Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

Бакалаврська кваліфікаційна робота

на тему: Розробка мобільного застосунку для організації та менеджменту особистих завдань

Виконав: студент ___IV__ курсу
групи ___5ПІ-21б____
спеціальності
121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Цимбал Максим Юрійович
(прізвище та ініціали)
Керівник: к.т.н., доц. каф. ПЗ Черноволик Г. О.
(прізвище та ініціали)
Рецензент: к.т.н., доц. каф ОТ Тарновський М. Г.
(прізвище та ініціали)

Допущено до захисту

Зав. кафедри ПЗ О. Н. Романюк
«09» червня 2025 р.

ВНТУ – 2025

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти перший (бакалаврський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
_____ О. Н. Романюк
«21» березня 2025 р.

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Цимбалу Максиму Юрійовичу

1. Тема роботи – розробка мобільного застосунку для організації та менеджменту особистих завдань. Керівник роботи: Черноволик Галина Олександрівна, к.т.н техн. наук, затверджені наказом вищого навчального закладу від 20 березня 2025 р. №97.
2. Строк подання студентом роботи 2 червня 2025
3. Вихідні дані до роботи: текстові дані для нотаток і чеклістів, структуровані списки завдань, параметри кастомізації інтерфейсу (теми, мови, шрифти), звукові дані для створення чеклістів, дані користувацьких налаштувань, хмарні дані для синхронізації та бекапу. Максимальний обсяг текстових даних не обмежений, підтримка офлайн- та онлайн-режимів, дані про структуру чеклістів, створених на основі текстового чи звукового вводу, налаштування для адаптивного інтерфейсу.
4. Зміст текстової частини: аналіз сучасних технологій управління особистими завданнями, методи створення та редагування нотаток і чеклістів, методи кастомізації інтерфейсу, алгоритми обробки вводу для створення чеклістів, методи хмарної синхронізації та бекапу даних, аналіз архітектури MVVM для мобільних застосунків, розробка адаптивного графічного інтерфейсу, розробка модуля створення та редагування нотаток і чеклістів, розробка модуля інтелектуального створення чеклістів на основі звуку, розробка модуля хмарного бекапу, розробка структури програми, аналіз методів тестування мобільних застосунків, результат тестування системи, розробка інструкцій користувача.
5. Перелік ілюстративної частини: основні етапи алгоритмів створення та редагування нотаток і чеклістів, алгоритми обробки вводу для створення чеклістів, схема хмарного бекапу, діаграма MVVM-архітектури застосунку, інтерфейс головного екрану зі списками нотаток і чеклістів, інтерфейс

налаштувань кастомізації, інтерфейс створення чеклістів з вводу, результати тестування системи.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
1-4	Черноволик Г. О., к.т.н., доц. каф. ПЗ.	24.03.25	01.06.25

7. Дата видачі завдання 24 березня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз підходів для систем менеджменту особистих завдань	25.03.25-03.04.25	вик.
2	Розробка архітектури мобільного застосунку	03.04.25-13.04.25	вик.
3	Розробка алгоритмів роботи програмного забезпечення	13.04.25-20.04.25	вик.
4	Розробка мобільного програмного забезпечення	20.04.25-10.05.25	вик.
5	Тестування програмного забезпечення	10.05.25-22.05.25	вик.
6	Оформлення матеріалів до захисту БКР	22.05.25-30.05.25	вик.

Студент (підпис) **М.Ю. Цимбал** (ініціали та прізвище)

Керівник бакалаврської кваліфікаційної роботи (підпис) **Г.О. Черноволик** (ініціали та прізвище)

АНОТАЦІЯ

УДК 004.4

Цимбал М.Ю. Розробка мобільного застосунку для організації та менеджменту особистих завдань: бакалаврська кваліфікаційна робота зі спеціальності 121 Інженерія програмного забезпечення, освітня програма – інженерія програмного забезпечення. Вінниця: ВНТУ, 2025.76 с.

На укр. мові. Бібліогр.: 23 назв; рис.: 35; табл.: 2.

У бакалаврській кваліфікаційній роботі розроблено програмні засоби, алгоритми та структури даних для теми “Розробка мобільного застосунку для організації та менеджменту особистих завдань із підтримкою хмарного бекапу та інтелектуального створення чеклістів”, обґрунтовано актуальність теми та доцільність розробки даної системи, а також проведено тестування програми для виявлення недоліків та їх усунення.

Було використано такі технології для реалізації системи: мова програмування Kotlin, середовище розробки Android Studio, бібліотека Jetpack Compose для створення адаптивного інтерфейсу, Room для локального зберігання даних, Firebase для хмарного бекапу та синхронізації, Hilt для ін’єкції залежностей, Coroutines і Flow для асинхронних операцій, а також бібліотеки для обробки вводу.

Реалізовано мобільний застосунок, який забезпечує створення, редагування та кастомізацію нотаток і чеклістів, інтелектуальне створення чеклістів на основі текстового та звукового вводу, а також хмарне зберігання даних із можливістю синхронізації.

Ключові слова: мобільний застосунок, управління завданнями, хмарний бекап, інтелектуальне створення чеклістів, обробка звукового вводу, Kotlin, Jetpack Compose, Firebase, архітектура MVVM.

ABSTRACT

UDC 004.4

Tsymbal M. Y. Development of a mobile application for organization and management of personal tasks: bachelor's thesis in specialty 121 – Software Engineering, educational program – Software Engineering. Vinnytsia: VNTU, 2025. 76 p.

In Ukrainian. Bibliography: 23 titles; Figures: 35; Tables: 2.

In the bachelor's thesis, software tools, algorithms, and data structures were developed for the topic "Development of a mobile application for organization and management of personal tasks". An analysis of modern technologies for personal task management was conducted, and the necessity of developing such a system to enhance user productivity and data accessibility was substantiated. Methods for creating and editing notes and checklists, customizing the user interface (themes, languages, fonts), processing audio input for checklist creation, and implementing cloud synchronization and backup were developed. The MVVM architecture was analyzed and applied to ensure modularity and scalability of the application.

The system was implemented using the Kotlin programming language, Android Studio, Jetpack Compose for adaptive UI, Room for local data storage, Firebase for cloud backup and synchronization, Hilt for dependency injection, Coroutines and Flow for asynchronous operations, and libraries for audio input processing. The developed application enables the creation, editing, and customization of notes and checklists, intelligent checklist generation from text input, and secure cloud-based data storage with synchronization capabilities.

Testing confirmed the reliability of the developed methods and their implementation. The application can be used for personal task management, improving efficiency in organizing daily activities.

Keywords: mobile application, task management, cloud backup, intelligent checklist creation, audio input processing, Kotlin, Jetpack Compose, Firebase, MVVM architecture.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження. Сучасний етап розвитку інформаційних технологій характеризується стрімким зростанням попиту на ефективні, інтуїтивно зрозумілі та багатофункціональні інструменти управління особистими завданнями, які забезпечують максимальну зручність, доступність, безпеку даних і адаптивність до потреб користувачів. Традиційні методи організації завдань, такі як паперові нотатки, календарі чи прості цифрові списки, часто виявляються обмеженими через низку недоліків, зокрема відсутність гнучкості, неможливість синхронізації між різними пристроями, обмежені можливості автоматизації та інтеграції з іншими сервісами. У цьому контексті розробка сучасних мобільних застосунків, здатних інтегрувати інтелектуальні функції, такі як автоматичне створення структурованих чеклістів на основі текстового чи звукового вводу, а також використання хмарного зберігання для забезпечення безперебійного доступу до даних, відкриває нові перспективи для підвищення продуктивності та зручності користувачів у різних сферах їхнього життя [1].

Мобільні застосунки для управління завданнями надають користувачам можливість створювати структуровані списки справ, налаштовувати інтерфейс відповідно до індивідуальних вподобань (наприклад, вибір тем оформлення, мовної локалізації, розміру шрифтів чи інших елементів дизайну), а також забезпечують безперебійний доступ до даних у будь-який час і з будь-якого пристрою завдяки хмарним технологіям [2]. Інноваційний підхід, який передбачає інтелектуальне створення чеклістів на основі довільного текстового вводу чи голосових команд, значно спрощує процес введення інформації, зменшує час, витрачений на організацію завдань, і підвищує загальну зручність використання застосунку. Такі функції мають значний потенціал для застосування в повсякденному житті, освітніх процесах, бізнес-середовищі, управлінні проектами та інших сферах, де швидка, ефективна та структурована організація завдань відіграє ключову роль у досягненні успіху [3].

Точність і надійність обробки даних, зокрема при створенні чеклістів на основі звукового вводу, є критично важливими аспектами для забезпечення високої якості користувацького досвіду та довіри до застосунку. Використання хмарного бекапу та синхронізації дозволяє не лише уникнути втрати даних у разі технічних збоїв чи втрати пристрою, а й забезпечує безперервний доступ до інформації на різних платформах, що є однією з ключових вимог сучасних користувачів у цифрову еру. На відміну від наявних аналогів, таких як прості менеджери завдань, які часто обмежуються базовими функціями створення списків і не підтримують інтелектуальну обробку звукового вводу чи повноцінну хмарну синхронізацію, запропонований підхід поєднує передові технології та орієнтацію на користувача. Таким чином, розробка мобільного застосунку з інтелектуальними функціями та хмарною інфраструктурою є актуальною задачею, яка має високу практичну значимість і може суттєво покращити досвід користувачів у процесі організації їхньої діяльності.

Зв'язок роботи з науковими програмами, планами, темами. Робота виконувалася згідно з планом виконання наукових досліджень на кафедрі програмного забезпечення Вінницького національного технічного університету.

Мета та завдання дослідження. Метою дослідження є підвищення продуктивності та зручності користувацького досвіду за рахунок розробки мобільного застосунку, який дозволяє створення та редагування нотаток і чеклістів, інтелектуального створення чеклістів на основі текстового та звукового вводу, а також хмарного бекапу та синхронізації.

Відповідно до поставленої мети в бакалаврській кваліфікаційній роботі потрібно вирішити такі **задачі**:

– провести аналіз предметної галузі та сучасних технологій управління особистими завданнями;

– розробити методи створення та редагування нотаток і чеклістів із підтримкою кастомізації інтерфейсу;

– розробити алгоритми обробки звукового вводу для інтелектуального створення чеклістів;

– розробити методи хмарної синхронізації та бекапу даних;

– реалізувати адаптивний графічний інтерфейс на основі архітектури MVVM;

– розробити модулі для створення чеклістів, обробки вводу та хмарного зберігання;

– провести тестування розробленого програмного забезпечення для забезпечення надійності та якості.

Об’єкт дослідження – процес організації та менеджменту особистих завдань за допомогою мобільного застосунку.

Предмет дослідження – методи та засоби створення нотаток і чеклістів, обробки вводу, хмарного бекапу даних у мобільному застосунку.

Методи дослідження. У процесі дослідження застосовувалися: аналіз сучасних технологій менеджменту завданнями, методи розробки мобільних застосунків, теорія обробки звукових даних, методи хмарного зберігання та синхронізації, архітектура MVVM для побудови модульної структури програми, а також методи тестування програмного забезпечення для перевірки теоретичних і практичних результатів.

Новизна отриманих результатів:

1. Запропоновано метод інтелектуального створення чеклістів на основі звукового вводу, який використовує обробку природної мови для автоматичного структурування завдань, що підвищує зручність і швидкість введення даних.

2. Запропоновано метод хмарної синхронізації та бекапу даних, який забезпечує безперебійний доступ до нотаток і чеклістів із різних пристроїв із мінімальними затримками.

3. Запропоновано підхід до кастомізації інтерфейсу, який інтегрує адаптивні теми, мови та шрифти, оптимізуючи користувацький досвід у мобільному застосунку, при цьому не перевантажуючи користувацький інтерфейс.

Практичне значення отриманих результатів полягає в тому, що розроблений мобільний застосунок може бути використаний для ефективного управління особистими завданнями в різних сферах, включаючи освіту, бізнес і

повсякденне життя. Реалізовані алгоритми та модулі забезпечують зручність, надійність і безпеку даних, що робить систему конкурентоспроможною серед аналогів.

Особистий внесок здобувача. Усі результати, подані в бакалаврській кваліфікаційній роботі, були отримані автором особисто. У науковій праці [4], опублікованій у співавторстві, автору належать такі результати: аналіз переваг та недоліків підходів розробки, визначення умов доцільності використання кожного з підходів.

Апробація матеріалів бакалаврської кваліфікаційної роботи. Результати бакалаврської кваліфікаційної роботи були представлені на конференції «Молодь в науці: дослідження, проблеми, перспективи (МН-2023)» та опубліковані у вигляді тез доповідей на цій конференції.

Публікації. Результати дослідження були опубліковані у матеріалах конференції - «Молодь в науці: дослідження, проблеми, перспективи (МН-2023)».

1 АНАЛІЗ ПІДХОДІВ ДО УПРАВЛІННЯ НОТАТКАМИ ТА ЧЕКЛІСТАМИ

1.1 Огляд сучасних технологій і концепцій управління нотатками та чеклістами

У сучасному світі ефективне управління особистими завданнями стало ключовою потребою для багатьох користувачів, адже швидкий ритм життя вимагає зручних інструментів для організації нотаток і чеклістів [1]. Технології, що застосовуються в цій сфері, постійно розвиваються, пропонуючи нові підходи до створення, редагування та синхронізації даних. Вони спрямовані на підвищення зручності та продуктивності, дозволяючи користувачам швидко фіксувати ідеї, планувати завдання та мати доступ до них із різних пристроїв. У цьому підрозділі розглядаються сучасні технології та концепції управління нотатками та чеклістами, які формують основу для розробки мобільних застосунків у цій галузі.

Однією з ключових технологій є використання текстового вводу для створення нотаток і чеклістів. Цей підхід є базовим для більшості застосунків, адже він дозволяє користувачам швидко записувати ідеї чи завдання у зрозумілій формі. Наприклад, текстові редактори вбудовані в такі програми, як Google Keep, дозволяють створювати нотатки з можливістю форматування тексту, додавання заголовків чи маркованих списків. Текстове введення часто доповнюється функціями автозбереження, що забезпечують захист даних від втрати у випадку збою. Проте текстовий ввід має обмеження: він вимагає від користувача ручного введення, що може бути незручним у ситуаціях, коли руки зайняті, наприклад, під час прогулянки чи водіння.

Для вирішення цієї проблеми активно застосовуються технології голосового вводу, які дозволяють користувачам створювати нотатки та чеклісти за допомогою мовлення [5]. Такі рішення базуються на системах розпізнавання мовлення, наприклад, Google Speech-to-Text або Apple Siri, які перетворюють голосові команди в текст. Голосовий ввід значно підвищує зручність, адже

користувач може просто продиктувати завдання, наприклад, "Додати до чекліста купити молоко", і система автоматично створить відповідний пункт. Проте точність розпізнавання залежить від якості мікрофона, рівня шуму та акценту користувача, що може призводити до помилок у транскрипції. Крім того, часто після голосового вводу потрібне ручне редагування, що зменшує ефективність процесу.

Ще однією важливою технологією є локальне зберігання даних, яке забезпечує доступ до нотаток і чеклістів у режимі офлайн. Для цього часто використовуються легкі бази даних, такі як SQLite або Room (у контексті Android) [6]. Локальне зберігання дозволяє швидко завантажувати дані без залежності від інтернет-з’єднання, що є критично важливим у ситуаціях із нестабільною мережею. Наприклад, Room, який використовується в багатьох Android-додатках, дозволяє створювати таблиці для нотаток із полями для ідентифікатора, заголовка та тексту, а також для чеклістів із полями для назви та статусу пунктів. Проте локальне зберігання має недоліки: якщо пристрій втрачено або пошкоджено, дані можуть бути втрачені без можливості відновлення, якщо не передбачено резервне копіювання.

Для забезпечення доступу до даних із різних пристроїв і захисту від їхньої втрати широко застосовуються хмарні сервіси. Такі технології, як Google Drive, iCloud або Dropbox, дозволяють синхронізувати нотатки та чеклісти між пристроями в реальному часі. Хмарна синхронізація базується на використанні API, наприклад, REST API, які дозволяють відправляти дані на сервер і отримувати їх за потреби. Це забезпечує безперебійний доступ до завдань із телефону, планшета чи комп’ютера, що підвищує продуктивність користувача. Однак хмарні рішення залежать від наявності інтернет-з’єднання, а затримки в синхронізації можуть створювати незручності, особливо при слабкому сигналі. Крім того, питання безпеки даних у хмарі залишається актуальним, адже витік інформації може призвести до порушення конфіденційності.

Сучасні концепції управління завданнями також включають інтуїтивний дизайн інтерфейсу, який відіграє ключову роль у підвищенні зручності. Одним із провідних стандартів є Material Design, розроблений Google, який

використовує принципи адаптивності, чіткої ієрархії елементів і природної анімації [7]. Наприклад, кнопки для створення нотаток чи чеклістів часто позначаються знайомими іконками (плюс для додавання), а екрани адаптуються до різних розмірів дисплеїв. Адаптивність інтерфейсу забезпечує комфортну роботу на пристроях із різною роздільною здатністю, що особливо важливо для мобільних додатків. Проте надмірна анімація чи складність інтерфейсу може заплутати користувача, особливо якщо він новачок.

Іншою концепцією є використання реактивного підходу до оновлення даних, що дозволяє інтерфейсу миттєво відображати зміни. Наприклад, у багатьох додатках застосовуються інструменти на кшталт LiveData або StateFlow (у контексті Android), які забезпечують автоматичне оновлення списків нотаток після їхнього створення чи редагування. Це підвищує зручність, адже користувач бачить актуальні дані без необхідності вручну оновлювати екран. Однак реактивний підхід може створювати додаткове навантаження на пристрій, особливо якщо список завдань великий, що впливає на швидкість роботи програми.

Технології автоматизації також відіграють важливу роль у сучасних системах управління завданнями. Зокрема, інтелектуальні функції, такі як автоматичне створення чеклістів на основі текстового чи звукового вводу, стають дедалі популярнішими. Наприклад, користувач може продиктувати "Запланувати похід у магазин: купити хліб, молоко, яйця", і система розпізнає це як команду для створення чекліста з трьома пунктами. Такі рішення базуються на обробці природної мови (NLP), яка дозволяє аналізувати текст чи голосові команди. Проте точність таких систем залежить від якості алгоритмів обробки, і помилки в розпізнаванні можуть призводити до неправильного формування завдань.

Ще одним трендом є інтеграція із зовнішніми сервісами, такими як календарі чи голосові помічники. Наприклад, додатки на зразок Google Keep дозволяють синхронізувати нотатки з Google Calendar, а голосові помічники, такі як Siri чи Google Assistant, можуть створювати завдання за командою. Це підвищує продуктивність, адже користувач може швидко перенести завдання в

календар або отримати нагадування. Однак інтеграція з зовнішніми сервісами ускладнює розробку, адже потребує підтримки різних API, а також може створювати залежність від сторонніх платформ, які можуть змінювати свої політики доступу.

Сучасні технології також враховують потребу в кастомізації інтерфейсу, що дозволяє користувачам адаптувати застосунок до своїх уподобань. Наприклад, вибір тем (світла, темна, кольорові варіанти), шрифтів чи мов інтерфейсу є стандартом у багатьох програмах. Такі рішення, як Jetpack Compose у Android, дозволяють розробникам створювати гнучкі інтерфейси, де зміни застосовуються в реальному часі. Кастомізація підвищує зручність, адже користувач може налаштувати застосунок під свої потреби, але надмірна кількість опцій може ускладнити інтерфейс, особливо для новачків.

Загалом, сучасні технології та концепції управління нотатками та чеклістами спрямовані на підвищення зручності та продуктивності користувачів. Текстове та звукове введення, локальне й хмарне зберігання, інтуїтивний дизайн і реактивні оновлення створюють основу для ефективної роботи із завданнями. Проте кожна з цих технологій має свої обмеження, такі як залежність від мережі, точність розпізнавання голосу чи складність інтеграції, що потребує збалансованого підходу при розробці мобільних застосунків.

1.2 Аналіз методів організації та синхронізації завдань

Організація особистих завдань і синхронізація даних між пристроями є основою сучасних систем управління нотатками та чеклістами. Ці методи дозволяють користувачам створювати, редагувати та мати доступ до своїх завдань у різних умовах, забезпечуючи зручність і безперервність роботи. У цьому підрозділі розглядаються основні методи організації нотаток і чеклістів, зокрема через текстове та звукове введення, а також методи синхронізації даних, включаючи локальне зберігання та хмарні рішення. Аналіз спрямований

на оцінку їхньої ефективності, переваг і недоліків із точки зору продуктивності та зручності.

Одним із базових методів організації завдань є ручне введення тексту, яке використовується в більшості додатків для створення нотаток і чеклістів. Цей метод передбачає, що користувач вводить текст за допомогою клавіатури, створюючи нотатки чи пункти чеклісту вручну. Наприклад, у програмах на зразок Microsoft To Do користувач може ввести завдання "Купити хліб", додавши його до списку покупок. Текстове введення забезпечує високу точність, адже користувач сам контролює зміст, а також дозволяє форматувати текст, наприклад, виділяти заголовки чи створювати марковані списки. Проте цей метод має недоліки: він вимагає часу на введення, що може бути незручним у ситуаціях, коли користувач поспішає, а також може бути складним для людей із обмеженими можливостями, які мають труднощі з набором тексту.

Для спрощення процесу створення завдань застосовуються методи автоматичного формування чеклістів на основі текстового вводу. Цей підхід передбачає аналіз введеного тексту для автоматичного створення структурованих списків. Наприклад, якщо користувач вводить "Купити хліб, молоко, яйця", система може розпізнати це як перелік і автоматично створити чекліст із трьома пунктами. Такі методи часто базуються на обробці природної мови (NLP), яка дозволяє виділяти ключові елементи тексту. Перевагою є економія часу, адже користувачу не потрібно вручну створювати кожен пункт. Однак точність залежить від якості алгоритмів: наприклад, якщо текст неоднозначний ("Зустрітися з друзями, купити подарунок завтра"), система може неправильно інтерпретувати завдання, що призведе до помилок у чеклісті.

Більш сучасним методом є використання звукового вводу для організації завдань, що дозволяє користувачам створювати нотатки та чеклісти за допомогою голосових команд. Цей метод базується на технологіях розпізнавання мовлення, таких як Google Speech-to-Text або Apple Speech Framework. Наприклад, користувач може сказати "Додати до чекліста купити молоко і хліб", і система перетворить це на текстовий список із двома

пунктами. Голосовий ввід значно підвищує зручність у ситуаціях, коли руки зайняті, наприклад, під час прогулянки чи готування. Крім того, він є доступним для людей із обмеженими можливостями, які не можуть вводити текст вручну. Проте точність розпізнавання залежить від багатьох факторів: якості мікрофона, рівня шуму, акценту користувача. У галасливому середовищі чи при нечіткій дикції система може неправильно інтерпретувати команди, що потребує додаткового редагування. Також обробка звукового вводу може бути ресурсоємною для пристрою, особливо якщо розпізнавання виконується локально.

Після створення завдань важливим є їхнє зберігання, і тут застосовуються методи локального зберігання даних. Локальні бази даних, такі як SQLite або Room (у контексті Android), дозволяють зберігати нотатки та чеклісти безпосередньо на пристрої. Наприклад, Room створює таблиці для нотаток із полями для ідентифікатора, заголовка та тексту, а для чеклістів – із полями для назви, списку пунктів і їхнього статусу. Локальне зберігання забезпечує швидкий доступ до даних без необхідності підключення до мережі, що є важливим у режимі офлайн. Це також зменшує затримки, адже дані завантажуються безпосередньо з пам’яті пристрою. Однак локальне зберігання має обмеження: якщо пристрій втрачено чи пошкоджено, дані можуть бути втрачені без можливості відновлення, якщо не передбачено резервне копіювання. Крім того, доступ до даних обмежений одним пристроєм, що ускладнює роботу в умовах, коли користувач хоче використовувати кілька гаджетів.

Для забезпечення доступу до завдань із різних пристроїв застосовуються методи хмарної синхронізації. Цей підхід базується на використанні хмарних сервісів, таких як Google Drive, iCloud або Firebase, які дозволяють зберігати дані на віддалених серверах і синхронізувати їх між пристроями. Наприклад, REST API використовується для відправлення нотаток на сервер і їхнього завантаження на іншому пристрої. Хмарна синхронізація дозволяє користувачу почати роботу на телефоні, а продовжити на планшеті чи комп’ютері, що підвищує продуктивність. Також вона забезпечує захист від втрати даних, адже

інформація зберігається на сервері. Проте хмарна синхронізація залежить від наявності інтернет-з’єднання, і затримки при слабкому сигналі можуть створювати незручності. Крім того, виникають питання безпеки: витік даних із хмари може призвести до порушення конфіденційності, якщо сервіс недостатньо захищений.

Іншим методом синхронізації є використання реактивних підходів, які дозволяють оновлювати дані в реальному часі. Наприклад, інструменти на зразок StateFlow або WebSocket забезпечують автоматичне оновлення списків завдань на всіх пристроях після їхньої зміни. Користувач додає нову нотатку на телефоні, і вона миттєво з’являється на планшеті, якщо обидва пристрої підключені до мережі. Реактивна синхронізація підвищує зручність, адже користувач завжди бачить актуальні дані. Однак цей метод вимагає стабільного з’єднання, і при великій кількості даних може створювати навантаження на мережу та пристрій. Якщо синхронізація переривається через проблеми з мережею, це може призвести до невідповідності даних між пристроями.

Ще одним аспектом організації завдань є методи кастомізації, які дозволяють адаптувати структуру даних до потреб користувача. Наприклад, користувач може налаштувати відображення чеклістів, обираючи формат (з позначками виконаності чи без), або додавати теги до нотаток для їхнього групування. Такі методи підвищують зручність, адже дозволяють користувачу організувати завдання відповідно до власних уподобань. Однак надмірна кастомізація може ускладнити інтерфейс, особливо для новачків, які не звикли до великої кількості опцій.

Методи інтеграції із зовнішніми сервісами також відіграють важливу роль у синхронізації та організації завдань. Наприклад, багато додатків дозволяють підключатися до календарів (Google Calendar) або голосових помічників (Siri, Google Assistant), щоб створювати завдання чи отримувати нагадування. Інтеграція з календарем дозволяє автоматично переносити дедлайни із чеклістів у розклад, що підвищує продуктивність. Проте інтеграція ускладнює розробку, адже потребує підтримки різних API, і може створювати

залежність від сторонніх сервісів, які можуть змінювати свої політики доступу чи припиняти роботу.

Загалом, методи організації та синхронізації завдань спрямовані на забезпечення зручності та безперервності роботи користувача. Текстове та звукове введення, локальне й хмарне зберігання, реактивна синхронізація та автоматизація дозволяють створювати ефективні системи управління завданнями. Проте кожен метод має свої обмеження: залежність від мережі, точність розпізнавання голосу, складність інтеграції чи ризик втрати даних. Ці аспекти необхідно враховувати при розробці мобільних додатків для управління завданнями.

1.3 Аналіз існуючих аналогів програм для управління завданнями

Сучасні програми для управління завданнями пропонують широкий спектр функцій для створення, організації та синхронізації нотаток і чеклістів, допомагаючи користувачам ефективно планувати свій час. Серед популярних рішень виділяються Google Keep, Todoist і Evernote, кожен із яких має свої сильні та слабкі сторони. Цей підрозділ присвячений аналізу цих аналогів, щоб визначити їхні переваги та недоліки, які можуть впливати на зручність і продуктивність користувачів.

Google Keep – це легкий і зручний інструмент, розроблений компанією Google, який фокусується на швидкому створенні нотаток і чеклістів [8] котрий зображено на рисунку 1.1. Застосунок має інтуїтивний інтерфейс, що дозволяє користувачам легко створювати текстові нотатки, марковані списки чи замітки з фотографіями. Однією з сильних сторін є підтримка голосового вводу: користувач може продиктувати завдання, наприклад, "Купити молоко", і програма створить нотатку. Хмарна синхронізація через Google Drive забезпечує доступ до даних із різних пристроїв, а офлайн-режим дозволяє працювати без мережі з подальшою . Google Keep також інтегрується з Google Calendar, що полегшує планування. Проте є суттєві недоліки: голосовий ввід не дозволяє автоматично створювати структуровані чеклісти – диктовка часто

потребує ручного редагування, щоб перетворити текст у список завдань. Це знижує зручність, адже користувачу доводиться додатково форматувати дані. Крім того, офлайн-функціонал обмежений: якщо зміни внесені без мережі, вони можуть не синхронізуватися коректно при слабкому з'єднанні, що створює ризик втрати даних. Також застосунок не пропонує гнучкої кастомізації інтерфейсу, наприклад, вибору шрифтів чи мов, що може бути незручним для користувачів із різними потребами.

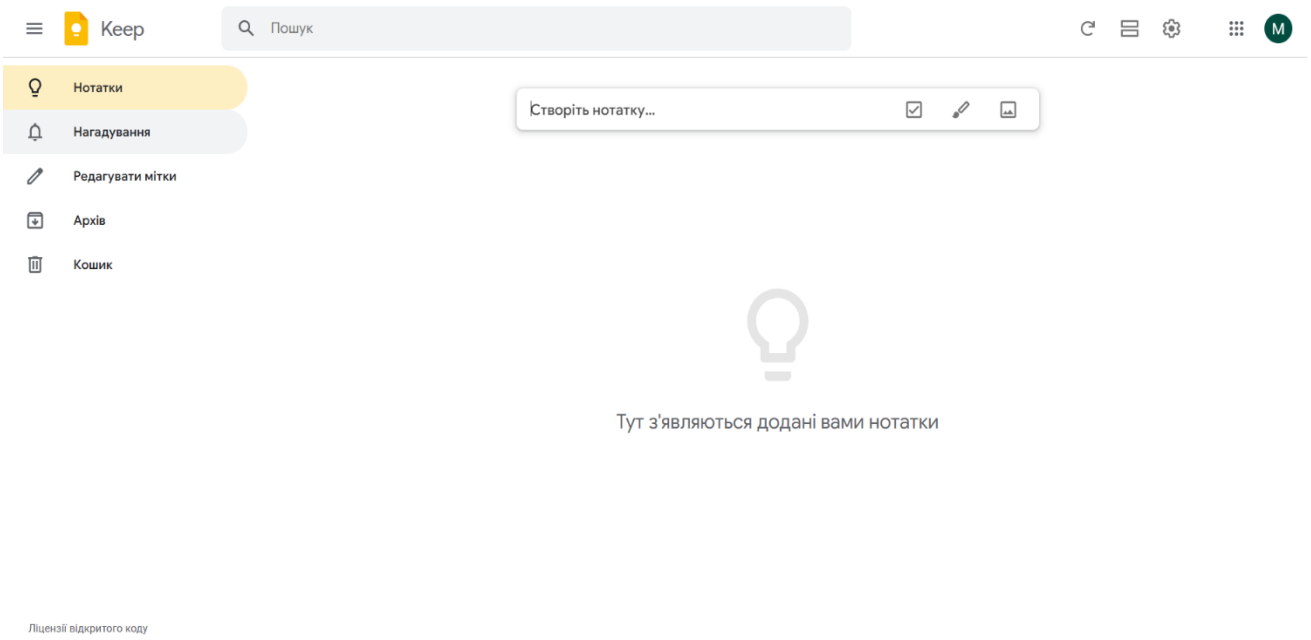


Рисунок 1.1 – Головний екран Google Keep

Todoist – це інструмент для управління завданнями, який орієнтований на створення чеклістів і проєктів із акцентом на планування [9]. Сильною стороною є гнучкість: користувачі можуть створювати детальні списки завдань із дедлайнами, пріоритетами та підзадачами, що ідеально підходить для складних проєктів. Хмарна синхронізація забезпечує доступ до даних із різних пристроїв, а інтеграція з Google Calendar і Dropbox дозволяє легко переносити завдання в розклад чи зберігати файли. Офлайн-доступ підтримується, що дозволяє працювати без мережі, із синхронізацією після підключення. Проте Todoist має недоліки, які впливають на базове використання: створення простих чеклістів із текстового вводу потребує ручного форматування, адже застосунок

не розпізнає автоматично списки із введеного тексту. Наприклад, введення "Купити хліб, молоко, яйця" не перетвориться на чекліст без додаткових дій користувача. Підтримка звукового вводу відсутня, що обмежує зручність для тих, хто воліє диктувати завдання. Крім того, для базового користувача інтерфейс може здаватися складним через велику кількість функцій, таких як пріоритети чи ярлики, які більше підходять для досвідчених користувачів, а не для швидкого створення простих списків.

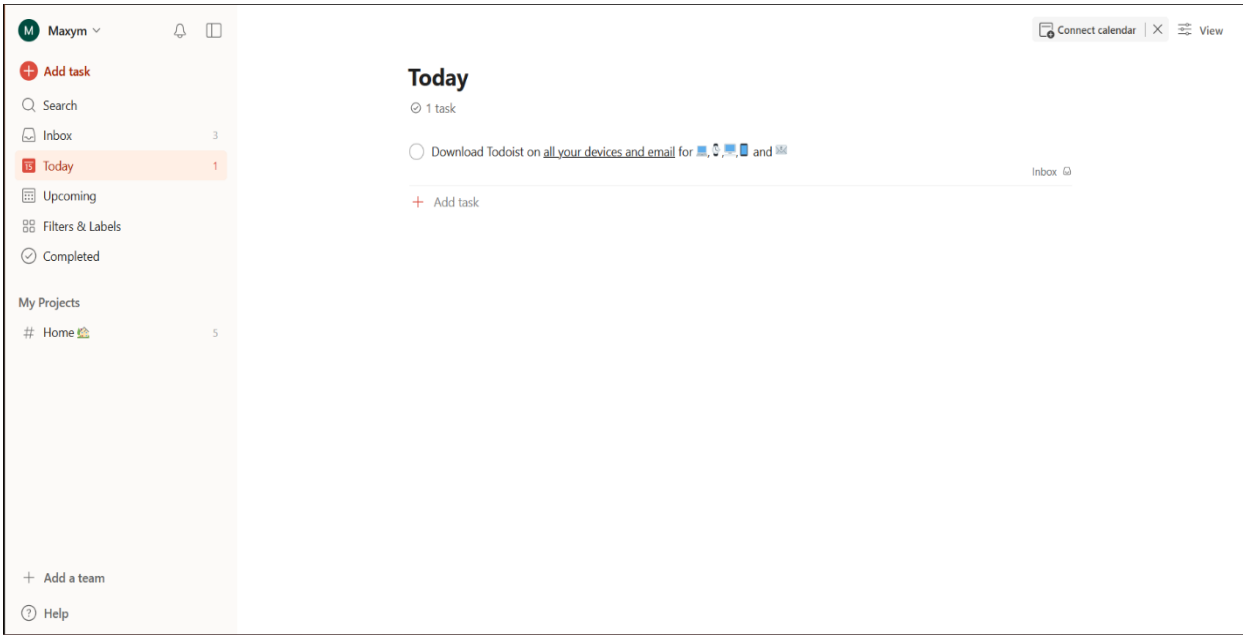


Рисунок 1.2 – Інтерфейс Todoist із плануванням завдань

Evernote – це універсальний інструмент для нотаток, який підтримує створення чеклістів і зберігання різноманітних даних, включаючи текст, зображення та вкладені файли [10]. Сильною стороною є багатозадачність: користувачі можуть створювати детальні нотатки з вкладеннями, наприклад, додавати фотографії чи документи, що робить застосунок корисним для роботи з великими обсягами інформації. Хмарний бекап через власний сервіс забезпечує доступ до даних із різних пристроїв, а функція пошуку дозволяє швидко знаходити нотатки за ключовими словами. Голосовий ввід підтримується, що дозволяє диктувати текст, наприклад, для створення швидких заміток. Проте Evernote має значні недоліки: інтерфейс перевантажений функціями, що ускладнює швидке створення простих чеклістів

чи нотаток для новачків. Наприклад, щоб додати чекліст, потрібно виконати кілька кроків, що знижує зручність у порівнянні з більш легкими додатками. Офлайн-доступ у безплатній версії обмежений: без підключення до мережі користувач не може повноцінно працювати, а синхронізація великих нотаток із вкладеннями часто затримується. Голосовий ввід не інтегрований для автоматичного створення чеклістів: диктовка тексту потребує додаткового форматування, що робить цю функцію менш практичною для швидкого використання.

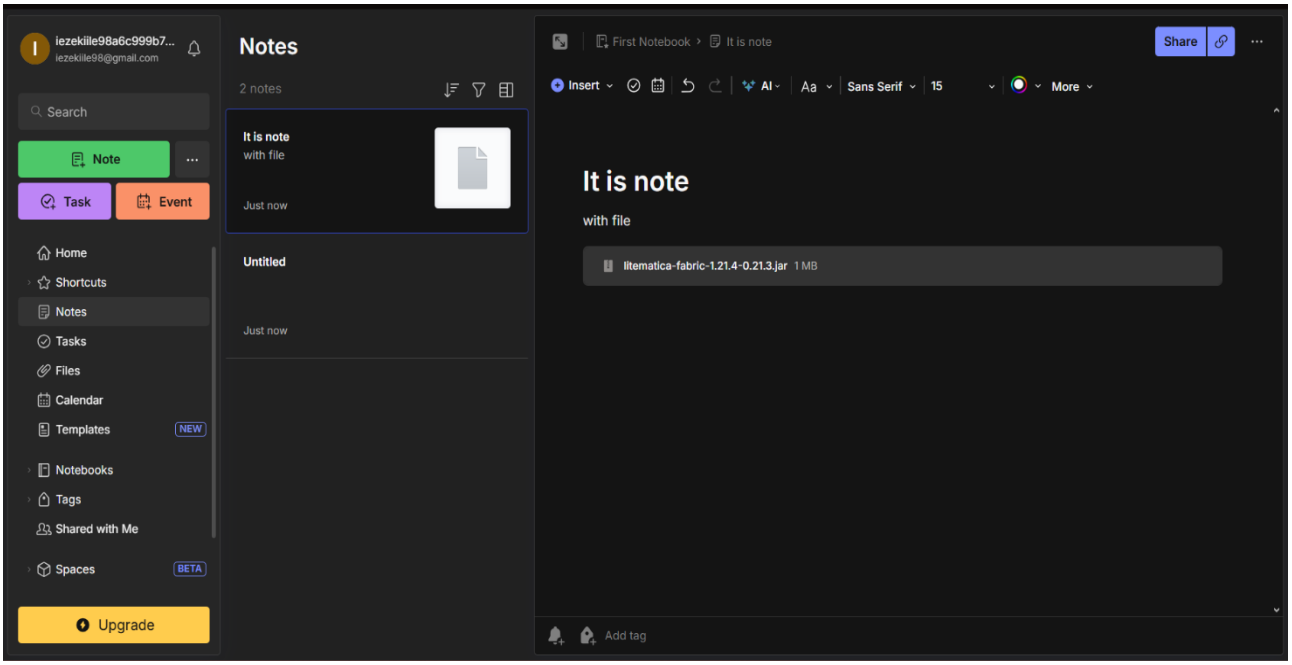


Рисунок 1.3 – Нотатка з вкладеннями в Evernote

Аналіз Google Keep, Todoist і Evernote показує, що кожен із цих продуктів має сильні сторони, які роблять їх популярними серед користувачів. Google Keep вирізняється простотою і швидким голосовим вводом, що дозволяє легко фіксувати ідеї. Todoist пропонує гнучкість для планування складних проєктів із дедлайнами та інтеграцією з іншими сервісами. Evernote забезпечує багатозадачність і можливість роботи з великими обсягами даних, включаючи вкладення. Проте недоліки цих аналогів створюють простір для вдосконалень. Google Keep не підтримує автоматичне створення чеклістів із голосового вводу, що потребує ручного редагування, а його офлайн-функціонал має обмеження в

синхронізації. Todoist ускладнює базове використання через відсутність автоматичного форматування тексту в чеклісти та звукового вводу, а також може бути незручним для новачків через надмірну функціональність. Evernote страждає від перевантаженого інтерфейсу, що уповільнює створення простих завдань, і слабкого офлайн-доступу в безплатній версії, що обмежує його використання без мережі. Ці недоліки можуть бути враховані при розробці нових рішень, які б запропонували більш зручні та інтуїтивні підходи до управління завданнями.

1.4 Визначення проблемних питань і постановка завдань проєкту

Аналіз сучасних підходів, методів і аналогів у сфері управління завданнями, проведений у попередніх підрозділах, виявив низку проблем, які впливають на зручність і продуктивність користувачів. Google Keep, Todoist і Evernote, хоча й пропонують широкий функціонал, мають суттєві недоліки: обмежена інтеграція звукового вводу для автоматичного створення чеклістів, складність інтерфейсу для базового використання, перевантаженість функціями та слабкий офлайн-доступ у безплатних версіях. Ці проблеми створюють потребу в розробці нового рішення, яке б забезпечило інтуїтивність, автоматизацію та стабільність роботи, підвищуючи зручність і продуктивність користувачів.

На основі виявлених недоліків пропонується розробка мобільного застосунку для організації та менеджменту особистих завдань. Основною метою проєкту є створення інструменту, який забезпечить створення та редагування нотаток і чеклістів, інтелектуальне створення чеклістів на основі текстового та звукового вводу, а також хмарний бекап і синхронізацію даних. Застосунок буде орієнтований на інтуїтивність: користувачі зможуть швидко створювати завдання через текстовий ввід, а також диктувати їх, після чого система автоматично сформує структурований чекліст без необхідності додаткового редагування. Наприклад, команда "Додати до списку покупок хліб, молоко, яйця" автоматично створить чекліст із трьома пунктами. Локальне

зберігання через локальну БД забезпечить стабільну роботу в офлайн-режимі, а хмарний бекап через BaaS рішення, що дозволить синхронізувати дані між пристроями з мінімальними налаштуваннями.

Інтерфейс буде розроблений за принципами Material Design, що забезпечить адаптивність і легкість використання.

Порівняльний аналіз із аналогами наведено в таблиці 1.1, яка ілюструє, як запропоноване рішення вирішує виявлені проблеми.

Таблиця 1.1 – Порівняння функціоналу аналогів із запропонованим рішенням

Функція	**Google Keep**	**Todoist**	**Evernote**	**Запропоноване рішення**
Створення нотаток і чеклістів	Так (ручне форматування)	Так (ручне форматування)	Так (складний процес)	Так (інтуїтивне створення)
Голосовий ввід	Так (з редагуванням)	Ні	Так (з редагуванням)	Так (автоматичне створення чеклістів)
Хмарний бекап і синхронізація	Так (затримки)	Так (складне налаштування)	Так (затримки)	Так (просте налаштування)
Офлайн-доступ	Так (обмежено)	Так (обмежено)	Обмежено (безплатна версія)	Так (стабільний через БД)
Інтелектуальні чеклісти	Ні	Ні	Ні	Так
Інтуїтивність інтерфейсу	Висока	Середня (складно для новачків)	Низька (перевантажений)	Висока (Material Design)

Запропоноване рішення перевершує аналоги за кількома аспектами: інтелектуальне створення чеклістів із текстового та звукового вводу усуває потребу в ручному редагуванні, що є проблемою Google Keep і Evernote; простий хмарний бекап вирішує складність налаштування в Todoist; стабільний офлайн-доступ через Room компенсує обмеження всіх трьох аналогів; а інтуїтивний інтерфейс на основі Material Design усуває перевантаженість Evernote.

Для реалізації мобільного застосунку для управління завданнями будуть використані засоби мобільної розробки і бібліотеки для створення графічних інтерфейсів та баз даних. Також необхідними є бібліотеки впровадження залежностей, а також інтеграція з BaaS сервісом або власним бекендом для хмарного бекапу.

Розробка застосунку передбачає кілька етапів. Спочатку буде створено модуль створення та редагування нотаток і чеклістів, який дозволить користувачам швидко додавати завдання через текстовий або голосовий ввід. Далі буде розроблено модуль інтелектуального створення чеклістів, який використовуватиме обробку текстового та звукового вводу для автоматичного формування списків. Наприклад, голосова команда "Запланувати похід у магазин: хліб, молоко" створить чекліст із відповідними пунктами. Потім буде реалізовано модуль хмарного бекапу та синхронізації, який використоуватиме Firebase для збереження даних у хмарі та їхнього завантаження на різних пристроях. Модуль кастомізації інтерфейсу дозволить користувачам налаштовувати теми, шрифти та мови, із миттєвим застосуванням змін через реактивне програмування. Локальне зберігання через локальна БД забезпечить офлайн-доступ, а архітектура MVVM із StateFlow гарантуватиме реактивне оновлення інтерфейсу.

На основі аналізу переваг і недоліків існуючих аналогів визначено наступні завдання, які необхідно виконати для успішної розробки власного мобільного застосунку:

- розробити модуль створення та редагування нотаток і чеклістів із інтуїтивним інтерфейсом;

- розробити модуль інтелектуального створення чеклістів на основі текстового та звукового вводу для автоматизації процесу;
- реалізувати модуль хмарного бекапу та синхронізації через BaaS, забезпечивши просте налаштування;
- розробити модуль кастомізації інтерфейсу, який дозволить користувачам налаштовувати теми, шрифти та мови;
- забезпечити стабільний офлайн-доступ через локальне зберігання даних із використанням локальної бази даних;
- реалізувати програмний продукт із використанням принципів Material Design для підвищення інтуїтивності;
- провести тестування програмного забезпечення для забезпечення зручності, стабільності та продуктивності.

Отже, розробка мобільного застосунку для управління завданнями є комплексним завданням, яке включає кілька етапів. Модулі будуть спроектовані, реалізовані та протестовані, щоб забезпечити користувачам зручний і продуктивний інструмент для організації особистих завдань.

1.5 Висновки

У першому розділі проведено аналіз сучасних підходів до управління особистими завданнями, розглянуто ключові технології та методи організації нотаток і чеклістів, а також проаналізовано популярні аналоги – Google Keep, Todoist і Evernote. Огляд технологій показав, що текстове та звукове введення, локальне зберігання через локальні бази даних і хмарна синхронізація через BaaS платформи є основою сучасних систем, але мають обмеження, такі як неточність розпізнавання голосу, залежність від мережі та складність інтеграції. Аналіз методів організації та синхронізації підкреслив важливість автоматизації, наприклад, через обробку природної мови для створення чеклістів, однак виявив проблеми з точністю та ресурсоємністю таких рішень. Аналоги продемонстрували сильні сторони, зокрема простоту Google Keep,

гнучкість Todoist і багатозадачність Evernote, але також мають недоліки, які впливають на зручність користувачів.

Виявлені проблеми стали основою для формування завдань проєкту. Google Keep не забезпечує автоматичного створення чеклістів із голосового вводу, Todoist ускладнює базове використання через відсутність автоматизації та надмірну функціональність, а Evernote страждає від перевантаженого інтерфейсу та слабкого офлайн-доступу в безплатній версії. На основі цього визначено потребу в розробці мобільного застосунку, який запропонує інтуїтивне створення та редагування нотаток і чеклістів, інтелектуальне формування чеклістів на основі текстового та звукового вводу, хмарний бекап через Firebase і стабільний офлайн-доступ. Ці завдання стануть основою для наступних етапів дослідження та розробки програмного продукту.

2 РОЗРОБКА МЕТОДІВ СИСТЕМИ МЕНЕДЖМЕНТУ ОСОБИСТИХ ЗАВДАНЬ

2.1 Розробка архітектурної моделі системи

Для кращого розуміння роботи модулів програмного застосунку з управління нотатками та чеклістами було розроблено функціональну модель системи у вигляді діаграми діяльності [11] (див. рисунок 2.1). Згідно з даною діаграмою, взаємодія користувача із застосунком починається з відкриття головного екрану, де доступні створені раніше нотатки та чеклісти.

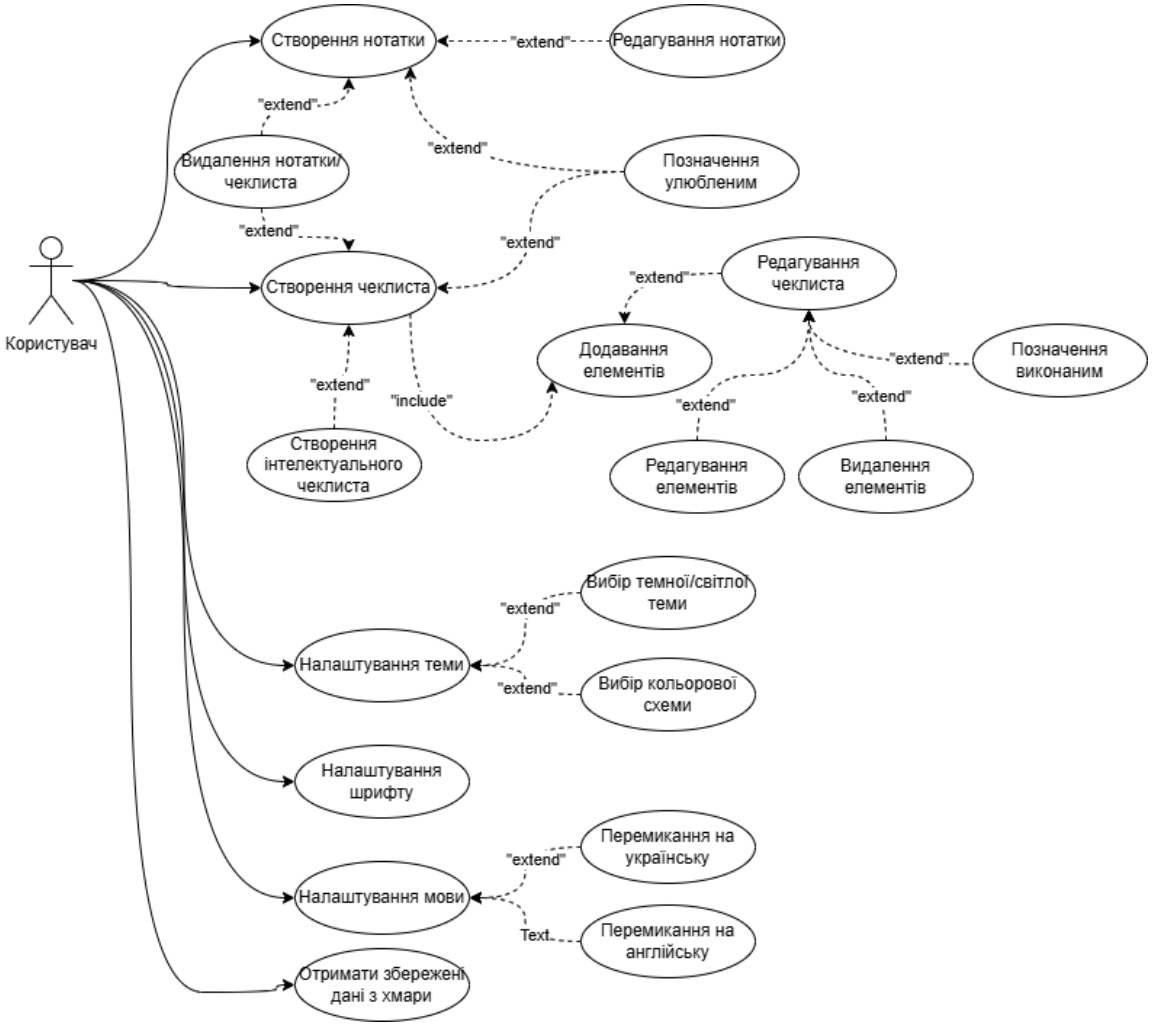


Рисунок 2.1 – Діаграма діяльності

Розроблена діаграма активностей представляє функціональну модель системи для створення та управління нотатками з елементами персоналізації інтерфейсу. Система забезпечує користувачам можливість ефективного створення, редагування та організації як текстових нотаток, так і інтерактивних списків завдань із гнучкими налаштуваннями відображення.

Діаграма активностей у даному проєкті слугує для наочного зображення усіх можливих варіантів використання системи та взаємозв'язків між ними. Вона демонструє як основні функціональні блоки, так і їхню внутрішню структуру через відношення включення (include) та розширення (extend).

Використання цієї діаграми забезпечує:

- структурований опис функціональності системи;
- чітке розмежування основних та похідних активностей;
- визначення логічних зв'язків між різними компонентами системи;
- комплексне розуміння поведінки системи з точки зору користувача.

Розробка діаграми діяльності була виконана у програмному середовищі draw.io.

З метою забезпечення гнучкості, масштабованості та зручності підтримки програмного застосунку було використано архітектурну модель на основі шаблону MVVM [12] (Model–View–ViewModel) (див. рисунок 2.2). В основі архітектури лежить розділення відповідальностей між компонентами: View відповідає за відображення інтерфейсу та взаємодію з користувачем, ViewModel опрацьовує події та містить бізнес-логіку, а Model реалізує доступ до даних.

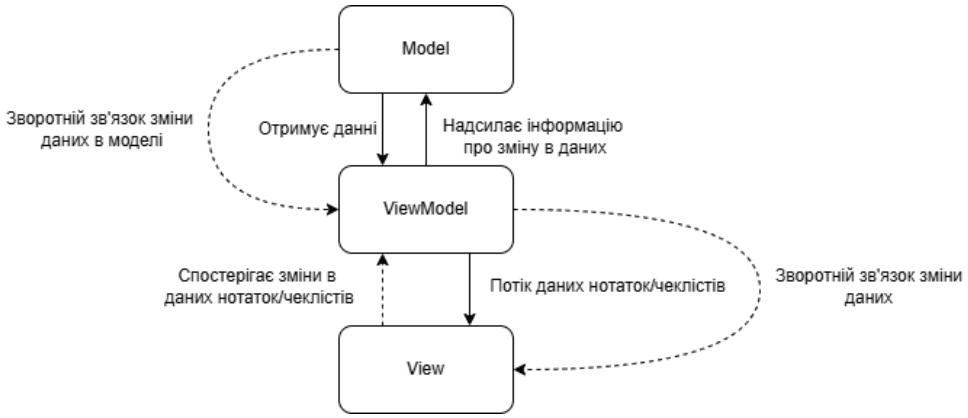


Рисунок 2.2 – Архітектурна модель системи

Координація між шарами реалізована за допомогою Kotlin coroutines, що дозволяє ефективно виконувати асинхронні операції без блокування основного потоку. Для спрощення інжекції залежностей застосовується бібліотека впровадження залежностей, що сприяє підтримці модульності та зменшенню зв'язності між компонентами.

На рисунку 2.3 зображено загальну архітектуру мобільного додатку та його взаємодію з зовнішніми сервісами: Firebase та API великої мовної моделі (LLM). Дана схема є діаграмою розгортання, яка демонструє фізичне розміщення компонентів та шляхи їх взаємодії між собою.

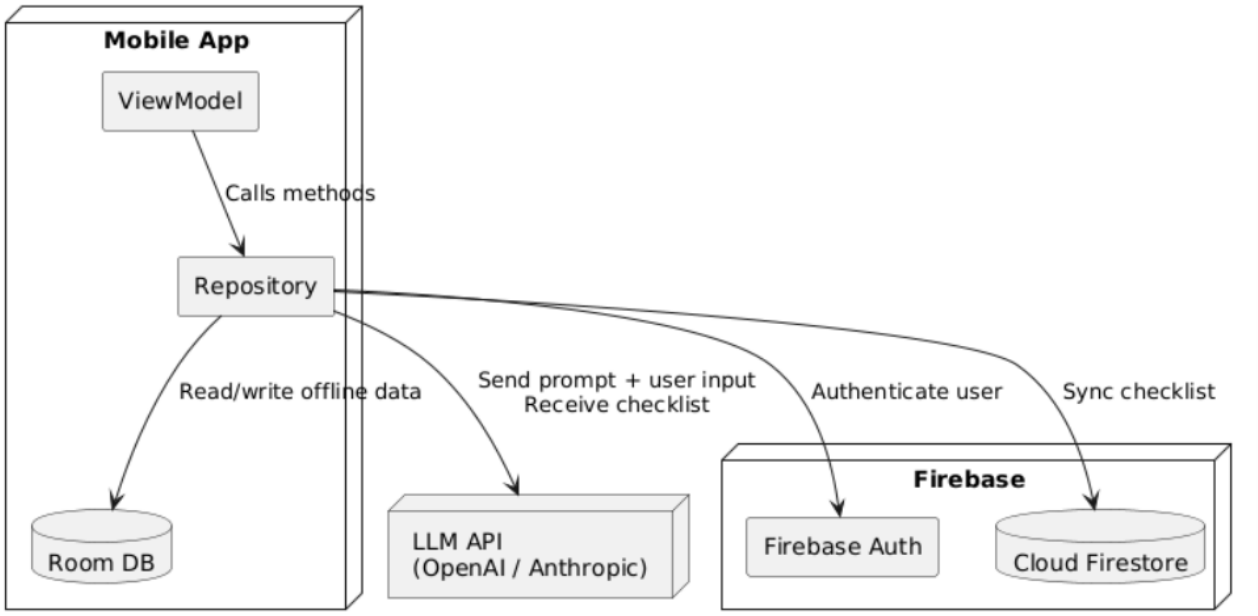


Рисунок 2.3 – Діаграма розгортання

У складі клієнтської частини додатку (Mobile App) виділено основні внутрішні модулі: ViewModel, Repository та DB. ViewModel відповідає за управління станом інтерфейсу користувача, Repository виступає як посередник між джерелами даних, а DB забезпечує зберігання інформації в офлайн-режимі на пристрої користувача. Взаємодія з зовнішніми системами реалізується саме через Repository [12-13].

LLM API, такий як OpenAI або Anthropic, використовується для генерації чеклистів на основі заданого prompt та введених користувачем даних. Запит

надсилається з Repository, а відповідь використовується для побудови динамічного змісту чеклиста.

З іншого боку, тут для прикладу – Firebase виконує роль хмарного бекенду. У схемі чітко показано використання двох сервісів: Firebase Auth для обов’язкової автентифікації користувача та Cloud Firestore для онлайн-зберігання чеклистів. Після проходження автентифікації користувачеві надається можливість зчитувати або зберігати дані у Firestore. Завдяки такому підходу забезпечується персоналізований доступ та збереження прогресу між сеансами.

Ця діаграма створена з метою візуалізації основних залежностей між компонентами застосунку та підтвердження доцільності використання вибраних сервісів. Вона демонструє розподіл відповідальностей між локальною обробкою, генеративним сервісом та хмарним сховищем, що дозволяє побудувати масштабовану, безпечну та зручну в користуванні систему. Така структурована подача взаємодій є необхідною для пояснення архітектурних рішень таких застосунків як вищеописаний.

Всі діаграми виконані у середовищі draw.io [14].

2.2 Розробка методів створення та управління особистих завдань

У рамках розробки методів створення та управління особистих завдань було створено алгоритм, який забезпечує ефективну роботу з особистими завданнями, включаючи створення нотаток і звичайних чеклістів (див. рисунок 2.4). Алгоритм охоплює ключові етапи обробки даних: від ініціалізації до завершення роботи з елементами, забезпечуючи зручну та безперебійну взаємодію користувача з системою.

Алгоритм розпочинається з ініціалізації модуля управління завданнями та налаштування частоти оновлення інтерфейсу. Цей параметр визначає, як часто система оновлює відображення завдань, нотаток або чеклістів на екрані, що дозволяє користувачу бачити актуальну інформацію без затримок. Частота оновлення адаптується залежно від активності користувача та доступних

ресурсів пристрою, уникаючи надмірного навантаження та забезпечуючи комфортну роботу.

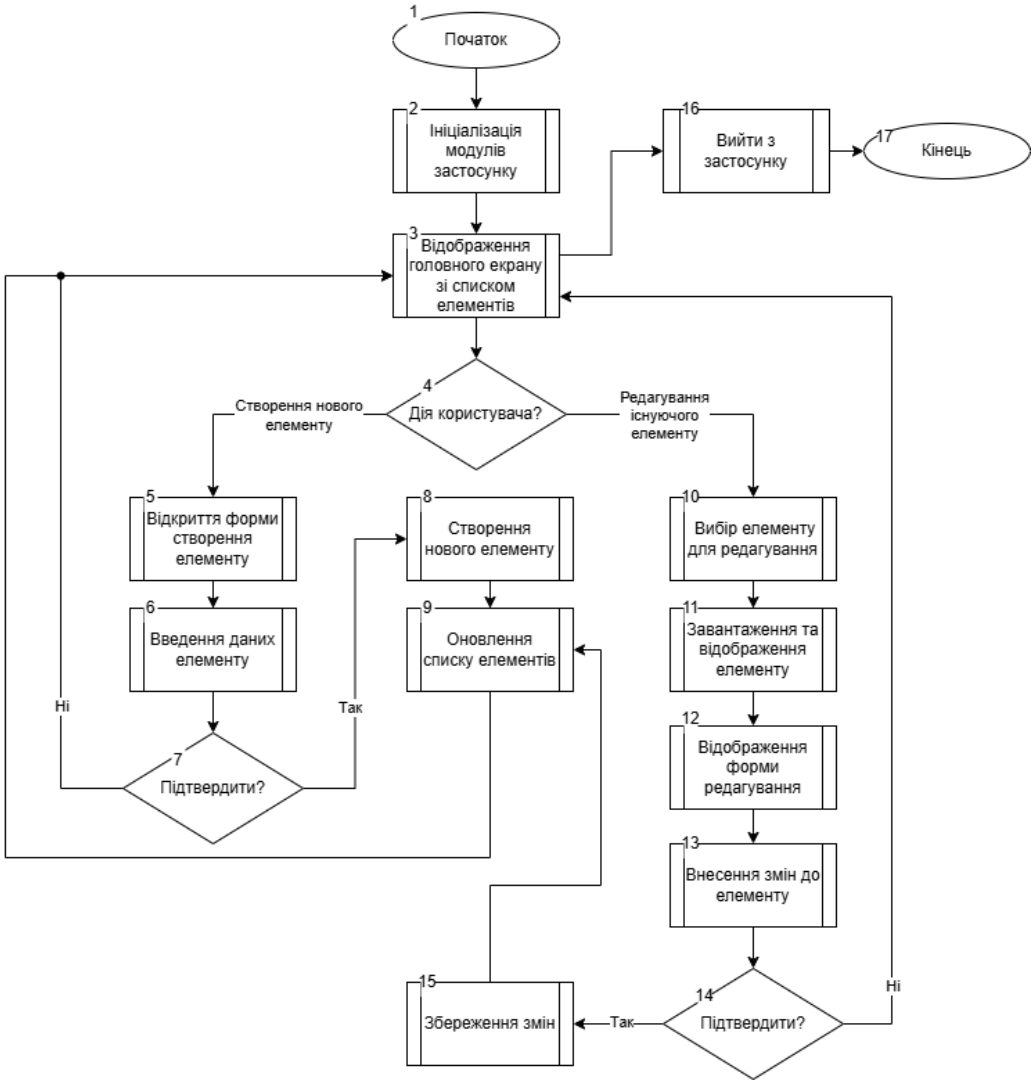


Рисунок 2.4 – Алгоритм взаємодії з елементами в застосунку

Далі користувач обирає елемент зі списку завдань – нотатку або звичайний чекліст. Алгоритм перевіряє наявність обраного елемента. Якщо елемент відсутній, система повертається до регулярного оновлення головного списку, зберігаючи простоту та зручність інтерфейсу. У разі наявності елемента користувач може або створити новий елемент, або відредагувати існуючий. При створенні нового елемента – наприклад, нотатки чи чеклісту – користувач вводить відповідні дані: текст для нотатки або пункти для чеклісту. Після

введення даних елемент зберігається в локальній базі даних, а список оновлюється.

У режимі редагування користувач може змінювати вміст нотатки чи пункти чеклісту, додавати нові або видаляти непотрібні. Зміни зберігаються в локальній базі даних, що забезпечує доступ до даних у режимі офлайн. Алгоритм дозволяє переривати роботу над елементом і повертатися до неї пізніше, зберігаючи всі зміни. Така гнучкість робить систему зручною для різних сценаріїв використання. Використання локальної бази даних гарантує збереження даних навіть без підключення до мережі, а підтримка кастомізації інтерфейсу дозволяє користувачам налаштувати систему під свої потреби.

Окремим напрямом розробки є інтеграція інтелектуальних чеклістів, які створюються на основі сирого тексту, такого як звуковий запис чи текстовий ввід користувача, без необхідності вручну вводити окремі пункти чи навіть назву чеклиста. Система аналізує введений текст через зовнішнє API мовної моделі, перетворюючи його на структурований і чіткий чекліст. Наприклад, якщо користувач вводить "потрібно підготуватися до подорожі, перевірити документи, купити квитки", модель автоматично формує список: "1. Перевірити документи", "2. Купити квитки", "3. Зібрати валізу", та сформує назву по типу “подорож” додаючи логічні пункти на основі контексту.

Процес роботи з інтелектуальними чеклістами простий: користувач вводить сирий текст із описом завдання, наприклад, під час розмови чи в текстовому полі. Система надсилає цей текст до API мовної моделі, яка обробляє його й повертає готовий чекліст із упорядкованими пунктами. Користувачу не потрібно клацати чи вручну додавати елементи – сформований список одразу відображається для перегляду або редагування, якщо потрібно. Після цього чекліст зберігається в локальній базі даних, забезпечуючи доступ у режимі офлайн.

Також в якості потенціального поліпшення можна розглянути адаптацію до потреб користувача, враховуючи контекст введеного тексту. Якщо користувач додаватиме деталі, наприклад, "подорож із дітьми", модель може включити пункти, такі як "упакувати іграшки" чи "перевірити дитячі

документи". Такий підхід дозволяє певною мірою автоматизувати планування, економлячи час і усуваючи необхідність ручної роботи з пунктами, що зробить систему особливо зручною для швидкого створення завдань але вимагатиме додатково контекст про особистість користувача. Подібний функціонал сильно ускладнить систему на даному етапі розробки та може виявитися надлишковим.

2.3 Розробка методів хмарного бекапу, синхронізації та кастомізації

Для забезпечення надійного збереження даних користувача, безперебійної роботи на різних пристроях і комфортної взаємодії з додатком було розроблено методи хмарного бекапу, синхронізації та кастомізації інтерфейсу. Ці методи дозволяють користувачу працювати з особистими завданнями, нотатками та чеклістами, не турбуючись про втрату даних, а також налаштовувати застосунок відповідно до своїх уподобань.

Метод хмарного бекапу розроблено для автоматичного збереження даних користувача, щоб забезпечити їхню безпеку та доступність у разі втрати чи заміни пристрою. Система працює за принципом щоденного оновлення хмарного сховища, яке відбувається у визначений час, коли користувач, найімовірніше, не використовує пристрій, – у період сну, наприклад, о 3 годині ночі. Такий підхід дозволяє уникнути перевантаження мережі в активні години та мінімізувати вплив на продуктивність пристрою під час роботи користувача.

Щоденний бекап охоплює всі дані, які залишилися невиконані з попереднього дня, включаючи незавершені завдання, нотатки та чеклісти. Водночас бекап не проводиться вранці, щоб уникнути додавання даних, які з більшою ймовірністю будуть виконані до наступного бекапу. Це забезпечує зменшення навантаження на хмарне сховище як в плані об’єму даних так і кількості звернень від користувачів, що дозволить підтримувати значно більший обсяг користувачів на початкових етапах експлуатації. Такий підхід дозволяє системі зберігати актуальний стан незавершених елементів, не перевантажуючи хмарне сховище зайвими оновленнями.

Користувач також має можливість у будь-який момент запросити свій бекап із хмарного сховища на пристрій. Це може бути як останній доступний бекап, так і попередні версії, що особливо корисно в разі випадкового видалення даних. Наприклад, якщо користувач випадково видалив важливу нотатку чи незакінчений чеклист, він може відновити їх з бекапу за попередній день. Для цього система надає простий інтерфейс, де користувач обирає потрібну дату бекапу, після чого дані завантажуються на пристрій і синхронізуються з локальною базою даних. Така функціональність підвищує надійність системи та забезпечує гнучкість у роботі з даними, діаграма активності для бекапу зображена на рисунку 2.5.

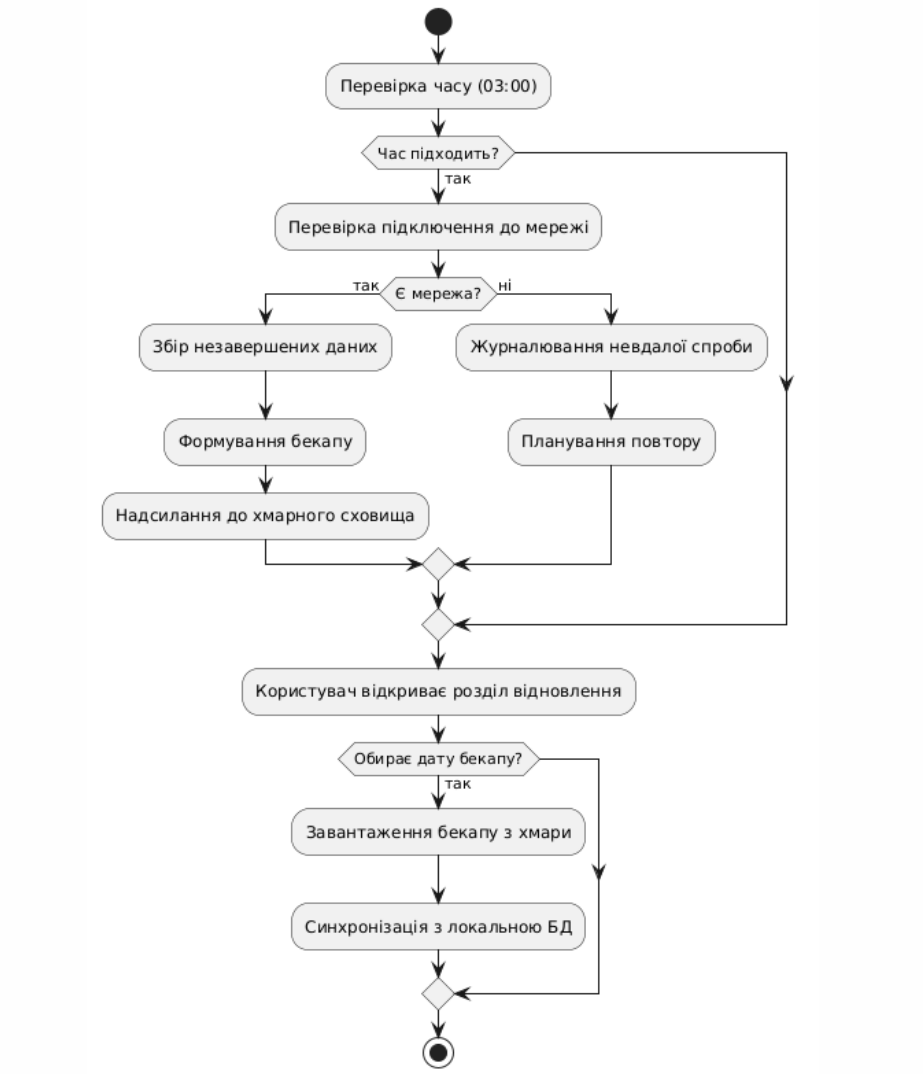


Рисунок 2.5 – Діаграма активності для хмарного бекапу

Синхронізація між пристроями та хмарним сховищем є ключовим елементом для забезпечення безперервної роботи користувача, незалежно від того, чи є доступ до мережі. Система розроблена з урахуванням офлайн-режиму, де локальна база даних виступає основним джерелом правди. Це означає, що всі зміни, внесені користувачем у режимі офлайн, зберігаються локально, а при підключенні до мережі синхронізуються з хмарним сховищем.

Під час синхронізації пріоритет віддається локальній базі даних, оскільки через природу щоденного бекапу (який відбувається вночі) локальні дані зазвичай є новішими. Якщо в хмарному сховищі є дані, яких немає на пристрої – наприклад, після використання іншого пристрою, – вони завантажуються та додаються до локальної бази даних. Такий підхід забезпечує, що користувач завжди має доступ до найактуальнішого набору даних, незалежно від того, на якому пристрої він працює.

На рисунку 2.6 зображено діаграму станів нотатки для процесу синхронізації, який детальніше описаний нижче.

Рисунок 2.6 – Діаграма станів для процесу синхронізації нотатки

Для вирішення можливих конфліктів, які виникають, коли одні й ті самі дані змінюються на різних пристроях, або хмарна копія відрізняється від локальної, система пропонує користувачу ручне управління. У разі виявлення конфлікту – наприклад, коли завдання було відредаговано в різний спосіб на двох пристроях, – користувачу надається простий інтерфейс із можливістю обрати, які зміни зберегти. Інтерфейс представлений у вигляді списку з опціями "зберегти стару версію" або "зберегти нову версію", де кожна опція супроводжується коротким описом змін, таких як дата редагування чи основний вміст. Це дозволяє користувачу швидко прийняти рішення без необхідності глибоко аналізувати технічні деталі, роблячи процес інтуїтивним і зручним.

Кастомізація інтерфейсу розроблена для забезпечення максимального комфорту користувача під час роботи з додатком. Система підтримує налаштування стилів шрифтів, кольорових схем, тем оформлення та мови додатку, що дозволяє адаптувати його до індивідуальних уподобань. Користувач може обирати між різними стилями шрифтів, які впливають на читабельність тексту в завданнях, нотатках і чеклістах. Наприклад, доступні шрифти з різною товщиною стилями, щоб користувач міг обрати той, який найкраще підходить для його зору чи естетичних уподобань.

Кольорові схеми дозволяють змінювати загальний вигляд додатку, включаючи фон, кольори тексту та елементів інтерфейсу. Користувач може обрати одну з кількох попередньо налаштованих схем – наприклад, спокійні пастельні персикові тони або контрастні комбінації для кращої видимості. Також підтримується вибір між світлою та темною темами, що особливо корисно для роботи в різний час доби. Світла тема підходить для використання при яскравому освітленні, тоді як темна тема зменшує навантаження на очі в умовах недостатнього світла, наприклад, уночі.

Мова додатку є ще одним важливим елементом кастомізації. Система підтримує дві мови – українську та англійську, що обумовлено цільовою аудиторією та практичними потребами. Українська мова є основною, оскільки застосунок буде розроблено передусім для україномовних користувачів, які

становлять основну частину цільової аудиторії. Англійська мова буде додана для забезпечення доступності для іноземних користувачів, а також для тих, хто використовує застосунок у професійному контексті, де англійська є мовою міжнародного спілкування.

Отже, розроблені методи хмарного бекапу, синхронізації та кастомізації інтерфейсу забезпечують надійне збереження даних, безперебійну роботу на різних пристроях і комфортну взаємодію з додатком. Щоденні бекапи, які враховують незавершені завдання, дозволяють користувачу не турбуватися про втрату даних, а можливість відновлення попередніх версій додає гнучкості. Синхронізація з пріоритетом локальної бази даних забезпечує актуальність даних і зручність роботи в офлайн-режимі. Кастомізація інтерфейсу, включаючи налаштування шрифтів, кольорових схем, тем і мов, робить застосунок адаптивним до потреб користувача, підвищуючи його зручність і привабливість порівняно з сухими корпоративними аналогами.

2.4 Висновки

У рамках другого розділу розроблено архітектурну модель системи на основі шаблону MVVM, що забезпечує модульність і масштабованість додатку. Створено діаграму діяльності для наочного відображення взаємодії користувача з нотатками та чеклістами, а також діаграму розгортання, яка демонструє інтеграцію з хмарними сервісами та API мовної моделі для генерації інтелектуальних чеклістів. Реалізовано алгоритм управління завданнями, який підтримує створення нотаток, звичайних та інтелектуальних чеклістів на основі сирого тексту, автоматизуючи планування. Запропоновано методи хмарного бекапу з щоденним нічним оновленням, синхронізації з пріоритетом локальної бази даних і ручним вирішенням конфліктів, а також кастомізації інтерфейсу з налаштуванням шрифтів, кольорових схем, тем і мов (української та англійської). Ці рішення забезпечують надійність, зручність і адаптивність системи до потреб користувача, створюючи основу для успішної реалізації проєкту.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ МЕНЕДЖМЕНТУ ОСОБИСТИХ ЗАВДАНЬ

3.1 Обґрунтування вибору засобів розробки програмного забезпечення

Розробка програмного забезпечення для управління завданнями потребує ретельного підбору інструментів, які забезпечать високу продуктивність, масштабованість і зручність підтримки. У цьому підрозділі обґрунтовано вибір платформи, підходу до розробки, середовища, мови програмування, архітектури, бібліотек і компонентів, із детальним поясненням їхньої відповідності вимогам проєкту.

Вибір платформи для створення додатку став першим і ключовим кроком, адже від цього залежить аудиторія та можливості реалізації. Android вирізняється своєю популярністю, адже за даними StatCounter на травень 2025 року він займає близько 70% світового ринку мобільних операційних систем, значно випереджаючи iOS із його 29%. Це означає, що застосунок, розроблений для Android, зможе охопити значно ширше коло користувачів, що є критично важливим для проєкту, орієнтованого на масове використання. Окрім чисельної переваги, відкрита екосистема Android дозволяє легко інтегрувати застосунок із різноманітними сервісами, такими як хмарні сховища чи API, що є необхідним для функціональності синхронізації та генерації інтелектуальних чеклістів. Наприклад, Android забезпечує гнучкість у налаштуванні фонових процесів, таких як щоденний нічний бекап, що було б складніше реалізувати на iOS через суворіші обмеження. До того ж, Android підтримує широкий спектр пристроїв – від бюджетних до преміальних, що дозволяє зробити застосунок доступним для користувачів із різними фінансовими можливостями, підвищуючи його привабливість і практичність.

Наступним етапом стало визначення підходу до розробки, де розглядалися кросплатформні фреймворки, такі як Flutter чи React Native, які пропонують єдину кодову базу для Android і iOS, скорочуючи час розробки.

Однак перевагу віддано нативній розробці для Android через її незаперечні переваги в продуктивності та інтеграції з платформою. Нативні додатки мають прямий доступ до апаратних ресурсів пристрою, таких як пам’ять, процесор чи мережа, що забезпечує швидшу роботу, особливо для ресурсоємних операцій, таких як синхронізація даних у офлайн-режимі [4]. Наприклад, під час роботи з хмарним сховищем нативний підхід дозволяє оптимізувати мережеві запити, зменшуючи затримки та споживання батареї, що було б складніше зробити в кросплатформних рішеннях через додатковий рівень абстракції. До того ж, нативна розробка дає змогу повноцінно використовувати можливості екосистеми Android – віджетів, сповіщень, фонових служб і навіть специфічних API, таких як служби доступності, які можуть бути корисними для кастомізації інтерфейсу. Кросплатформні рішення, хоч і економлять час на початковому етапі, часто поступаються в швидкості й глибині інтеграції, що могло б негативно вплинути на користувацький досвід, особливо для такого функціонально насиченого додатку, як цей.

Для створення додатку обрано Android Studio як основне середовище розробки завдяки його спеціалізованим інструментам, які значно перевершують можливості інших IDE, таких як IntelliJ IDEA чи Visual Studio Code із плагінами для Android-розробки. Android Studio надає вбудований емулятор, профайлер продуктивності, підтримку Gradle для управління залежностями та інтеграцію з Android SDK, що дозволяє розробникам працювати швидше й ефективніше. Наприклад, емулятор дає змогу тестувати застосунок на різних пристроях із різними версіями Android без фізичного доступу до них, що економить час і ресурси, особливо коли потрібно перевірити, як застосунок виглядатиме на старих чи бюджетних пристроях. Профайлер продуктивності допомагає виявляти вузькі місця, такі як повільні запити до бази даних чи надмірне споживання пам’яті під час синхронізації, що дозволяє оптимізувати застосунок ще на етапі розробки. IntelliJ IDEA чи Visual Studio Code, хоч і можуть бути адаптовані для Android-розробки, не мають такого рівня інтеграції, що ускладнює налагодження, тестування й автоматизацію. Android Studio також підтримує вбудовані інструменти для роботи з Git, що полегшує командну

розробку та контроль версій, а інтеграція з Gradle спрощує підключення бібліотек, таких як Firebase чи Room, роблячи його ідеальним вибором для цього проєкту.

Вибір мови програмування став важливим етапом, де перевагу віддано Kotlin замість традиційної Java, яка довгий час була основною для Android-розробки. Kotlin вирізняється лаконічністю коду, безпекою типів і підтримкою сучасних підходів, таких як функціональне програмування, що робить його більш зручним для розробки складних додатків [15]. Механізм null-safety у Kotlin усуває ризики помилок, пов’язаних із null-значеннями, які в Java потребують ручної перевірки, що може призводити до аварійного завершення роботи додатку, наприклад, під час спроби обробити неіснуючу нотатку. Асинхронне програмування через Coroutines у Kotlin значно простіше, ніж у Java з її RxJava, що дозволяє легко обробляти мережеві запити, наприклад, для інтелектуальних чеклістів, без надмірного ускладнення коду. До того ж, Kotlin підтримує сучасні конструкції, такі як data classes, які ідеально підходять для моделювання об’єктів, наприклад, нотаток чи чеклістів, зменшуючи обсяг шаблонного коду порівняно з Java. Ця мова дозволяє створювати чистий і зрозумілий код, що полегшує його підтримку та масштабування в майбутньому, коли застосунок може отримати нові функції, такі як інтеграція з іншими сервісами чи підтримка нових типів завдань.

Для структури додатку обрано архітектурний шаблон MVVM, який виявився більш відповідним порівняно з іншими підходами, такими як MVC чи MVP. MVVM ефективно інтегрується з Android Jetpack, зокрема з ViewModel, що забезпечує надійне управління станом інтерфейсу, розділяючи бізнес-логіку від відображення [12]. На відміну від MVC, де тісний зв’язок контролера з View ускладнює модульність і тестування, MVVM дозволяє ізолювати логіку у ViewModel, що спрощує перевірку функціональності, наприклад, обробки чеклістів, без прив’язки до інтерфейсу. MVP, хоча й подібний до MVVM, вимагає більше ручного коду для управління станом, тоді як MVVM використовує LiveData для автоматичного оновлення інтерфейсу, що зменшує кількість помилок і прискорює розробку. У проєкті MVVM ідеально підходить

для управління нотатками, чеклістами та кастомізацією інтерфейсу: наприклад, перемикання між світлою та темною темами можна реалізувати через ViewModel, яка оновлює UI без прямого втручання в код відображення, забезпечуючи гнучкість і зручність.

Для обробки асинхронних операцій обрано Kotlin Coroutines, які перевершують RxJava за простотою та інтеграцією з Kotlin. Coroutines дозволяють писати асинхронний код у послідовному стилі за допомогою підвішених функцій, що значно спрощує роботу з мережевими запитами до OpenAI API чи Firebase порівняно з RxJava, де оператори, як-от map чи flatMap, ускладнюють код і його читабельність. Coroutines підтримують структуровану конкурентність, що зменшує ризик витоків пам’яті, які можуть виникати при неправильному управлінні потоками. У проєкті це критично, адже асинхронність потрібна для створення інтелектуальних чеклістів і синхронізації даних: наприклад, запит до OpenAI API для генерації пунктів чекліста виконується у фоновому режимі, а Coroutines забезпечують, що інтерфейс залишиться чутливим. До того ж, Coroutines дозволяють легко скасовувати операції, якщо користувач закриває екран, що економить ресурси пристрою та покращує досвід користувача.

Для управління залежностями обрано Hilt, який базується на Dagger2, але спрощує його використання. Hilt надає готові анотації та інтеграцію з Android-компонентами, такими як ViewModel і Activity, автоматично інжектуючи залежності, тоді як Dagger2 вимагає ручного налаштування, що збільшує обсяг коду й ризик помилок. Hilt зменшує шаблонний код, дозволяючи розробникам зосередитися на бізнес-логіці, а не на конфігурації залежностей [16]. У проєкті Hilt полегшує інжекцію репозиторіїв для роботи з локальними та хмарними даними: наприклад, репозиторій для доступу до Firebase чи Room автоматично інжектується у ViewModel, що спрощує код і сприяє модульності. Це також полегшує тестування, адже залежності можна легко замінити на мок-об’єкти, що важливо для перевірки функціональності синхронізації чи обробки даних.

Для локального зберігання даних обрано Room DB, який базується на SQLite, але пропонує вищий рівень абстракції. Room DB автоматично

відображає об’єкти Kotlin на таблиці бази даних і перевіряє запити під час компіляції, усуваючи помилки, пов’язані з ручним написанням SQL-запитів у SQLite, що може призводити до складно виявлюваних багів. Його підтримка міграцій дозволяє легко оновлювати структуру даних у майбутньому, наприклад, при додаванні нових типів завдань чи налаштувань користувача. Інтеграція з LiveData забезпечує автоматичне оновлення інтерфейсу при зміні даних, наприклад, коли користувач додає нову нотатку, що зменшує обсяг коду для оновлення UI. У проєкті Room DB ефективно зберігає нотатки, чеклісти та підтримує пріоритет локальних даних під час синхронізації, що гарантує доступність даних у офлайн-режимі та їхню актуальність при підключенні до мережі.

Для хмарних операцій обрано Firebase, зокрема Firebase Auth і Cloud Firestore. Firebase Auth забезпечує безпечну автентифікацію через електронну пошту чи Google, що необхідно для персоналізованого доступу до даних, дозволяючи користувачу безпечно входити в систему та зберігати свої нотатки й чеклісти [17-18]. Cloud Firestore дозволяє синхронізувати дані між пристроями, підтримуючи офлайн-режим і автоматичну синхронізацію, що ідеально відповідає вимогам проєкту. Наприклад, якщо користувач змінює чекліст на одному пристрої без підключення, Firestore синхронізує ці зміни на інших пристроях, щойно з’явиться мережа. Firebase також забезпечує масштабованість і інструменти моніторингу, що дає змогу аналізувати продуктивність додатку та масштабувати його в майбутньому, коли кількість користувачів зросте. У проєкті Firebase гарантує надійну синхронізацію та безпеку даних, що є ключовим для безперебійної роботи.

Для створення інтелектуальних чеклістів обрано OpenAI API, яке забезпечує високу якість обробки природної мови. Воно здатне генерувати контекстно релевантні пункти, наприклад, додаючи "упакувати іграшки" для запиту "подорож із дітьми", що робить процес створення чеклістів інтуїтивним і швидким. Автоматизація через OpenAI API економить час користувача, адже йому не потрібно вручну вводити кожен пункт, а інтеграція з Kotlin Coroutines гарантує асинхронність, зберігаючи чутливість інтерфейсу під час обробки

запитів. У проєкті це дозволяє швидко формувати структуровані чеклісти, підвищуючи зручність і додаючи інноваційну цінність додатку.

Для інтерфейсу обрано Jetpack Compose із Material Design 3 (MD3). Compose, будучи декларативним фреймворком на Kotlin, зменшує шаблонний код, дозволяючи створювати UI безпосередньо в коді та легко змінювати теми через Theme-компоненти, наприклад, для перемикання між світлою та темною темами. MD3 надає сучасний дизайн із адаптивними елементами, такими як кнопки чи картки завдань, що відповідає стандартам Android і забезпечує приємний візуальний досвід [7]. Реактивність Compose спрощує оновлення інтерфейсу, наприклад, списку завдань, коли користувач додає чи видаляє пункти [19]. У проєкті це створює інтуїтивний і кастомізований досвід, дозволяючи користувачу налаштовувати шрифти, кольори та мову додатку (українську чи англійську) з мінімальними зусиллями.

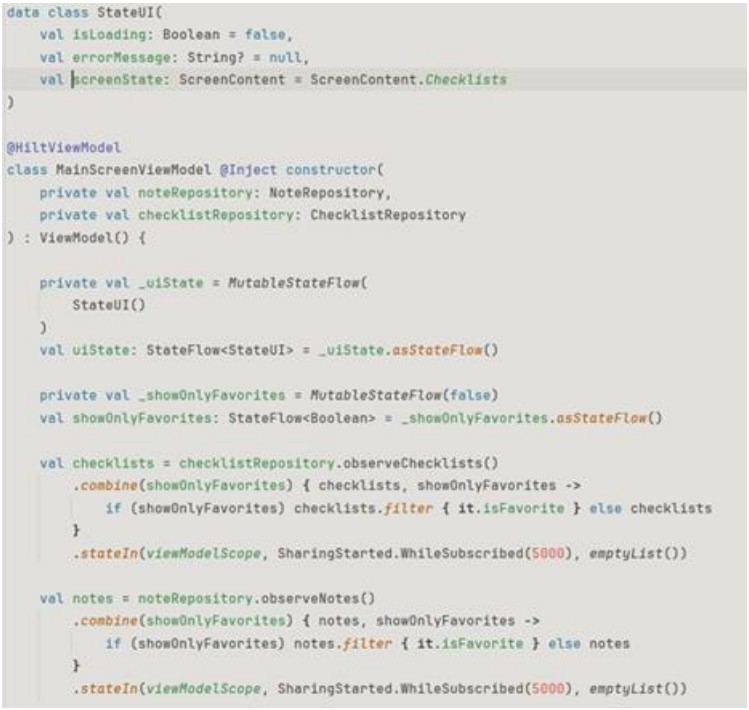
Отже, Android Studio забезпечує інтеграцію з усіма інструментами, створюючи міцну основу для розробки. Стек Kotlin, MVVM, Hilt, Room DB, проєкту. Kotlin гарантує чистоту й безпеку коду, MVVM і Hilt забезпечують модульність, Room DB і Firebase – надійність даних, OpenAI API додає інновації, а Jetpack Compose і MD3 створюють сучасний інтерфейс. Цей набір інструментів дозволяє розробити ефективний, зручний і масштабований застосунок для управління завданнями, готовий до майбутнього розвитку.

3.2 Розробка модуля створення та управління завданнями

Розробка модуля створення та редагування нотаток і чеклістів є центральною частиною програмного модуля мобільної системи управління завданнями, оскільки саме цей компонент забезпечує користувачу основні інструменти для організації щоденної діяльності. Модуль реалізує базовий функціонал, який включає створення текстових нотаток із можливістю їхнього детального редагування, формування чеклістів із завданнями та інтерактивними позначками для відстеження виконання, редагування окремих елементів у цих чеклістах, додавання нотаток і чеклістів до улюблених для швидкого доступу, а

також видалення елементів чеклістів через зручний жест "swipe to delete". Цей набір функцій розроблено з урахуванням потреб сучасного користувача, який потребує гнучкості, інтуїтивності та ефективності в управлінні завданнями, особливо в умовах мультимедійного та офлайн-доступу. Використання таких можливостей, як інтерактивні позначки та жестове видалення, не лише підвищує зручність, але й робить застосунок конкурентним серед аналогічних рішень, дозволяючи користувачу швидко адаптуватися до інтерфейсу. Цей модуль є основою для подальшого розширення функціоналу, наприклад, інтеграції з хмарними сервісами чи інтелектуальними алгоритмами, що планується в майбутніх версіях.

Для реалізації цього модуля обрано архітектуру MVVM (Model-View-ViewModel), яка забезпечує чіткий розподіл обов’язків між компонентами системи. Цей підхід був вибраний через його здатність ізолювати бізнес-логіку від відображення, що полегшує тестування, підтримку та масштабування проєкту. Модуль включає три основні екрани: головний, де відображаються списки нотаток і чеклістів для швидкого огляду та доступу, екран детального перегляду нотатки для редагування тексту та метаданих, а також екран перегляду та редагування чеклісту з інтерактивними елементами. Кожен екран має власну ViewModel: MainScreenViewModel для головного екрану, NoteViewModel для роботи з нотатками та ChecklistViewModel для управління чеклістами. Ці ViewModel-класи відповідають за обробку даних, їхню підготовку та передачу в відповідні репозиторії, що дозволяє уникнути дублювання коду та спрощує взаємодію між шарами. Наприклад, у MainScreenViewModel реалізовано логіку завантаження списків нотаток і чеклістів із репозиторіїв, які отримують дані з локальної бази чи хмари, забезпечуючи їхнє відображення на головному екрані в реальному часі, приклад коду зображено на рисунку 3.1. Такий підхід підвищує стабільність системи, адже зміни в логіці не впливають на зовнішній вигляд, а також полегшує додавання нових функцій, таких як сортування чи фільтрація списків у майбутньому.



```
data class StateUI(
    val isLoading: Boolean = false,
    val errorMessage: String? = null,
    val screenState: ScreenContent = ScreenContent.Checklists
)

@HiltViewModel
class MainScreenViewModel @Inject constructor(
    private val noteRepository: NoteRepository,
    private val checklistRepository: ChecklistRepository
) : ViewModel() {

    private val _uiState = MutableStateFlow(
        StateUI()
    )
    val uiState: StateFlow<StateUI> = _uiState.asStateFlow()

    private val _showOnlyFavorites = MutableStateFlow(false)
    val showOnlyFavorites: StateFlow<Boolean> = _showOnlyFavorites.asStateFlow()

    val checklists = checklistRepository.observeChecklists()
        .combine(showOnlyFavorites) { checklists, showOnlyFavorites ->
            if (showOnlyFavorites) checklists.filter { it.isFavorite } else checklists
        }
        .stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), emptyList())

    val notes = noteRepository.observeNotes()
        .combine(showOnlyFavorites) { notes, showOnlyFavorites ->
            if (showOnlyFavorites) notes.filter { it.isFavorite } else notes
        }
        .stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), emptyList())
```

Рисунок 3.1 – Реалізація MainViewModel для головного екрану

Дані нотаток і чеклістів зберігаються за допомогою двох спеціалізованих репозиторіїв: NoteRepository та ChecklistRepository, що забезпечує логічну організацію даних і спрощує їхнє управління. NoteRepository призначено для обробки текстових нотаток, які зберігаються в таблиці Room із полями для унікального ідентифікатора, заголовка, основного тексту та статусу улюбленого, що дозволяє швидко знаходити та відфільтровувати важливі записи. ChecklistRepository, своєю чергою, управляє чеклістами, зберігаючи назву, список пунктів із їхнім статусом (виконано чи не виконано) та статус улюбленого, що дає змогу відстежувати прогрес і пріоритети користувача. Для передачі даних між ViewModel і інтерфейсом використано StateFlow, реактивний механізм, який гарантує миттєве оновлення екрану при зміні даних, наприклад, коли користувач позначає пункт чеклісту як виконаний. У ChecklistRepository реалізовано методи для створення нових пунктів, їхнього оновлення чи видалення, що дозволяє гнучко адаптувати структуру чеклістів до потреб користувача, таких як додавання підзадач чи реорганізація списку. Цей підхід забезпечує високу ефективність роботи з даними, зменшує навантаження

на систему завдяки кешуванню та полегшує синхронізацію з хмарним сховищем у фоновому режимі.зберігаючи назву, список пунктів, їхній статус (виконано/не виконано) та статус улюбленого. Для передачі даних між ViewModel і інтерфейсом використано StateFlow, що забезпечує реактивне оновлення даних на екранах. Наприклад, у ChecklistRepository реалізовано методи для створення, оновлення та видалення пунктів чеклісту (див. рисунок 3.2).

```
@Singleton
class ChecklistRepositoryImpl @Inject constructor(private val checklistDao: ChecklistDao) :
    ChecklistRepository {

    override fun observeChecklists(): Flow<List<Checklist>> {
        return checklistDao.observeChecklists().map { notes ->
            notes.map { it.toModel() }
        }
    }

    override suspend fun getAll(): List<Checklist> {
        return checklistDao.getAllChecklistsWithTasks().map { it.toModel() }
    }

    override suspend fun getById(id: String): Checklist {
        return checklistDao.getChecklistWithTasksById(id).toModel()
    }

    override suspend fun save(item: Checklist) {
        checklistDao.updateChecklist(item.toEntity())
    }

    override suspend fun save(title: String) {
        checklistDao.insertChecklist(Checklist(title = title).toEntity())
    }

    override suspend fun delete(id: String) {
        checklistDao.deleteChecklistById(id)
    }
```

Рисунок 3.2 – Реалізація ChecklistRepository для роботи з чеклістами

Hilt використано для впровадження залежностей, що стало логічним рішенням для спрощення структури проєкту та уникнення складних конфігурацій. Замість створення окремого доменного шару, який би додав зайву абстракцію і ускладнив код, Hilt забезпечує пряму інжекцію репозиторіїв у відповідні ViewModel-класи, наприклад, NoteRepository інжектується в NoteViewModel. Це дозволяє централізовано управляти залежностями, такими як доступ до бази даних чи хмарних сервісів, без необхідності вручну створювати екземпляри класів. У NoteViewModel реалізовано методи для збереження тексту нотатки, зміни статусу улюбленого та оновлення даних у реальному часі, що підвищує зручність для користувача, адже зміни миттєво відображаються на екрані. Такий підхід зменшує ймовірність помилок,

пов’язаних із неправильною ініціалізацією об’єктів, і спрощує тестування, адже залежності можна легко замінити на мок-об’єкти для імітації роботи з даними. Це також пришвидшує розробку, дозволяючи команді фокусуватися на функціональності, а не на технічних деталях інжекції. У NoteViewModel реалізовано методи для збереження тексту нотатки та зміни статусу улюбленого, що зображено на рисунку 3.3.

```
@HiltViewModel
class NoteViewModel @Inject constructor(
    private val noteRepository: NoteRepository
) : ViewModel() {

    val notes = noteRepository.observeNotes()
        .stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), emptyList())
```

Рисунок 3.3 – Впровадження NoteRepository у NoteViewModel

Функціонал редагування елементів чеклісту включає можливість зміни тексту пунктів, позначення їх як виконаних чи не виконаних, а також видалення через жест "swipe to delete", реалізований за допомогою бібліотеки Jetpack Compose. Цей вибір обґрунтовано сучасними тенденціями дизайну, де інтерактивність і природність жестів покращують досвід користувача. Наприклад, зміна тексту пункту дозволяє адаптувати завдання до нових обставин, а позначення як виконаних дає візуальний зворотний зв’язок, що мотивує користувача. Жест "swipe to delete" реалізовано з анімацією, що робить видалення інтуїтивним і безпечним, адже користувач може скасувати дію перед її завершенням (див. рисунок 3.4). Додавання до улюблених реалізовано через окреме поле в базі даних, яке оновлюється при виборі відповідної опції, що дозволяє швидко знаходити важливі записи через спеціальний фільтр. У ChecklistScreen логіка обробки жесту "swipe to delete" та позначення пунктів як виконаних інтегрована з ViewModel, що забезпечує узгодженість даних між екраном і сховищем. Цей підхід підвищує ергономіку, роблячи управління завданнями швидким і приємним, а також відкриває можливості для додавання

нових жестів, таких як перетягування для зміни порядку пунктів у подальшій розробці.

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun TaskRow(
    task: Task,
    index: Int,
    onTaskCheckedChange: (Boolean) -> Unit,
    onEditTask: (taskId: String) -> Unit,
    onRemoveTask: (Task) -> Unit,
) {
    val context = LocalContext.current
    val currentTask by rememberUpdatedState(task)
    val dismissState = rememberSwipeToDismissBoxState(
        confirmValueChange = {
            if (it == SwipeToDismissBoxValue.StartToEnd) {
                onRemoveTask(currentTask)
                Toast.makeText(context, "Task deleted", Toast.LENGTH_SHORT).show()
                return@rememberSwipeToDismissBoxState true
            }
            return@rememberSwipeToDismissBoxState false
        },
        positionalThreshold = { it * 0.25f }
    )
```

Рисунок 3.4 – Реалізація ChecklistViewModel для редагування чеклістів

Модуль забезпечує стабільну роботу з нотатками та чеклістами, дозволяючи користувачу ефективно створювати, редагувати та управляти завданнями з урахуванням їхнього статусу та вподобань. Стабільність досягається завдяки чіткій архітектурі MVVM і реактивним механізмам, таким як StateFlow, які гарантують, що інтерфейс завжди відображає актуальні дані. Ефективність проявляється в швидкому відгуку на дії користувача, наприклад, при видаленні пункту чи збереженні нотатки, що досягається завдяки оптимізованій роботі з базою даних Room і асинхронності Coroutines. Управління статусами, такими як улюблені чи виконані, додає гнучкості, дозволяючи користувачу персоналізувати свій робочий простір. Цей модуль не лише виконує базові функції, але й закладає міцну основу для майбутнього розширення, наприклад, інтеграції з голосовими командами чи аналітикою

продуктивності, що зробить застосунок ще більш універсальним і корисним для різноманітних сценаріїв використання.

3.3 Розробка модуля хмарного бекапу та синхронізації

Щоденний нічний бекап є критичною функцією мобільної системи управління завданнями, яка забезпечує збереження даних користувача, таких як нотатки та чеклісти, на випадок втрати пристрою чи видалення додатку. Ця функція дозволяє синхронізувати локальні дані з хмарним сховищем у фоновому режимі, гарантуючи їхню доступність і цілісність. Реалізація бекапу виконується автоматично щоденно вночі, коли пристрій, ймовірно, не використовується, що мінімізує вплив на продуктивність і зручність користувача. Використання хмарного сховища, зокрема Firebase Firestore, обрано через його підтримку офлайн-режиму та автоматичну синхронізацію, що ідеально відповідає вимогам проєкту. Автентифікація для доступу до хмарних даних здійснюється через Google OAuth із використанням Gmail-акаунта, що забезпечує безпечний і зручний вхід, а також інтеграцію з екосистемою Google. Цей підхід не лише захищає дані, але й дозволяє користувачу легко відновити доступ із іншого пристрою, що є важливим для довіри до системи. Реалізація включає кілька ключових класів, які координують процес бекапу, обробку автентифікації та взаємодію з хмарою.

Основним класом, який керує процесом бекапу, є BackupScheduler, частино коду наведено на рисунку 3.5. Цей клас відповідає за планування щоденного завдання вночі, використовуючи WorkManager – бібліотеку Android, яка дозволяє виконувати фонові операції навіть після закриття додатку. BackupScheduler ініціалізує періодичне завдання, яке запускається о 2:00 ночі за місцевим часом, враховуючи часовий пояс користувача, що забезпечує гнучкість для глобальної аудиторії. Використання WorkManager обрано через його надійність: воно гарантує виконання навіть при перезавантаженні пристрою чи низькому заряді батареї, що є критично важливим для автоматичного бекапу. Клас містить логіку перевірки підключення до мережі

перед початком роботи, щоб уникнути нестабільних синхронізацій, а також метод для відкладеного виконання, якщо пристрій перебуває у сплячому режимі. Перевагою такого підходу є мінімальне втручання в роботу користувача, адже бекап виконується непомітно, а також можливість налаштувати повторні спроби в разі збою через погане з'єднання, що підвищує надійність системи.

```
@HiltWorker
class BackupWorker @AssistedInject constructor(
    @Assisted context: Context,
    @Assisted params: WorkerParameters,
    private val coordinator: BackupCoordinator
) : CoroutineWorker(context, params) {
    override suspend fun doWork(): Result {
        return coordinator.runBackup()
    }
}

object BackupScheduler {
    fun schedule(context: Context) {
        val constraints = Constraints.Builder()
            .setRequiredNetworkType(NetworkType.CONNECTED)
            .setRequiresBatteryNotLow(true)
            .build()

        val workRequest = PeriodicWorkRequestBuilder<BackupWorker>(1, TimeUnit.DAYS)
            .setConstraints(constraints)
            .setInitialDelay(calculateInitialDelay())
            .build()

        WorkManager.getInstance(context).enqueueUniquePeriodicWork(
            "NightlyBackup",
            ExistingPeriodicWorkPolicy.UPDATE,
            workRequest
        )
    }
```

Рисунок 3.5 – Реалізація BackupScheduler для процесу бекапу

Другий ключовий клас – BackupService відповідає за безпосереднє виконання бекапу. Цей клас отримує дані від BackupScheduler і координує взаємодію з локальною базою даних Room, де зберігаються нотатки та чеклісти, а також із Firebase Firestore для їхнього завантаження (див. рисунок 3.6). BackupService використовує асинхронні операції через Kotlin Coroutines, що дозволяє обробляти великі обсяги даних, такі як сотні нотаток із вкладеними чеклістами, без блокування основного потоку. Клас включає методи для

вибірки даних із Room, їхньої серіалізації в JSON-формат для передачі та оновлення стану в Firestore. Використання Coroutines обрано через їхню здатність управляти складними потоками, забезпечуючи плавність роботи навіть при великій кількості записів. Перевагою є також можливість паралельного завантаження даних, що скорочує час бекапу, наприклад, обробляючи нотатки та чеклісти одночасно. Клас також логірує успішне завершення чи помилки, що дозволяє розробникам аналізувати проблеми, такі як перевищення ліміту API, і вдосконалювати систему в майбутньому.

```
class BackupService @Inject constructor(
    private val database: AppDatabase,
    private val firestore: FirebaseFirestore
) {
    suspend fun performBackup(userId: String) {
        val notes = database.noteDao().getAllNotes()
        val checklists = database.checklistDao().getAll()

        val batch = firestore.batch()
        val userRef = firestore.collection("backups").document(userId)

        notes.forEach {
            val ref = userRef.collection("notes").document(it.id)
            batch.set(ref, it)
        }

        checklists.forEach {
            val ref = userRef.collection("checklists").document(it.id)
            batch.set(ref, it)
        }

        batch.commit().await()
    }
}
```

Рисунок 3.6 – Реалізація BackupService для виконання бекапу

Автентифікація для доступу до хмарного сховища реалізується через Google OAuth із Gmail акаунтом, що забезпечує безпечний і зручний вхід. Процес починається у класі AuthManager, який налаштовує запит на отримання доступу до профілю та Firestore (див. рисунок 3.7). Користувач авторизується через стандартний діалог Google, надаючи дозвіл на доступ до своїх даних, що зберігаються в безпечному токені. Цей токен передається до Firebase

Authentication для верифікації та створення користувацької сесії, яка використовується для авторизованих запитів до Firestore. Такий підхід обрано через інтеграцію з існуючою екосистемою Google, що спрощує вхід для більшості користувачів, які вже мають Gmail-акаунти. Переваги включають високий рівень безпеки завдяки шифруванню токенів, автоматичне оновлення доступу та можливість одноразового входу з подальшою безперервною роботою. Наприклад, після першого входу AuthManager зберігає стан автентифікації локально, що дозволяє бекапу виконуватися без повторного запиту дозволу, підвищуючи швидкість і зручність. У разі втрати токена клас автоматично ініціює повторну автентифікацію, що забезпечує безперервність роботи.

```
class AuthManager @Inject constructor(
    private val firebaseAuth: FirebaseAuth,
    private val context: Context
) {
    fun isSignedIn(): Boolean = firebaseAuth.currentUser != null
    fun getUserId(): String = firebaseAuth.currentUser?.uid.orEmpty()
}
```

Рисунок 3.7 – Реалізація AuthManager для процесу авторизації

Для координації між BackupScheduler, BackupService і AuthManager створено допоміжний клас BackupCoordinator (див. рисунок 3.8). Цей клас виступає посередником, забезпечуючи передачу даних про стан автентифікації та результати бекапу між компонентами. BackupCoordinator перевіряє, чи користувач авторизований, перед запуском бекапу, і у випадку відсутності доступу викликає AuthManager для повторного входу. Він також обробляє помилки, такі як нестабільне з’єднання чи перевищення квоти Firestore, пропонуючи альтернативні стратегії, наприклад, відкладений бекап. Використання такого координатора обрано для централізації логіки, що зменшує зв’язність між класами й полегшує їхнє тестування. Перевагою є можливість розширити функціонал, наприклад, додавши сповіщення

користувачу про успішний бекап чи проблеми з мережею, що підвищить прозорість системи. Клас також підтримує конфігурацію часу бекапу, дозволяючи в майбутньому додати опцію налаштування користувачем, що зробить застосунок більш гнучким.

```
class BackupCoordinator @Inject constructor(
    private val authManager: AuthManager,
    private val backupService: BackupService
) {
    suspend fun runBackup(): Result {
        return try {
            if (!authManager.isSignedIn()) {
                return Result.retry()
            }
            backupService.performBackup(authManager.getUserId())
            Result.success()
        } catch (e: Exception) {
            Result.retry()
        }
    }
}
```

Рисунок 3.8 – Реалізація BackupCoordinator для взаємозв’язку між класами

Процес бекапу починається з ініціалізації BackupScheduler, який планує завдання через WorkManager на 2:00 ночі. Перед запуском BackupCoordinator перевіряє стан автентифікації через AuthManager, використовуючи збережений токен Google OAuth. Якщо токен дійсний, BackupService отримує дані з Room, серіалізує їх і передає до Firestore. У разі успіху BackupCoordinator логірує результат, а при помилках, таких як відсутність мережі, планує повторну спробу через годину. Такий підхід забезпечує надійність, адже навіть при тимчасових збоїв дані не втрачаються, а система адаптується до умов. Перевагою є автоматизація всього процесу, що звільняє користувача від ручного управління, а також гнучкість у обробці помилок, яка дозволяє підтримувати стабільність навіть у складних сценаріях, таких як подорожі з мінливим покриттям мережі.

Реалізація щоденного нічного бекапу забезпечує користувачу впевненість у збереженні даних і полегшує відновлення після втрати пристрою.

Використання WorkManager гарантує виконання навіть при закритому додатку, а Coroutines оптимізують обробку даних, зменшуючи час операції. Google OAuth із Gmail забезпечує безпечний доступ, а централізована логіка в BackupCoordinator полегшує розширення функціоналу. Ця система закладає основу для майбутніх удосконалень, таких як інкрементальний бекап чи підтримка кількох хмарних провайдерів, що зробить застосунок ще більш надійним і універсальним.

3.4 Розробка модулів графічного інтерфейсу та кастомізації

При розробці інтерфейсу особливу увагу приділено зрозумілості та зручності, щоб забезпечити інтуїтивне керування нотатками, чеклістами та кастомізацією. Інтерфейс спроєктовано відповідно до сучасних стандартів дизайну мобільних додатків, що сприяє швидкому освоєнню для користувачів із різним рівнем досвіду [20-21].

Базовою темою інтерфейсу обрано персикову, яка створює м’який і приємний візуальний досвід, сприяючи зосередженості під час роботи із завданнями. Додаток також підтримує різні кольорові теми (Pitch, Purple, Green, Amber), які розроблені за стандартом Material Design 3 із адаптованими кольорами шрифтів до поверхні, що забезпечує контрастність і читабельність у всіх режимах. Кольорові схеми підібрано так, щоб підтримувати адаптивність інтерфейсу до потреб користувачів.

Розробка інтерфейсу проводитиметься в середовищі Android Studio з використанням бібліотеки Jetpack Compose, яка забезпечує створення динамічних і адаптивних компонентів інтерфейсу. Jetpack Compose дозволила реалізувати гнучкі елементи, такі як списки, діалогові вікна та перемикачі налаштувань, що сприяє зручності користувацького досвіду.

При запуску додатку користувач потрапляє на початковий екран списку чеклістів (див. рисунок 3.9), який відображає створені списки, у простому та зручному форматі. Цей екран забезпечує швидкий доступ до основного функціоналу, дозволяючи створювати нові чеклісти та нотатки чи відкривати наявні для редагування.

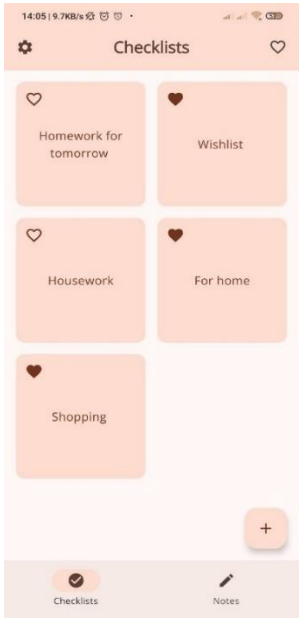


Рисунок 3.9 – Екран списку чеклістів

Екран налаштувань (див. рисунок 3.10) надає доступ до основних функцій кастомізації та інформації про додаток.

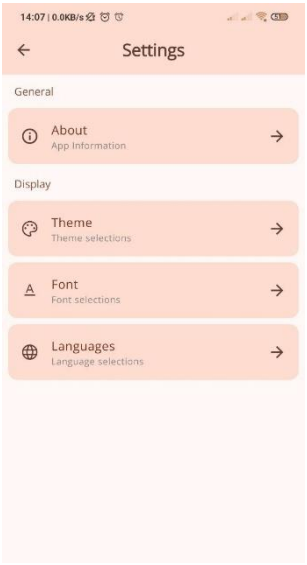


Рисунок 3.10 – Екран налаштувань

Екран налаштувань містить пункти меню, такі як "About" для перегляду інформації про програму, "Theme selections" для вибору кольорової теми, "Font selections" для налаштування шрифтів і "Language selections" для вибору мови інтерфейсу. Інтерфейс виконано у вигляді структурованого списку, що полегшує навігацію.

Екран чеклісту (див. рисунок 3.11) відображає список завдань для обраного чеклісту. На цьому екрані зображені пункти чеклисту із позначками виконання. Користувач може редагувати пункти, видаляти їх або додавати нові, використовуючи кнопки у правій частині екрана. Інтерфейс виконано в зрозумілому стилі, із чітким розмежуванням між виконаними та невиконаними і парними та непарними завданнями для кращого розділення.

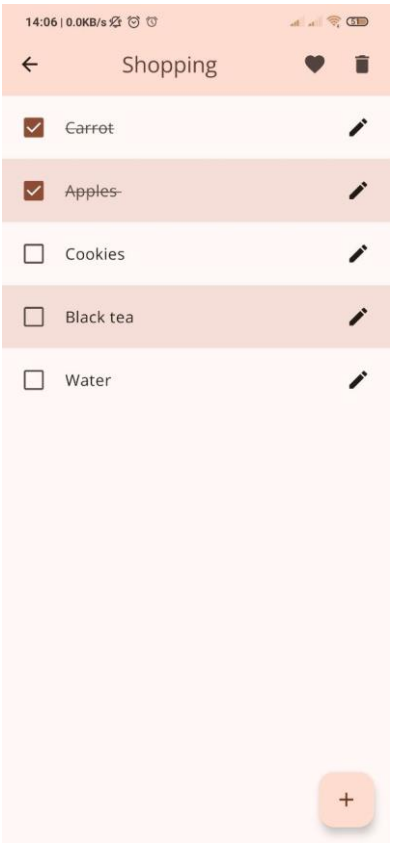


Рисунок 3.11 – Екран чеклісту

Екран створення пункту чеклісту (див. рисунок 2.4) дозволяє користувачу додавати нові завдання до списку, наприклад. При натисканні на кнопку додавання з’являється діалогове вікно "Input new task title", де можна ввести назву завдання, і підтвердити її. Інтерфейс цього екрана виконано в мінімалістичному стилі, із чіткими полями введення та кнопками "Cancel" і "OK", що забезпечує зручність взаємодії. Ідентичні екрани створення застосовуються у всіх місцях в додатку де потрібно щось додати.

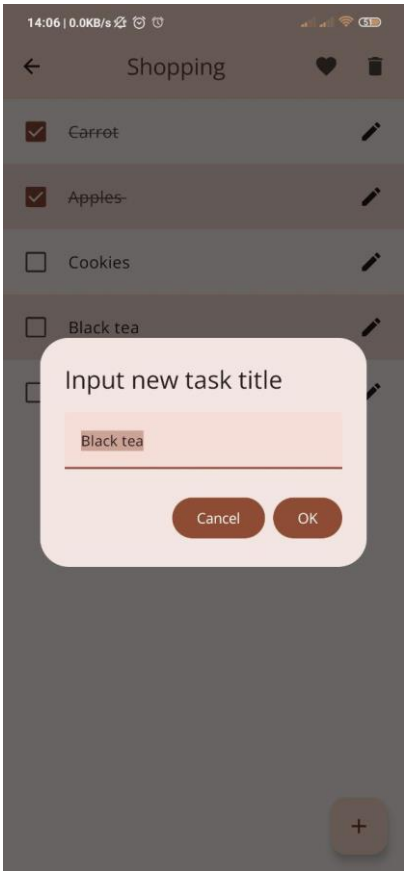


Рисунок 3.12 – Екран створення пункту чеклісту

Отже, інтерфейс власного модуля розроблено з акцентом на інтуїтивність і гнучкість, що забезпечується продуманим вибором кольорових тем, зручними екранами та підтримкою кастомізації. Використання Jetpack Compose та гайдлайнів Meterial Design 3 дозволило створити сучасний інтерфейс, який відповідає потребам користувачів у ефективному управлінні особистими завданнями.

Розробка модуля кастомізації інтерфейсу користувача є важливою частиною застосунку, що забезпечує адаптивність додатку до індивідуальних потреб користувача. Модуль дозволяє налаштовувати, шрифт і мову інтерфейсу. Зміни зберігаються локально та застосовуються в реальному часі без перезапуску застосунку.

Модуль реалізовано з використанням архітектури MVVM, що забезпечує чіткий розподіл обов'язків між компонентами програми. Для обробки налаштувань створено SettingsViewModel, яка відповідає за управління даними кастомізації. SettingsViewModel використовує Flow для передачі даних, які колектяться як State у MainActivity, що дозволяє реактивно оновлювати інтерфейс. Наприклад, при виборі нової теми чи шрифту SettingsViewModel передає оновлені значення через Flow, і інтерфейс миттєво відображає зміни. Реалізацію зображено на рисунку 3.13.

```
data class StateUI(
    var screenState: ScreenContent = ScreenContent.Settings
)

@HiltViewModel
class SettingsViewModel @Inject constructor(
    private val settingsRepository: SettingsRepository
) : ViewModel() {

    private val _uiState = MutableStateFlow(
        StateUI()
    )
    val uiState: StateFlow<StateUI> = _uiState.asStateFlow()

    fun updateUi(newState: ScreenContent) {
        _uiState.update { it.copy(screenState = newState) }
    }
```

Рисунок 3.13 – Реалізація SettingsViewModel із Flow для обробки налаштувань

Налаштування зберігаються локально за допомогою DataStore, що забезпечує асинхронне та безпечне зберігання даних. Для управління темами та шрифтами створено клас Themes, який містить методи для збереження та отримання відповідних параметрів. Наприклад, метод збереження теми в

DataStore записує вибране значення, яке потім застосовується до всіх елементів інтерфейсу. Цей клас також забезпечує методи для оновлення шрифтів, дозволяючи користувачу бачити зміни безпосередньо на кнопках вибору. Реалізацію зображено на рисунку 3.14.

```
@Composable
fun MyNotePPTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    dynamicColor: Boolean = false,
    selectedFont: FontType,
    selectedThemeIndex: Int,
    content: @Composable () -> Unit
) {

    val colorScheme = when (selectedThemeIndex) {
        0 -> if (darkTheme) darkPitchScheme else lightPitchScheme
        1 -> if (darkTheme) darkPurpleScheme else lightPurpleScheme
        2 -> if (darkTheme) darkGreenScheme else lightGreenScheme
        3 -> if (darkTheme) darkAmberScheme else lightAmberScheme
        else -> if (darkTheme) darkPitchScheme else lightPitchScheme
    }

    val typography = when (selectedFont) {
        FontType.OPEN_SANS -> openSansTypography
        FontType.LATO -> latoTypography
        FontType.NUNITO -> nunitoTypography
        FontType.POPPINS -> poppinsTypography
        FontType.QUICKSAND -> quicksandTypography
        FontType.RALEWAY -> ralewayTypography
        FontType.MONTSERRAT -> montserratTypography
    }
```

Рисунок 3.14 – Реалізація класу Themes для збереження тем і шрифтів у DataStore

Мова інтерфейсу та застосування тем і шрифтів обробляються через MainActivity. Зміна мови потребує перезапуску активності, щоб оновити всі текстові ресурси програми. MainActivity отримує дані з DataStore і застосовує налаштування до інтерфейсу, забезпечуючи синхронізацію між вибраними параметрами та їхнім відображенням. Наприклад, при виборі української мови MainActivity перезапускає активність, після чого всі текстові елементи оновлюються (див. рисунок 3.15).

Рисунок 3.15 – Реалізація MainActivity для застосування мови

Hilt використано для впровадження залежностей у модулі. SettingsViewModel отримує доступ до DataStore через прямий інжект без доменного шару, що спрощує структуру коду та підвищує ефективність [17]. Наприклад, інжекція в SettingsViewModel дозволяє швидко отримувати та оновлювати налаштування, забезпечуючи модульність і легкість тестування (див. рисунок 3.16).

Рисунок 3.16 – Впровадження залежностей у SettingsViewModel через Hilt

Оскільки попередній перегляд змін не передбачено, користувач бачить, як виглядає шрифт чи колір безпосередньо на кнопках вибору. Наприклад, кнопка Amber при вибору будь-якої теми виглядає так ніби обрана тема Amber, а шрифт кнопки вибору відображається у вибраному стилі, що дозволяє користувачу одразу оцінити результат (див. рисунок 3.17).

```
items(themes) { theme ->
    MyNotePPTheme(
        selectedThemeIndex = themes.indexOf(theme),
        selectedFont = selectedFont,
        darkTheme = selectedMode
    ) {
        ThemeItem(
            themeColorTitle = theme.toString(),
            isSelected = themes.indexOf(theme) == selectedTheme,
            painter = painterResource(R.drawable.ic_palette),
            onClick = { onThemeSelected(themes.indexOf(theme)) }
        )
    }
}
```

Рисунок 3.17 – Реалізація попереднього перегляду завдяки темізації окремих компонентів інтерфейсу в Jetpack Compose

Модуль кастомізації забезпечує адаптивність і зручність додатку, дозволяючи користувачу налаштовувати інтерфейс із миттєвим відображенням змін, що значно покращує досвід роботи з нотатками та чеклістами, відповідаючи індивідуальним уподобанням.

3.5 Висновки

У третьому розділі було виконано комплексну розробку ключових компонентів мобільної системи управління завданнями. Реалізовано модуль створення та редагування нотаток і чеклістів, який включає створення текстових нотаток, їхнє редагування, управління чеклістами з інтерактивними позначками, додавання до улюблених і видалення через жест "swipe to delete", використовуючи архітектуру MVVM, Room DB, Hilt і Jetpack Compose для інтуїтивного інтерфейсу. Розроблено механізм щоденного нічного інтегрованого з Firebase Firestore через Google OAuth із Gmail для безпечної синхронізації даних. Також створено модулі графічного інтерфейсу та кастомізації, які забезпечують адаптивний дизайн із підтримкою тем, шрифтів і мов.

4 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Тестування програмного забезпечення

Тестування програмного забезпечення є невід’ємною частиною життєвого циклу розробки будь-якого додатку, незалежно від його складності чи масштабів [22]. Воно дозволяє виявити критичні помилки, оцінити відповідність функціональності поставленим вимогам і забезпечити необхідний рівень якості продукту перед його впровадженням та використанням кінцевими користувачами.

Під час тестування мобільного додатку для управління завданнями особлива увага приділялася перевірці сумісності з різними версіями Android: 9.0 (Pie), 10.0 (Q) та 11.0 (R). Для забезпечення максимальної охопленості тестування використовувалися як фізичні пристрої, так і емулятори Android Studio.

Перед безпосереднім початком тестування було складено детальний план, який охоплював кілька основних напрямів: функціональне тестування, тестування на відмову, продуктивності та сумісності. Кожен із цих етапів мав на меті всебічно перевірити додаток, зокрема його здатність виконувати заявлені функції, відновлювати роботу після помилок, працювати з адекватною швидкістю навіть на слабких пристроях і бути стабільним незалежно від версії ОС чи типу пристрою [23].

Функціональне тестування було зосереджене на перевірці коректної роботи основних можливостей додатку. Одним із перших тестів стала перевірка створення нотатки. Користувач вводив текст, наприклад, "Meeting notes", після чого зберігав його. У результаті додаток створював відповідну нотатку і відображав її у загальному списку на головному екрані (див. рисунок 4.1), що свідчить про правильне збереження і виведення даних. Наступним кроком стало тестування функції редагування. Текст існуючої нотатки змінювався на "Updated meeting notes", після чого перевірялося, чи збережено ці зміни. Додаток успішно оновлював вміст нотатки, і новий текст коректно

відображався (див. рисунок 4.2), що підтвердило працездатність відповідного функціоналу.

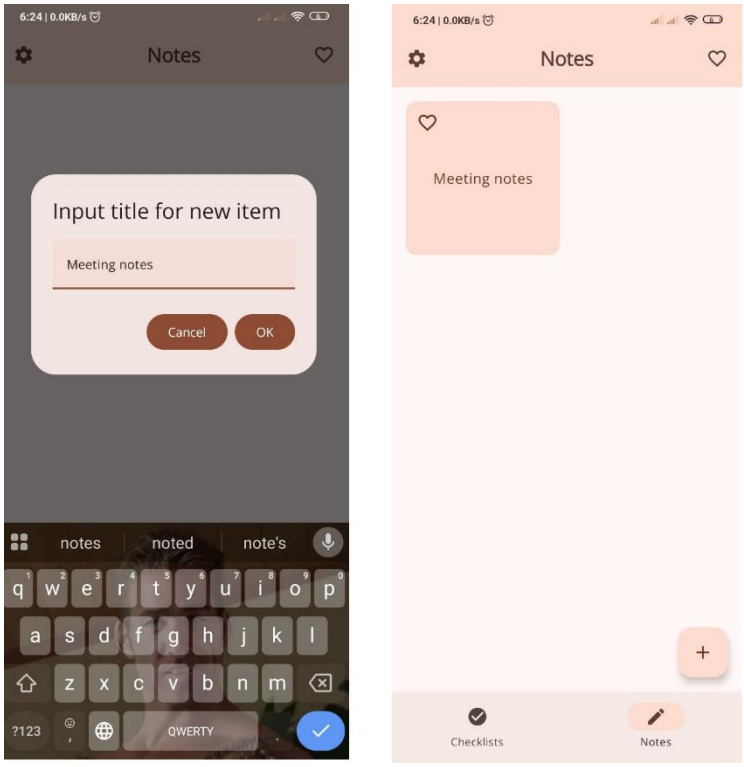


Рисунок 4.1 – Результат створення нотатки

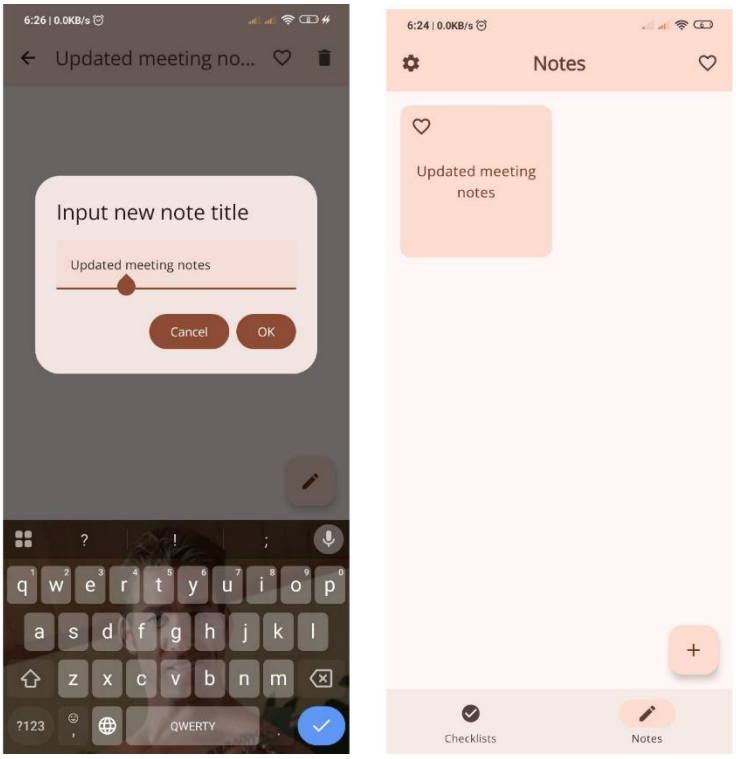


Рисунок 4.2 – Результат редагування нотатки

Наступним етапом тестування стала перевірка функціоналу, пов’язаного з роботою з чеклістами, який є ключовим для користувачів, що прагнуть структуровано фіксувати та відстежувати виконання завдань. Було перевірено можливість створення нового чеклісту під назвою “Shopping”, який містить типові пункти покупок, наприклад, “Carrot” та “Apples”. Процес додавання елементів до списку пройшов без збоїв – введення назви списку та окремих пунктів виконується у зрозумілому форматі, з автоматичним фокусом на полі вводу нового пункту. Після завершення додавання всіх необхідних позицій список був збережений, і всі введені пункти відобразилися у загальному списку чеклістів, що підтверджує правильність збереження структури даних.

Наступним кроком було тестування функціоналу позначення пунктів чеклісту як виконаних ” (див. рисунок 4.3). При натисканні на чекбокс поряд із назвою пункту, додаток миттєво змінював стан пункту – візуально це супроводжувалося зміною стилю тексту (наприклад, через перекреслення), що є стандартною практикою у подібних застосунках. Усі зміни були збережені в локальній базі даних, і після перезапуску додатку статуси пунктів збереглися, що вказує на правильну інтеграцію з Room та коректне оброблення подій UI.

Окрему увагу було приділено тестуванню функції видалення елементів списку через жест “swipe to delete” (див. рисунок 4.4). Для цього пункт “Apples” був видалений за допомогою горизонтального свайпу, після чого елемент одразу зник зі списку. Жодних візуальних чи логічних помилок при цьому не зафіксовано – ані помилок у логах, ані збоїв у поведінці інтерфейсу. Повторна перевірка з іншими пунктами чеклісту показала аналогічний результат, що дозволяє вважати реалізацію цього механізму стабільною та надійною.

Також було протестовано поведінку у випадку, якщо користувач видаляє усі пункти чеклісту, залишаючи список порожнім. Додаток коректно обробив цю ситуацію: відображення списку оновилося, і було запропоновано додати нові пункти. Жодних збоїв або зависань інтерфейсу внаслідок відсутності елементів не виявлено.

Загалом результати тестування функцій створення, редагування, позначення як виконаних та видалення пунктів чеклісту підтвердили їх коректну реалізацію. Всі зміни були оперативно застосовані до інтерфейсу, синхронізовані з локальною базою та зберігалися між сеансами використання. Реалізація відповідає сучасним вимогам до UX для продуктивних застосунків і забезпечує користувачу швидкий, зручний та передбачуваний досвід керування завданнями.

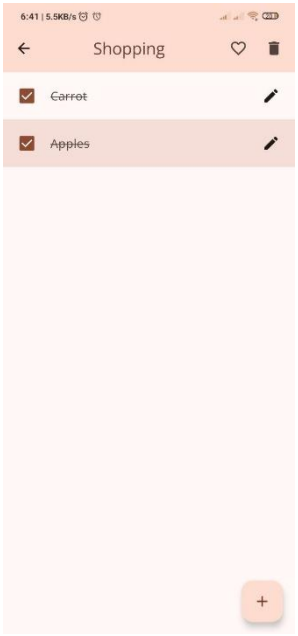


Рисунок 4.3 – Результат створення та позначення пунктів чеклісту

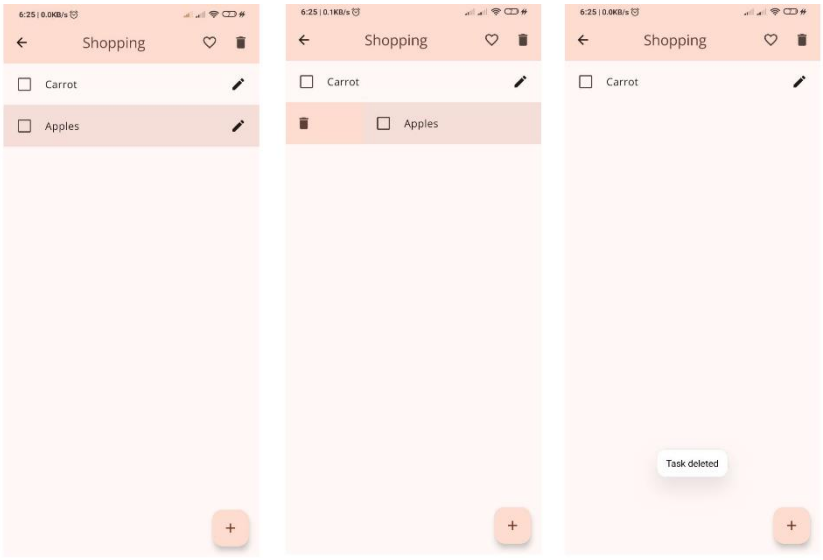


Рисунок 4.4 – Результат видалення пункту через "swipe to delete"

Окремий етап тестування було присвячено обробці відмов та перевірці стійкості додатку до нетипових або некоректних сценаріїв використання. Таке тестування є критично важливим для гарантування стабільної роботи застосунку в умовах, коли користувач виконує дії, не передбачені звичайним алгоритмом взаємодії, або ж виникають технічні збої, пов'язані з обладнанням або системним середовищем.

Одним із базових тестів стало моделювання ситуації, коли користувач намагається створити нову нотатку, не ввівши жодного тексту. У багатьох недопрацьованих застосунках подібна дія призводить до непередбачуваних результатів або навіть аварійного завершення роботи. Проте в тестованому додатку було реалізовано валідацію вводу на рівні UI: при спробі збереження порожнього поля система виводить попередження, а сама дія блокується. Завдяки цьому додаток не переходить до збереження некоректних даних і не створює порожні об’єкти в базі. Такий підхід свідчить про передбачення потенційно хибних сценаріїв та адекватну реалізацію механізмів запобігання логічним помилкам користувача (див. рисунок 4.5).

Крім цього, було змодельовано ситуацію раптового завершення роботи додатку в момент редагування вже існуючої нотатки. Для цього під час відкриття нотатки та внесення змін було штучно викликано примусове завершення процесу (через меню розробника Android або adb). Після повторного запуску додатку перевірялося, чи було збережено внесені зміни. Завдяки використанню бази даних Room, яка підтримує транзакції та виконує збереження в момент підтвердження дій, усі зміни, що були явно збережені користувачем до аварійного завершення, залишились доступними після повторного запуску. Важливо, що не зберігалися ті зміни, які не були підтверджені – це також відповідає вимогам до передбачуваного UX і дозволяє уникнути часткового або неконсистентного збереження даних (див. рисунок 4.6).

Додатково було протестовано випадки, коли система Android автоматично вивантажує додаток з пам’яті (наприклад, при нестачі оперативної пам’яті) під час активної взаємодії користувача. Після повернення до додатку

через останні запущені програми, додаток коректно відновлював свій стан завдяки використанню Jetpack ViewModel та збереження стану UI, що підтверджує дотримання принципів життєвого циклу компонентів Android та архітектурної стійкості.

Таким чином, результати тестування обробки відмов демонструють, що додаток стійко реагує на поширені помилки користувача та технічні збої, не допускаючи втрати даних або аварійного завершення роботи. Реалізовані механізми валідації вводу, надійної роботи з базою даних та дотримання життєвого циклу компонентів забезпечують стабільність роботи застосунку навіть у несприятливих умовах експлуатації.

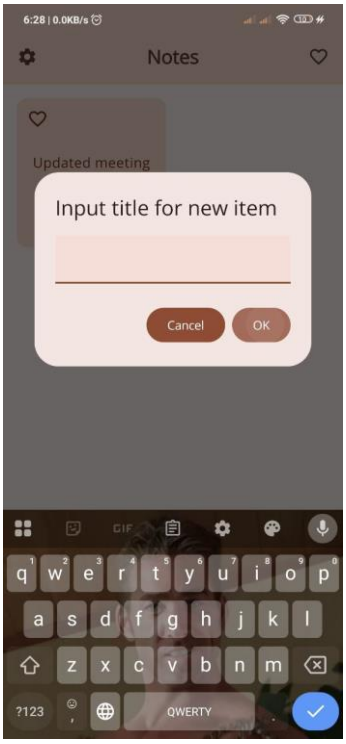


Рисунок 4.5 – Неможливість створення пустої назви

Тестування продуктивності оцінювало швидкість виконання операцій. Час створення нотатки склав <0.1 секунди, а оновлення чеклісту з 10 пунктами – <0.15 секунди, що відповідає стандартам для мобільних додатків. Перевірка сумісності показала стабільну роботу на всіх тестових версіях Android без збоїв.

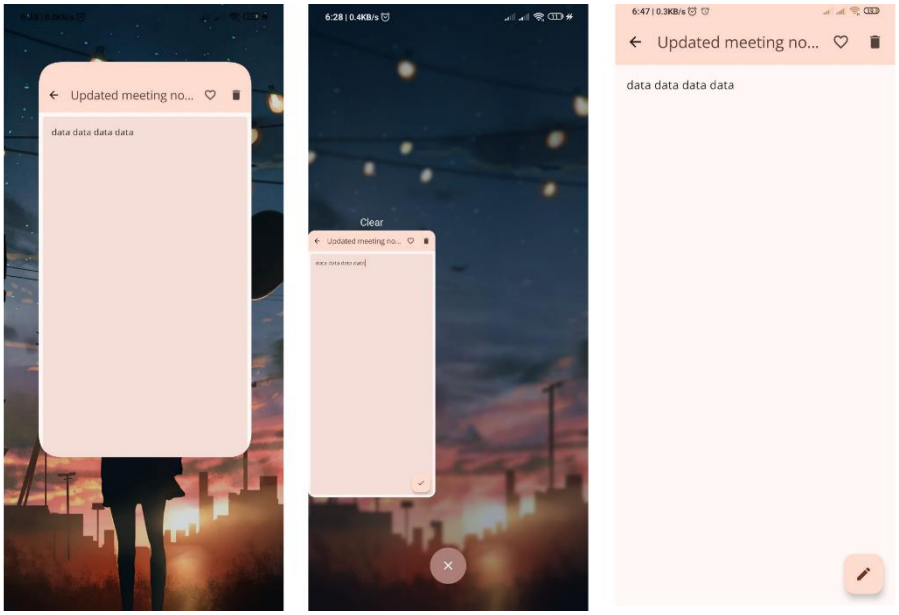


Рисунок 4.6 – Відновлення даних після раптового завершення

Тестування кастомізації перевіряло зміну теми та мови. Вибір теми Purple застосувався миттєво, а зміна мови на українську після перезапуску активності оновила інтерфейс коректно (див. рисунок 4.7). Загалом тестування підтвердило, що додаток працює стабільно, відповідає функціональним вимогам і забезпечує зручність використання.

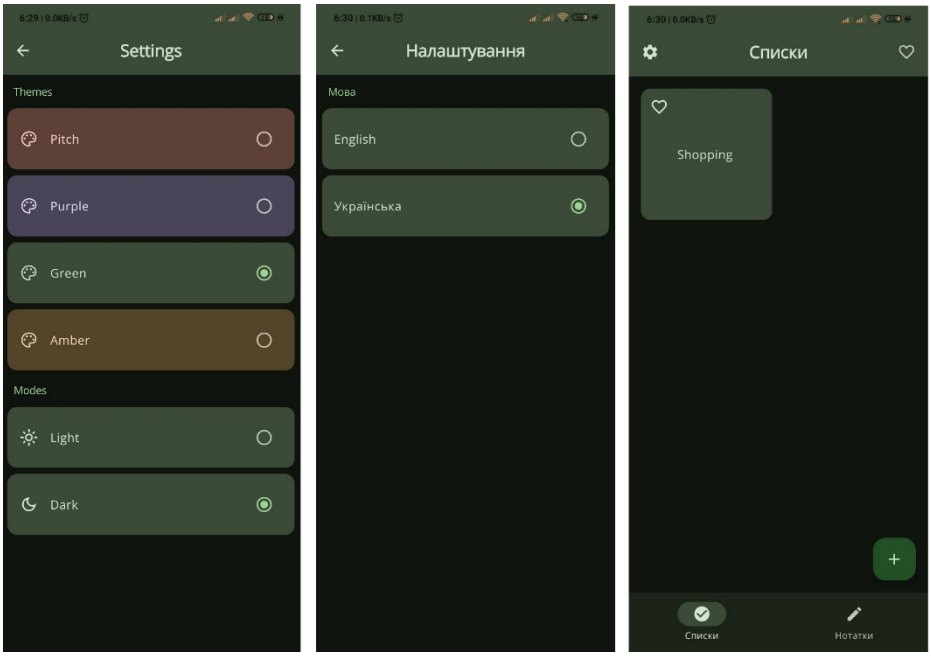


Рисунок 4.7 – Результат застосування нової теми та мови

У підсумку, тестування мобільного додатку для управління завданнями продемонструвало його відповідність заявленим функціональним вимогам та стабільну роботу в різноманітних умовах. Було перевірено основні сценарії взаємодії з інтерфейсом: створення, редагування та видалення нотаток і чеклістів, зміна теми та мови, а також реакція на нестандартні ситуації, зокрема відсутність вводу або раптове завершення програми. Усі ці дії додаток виконував без збоїв, з належним рівнем стабільності й збереженням даних.

Крім того, тестування охопило кілька типів пристроїв з різними версіями Android, що дозволило перевірити сумісність та адаптивність додатку до різних конфігурацій. Особливу увагу приділено продуктивності: час виконання основних операцій виявився мінімальним, що відповідає сучасним вимогам до швидкодії мобільних застосунків [21]. Усі виявлені під час тестування незначні помилки були проаналізовані та виправлені. Таким чином, можна зробити висновок, що додаток функціонує надійно, зручно для користувача та готовий до подальшого розгортання або використання.

Тестування локального завантаження перевіряло правильність стягнення чеклістів із хмарного сховища Firebase Firestore на локальну базу Room, що є ключовим для забезпечення доступності даних у офлайн-режимі. Цей процес критично важливий, адже користувачі часто працюють у умовах нестабільного інтернет-з’єднання, наприклад, у транспорті чи віддалених місцях, де доступ до мережі обмежений, і система має гарантувати безперебійний доступ до їхніх чеклістів без втрати даних. На смартфоні Redmi Note 7 і емульованому Google Pixel у середовищі Android Studio було виконано синхронізацію, після чого чеклісти успішно завантажилися на локальне сховище, з точним відображенням статусів для пунктів які відмічені як улюблені (див. рисунки 4.8). Тестування підтвердило, що локальне завантаження працює коректно, забезпечуючи цілісність даних чеклістів, що дозволяє користувачу покладатися на додаток навіть без підключення до мережі.

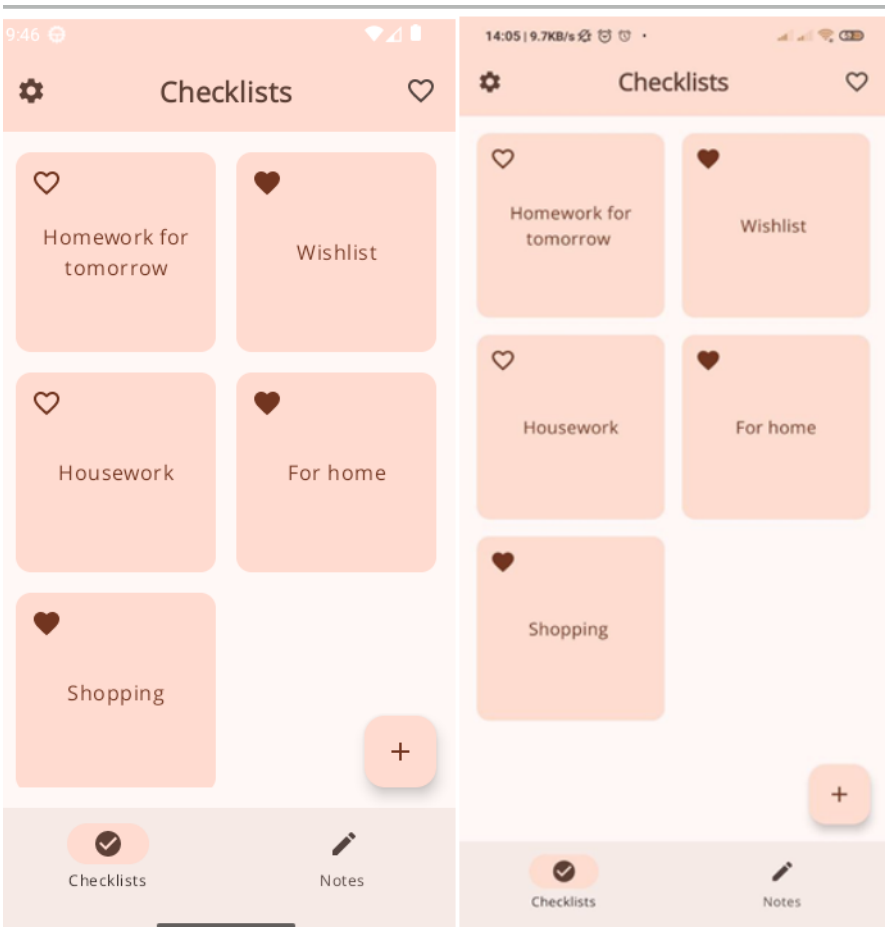


Рисунок 4.8 – Тестування хмарної синхронізації на різних пристроях

Тестування інтелектуального створення чеклістів перевіряло коректність обробки голосового вводу через OpenAI API для автоматичного формування списків завдань. Користувач ввів голосову команду "я йду в магазин мені треба купити молоко яйця дві буханки хліба крупи пшеничної і напевно мінералки теж взяти", після чого API коректно інтерпретувало ввід і сформувало чекліст із пунктами: "молоко", "яйця", "дві буханки хліба", "крупи пшеничної", "мінеральна вода", врахувавши неформальну фразу (див. рисунок 4.9). Тестування показало, що інтелектуальне створення чеклістів працює ефективно, адаптуючись до природної мови з додатковими уточненнями, і забезпечує зручність автоматизації для користувача.

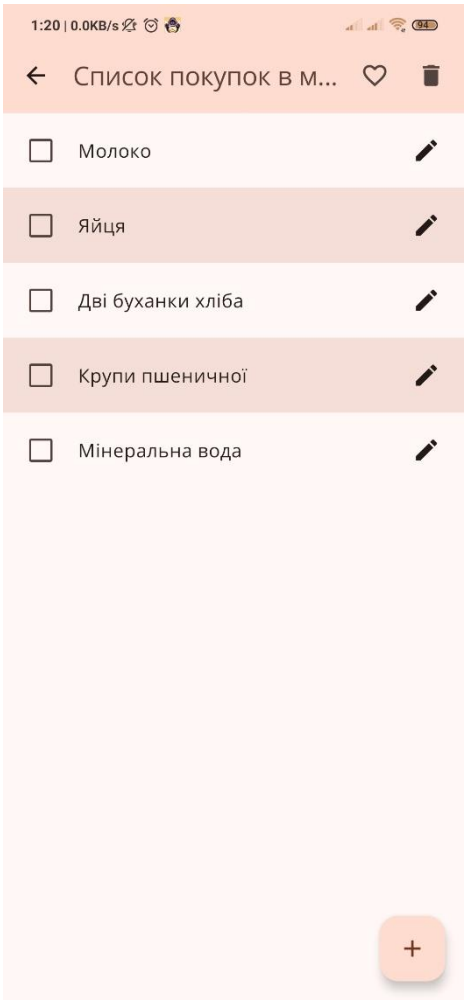


Рисунок 4.9 – Тестування формування інтелектуального чекліста на основі вводу користувача

4.2 Розробка інструкції користувача

Інструкція користувача визначає ключові вимоги до мобільного додатку управління завданнями, а також описує базову логіку взаємодії з інтерфейсом. Зважаючи на те, що додаток реалізовано відповідно до рекомендацій Material Design 3, які передбачають інтуїтивно зрозумілу структуру та елементи управління, повноцінна покрокова інструкція не є обов’язковою [7]. Основна увага приділяється технічним характеристикам пристрою, з яким додаток працює стабільно, та короткому опису ключових функцій.

Мобільний додаток орієнтований на сучасні Android-пристрої, однак завдяки оптимізації його робота є можливою і на старіших версіях системи.

Таблиця нижче наводить рекомендовані та мінімальні технічні характеристики, для різних сценаріїв використання затсосунку, рекомендовані не є необхідними, але забезпечують максимальну продуктивність:

Таблиця 4.1 – Вимоги до пристрою

Вимоги до конфігурування	**Рекомендовано**	**Мінімально**
Операційна система	Android 12.0 або вище	Android 9.0
Процесор	8-ядерний, 2.0 ГГц	4-ядерний, 1.5 ГГц
Оперативна пам’ять	4 ГБ RAM	2 ГБ RAM
Вільний простір	20 МБ	10 МБ

Такі параметри обґрунтовані реаліями мобільного ринку: додаток повинен залишатися легким і доступним, не навантажуючи систему, при цьому працюючи без лагів. Підтримка Android 9.0 забезпечує сумісність із більшістю пристроїв, які ще використовуються на ринку, водночас рекомендовані характеристики гарантують максимальну продуктивність і плавність інтерфейсу, особливо під час хмарної синхронізації чи роботи з інтелектуальними чеклістами.

Після встановлення додатку з Google Play та першого запуску користувач має авторизуватися через Google OAuth, використовуючи свій Gmail-акаунт, для доступу до хмарних функцій, таких як синхронізація та нічний бекап. Після авторизації користувач потрапляє на головний екран зі списком створених нотаток і чеклістів. Адаптивний інтерфейс забезпечує коректне відображення незалежно від діагоналі екрана чи роздільної здатності пристрою. Робота з додатком починається з натискання кнопки "Додати", після чого користувач може створити текстову нотатку, звичайний чекліст або скористатися інтелектуальним створенням чекліста через голосовий ввід.

Створена нотатка чи чекліст одразу зберігається в локальній базі даних (Room) і відображається у загальному списку, а також синхронізується з хмарним сховищем Firebase Firestore за наявності інтернет-з’єднання. Для

чеклісту доступні дії з позначення виконаних пунктів, видалення окремих елементів жестом "swipe to delete", а також можливість редагування назви чи вмісту. Інтелектуальне створення чеклістів дозволяє генерувати списки через голосові команди, наприклад, для покупок чи подорожей, за допомогою OpenAI API. Окрім цього, кожен об’єкт можна додати до "обраного", що полегшує доступ до важливої інформації.

Розділ налаштувань дозволяє персоналізувати додаток. Користувач може змінити тему інтерфейсу, розмір шрифту або мову. Застосування більшості змін відбувається миттєво, однак для оновлення мовного інтерфейсу потрібен перезапуск активності. Цей підхід дозволяє забезпечити просту локалізацію та адаптацію для різних користувачів, включаючи підтримку української та англійської мов.

Оскільки додаток виконує роль інструменту для особистого управління завданнями, акцент зроблено на простоті взаємодії, стабільності та офлайн-доступі. Локальне зберігання даних у Room забезпечує можливість роботи без інтернет-з’єднання, що робить додаток зручним у дорозі, за кордоном чи в зонах із нестабільним сигналом, тоді як хмарна синхронізація через Firebase Firestore гарантує збереження даних і доступ до них із різних пристроїв.

Функціональні можливості охоплюють ключові дії, які користувач очікує від сучасного TODO-додатку: створення, редагування, видалення, фільтрація та кастомізація вигляду. Додатково реалізовано інноваційні функції, такі як інтелектуальне створення чеклістів і щоденний нічний бекап, що підвищують зручність і безпеку. Водночас додаток залишається легким і швидким завдяки оптимізації, а авторизація через Google OAuth забезпечує безпечний доступ до хмарних функцій без ускладнення процесу входу.

4.3 Висновки

У четвертому розділі було проведено тестування основних функціональних модулів мобільного додатку для управління завданнями. Перевірено коректність реалізації ключових можливостей, таких як створення,

редагування, видалення нотаток і чеклістів, а також кастомізація інтерфейсу користувача. Окрему увагу приділено оцінці стабільності роботи у випадках помилкового вводу та реакції додатку на нештатні ситуації.

Проведене тестування підтвердило сумісність додатку з різними версіями операційної системи Android, включаючи 9.0, 10.0 та 11.0. Як фізичні пристрої, так і емулятори засвідчили коректну роботу інтерфейсу та логіки незалежно від платформи, що відповідає вимогам щодо кросверсійної стабільності.

Окрім того, в межах розділу було розроблено інструкцію користувача, яка містить перелік технічних вимог до пристрою, а також опис основних дій для початку роботи з додатком. Завдяки інтуїтивному інтерфейсу на основі Material Design 3 інструкція не потребує деталізації, але забезпечує базову орієнтацію користувача щодо функціональності системи.

ВИСНОВКИ

У бакалаврській кваліфікаційній роботі розроблено методи та програмні засоби для менеджменту особистих завдань в мобільному додатку з підтримкою хмарної синхронізації та інтелектуального створення чеклістів.

Проведено аналіз сучасних підходів до управління завданнями в мобільних додатках, включаючи технології локального та хмарного зберігання даних, а також методи обробки природної мови для автоматизації створення списків завдань. Обґрунтовано необхідність розробки методів інтеграції хмарної синхронізації та інтелектуальних функцій з метою підвищення зручності використання, доступності даних і автоматизації рутинних задач користувача.

Вперше запропоновано метод інтелектуального створення чеклістів на основі звукового вводу з використанням OpenAI API, який застосовує обробку природної мови для автоматичного структурування завдань, враховуючи неформальні фрази, що підвищує зручність і швидкість введення даних для користувача. Наприклад, голосовий ввід "я йду в магазин мені треба купити молоко яйця дві буханки хліба крупи пшеничної і напевно мінералки теж взяти" точно перетворюється на структурований список із п’ятьма пунктами.

Запропоновано метод хмарної синхронізації та бекапу даних через Firebase Firestore, який забезпечує безперебійний доступ до нотаток і чеклістів із різних пристроїв із мінімальними затримками завдяки автоматичному плануванню синхронізації через WorkManager о 2:00 ночі з урахуванням часового поясу та автентифікацією через Google OAuth, що гарантує безпеку даних.

Запропоновано підхід до кастомізації інтерфейсу в мобільному додатку, який інтегрує адаптивні теми, мови та шрифти, оптимізуючи користувацький досвід шляхом швидкої зміни вигляду інтерфейсу без перевантаження системи, наприклад, миттєве застосування теми Purple чи зміна мови на українську після перезапуску активності.

Практичне значення отриманих результатів полягає в тому, що розроблений мобільний застосунок може бути використаний для ефективного управління особистими завданнями в різних сферах, включаючи освіту, бізнес і повсякденне життя.

Розроблено методи інтеграції хмарної синхронізації та інтелектуального створення чеклістів, які дозволяють з високою точністю автоматизувати створення списків завдань і забезпечувати безперебійний доступ до даних користувача на різних пристроях, використовуючи автентифікацію через Google OAuth для захисту інформації.

На основі проведених досліджень розроблено алгоритми та програмні засоби для управління завданнями, включаючи створення інтелектуальних чеклістів, хмарну синхронізацію та офлайн-доступ. Проведене тестування на пристроях Redmi Note 7 і емульованому Google Pixel підтвердило достовірність теоретичних досліджень, зокрема коректність локального завантаження даних, синхронізації між пристроями та обробки голосових команд. Розроблені засоби можна використовувати для оптимізації щоденних задач у різних сценаріях, наприклад, для планування покупок чи управління особистими проєктами.

Ключові слова: мобільний застосунок, управління завданнями, хмарний бекап, інтелектуальне створення чеклістів, обробка звукового вводу, Kotlin, Jetpack Compose, Firebase, архітектура MVVM.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Продуктивність і тайм-менеджмент: методи ефективного керування особистими завданнями. URL: https://studway.com.ua/productivity-management (дата звернення: 26.03.2025).

2. Тарасенко В. Топ застосунків для управління особистими завданнями: огляд популярних рішень. URL: https://dou.ua/task-management-apps-review (дата звернення: 26.03.2025).

3. Психологія продуктивності: як ефективно керувати особистими завданнями. URL: https://upj.com.ua/productivity-psychology (дата звернення: 26.03.2025).

4. Tsymbal M., Romaniuk O. Comparative Analysis of Native and Cross-Platform Development. In Materials of the Youth Scientific and Practical Internet Conference of Students, Postgraduates, and Young Scientists "Youth in Science: Research, Problems, Perspectives (MN-2023)". Vinnytsia: VNTU, 2023. pp. 721-723.

5. Voice Recognition Technology Market Report. Grand View Research. URL: https://www.grandviewresearch.com/industry-analysis/voice-recognition-market (дата звернення: 29.03.2025).

6. Room Database Documentation. Android Developers. URL: https://developer.android.com/reference/androidx/room/Room (дата звернення: 01.04.2025).

7. Material Design 3 Guidelines. URL: https://m3.material.io/ (дата звернення: 02.04.2025).

8. Google Keep. URL: https://keep.google.com (дата звернення: 07.04.2025).

9. Todoist: To-Do List & Task Manager. URL: https://todoist.com (дата звернення: 07.04.2025).

10. Evernote: Note Taking App. URL: https://evernote.com (дата звернення: 07.04.2025).

11. UML Specification. Object Management Group. URL: https://www.omg.org/spec/UML/ (дата звернення: 12.04.2025).

12. Anderson, P., Brown, K. MVVM Architecture Patterns in Android Development. Proceedings of the International Conference on Software Engineering. San Francisco, 2019. pp. 234-241.

13. Bass, L., Clements, P., Kazman, R. Software Architecture in Practice. 4th ed. Boston : Addison-Wesley, 2021. 464 p.

14. Draw.io. URL: https://app.diagrams.net (дата звернення: 17.04.2025).

15. Kotlin. URL: https://developer.android.com/topic/architecture?hl=en (дата звернення: 19.04.2025).

16. Hilt Dependency Injection. Android Developers. URL: https://developer.android.com/training/dependency-injection/hilt-android (дата звернення: 20.04.2025).

17. Firebase Documentation. Cloud Firestore. URL: https://firebase.google.com/docs/firestore (дата звернення: 23.04.2025).

18. Google OAuth 2.0 Documentation URL: https://developers.google.com/identity/protocols/oauth2 (дата звернення: 24.04.2024).

19. Jetpack Compose Documentation. URL: https://developer.android.com/jetpack/compose (дата звернення: 27.04.2024).

20. Norman D. The Design of Everyday Things: Revised and Expanded Edition. New York : Basic Books, 2013. 368 p.

21. Tidwell J., Brewer C., Valencia A. Designing Interfaces: Patterns for Effective Interaction Design. 3rd ed. Sebastopol : O'Reilly Media, 2020. 600 p.

22. QATestLab. URL: https://training.qatestlab.com/blog/technical-articles/testing-mobile-devices/ (дата звернення: 02.05.2025).

23. Wezom URL: https://wezom.com.ua/ua/blog/chek-list-testirovanija-mobilnogo-prilozhenija (дата звернення: 03.05.2025).

ДОДАТКИ

77

Додаток А – Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
д.т.н., професор
_____ О. Н. Романюк
«21» березня 2025 р.

Технічне завдання
на бакалаврську кваліфікаційну роботу «Розробка мобільного застосунку для організації та менеджменту особистих завдань» за спеціальністю 121 – Інженерія програмного забезпечення

Керівник бакалаврської кваліфікаційної роботи:
_____ к.т.н., доцент. Г.О. Черноволик
«21» березня 2025 р.

Виконав:
_____ студент гр.5ПІ-21б М.Ю. Цимбал
«21» березня 2025 р.

Вінниця – 2025 року

1. Найменування та галузь застосування

Бакалаврська кваліфікаційна робота: «Розробка мобільного застосунку для організації та менеджменту особистих завдань».

Галузь застосування – програмне забезпечення для управління завданнями.

2. Підстава для розробки.

Підставою для виконання бакалаврської кваліфікаційної роботи (БКР) є індивідуальне завдання на БКР та наказ №97 від 20 березня 2025 року ректора по ВНТУ про закріплення тем БКР.

3. Мета та призначення розробки.

Метою дослідження є підвищення продуктивності та зручності користувацького досвіду за рахунок розробки мобільного застосунку, який дозволяє створення та редагування нотаток і чеклістів, інтелектуального створення чеклістів на основі текстового та звукового вводу, а також хмарного бекапу та синхронізації.

Призначення роботи – створення мобільного програмного забезпечення, що дозволяє користувачам формувати, редагувати, планувати та відстежувати виконання особистих завдань з можливістю резервного копіювання.

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись БКР:

1. Tsymbal M., Romaniuk O. Comparative Analysis of Native and Cross-Platform Development. In Materials of the Youth Scientific and Practical Internet Conference of Students, Postgraduates, and Young Scientists "Youth in Science: Research, Problems, Perspectives (MN-2023)". Vinnytsia: VNTU, 2023. pp. 721-723.

2. Norman D. The Design of Everyday Things: Revised and Expanded Edition. New York : Basic Books, 2013. 368 p.

5. Технічні вимоги

Для коректної роботи мобільного застосунку рекомендується використовувати пристрої з операційною системою Android 12.0 або вище, 8-ядерним процесором з тактовою частотою не менше 2.0 ГГц, 4 ГБ оперативної пам’яті та не менше 20 МБ вільного простору у внутрішньому сховищі. Мінімальні вимоги для запуску програми: Android 9.0, 4-ядерний процесор (1.5 ГГц), 2 ГБ RAM і 10 МБ вільної пам’яті.

6. Конструктивні вимоги.

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред’являється по закінченню робіт:

- пояснювальна записка до БКР;
- технічне завдання;
- лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

9. Стадії та етапи розробки:

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи
1	Аналіз підходів для систем менеджменту особистих завдань	25.03.25-03.04.25
2	Розробка архітектури мобільного застосунку	03.04.25-13.04.25
3	Розробка алгоритмів роботи програмного забезпечення	13.04.25-20.04.25

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи
4	Розробка мобільного програмного забезпечення	20.04.25-10.05.25
5	Тестування програмного забезпечення	10.05.25-22.05.25
6	Оформлення матеріалів до захисту БКР	22.05.25-30.05.25

10. Порядок контролю та прийняття.

Виконання етапів бакалаврської кваліфікаційної роботи контролюється керівником згідно з графіком виконання роботи. Захист бакалаврської кваліфікаційної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком

Додаток Б – Протокол перевірки на плагіат

81

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Розробка мобільного застосунку для організації та менеджменту особистих завдань

Тип роботи: бакалаврська кваліфікаційна робота
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ кафедра програмного забезпечення, ФІТКІ, 5ПІ-21б
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 3,4 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

■ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

☐ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

_____ _____
(прізвище, ініціали, посада) (підпис)

_____ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Черноволик Г. О.

З висновком експертної комісії ознайомлений(-на)

Керівник _____ Черноволик Г. О. к.т.н., доц. каф. ПЗ
(підпис)

Здобувач _____ Цимбал М. Ю.
(підпис)

Додаток В – Лістинг програмного забезпечення

```
package dev.mynoteapp.ui

import android.content.Context
import android.content.res.Configuration
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.material3.Surface
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.core.view.WindowCompat
import androidx.hilt.navigation.compose.hiltViewModel
import dagger.hilt.android.AndroidEntryPoint
import dev.mynoteapp.ui.navigation.NavApp
import dev.mynoteapp.ui.screens.settings.SettingsViewModel
import dev.mynoteapp.ui.theme.FontType
import dev.mynoteapp.ui.theme.MyNotePPTheme
import java.util.Locale

@AndroidEntryPoint
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        WindowCompat.setDecorFitsSystemWindows(window, false)
        enableEdgeToEdge()
        setContent {
            val viewModel: SettingsViewModel = hiltViewModel()
            val darkTheme by viewModel.darkMode.collectAsState()
            val themeIndex by viewModel.themeIndex.collectAsState()
            val language by viewModel.language.collectAsState()
            val font by viewModel.font.collectAsState()
            updateAppLocale(this, language)
            MyNotePPTheme(
                darkTheme = darkTheme,
                selectedThemeIndex = themeIndex,
                selectedFont = FontType.entries.firstOrNull { it.fontName == font } ?:
FontType.OPEN_SANS
            ) {
                Surface { NavApp() }
            }
        }
    }
}

fun updateAppLocale(context: Context, languageCode: String) {
    val locale = Locale(languageCode)
    Locale.setDefault(locale)

    val config = Configuration(context.resources.configuration)
```

```
    config.setLocale(locale)

    context.resources.updateConfiguration(config, context.resources.displayMetrics)
}

package dev.mynoteapp.ui.screens.home

import androidx.compose.animation.Crossfade
import androidx.compose.animation.animateContentSize
import androidx.compose.animation.core.animateFloatAsState
import androidx.compose.animation.core.tween
import androidx.compose.foundation.ExperimentalFoundationApi
import androidx.compose.foundation.combinedClickable
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.aspectRatio
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.grid.GridCells
import androidx.compose.foundation.lazy.grid.LazyVerticalGrid
import androidx.compose.foundation.lazy.grid.items
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Add
import androidx.compose.material.icons.filled.CheckCircle
import androidx.compose.material.icons.filled.Close
import androidx.compose.material.icons.filled.Delete
import androidx.compose.material.icons.filled.Edit
import androidx.compose.material.icons.filled.Favorite
import androidx.compose.material.icons.filled.FavoriteBorder
import androidx.compose.material.icons.filled.Settings
import androidx.compose.material3.Card
import androidx.compose.material3.CardDefaults
import androidx.compose.material3.CenterAlignedTopAppBar
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.FabPosition
import androidx.compose.material3.FloatingActionButton
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.NavigationBar
import androidx.compose.material3.NavigationBarItem
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.material3.TopAppBarDefaults
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.graphicsLayer
import androidx.compose.ui.res.stringResource
```

```
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import dev.mynoteapp.R
import dev.mynoteapp.model.BaseItem
import dev.mynoteapp.ui.navigation.ChecklistScreenRoute
import dev.mynoteapp.ui.navigation.NoteScreenRoute
import dev.mynoteapp.ui.navigation.SettingsScreenRoute
import dev.mynoteapp.ui.screens.dialogs.InputDialog


@Composable
fun HomeScreen() {
    val navController = dev.mynoteapp.ui.navigation.LocalNavController.current
    HomeContent(
        onLaunchChecklistScreen = { navController.navigate(ChecklistScreenRoute(it)) },
        onLaunchNoteScreen = { navController.navigate(NoteScreenRoute(it)) },
        onLaunchSettingsScreen = {navController.navigate(SettingsScreenRoute)}
    )
}

@Composable
fun HomeContent(
    onLaunchChecklistScreen: (checklistId: String) -> Unit,
    onLaunchNoteScreen: (noteId: String) -> Unit,
    onLaunchSettingsScreen: () -> Unit
) {
    val viewModel: MainScreenViewModel = hiltViewModel()
    val uiState = viewModel.uiState.collectAsStateWithLifecycle()
    val notesScreen = viewModel.notes.collectAsStateWithLifecycle().value
    val checklistsScreen = viewModel.checklists.collectAsStateWithLifecycle().value
    val isShowOnlyFavourite = viewModel.showOnlyFavorites.collectAsStateWithLifecycle().value
    var isNewItemAdding by remember { mutableStateOf(false) }
    var isNavigating by remember { mutableStateOf(false) }

    var isMultiSelectMode by remember { mutableStateOf(false) }
    var selectedItems by remember { mutableStateOf<String>()) }

    if (isNewItemAdding) {
        InputDialog(
            title = stringResource(R.string.input_title_for_new_item),
            previousText = stringResource(R.string.new_item),
            onDismiss = { isNewItemAdding = false },
            onConfirm = { itemTitle ->
                viewModel.addNewItem(itemTitle)
                isNewItemAdding = false
            }
        )
    }

    Crossfade(
```

```
    targetState = uiState.value.screenState,
    label = stringResource(R.string.screenchange),
    animationSpec = tween(durationMillis = 500)
) { screen ->
    val items: List<BaseItem> = when (screen) {
        ScreenContent.Notes -> notesScreen
        ScreenContent.Checklists -> checklistsScreen
    }

    Scaffold(
        topBar = {
            HomeTopBar(
                isMultiSelectMode = isMultiSelectMode,
                selectedItemsCount = selectedItems.size,
                onNavigate = {
                    isNavigating = true
                },
                onExitMultiSelect = {
                    isMultiSelectMode = false
                    selectedItems = emptySet()
                },
                onMarkFavorites = {
                    viewModel.markItemsAsFavorite(selectedItems)
                    isMultiSelectMode = false
                    selectedItems = emptySet()
                },
                onDeleteSelected = {
                    viewModel.deleteItems(selectedItems)
                    isMultiSelectMode = false
                    selectedItems = emptySet()
                },
                isShowOnlyFavorite = isShowOnlyFavourite ,
                toggleShowOnlyFavorites = { viewModel.toggleShowOnlyFavorites() },
                onSettingsClick = {onLaunchSettingsScreen()},
                uiState = uiState.value
            )
        },
        bottomBar = {
            if (!isMultiSelectMode) {
                NavigationBar {
                    NavigationBarItem(
                        selected = uiState.value.screenState == ScreenContent.Checklists,
                        label = { Text(stringResource(R.string.checklists)) },
                        onClick = { viewModel.toggleScreenState() },
                        icon = {
                            Icon(
                                imageVector = Icons.Default.CheckCircle,
                                contentDescription = stringResource(R.string.checklists),
                            )
                        }
                    )
                    NavigationBarItem(
                        selected = uiState.value.screenState == ScreenContent.Notes,
                        label = { Text(stringResource(R.string.notes)) },
```

```
                onClick = { viewModel.toggleScreenState() },
                icon = {
                    Icon(
                        imageVector = Icons.Default.Edit,
                        contentDescription = stringResource(R.string.notes)
                    )
                }
            )
        }
    }
},
floatingActionButton = {
    if (!isMultiSelectMode) {
        FloatingActionButton(onClick = {
            isNewItemAdding = true
        }) {
            Icon(
                imageVector = Icons.Default.Add,
                contentDescription = stringResource(R.string.add)
            )
        }
    }
},
floatingActionButtonPosition = FabPosition.End
) { paddingValues ->
    LazyVerticalGrid(
        columns = GridCells.Fixed(2),
        horizontalArrangement = Arrangement.spacedBy(16.dp),
        verticalArrangement = Arrangement.spacedBy(16.dp),
        modifier = Modifier
            .padding(paddingValues)
            .padding(16.dp)
    ) {
        items(items) { item ->
            val isSelected = selectedItems.contains(item.id)
            GridItem(
                item = item,
                isSelected = isSelected,
                isMultiSelectMode = isMultiSelectMode,
                onItemSelect = {
                    selectedItems = if (isSelected) selectedItems - it.id else selectedItems + it.id
                    if (selectedItems.isEmpty()) isMultiSelectMode = false
                },
                onItemClick = {
                    if (!isMultiSelectMode) {
                        isNavigating = true
                        if (uiState.value.screenState == ScreenContent.Checklists) {
                            onLaunchChecklistScreen(item.id)
                        } else {
                            onLaunchNoteScreen(item.id)
                        }
                    }
                },
                onItemLongClick = {
```

```
                    if (!isMultiSelectMode) {
                        isMultiSelectMode = true
                        selectedItems = selectedItems + it.id
                    }
                },

                onToggleFavorite = { viewModel.toggleFavourite(it) },
            )
        }
    }
  }
}

@OptIn(ExperimentalFoundationApi::class)
@Composable
fun GridItem(
    item: BaseItem,
    isSelected: Boolean,
    isMultiSelectMode: Boolean,
    onToggleFavorite: (BaseItem) -> Unit,
    onItemSelect: (BaseItem) -> Unit,
    onItemClick: (BaseItem) -> Unit,
    onItemLongClick: (BaseItem) -> Unit
) {
    val scale by animateFloatAsState(if (isSelected) 0.9f else 1f, label =
stringResource(R.string.scale_animation))
    val backgroundColor = if (isSelected)
MaterialTheme.colorScheme.secondaryContainer.copy(alpha = 0.5f) else
MaterialTheme.colorScheme.secondaryContainer

    Card(
        modifier = Modifier
            .fillMaxWidth()
            .aspectRatio(1f)
            .graphicsLayer(scaleX = scale, scaleY = scale)
            .combinedClickable(
                onClick = {
                    if (isMultiSelectMode) {
                        onItemSelect(item)
                    } else {
                        onItemClick(item)
                    }
                },
                onLongClick = { onItemLongClick(item) }
            )
            .animateContentSize(),
        colors = CardDefaults.cardColors(containerColor = backgroundColor)
    ) {
        Box(
            modifier = Modifier.fillMaxSize()
        ) {
            IconButton(
                onClick = { onToggleFavorite(item) },
```

```
            modifier = Modifier.align(Alignment.TopStart)
        ) {
            Icon(
                imageVector = if (item.isFavorite) Icons.Default.Favorite else
Icons.Default.FavoriteBorder,
                contentDescription = stringResource(R.string.favorite)
            )
        }

        Text(
            text = item.title,
            textAlign = TextAlign.Center,
            modifier = Modifier
                .fillMaxWidth()
                .align(Alignment.Center),
            maxLines = 3,
            overflow = TextOverflow.Ellipsis
        )
    }
  }
}

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun HomeTopBar(
    isMultiSelectMode: Boolean,
    selectedItemsCount: Int,
    isShowOnlyFavorite: Boolean,
    onNavigate: () -> Unit,
    onExitMultiSelect: () -> Unit,
    onMarkFavorites: () -> Unit,
    onDeleteSelected: () -> Unit,
    onSettingsClick: () -> Unit,
    toggleShowOnlyFavorites: () -> Unit,
    uiState: StateUI
) {
    CenterAlignedTopAppBar(
        title = {
            Text(
                text = if (isMultiSelectMode) "$selectedItemsCount selected" else {
                    when(uiState.screenState){
                        ScreenContent.Checklists -> stringResource(R.string.checklists)
                        ScreenContent.Notes -> stringResource(R.string.notes)
                    }
                },
                fontWeight = FontWeight.Bold
            )
        },
        colors = TopAppBarDefaults.largeTopAppBarColors(
            containerColor = MaterialTheme.colorScheme.secondaryContainer,
            titleContentColor = MaterialTheme.colorScheme.onSecondaryContainer
        ),
        navigationIcon = {
            if (isMultiSelectMode) {
```

```
            IconButton(onClick = onExitMultiSelect) {
                Icon(imageVector = Icons.Default.Close, contentDescription =
stringResource(R.string.exit_selection))
            }
        } else {
            IconButton(onClick = {onSettingsClick()}) {
                Icon(
                    imageVector = Icons.Default.Settings,
                    contentDescription = stringResource(R.string.settings) ,
                    tint = MaterialTheme.colorScheme.onSecondaryContainer
                )
            }
        }
    },
    actions = {
        if (isMultiSelectMode) {
            IconButton(onClick = onMarkFavorites) {
                Icon(imageVector = Icons.Default.Favorite, contentDescription =
stringResource(R.string.mark_as_favorite))
            }
            IconButton(onClick = onDeleteSelected) {
                Icon(imageVector = Icons.Default.Delete, contentDescription =
stringResource(R.string.delete_selected))
            }
        } else {
            IconButton(onClick = { toggleShowOnlyFavorites() }) {
                Icon(
                    imageVector = if (isShowOnlyFavorite) Icons.Default.Favorite else
Icons.Default.FavoriteBorder,
                    contentDescription = stringResource(R.string.favorite)
                )
            }
        }
    )
}

package dev.mynoteapp.ui.screens.note

import android.widget.Toast
import androidx.compose.foundation.gestures.detectTapGestures
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.WindowInsets
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.ime
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.verticalScroll
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ArrowBack
import androidx.compose.material.icons.filled.Check
import androidx.compose.material.icons.filled.Delete
```

```
import androidx.compose.material.icons.filled.Edit
import androidx.compose.material.icons.filled.Favorite
import androidx.compose.material.icons.filled.FavoriteBorder
import androidx.compose.material3.CenterAlignedTopAppBar
import androidx.compose.material3.CircularProgressIndicator
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.FloatingActionButton
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.material3.TextField
import androidx.compose.material3.TextFieldDefaults
import androidx.compose.material3.TopAppBarDefaults
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.saveable.rememberSaveable
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.focus.FocusRequester
import androidx.compose.ui.focus.focusRequester
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.input.pointer.pointerInput
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.platform.LocalSoftwareKeyboardController
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.TextRange
import androidx.compose.ui.text.input.TextFieldValue
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import dev.mynoteapp.R
import dev.mynoteapp.ui.navigation.LocalNavController
import dev.mynoteapp.ui.screens.dialogs.InputDialog

@Composable
fun NoteScreen(noteId: String) {
    val navController = LocalNavController.current
    var isNavigating by remember { mutableStateOf(false) }
    NoteContent(
        noteId
    ) {
            if (!isNavigating && navController.previousBackStackEntry != null) {
                isNavigating = true
                navController.popBackStack()
            }
    }
```

```
}

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun NoteContent(
    noteId: String,
    onBackScreen: () -> Unit
) {
    val viewModel: NoteViewModel = hiltViewModel()
    val note = viewModel.notes.collectAsStateWithLifecycle().value.firstOrNull { it.id == noteId }
    var isTitleEditing by rememberSaveable { mutableStateOf(false) }
    var isContentEditing by rememberSaveable { mutableStateOf(false) }
    val context = LocalContext.current
    val keyboardController = LocalSoftwareKeyboardController.current
    val focusRequester = remember { FocusRequester() }
    if (note == null) {
        Box(
            modifier = Modifier.fillMaxSize(),
            contentAlignment = Alignment.Center
        ) {
            CircularProgressIndicator()
        }
    } else {
        var textState by remember { mutableStateOf(TextFieldValue(note.content)) }
        if (isTitleEditing) {
            InputDialog(
                title = stringResource(R.string.input_new_note_title),
                previousText = note.title,
                onDismiss = { isTitleEditing = false },
                onConfirm = { taskText ->
                    viewModel.editTitle(noteId, taskText)
                    isTitleEditing = false
                }
            )
        }

        LaunchedEffect(isContentEditing) {
            if (isContentEditing) {
                focusRequester.requestFocus()
                keyboardController?.show()
                textState = textState.copy(selection = TextRange(textState.text.length))
            } else {
                keyboardController?.hide()
            }
        }

        Scaffold(
            topBar = {
                CenterAlignedTopAppBar(
                    navigationIcon = {
                        IconButton(onClick = {
                            onBackScreen()
                        }) {
                            Icon(
```

```
                        imageVector = Icons.AutoMirrored.Filled.ArrowBack,
                        contentDescription = stringResource(R.string.back)
                    )
                }
            },
            title = {
                Text(
                    text = note.title,
                    maxLines = 1,
                    overflow = TextOverflow.Ellipsis,
                    modifier = Modifier
                        .fillMaxWidth()
                        .pointerInput(Unit) {
                            detectTapGestures(
                                onLongPress = { isTitleEditing = true }
                            )
                        },
                    style = MaterialTheme.typography.titleLarge,
                    textAlign = TextAlign.Center
                )
            },
            colors = TopAppBarDefaults.largeTopAppBarColors(
                containerColor = MaterialTheme.colorScheme.secondaryContainer,
                titleContentColor = MaterialTheme.colorScheme.onSecondaryContainer
            ),
            actions = {
                IconButton(onClick = { viewModel.toggleFavorite(noteId) }) {
                    Icon(
                        imageVector = if (note.isFavorite) Icons.Default.Favorite else
Icons.Default.FavoriteBorder,
                        contentDescription = stringResource(R.string.favorite)
                    )
                }
                IconButton(onClick = {
                    viewModel.deleteNote(noteId)
                    Toast.makeText(context,
                        context.getString(R.string.note_deleted), Toast.LENGTH_SHORT).show()
                    onBackScreen()
                }) {
                    Icon(
                        imageVector = Icons.Default.Delete,
                        contentDescription = stringResource(R.string.delete)
                    )
                }
            }
        )
    },
    floatingActionButton = {
        FloatingActionButton(
            onClick = {
                if (isContentEditing) {
                    viewModel.editContent(noteId, textState.text)
                }
                isContentEditing = !isContentEditing
```

```
                },
            ) {
                Icon(
                    imageVector = if (isContentEditing) Icons.Default.Check else Icons.Default.Edit,
                    contentDescription = if (isContentEditing) stringResource(R.string.save_note) else
stringResource(
                        R.string.edit_note
                    )
                )
            }
        },
        contentWindowInsets = WindowInsets.ime,
    ) { padding ->
        Box(
            modifier = Modifier
                .fillMaxSize()
                .padding(padding)
        ) {
            if (isContentEditing) {
                TextField(
                    value = textState,
                    onValueChange = { newValue -> textState = newValue },
                    modifier = Modifier
                        .fillMaxSize()
                        .padding(8.dp)
                        .focusRequester(focusRequester)
                        .verticalScroll(rememberScrollState()),
                    colors = TextFieldDefaults.colors(
                        focusedIndicatorColor = Color.Transparent,
                        unfocusedIndicatorColor = Color.Transparent
                    ),
                )
            } else {
                Box(
                    modifier = Modifier
                        .fillMaxSize()
                        .padding(16.dp)
                        .focusRequester(focusRequester)
                        .pointerInput(Unit) {
                            detectTapGestures(
                                onLongPress = { isContentEditing = true }
                            )
                        }
                ) {
                    Column(
                        modifier = Modifier
                            .fillMaxSize()
                            .verticalScroll(rememberScrollState())
                    ) {
                        Text(
                            text = note.content,
                            modifier = Modifier.fillMaxWidth(),
                            style = MaterialTheme.typography.bodyLarge
                        )
```

```
                }
            }

        }
    }
}
}
```

```
package dev.mynoteapp.ui.screens.home

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import dagger.hilt.android.lifecycle.HiltViewModel
import dev.mynoteapp.data.checklist.ChecklistRepository
import dev.mynoteapp.data.note.NoteRepository
import dev.mynoteapp.model.BaseItem
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.combine
import kotlinx.coroutines.flow.stateIn
import kotlinx.coroutines.flow.update
import kotlinx.coroutines.launch
import javax.inject.Inject

enum class ScreenContent {
    Checklists, Notes
}

data class StateUI(
    val isLoading: Boolean = false,
    val errorMessage: String? = null,
    val screenState: ScreenContent = ScreenContent.Checklists
)

@HiltViewModel
class MainScreenViewModel @Inject constructor(
    private val noteRepository: NoteRepository,
    private val checklistRepository: ChecklistRepository
) : ViewModel() {

    private val _uiState = MutableStateFlow(
        StateUI()
    )
    val uiState: StateFlow<StateUI> = _uiState.asStateFlow()

    private val _showOnlyFavorites = MutableStateFlow(false)
    val showOnlyFavorites: StateFlow<Boolean> = _showOnlyFavorites.asStateFlow()

    val checklists = checklistRepository.observeChecklists()
        .combine(showOnlyFavorites) { checklists, showOnlyFavorites ->
            if (showOnlyFavorites) checklists.filter { it.isFavorite } else checklists
```

```
    }
    .stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), emptyList())

val notes = noteRepository.observeNotes()
    .combine(showOnlyFavorites) { notes, showOnlyFavorites ->
        if (showOnlyFavorites) notes.filter { it.isFavorite } else notes
    }
    .stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), emptyList())

fun toggleShowOnlyFavorites() {
    _showOnlyFavorites.update { !it }
}

fun toggleFavourite(item: BaseItem) {
    viewModelScope.launch {
        when (uiState.value.screenState) {
            ScreenContent.Notes -> noteRepository.toggleFavorite(item.id)
            ScreenContent.Checklists -> checklistRepository.toggleFavorite(item.id)
        }
    }
}

fun addNewItem(title: String) {
    viewModelScope.launch {
        when (uiState.value.screenState) {
            ScreenContent.Notes -> noteRepository.save(title)
            ScreenContent.Checklists -> checklistRepository.save(title)
        }
    }
}

fun deleteItem(item: BaseItem) {
    viewModelScope.launch {
        when (uiState.value.screenState) {
            ScreenContent.Notes -> noteRepository.delete(item.id)
            ScreenContent.Checklists -> checklistRepository.delete(item.id)
        }
    }
}

fun toggleScreenState() {
    viewModelScope.launch {
        _uiState.update { state ->
            when (uiState.value.screenState) {
                ScreenContent.Checklists -> {
                    state.copy(screenState = ScreenContent.Notes)
                }

                ScreenContent.Notes -> {
                    state.copy(screenState = ScreenContent.Checklists)
                }
            }
        }
    }
```

```
    }

    fun markItemsAsFavorite(selectedItems: Set<String>) {
        viewModelScope.launch {
            when (uiState.value.screenState) {
                ScreenContent.Notes -> noteRepository.toggleFavorite(selectedItems)
                ScreenContent.Checklists -> checklistRepository.toggleFavorite(selectedItems)
            }
        }
    }

    fun deleteItems(selectedItems: Set<String>) {
        viewModelScope.launch {
            when (uiState.value.screenState) {
                ScreenContent.Notes -> noteRepository.delete(selectedItems)
                ScreenContent.Checklists -> checklistRepository.delete(selectedItems)
            }
        }
    }
}

package dev.mynoteapp.data.checklist.impl

import dev.mynoteapp.data.checklist.ChecklistDao
import dev.mynoteapp.data.checklist.ChecklistRepository
import dev.mynoteapp.data.local.toEntity
import dev.mynoteapp.data.local.toModel
import dev.mynoteapp.model.Checklist
import dev.mynoteapp.model.Task
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.map
import javax.inject.Inject
import javax.inject.Singleton

@Singleton
class ChecklistRepositoryImpl @Inject constructor(private val checklistDao: ChecklistDao) :
    ChecklistRepository {

    override fun observeChecklists(): Flow<List<Checklist>> {
        return checklistDao.observeChecklists().map { notes ->
            notes.map { it.toModel() }
        }
    }

    override suspend fun getAll(): List<Checklist> {
        return checklistDao.getAllChecklistsWithTasks().map { it.toModel() }
    }

    override suspend fun getById(id: String): Checklist {
        return checklistDao.getChecklistWithTasksById(id).toModel()
    }

    override suspend fun save(item: Checklist) {
        checklistDao.updateChecklist(item.toEntity())
```

```
}

override suspend fun save(title: String) {
    checklistDao.insertChecklist(Checklist(title = title).toEntity())
}

override suspend fun delete(id: String) {
    checklistDao.deleteChecklistById(id)
}

override suspend fun toggleFavorite(id: String) {
    val checklist = checklistDao.getChecklistById(id)
    checklist.let {
        checklistDao.updateChecklist(it.copy(isFavorite = !it.isFavorite))
    }
}

override suspend fun deleteTask(checklistId: String, taskId: String) {
    checklistDao.deleteTaskFromChecklist(checklistId, taskId)
}


override suspend fun addTask(checklistId: String, title: String) {
    checklistDao.insertTask(
        Task(title = title).toEntity(
        checklistId = checklistId
    ))
}

override suspend fun toggleTaskState(checklistId: String, taskId: String, state: Boolean) {
    val task = checklistDao.getTaskById(checklistId, taskId)
    task.let {
        checklistDao.updateTask(it.copy(isDone = state))
    }
}

override suspend fun editChecklistTitle(checklistId: String, newTitle: String) {
    val checklist = checklistDao.getChecklistById(checklistId)
    checklist.let {
        checklistDao.updateChecklist(it.copy(title = newTitle))
    }
}

override suspend fun toggleFavorite(selectedItems: Set<String>) {
    selectedItems.forEach { itemId ->
        val note = checklistDao.getChecklistById(itemId)
        note.let {
            checklistDao.updateChecklist(it.copy(isFavorite = !it.isFavorite))
        }
    }
}

override suspend fun delete(selectedItems: Set<String>) {
    selectedItems.forEach { itemId ->
```

```
            checklistDao.deleteChecklistById(itemId)
        }
    }
}

package dev.mynoteapp.data.note.impl

import dev.mynoteapp.data.local.toEntity
import dev.mynoteapp.data.local.toModel
import dev.mynoteapp.data.note.NoteDao
import dev.mynoteapp.data.note.NoteRepository
import dev.mynoteapp.model.Note
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.map
import javax.inject.Inject
import javax.inject.Singleton

@Singleton
class NoteRepositoryImpl @Inject constructor(private val noteDao: NoteDao) : NoteRepository {

    override fun observeNotes(): Flow<List<Note>> {
        return noteDao.observeNotes().map { notes ->
            notes.map { it.toModel() }
        }
    }

    override suspend fun getAll(): List<Note> {
        return noteDao.getAllNotes().map { it.toModel() }
    }

    override suspend fun getById(id: String): Note {
        return noteDao.getNoteById(id).toModel()
    }

    override suspend fun save(item: Note) {
        noteDao.updateNote(item.toEntity())
    }

    override suspend fun save(title: String) {
        noteDao.insertNote(Note(title = title).toEntity())
    }

    override suspend fun delete(id: String) {
        noteDao.deleteNoteById(id)
    }

    override suspend fun toggleFavorite(id: String) {
        val note = noteDao.getNoteById(id)
        note.let {
            noteDao.updateNote(it.copy(isFavorite = !it.isFavorite))
        }
    }

    override suspend fun editNoteContent(noteId: String, newContent: String) {
```

```
        val note = noteDao.getNoteById(noteId)
        note.let {
            noteDao.updateNote(it.copy(content = newContent))
        }
    }

    override suspend fun editNoteTitle(noteId: String, newTitle: String) {
        val note = noteDao.getNoteById(noteId)
        note.let {
            noteDao.updateNote(it.copy(title = newTitle))
        }
    }

    override suspend fun toggleFavorite(selectedItems: Set<String>) {
        selectedItems.forEach { itemId ->
            val note = noteDao.getNoteById(itemId)
            note.let {
                noteDao.updateNote(it.copy(isFavorite = !it.isFavorite))
            }
        }
    }

    override suspend fun delete(selectedItems: Set<String>) {
        selectedItems.forEach { itemId ->
          noteDao.deleteNoteById(itemId)
        }
    }
}
```

```
package dev.mynoteapp.data.local

import androidx.room.Database
import androidx.room.RoomDatabase
import dev.mynoteapp.data.checklist.ChecklistDao
import dev.mynoteapp.data.checklist.local.ChecklistEntity
import dev.mynoteapp.data.checklist.local.TaskEntity
import dev.mynoteapp.data.note.NoteDao
import dev.mynoteapp.data.note.local.NoteEntity


@Database(entities = [ChecklistEntity::class, TaskEntity::class,  NoteEntity::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun checklistDao(): ChecklistDao
    abstract fun noteDao(): NoteDao
}
```

```
package dev.mynoteapp.data.local

import dev.mynoteapp.data.checklist.local.ChecklistEntity
import dev.mynoteapp.data.checklist.local.ChecklistWithTasks
import dev.mynoteapp.data.checklist.local.TaskEntity
import dev.mynoteapp.data.note.local.NoteEntity
import dev.mynoteapp.model.Checklist
import dev.mynoteapp.model.Note
```

```
import dev.mynoteapp.model.Task


fun ChecklistWithTasks.toModel(): Checklist {
    return Checklist(
        id = checklist.id,
        title = checklist.title,
        isFavorite = checklist.isFavorite,
        createdAt = checklist.createdAt,
        tasks = tasks.map { it.toModel() }
    )
}

fun ChecklistEntity.toModel(tasks: List<TaskEntity>): Checklist {
    return Checklist(
        id = this.id,
        title = this.title,
        isFavorite = this.isFavorite,
        createdAt = this.createdAt,
        tasks = tasks.map { it.toModel() }
    )
}

fun Checklist.toEntity(): ChecklistEntity {
    return ChecklistEntity(
        id = this.id,
        title = this.title,
        isFavorite = this.isFavorite,
        createdAt = this.createdAt
    )
}

fun TaskEntity.toModel(): Task {
    return Task(
        id = this.id,
        title = this.title,
        isDone = this.isDone
    )
}

fun Task.toEntity(checklistId: String): TaskEntity {
    return TaskEntity(
        id = this.id,
        checklistId = checklistId,
        title = this.title,
        isDone = this.isDone,
    )
}

fun NoteEntity.toModel(): Note {
    return Note(
        id = this.id,
        title = this.title,
        content = this.content,
```

```
        isFavorite = this.isFavorite,
        createdAt = this.createdAt
    )
}

fun Note.toEntity(): NoteEntity {
    return NoteEntity(
        id = this.id,
        title = this.title,
        content = this.content,
        isFavorite = this.isFavorite,
        createdAt = this.createdAt
    )
}

package dev.mynoteapp.di

import android.content.Context
import androidx.room.Room
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.android.qualifiers.ApplicationContext
import dagger.hilt.components.SingletonComponent
import dev.mynoteapp.data.checklist.ChecklistDao
import dev.mynoteapp.data.local.AppDatabase
import dev.mynoteapp.data.note.NoteDao
import javax.inject.Singleton

@Module
@InstallIn(SingletonComponent::class)
object DatabaseModule {

    @Provides
    @Singleton
    fun provideAppDatabase(@ApplicationContext context: Context): AppDatabase {
        return Room.databaseBuilder(
            context.applicationContext,
            AppDatabase::class.java,
            "app_database"
        ).build()
    }

    @Provides
    @Singleton
    fun provideChecklistDao(appDatabase: AppDatabase): ChecklistDao {
        return appDatabase.checklistDao()
    }

    @Provides
    @Singleton
    fun provideNoteDao(appDatabase: AppDatabase): NoteDao {
        return appDatabase.noteDao()
```

```
    }
}

package dev.mynoteapp.ui.navigation

import androidx.compose.animation.fadeIn
import androidx.compose.animation.fadeOut
import androidx.compose.animation.scaleIn
import androidx.compose.animation.scaleOut
import androidx.compose.runtime.Composable
import androidx.compose.runtime.CompositionLocalProvider
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import androidx.navigation.compose.rememberNavController
import androidx.navigation.toRoute
import dev.mynoteapp.ui.screens.checklist.ChecklistScreen
import dev.mynoteapp.ui.screens.home.HomeScreen
import dev.mynoteapp.ui.screens.note.NoteScreen
import dev.mynoteapp.ui.screens.settings.SettingsScreen

@Composable
fun NavApp() {
    val navController = rememberNavController()
    CompositionLocalProvider(dev.mynoteapp.ui.navigation.LocalNavController provides
navController) {
        NavHost(
            navController = navController,
            startDestination =  HomeScreenRoute,
            enterTransition = {
                scaleIn(initialScale = 0.8f) + fadeIn()
            },
            exitTransition = {
                scaleOut(targetScale = 1.2f) + fadeOut()
            },
            popEnterTransition = {
                scaleIn(initialScale = 1.2f) + fadeIn()
            },
            popExitTransition = {
                scaleOut(targetScale = 0.8f) + fadeOut()
            }
        ) {
            composable<HomeScreenRoute> { HomeScreen() }
            composable<SettingsScreenRoute> { SettingsScreen() }
            composable<NoteScreenRoute> { entry ->
                val route: ChecklistScreenRoute = entry.toRoute()
                NoteScreen(noteId = route.id)
            }
            composable<ChecklistScreenRoute> { entry ->
                val route: ChecklistScreenRoute = entry.toRoute()
                ChecklistScreen(checklistId = route.id)
            }
        }
    }
}
```

Додаток Г – Ілюстративна частина

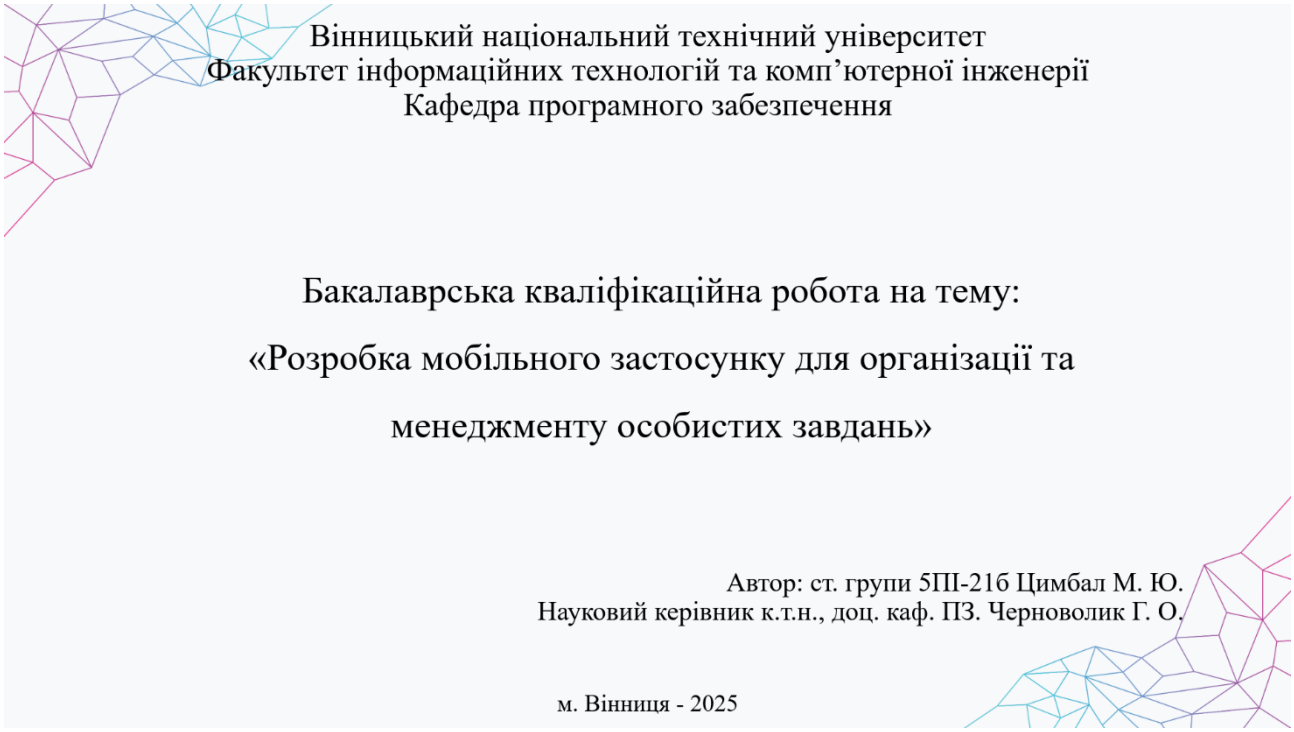


Рисунок Г.1 – Титульний слайд

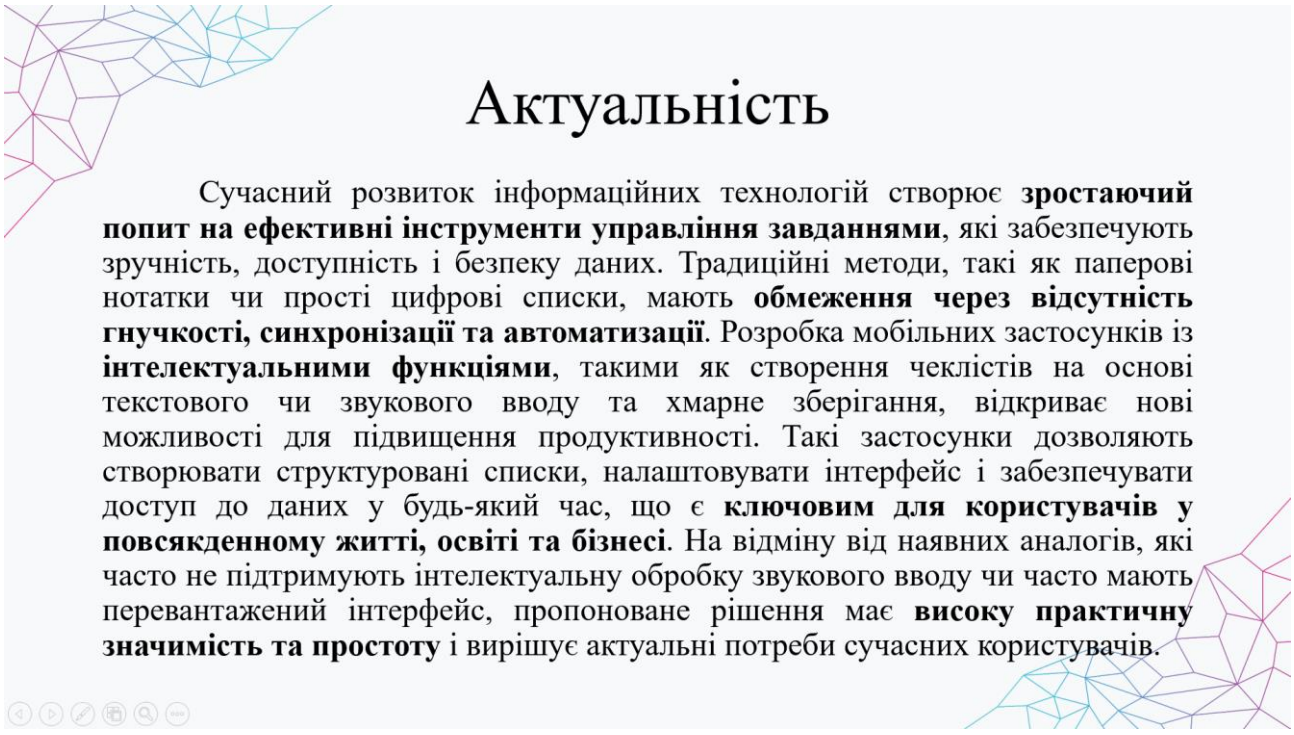


Рисунок Г.2 – Актуальність

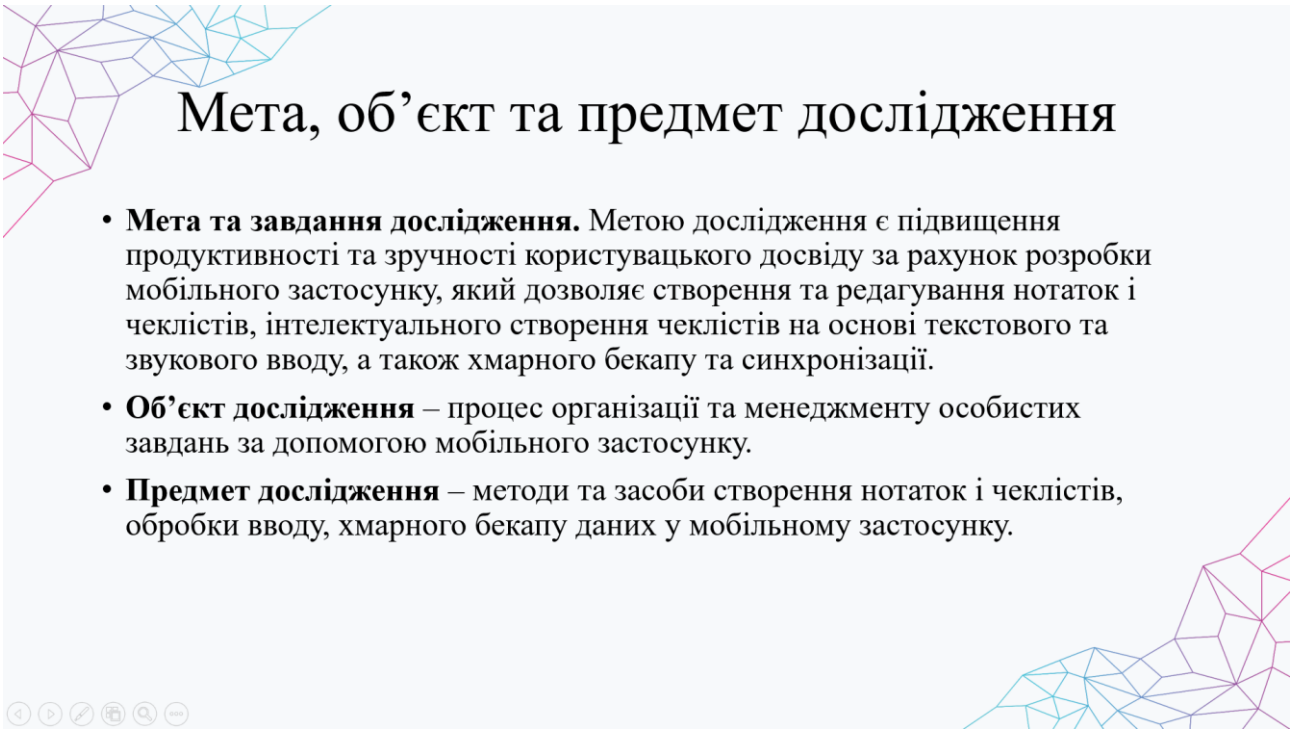


Рисунок Г.3 – Мета, об’єкт та предмет дослідження

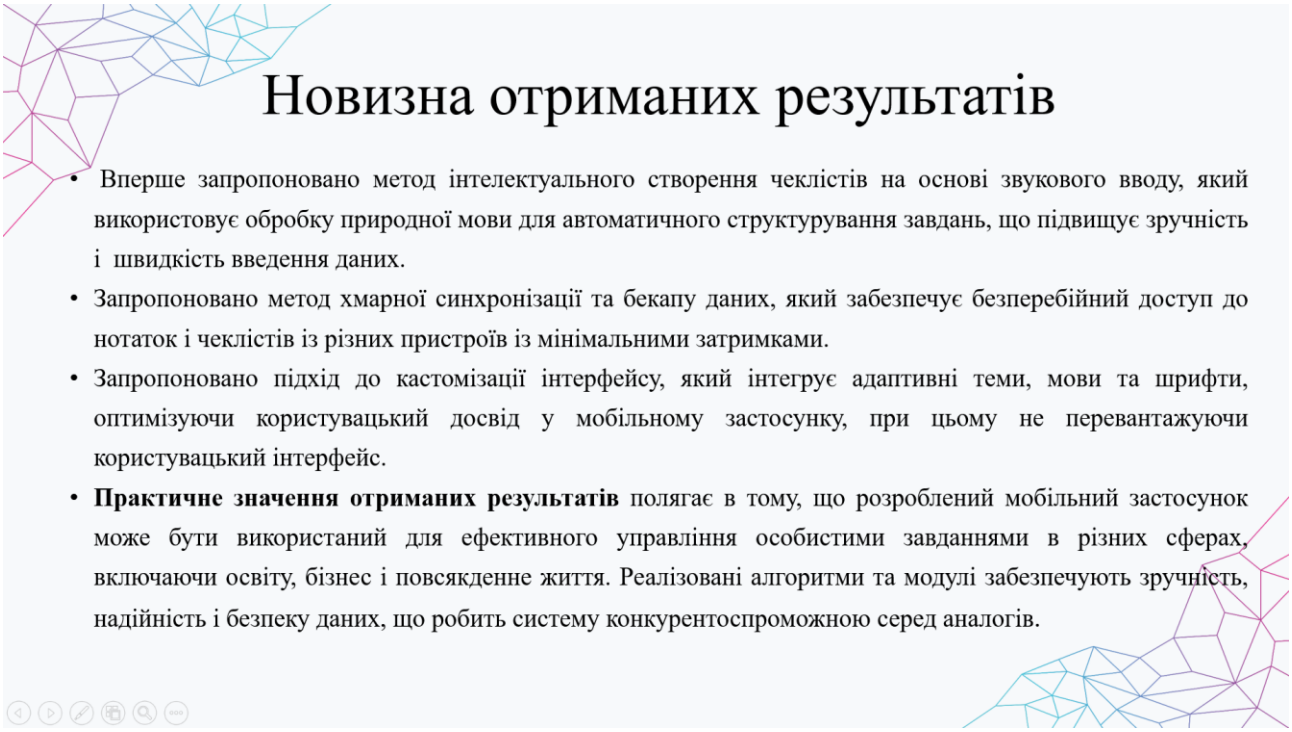


Рисунок Г.4 – Новизна отриманих результатів

Рисунок Г.5 – Порівняльний аналіз аналогів

Рисунок Г.6 – Архітектурна модель системи

Рисунок Г.7 – Методи модуля створення елементів

Рисунок Г.8 – Методи хмарної синхронізації системи

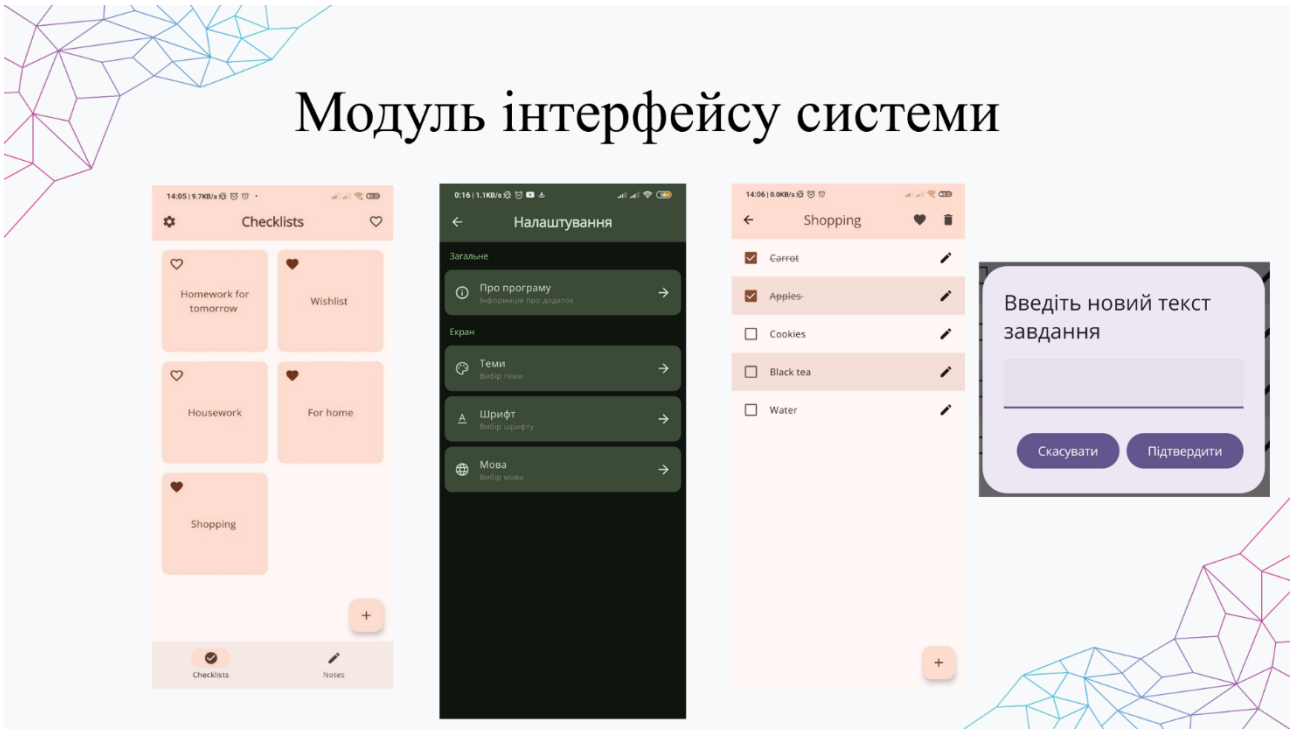


Рисунок Г.9 – Модуль інтерфейсу системи

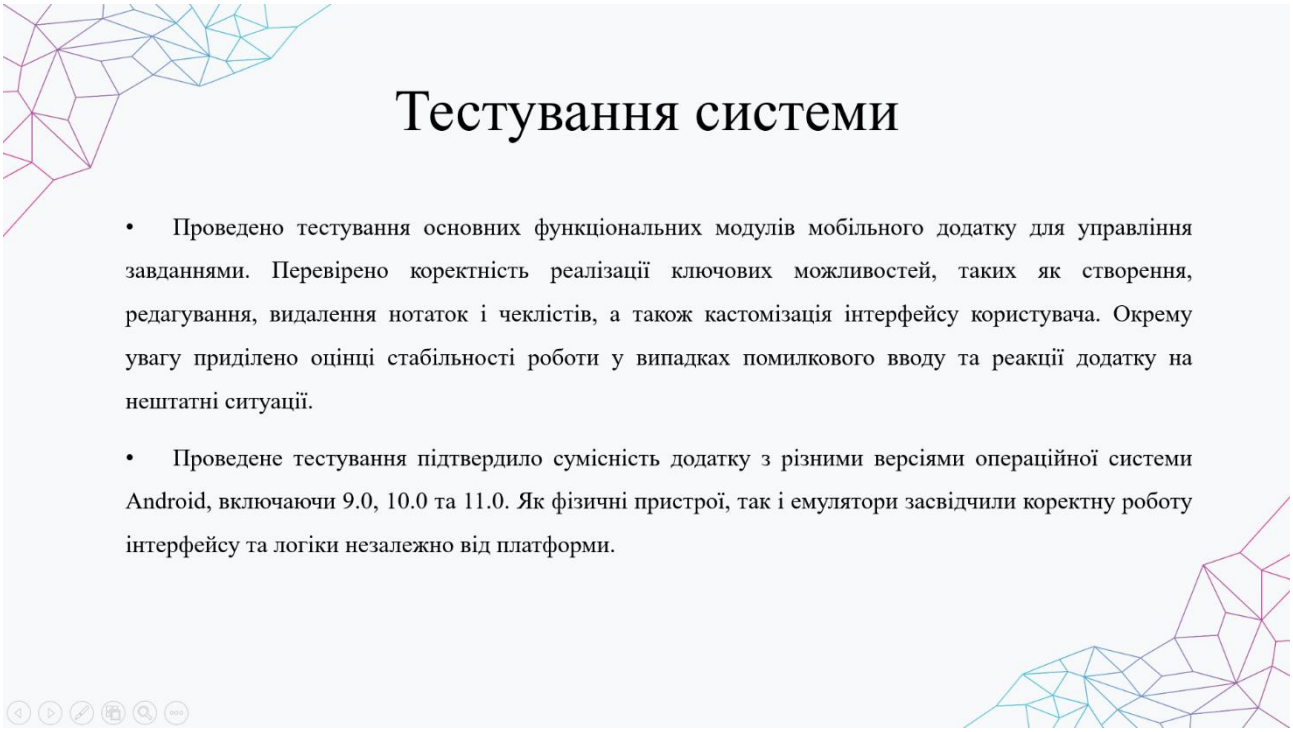


Рисунок Г.10 – Тестування системи

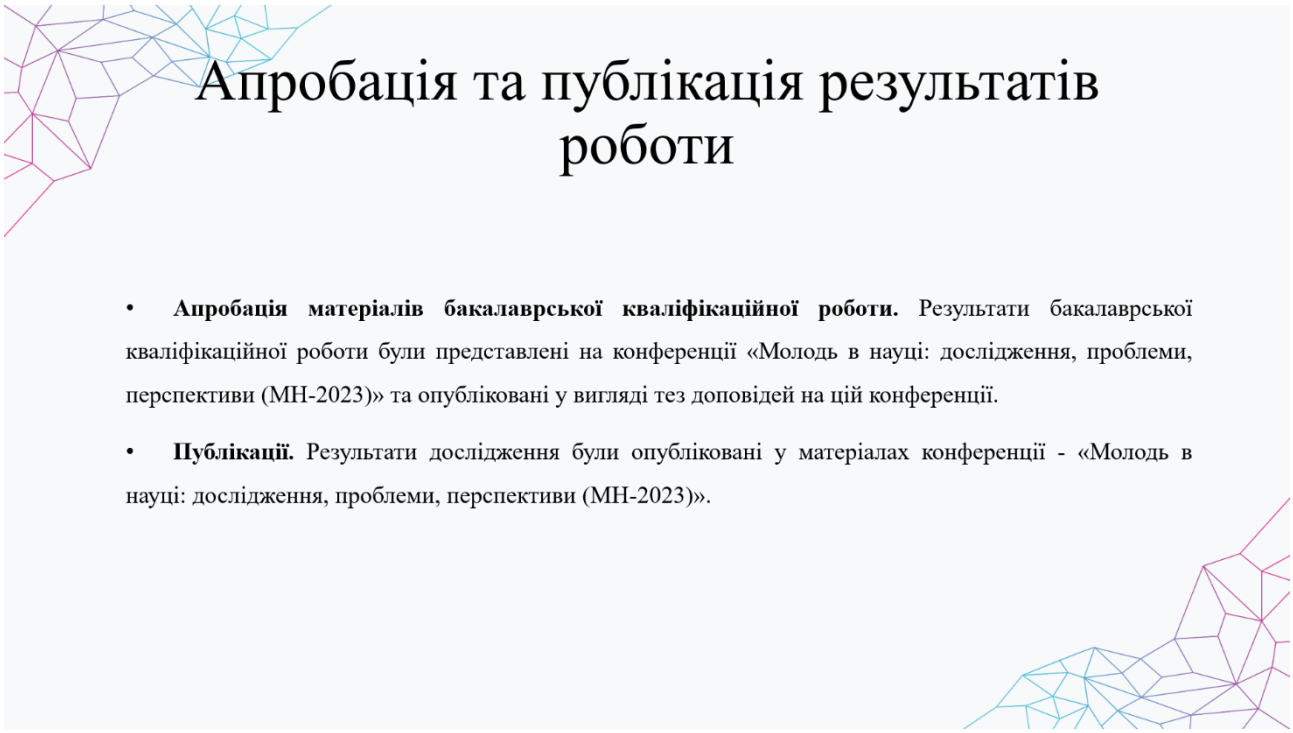


Рисунок Г.11 – Апробація та публікація результатів роботи

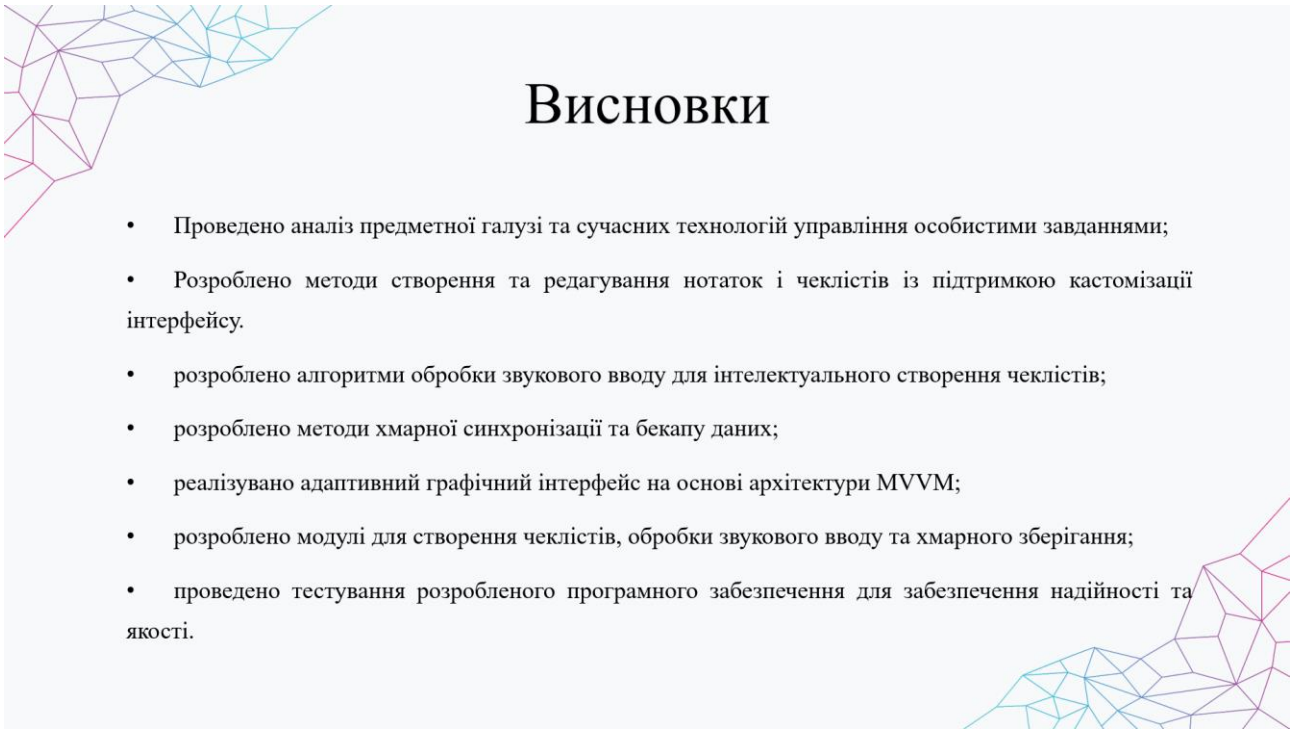


Рисунок Г.12 – Висновки

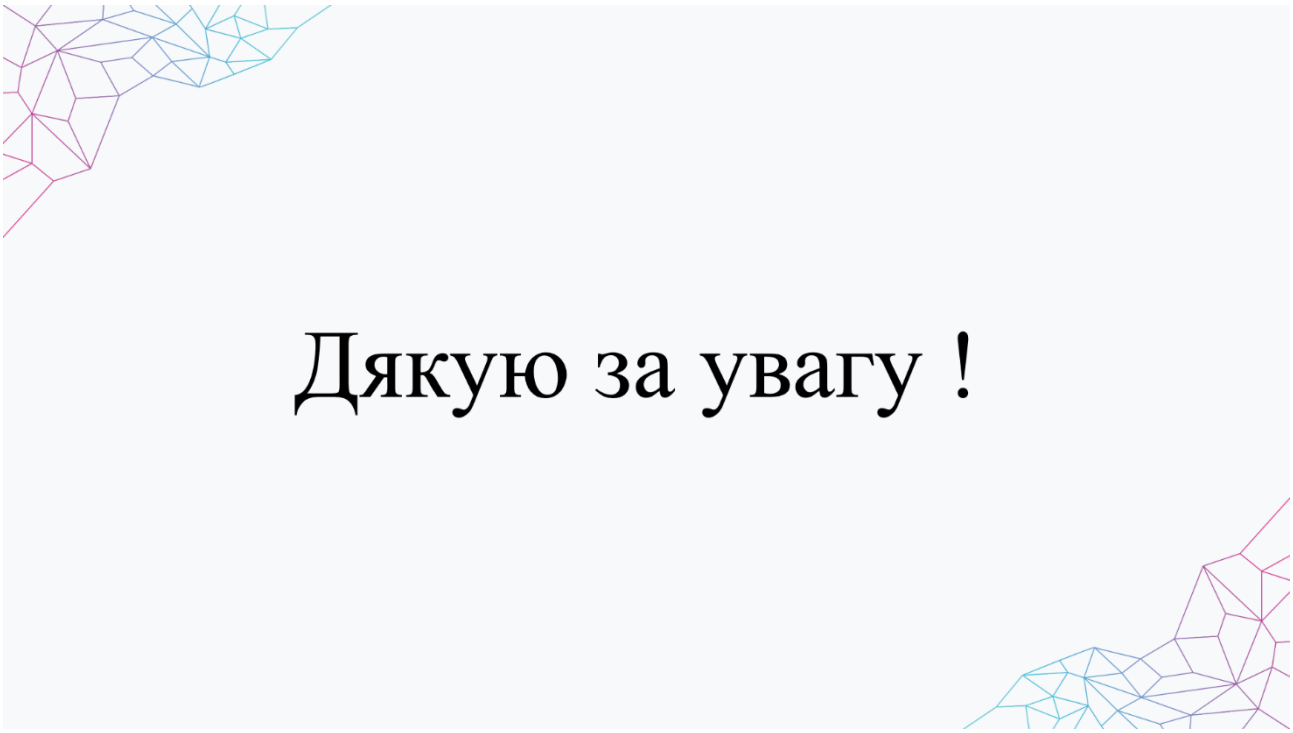


Рисунок Г.13 – Закінчення презентації