобів категоризації. Класичні хмарні нотатники надають базові можливості структурування, але порушують принцип приватності та з часом перевантажують візуальний інтерфейс. Локальні системи управління знаннями гарантують конфіденційність даних, однак мають високий поріг входу та складний для повсякденного використання інтерфейс.

Відповідно до результатів аналізу обґрунтовано доцільність розроблення нового мобільного застосунку, який синтезує найкращі практики розглянутих рішень: концепцію діалогового інтерфейсу, парадигму локального збереження даних та можливість структурування записів через систему тематичних чатів. Розроблюваний продукт орієнтується на концепцію мінімально життєздатного продукту [12], що передбачає виокремлення мінімального набору функціональних можливостей, достатнього для забезпечення основної цінності для користувача. Такий гібридний підхід дозволяє зберегти швидкість фіксації, властиву месенджерам, поєднавши її з гарантованою конфіденційністю та зручною категоризацією інформації.

На основі проведеного дослідження сформульовано функціональні та нефункціональні вимоги до системи, які формалізовано засобами мови UML у вигляді діаграми варіантів використання та узагальнено в таблицях специфікацій. Визначено основного актора системи — користувача — та десять базових варіантів використання, що охоплюють повний цикл роботи із застосунком: від створення тематичних чатів до експорту резервних копій даних.

З огляду на отримані результати для подальшого розроблення мобільного застосунку необхідно розв'язати такі задачі проєктування:

– спроєктувати архітектуру застосунку;

– розробити структуру локальної бази даних для зберігання тематичних чатів, повідомлень, мультимедійних вкладень, тегів і нагадувань з урахуванням взаємозв'язків між сутностями;

– спроєктувати інтерфейс користувача відповідно до специфікацій Material Design 3 з пріоритетом на ергономіку повсякденного використання та мінімізацію когнітивного навантаження;

– розробити алгоритми ключових процесів функціонування застосунку;

– обґрунтувати вибір технологічного стеку для реалізації застосунку, зокрема мови програмування, інструментарію побудови інтерфейсу та засобів роботи з локальним сховищем даних;

– виконати програмну реалізацію застосунку на основі розробленого проєкту, забезпечивши коректну роботу під управлінням операційної системи Android версії 8.0 та новіших;

– провести тестування розробленого застосунку для перевірки відповідності реалізованої функціональності визначеним вимогам та оцінювання якісних характеристик системи.

Розв'язання поставлених задач забезпечить створення функціонально завершеного мобільного застосунку, який відповідає сформованим вимогам та закриває виявлені обмеження наявних аналогів.

2 ПРОЄКТУВАННЯ МОБІЛЬНОГО ЗАСТОСУНКУ

2.1 Архітектура та функціональна структура застосунку

Архітектура визначає масштабованість, тестовність та підтримуваність продукту [13], а помилки цього етапу у подальшому потребують значних ресурсів для виправлення, тому вибір повинен ґрунтуватися на аналізі альтернатив із урахуванням специфіки програмного засобу [14].

Сучасна практика розроблення для платформи Android передбачає прийняття рішень на двох рівнях [15]: загальна структура застосунку, що описує розподіл коду між функціональними блоками й правила їх взаємодії, та патерн представлення (UI-патерн), який регламентує взаємодію інтерфейсу користувача з логікою застосунку [16].

Історично для мобільного розроблення сформувалися чотири основні патерни розподілу обов'язків між інтерфейсом та логікою: Model-View-Controller (MVC), Model-View-Presenter (MVP), Model-View-ViewModel (MVVM) та Model-View-Intent (MVI). Кожен із цих патернів має власні особливості, переваги та обмеження, які визначають доцільність застосування у конкретних умовах. Порівняльну характеристику цих патернів подано в таблиці 2.1.

Таблиця 2.1 – Порівняльна характеристика патернів представлення

Критерій	MVC	MVP	MVVM	MVI
Зв'язність шару View з логікою	Висока	Середня	Низька	Низька
Тестовність бізнес-логіки	Низька	Середня	Висока	Висока
Однонаправленість потоку даних	Відсутня	Відсутня	Часткова	Повна
Кількість шаблонного коду	Низька	Висока	Середня	Середня
Підтримка незмінного стану	Відсутня	Відсутня	Часткова	Повна
Сумісність з декларативним UI	Низька	Низька	Висока	Найвища

Аналіз поданих у таблиці характеристик показує, що для застосунків, побудованих на декларативному підході, патерн MVI є найбільш доцільним вибором. Він базується на трьох ключових принципах: незмінному стані екрана (UiState), однонаправленому потоці даних та представленні дій користувача у формі дискретних намірів (Intent) [16]. Окрім того, MVI забезпечує передбачувану поведінку застосунку: для будь-якої комбінації стану та наміру існує єдиний результат, що значно спрощує налагодження та автоматизоване тестування.

На рівні загальної структури обрано Clean Architecture, запропоновану Р. Мартіном [17]: код розподілено на ізольовані шари з чітким правилом залежності — зовнішні шари можуть залежати від внутрішніх, але не навпаки. Це гарантує незалежність бізнес-логіки від конкретного фреймворку, бази даних чи інтерфейсу та узгоджується з гнучкими методологіями розроблення [18].

У межах розроблюваного застосунку Clean Architecture реалізовано у вигляді трьох шарів: шару представлення (presentation), шару бізнес-логіки (domain) та шару доступу до даних (data). Графічне зображення обраної архітектури з її компонентами подано на рисунку 2.1.

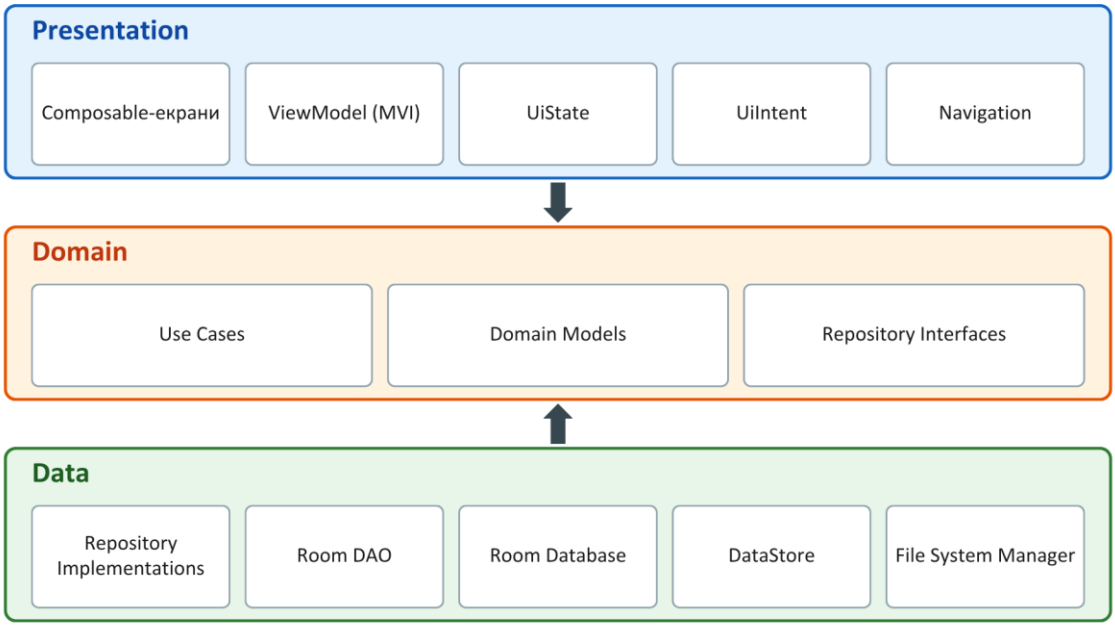


Рисунок 2.1 – Шарова архітектура застосунку

Шар представлення відповідає за відображення інформації та обробку дій користувача. У ньому розміщуються комоненти UI, що формують інтерфейс, та компоненти ViewModel. ViewModel приймає наміри від користувача (UiIntent), звертається до сценаріїв використання шару бізнес-логіки та оновлює незмінний об'єкт стану (UiState).

Шар бізнес-логіки містить доменні та сценарії використання, кожен з яких відповідає за одну бізнес-операцію. Шар реалізовано виключно обраною мовою програмування без залежностей від платформи Android, що забезпечує його платформну незалежність та повну тестовність. Доступ до зовнішніх ресурсів отримується через інтерфейси репозиторіїв.

Шар доступу до даних інкапсулює всі взаємодії із локальним сховищем: містить реалізації репозиторіїв, об'єкти доступу до бази даних (DAO), екземпляр бази даних, компоненти роботи з файловою системою та сховищем налаштувань.

Функціональну структуру побудовано за принципом групування коду навколо функціональних можливостей: кожен блок містить компоненти всіх трьох архітектурних шарів та взаємодіє зі спільним ядром через інтерфейси репозиторіїв. Узагальнену функціональну структуру зображено на рисунку 2.2.

Рисунок 2.2 – Функціональна структура застосунку

Обрана архітектура має низку суттєвих переваг. По-перше, чіткий розподіл обов'язків між шарами полегшує розуміння кодової бази та внесення змін. По-друге, ізольованість шару бізнес-логіки від платформних залежностей дозволяє покривати ключові алгоритми застосунку модульними тестами без необхідності використання інструментів інструментального тестування Android. По-третє, поєднання Clean Architecture з патерном MVI забезпечує передбачувану поведінку інтерфейсу: будь-яка зміна на екрані є результатом обробки конкретного наміру.

Водночас обрана архітектура має певні недоліки. Шарова структура та використання абстрактних інтерфейсів спричиняють появу значної кількості шаблонного коду. Окрім того, поріг входу для розробників, які не знайомі з принципами Clean Architecture, є вищим порівняно з іншими підходами. Однак ці витрати виправдовуються перевагами в підтримуваності та тестовності.

Отже, обрано архітектуру на основі Clean Architecture з трьома шарами (presentation, domain, data), доповнену патерном MVI на рівні представлення та функціонального групування коду. Така комбінація рішень забезпечує необхідний рівень модульності та тестовності для подальшої програмної реалізації.

2.2 Проєктування структури бази даних

Для забезпечення локального зберігання даних мобільного застосунку необхідно спроєктувати структуру бази даних, яка одночасно задовольняє трьом вимогам: відповідає функціональним сценаріям, визначеним у попередньому розділі; забезпечує цілісність даних на рівні самої БД; гарантує прийнятну продуктивність типових запитів.

Процес проєктування БД виконується у дві стадії: побудова логічної моделі, що описує сутності та зв'язки між ними у предметно-орієнтованих термінах, та побудова фізичної моделі, у якій логічна структура відображається на конкретні засоби обраної СКБД [19].

На основі аналізу функціональних вимог та діаграми варіантів використання, наведеної у підрозділі 1.3, виділено шість основних сутностей, які повинні зберігатися у локальній базі даних: чат, повідомлення, вкладення, тег, зв'язок повідомлення з тегом та нагадування. Для кожної сутності визначено перелік атрибутів та ключові поля, детальний опис сутностей наведено нижче.

Сутність «Чат» описує окрему тематичну категорію застосунку. Атрибутний склад сутності подано в таблиці 2.2.

Таблиця 2.2 – Атрибутний склад сутності «Чат»

Атрибут	Тип	Обов'язковість	Призначення
id	INTEGER	Так (PK)	Унікальний ідентифікатор чату
name	TEXT	Так	Назва чату
background_color	INTEGER	Так	Колір тла, кодований у форматі ARGB
icon_text	TEXT	Так	Емоджі або символ для візуальної ідентифікації
image_path	TEXT	Ні	Відносний шлях до фонового зображення у внутрішньому сховищі
order	INTEGER	Так	Позиція у списку чатів для ручного впорядкування
created_at	INTEGER	Так	Час створення чату у мілісекундах епохи Unix

Сутність «Повідомлення» є центральною у моделі. Окрім посилання на батьківський чат, вона підтримує опціональне посилання на повідомлення-першоджерело для функції відповідей та серіалізований перелік діапазонів форматування тексту. Атрибутний склад подано в таблиці 2.3.

Таблиця 2.3 – Атрибутний склад сутності «Повідомлення»

Атрибут	Тип	Обов'язковість	Призначення
id	INTEGER	Так (PK)	Унікальний ідентифікатор повідомлення
chat_id	INTEGER	Так (FK)	Посилання на батьківський чат
text	TEXT	Так	Текстовий вміст нотатки
is_pinned	INTEGER	Так	Ознака закріплення повідомлення у чаті

Кінець таблиці 2.3

is_checked	INTEGER	Так	Ознака виконання
is_opposite	INTEGER	Так	Ознака альтернативного вирівнювання у чаті
reply_to_message_id	INTEGER	Ні (FK)	Посилання на повідомлення-першоджерело
created_at	INTEGER	Так	Час створення у мілісекундах епохи Unix
styles	TEXT	Ні	Серіалізований перелік діапазонів форматування

Сутність «Вкладення» інкапсулює інформацію про мультимедійні файли. Набір метаданих залежить від типу вкладення: для зображень і відео фіксуються розміри в пікселях, для аудіо- та відеофайлів — тривалість, для документів — оригінальна назва файлу та MIME-тип. Необов'язкові атрибути заповнюються лише для відповідного типу вкладення. Атрибутний склад подано в таблиці 2.4.

Таблиця 2.4 – Атрибутний склад сутності «Вкладення»

Атрибут	Тип	Обов'язковість	Призначення
id	INTEGER	Так (PK)	Унікальний ідентифікатор вкладення
chat_id	INTEGER	Так	Посилання на чат
message_id	INTEGER	Так (FK)	Посилання на батьківське повідомлення
attached_at	INTEGER	Так	Час прикріплення
attachment_type	TEXT	Так	Тип вкладення (image, video, audio, file)
path	TEXT	Так	Відносний шлях до файлу у сховищі
file_name	TEXT	Ні	Оригінальна назва файлу
mime_type	TEXT	Ні	MIME-тип вмісту
file_size_in_bytes	INTEGER	Ні	Розмір файлу у байтах
duration_in_seconds	INTEGER	Ні	Тривалість для аудіо- та відеофайлів
width	INTEGER	Ні	Ширина зображення або відео у пікселях
height	INTEGER	Ні	Висота зображення або відео у пікселях

Сутність «Тег» описує користувацьку мітку для додаткової категоризації повідомлень. На відміну від чату, теги не утворюють ієрархії і можуть призначатися повідомленням у довільних комбінаціях. Це забезпечує гнучку систему категоризації, за якої одне повідомлення може одночасно належати до кількох тематичних груп. Атрибутний склад сутності подано в таблиці 2.5

Таблиця 2.5 – Атрибутний склад сутності «Тег»

Атрибут	Тип	Обов'язковість	Призначення
id	INTEGER	Так (PK)	Унікальний ідентифікатор тегу
name	TEXT	Так	Назва тегу
emoji	TEXT	Так	Емоджі-маркер
color	INTEGER	Так	Колір тегу у форматі ARGB

Сутність «Зв'язок повідомлення з тегом» є сполучною та реалізує відношення «багато до багатьох» через складений первинний ключ з пари зовнішніх ключів. Атрибутний склад подано в таблиці 2.6.

Таблиця 2.6 – Атрибутний склад сутності «Зв'язок повідомлення з тегом»

Атрибут	Тип	Обов'язковість	Призначення
message_id	INTEGER	Так (PK, FK)	Посилання на повідомлення
tag_id	INTEGER	Так (PK, FK)	Посилання на тег

Сутність «Нагадування» описує заплановану подію сповіщення для конкретного повідомлення. Реалізація зв'язку «один до одного» з повідомленням забезпечується унікальним індексом на полі message_id. Атрибутний склад подано в таблиці 2.7.

Таблиця 2.7 – Атрибутний склад сутності «Нагадування»

Атрибут	Тип	Обов'язковість	Призначення
id	INTEGER	Так (PK)	Унікальний ідентифікатор нагадування
message_id	INTEGER	Так (FK, унікальний)	Посилання на повідомлення
time_millis	INTEGER	Так	Запланований час спрацювання у мілісекундах

Між сутностями встановлено такі зв'язки. Один чат пов'язаний з багатьма повідомленнями (зв'язок «один до багатьох» через зовнішній ключ chat_id у таблиці повідомлень). Одне повідомлення може мати багато вкладень (зв'язок «один до багатьох» через message_id у таблиці вкладень). Одне повідомлення може мати не більше одного нагадування (зв'язок «один до одного», що

забезпечується унікальним індексом на message_id у таблиці нагадувань). Між повідомленнями та тегами існує зв'язок «багато до багатьох», реалізований через сполучну таблицю message_tags. Окремо виділено рекурсивний зв'язок «один до багатьох» у межах самої таблиці повідомлень для підтримки функції відповідей: кожне повідомлення може посилатися на інше повідомлення-першоджерело через поле reply_to_message_id.

Для забезпечення узгодженості даних на рівні бази визначено правила каскадної поведінки під час видалення. Видалення чату спричиняє каскадне видалення всіх його. Видалення повідомлення-першоджерела для відповіді не призводить до видалення відповіді — натомість поле reply_to_message_id встановлюється у NULL. Видалення тегу автоматично прибирає всі його зв'язки з повідомленнями завдяки каскадному правилу у сполучній таблиці.

Графічне представлення логічної моделі бази даних у вигляді ER-діаграми зображено на рисунку 2.3.

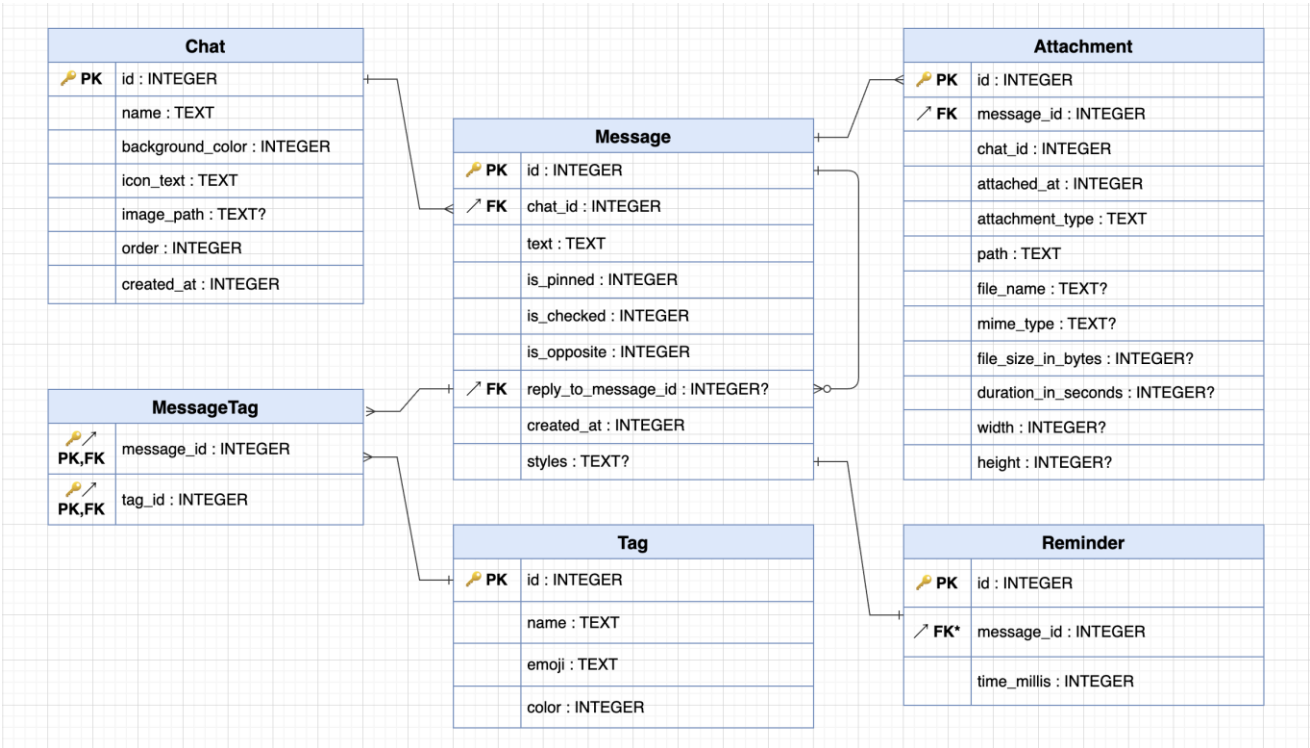


Рисунок 2.3 – Логічна модель бази даних (ER-діаграма)

Спроєктовану логічну модель було перевірено на відповідність першим трьом нормальним формам. Усі атрибути таблиць є атомарними, що відповідає першій нормальній формі. Кожен неключовий атрибут функціонально залежить від повного первинного ключа (у сполучній таблиці message_tags, що має складений ключ, відсутні власні неключові поля), що відповідає другій нормальній формі. Транзитивних залежностей між неключовими атрибутами не виявлено, що відповідає третій нормальній формі.

2.3 Проєктування інтерфейсу користувача

Інтерфейс користувача мобільного застосунку є основним каналом взаємодії між людиною та програмним продуктом, тому якість його проєктування безпосередньо впливає на досягнення головної мети роботи — мінімізації когнітивного навантаження під час фіксації нотаток. Проєктування інтерфейсу здійснюється з опорою на принципи Material Design 3 — офіційну мову дизайну платформи Android, що регламентує підхід до типографіки, кольорів, проміжків та поведінки інтерактивних елементів [20].

Під час проєктування дотримано чотирьох ключових принципів. Принцип ергономічності передбачає розташування основних елементів керування у нижній третині екрана — у зоні зручного доступу великого пальця руки [21]. Принцип візуальної ієрархії реалізовано підбором розміру, кольору та контрастності так, щоб увага природно спрямовувалася на стрічку повідомлень та поле введення. Принцип консистентності забезпечує, що однакові за сенсом дії викликаються однаковими жестами та мають однакову візуальну форму на всіх екранах [22]. Принцип відповідності патернам месенджерів означає, що компонування екрана чату наслідує розташування елементів, відоме користувачам із Telegram, WhatsApp та аналогічних застосунків [21].

Структуру навігації застосунку сплановано з мінімальною глибиною — переважна більшість сценаріїв використання виконується у межах одного-двох

переходів від головного екрана. Карту переходів між основними екранами подано на рисунку 2.4.

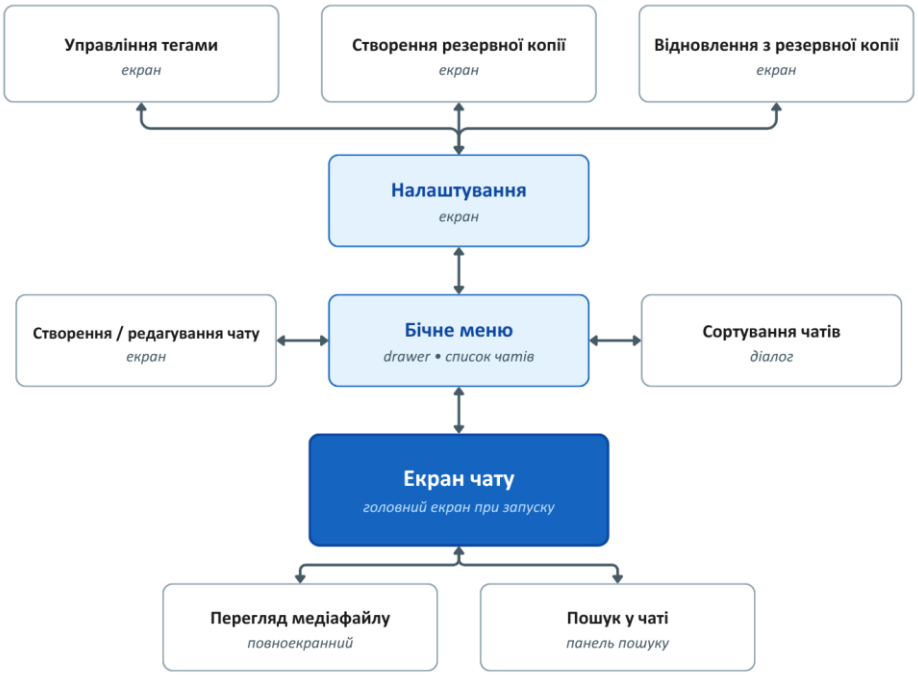


Рисунок 2.4 – Карта навігації застосунку

Як видно з карти, центральним вузлом навігації є екран чату, що відповідає одній із ключових проєктних вимог — при запуску застосунку користувач опиняється безпосередньо в останньому відкритому чаті без проміжних екранів, що мінімізує час між запуском застосунку та можливістю фіксації думки. Це рішення є свідомим відхиленням від типового патерну з головним екраном-списком на користь швидкості доступу до контенту.

Одним з головних екран застосунку, який являє собою список тематичних чатів, виконує функцію навігаційного хабу та відкривається у вигляді бічного висувного меню з екрана чату. У верхній частині розміщено пошукове поле та кнопку переходу до налаштувань. Під ним розташовано елементами керування сортуванням та кнопкою створення нового чату. Основну площу екрана займає список наявних чатів. Макет головного екрана зображено на рисунку 2.5.

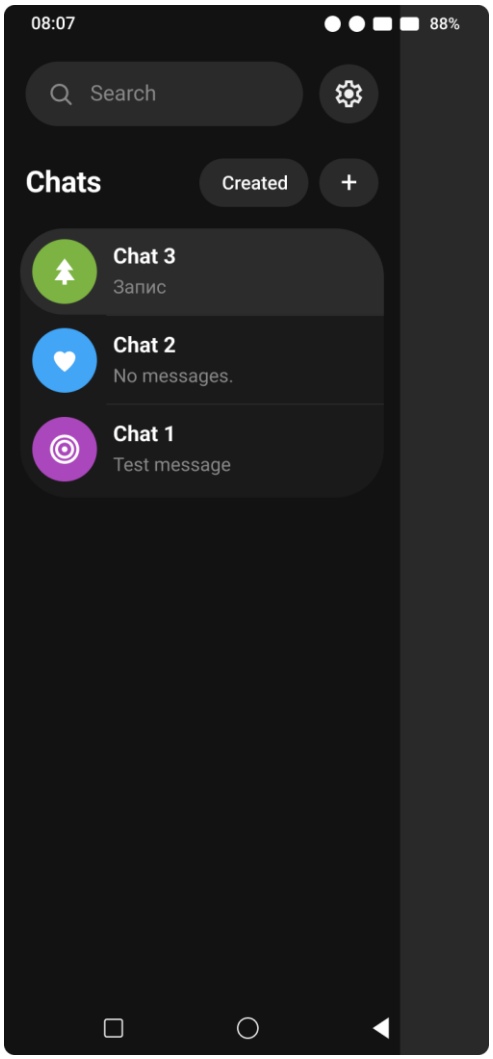


Рисунок 2.5 – Макет головного екрана зі списком чатів

Екран чату є основним робочим простором користувача та реалізує діалогову парадигму взаємодії, обґрунтовану в підрозділі 1.1. У верхній частині розташовано панель навігації з кнопкою меню, іконкою чату, його назвою, кнопками пошуку та виклику контекстного меню. Безпосередньо під панеллю передбачено опціональну смугу закріплених повідомлень, що залишається видимою під час прокручування. Центральну частину екрана займає стрічка повідомлень, згрупованих за днями створення. У нижній частині екрана розташовано панель введення з полем для тексту, кнопкою прикріплення файлів та кнопкою запису аудіоповідомлення, яка перетворюється на кнопку надсилання за наявності тексту в полі введення. Макет екрана чату подано на рисунку 2.6.

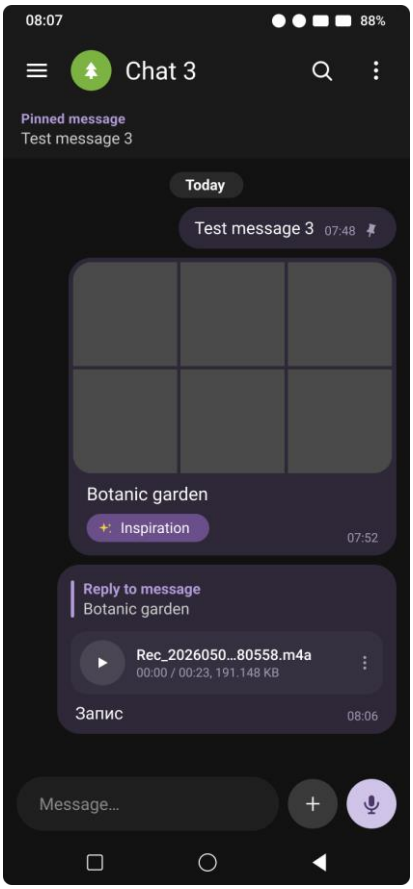


Рисунок 2.6 – Макет екрана чату з повідомленнями

Окрім двох головних екранів, у застосунку передбачено набір допоміжних інтерфейсів для управління даними та налаштуваннями, показаних на рисунку 2.7.

Екран створення або редагування чату (рисунок 2.7, а) містить великий круглий індикатор обраного візуального оформлення, дві кнопки керування «Upload Photo» та «Pick Emoji» для завантаження фонового зображення і вибору символу-іконки, поле введення назви чату, горизонтальну палітру з заздалегідь визначених кольорів та кнопку підтвердження «Done».

Екран налаштувань (рисунок 2.7, б) реалізовано у вигляді вертикального переліку згрупованих блоків, кожен з яких оформлено прямокутною карткою з заголовком та коротким описом дії. Блок «App Theme» містить сегментований перемикач з трьох опцій («Dark», «Light», «System») та чекбокс активації динамічних кольорів; наступні блоки керують переходом до управління тегами, створенням резервної копії з переліком наявних архівів та діалогом відновлення.

Екран управління тегами (рисунок 2.7, в) відображає вертикальний список наявних міток, де кожен елемент містить кольорову смугу-маркер, емоджі-іконку, назву тегу та кнопку видалення, а в нижньому правому куті розташовано плаваючу кнопку додавання нового тегу. Екран створення тегу за компонуванням аналогічний екрану створення чату, що відповідає принципу консистентності.

Окремо передбачено повноекранний режим перегляду мультимедійного контенту (рисунок 2.7, г), який активується при натисканні на зображення або відео у стрічці чату. У цьому режимі контент відображається на повну ширину екрана з мінімальною панеллю керування у верхній частині, що містить кнопку повернення, назву чату, кнопку поширення та контекстне меню.

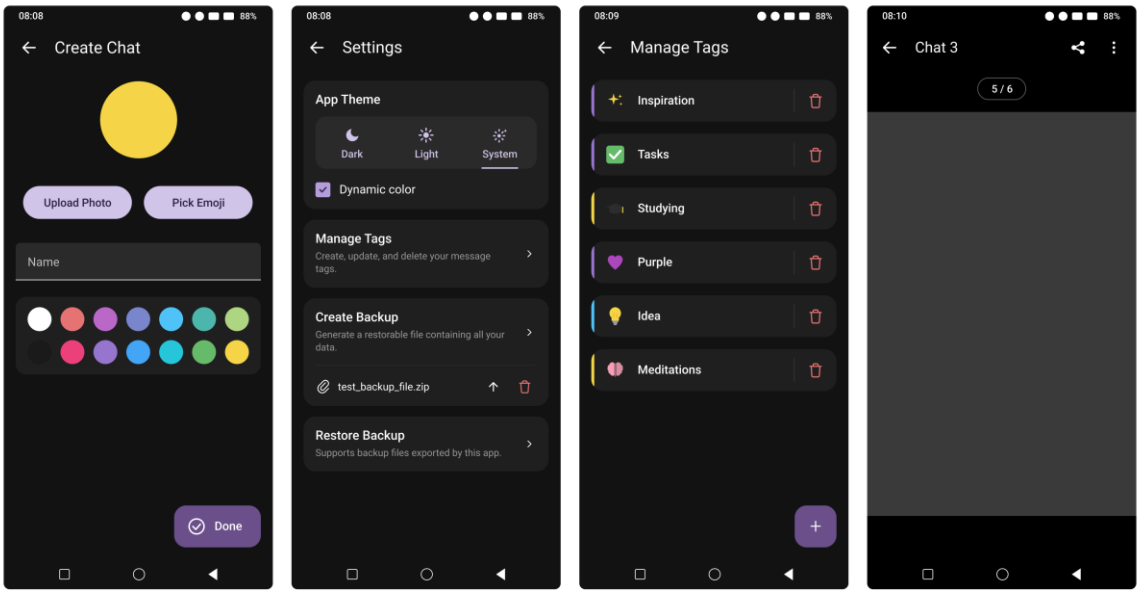


Рисунок 2.7 – Макети допоміжних екранів застосунку: а – створення чату; б – налаштування; в – управління тегами; г – повноекранний перегляд медіафайлів

Отже, спроєктований інтерфейс реалізує принципи мінімалізму та когнітивної економії: користувач отримує миттєвий доступ до основної функції без проміжних екранів, а вторинні функції організовано у логічну ієрархію з невеликою глибиною.

2.4 Розроблення алгоритму роботи мобільного застосунку

Алгоритмічна основа застосунку складається з послідовностей дій, які реалізують функціональні сценарії, визначені діаграмою варіантів використання у підрозділі 1.3. Кожен алгоритм описує впорядкований набір кроків, що повинен виконати застосунок для досягнення конкретного результату. Для повного розкриття алгоритмічної логіки в роботі описано три ключові процеси, які демонструють різні аспекти функціонування застосунку.

Першим серед розглянутих процесів є створення тематичного чату. Цей алгоритм активується при натисканні кнопки додавання у бічному меню списку чатів і завершується додаванням нового запису до бази даних та переходом на щойно створений чат. Особливість алгоритму полягає в обробці двох взаємовиключних варіантів візуального оформлення та валідації введення. Блок-схему алгоритму створення чату подано на рисунку 2.8.

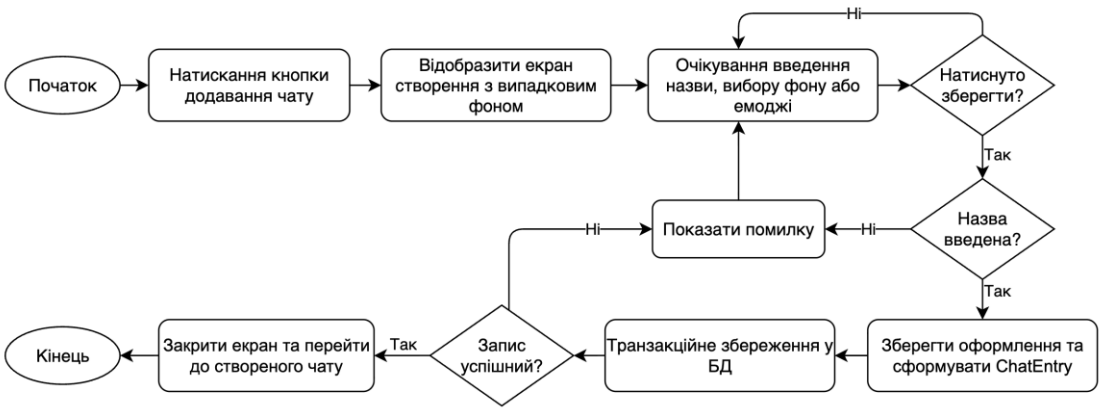


Рисунок 2.8 – Блок-схема алгоритму створення тематичного чату

Наступним розглянуто алгоритмом надсилання повідомлення з прикріпленням файлу. Особливістю алгоритму виступає транзакційне збереження повідомлення разом з усіма його вкладеннями, що гарантує атомарність операції — у разі помилки на будь-якому етапі база даних залишається у консистентному стані. Блок-схему алгоритму надсилання повідомлення наведено на рисунку 2.9.

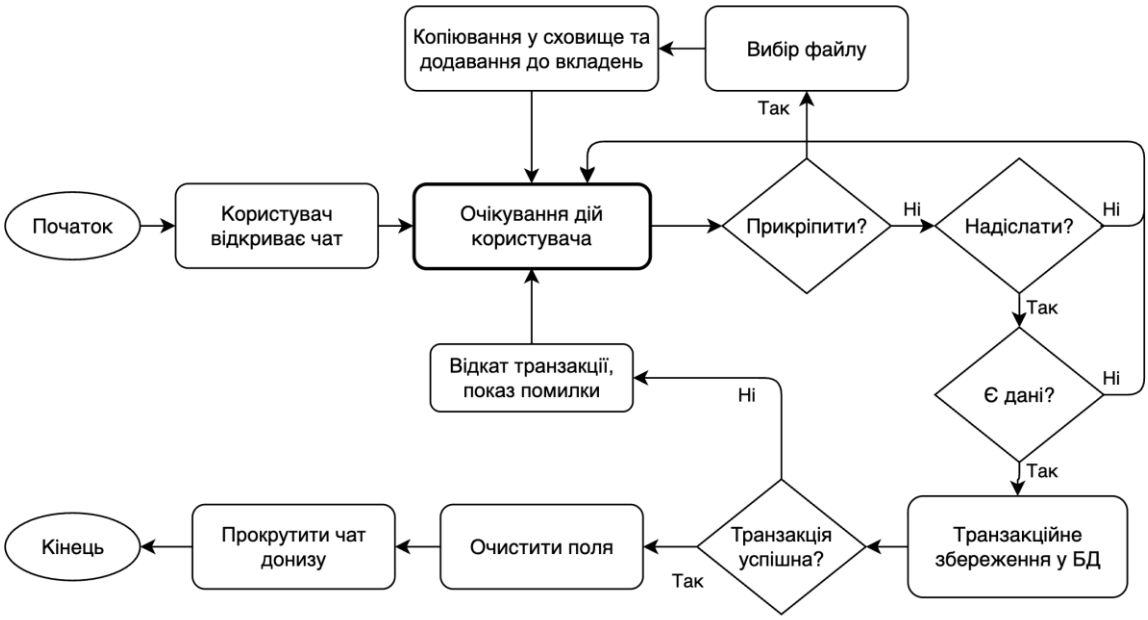


Рисунок 2.9 – Блок-схема алгоритму надсилання повідомлення з файлами

Завершальним серед розглянутих процесів є запис аудіоповідомлення, який демонструє роботу застосунку з апаратним забезпеченням пристрою, керування дозволом RECORD_AUDIO та повноцінне керування життєвим циклом запису з можливістю призупинення та продовження. Блок-схему алгоритму запису аудіоповідомлення подано на рисунку 2.10.

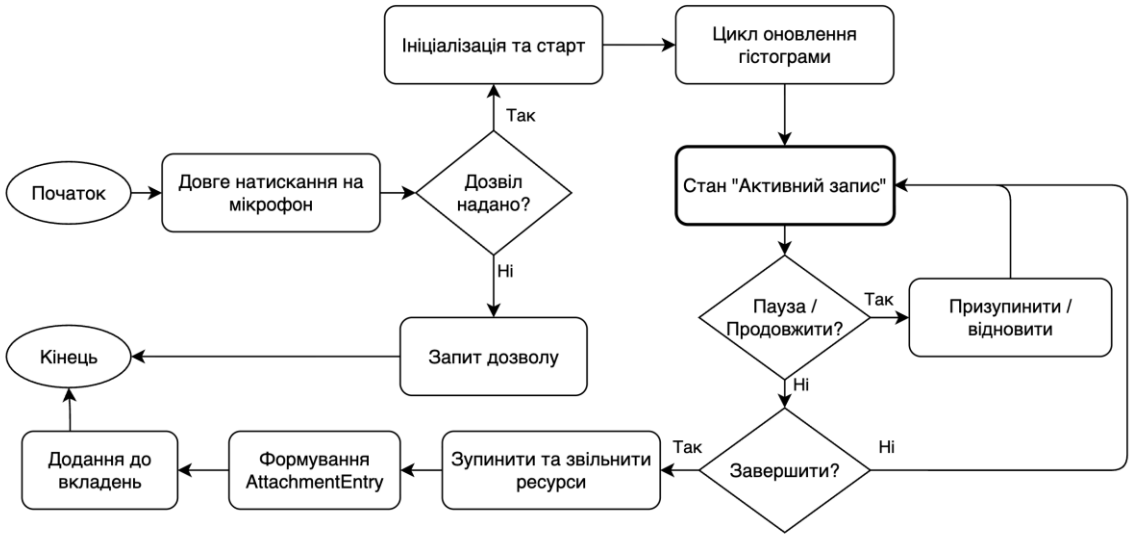


Рисунок 2.10 – Блок-схема алгоритму запису аудіоповідомлення

Розроблені алгоритми охоплюють три ключові сценарії функціонування застосунку та демонструють підходи до розв'язання типових задач мобільного розроблення, серед яких керування транзакціями бази даних, робота з файловою системою, обробка runtime-дозволів та інтеграція з апаратним забезпеченням.

2.5 Створення прототипу мобільного застосунку

На основі макетів інтерфейсу та карти навігації з підрозділу 2.3 побудовано повноколірний динамічни прототип, що візуалізує екрани у їх остаточному вигляді, фіксує узгодженість оформлення між екранами та точки переходу між ними. Це усуває потребу у прийнятті дизайн-рішень під час написання коду.

Як інструмент прототипування обрано хмарний редактор Figma — він спеціалізується на проєктуванні інтерфейсів, підтримує повторно використовувані компоненти та автоматичне компонування, надає бібліотеку ресурсів за специфікацією Material Design 3 і має безкоштовний тарифний план.

Прототип створювався у такій послідовності. Спочатку визначено колірну палітру з основних та акцентних кольорів за специфікацією Material Design 3 та сформовано типографічну. На основі цих систем побудовано повторно використовувані компоненти, що забезпечило узгодженість однотипних елементів між екранами.

Далі виконано повноколірне оформлення всіх екранів. Кадри заповнено реалістичним демонстраційним вмістом, що дозволило оцінити поведінку інтерфейсу за умов, наближених до експлуатаційних.

Завершальним кроком стало об'єднання окремих екранів у динамічний прототип шляхом встановлення зв'язків переходу між ними. У результаті отримано цілісну модель, у якій можна простежити сценарії використання, передбачені діаграмою варіантів використання у підрозділі 1.3, без написання програмного коду. Загальний вигляд прототипу з розташуванням екранів та зв'язками переходу між ними подано на рисунку 2.11.

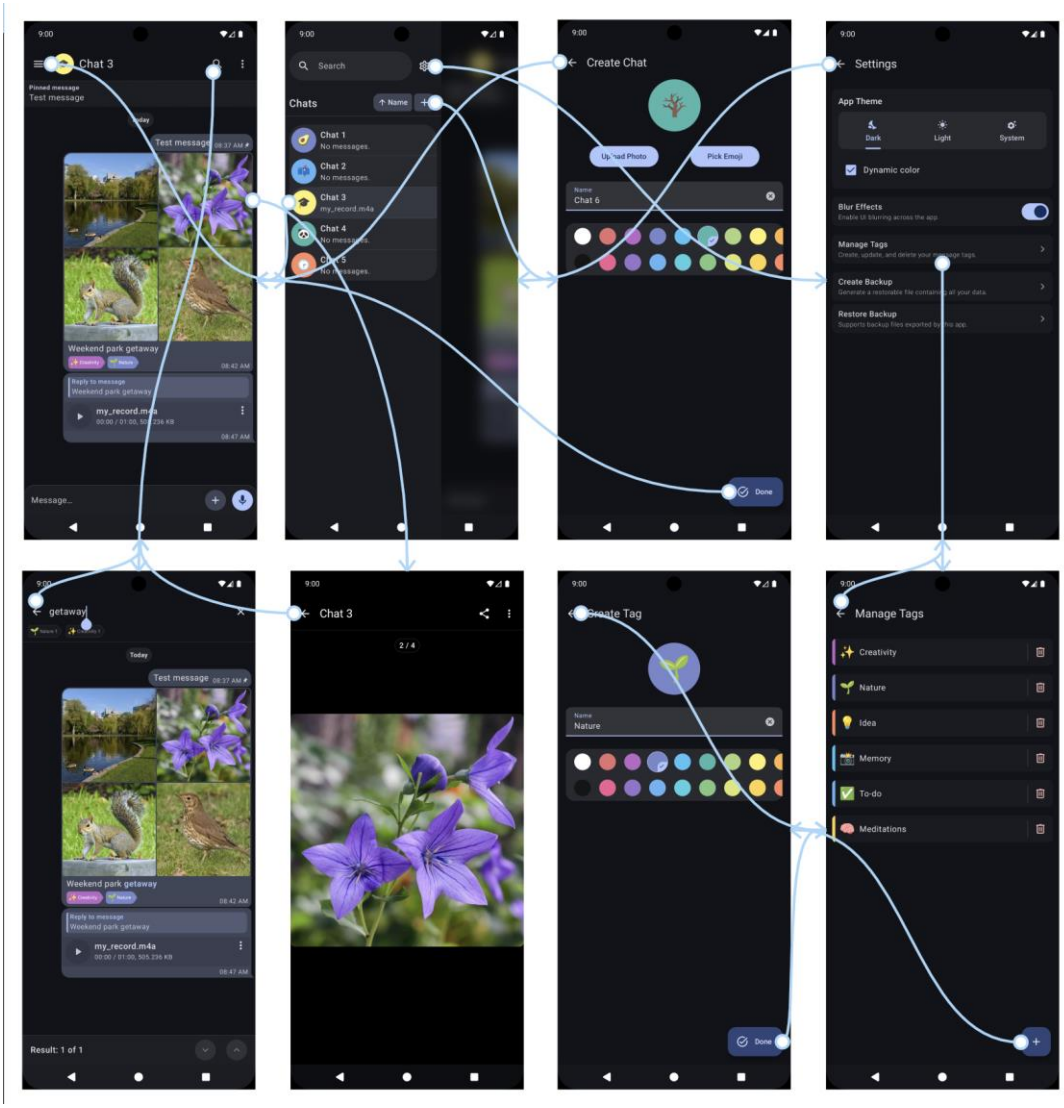


Рисунок 2.11 – Загальний вигляд прототипу в середовищі Figma

Топологія прототипу мінімізує середню кількість переходів, необхідних для досягнення будь-якої функції застосунку, та відповідає карті навігації з підрозділу 2.3. Прототип у подальшому використовуватиметься як еталонне візуальне джерело під час реалізації розмітки інтерфейсу.

2.6 Аналіз та вибір технологій і методів реалізації застосунку

У даному підрозділі здійснюється порівняльний аналіз технологічних альтернатив для платформи Android та обґрунтування вибір засобів, що відповідають спроєктованій архітектурі.

Цільову операційну систему задано темою кваліфікаційної роботи. Мінімальною підтримуваною версією обрано Android 8.0, що охоплює переважну більшість активно використовуваних пристроїв та дозволяє застосовувати сучасні API без втрати значної частини потенційної аудиторії.

Серед мов, які офіційно підтримуються для розроблення під Android, до розгляду взято Java та Kotlin — решта мов потребує проміжних кросплатформних фреймворків. Java має тривалу історію використання та обширну екосистему бібліотек, проте характеризується значним обсягом шаблонного коду, відсутністю вбудованих засобів захисту від нульових посилань та обмеженою виразністю для опису незмінних структур даних. Натомість Kotlin із 2019 року має офіційний статус пріоритетної мови для Android. Його переваги — компактний синтаксис, вбудована система обмеження нульових посилань, підтримка корутин на рівні мови [23], розвинений набір засобів функціонального програмування та повна сумісність із байт-кодом Java. Зважаючи на перераховані переваги та офіційний пріоритетний статус мови, для подальшої реалізації обрано мову Kotlin.

Як інструментарій побудови інтерфейсу на платформі Android розглянуто два підходи — традиційний імперативний на основі ієрархії об'єктів класу View з описом розмітки в XML-файлах та декларативний Jetpack Compose [24]. Імперативний підхід має тривалу історію та обширний набір готових компонентів, проте вимагає ручного зв'язування елементів керування з кодом і покладає синхронізацію між станом моделі та інтерфейсом на розробника, що часто призводить до помилок. Jetpack Compose описує інтерфейс композиційними функціями, які отримують стан як параметр та повертають візуальне представлення; цей інструментарій природно інтегрується з Kotlin, автоматично перерахову лише змінені частини інтерфейсу, має вбудовану підтримку Material Design 3 та глибоку сумісність із корутинами і Flow. З огляду на узгодженість з обраною мовою та концепцією інтерфейсу з підрозділу 2.3 для побудови інтерфейсу обрано Jetpack Compose.

Наступним кроком обрано інтегроване середовище розроблення. До розгляду взято Android Studio та IntelliJ IDEA Community Edition, побудовані на спільній платформі IntelliJ із повноцінною підтримкою Kotlin. У IntelliJ IDEA Community Edition специфічні для Android інструменти (графічний редактор макетів, попередній перегляд композиційних функцій, інспектор схем Room, профайлер ресурсів, емулятори пристроїв) або відсутні, або потребують додаткових модулів. Android Studio є офіційним середовищем розроблення під Android з повним набором цих інструментів. З огляду на повну орієнтованість проєкту на Android та активне використання Jetpack обрано Android Studio.

Для реалізації принципу інверсії залежностей, обґрунтованого у підрозділі 2.1, серед Kotlin-фреймворків впровадження залежностей розглянуто Koin та Hilt. Koin формує контейнер залежностей під час виконання й має простий API без генерації коду, проте помилки конфігурації виявляються лише після запуску застосунку. Hilt побудовано на основі Dagger, граф залежностей формується на стадії компіляції, що дає змогу виявляти некоректне зв'язування ще під час збірки; додатково Hilt має офіційний статус у складі Android Jetpack та вбудовану інтеграцію з ViewModel. З огляду на раннє виявлення помилок та офіційний статус обрано Hilt.

Основою асинхронного виконання має стати засіб, що дозволяє виконувати операції з базою даних, медіафайлами та резервними копіями без блокування потоку інтерфейсу. До розгляду взято дві альтернативи — RxJava та Kotlin Coroutines. RxJava має зрілу екосистему операторів та тривалу історію застосування у промислових проєктах, проте характеризується значною кривою навчання, потребує ретельного керування підписками для уникнення витоків ресурсів та збільшує обсяг бінарного коду. Kotlin Coroutines інтегровані безпосередньо у мову на рівні синтаксису й реалізують концепцію структурованої конкурентності, за якої кожна асинхронна операція автоматично припиняється під час руйнування своєї області видимості [25], що усуває цілий клас витоків

ресурсів. З огляду на офіційну рекомендацію Google для нових Android-проєктів та узгодженість з обраною мовою обрано Kotlin Coroutines.

Для відображення зображень, які користувач долучає до повідомлень, до розгляду взято три бібліотеки — Picasso, Glide та Coil. Picasso перебуває у стані обмеженого розвитку та не отримує суттєвих оновлень. Glide є зрілим і продуктивним рішенням, проте побудований на основі Java для імперативного інтерфейсу та оперує моделлю колбеків, яка не узгоджується з декларативним підходом Jetpack Compose. Coil розроблений мовою Kotlin із застосуванням корутин, безшовно інтегрується з композиційними функціями та має менший розмір бінарного коду; третя версія додає підтримку мультиплатформної моделі, актуальних форматів та сучасного мережевого клієнта Ktor [26]. З огляду на узгодженість з декларативним інтерфейсом обрано Coil третьої версії.

Для відтворення відео та аудіо вкладень розглянуто вбудований клас MediaPlayer та бібліотеку AndroidX Media3. MediaPlayer не потребує додаткових залежностей, але підтримує обмежений набір контейнерних форматів, не пристосований до адаптивного потокового відтворення та має складну модель станів. AndroidX Media3 є офіційним наступником ExoPlayer і об'єднує декілька раніше окремих гілок медіа-API в одну послідовну модель [27], покриває локальне та потокове відтворення, підтримує сучасні формати й містить готові компоненти інтерфейсу для Jetpack Compose. З огляду на необхідність якісного відтворення медіафайлів обрано AndroidX Media3.

Отже, прийняті рішення утворюють сучасний та офіційно підтримуваний технологічний стек, орієнтований на платформу Android 8.0 та новіші версії, де перевагу надано компонентам із офіційним статусом у складі Android Jetpack або з активним розвитком. Єдина мовна основа Kotlin, єдиний механізм асинхронності у вигляді корутин та єдиний декларативний інструментарій Jetpack Compose забезпечують внутрішню узгодженість стека та готовність кодової бази до тривалого супроводу.

2.7 Висновки

У межах розділу виконано проєктування мобільного застосунку, результати якого є основою для його подальшої програмної реалізації. На початковому етапі обрано та обґрунтовано архітектурний підхід, після чого побудовано загальну структуру застосунку та принципи взаємодії між її складовими частинами. Прийняті архітектурні рішення забезпечують необхідний рівень модульності та розділення відповідальності між компонентами.

Враховуючи потребу у зберіганні даних на пристрої користувача, окрему увагу приділено проєктуванню локальної бази даних: визначено перелік сутностей, склад їх полів та формати збереження інформації, а також встановлено міжтабличні зв'язки разом із правилами забезпечення цілісності.

Наступним кроком за допомогою відповідних інструментів сформовано макети інтерфейсу, при розробленні яких бралися до уваги особливості цільової аудиторії та умови комфортного користування застосунком. На базі готових макетів побудовано інтерактивний прототип, що в подальшому має послугувати орієнтиром під час реалізації екранних відображень.

Спираючись на сформульовані функціональні вимоги, розроблено основні алгоритми роботи застосунку, що відображають послідовність кроків для досягнення відповідних результатів. Графічне представлення алгоритмів виконано у вигляді блок-схем.

Завершальною частиною роботи над розділом став порівняльний розгляд технологічних альтернатив, за результатами якого сформовано стек засобів реалізації. Поетапно визначено мову програмування, середовище розроблення, інструментарій побудови інтерфейсу, фреймворк впровадження залежностей, механізм асинхронної обробки та низку допоміжних бібліотек, що разом утворюють цілісну технологічну основу. Поставлені на початку розділу задачі проєктування виконано у повному обсязі. Сукупність отриманих проєктних артефактів формує вичерпну основу для переходу до етапу програмної реалізації.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Реалізація логіки мобільного застосунку

Програмну реалізацію виконано відповідно до архітектурних рішень підрозділу 2.1. Кожен екран працює як самостійна одиниця: отримує дії користувача, перетворює їх на бізнес-операції, виконує ці операції у фоновому потоці та повертає до інтерфейсу оновлений стан.

Центральною складовою логіки кожного екрана є тримач стану — клас, що наслідується від ViewModel із бібліотеки Jetpack [15]. Усередині тримача оголошується об'єкт стану — незмінний клас даних, поля якого визначають візуальне відображення екрана; об'єкт зберігається у приватній змінній типу MutableStateFlow, а зовні експонується як незмінний потік StateFlow [28]. Кожна дія користувача обробляється окремим публічним методом тримача, який або синхронно оновлює об'єкт стану, або запускає фонову корутину для асинхронних операцій з подальшим оновленням стану [29]. Окрім стану, тримач експонує потік MutableSharedFlow для одноразових подій інтерфейсу — повідомлень про помилки валідації або збереження.

Описані принципи розглянуто на прикладі тримача стану екрана створення та редагування чату — класу AddEditChatViewModel.

Стан екрана описується класом AddEditChatUiState із сімома полями та двома обчислюваними властивостями. Поле originalChat зберігає завантажений з бази даних чат у режимі редагування або null у режимі створення — за ним тримач визначає сценарій збереження. Поле currentDialog відображає активний модальний діалог. Поля chatName, chatColor та chatIconText зберігають введені значення відповідно до полів сутності «Чат». Поле chatImageState утримує стан фонового зображення у вигляді запечатаного типу, опис якого подано далі. Поле isLoading керує відображенням індикатора збереження. Властивість hasUnsavedChanges повертає істину, якщо хоча б одне поле стану відрізняється від

оригінального чату або в режимі створення введено непорожнє значення; вона використовується для показу діалогу підтвердження виходу. Властивість hasImage повертає істину за наявності активного зображення. Структуру стану подано нижче.

Код стану «AddEditChatUiState»:

```
data class AddEditChatUiState(
    val originalChat: ChatPresentationModel?,
    val currentDialog: AddEditChatDialog?,
    val chatName: String,
    val chatColor: Color,
    val chatIconText: String,
    val chatImageState: ChatImageState,
    val isLoading: Boolean
) {
    val hasUnsavedChanges = hasUnsavedChanges()
    val hasImage = chatImageState is ChatImageState.NewlyAdded
            || chatImageState is ChatImageState.Replaced
            || chatImageState is ChatImageState.Existing
}
```

У блоці ініціалізації тримач звертається до SavedStateHandle для отримання ідентифікатора чату, переданого через параметр навігації. Якщо цей ідентифікатор присутній, тримач асинхронно завантажує відповідний запис з бази даних через сценарій GetChatByIdUseCase та переносить значення його полів до об'єкта стану — таким чином екран автоматично переходить у режим редагування. Якщо ідентифікатор відсутній, екран залишається в режимі створення з початково ініціалізованими значеннями. Такий підхід дозволяє використовувати один і той самий екран як для створення, так і для редагування.

Тримач містить набір методів-мутаторів, кожен з яких відповідає одній дії користувача та виконує атомарне оновлення одного поля стану. Усі вони реалізовані ідентично — через виклик розширення update для типу MutableStateFlow з передаванням лямбди, що повертає копію поточного стану зі

зміненим полем; такий підхід гарантує атомарність навіть за конкурентного доступу з різних корутин. Приклад одного з методів подано нижче.

Код методу «updateChatName»:

```
fun updateChatName(name: String) {
    _uiState.update { uiState ->
        uiState.copy(chatName = name)
    }
}
```

Найскладнішим методом тримача є saveChat. Він запускається у фоновій корутині та починається з валідації введеної назви: метод validateChatNameWithError емітує код помилки в потік помилок і повертає null, що завершує виконання. У разі успішної валідації основний блок обгортається функцією timedLoadingTask. Послідовно виконуються три кроки: завантаження зображення до внутрішнього сховища, формування доменної моделі InsertChatDomainModel та виклик UpsertChatUseCase. Результат обробляється методом fold: у разі успіху запускається видалення замінених або скасованих файлів зображення, відправляється аналітична подія та викликається onSuccess з ідентифікатором збереженого чату для навігації на новий чат; у разі помилки тримач видаляє завантажене зображення та надсилає повідомлення про помилку.

Сценарії використання реалізовані як класи з єдиним публічним методом invoke (з модифікатором operator), що дозволяє викликати сценарій за іменем класу. Кожен сценарій інкапсулює одну бізнес-операцію та отримує залежності — абстрактні інтерфейси репозиторіїв — через конструктор. Використання абстрактних інтерфейсів забезпечує незалежність бізнес-логіки від деталей збереження даних та спрощує написання модульних тестів. UpsertChatUseCase реалізує об'єднану операцію створення або оновлення чату: за ідентифікатором більшим за нуль викликається метод оновлення репозиторію, інакше — метод створення, що повертає ідентифікатор нового запису. Реалізацію подано нижче.

Код сценарію «UpsertChatUseCase»:

```
class UpsertChatUseCase @Inject constructor(
    private val chatRepository: ChatRepository
) {
    suspend operator fun invoke(chat: InsertChatDomainModel): Result<Long> {
        return if (chat.id > 0) {
            chatRepository.updateChat(chat).map { chat.id }
        } else {
            chatRepository.createChat(chat)
        }
    }
}
```

Такий підхід об'єднання двох операцій в одному сценарії дозволяє тримачу стану екрана не розрізняти режим створення та режим редагування на рівні викликів — обидві гілки коду, описані раніше, використовують один і той самий сценарій, що зменшує дублювання логіки.

Решта екранів застосунку реалізовані за описаним вище принципом: кожному з них відповідає окремий тримач стану з власним класом стану та набором публічних методів-мутаторів, кожен з яких звертається до відповідних сценаріїв використання та оновлює поля стану.

3.2 Реалізація розмітки мобільного застосунку

Після того, як було розроблено логіку роботи мобільного застосунку, необхідним є реалізувати його розмітку, для чого слід скористатися раніше створеними макетами інтерфейсу користувача та прототипами, побудованими на основі цих макетів. На цьому етапі прийняті проєктні рішення переносяться у програмний код з використанням обраного у підрозділі 2.6 декларативного інструментарію Jetpack Compose.

Jetpack Compose дозволяє описувати розмітку мовою Kotlin у вигляді композиційних функцій. Така функція позначається анотацією @Composable,

отримує поточний стан як вхідні параметри та визначає, які елементи інтерфейсу і з якими властивостями повинні бути показані на екрані [30]. При зміні переданого стану фреймворк автоматично перебудовує лише ті частини інтерфейсу, для яких ця зміна є необхідною. Такий механізм вибіркової рекомпозиції забезпечує високу продуктивність інтерфейсу навіть за частих оновлень стану. Він є одним із ключових архітектурних переваг Jetpack Compose порівняно з підходом на основі XML.

Кожний екран застосунку реалізовано як окрема композиційна функція, що отримує контролер навігації та тримач стану як параметри. Розмітка усіх екранів відповідає єдиній зовнішній структурі, побудованій на основі компонента Scaffold зі специфікації Material Design 3 [20]. Цей компонент є типовим контейнером верхнього рівня та надає чотири іменовані слоти для розміщення стандартних частин екрана. Слоти верхньої панелі, нижньої панелі, плаваючої кнопки та основного вмісту є стандартними точками розширення, що охоплюють типові потреби більшості екранів. Використання Scaffold усуває необхідність вручну керувати відступами між системними панелями та вмістом екрана.

Для зображення прикладів реальної розмітки розглянемо реалізацію екрана створення та редагування чату — функції AddEditChatScreen, яка одночасно ілюструє типові рішення для решти екранів застосунку.

На початку функції з тримача стану отримується актуальний об'єкт стану через виклик collectAsStateWithLifecycle(), що забезпечує автоматичне відписування від потоку при знищенні екрана. Далі через ефект LaunchedEffect налаштовується підписка на потік разових подій помилок. Окремий обробник системної кнопки повернення BackHandler перехоплює натискання.

Основна структура екрана описується в межах компонента Scaffold. У слот плаваючої кнопки дії передається компонент ExtendedFloatingActionButton, натискання на яку викликає метод saveChat тримача стану. У слот верхньої панелі передається компонент TopAppBar.

Основна область вмісту реалізована через контейнер ConstraintLayout, який дозволяє декларативно описати взаємне розташування елементів за допомогою обмежень. На відміну від простіших контейнерів Column та Row, що розміщують вкладені елементи лінійно, ConstraintLayout надає можливість прив'язати кожен елемент до меж батьківського контейнера або інших елементів та змінювати ці прив'язки залежно від орієнтації пристрою.

Усередині ConstraintLayout розміщується п'ять елементів. ChatIcon — кругла іконка попереднього перегляду, яка відображає обраний колір тла, текст-емоджі або фонове зображення відповідно до полів тримача стану. ImagePickerButton — кнопка завантаження або видалення фонового зображення, поведінка якої залежить від поля hasImage. EmojiPickerButton — кнопка відкриття нижнього модального листа з категоризованим переліком символів та полем пошуку. ChatNameField — однорядкове поле введення назви чату на основі компонента TextField з кінцевою іконкою очищення. ChatColorsSection — горизонтальна палітра із заздалегідь визначених кольорів, натискання на які викликає метод updateChatColor та одразу відображає колір на іконці попереднього перегляду.

Адаптивна поведінка розмітки досягається перевіркою орієнтації пристрою через значення LocalConfiguration.current.orientation на початку області вмісту. У ландшафтній орієнтації елементи розташовуються у двох вертикальних колонках. У портретній орієнтації застосовується вертикальне компонування, у якому елементи слідують один за одним зверху донизу. Така адаптивність відповідає принципам Material Design 3 щодо підтримки усіх форм-факторів пристроїв.

Решта екранів застосунку — реалізовані за тим самим принципом: композиційна функція з контейнером Scaffold верхнього рівня, отримання стану з тримача через collectAsStateWithLifecycle та опис основного вмісту у відповідному контейнері-розмітці. Така одноманітність структури спрощує супровід кодової бази та забезпечує єдину поведінку усіх екранів стосовно орієнтації, екранної клавіатури та системних панелей.

Повний текст композиційних функцій подано у додатку Б.

3.3 Розроблення бази даних

Після того, як було реалізовано логіку та розмітку мобільного застосунку, наступним кроком є реалізація локального сховища даних на основі логічної моделі бази даних, спроєктованої у підрозділі 2.2. Завданням цього етапу є перенесення спроєктованої структури сутностей у програмний код у вигляді конкретних класів та інтерфейсів.

Як засіб реалізації локальної бази даних обрано бібліотеку Room — офіційну абстракцію над вбудованою у платформу Android системою керування базами даних SQLite [31]. Бібліотека Room надає шар об'єктно-реляційного відображення (ORM), який звільняє розробника від ручного формування SQL-запитів для типових операцій, перевіряє коректність запитів на етапі компіляції та автоматично генерує допоміжний код для перетворення між реляційними рядками таблиць та об'єктами мови Kotlin. Крім того, бібліотека має нативну інтеграцію з реактивними потоками Flow та з механізмом постраничного завантаження даних Paging 3 [32], що відповідає обраному у підрозділі 2.1 архітектурному рішенню.

Робота з базою даних в обраній бібліотеці передбачає наявність трьох основних компонентів: сутностей (Entity), які описують структуру окремих таблиць; об'єктів доступу до даних (Data Access Object), які описують операції над цими таблицями; та абстрактного класу самої бази даних, який об'єднує сутності та DAO у єдину одиницю та керує процесом створення фізичного файла бази даних на пристрої.

Для опису кожної таблиці бази даних створено окремий клас даних мови Kotlin, позначений анотацією @Entity з обов'язковим атрибутом tableName, який задає ім'я таблиці у фізичному файлі. Поля класу позначаються анотацією @ColumnInfo для явного зіставлення імен властивостей класу з іменами стовпців таблиці. Первинний ключ кожної сутності позначається анотацією @PrimaryKey з параметром autoGenerate = true, що делегує генерацію значень ідентифікатора самій базі даних. Розглянемо сутність чату, фрагмент опису якої подано нижче.

Код сутності «Чат»:

```
@Entity(
    tableName = "chats",
    indices = [Index(value = ["order"])]
)
data class ChatEntry(
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "id")
    val id: Long,
    @ColumnInfo(name = "name")
    val name: String,
    @ColumnInfo(name = "background_color")
    val backgroundColor: Int,
    @ColumnInfo(name = "image_path")
    val imageRelativePath: String?,
    @ColumnInfo(name = "order")
    val order: Long,
    @ColumnInfo(name = "created_at")
    val createdAt: Long
)
```

У наведеному прикладі параметр indices оголошує додатковий індекс за полем порядку, що прискорює сортування переліку чатів. Для сутностей, які перебувають у відношеннях з іншими сутностями, додатково оголошуються зовнішні ключі через параметр foreignKeys анотації @Entity. Так, сутність повідомлення містить два зовнішні ключі: один прив'язує повідомлення до чату, а другий — до повідомлення-джерела для повідомлень-відповідей.

Для виконання операцій над таблицями кожній сутності відповідає окремий інтерфейс об'єкта доступу до даних, позначений анотацією @Dao. Методи цих інтерфейсів реалізують конкретні запити до бази даних та можуть належати до однієї з трьох категорій: методи зі стандартними анотаціями @Insert, @Update та @Delete, для яких бібліотека самостійно генерує SQL-запити; методи з анотацією @Query, тіло яких містить явний SQL-запит; та методи з анотацією @Transaction, які об'єднують декілька операцій в єдину атомарну транзакцію.

Розглянемо приклад інтерфейсу доступу до таблиці чатів, скорочений фрагмент якого подано нижче.

Код інтерфейсу «ChatDao»:

```
@Dao
interface ChatDao {
    @Transaction
    suspend fun createChat(chat: ChatEntry): Long {
        val maxOrder = getMaxOrder() ?: -1L
        val chatWithOrder = chat.copy(id = 0, order = maxOrder + 1)
        return insertChat(chatWithOrder)
    }

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insertChat(chat: ChatEntry): Long

    @Update
    suspend fun updateChat(chat: ChatEntry): Int

    @Query("SELECT MAX(`order`) FROM chats")
    suspend fun getMaxOrder(): Long?

    @Transaction
    @Query("SELECT * FROM chats WHERE id = :chatId")
    fun getChatByIdFlow(chatId: Long): Flow<ChatWrapper?>
}
```

Особливістю наведеного інтерфейсу є метод createChat, який оголошено з анотацією @Transaction та містить тіло на мові Kotlin замість простого SQL-запиту. Перед вставленням нового запису метод обчислює максимальне значення поля порядку серед наявних чатів та призначає новому чату наступне за ним значення, що забезпечує природне впорядкування за часом створення. Усі складові операції виконуються в межах однієї транзакції, тому навіть у разі несподіваного завершення процесу база даних залишається в узгодженому стані. За тим самим принципом побудовано інтерфейс доступу до таблиці повідомлень,

який реалізує транзакційні операції надсилання та редагування повідомлень разом з їхніми вкладеннями, а також інтерфейси доступу до решти таблиць.

Усі сутності та об'єкти доступу до даних об'єднуються в єдиний абстрактний клас, що наслідується від базового класу RoomDatabase та позначається анотацією @Database. У параметрах цієї анотації перераховуються всі сутності, поточна версія схеми та спеціальні проекції. Створення та доступ до бази даних реалізовано за патерном Singleton з подвійною перевіркою блокування.

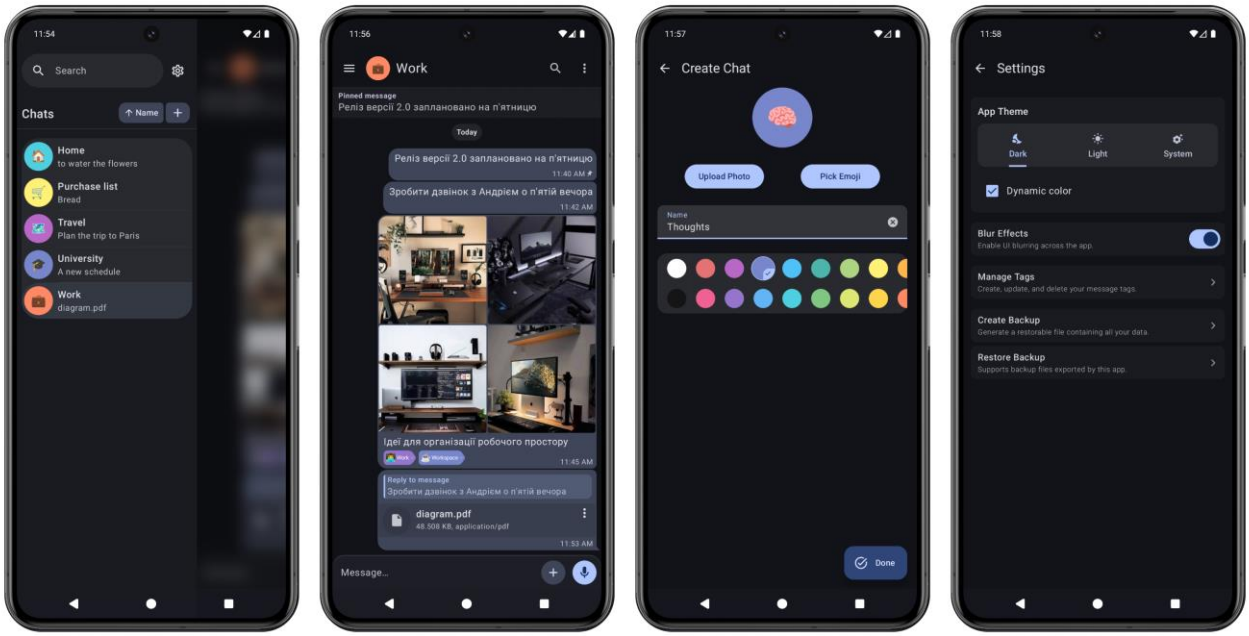
Схема бази даних підтримує механізм міграцій, який дозволяє оновлювати її структуру між версіями застосунку без втрати збережених користувачем даних. Це є критично важливою вимогою для застосунків із локальним збереженням, де відновлення втрачених даних є неможливим. Поточна версія схеми позначена у параметрах анотації @Database та слугує відправною точкою для майбутніх міграцій. Реалізований підхід до керування версіями схеми відповідає офіційним рекомендаціям бібліотеки Room.

Повний код сутностей, інтерфейсів доступу до даних, класу бази даних, та інших компонентів подано у додатку Б.

3.4 Керівництво користувача

Перед першим запуском користувач встановлює застосунок на пристрої під управлінням операційної системи Android. Реєстрація, вхід до облікового запису чи підтвердження адреси електронної пошти не передбачені — застосунок працює виключно з локальними даними та не має серверної частини, що гарантує повну приватність даних і доступність застосунку без підключення до мережі Інтернет.

При першому запуску користувачеві відображається порожній екран чату з пропозицією створити перший тематичний чат. При кожному наступному запуску застосунок одразу відкриває останній використовуваний чат, що мінімізує кількість дій до моменту фіксації нової нотатки. Загальний вигляд основних екранів застосунку подано на рисунку 3.1.



а б в г

Рисунок 3.1 – Основні екрани застосунку: а – Переліку чатів; б – екран чату; в – екран створення чату; г – екран налаштувань

Перелік усіх створених чатів реалізовано у вигляді бічної панелі (рисунок 3.1, а), яка викликається з основного екрана натисканням кнопки у лівій частині верхньої панелі або горизонтальним свайпом від лівого краю екрана. Бічна панель містить рядок пошуку для фільтрації чатів за назвою, кнопку відкриття налаштувань, кнопку зміни порядку сортування, кнопку додавання нового чату та власне перелік чатів. Натискання на елемент переліку перемикає поточний чат.

Створення нового чату (рисунок 3.1, в) виконується натисканням кнопки додавання у бічній панелі. На відкритому екрані користувач вводить назву чату, обирає колір тла із запропонованої палітри та за бажанням завантажує фонове зображення з галереї пристрою або вибирає символ-емоджі для іконки. Усі обрані параметри миттєво відображаються на круговій області попереднього перегляду у верхній частині екрана, що дозволяє оцінити результат до збереження. Натискання кнопки «Done» зберігає новий чат та повертає користувача на

основний екран. Редагування наявного чату виконується аналогічно та доступне через контекстне меню чату.

Основним робочим простором користувача є екран чату з повідомленнями (рисунок 3.1, б). Надсилання нового повідомлення виконується введенням тексту у поле внизу екрана з подальшим натисканням кнопки надсилання. До повідомлення можна прикріпити вкладення через меню, що викликається кнопкою прикріплення поряд з полем введення. Голосове повідомлення записується утриманням кнопки мікрофона. Натискання на наявне повідомлення у стрічці викликає контекстне меню з повним переліком операцій над ним: закріплення, копіювання тексту, додавання нагадування, призначення тегів, перенесення до іншого чату, редагування та видалення. Натискання на зображення у повідомленні відкриває окремий екран повноекранного перегляду, на якому користувач може гортати усі зображення чату, поділитися медіафайлом стандартними засобами операційної системи або зберегти його на пристрої. Пошук повідомлень за фрагментом тексту здійснюється через відповідну кнопку у верхній панелі екрана чату.

Налаштування застосунку (рисунок 3.1, г) викликаються з бічної панелі переліку чатів через кнопку шестерні. На цьому екрані користувач обирає тему оформлення — темну, світлу або системну, що відстежує налаштування пристрою, — а також керує застосуванням динамічної палітри та ефекту розмиття у різних частинах інтерфейсу. Пункт керування тегами відкриває окремий екран, на якому створюються, редагуються та видаляються позначки для категоризації повідомлень; кожен тег має назву, символ-емоджі та колір. Пункти створення та відновлення резервної копії дозволяють сформувати єдиний файл архіву з усіма даними застосунку та згодом повернути стан застосунку з раніше створеного файла на тому самому або новому пристрої.

Усі дії користувача виконуються миттєво на пристрої та не потребують підключення до мережі Інтернет. Створені чати, надіслані повідомлення, призначені теги, сформовані резервні копії повністю зберігаються в локальному

сховищі та не передаються на зовнішні сервери, що відповідає принципу повного контролю користувача над інформацією, обґрунтованому у підрозділі 1.3.

3.5 Технічні характеристики мобільного застосунку

Для нормальної роботи розробленого мобільного застосунку пристрій користувача повинен задовольняти певному набору технічних вимог, що визначаються особливостями платформи Android, обраним технологічним стеком та функціональними можливостями застосунку. У цьому підрозділі викладено мінімальні та рекомендовані вимоги до апаратного та програмного забезпечення пристрою, а також наведено перелік системних дозволів, які запитує застосунок під час встановлення та роботи.

Розроблений застосунок призначений для роботи на пристроях під управлінням операційної системи Android. Мінімальною підтримуваною версією платформи є Android 8.0 Oreo, що відповідає рівню програмного інтерфейсу 26. Цей вибір зумовлений необхідністю використання сучасних можливостей платформи — зокрема, каналів сповіщень, оновленої системи фонової роботи та підтримки векторних піктограм адаптивного розміру.

Цільовою версією платформи є Android 16, що відповідає рівню програмного інтерфейсу 36. Така конфігурація гарантує, що застосунок використовує актуальні можливості новітніх версій операційної системи і відповідає вимогам Google Play щодо рівня сумісності для застосунків, опублікованих у поточному році. На пристроях з версіями платформи між мінімальною та цільовою застосунок використовує механізми зворотної сумісності для уніфікованої поведінки на всіх підтримуваних пристроях.

Мінімальні апаратні вимоги до пристрою користувача визначаються вимогами самої операційної системи Android 8.0 та обсягом ресурсів, необхідних для роботи з декларативним інструментарієм Jetpack Compose. Рекомендованим є пристрій з обсягом оперативної пам'яті не менше ніж 2 ГБ, чотириядерним

процесором ARM-архітектури та постійним сховищем не менше ніж 16 ГБ. Розроблений застосунок не містить власного нативного коду та працює на будь-якій архітектурі процесора, що підтримується платформою Android — arm64-v8a, armeabi-v7a, x86 або x86_64, оскільки усю логіку реалізовано мовою Kotlin, яка компілюється у байт-код віртуальної машини Android Runtime. Особливих вимог до графічної підсистеми, наявності датчиків або апаратних модулів зв'язку застосунок не пред'являє.

Розмір інсталяційного пакета застосунку для встановлення з Google Play становить близько 7,9 МБ. Ця компактність досягається завдяки оптимізації засобами обфускатора R8 у режимі повної мінімізації, видаленню невикористаного коду бібліотек та автоматичному формуванню оптимізованих наборів ресурсів для конкретного пристрою через механізм Android App Bundle [33]. Після встановлення застосунок займає близько 30 МБ постійного сховища, включно з кешем ресурсів та порожньою базою даних. Подальше зростання обсягу займаного простору залежить виключно від обсягу даних користувача — кількості повідомлень та розміру прикріплених мультимедійних вкладень.

Для виконання своїх функцій застосунок запитує сім системних дозволів платформи Android. Дозвіл RECORD_AUDIO необхідний для запису голосових повідомлень безпосередньо в інтерфейсі чату. Дозвіл POST_NOTIFICATIONS, обов'язковий, починаючи з Android 13, дає змогу відображати сповіщення про спрацьовані нагадування у системній шторці пристрою. Дозвіл SCHEDULE_EXACT_ALARM забезпечує можливість точного у часі спрацювання нагадувань через системний компонент AlarmManager — без цього дозволу платформа залишає за собою право відкласти спрацювання нагадування на невизначений час задля економії ресурсів пристрою. Дозвіл RECEIVE_BOOT_COMPLETED дозволяє застосунку отримати сповіщення про завершення завантаження операційної системи та повторно зареєструвати усі заплановані нагадування, які могли бути втрачені при вимиканні пристрою. Дозвіл WAKE_LOCK потрібен для тимчасового виведення процесора з режиму

глибокого сну в момент спрацювання нагадування, що гарантує своєчасну доставку сповіщення навіть при вимкненому екрані. Дозволи FOREGROUND_SERVICE та FOREGROUND_SERVICE_DATA_SYNC дозволяють виконувати тривалі операції — формування резервної копії всіх даних застосунку та відновлення з раніше створеного архіву — у вигляді сервісу переднього плану з відображенням постійного сповіщення, що відповідає вимогам платформи до прозорості тривалих фонових операцій.

Усі перелічені дозволи запитуються або декларативно на етапі встановлення застосунку, або динамічно на етапі першого використання відповідної функції (для дозволів, що належать до небезпечного рівня — RECORD_AUDIO та POST_NOTIFICATIONS). Користувач у будь-який момент може відкликати наданий дозвіл через системні налаштування пристрою, після чого відповідна функція тимчасово стане недоступною без впливу на решту функцій застосунку. Жодних дозволів на доступ до мережі Інтернет, місцеположення, контактів, календаря, камери чи інших персональних даних користувача застосунок не потребує і не запитує.

3.6 Тестування мобільного застосунку

3.6.1 Аналіз методів тестування мобільного застосунку

Тестування програмного забезпечення може бути класифіковане за низкою ознак — рівнем виконання (модульне, інтеграційне, системне), доступом до внутрішньої структури коду (метод «білої скриньки», метод «чорної скриньки»), способом виконання (автоматизоване, ручне) та характером перевірки (функціональне, нефункціональне) [34]. Вибір конкретних методів та засобів тестування залежить від специфіки розроблюваного продукту та від рівня критичності виявлення тих чи інших класів дефектів.

Для розробленого мобільного застосунку основним методом тестування обрано ручне функціональне тестування за методом «чорної скриньки». Цей метод не передбачає знання внутрішньої структури коду та зосереджується на перевірці зовнішньої поведінки застосунку через його інтерфейс, що відповідає характеру розроблюваного продукту, у якому переважна більшість функцій реалізує безпосередню взаємодію користувача з інтерфейсом, базою даних та файловою системою пристрою. Ручне тестування дозволяє тестувальнику відтворювати реальні сценарії використання та виявляти дефекти, які виникають саме при поєднанні дій користувача з реакціями системних підсистем платформи Android — таких, як життєвий цикл компонентів, обробка змін конфігурації пристрою або взаємодія з системними компонентами операційної системи. Окремо проведено перевірку поведінки застосунку при змінах конфігурації пристрою, зокрема, при повороті екрана між портретною та ландшафтною орієнтаціями.

Тестування здійснювалося у середовищі розробки Android Studio з використанням двох цільових платформ. Першою є вбудований у Android Studio емулятор Android Virtual Device, налаштований на симуляцію пристрою з різними версіями платформи від мінімально підтримуваної до цільової. Другою є реальний фізичний пристрій, підключений до середовища розробки через інтерфейс налагодження Android Debug Bridge. Поєднання тестування на емуляторі та реальному пристрої відповідає сучасним методичним рекомендаціям щодо забезпечення якості мобільних застосунків [35], оскільки емулятор забезпечує швидку перевірку логіки на широкій матриці версій ОС, тоді як реальний пристрій додатково виявляє дефекти, пов'язані з апаратними особливостями та реальною продуктивністю. Поведінка застосунку на емуляторі та на реальному пристрої виявилася ідентичною у межах усіх перевірених сценаріїв, тому в подальшому викладі результати тестування подано без розділення за типом цільової платформи.

3.6.2 Тестування мобільного застосунку за допомогою емулятора

Тестування виконувалося шляхом послідовного проходження тест-кейсів, розроблених на основі функціональних вимог та діаграми варіантів використання з підрозділу 1.3. Сценарії перевірялися як з коректними, так і з некоректними або граничними вхідними даними. Тест-кейси згруповано за функціональними областями у чотирьох таблицях.

Першу групу тест-кейсів становлять сценарії, пов'язані з керуванням чатами — створенням нових чатів, редагуванням наявних та обробкою граничних ситуацій валідації введених користувачем даних. Перелік цих тест-кейсів подано у таблиці 3.1.

Таблиця 3.1 – Тест-кейси керування чатами

Дія	Очікуваний результат	Фактичний результат
Створити новий чат із заданими назвою, кольором та емоджі	Новий чат з'являється у переліку	Правильно
Створити чат із порожньою назвою	Виводиться повідомлення про помилку, чат не зберігається	Правильно
Редагувати наявний чат, змінити колір та зберегти	Зміни відображаються у переліку чатів	Правильно
Вийти з екрана редагування з незбереженими змінами	Виводиться діалог підтвердження виходу	Правильно
Завантажити фонове зображення для чату з галереї	Зображення відображається на іконці чату	Правильно

Другу групу тест-кейсів становлять сценарії роботи з повідомленнями — їх надсилання, прикріплення мультимедійних вкладень різних типів та операції, що виконуються через контекстне меню повідомлення. Особлива увага у цій групі приділена перевірці поведінки інтерфейсу при надсиланні повідомлень з декількома вкладеннями та з відеофайлами, оскільки такі сценарії потребують узгодженої роботи декількох системних підсистем платформи Android. Перелік відповідних тест-кейсів подано у таблиці 3.2.

Таблиця 3.2 – Тест-кейси роботи з повідомленнями

Дія	Очікуваний результат	Фактичний результат
Надіслати текстове повідомлення в чат	Повідомлення з'являється у стрічці у хронологічному порядку	Правильно
Прикріпити одне зображення до повідомлення	Зображення відображається у стрічці у вигляді картки	Правильно
Прикріпити декілька зображень до повідомлення	Зображення формують мережу у вигляді картки	Правильно
Прикріпити відео до повідомлення	Відео відображається у стрічці у вигляді картки	Правильно
Записати голосове повідомлення утриманням мікрофона	Записаний аудіофайл з'являється у стрічці	Правильно
Закріпити повідомлення через контекстне меню	Повідомлення відображається у блоці закріплених	Правильно
Відповісти на повідомлення через контекстне меню	Цитоване повідомлення показується у картці відповіді	Правильно
Видалити повідомлення з вкладеннями	Повідомлення зникає зі стрічки, файли вкладень видаляються	Правильно

Третю групу тест-кейсів становлять сценарії, пов'язані з допоміжними функціями застосунку — системою тегів повідомлень, нагадуваннями та пошуком. Перелік відповідних тест-кейсів подано у таблиці 3.3.

Таблиця 3.3 – Тест-кейси системи тегів, нагадувань та пошуку

Дія	Очікуваний результат	Фактичний результат
Створити тег та призначити його повідомленню	Кольорова позначка тегу відображається під повідомленням	Правильно
Фільтрувати повідомлення за обраним тегом	У стрічці залишаються лише повідомлення з обраним тегом	Правильно
Видалити тег, призначений декільком повідомленням	Тег зникає з усіх повідомлень, до яких був призначений	Правильно
Створити нагадування на повідомлення	Сповіщення спрацьовує у визначений час	Правильно
Виконати пошук повідомлень за фрагментом тексту	Знайдені повідомлення підсвічуються та доступні для переходу	Правильно

Четверту групу тест-кейсів становлять сценарії, пов'язані з налаштуваннями застосунку, резервним копіюванням даних та системними дозволами. Ця група охоплює функції, які виходять за межі основних сценаріїв спілкування, але є

критично важливими для довгострокового використання застосунку. Перелік відповідних тест-кейсів подано у таблиці 3.4.

Таблиця 3.4 – Тест-кейси налаштувань, резервного копіювання та системних дозволів

Дія	Очікуваний результат	Фактичний результат
Створити резервну копію всіх даних	Файл архіву формується та відображається у переліку	Правильно
Відновити стан застосунку з раніше створеного файла резервної копії	Усі чати, повідомлення та теги повертаються до попереднього стану	Правильно
Перемкнути тему оформлення з темної на світлу	Інтерфейс миттєво змінює оформлення без перезапуску	Правильно
Запустити застосунок без наданого дозволу RECORD_AUDIO	Кнопка мікрофона запитує дозвіл при першому натисканні	Правильно

У всіх перевірених випадках стан екрана коректно зберігався при змінах конфігурації пристрою завдяки використанню класу ViewModel бібліотеки Jetpack, описаному у підрозділі 3.1, а введені користувачем дані не втрачалися. На пристрої з мінімальними апаратними характеристиками, зокрема емулятор Android Virtual Device з 2 ГБ оперативної пам'яті, застосунок зберігав плавність прокручування стрічки повідомлень, що містила декілька сотень записів, завдяки використанню постраничного завантаження через бібліотеку Paging 3 — повідомлення завантажуються з бази даних порціями у міру прокручування, що унеможливлює одночасне утримання у пам'яті усіх записів. Окремо було перевірено поведінку застосунку в умовах переривань — вхідних дзвінків, повідомлень з інших застосунків та вимкнення екрана під час введення тексту. В усіх випадках стан екрана коректно відновлювався після повернення до застосунку без втрати введених даних.

3.6.3 Аналіз результатів тестування мобільного застосунку

За результатами тестування підтверджено, що розроблений застосунок відповідає функціональним та нефункціональним вимогам з підрозділу 1.3. Усі заплановані сценарії використання працюють коректно як на емуляторі, так і на реальному пристрої.

У процесі тестування було виявлено два суттєві дефекти, які вплинули на коректність роботи застосунку у відповідних сценаріях. Обидва дефекти було локалізовано, проаналізовано та виправлено до завершення робіт над застосунком.

Перший дефект полягав у некоректному визначенні розмірів відеофайлів, прикріплених до повідомлень. При прикріпленні відео, записаного у портретній орієнтації, картка повідомлення відображала його з неправильним співвідношенням сторін — відео розтягувалося або обрізалося. Аналіз показав, що метадані відеофайлу у системному сховищі Android Media Store містять окреме поле повороту, яке не враховується при безпосередньому отриманні значень ширини та висоти. Для значень повороту 90 та 270 градусів отримані ширина та висота фактично відповідають оригінальній орієнтації запису камери, а не тому, як відео повинно відображатися. Дефект усунено шляхом додаткового зчитування метаданих повороту відеофайлу та коригування отриманих значень розмірів відповідно до фактичної орієнтації запису.

Другий дефект виявлено у сценарії формування резервної копії даних. Для забезпечення цілісності експортованої бази даних під час формування архіву виконувалося явне закриття активного з'єднання з базою. Однак після завершення формування архіву та повернення користувача до основного інтерфейсу інтерфейсний шар припиняв реагувати на зміни даних — нові повідомлення не з'являлися у стрічці у реальному часі, перелік чатів не оновлювався, а постраничне завантаження зупинялося. Аналіз показав, що реактивні підписки, створені до моменту закриття бази даних, ставали недійсними і не відновлювалися

автоматично після подальшої роботи з тим самим екземпляром бази даних. Дефект усунено шляхом додавання механізму повторного відкриття бази даних після завершення процедури резервного копіювання, що забезпечує створення нових реактивних підписок та відновлення очікуваної поведінки інтерфейсу.

Виявлення обох дефектів продемонструвало доцільність обраного підходу до тестування — обидва вони пов'язані з нюансами взаємодії застосунку з системними підсистемами платформи Android, і їх практично неможливо виявити автоматизованими методами, які виконуються в ізольованому середовищі без участі цих підсистем. Ручне функціональне тестування на реальному пристрої виявилося ефективним методом виявлення таких класів дефектів.

Підсумовуючи проведене тестування, можна зробити такий висновок, що розроблений мобільний застосунок є повністю працездатним, реалізує усі заплановані функції, коректно поводиться у нестандартних ситуаціях та відповідає вимогам технічного завдання. Виявлені у процесі тестування дефекти усунено, що підтверджується повторним проходженням відповідних тест-кейсів. Застосунок готовий до публікації у Google Play та подальшого використання кінцевими користувачами.

3.7 Висновки

Після стадії проектування було здійснено програмну реалізацію та тестування мобільного застосунку відповідно до архітектурних та проєктних рішень, обґрунтованих у попередньому розділі.

Реалізовано логіку всіх екранів застосунку у вигляді тримачів стану, які обробляють дії користувача, виконують необхідні бізнес-операції у фоновому режимі та повертають оновлений стан до інтерфейсу через реактивні потоки даних. Бізнес-логіку застосунку виокремлено у вигляді сценаріїв використання, кожен з яких інкапсулює одну операцію над сутностями застосунку та звертається до шару даних виключно через абстрактні інтерфейси.

Розмітку інтерфейсу побудовано із застосуванням обраного декларативного інструментарію та відповідно до сучасної специфікації візуального дизайну платформи Android. Інтерфейс підтримує темний, світлий та системний режими оформлення, динамічну палітру кольорів на основі шпалер пристрою, а також адаптивну поведінку у портретній та ландшафтній орієнтаціях.

Реалізовано локальну базу даних застосунку, описано структуру її сутностей, об'єктів доступу до даних та механізм керування версіями схеми. Передбачено механізм оновлення схеми бази даних на пристроях користувачів без втрати наявних даних.

Складено керівництво користувача, у якому послідовно описано основні сценарії взаємодії з застосунком — від створення чатів та надсилання повідомлень до керування тегами, нагадуваннями та резервним копіюванням даних. Окремо викладено технічні характеристики розробленого застосунку та перелік системних дозволів, необхідних для його повноцінної роботи.

Тестування застосунку проведено методом ручного функціонального тестування за принципом «чорної скриньки» з використанням як емулятора, так і реального фізичного пристрою. Розроблено та виконано тест-кейси, що охоплюють усі основні функціональні області застосунку. У процесі тестування виявлено та усунено всі дефекти.

Таким чином, у результаті виконання робіт цього розділу одержано працездатний мобільний застосунок, який відповідає сформульованим у технічному завданні функціональним та нефункціональним вимогам та готовий до подальшого використання. Отримані у процесі розроблення та тестування результати підтверджують доцільність обраного технологічного стеку та архітектурних рішень для задач цього класу. Застосунок демонструє стабільну роботу в різних умовах використання — від типових сценаріїв до граничних випадків, виявлених під час тестування. Сукупність реалізованих функцій повністю задовольняє потреби цільової аудиторії, визначені у першому розділі.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи розв'язано актуальну практичну задачу інженерії програмного забезпечення — розроблено локальний мобільний застосунок для платформи Android, призначений для створення та збереження персональних нотаток у форматі тематичних чатів. Створений програмний продукт надає користувачеві можливість оперативно фіксувати інформацію у звичному діалоговому форматі, систематизувати її засобами категоризації та зберігати винятково на пристрої, не вдаючись до сторонніх хмарних сервісів.

На першому етапі виконано змістовий аналіз предметної області з використанням порівняльного методу дослідження наявних рішень та методик моделювання вимог засобами діаграм варіантів використання та потоків даних. У результаті виявлено три критичні потреби користувача — швидкість фіксації інформації, можливості її структурування та конфіденційність даних, — які наявні на ринку рішення задовольняють лише частково. Виконано порівняльний аналіз поширених аналогів та обґрунтовано доцільність розроблення нового програмного продукту, що поєднує переваги діалогового інтерфейсу з парадигмою локального зберігання даних. На основі проведеного аналізу сформульовано функціональні та нефункціональні вимоги до застосунку, оформлені у вигляді деталізованого технічного завдання.

На другому етапі виконано проєктні роботи із застосуванням принципів шарової архітектури з чітким розподілом обов'язків між складовими частинами застосунку, методів нормалізації реляційних даних та принципів юзабіліті проєктування інтерфейсу. Обґрунтовано вибір архітектурного рішення, що забезпечує необхідний рівень модульності, тестовності та супроводжуваності продукту впродовж усього його життєвого циклу. Спроєктовано структуру локальної бази даних із виконанням нормалізації даних до третьої нормальної форми. Сформовано макети основних і допоміжних екранів та створено

інтерактивний прототип. Деталізовано алгоритми ключових процесів функціонування застосунку. За результатами порівняльного огляду доступних альтернатив обрано технологічний стек для подальшої реалізації.

На третьому етапі виконано програмну реалізацію та тестування застосунку із застосуванням сучасної мови програмування для обраної платформи, декларативного інструментарію побудови інтерфейсу, локальної бази даних на основі обраної СКБД та сучасної специфікації візуального дизайну платформи Android. Реалізовано логіку всіх екранів та побудовано адаптивну розмітку інтерфейсу, створено локальну базу даних із механізмом керування версіями схеми, складено керівництво користувача. Проведено ручне функціональне тестування за принципом «чорної скриньки» з випробуваннями як на емуляторі, так і на реальному фізичному пристрої; виявлені дефекти усунено, що підтверджено повторним проходженням відповідних перевірок.

Отриманим результатом виконаної роботи є функціонально завершений мобільний застосунок, який повністю відповідає вимогам технічного завдання, реалізує всі заплановані сценарії використання та готовий до подальшого розповсюдження серед кінцевих користувачів. Поставлені перед кваліфікаційною роботою задачі виконано у повному обсязі, мети досягнуто.

Впровадження розробленого застосунку надає користувачам низку істотних переваг. Діалоговий формат взаємодії скорочує час на фіксацію думок та ідей і мінімізує когнітивне навантаження, а локальне зберігання гарантує повний контроль користувача над власною інформацією, її конфіденційність та автономну роботу за відсутності підключення до Інтернету. Засоби структурування записів за тематичними чатами та системою тегів спрощують подальший пошук і повторне використання збережених даних.

Перспективними напрямами подальшого розвитку розробленого застосунку є посилення захисту локального сховища засобами шифрування даних, інтеграція функцій автоматичного перетворення голосових повідомлень на текст, а також адаптація застосунку для інших мобільних.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Forte T. Building a Second Brain: A Proven Method to Organize Your Digital Life and Unlock Your Creative Potential. New York : Atria Books, 2022. P. 11–34. ISBN 978-1-9821-6738-7.
2. Note Taking App Market Report : Research and Markets. URL: https://www.researchandmarkets.com/reports/5790688/note-taking-app-market-report (дата звернення: 14.04.2026).
3. Sweller J. Cognitive load theory and individual differences // Learning and Individual Differences. 2024. Vol. 110. Article 102423. P. 1–8. DOI: 10.1016/j.lindif.2023.102423.
4. Grinschgl S., Meyerhoff H. S., Papenmeier F. Interface and interaction design: How mobile touch devices foster cognitive offloading // Computers in Human Behavior. 2020. Vol. 108. Article 106317. P. 1–10. DOI: 10.1016/j.chb.2020.106317.
5. Kelly M. O., Risko E. F. Study effort and the memory cost of external store availability // Cognition. 2022. Vol. 228. Article 105228. P. 1–14. DOI: 10.1016/j.cognition.2022.105228.
6. Gilbert S. J., Boldt A., Sachdeva C., Scarampi C., Tsai P.-C. Outsourcing memory to external tools: A review of 'intention offloading' // Psychonomic Bulletin & Review. 2023. Vol. 30, No. 1. P. 60–76. DOI: 10.3758/s13423-022-02139-4.
7. Parreira Junior P. A., Freire A. P. Mobile User Interaction Design Patterns: A Systematic Mapping Study // Information. 2022. Vol. 13, No. 5. Article 236. P. 1–24. DOI: 10.3390/info13050236.
8. Kleppmann M., Wiggins A., Van Hardenberg P., McGranaghan M. Local-first software: You own your data, in spite of the cloud. Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New

Paradigms, and Reflections on Programming and Software (Onward! 2019). New York : ACM, 2019. P. 154–178. DOI: 10.1145/3359591.3359737.

9. Iqbal S., Al-Azzoni I., Allen G., Khan H. U. Extending UML use case diagrams to represent non-interactive functional requirements // E-Informatica Software Engineering Journal. 2020. Vol. 14, No. 1. P. 97–115.
10. Sommerville I. Engineering Software Products: An Introduction to Modern Software Engineering. Harlow : Pearson, 2020. P. 67–95. ISBN 978-1-292-37634-9.
11. Weichbroth P. Usability of mobile applications: a systematic literature study // IEEE Access. 2020. Vol. 8. P. 55563–55577.
12. Lortie J., Cox K., DeRosset S., Thompson R., Kelly S. Unpacking the minimum viable product (MVP): a framework for use, goals and essential elements // Journal of Small Business and Enterprise Development. 2025. Vol. 32, No. 1. P. 212–235.
13. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. Sebastopol : O'Reilly Media, 2020. P. 3–25. ISBN 978-1-4920-4345-4.
14. Bass L., Clements P., Kazman R. Software Architecture in Practice. 4th ed. Boston : Addison-Wesley Professional, 2021. P. 51–80. ISBN 978-0-13-688609-9.
15. Guide to app architecture : офіційна документація Android Developers. URL: https://developer.android.com/topic/architecture (дата звернення: 15.04.2026).
16. Model-View-Intent for Android : офіційна документація Android Developers. URL: https://developer.android.com/topic/architecture/ui-layer (дата звернення: 18.04.2026).
17. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston : Prentice Hall, 2017. P. 197–209. ISBN 978-0-13-449416-6.

18. Anik M. R. H. The Impact of Agile Methodology on Software Project Management : Master's thesis / Vaasa University of Applied Sciences. Vaasa, 2025. P. 25–27.
19. Coronel C., Morris S. Database Systems: Design, Implementation, & Management. 14th ed. Boston : Cengage Learning, 2023.
20. Material Design 3 : офіційна документація Google. URL: https://m3.material.io/ (дата звернення: 02.05.2026).
21. Punchoojit L., Hongwarittorrn N. Usability Studies on Mobile User Interface Design Patterns: A Systematic Literature Review // Advances in Human-Computer Interaction. 2017. Vol. 2017. Article 6787504. P. 1–22. DOI: 10.1155/2017/6787504.
22. Tidwell J., Brewer C., Valencia A. Designing Interfaces: Patterns for Effective Interaction Design. 3rd ed. Sebastopol : O'Reilly Media, 2020. P. 73–118. ISBN 978-1-4920-5196-1.
23. Laurence P.-O., Hinchman-Dominguez A., Meike G. B., Dunn M. Programming Android with Kotlin: Achieving Structured Concurrency with Coroutines. Sebastopol : O'Reilly Media, 2021. P. 75–160. ISBN 978-1-4920-6300-1.
24. Jetpack Compose : офіційна документація Android Developers. URL: https://developer.android.com/jetpack/compose (дата звернення: 05.05.2026).
25. Moskała M. Kotlin Coroutines: Deep Dive. Warsaw : Kt. Academy, 2023. P. 41–110. ISBN 978-83-963825-0-7.
26. Coil : офіційна документація бібліотеки. URL: https://coil-kt.github.io/coil/ (дата звернення: 05.05.2026).
27. AndroidX Media3 : офіційна документація Android Developers. URL: https://developer.android.com/media/media3 (дата звернення: 05.05.2026).

28. StateFlow and SharedFlow : офіційна документація Kotlin. URL: https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-state-flow (дата звернення: 05.05.2026).
29. Kotlin Coroutines on Android : офіційна документація Android Developers. URL: https://developer.android.com/kotlin/coroutines (дата звернення: 23.04.2026).
30. Künneth T. Android UI Development with Jetpack Compose: Bring declarative and native UI to life quickly and easily on Android using Jetpack Compose and Kotlin. 2nd ed. Birmingham : Packt Publishing, 2023. P. 25–60
31. Save data in a local database using Room : офіційна документація Android Developers. URL: https://developer.android.com/training/data-storage/room (дата звернення: 05.05.2026).
32. Paging library overview : офіційна документація Android Developers. URL: https://developer.android.com/topic/libraries/architecture/paging/v3-overview (дата звернення: 05.05.2026).
33. About Android App Bundles : офіційна документація Android Developers. URL: https://developer.android.com/guide/app-bundle (дата звернення: 06.05.2026).
34. Spillner A., Linz T. Software Testing Foundations: A Study Guide for the Certified Tester Exam — Foundation Level — ISTQB Compliant. 5th ed. Heidelberg : dpunkt.verlag, 2021. P. 41–95. ISBN 978-3-86490-834-7.
35. Knott D. Hands-On Mobile App Testing: A Guide for Mobile Testers and Anyone Involved in the Mobile App Business. 2nd ed. Independently published, 2022. P. 65–130. ISBN 979-8-8316-4775-4.

ДОДАТОК А

(обов’язковий)

ТЕХНІЧНЕ ЗАВДАННЯ

Введення

Робота виконується в рамках проєкту розроблення мобільного застосунку для створення й збереження нотаток у форматі особистого чату на платформі Android.

Умовне позначення розробки: мобільний застосунок для ведення нотаток у форматі тематичних чатів.

1 Підстава для розробки

Підставою для розробки є «Завдання на кваліфікаційну роботу», затверджене завідувачем кафедри інженерії програмного забезпечення Хмельницького національного університету.

Найменування розробки: «Мобільний застосунок для створення й збереження нотаток у форматі особистого чату на платформі Android».

Виконавець: студент IV курсу, групи ІПЗ-22-1 Хотич Микола Володимирович.

Керівник: д-р фіз.-мат. наук, професор Бедратюк Леонід Петрович.

2 Призначення розробки

2.1 Функціональне призначення

Функціональне призначення – надати користувачам зручний інструмент для оперативного створення, збереження та структурування персональних нотаток у форматі тематичних чатів із гарантованою конфіденційністю завдяки виключно локальному зберіганню даних.

Застосунок надаватиме можливість:

– організовувати нотатки у вигляді тематичних чатів з унікальними назвами та візуальним оформленням;

– відправляти, редагувати та видаляти текстові нотатки з підтримкою базового форматування Markdown;

– прикріплювати мультимедійні вкладення (зображення, відео, аудіо, файли) до повідомлень;

– записувати та відтворювати голосові нотатки в інтерфейсі чату;

– позначати повідомлення тегами, закріплювати важливі записи та відповідати на конкретні повідомлення;

– встановлювати нагадування з отриманням системних сповіщень у визначений час;

– здійснювати пошук записів та фільтрацію за тегами у межах обраного чату;

– формувати та відновлювати резервні копії усіх даних у форматі ZIP-архіву.

2.2 Експлуатаційне призначення

Мобільний застосунок призначено для використання на смартфонах під управлінням операційної системи Android версії 8.0 та новіших. Кінцевими користувачами є особи, які потребують оперативного, структурованого та конфіденційного інструменту для ведення персональних записів без прив'язки до Інтернет-з'єднання. Застосунок не потребує реєстрації, налаштування хмарних акаунтів або додаткових налаштувань для початку роботи.

3 Вимоги до програмного продукту

3.1 Вимоги до функціональних характеристик

Застосунок повинен забезпечувати такі функціональні можливості:

– створення, редагування та видалення тематичних чатів;

– функціонування екрана останнього відкритого чату як головного екрана застосунку (при запуску застосунку користувач одразу потрапляє до останнього чату без проміжних екранів);

– відправлення, редагування та видалення текстових повідомлень із підтримкою базового форматування Markdown;

– відповідь на повідомлення для встановлення логічних зв'язків між записами;

– закріплення важливих повідомлень для швидкого доступу;

– встановлення нагадувань на конкретні повідомлення з можливістю отримання системних сповіщень у визначений час;

– позначення повідомлень як виконаних із відповідним візуальним відображенням;

– додавання до повідомлень мультимедійних вкладень: зображень та відео з галереї пристрою, нових фото чи відео через камеру, аудіофайлів та довільних файлів із файлової системи;

– запис і відтворення голосових нотаток безпосередньо в інтерфейсі чату;

– перегляд вкладень популярних форматів (зображень, відео, аудіо) безпосередньо в застосунку без залучення сторонніх програм;

– присвоєння повідомленням користувацьких тегів для категоризації записів;

– пошук записів у межах обраного чату та фільтрація за визначеними тегами;

– налаштування візуального оформлення чату: встановлення унікального кольору, емоджі або фонового зображення з галереї;

– сортування та пошук чатів у бічному навігаційному меню;

– отримання контенту зі сторонніх застосунків через системний механізм Intent;

– пересилання повідомлень і вкладень у сторонні застосунки;

– формування резервних копій усіх даних застосунку у форматі ZIP-архіву та їх відновлення.

3.2 Вимоги до надійності

Застосунок повинен відповідати таким вимогам до надійності:

– усі введені користувачем дані (текстові нотатки, вкладення, теги, нагадування) зберігаються в локальній базі даних і не втрачаються при згортанні або ненавмисному закритті застосунку;

– зареєстровані нагадування відновлюються після перезавантаження пристрою завдяки обробнику системного події RECEIVE_BOOT_COMPLETED;

– обфускація та мінімізація програмного коду засобами R8 та ProGuard гарантують захист від зворотної інженерії;

– усі операції читання та запису до бази даних і файлової системи виконуються у фонових корутинах і не блокують головний потік інтерфейсу;

– застосунок не виходить з ладу при введенні некоректних або порожніх даних — такі ситуації обробляються з виведенням відповідних повідомлень;

– тривалі операції (формування та відновлення резервної копії) виконуються у вигляді сервісу переднього плану з відображенням постійного системного сповіщення про хід операції;

– відкликання системних дозволів користувачем вимикає лише відповідну функцію без впливу на решту функціональності застосунку.

Для забезпечення якості застосунок пройшов модульне тестування бізнес-логіки та тестування за допомогою емулятора пристрою Android.

3.3 Умови експлуатації

Умови експлуатації повинні відповідати санітарним і технічним нормам експлуатації мобільних пристроїв, визначеним виробником пристрою. Застосунок призначений виключно для персонального використання одним користувачем на його особистому смартфоні. Мережеве з'єднання не є обов'язковою умовою роботи застосунку — усі основні функції доступні в автономному режимі.

3.4 Вимоги до складу та параметрів технічних засобів

Мінімальна конфігурація пристрою:

– операційна система: Android 8.0 Oreo (API рівень 26) або новіша;

– архітектура процесора: ARM (armeabi-v7a або arm64-v8a), x86 або x86_64;

– обсяг оперативної пам'яті: не менше 2 ГБ;

– обсяг вільного постійного сховища: не менше 100 МБ (розмір інсталяційного пакета — близько 7,9 МБ, після встановлення — близько 30 МБ; подальший приріст залежить від обсягу даних користувача).

Рекомендована конфігурація пристрою:

– операційна система: Android 12.0 (API рівень 31) або новіша;

– обсяг оперативної пам'яті: 4 ГБ або більше;

– обсяг вільного постійного сховища: не менше 1 ГБ.

Цільовою версією платформи є Android 16 (API рівень 36). Застосунок не пред'являє особливих вимог до графічної підсистеми або наявності специфічних апаратних модулів.

3.5 Вимоги до інтерфейсу користувача

Інтерфейс застосунку повинен відповідати таким вимогам:

– головним екраном, що відображається при запуску, є останній відкритий користувачем чат, що забезпечує миттєву фіксацію нотаток без зайвої навігації;

– компонування екрана чату відповідає звичним патернам популярних месенджерів: стрічка повідомлень займає основну площу, поле введення та кнопка надсилання розміщуються у нижній частині екрана;

– список чатів відображається у вигляді бічного висувного навігаційного меню;

– колірна схема, типографіка та поведінка інтерактивних елементів відповідають специфікаціям Material Design 3;

– інтерфейс адаптується до різних розмірів і роздільної здатності екранів без втрати читабельності та зручності;

– підтримується системна темна та світла теми оформлення.

3.6 Вимоги до програмної документації

Замовнику має бути надана вся необхідна документація по програмному застосунку, а також технічне завдання.

Відповідно замовнику надається розроблений програмний застосунок з відповідною до нього інструкцією користувача.

4 Обґрунтування корисності розробки

Практична цінність розробки полягає в реалізації мінімально життєздатного програмного продукту, що поєднує найкращі практики наявних рішень:

– діалоговий інтерфейс у стилі месенджерів забезпечує мінімальне когнітивне навантаження та максимальну швидкість фіксації нотаток — аналогічно до Telegram Saved Messages, але з підтримкою тематичних чатів;

– парадигма локального збереження даних гарантує повну конфіденційність, незалежність від Інтернет-з'єднання та миттєвий відгук інтерфейсу — на відміну від хмарних аналогів Google Keep;

– простий і звичний інтерфейс знижує поріг входу порівняно з комплексними системами управління знаннями (Obsidian), зберігаючи при цьому можливості структурування через тематичні чати та теги.

Розроблений застосунок дозволяє:

– оперативно фіксувати персональну інформацію у звичному діалоговому форматі без реєстрації та налаштувань;

– структурувати записи засобами тематичних чатів і тегів;

– зберігати конфіденційність даних, оскільки інформація фізично залишається на пристрої користувача.

5 Стадії та етапи розробки

Стадії та етапи розробки проєкту «Мобільний застосунок для створення й збереження нотаток у форматі особистого чату на платформі Android» наведено в таблиці А.1.

Таблиця А.1 – Стадії та етапи розробки проєкту

Строк виконання	Назва етапу	Зміст робіт
01.12–31.12.2025	Ознайомлення з тематикою, визначення та узгодження теми КвР	Розгляд тематики кваліфікаційних робіт, визначення та узгодження індивідуальної теми
01.01–20.02.2026	Збір матеріалу, дослідження предметної області, розробка ТЗ	Дослідження предметної області, аналіз наявних аналогів, визначення вимог, розроблення технічного завдання
21.02–20.03.2026	Проєктування програмного забезпечення	Обґрунтування архітектурного рішення, проєктування БД, інтерфейсу, розроблення алгоритмів, вибір технологічного стеку
21.03–30.04.2026	Програмна реалізація та тестування ПЗ	Реалізація застосунку засобами Kotlin, Jetpack Compose, Room, Hilt; проведення модульного та інтеграційного тестування
01.05–25.05.2026	Написання вступу, висновків, оформлення списку джерел та додатків	Оформлення пояснювальної записки відповідно до вимог методичних вказівок кафедри ІПЗ
Травень 2026	Попередній захист КвР	Попередній захист з отриманням рекомендацій щодо вдосконалення пояснювальної записки
26.05–30.06.2026	Перевірка на плагіат, нормоконтроль, отримання відгуків, брошурування	Отримання необхідних документів для затвердження кваліфікаційної роботи
з 01.06.2026	Захист КвР	Підготовка до захисту та захист кваліфікаційної роботи

6 Порядок контролю та приймання

Контроль і приймання розробленого програмного застосунку здійснюється керівником кваліфікаційної роботи. Перед прийманням повинні бути проведені всі передбачені види тестування для перевірки відповідності визначеним вимогам.

ДОДАТОК Б

(обов’язковий)

КОД (ЛІСТИНГ) ПРОГРАМИ

Код класу MainActivity

```
import android.content.Intent
import android.net.Uri
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.activity.viewModels
import androidx.compose.foundation.isSystemInDarkTheme
import androidx.compose.material3.SnackbarDuration
import androidx.compose.material3.SnackbarHostState
import androidx.compose.material3.SnackbarResult
import androidx.compose.runtime.Composable
import androidx.compose.runtime.CompositionLocalProvider
import androidx.compose.runtime.DisposableEffect
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.runtime.remember
import androidx.core.splashscreen.SplashScreen.Companion.installSplashScreen
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.lifecycle.lifecycleScope
import androidx.navigation.NavController
import androidx.navigation.NavDestination
import androidx.navigation.NavHostController
import androidx.navigation.compose.rememberNavController
import com.google.android.play.core.appupdate.AppUpdateManager
import com.google.android.play.core.appupdate.AppUpdateManagerFactory
import com.google.android.play.core.appupdate.AppUpdateOptions
import com.google.android.play.core.install.InstallStateUpdatedListener
import com.google.android.play.core.install.model.AppUpdateType
import com.google.android.play.core.install.model.InstallStatus
import com.google.android.play.core.install.model.UpdateAvailability
import com.google.android.play.core.ktx.isFlexibleUpdateAllowed
import com.kappdev.selfthread.core.analytics.events.ReceivedOpenMessageEvent
import com.kappdev.selfthread.core.analytics.events.ReceivedSharedContentEvent
import com.kappdev.selfthread.core.analytics.events.ScreenViewEvent
import com.kappdev.selfthread.core.analytics.provider.LocalAnalyticsSender
import com.kappdev.selfthread.core.analytics.sender.AnalyticsSender
import com.kappdev.selfthread.core.common.appnavigator.AppNavigator
import com.kappdev.selfthread.core.common.blur.BlurState
```

```
import com.kappdev.selfthread.core.common.blur.LocalBlurState
import com.kappdev.selfthread.core.common.intent.getLongExtraOrNull
import com.kappdev.selfthread.core.common.intent.getParcelable
import com.kappdev.selfthread.core.common.intent.getParcelableList
import com.kappdev.selfthread.core.common.theme.isDynamicColorAllowed
import com.kappdev.selfthread.core.designsystem.theme.SelfThreadTheme
import com.kappdev.selfthread.core.localization.LocalizableStrings
import com.kappdev.selfthread.core.presentation.common.components.DraggableSnackbar
import com.kappdev.selfthread.core.presentation.navigation.AppNavGraph
import com.kappdev.selfthread.core.presentation.navigation.Screen
import
com.kappdev.selfthread.core.realmmigration.presentation.components.RealmToRoomMigrationDial
og
import com.kappdev.selfthread.realmtoroom.presentation.MigrationViewModel
import com.kappdev.selfthread.shared.backup.presentation.ui.route.BackupProgressOverlayRoute
import com.kappdev.selfthread.shared.backup.presentation.ui.viewmodel.BackupViewModel
import com.kappdev.selfthread.shared.settings.models.presentation.AppThemePresentationModel
import
com.kappdev.selfthread.shared.settings.models.presentation.AppThemeTypePresentationModel
import com.kappdev.selfthread.shared.sharedcontent.models.domain.SharedContentDomainModel
import dagger.hilt.android.AndroidEntryPoint
import kotlinx.coroutines.flow.MutableSharedFlow
import kotlinx.coroutines.launch
import javax.inject.Inject

@AndroidEntryPoint
class MainActivity : ComponentActivity() {
    @Inject
    lateinit var analyticsSender: AnalyticsSender

    private lateinit var navController: NavHostController
    private lateinit var appUpdateManager: AppUpdateManager

    private val updateSnackbarEvent = MutableSharedFlow<Unit>()

    private val migrationViewModel: MigrationViewModel by viewModels()
    private val mainViewModel: MainViewModel by viewModels()
    private val backupViewModel: BackupViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        val splashScreen = installSplashScreen()
        super.onCreate(savedInstanceState)
        splashScreen.setKeepOnScreenCondition {
            mainViewModel.uiState.value.settings == null
        }
        enableEdgeToEdge()

        appUpdateManager = AppUpdateManagerFactory.create(this)
        appUpdateManager.registerListener(installStateUpdateListener)
        checkForAppUpdates()
```

```
        processIntent(intent)

        setContent {
            val uiState by mainViewModel.uiState.collectAsStateWithLifecycle()
            val migrationState by migrationViewModel.state.collectAsStateWithLifecycle()

            CompositionLocalProvider(
                LocalAnalyticsSender provides analyticsSender,
                LocalBlurState provides if (uiState.settings?.isBlurEnabled == true) BlurState.ENABLED
else BlurState.DISABLED
            ) {
                SelfThreadTheme(appTheme = uiState.settings?.appTheme) {
                    navController = rememberNavController()

                    DisposableEffect(navController) {
                        val listener = NavController.OnDestinationChangedListener { _, destination, _ ->
                            destination.parseScreenName()?.let { screenName ->
                                analyticsSender.sendEvent(ScreenViewEvent(screenName))
                            }
                        }
                        navController.addOnDestinationChangedListener(listener)
                        onDispose {
                            navController.removeOnDestinationChangedListener(listener)
                        }
                    }

                    AppNavGraph(navController = navController)

                    val snackbarHostState = remember { SnackbarHostState() }

                    LaunchedEffect(Unit) {
                        updateSnackbarEvent.collect {
                            val result = snackbarHostState.showSnackbar(
                                message = getString(LocalizableStrings.the_update_is_ready),
                                actionLabel = getString(LocalizableStrings.install),
                                duration = SnackbarDuration.Indefinite
                            )
                            if (result == SnackbarResult.ActionPerformed) {
                                appUpdateManager.completeUpdate()
                            }
                        }
                    }

                    DraggableSnackbar(snackbarHostState)

                    if (migrationState.showDialog) {
                        RealmToRoomMigrationDialog(progress = migrationState.progress)
                    }
```

```
                BackupProgressOverlayRoute(viewModel = backupViewModel)
            }
        }
    }
}

private fun checkForAppUpdates() {
    appUpdateManager.appUpdateInfo.addOnSuccessListener { info ->
        val isUpdateAvailable = info.updateAvailability() ==
UpdateAvailability.UPDATE_AVAILABLE
        val isUpdateAllowed = info.isFlexibleUpdateAllowed
        if (isUpdateAvailable && isUpdateAllowed) {
            val updateOptions = AppUpdateOptions.defaultOptions(AppUpdateType.FLEXIBLE)
            appUpdateManager.startUpdateFlow(info, this, updateOptions)
        } else if (info.installStatus() == InstallStatus.DOWNLOADED) {
            appUpdateManager.completeUpdate()
        }
    }
}

private val installStateUpdateListener = InstallStateUpdatedListener { state ->
    if (state.installStatus() == InstallStatus.DOWNLOADED) {
        lifecycleScope.launch {
            updateSnackbarEvent.emit(Unit)
        }
    }
}

override fun onNewIntent(intent: Intent) {
    super.onNewIntent(intent)
    processIntent(intent)
}

private fun processIntent(intent: Intent) {
    when (intent.action) {
        AppNavigator.Action.NEW_OPEN_MESSAGE,
AppNavigator.Action.OLD_OPEN_MESSAGE -> processScrollToMessage(intent)
        AppNavigator.Action.OPEN_CHAT -> processOpenChat(intent)
        else -> processSharedContent(intent)
    }
}

private fun processSharedContent(intent: Intent) {
    getSharedContentIfCan(intent)?.let { sharedContent ->
        mainViewModel.pushSharedContent(sharedContent)
        popNavStackToChat()
        clearIntent()
        analyticsSender.sendEvent(ReceivedSharedContentEvent)
    }
}
```

```
private fun processOpenChat(intent: Intent) {
    getOpenedChatIdIfCan(intent)?.let { chatId ->
        mainViewModel.pushOpenChat(chatId)
        popNavStackToChat()
        clearIntent()
    }
}

private fun processScrollToMessage(intent: Intent) {
    getOpenedMessageIdIfCan(intent)?.let { messageId ->
        mainViewModel.scrollToMessage(messageId)
        popNavStackToChat()
        clearIntent()
        analyticsSender.sendEvent(ReceivedOpenMessageEvent)
    }
}

private fun getOpenedChatIdIfCan(intent: Intent): Long? {
    return intent.getLongExtraOrNull(AppNavigator.Extra.CHAT_ID)
}

private fun getOpenedMessageIdIfCan(intent: Intent): Long? {
    return intent.getLongExtraOrNull(AppNavigator.Extra.MESSAGE_ID)
}

private fun popNavStackToChat() {
    if (::navController.isInitialized) {
        navController.popBackStack(route = Screen.Chat, inclusive = false)
    }
}

private fun clearIntent() {
    this.intent = Intent()
}

private fun getSharedContentIfCan(intent: Intent): SharedContentDomainModel? {
    val text = intent.getStringExtra(Intent.EXTRA_TEXT)
    return when (intent.action) {
        Intent.ACTION_SEND -> {
            val uri = intent.getParcelable<Uri>(Intent.EXTRA_STREAM)
            analyticsSender.sendEvent(ReceivedSharedContentEvent)
            SharedContentDomainModel(text = text, uris = uri?.let { listOf(it) }.orEmpty())
        }
        Intent.ACTION_SEND_MULTIPLE -> {
            analyticsSender.sendEvent(ReceivedSharedContentEvent)
            val uris = intent.getParcelableList<Uri>(Intent.EXTRA_STREAM)
            SharedContentDomainModel(text = text, uris = uris.orEmpty())
        }
        else -> null
```

```
        }
    }

    private fun NavDestination.parseScreenName(): String? {
        return route?.substringAfterLast(".")?.substringBefore("?")?.substringBefore("/")
    }

    override fun onDestroy() {
        super.onDestroy()
        appUpdateManager.unregisterListener(installStateUpdateListener)
    }
}

@Composable
private fun SelfThreadTheme(
    appTheme: AppThemePresentationModel?,
    content: @Composable () -> Unit
) {
    SelfThreadTheme(
        darkTheme = when (appTheme?.type) {
            AppThemeTypePresentationModel.DARK -> true
            AppThemeTypePresentationModel.LIGHT -> false
            else -> isSystemInDarkTheme()
        },
        dynamicColor = appTheme?.enableDynamicColor ?: isDynamicColorAllowed(),
        content = content
    )
}
```

Код класу AddEditChatViewModel

```
import android.net.Uri
import androidx.annotation.StringRes
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.toArgb
import androidx.lifecycle.SavedStateHandle
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.navigation.toRoute
import com.kappdev.selfthread.chat_feature.domain.use_case.GetRandomChatIconColor
import com.kappdev.selfthread.core.analytics.events.ChatCreatedEvent
import com.kappdev.selfthread.core.analytics.events.ChatEditedEvent
import com.kappdev.selfthread.core.analytics.sender.AnalyticsSender
import com.kappdev.selfthread.core.common.coroutine.ApplicationCoroutineScope
import com.kappdev.selfthread.core.common.loadingtask.timedLoadingTask
import com.kappdev.selfthread.core.common.result.ResultWrapper
import com.kappdev.selfthread.core.common.result.ResultWrapper.Failure
import com.kappdev.selfthread.core.common.result.ResultWrapper.Success
import com.kappdev.selfthread.core.common.result.getOrElse
```

```
import com.kappdev.selfthread.core.designsystem.resources.dimens.Constants
import com.kappdev.selfthread.core.emojis.presentation.EmojiPickerViewModelComponent
import com.kappdev.selfthread.core.localization.LocalizableStrings
import com.kappdev.selfthread.core.presentation.navigation.Screen
import com.kappdev.selfthread.shared.chat.domain.usecases.DeleteChatImageUseCase
import com.kappdev.selfthread.shared.chat.domain.usecases.GetChatByIdUseCase
import com.kappdev.selfthread.shared.chat.domain.usecases.UploadChatImageUseCase
import com.kappdev.selfthread.shared.chat.domain.usecases.UpsertChatUseCase
import com.kappdev.selfthread.shared.chat.models.domain.ChatDomainModel
import com.kappdev.selfthread.shared.chat.models.domain.InsertChatDomainModel
import com.kappdev.selfthread.shared.chat.models.presentation.toPresentationModel
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.MutableSharedFlow
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asSharedFlow
import kotlinx.coroutines.flow.update
import kotlinx.coroutines.launch
import kotlinx.coroutines.withContext
import java.time.LocalDateTime
import javax.inject.Inject

@HiltViewModel
class AddEditChatViewModel @Inject constructor(
    getRandomChatIconColor: GetRandomChatIconColor,
    private val upsertChatUseCase: UpsertChatUseCase,
    private val analyticsSender: AnalyticsSender,
    private val uploadChatImageUseCase: UploadChatImageUseCase,
    private val deleteChatImageUseCase: DeleteChatImageUseCase,
    private val getChatByIdUseCase: GetChatByIdUseCase,
    @ApplicationCoroutineScope private val applicationCoroutineScope: CoroutineScope,
    emojiPickerViewModelComponentFactory: EmojiPickerViewModelComponent.Factory,
    savedStateHandle: SavedStateHandle
) : ViewModel() {
    private val _errorsFlow = MutableSharedFlow<Int>()
    val errorsFlow = _errorsFlow.asSharedFlow()

    private val _uiState = MutableStateFlow(
        AddEditChatUiState(
            originalChat = null,
            currentDialog = null,
            chatName = Constants.EMPTY_STRING,
            chatColor = getRandomChatIconColor(),
            chatIconText = Constants.EMPTY_STRING,
            chatImageState = ChatImageState.Empty,
            isLoading = false
        )
    )
```

```
val uiState: StateFlow<AddEditChatUiState> = _uiState

val emojiPickerComponent = emojiPickerViewModelComponentFactory.create(
    coroutineScope = viewModelScope,
    onEmojiSelected = ::updateChatIconText
)

init {
    savedStateHandle.toRoute<Screen.AddEditChat>().chatId?.let { chatId ->
        fetchChat(chatId)
    }
}

fun saveChat(onSuccess: (chatId: Long) -> Unit) {
    viewModelScope.launch(Dispatchers.IO) {
        val state = uiState.value
        val chatName = validateChatNameWithError(state.chatName) ?: return@launch


        val (imageRelativePath, result) = timedLoadingTask(::setLoading) {
            val imageRelativePath = uploadChatImage(state.chatImageState).getOrElse {
                showError(LocalizableStrings.chat_image_upload_fail_error)
                return@launch
            }

            val result = upsertChatUseCase(InsertChatDomainModel(
                id = state.originalChat?.id ?: 0,
                name = chatName,
                backgroundColor = state.chatColor.toArgb(),
                iconText = state.chatIconText,
                imageRelativePath = imageRelativePath,
                order = state.originalChat?.order ?: 0,
                createdAt = state.originalChat?.createdAt ?: LocalDateTime.now()
            ))

            imageRelativePath to result
        }

        result.fold(
            onSuccess = { chatId ->
                launchImageDeletionIfNeed(state.chatImageState)
                sendChatInsertAnalyticsEvent(
                    isNewChat = state.originalChat == null,
                    nameSize = chatName.length,
                    hasEmoji = state.chatIconText.isNotEmpty(),
                    hasPhoto = imageRelativePath != null
                )
                withContext(Dispatchers.Main) { onSuccess(chatId) }
            },
            onFailure = {
```

```
                imageRelativePath?.let(::launchImageDeletion)
                showError(LocalizableStrings.failed_to_save_chat_error)
            }
        )
    }
}

private fun fetchChat(chatId: Long) {
    viewModelScope.launch(Dispatchers.IO) {
        timedLoadingTask(::setLoading) {
            val chat = getChatByIdUseCase(chatId)

            if (chat != null) {
                unpackChat(chat)
            } else {
                showError(LocalizableStrings.chat_not_found)
            }
        }
    }
}

private fun unpackChat(chat: ChatDomainModel) {
    _uiState.update { uiState ->
        val originalChat = chat.toPresentationModel()
        uiState.copy(
            originalChat = originalChat,
            chatName = originalChat.name,
            chatColor = originalChat.backgroundColor,
            chatIconText = originalChat.iconText,
            chatImageState = ChatImageState.from(originalChat.imageRelativePath)
        )
    }
}

private fun uploadChatImage(imageState: ChatImageState): ResultWrapper<String?, Unit> {
    return when (imageState) {
        ChatImageState.Empty -> Success(null)
        is ChatImageState.Removed -> Success(null)
        is ChatImageState.Existing -> Success(imageState.relativePath)
        is ChatImageState.NewlyAdded -> {
            val result = uploadChatImageUseCase(imageState.uri.toString())
            return result.getOrNull()?.let { Success(it) } ?: Failure(Unit)
        }
        is ChatImageState.Replaced -> {
            val result = uploadChatImageUseCase(imageState.newUri.toString())
            return result.getOrNull()?.let { Success(it) } ?: Failure(Unit)
        }
    }
}
```

```
    private fun launchImageDeletion(relativeImagePath: String) {
        applicationCoroutineScope.launch {
            deleteChatImageUseCase(relativePath = relativeImagePath)
        }
    }

    private fun launchImageDeletionIfNeed(imageState: ChatImageState) {
        applicationCoroutineScope.launch(Dispatchers.IO) {
            if (imageState is ChatImageState.Removed) {
                deleteChatImageUseCase(imageState.removedRelativePath)
            } else if (imageState is ChatImageState.Replaced) {
                deleteChatImageUseCase(imageState.oldRelativePath)
            }
        }
    }

    private fun sendChatInsertAnalyticsEvent(isNewChat: Boolean, nameSize: Int, hasEmoji: Boolean,
hasPhoto: Boolean) {
        if (isNewChat) {
            analyticsSender.sendEvent(ChatCreatedEvent(nameSize, hasEmoji, hasPhoto))
        } else {
            analyticsSender.sendEvent(ChatEditedEvent(nameSize, hasEmoji, hasPhoto))
        }
    }

    private fun validateChatNameWithError(chatName: String): String? {
        return validateChatName(chatName).also { validName ->
            if (validName == null) {
                showError(LocalizableStrings.blank_chat_name_error)
            }
        }
    }

    private fun validateChatName(chatName: String): String? {
        return chatName.takeIf(String::isNotBlank)?.trim()
    }

    fun removeImage() {
        _uiState.update { uiState ->
            uiState.copy(chatImageState = uiState.chatImageState.removeImage())
        }
    }

    fun setImageUri(uri: Uri) {
        _uiState.update { uiState ->
            uiState.copy(chatImageState = uiState.chatImageState.updateWith(uri))
        }
    }

    fun openDialog(dialog: AddEditChatDialog) {
```

```
        _uiState.update { uiState ->
            uiState.copy(currentDialog = dialog)
        }
    }

    fun closeDialog() {
        _uiState.update { uiState ->
            uiState.copy(currentDialog = null)
        }
    }

    fun updateChatName(name: String) {
        _uiState.update { uiState ->
            uiState.copy(chatName = name)
        }
    }

    private fun updateChatIconText(text: String) {
        _uiState.update { uiState ->
            uiState.copy(chatIconText = text)
        }
    }

    fun updateChatColor(color: Color) {
        _uiState.update { uiState ->
            uiState.copy(chatColor = color)
        }
    }

    fun showError(@StringRes resId: Int) {
        viewModelScope.launch {
            _errorsFlow.emit(resId)
        }
    }

    fun setLoading(isLoading: Boolean) {
        _uiState.update { uiState ->
            uiState.copy(isLoading = isLoading)
        }
    }
}
```

Код класу AddEditChatUiState

```
import androidx.compose.ui.graphics.Color
import com.kappdev.selfthread.shared.chat.models.presentation.ChatPresentationModel

data class AddEditChatUiState(
    val originalChat: ChatPresentationModel?,
```

```
    val currentDialog: AddEditChatDialog?,
    val chatName: String,
    val chatColor: Color,
    val chatIconText: String,
    val chatImageState: ChatImageState,
    val isLoading: Boolean
) {
    val hasUnsavedChanges = hasUnsavedChanges()

    val hasImage = chatImageState is ChatImageState.NewlyAdded || chatImageState is
ChatImageState.Replaced || chatImageState is ChatImageState.Existing
}

fun AddEditChatUiState.hasUnsavedChanges(): Boolean {
    return originalChat?.let { chat ->
        chatName != chat.name || chatColor != chat.backgroundColor || chatIconText != chat.iconText ||
chatImageState.hasChanged()
    } ?: run {
        chatName.isNotEmpty() || chatIconText.isNotEmpty() || chatImageState.hasChanged()
    }
}
```

Код класу ChatImageState

```
import android.content.Context
import android.net.Uri
import androidx.compose.runtime.Composable
import androidx.compose.runtime.ReadOnlyComposable
import androidx.compose.ui.platform.LocalContext
import androidx.core.net.toUri
import java.io.File

sealed interface ChatImageState {
    data object Empty: ChatImageState

    data class NewlyAdded(val uri: Uri): ChatImageState
    data class Existing(val relativePath: String): ChatImageState
    data class Replaced(val newUri: Uri, val oldRelativePath: String): ChatImageState
    data class Removed(val removedRelativePath: String): ChatImageState

    companion object
}

fun ChatImageState.Companion.from(relativePath: String?): ChatImageState {
    return relativePath?.let { path -> ChatImageState.Existing(path) } ?: ChatImageState.Empty
}

fun ChatImageState.removeImage(): ChatImageState {
    return when (this) {
```

```
        ChatImageState.Empty -> this
        is ChatImageState.NewlyAdded -> ChatImageState.Empty
        is ChatImageState.Existing -> ChatImageState.Removed(removedRelativePath =
this.relativePath)
        is ChatImageState.Replaced -> ChatImageState.Removed(removedRelativePath =
this.oldRelativePath)
        is ChatImageState.Removed -> this
    }
}

fun ChatImageState.updateWith(newUri: Uri): ChatImageState {
    return when (this) {
        ChatImageState.Empty -> ChatImageState.NewlyAdded(newUri)
        is ChatImageState.NewlyAdded -> ChatImageState.NewlyAdded(newUri)
        is ChatImageState.Existing ->  ChatImageState.Replaced(oldRelativePath = this.relativePath,
newUri = newUri)
        is ChatImageState.Replaced -> ChatImageState.Replaced(oldRelativePath =
this.oldRelativePath, newUri = newUri)
        is ChatImageState.Removed -> ChatImageState.Replaced(oldRelativePath =
this.removedRelativePath, newUri = newUri)
    }
}

fun ChatImageState.hasChanged(): Boolean {
    return when (this) {
        is ChatImageState.NewlyAdded, is ChatImageState.Removed, is ChatImageState.Replaced ->
true
        else -> false
    }
}

@Composable
@ReadOnlyComposable
fun ChatImageState.getImageUri(): Uri? {
    return getImageUri(LocalContext.current)
}

fun ChatImageState.getImageUri(context: Context): Uri? {
    return when (this) {
        is ChatImageState.Empty, is ChatImageState.Removed -> null
        is ChatImageState.Existing -> File(context.filesDir, this.relativePath).toUri()
        is ChatImageState.NewlyAdded -> this.uri
        is ChatImageState.Replaced -> this.newUri
    }
}
```

Код екрану чату

```
import android.content.res.Configuration
```

```
import android.net.Uri
import android.widget.Toast
import androidx.activity.compose.BackHandler
import androidx.activity.compose.rememberLauncherForActivityResult
import androidx.activity.result.contract.ActivityResultContracts
import androidx.compose.foundation.layout.ExperimentalLayoutApi
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.WindowInsets
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.imePadding
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.statusBarsIgnoringVisibility
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.foundation.text.KeyboardOptions
import androidx.compose.foundation.verticalScroll
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.rounded.ArrowBack
import androidx.compose.material.icons.rounded.Cancel
import androidx.compose.material.icons.rounded.TaskAlt
import androidx.compose.material3.Button
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.ExtendedFloatingActionButton
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.LocalTextStyle
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.material3.TextField
import androidx.compose.material3.TopAppBar
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.platform.LocalConfiguration
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.platform.LocalDensity
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.input.KeyboardCapitalization
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import androidx.constraintlayout.compose.ConstraintLayout
import androidx.constraintlayout.compose.Dimension
```

```
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.navigation.NavHostController
import androidx.navigation.compose.rememberNavController
import com.kappdev.selfthread.chat_feature.presentation.add_edit_chat.AddEditChatDialog
import com.kappdev.selfthread.chat_feature.presentation.add_edit_chat.AddEditChatViewModel
import com.kappdev.selfthread.chat_feature.presentation.add_edit_chat.getImageUri
import com.kappdev.selfthread.chat_feature.presentation.common.components.ChatIcon
import com.kappdev.selfthread.core.designsystem.components.dialog.LoadingDialog
import com.kappdev.selfthread.core.designsystem.theme.SelfThreadTheme
import com.kappdev.selfthread.core.emojis.presentation.screen.EmojiPickerScreenComponent
import com.kappdev.selfthread.core.localization.LocalizableStrings
import com.kappdev.selfthread.core.presentation.navigation.NavConst
import kotlinx.coroutines.flow.collectLatest

@Preview
@Composable
private fun AddEditChatScreenPreview() {
    SelfThreadTheme {
        AddEditChatScreen(
            navController = rememberNavController()
        )
    }
}

@OptIn(ExperimentalMaterial3Api::class, ExperimentalLayoutApi::class)
@Composable
fun AddEditChatScreen(
    navController: NavHostController,
    chatId: Long? = null,
    viewModel: AddEditChatViewModel = hiltViewModel()
) {
    val context = LocalContext.current
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()
    val emojiPickerScreenComponent = remember {
        EmojiPickerScreenComponent(viewModel.emojiPickerComponent)
    }

    LaunchedEffect(viewModel.errorsFlow) {
        viewModel.errorsFlow.collectLatest { errorRes ->
            Toast.makeText(context, errorRes, Toast.LENGTH_SHORT).show()
        }
    }

    AddEditChatDialogHandler(uiState.currentDialog, onHide = viewModel::closeDialog)

    LoadingDialog(uiState.isLoading)

    emojiPickerScreenComponent()
```

```
    fun navigateUpWithChatId(chatId: Long) {
        navController.previousBackStackEntry?.savedStateHandle?.set(NavConst.OPEN_CHAT_ID,
chatId)
        navController.navigateUp()
    }

    BackHandler(uiState.hasUnsavedChanges) {
        viewModel.openDialog(AddEditChatDialog.SaveUnsavedChanges(
            onSave = { viewModel.saveChat(::navigateUpWithChatId) },
            onDiscard = { navController.navigateUp() }
        ))
    }

    Scaffold(
        modifier = Modifier.imePadding(),
        floatingActionButton = {
            ExtendedFloatingActionButton(
                icon = {
                    Icon(
                        imageVector = Icons.Rounded.TaskAlt,
                        contentDescription = "Done"
                    )
                },
                text = {
                    Text(stringResource(LocalizableStrings.btn_done))
                },
                onClick = {
                    viewModel.saveChat(::navigateUpWithChatId)
                }
            )
        },
        topBar = {
            TopAppBar(
                navigationIcon = {
                    IconButton(
                        onClick = {
                            if (uiState.hasUnsavedChanges) {
                                viewModel.openDialog(AddEditChatDialog.SaveUnsavedChanges(
                                    onSave = { viewModel.saveChat(::navigateUpWithChatId) },
                                    onDiscard = { navController.navigateUp() }
                                ))
                            } else {
                                navController.navigateUp()
                            }
                        }
                    ) {
                        Icon(
                            imageVector = Icons.AutoMirrored.Rounded.ArrowBack,
                            contentDescription = "Go back"
                        )
```

```
                }
            },
            title = {
                Text(
                    text = if (chatId != null)
                        stringResource(LocalizableStrings.edit_chat)
                    else {
                        stringResource(LocalizableStrings.create_chat)
                    }
                )
            }
        )
    }
) { paddingValues ->
    val statusBarHeight = with(LocalDensity.current) {
        WindowInsets.statusBarsIgnoringVisibility.getTop(this).toDp()
    }
    val isLandscape = LocalConfiguration.current.orientation ==
Configuration.ORIENTATION_LANDSCAPE

    ConstraintLayout(
        modifier = Modifier
            .fillMaxSize()
            .padding(paddingValues)
            .verticalScroll(rememberScrollState())
    ) {
        val (chatIcon, imageButton, emojiButton, chatName, colorsPanel) = createRefs()

        ChatIcon(
            color = uiState.chatColor,
            text = uiState.chatIconText,
            imageUri = uiState.chatImageState.getImageUri(),
            modifier = Modifier.size(100.dp).constrainAs(chatIcon) {
                top.linkTo(parent.top)
                start.linkTo(parent.start, if (isLandscape) statusBarHeight + 32.dp else 0.dp)
                if (!isLandscape) {
                    end.linkTo(parent.end)
                }
            }
        )

        ImagePickerButton(
            hasImage = uiState.hasImage,
            onRemove = viewModel::removeImage,
            onPick = viewModel::setImageUri,
            modifier = Modifier.width(ButtonWidth).constrainAs(imageButton) {
                start.linkTo(parent.start, if (isLandscape) statusBarHeight + 16.dp else 0.dp)
                top.linkTo(chatIcon.bottom, 24.dp)
                if (!isLandscape) {
                    end.linkTo(emojiButton.start)
```

```
            }
        }
    )

    EmojiPickerButton(
        onClick = viewModel.emojiPickerComponent::showEmojisSheet,
        modifier = Modifier.width(ButtonWidth).constrainAs(emojiButton) {
            if (isLandscape) {
                start.linkTo(imageButton.start)
                top.linkTo(imageButton.bottom, 16.dp)
            } else {
                start.linkTo(imageButton.end)
                top.linkTo(imageButton.top)
                end.linkTo(parent.end)
            }
        }
    )

    ChatNameField(
        chatName = uiState.chatName,
        onChatNameChange = viewModel::updateChatName,
        modifier = Modifier.constrainAs(chatName) {
            if (isLandscape) {
                start.linkTo(imageButton.end, 32.dp)
                end.linkTo(parent.end, 32.dp)
                top.linkTo(parent.top, 16.dp)
            } else {
                start.linkTo(parent.start, 16.dp)
                end.linkTo(parent.end, 16.dp)
                top.linkTo(imageButton.bottom, 24.dp)
            }
            width = Dimension.preferredValue(450.dp)
        }
    )

    ChatColorsSection(
        selectedColor = uiState.chatColor,
        selectColor = viewModel::updateChatColor,
        modifier = Modifier.constrainAs(colorsPanel) {
            start.linkTo(chatName.start)
            end.linkTo(chatName.end)
            top.linkTo(chatName.bottom, 24.dp)
            width = Dimension.fillToConstraints
        }
    )
  }
 }
}

@Preview
```

```
@Composable
private fun ChatNameFieldPreview() {
    SelfThreadTheme {
        var value by remember { mutableStateOf("") }
        ChatNameField(
            chatName = value,
            modifier = Modifier.width(300.dp),
            onChatNameChange = { value = it }
        )
    }
}

@Composable
private fun ChatNameField(
    chatName: String,
    modifier: Modifier = Modifier,
    onChatNameChange: (String) -> Unit
) {
    val fieldStyle = LocalTextStyle.current.copy(
        color = MaterialTheme.colorScheme.onSurface,
        fontSize = 16.sp,
        lineHeight = 22.sp
    )

    TextField(
        value = chatName,
        shape = RoundedCornerShape(topStart = 10.dp, topEnd = 10.dp),
        modifier = modifier,
        singleLine = true,
        onValueChange = onChatNameChange,
        keyboardOptions = KeyboardOptions(
            capitalization = KeyboardCapitalization.Sentences
        ),
        textStyle = fieldStyle,
        placeholder = {
            Text(
                text = stringResource(LocalizableStrings.hint_chat_name)
            )
        },
        label = {
            Text(
                text = stringResource(LocalizableStrings.label_chat_name)
            )
        },
        trailingIcon = {
            if (chatName.isNotEmpty()) {
                IconButton(
                    onClick = {
                        onChatNameChange("")
                    }
```

```
            ) {
                Icon(
                    imageVector = Icons.Rounded.Cancel,
                    modifier = Modifier.size(20.dp),
                    contentDescription = "Clear chat name"
                )
            }
        }
    }
)
}

@Composable
private fun EmojiPickerButton(
    modifier: Modifier = Modifier,
    onClick: () -> Unit
) {
    Button(
        onClick = onClick,
        modifier = modifier,
        contentPadding = PaddingValues(8.dp)
    ) {
        Text(stringResource(LocalizableStrings.btn_pick_emoji))
    }
}

@Composable
private fun ImagePickerButton(
    hasImage: Boolean,
    modifier: Modifier = Modifier,
    onPick: (Uri) -> Unit,
    onRemove: () -> Unit
) {
    val imagePickerLauncher = rememberLauncherForActivityResult(
        contract = ActivityResultContracts.GetContent(),
        onResult = { uri ->
            uri?.let(onPick)
        }
    )

    Button(
        modifier = modifier,
        contentPadding = PaddingValues(8.dp),
        onClick = {
            if (hasImage) {
                onRemove()
            } else {
                imagePickerLauncher.launch("image/*")
            }
        }
```

```
    ) {
        Text(
            text = if (hasImage) stringResource(LocalizableStrings.btn_remove_image) else
stringResource(LocalizableStrings.btn_upload_image)
        )
    }
}

private val ButtonWidth = 132.dp
```

Код класу SelfThreadDatabase

```
import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import androidx.room.TypeConverters
import androidx.room.migration.Migration
import androidx.sqlite.db.SupportSQLiteDatabase
import com.kappdev.selfthread.core.database.Migrations.MIGRATION_1_2
import com.kappdev.selfthread.core.database.Migrations.MIGRATION_2_3
import com.kappdev.selfthread.core.database.SelfThreadDatabase.Companion.VERSION
import com.kappdev.selfthread.core.database.converters.TextStyleConverter
import com.kappdev.selfthread.shared.chat.data.local.dao.AttachmentDao
import com.kappdev.selfthread.shared.chat.data.local.dao.ChatDao
import com.kappdev.selfthread.shared.chat.data.local.dao.MessageDao
import com.kappdev.selfthread.shared.chat.data.local.dao.MessageTagDao
import com.kappdev.selfthread.shared.chat.data.local.dao.ReminderDao
import com.kappdev.selfthread.shared.chat.models.local.AttachmentEntry
import com.kappdev.selfthread.shared.chat.models.local.ChatEntry
import com.kappdev.selfthread.shared.chat.models.local.LastMessageProjection
import com.kappdev.selfthread.shared.chat.models.local.MessageEntry
import com.kappdev.selfthread.shared.chat.models.local.MessageTagEntry
import com.kappdev.selfthread.shared.chat.models.local.ReminderEntry
import com.kappdev.selfthread.shared.chat.models.local.ReplyMessageProjection
import com.kappdev.selfthread.shared.tags.data.local.dao.TagDao
import com.kappdev.selfthread.shared.tags.models.local.TagEntry

@Database(
    entities = [
        ChatEntry::class,
        MessageEntry::class,
        AttachmentEntry::class,
        ReminderEntry::class,
        TagEntry::class,
        MessageTagEntry::class
    ],
    version = VERSION,
    views = [LastMessageProjection::class, ReplyMessageProjection::class],
```

```
    exportSchema = true
)
@TypeConverters(TextStyleConverter::class)
abstract class SelfThreadDatabase : RoomDatabase() {

    abstract fun getChatDao(): ChatDao
    abstract fun getMessageDao(): MessageDao
    abstract fun getAttachmentDao(): AttachmentDao
    abstract fun getReminderDao(): ReminderDao
    abstract fun getTagDao(): TagDao
    abstract fun getMessageTagDao(): MessageTagDao

    companion object {
        const val NAME = "selfthread_database"
        const val VERSION = 3

        @Volatile
        private var INSTANCE: SelfThreadDatabase? = null

        fun getInstance(context: Context): SelfThreadDatabase {
            return synchronized(this) {
                val currentDb = INSTANCE
                if (currentDb == null || !currentDb.isOpen) {
                    INSTANCE = Room
                        .databaseBuilder(
                            context = context.applicationContext,
                            klass = SelfThreadDatabase::class.java,
                            name = NAME
                        )
                        .addMigrations(MIGRATION_1_2, MIGRATION_2_3)
                        .enableMultiInstanceInvalidation()
                        .build()
                }
                INSTANCE!!
            }
        }

        fun closeDatabase() {
            synchronized(this) {
                INSTANCE?.close()
                INSTANCE = null
            }
        }
    }
}
```

Код класу AttachmentEntry

```
import androidx.room.ColumnInfo
```

```
import androidx.room.Entity
import androidx.room.ForeignKey
import androidx.room.Index
import androidx.room.PrimaryKey
import com.kappdev.selfthread.core.common.datetime.DateTimeUtils
import com.kappdev.selfthread.shared.chat.models.domain.AttachmentDomainModel
import com.kappdev.selfthread.shared.chat.models.domain.AttachmentTypeDomainModel

@Entity(
    tableName = "attachments",
    indices = [
        Index(value = ["chat_id", "attachment_type"]),
        Index(value = ["message_id"])
    ],
    foreignKeys = [
        ForeignKey(
            entity = MessageEntry::class,
            parentColumns = ["id"],
            childColumns = ["message_id"],
            onDelete = ForeignKey.CASCADE
        )
    ]
)
data class AttachmentEntry(
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "id")
    val id: Long,

    @ColumnInfo(name = "chat_id")
    val chatId: Long,

    @ColumnInfo(name = "message_id")
    val messageId: Long,

    @ColumnInfo(name = "attached_at")
    val attachedAt: Long,

    @ColumnInfo(name = "attachment_type")
    val attachmentType: String,

    @ColumnInfo(name = "path")
    val relativePath: String,

    @ColumnInfo(name = "file_name")
    val fileName: String?,

    @ColumnInfo(name = "mime_type")
    val mimeType: String?,

    @ColumnInfo(name = "file_size_in_bytes")
```

```
    val fileSizeInBytes: Long?,

    @ColumnInfo(name = "duration_in_seconds")
    val durationInSeconds: Long?,

    @ColumnInfo(name = "width")
    val width: Int?,

    @ColumnInfo(name = "height")
    val height: Int?
)
```

Код класу ChatEntry

```
import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.Index
import androidx.room.PrimaryKey
import com.kappdev.selfthread.core.common.datetime.DateTimeUtils
import com.kappdev.selfthread.shared.chat.models.domain.ChatDomainModel
import com.kappdev.selfthread.shared.chat.models.domain.InsertChatDomainModel

@Entity(
    tableName = "chats",
    indices = [
        Index(value = ["order"])
    ]
)
data class ChatEntry(
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "id")
    val id: Long,

    @ColumnInfo(name = "name")
    val name: String,

    @ColumnInfo(name = "background_color")
    val backgroundColor: Int,

    @ColumnInfo(name = "icon_text")
    val iconText: String,

    @ColumnInfo(name = "image_path")
    val imageRelativePath: String?,

    @ColumnInfo(name = "order")
    val order: Long,

    @ColumnInfo(name = "created_at")
```

```
    val createdAt: Long
)
```

Код класу MessageEntry

```
import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.ForeignKey
import androidx.room.Index
import androidx.room.PrimaryKey
import com.kappdev.selfthread.shared.chat.models.domain.InsertMessageDomainModel

@Entity(
    tableName = "messages",
    indices = [
        Index(value = ["chat_id", "created_at"]),
        Index(value = ["chat_id", "is_pinned", "created_at"]),
        Index(value = ["reply_to_message_id"])
    ],
    foreignKeys = [
        ForeignKey(
            entity = ChatEntry::class,
            parentColumns = ["id"],
            childColumns = ["chat_id"],
            onDelete = ForeignKey.CASCADE
        ),
        ForeignKey(
            entity = MessageEntry::class,
            parentColumns = ["id"],
            childColumns = ["reply_to_message_id"],
            onDelete = ForeignKey.SET_NULL
        )
    ]
)
data class MessageEntry(
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "id")
    val id: Long,

    @ColumnInfo(name = "chat_id")
    val chatId: Long,

    @ColumnInfo(name = "text")
    val text: String,

    @ColumnInfo(name = "is_pinned")
    val isPinned: Boolean,

    @ColumnInfo(name = "is_checked")
```

```
    val isChecked: Boolean,

    @ColumnInfo(name = "is_opposite")
    val isOpposite: Boolean,

    @ColumnInfo(name = "reply_to_message_id")
    val replyToMessageId: Long?,

    @ColumnInfo(name = "created_at")
    val createdAt: Long,

    @ColumnInfo(name = "styles")
    val styles: List<TextStyleRangeEntry>
)
```

Код класу MessageTagEntry

```
import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.ForeignKey
import androidx.room.Index
import com.kappdev.selfthread.shared.tags.models.local.TagEntry

@Entity(
    tableName = "message_tags",
    primaryKeys = ["message_id", "tag_id"],
    indices = [Index("tag_id"), Index("message_id")],
    foreignKeys = [
        ForeignKey(
            entity = MessageEntry::class,
            parentColumns = ["id"],
            childColumns = ["message_id"],
            onDelete = ForeignKey.CASCADE
        ),
        ForeignKey(
            entity = TagEntry::class,
            parentColumns = ["id"],
            childColumns = ["tag_id"],
            onDelete = ForeignKey.CASCADE
        )
    ]
)
data class MessageTagEntry(
    @ColumnInfo(name = "message_id")
    val messageId: Long,
    @ColumnInfo(name = "tag_id")
    val tagId: Long
)
```

Код класу ReminderEntry

```
import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.ForeignKey
import androidx.room.Index
import androidx.room.PrimaryKey
import com.kappdev.selfthread.core.common.datetime.DateTimeUtils
import com.kappdev.selfthread.shared.chat.models.domain.ReminderDomainModel

@Entity(
    tableName = "reminders",
    indices = [Index(value = ["message_id"], unique = true)],
    foreignKeys = [
        ForeignKey(
            entity = MessageEntry::class,
            parentColumns = ["id"],
            childColumns = ["message_id"],
            onDelete = ForeignKey.CASCADE
        )
    ]
)
data class ReminderEntry(
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "id")
    val id: Long,

    @ColumnInfo(name = "message_id")
    val messageId: Long,

    @ColumnInfo(name = "time_millis")
    val time: Long
)
```

Код класу MessageDao

```
import androidx.paging.PagingSource
import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import androidx.room.Transaction
import androidx.room.Update
import com.kappdev.selfthread.shared.chat.models.domain.EditMessageResultDomainModel
import com.kappdev.selfthread.shared.chat.models.domain.SendMessageResultDomainModel
import com.kappdev.selfthread.shared.chat.models.local.AttachmentEntry
import com.kappdev.selfthread.shared.chat.models.local.HighlightMessageProjection
import com.kappdev.selfthread.shared.chat.models.local.MessageEntry
import com.kappdev.selfthread.shared.chat.models.local.MessageWrapper
```

```
import com.kappdev.selfthread.shared.chat.models.local.NotificationMessageProjection
import com.kappdev.selfthread.shared.chat.models.local.PinnedMessageProjection
import com.kappdev.selfthread.shared.chat.models.local.SearchMessageProjection

@Dao
interface MessageDao {

    @Transaction
    @Query("SELECT * FROM messages WHERE chat_id = :chatId ORDER BY created_at DESC")
    fun getMessagesPaged(chatId: Long): PagingSource<Int, MessageWrapper>

    @Transaction
    @Query("SELECT id, text FROM messages WHERE chat_id = :chatId AND is_pinned = 1
ORDER BY created_at ASC")
    fun getPinnedMessagesPaged(chatId: Long): PagingSource<Int, PinnedMessageProjection>

    @Transaction
    @Query("""
        SELECT m.id, m.text, m.chat_id
        FROM messages AS m
        INNER JOIN reminders AS r ON m.id = r.message_id
        WHERE r.id = :reminderId
        LIMIT 1
    """)
    suspend fun getNotificationMessage(reminderId: Long): NotificationMessageProjection?

    @Transaction
    suspend fun sendMessage(
        message: MessageEntry,
        attachments: List<AttachmentEntry>
    ): SendMessageResultDomainModel {
        val messageId = insertMessage(message)
        val readyAttachments = attachments.map {
            it.copy(chatId = message.chatId, messageId = messageId)
        }
        val attachmentIds = insertAttachments(readyAttachments)
        return SendMessageResultDomainModel(
            messageId = messageId,
            attachmentIds = attachmentIds
        )
    }

    @Transaction
    suspend fun editMessage(
        message: MessageEntry,
        deletedAttachments: List<AttachmentEntry>,
        newAttachments: List<AttachmentEntry>
    ): EditMessageResultDomainModel {
        updateMessage(message)
        val readyAttachments = newAttachments.map {
```

```
            it.copy(chatId = message.chatId, messageId = message.id)
        }
        val attachmentIds = insertAttachments(readyAttachments)
        deleteAttachments(deletedAttachments)
        return EditMessageResultDomainModel(attachmentIds = attachmentIds)
    }

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insertMessage(message: MessageEntry): Long

    @Update
    suspend fun updateMessage(message: MessageEntry): Int

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insertAttachments(attachments: List<AttachmentEntry>): List<Long>

    @Delete
    suspend fun deleteAttachments(attachments: List<AttachmentEntry>): Int

    @Query("DELETE FROM messages WHERE id = :messageId")
    suspend fun deleteMessageById(messageId: Long): Int

    @Query("UPDATE messages SET is_pinned = :isPinned WHERE id = :messageId")
    suspend fun setPinnedStatus(messageId: Long, isPinned: Boolean): Int

    @Query("UPDATE messages SET is_checked = :isChecked WHERE id = :messageId")
    suspend fun setCheckedStatus(messageId: Long, isChecked: Boolean): Int

    @Query("UPDATE messages SET is_opposite = :isOpposite WHERE id = :messageId")
    suspend fun setOppositeStatus(messageId: Long, isOpposite: Boolean): Int

    @Query("UPDATE messages SET reply_to_message_id = :replyMessageId WHERE id
= :originalMessageId")
    suspend fun replyToMessage(originalMessageId: Long, replyMessageId: Long): Int

    @Transaction
    suspend fun updateMessageChat(messageId: Long, newChatId: Long) {
        updateMessageChatId(messageId, newChatId)
        updateAttachmentsChatId(messageId, newChatId)
    }

    @Query("UPDATE messages SET chat_id = :newChatId WHERE id = :messageId")
    suspend fun updateMessageChatId(messageId: Long, newChatId: Long)

    @Query("UPDATE attachments SET chat_id = :newChatId WHERE message_id = :messageId")
    suspend fun updateAttachmentsChatId(messageId: Long, newChatId: Long)

    @Query("SELECT id FROM messages WHERE chat_id = :chatId AND text LIKE '%' || :query ||
'%' ORDER BY created_at DESC")
```

```
    fun findMessagesPaged(chatId: Long, query: String): PagingSource<Int,
SearchMessageProjection>

    @Transaction
    @Query("""
        SELECT * FROM messages AS m
        WHERE m.chat_id = :chatId
          AND (:tagCount = 0 OR EXISTS (
              SELECT 1
              FROM message_tags mt
              WHERE mt.message_id = m.id AND mt.tag_id IN (:tagIds)
          ))
        ORDER BY m.created_at DESC
    """)
    fun getMessagesPagedFiltered(
        chatId: Long,
        tagIds: List<Long>,
        tagCount: Int
    ): PagingSource<Int, MessageWrapper>

    @Query("""
        SELECT m.id
        FROM messages AS m
        WHERE m.chat_id = :chatId
          AND (:query = '' OR m.text LIKE '%' || :query || '%')
          AND (:tagCount = 0 OR EXISTS (
              SELECT 1
              FROM message_tags mt
              WHERE mt.message_id = m.id AND mt.tag_id IN (:tagIds)
          ))
        ORDER BY m.created_at DESC
    """)
    fun findMessagesPagedFiltered(
        chatId: Long,
        query: String,
        tagIds: List<Long>,
        tagCount: Int
    ): PagingSource<Int, SearchMessageProjection>

    @Transaction
    suspend fun getMessageHighlightInfo(messageId: Long, tagIds: List<Long>):
HighlightMessageProjection? {
        val chatId = getMessageChatId(messageId) ?: return null

        val messageCount = countMessagesBefore(chatId, messageId, tagIds, tagIds.size)
        val dateHeaderCount = countDateHeadersBefore(chatId, messageId, tagIds, tagIds.size)
            .minus(1)
            .coerceAtLeast(0)

        return HighlightMessageProjection(
```

```
            id = messageId,
            chatId = chatId,
            messageIndexInTheChat = messageCount + dateHeaderCount
        )
    }

    @Query("""
        SELECT COUNT(DISTINCT date(created_at / 1000, 'unixepoch'))
        FROM messages AS m
        WHERE chat_id = :chatId
              AND (:tagCount = 0 OR EXISTS (
            SELECT 1
            FROM message_tags mt
            WHERE mt.message_id = m.id AND mt.tag_id IN (:tagIds)
         ))
        AND created_at > (SELECT created_at FROM messages WHERE id = :messageId)
    """)
    suspend fun countDateHeadersBefore(chatId: Long, messageId: Long, tagIds: List<Long>,
tagCount: Int): Int

    @Query("""
        SELECT COUNT(*)
        FROM messages AS m
        WHERE chat_id = :chatId
        AND (:tagCount = 0 OR EXISTS (
            SELECT 1
            FROM message_tags mt
            WHERE mt.message_id = m.id AND mt.tag_id IN (:tagIds)
         ))
        AND created_at > (SELECT created_at FROM messages WHERE id = :messageId)
    """)
    suspend fun countMessagesBefore(chatId: Long, messageId: Long, tagIds: List<Long>, tagCount:
Int): Int

    @Query("SELECT chat_id FROM messages WHERE id = :messageId")
    suspend fun getMessageChatId(messageId: Long): Long?
}
```

ДОДАТОК В

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

Хмельницький національний університет

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Кваліфікаційна робота, на тему:

Мобільний застосунок для створення й збереження нотаток у форматі особистого чату на платформі Android

Виконав

Хотич Микола Володимирович

студент IV курсу, групи ІПЗ-22-1

Керівник

Бедратюк Леонід Петрович

доктор фізико-математичних наук, професор

Хмельницький 2026

Актуальність

11,1 → 23,8

млрд $ — прогнозоване зростання глобального ринку застосунків для нотаток до 2029 р., CAGR 21,0% (Research and Markets, 2025)

Три ключові потреби користувача, які наявні рішення задовольняють лише частково

01

Швидкість фіксації

Мінімум дій для створення запису — без вибору папки, заголовка чи тегів перед введенням тексту

02

Структурування

Розподіл нотаток за тематичними чатами, теги, нагадування та можливість пошуку

03

Конфіденційність

Локальне зберігання усіх даних на пристрої без передачі на сторонні сервери

Мета та завдання

Мета

Розроблення локального мобільного застосунку для платформи Android, що надає можливість створювати нотатки у форматі тематичних чатів — із високою швидкістю фіксації, ефективною категоризацією та гарантованою конфіденційністю завдяки локальному зберіганню даних.

Завдання

01 Проаналізувати предметну область та виявити її структурні і функціональні особливості

02 Виконати порівняльний аналіз наявних рішень, визначити їхні переваги та обмеження

03 Сформулювати функціональні та нефункціональні вимоги до застосунку

04 Спроєктувати архітектуру та обґрунтувати вибір архітектурного рішення

05 Розробити концепцію інтерфейсу користувача та візуального дизайну

06 Розробити алгоритми ключових процесів функціонування застосунку

07 Обґрунтувати вибір технологічного стеку та інструментальних засобів

08 Виконати програмну реалізацію та провести тестування

Аналіз аналогів

Критерій	Telegram Saved Messages	Google Keep	Obsidian
Формат інтерфейсу	**Чат ✓**	Картки та списки	Текстовий редактор
Швидкість створення запису	**Висока ✓**	Середня	Низька
Категоризація	Відсутня	**Теги, кольори ✓**	**Папки, теги ✓**
Архітектура збереження	Хмарна	Хмарна	**Локальна ✓**
Складність інтерфейсу	**Низька ✓**	Середня	Висока

> Жодне рішення одночасно не задовольняє всі три ключові потреби. Проєктований застосунок реалізує гібридний підхід: діалоговий інтерфейс із месенджерів + категоризація тегами та кольорами з Google Keep + локальне зберігання з Obsidian.

Предметна область та варіанти використання

Досліджено еволюцію процесу ведення нотаток від фізичних носіїв до цифрових систем управління знаннями. На основі аналізу функціональні вимоги формалізовано засобами UML.

Основні варіанти використання

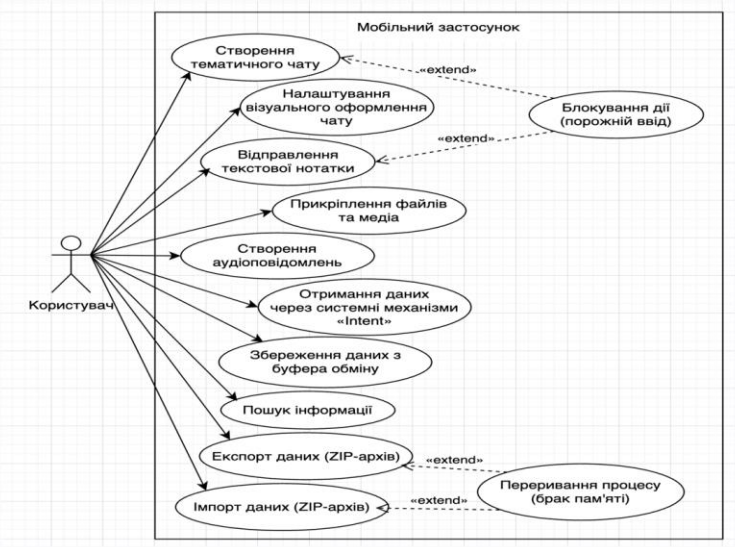
— Створення тематичних чатів і налаштування оформлення

— Відправлення текстових нотаток, прикріплення файлів і медіа

— Створення аудіоповідомлень

— Отримання даних через системні механізми

— Пошук інформації, експорт та імпорт даних



Вимоги до програмного забезпечення

Функціональні

— Створення тематичних чатів з індивідуальним візуальним оформленням

— Відправлення, редагування та видалення повідомлень з форматуванням

— Відповіді, закріплення та позначення повідомлень як виконаних

— Встановлення нагадувань зі сповіщеннями у визначений час

— Додавання та перегляд мультимедіа (зображення, відео, аудіо, файли)

— Призначення користувацьких тегів і фільтрація за ними

— Пошук у межах чату, обмін інформацією зі сторонніми застосунками

— Формування й відновлення резервних копій

Нефункціональні

— Підтримка операційної системи Android 8.0 та новіших версій

— Виключно локальне зберігання текстових записів і медіа

— Час холодного запуску не більше двох секунд

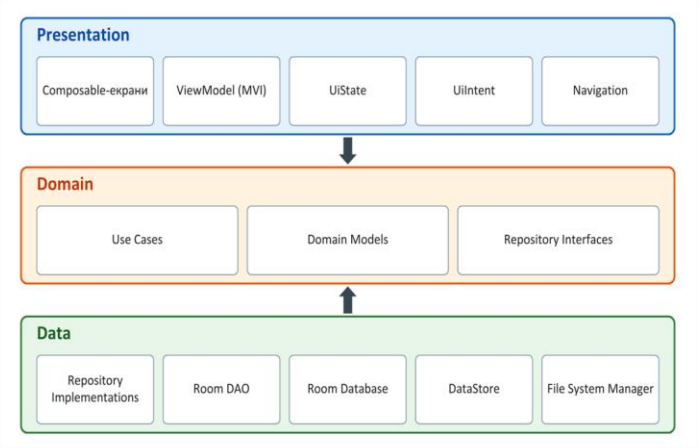
— Візуальне оформлення на принципах Material Design 3

— Обфускація програмного коду для захисту від зворотної інженерії

— Адаптивність інтерфейсу під різні розміри екранів

— Інтуїтивно зрозумілі патерни взаємодії, звичні мобільним користувачам

Архітектура та UI-патерн



Загальна структура

Clean Architecture

Три шари — Presentation, Domain, Data — обрано для забезпечення:

— тестовності бізнес-логіки
— незалежності від платформних залежностей
— масштабованості та зручного супроводу коду

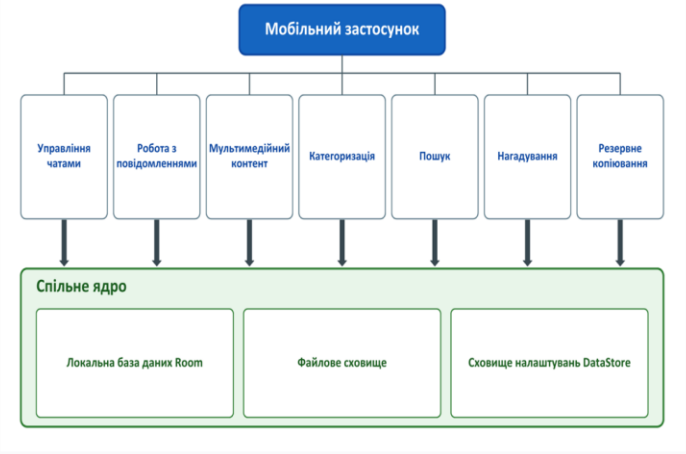
UI-патерн

MVI

Незмінний стан екрана та однонаправлений потік даних забезпечують:

— передбачувану поведінку інтерфейсу
— спрощену локалізацію помилок
— природну сумісність з реактивними потоками

Функціональна декомпозиція

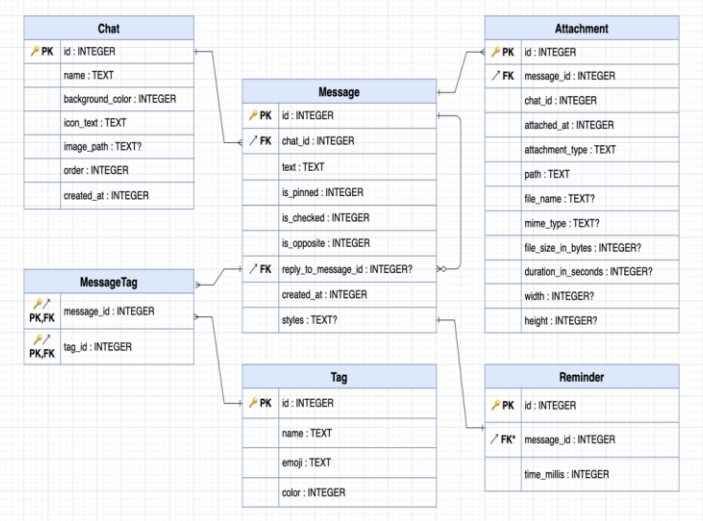


Код групується навколо функціональних можливостей, а не технічних категорій. Кожен модуль інкапсулює компоненти всіх трьох шарів і взаємодіє зі спільним ядром через інтерфейси.

Сім функціональних модулів

— Управління чатами
— Робота з повідомленнями
— Мультимедійний контент
— Категоризація через теги
— Пошук
— Нагадування
— Резервне копіювання

Проєктування бази даних



Сутності

6

Chat, Message, Attachment, Tag, MessageTag, Reminder

Нормалізація

3НФ

Усунення надлишковості даних, цілісність та узгодженість записів

Цілісність даних

Каскадні правила видалення зберігають консистентний стан бази. Індекси оптимізовано для частих запитів інтерфейсу.

Аналіз і вибір технологій

Мова

Kotlin

Офіційно рекомендована мова для нових Android-проєктів; виразність, безпека від null

Середовище

Android Studio

Офіційне IDE для Android-розробки; інструменти профілювання, налагодження та емуляції

UI-фреймворк

Jetpack Compose

Декларативний підхід; узгодженість з патерном MVI та реактивними потоками

Асинхронність

Coroutines + Flow

Структурована паралельність; реактивні потоки даних для оновлення інтерфейсу

База даних

Room

Офіційна ORM-абстракція над SQLite; перевірка SQL на етапі компіляції

Впровадження залежностей

Hilt

Впровадження залежностей через конструктор; інтеграція з Compose та ViewModel

Зображення

Coil 3

Інтеграція з Jetpack Compose; менший розмір бінарного коду, ніж у альтернатив

Медіа

AndroidX Media3

Офіційний наступник ExoPlayer; готові Compose-компоненти; широкий набір форматів

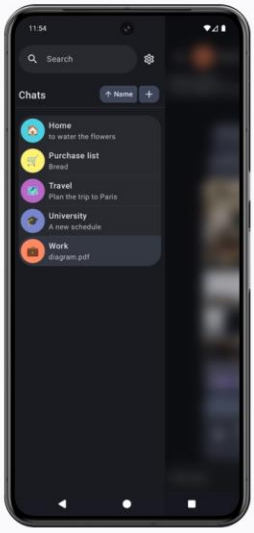
Дизайн

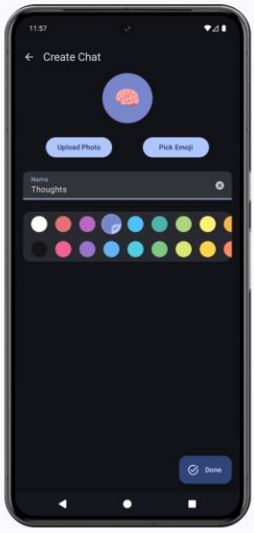
Material Design 3

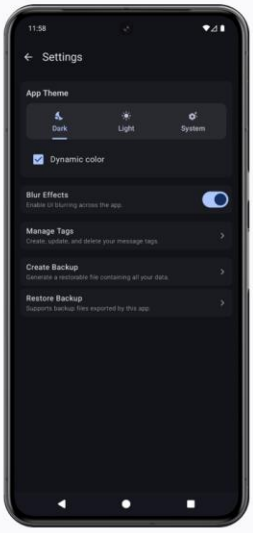
Офіційна мова дизайну Android; підтримка динамічних кольорів Material You

Програмна реалізація та інтерфейс

Розроблений застосунок повністю реалізує усі визначені функціональні вимоги. Інтерфейс відповідає принципам Material Design 3 та підтримує темну, світлу і динамічну теми оформлення.







Вимоги до технічного забезпечення

Платформа

Android 8+

— API 26 — мінімальний рівень
— API 36 (Android 16) — цільовий рівень
— Архітектури ARM, x86, x86_64

Апаратні вимоги

2 ГБ RAM

— Чотириядерний процесор
— 16 ГБ постійного сховища
— Без вимог до датчиків та модулів зв'язку

Вільна пам'ять

2 ГБ

— Об'єм застосунку — 30 МБ
— Локальна база даних користувача
— Запас для мультимедійних вкладень

Доступність

Працює повністю без підключення до мережі Інтернет — усі дані зберігаються виключно на пристрої користувача.

Дозволи

POST_NOTIFICATIONS — для нагадувань;

SCHEDULE_EXACT_ALARM — для точного часу спрацювання;

RECORD_AUDIO — для голосових повідомлень.

Спосіб розповсюдження

Інсталяційний пакет APK для прямого встановлення або публікація через офіційний магазин Google Play.

Тестування та результати

Проведено ручне функціональне та юзабіліті-тестування за принципом «чорної скриньки» на реальному мобільному пристрої Android та на емуляторі. Поведінка ідентична в межах усіх перевірених сценаріїв.

Група 01

Керування чатами

Створення, редагування, валідація порожньої назви, завантаження фону, підтвердження виходу

Група 02

Робота з повідомленнями

Текст, прикріплення зображень, відео, голосових повідомлень, відповіді та закріплення

Група 03

Теги, нагадування, пошук

Створення тегів, фільтрація, видалення з каскадом, нагадування, повнотекстовий пошук

Група 04

Налаштування та бекап

Створення резервної копії, відновлення, перемикання теми, обробка системних дозволів

Дефект 1 · усунено

Некоректне відображення відеозаписів у портретній орієнтації

Картка повідомлення показувала відео з неправильним співвідношенням сторін. Додано коректне врахування орієнтації запису при відображенні.

Дефект 2 · усунено

Зупинка оновлень інтерфейсу після створення резервної копії

Стрічка повідомлень переставала оновлюватися. Додано механізм автоматичного поновлення зв'язку та підписників для бази даних після завершення бекапу.

Висновки

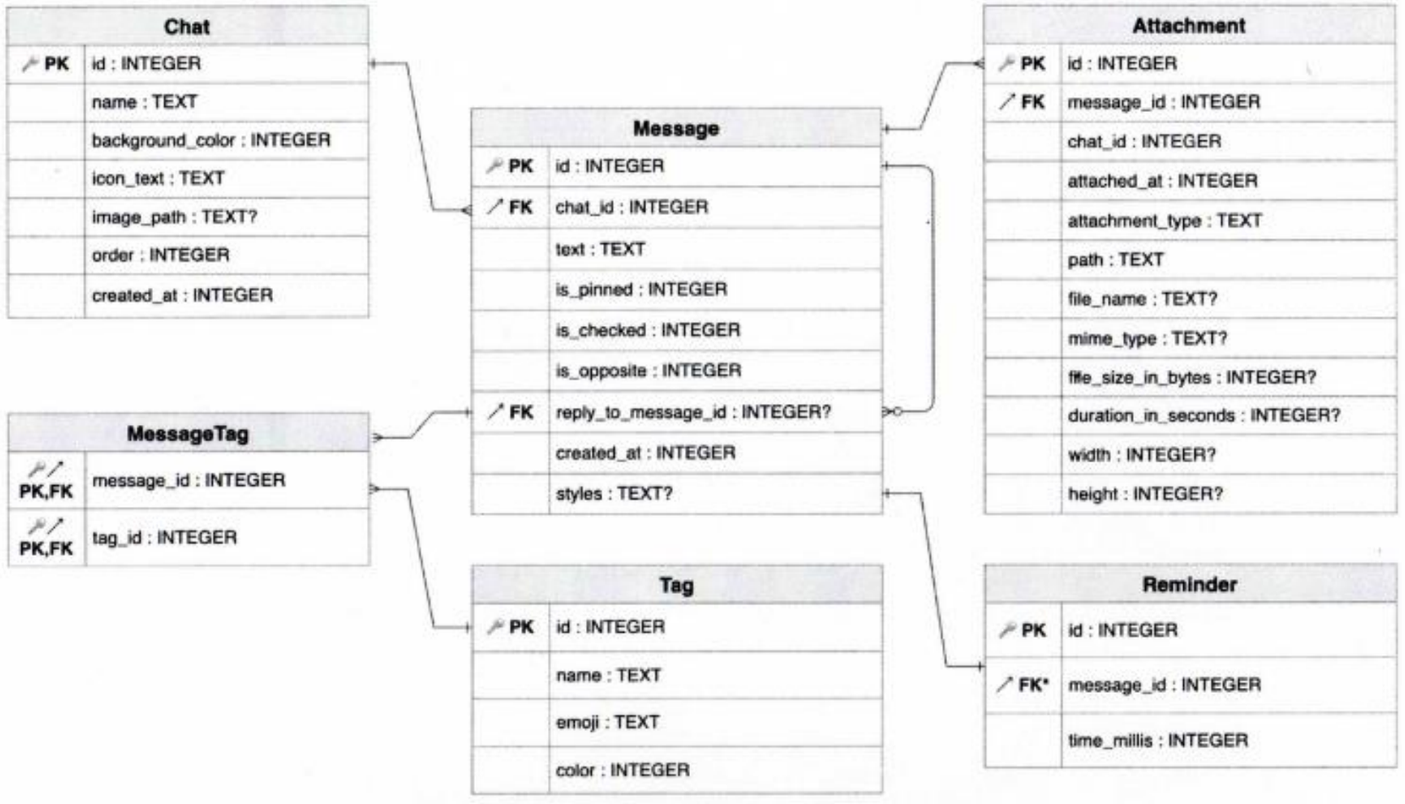
№	Завдання	Виконано
01	**Аналіз предметної області**	Досліджено еволюцію процесу ведення нотаток, виокремлено три ключові потреби
02	**Порівняльний аналіз аналогів**	Проаналізовано Telegram, Google Keep, Obsidian; обґрунтовано гібридний підхід
03	**Функц. та нефункц. вимоги**	Сформовано вимоги, оформлено детальне технічне завдання
04	**Архітектура застосунку**	Спроєктовано Clean Architecture з трьома шарами та UI-патерном MVI
05	**Інтерфейс і візуальний дизайн**	Розроблено макети та інтерактивний прототип за принципами Material Design 3
06	**Алгоритми ключових процесів**	Деталізовано блок-схеми: створення чату, надсилання повідомлень, аудіозапис
07	**Вибір технологічного стеку**	Обґрунтовано Kotlin, Android Studio, Jetpack Compose, Room, Hilt, Coil 3, Media3
08	**Програмна реалізація та тестування**	Реалізовано усі модулі; виявлено та усунено усі дефекти

Розроблений застосунок є функціонально завершеним продуктом, що реалізує концепцію мінімально життєздатного продукту — забезпечує оперативну та конфіденційну фіксацію інформації.

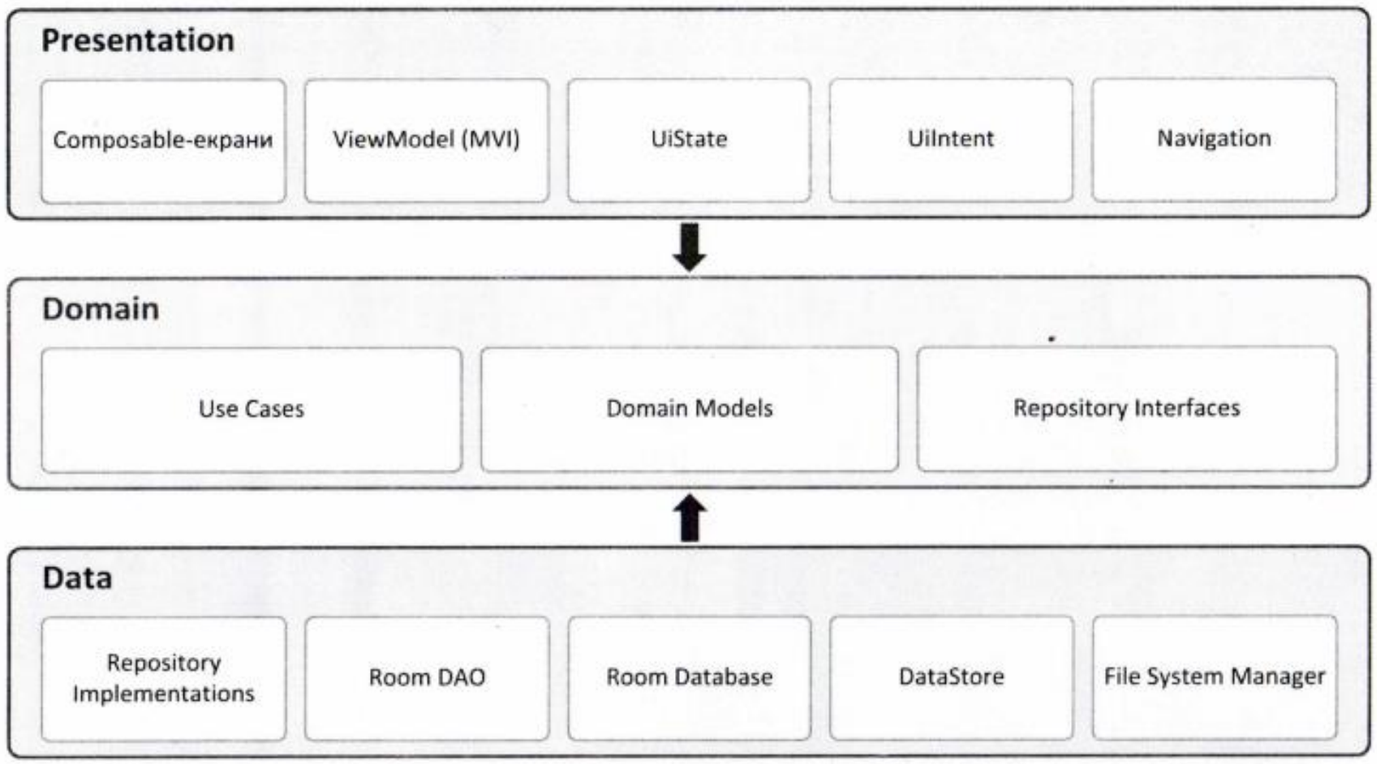
Дякую за увагу

ГРАФІЧНА ЧАСТИНА

					КвРІПЗ. 2201114.01.19.Е2			
Змн.	Лист	№ докум.	Підпис	Дат	Мобільний застосунок для створення й збереження нотаток у форматі особистого чату на платформі Android	Літ.	Маса	Масштаб
Виконав		Хотич М.В.		31.05.2026				
Керівник		Бедратюк Л.П.		1.06.2026		Арк. 1	Аркушів 3	
Н. контр.		Бойко В.О.		1.06.2026	Діаграма варіантів використання	ХНУ, ІПЗ–22–1		
Зав. каф.		Бедратюк Л.П.		1.06.2026				



					КвРІПЗ. 2201114.01.19.Е1		
Змн.	Лист	№ докум.	Підпис	Дат	Мобільний застосунок для створення й збереження нотаток у форматі особистого чату на платформі Android	Літ.	Маса / Масштаб
Виконав		Хотич М.В.		3.06.2026			
Керівник		Бедратюк Л.П.		1.06.2026		Арк. 2	Аркушів 3
Н. контр.		Бойко В.О.		1.06.2026	Логічна модель бази даних	ХНУ, ІПЗ–22–1	
Зав. каф.		Бедратюк Л.П.		1.06.2026			



					КвРІПЗ. 2201114.01.19.Е1			
Змн.	Лист	№ докум.	Підпис	Дат	Мобільний застосунок для створення й збереження нотаток у форматі особистого чату на платформі Android	Літ.	Маса	Масштаб
Виконав		Хотич М.В.		28.05.2026				
Керівник		Бедратюк Л.П.		01.06.2026		Арк. 3	Аркушів 3	
Н. контр.		Бойко В.О.		1.06.2026	Шарова архітектура застосунку	ХНУ, ІПЗ–22–1		
Зав. каф.		Бедратюк Л.П.		1.06.2026				

СУПРОВІДНІ ДОКУМЕНТИ

Завідувачу кафедри інженерії програмного
забезпечення проф. Леоніду БЕДРАТЮКУ
здобувача вищої освіти
Хотича Миколи Володимировича
факультет ІТ, IV курс, 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, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений 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.

02.09.2025
дата

підпис

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

Максимальне співпадіння з одним документом 3.0%

Словники перевірки: UA, US, RU. **Помилок в документах: 13%**

ID: 272658 Назва: БКР_Мобільний застосунок для створення й збереження нотаток у формат особистого чату на платформі Android Додано в БД: 2026-05-28 Автора: Микола ХОТИЧ Керівники: д-р фіз.-мат. наук, професор. Леонід БЕДРАТЮК Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	102153	830	8017 (8%)	92 (11%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

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: Микола ХОТИЧ

Співавтор:

Назва: Мобільний застосунок для створення й збереження нотаток у формат особистого чату на платформі Android

Науковий керівник: д-р фіз.-мат. наук, професор.Леонід БЕДРАТЮК

Підрозділ: Кафедра інженерії програмного забезпечення

Коефіцієнт подібності 1:6.79%

Коефіцієнт подібності 2:1.04%

Мікропробіли: 0

Заміна букв: 17

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2026-05-28 18:25:05.0

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

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ
освітнього ступеня «Бакалавр»

Дипломник Хотич Микола Володимирович

Тема Мобільний застосунок для створення й збереження нотаток у форматі особистого чату на платформі Android

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг кваліфікаційної роботи:

Кількість листів креслень 3; кількість сторінок записки 74

1.Короткий зміст пояснювальної записки та прийнятих рішень У кваліфікаційній роботі досліджено предметну область, визначено функціональні та нефункціональні вимоги. Проведено аналіз існуючих застосунків на ринку, розглянуто їх переваги і недоліки та доведено актуальність розробки нового програмного забезпечення. Розглянуто інструменти для реалізації спроєктованих рішень, у результаті чого створено застосунок. Також проведено тестування програми, за результатами якого підтверджено коректну роботу розробленого застосунку.

2. Висновок про відповідність роботи поставленому завданню Кваліфікаційна робота виконана відповідно до поставленого завдання та з дотриманням усіх 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 проєктування. У першому розділі проведено аналіз предметної області, розглянуто існуючі рішення та визначено функціональні і нефункціональні вимоги до розроблюваного програмного забезпечення. У другому розділі обрано та обґрунтовано архітектуру Clean Architecture з патерном MVI, спроєктовано структуру бази даних, визначено технологічний стек: Kotlin, Jetpack Compose, Room, Hilt. У третьому розділі виконано програмну реалізацію модулів застосунку та проведено функціональне тестування, за результатами якого підтверджено коректну роботу програми.

4. Позитивні сторони роботи Тематика кваліфікаційної роботи є актуальною, оскільки застосунки формату «особистий чат для нотаток» мають значний попит серед користувачів мобільних платформ. Застосовано новітні технології для побудови програмного продукту та актуальні архітектурні рішення.

5. Негативні сторони роботи _ У роботі відсутня реалізація хмарної синхронізації даних між пристроями — було б доцільно передбачити цю можливість. Також варто було б розширити розділ тестування результатами нефункціонального тестування.

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

7. Відгук про кваліфікаційну роботу в цілому _ Кваліфікаційна робота заслуговує на позитивну оцінку. Матеріал пояснювальної записки структурований, послідовний та чіткий, що дозволяє зрозуміти викладений матеріал у рамках тематики проєктування. Графічний матеріал дає можливість наочно побачити деталі проєктування системи.

8. Інші зауваження

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

РЕЦЕНЗЕНТ Кисіль Ольгерд Миколайович
к. ф.-м. н., доцент кафедри КІТС

"28" 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: «Мобільний застосунок для створення й збереження нотаток у форматі особистого чату на платформі Android»
Автор: Хотич Миколи Володимирович
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Спеціальність: 121 – Інженерія програмного забезпечення
Науковий керівник: Бедратюк Леонід Петрович, д-р фіз.-мат. наук, професор.
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 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, визначений системою Anti-Plagiarism, складає 3.0% з одного джерела. Загальна сумарна подібність у базі даних складає 8% за символами та 11% за лексемами. Крім того, за результатами додаткового аналізу коефіцієнт подібності 1 становить 6.79%, коефіцієнт подібності 2 1.04%. Не виявлено мікропробілів, зайвих білих знаків або маніпуляцій з інтервалами. З урахуванням наведених обгрунтувань, 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 _____ Леонід БЕДРАТЮК