ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ

Пояснювальна записка

до кваліфікаційної роботи магістра

на тему:

«Дослідження та імплементація стратегій оптимізації взаємодії з хмарними сервісами на прикладі мобільного застосунку**»**

Виконав: здобувач 6 курсу, групи ПР2
спеціальності 121 «Інженерія
програмного забезпечення»
Мантур В.А.
(прізвище та ініціали)

Керівник: к.т.н., доц.Вишемирська С. В.
(прізвище та ініціали)

Рецензент: к.т.н., доц.Вороненко М. О.
(прізвище та ініціали)

Хмельницький – 2025р.

Факультет **Інформаційних технологій та дизайну**
Кафедра **Програмних засобів і технологій**
Освітньо-кваліфікаційний рівень **магістр**
Галузі знань **12 «Інформаційні технології»**
Спеціальність **121 «Інженерія програмного забезпечення»**
Освітньо-професійної програми **«Програмна інженерія»**

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗіТ,
к.т.н., доцент
_____О.Є.Огнєва
«____» _____ 20__ року

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ ЗДОБУВАЧУ

_____
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження та імплементація стратегій оптимізації взаємодії з хмарними сервісами на прикладі мобільного застосунку
керівник роботи Вишемирська Світлана Вікторівна, к.т.н., доцент,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ХНТУ від «15» вересня 2025 року №417-с.
2. Строк подання здобувачем роботи _____
3. Вихідні дані до роботи літературні та періодичні джерела, матеріали переддипломної практики
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Огляд теоретичних та практичних аспектів взаємодії мобільних застосунків і хмарних систем
2. Системний аналіз та формування технічного завдання
3. Методологічні підходи та вибір технологічних рішень
4. Розроблення та оцінювання оптимізованої взаємодії мобільного застосунку з хмарними сервісами

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
_____
_____
_____
_____

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного роботи	Строк виконання етапів роботи	Примітка
	1. Ознайомлення з тематикою дипломного проєктування, визначення та узгодження теми кваліфікаційної роботи	03.10.2025 – 07.10.2025	Готово
	2. Аналітичний огляд наукових і технічних джерел з мобільної розробки, хмарних сервісів та гібридних архітектур	08.10.2025 – 21.10.2025	Готово
	3. Системний аналіз предметної області та обґрунтування проблеми оптимізації взаємодії з хмарними сервісами	22.10.2025 – 30.10.2025	Готово
	4. Розроблення концепції архітектури мобільного застосунку для колекціонерів	31.10.2025 – 06.11.2025	Готово
	5. Вибір технологій, інструментів та методів реалізації мобільного застосунку	07.11.2025 – 12.11.2025	Готово
	6. Проєктування структури програмного забезпечення	13.11.2025 – 19.11.2025	Готово
	7. Реалізація прототипу застосунку	20.11.2025 – 02.12.2025	Готово
	8. Реалізація програмного прототипу мобільного застосунку з моделлю зберігання даних	03.12.2025 – 04.12.2025	Готово
	9. Тестування, налагодження та оцінювання ефективності роботи застосунку в онлайн- та офлайн-режимах	05.12.2025 – 06.12.2025	Готово
	10. Оформлення пояснювальної записки: написання розділів, висновків, списку використаних джерел і додатків	06.12.2025 – 07.12.2025	Готово

Здобувач _____ Мантур В.А

Керівник роботи _____ Вишемирська С.В.

(підпис) (прізвище та ініціали)

Анотація

У дипломній роботі виконано комплексне дослідження та практичну реалізацію мобільного застосунку для колекціонерів, орієнтованого на оптимізацію взаємодії з хмарними сервісами в умовах нестабільного мережевого середовища. Основну увагу зосереджено на побудові гібридної архітектури, що поєднує локальне зберігання даних і хмарні платформи, з метою підвищення продуктивності, надійності та зручності користування.

У роботі проаналізовано сучасні підходи до організації клієнт–серверної взаємодії в мобільних застосунках, розглянуто методи кешування, синхронізації та оптимізації передачі даних. Обґрунтовано доцільність використання гібридної моделі зберігання, яка поєднує локальну базу даних SQLite з хмарним сховищем Firebase Firestore, а також застосування сервісу Cloudinary для ефективної обробки та доставки медіаконтенту.

Розроблено архітектурну концепцію застосунку з чітким розподілом на рівень користувацького інтерфейсу, прикладної логіки та рівень даних. Описано структуру програмного проєкту, виконано декомпозицію системи з використанням UML-діаграм, змодельовано ключові бізнес-процеси та логічну модель даних. Реалізовано механізм двосторонньої синхронізації, який забезпечує коректну роботу застосунку як в онлайн-, так і в офлайн-режимі.

Проведено тестування та налагодження програмного продукту, результати яких підтвердили стабільність роботи, відповідність функціональним і нефункціональним вимогам, а також ефективність застосованих стратегій оптимізації. Отримані результати демонструють практичну цінність запропонованих рішень і можливість їх використання при розробці мобільних застосунків із підвищеними вимогами до автономності, продуктивності та масштабованості.

Annotation

This diploma thesis presents a comprehensive study and practical implementation of a mobile application for collectors, aimed at optimizing interaction with cloud services under conditions of unstable network connectivity. The primary focus is placed on the design of a hybrid architecture that combines local data storage with cloud platforms in order to improve performance, reliability, and overall user experience.

The paper analyzes modern approaches to organizing client–server interaction in mobile applications and examines methods of caching, data synchronization, and transmission optimization. The feasibility of using a hybrid storage model that integrates a local SQLite database with the Firebase Firestore cloud service is substantiated, along with the application of the Cloudinary service for efficient processing and delivery of media content.

An architectural concept of the application is developed with a clear separation into the user interface layer, application logic layer, and data layer. The structure of the software project is described, system decomposition is performed using UML diagrams, key business processes are modeled, and a logical data model is defined. A bidirectional synchronization mechanism is implemented, ensuring correct application behavior in both online and offline modes.

Testing and debugging of the software product were carried out, and the results confirmed the stability of the system, compliance with functional and non-functional requirements, as well as the effectiveness of the applied optimization strategies. The obtained results demonstrate the practical value of the proposed solutions and their applicability in the development of mobile applications with increased demands for autonomy, performance, and scalability.

РЕФЕРАТ

Дипломна робота містить 82 сторінки, 13 рисунків, 5 таблиць і 10 джерел.

Мета роботи – аналіз існуючих підходів до оптимізації взаємодії мобільних застосунків із хмарними сервісами, виявлення їх переваг та недоліків, а також розробка рекомендацій щодо підвищення ефективності обміну даними.

Актуальність теми. З стрімким розвитком мобільних та хмарних технологій все більше постає необхідність створення ефективних, безпечних і високонавантажених систем, що використовують зручну онлайн-інфраструктуру [3]. Дослідження та впровадження оптимізаційних стратегій дозволяє досягти кращого користувацького досвіду, підвищити швидкість обробки запитів та знизити витрати на використання хмарних ресурсів.

Об’єкт дослідження – методології оптимізації взаємодії з хмарними сервісами.

Предмет дослідження – процеси взаємодії мобільних застосунків із хмарними сервісами, а також методи та технології, що використовуються для їх оптимізації.

Наукова новизна одержаних результатів.

– систематизація та аналіз сучасних підходів до оптимізації обміну даними;

– розробка комбінованої стратегії оптимізації;

– удосконалення механізмів кешування та адаптивного управління трафіком;

– запропонування оптимальних конфігурацій використання serverless-архітектур

Ключові слова: мобільний застосунок, хмарні сервіси, гібридна архітектура, офлайн-режим, синхронізація даних, Firebase, SQLite, Cloudinary.

ЗМІСТ

ВСТУП.....8
РОЗДІЛ 1. ОГЛЯД ТЕОРЕТИЧНИХ ТА ПРАКТИЧНИХ АСПЕКТІВ ВЗАЄМОДІЇ МОБІЛЬНИХ ЗАСТОСУНКІВ І ХМАРНИХ СИСТЕМ.....12
1.1 Аналіз особливостей предметної області.....12
1.2 Дослідження наявних архітектур взаємодії клієнт–хмара.....14
1.3 Розгляд сучасних методів оптимізації.....16
1.4 Виявлення ключових проблем у роботі мобільних застосунків із хмарою.....19
1.5 Порівняння методів підвищення ефективності хмарної взаємодії.....22
1.6 Висновки до розділу 1.....24
РОЗДІЛ 2. СИСТЕМНИЙ АНАЛІЗ ТА ФОРМУВАННЯ ТЕХНІЧНОГО ЗАВДАННЯ.....26
2.1 Аналіз об’єкта дослідження та параметрів предметної області.....26
2.2 Формування функціональних і нефункціональних вимог до системи.....27
2.3 Визначення припущень, обмежень та зовнішніх чинників.....31
2.4 Підсумки системного аналізу.....33
РОЗДІЛ 3. МЕТОДОЛОГІЧНІ ПІДХОДИ ТА ВИБІР ТЕХНОЛОГІЧНИХ РІШЕНЬ. 34
3.1 Аналіз існуючих рішень.....34
3.3 Визначення сценаріїв функціонування оптимізованої системи.....38
3.4 Розроблення архітектурної концепції мобільного застосунку.....41
3.5 Висновки щодо вибраних методів та засобів.....43
РОЗДІЛ 4. РОЗРОБЛЕННЯ ТА ОЦІНЮВАННЯ ОПТИМІЗОВАНОЇ ВЗАЄМОДІЇ МОБІЛЬНОГО ЗАСТОСУНКУ З ХМАРНИМИ СЕРВІСАМИ.....44
4.1. Архітектура і дизайн додатку.....44
4.2. Використані інструменти і технології.....47
4.3. Опис структури проекту.....49
4.4. Декомпозиція програмного додатку.....53
4.5. Процес тестування та налагодження.....59
5.6. Результати розробки програмного продукту.....60
4.7. Послідовність роботи користувача з додатком.....63
4.8. Аналіз результатів застосування продукту.....65
4.9. Перспективи подальшого розвитку додатку.....67
4.10 Висновки до 4 розділу.....68
ВИСНОВКИ.....70
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....72
ДОДАТКИ.....73

ВСТУП

Стрімкий розвиток технологій оброблення даних та зростання популярності мобільних застосунків зумовлюють підвищення вимог до якості взаємодії між клієнтськими пристроями та хмарними інфраструктурами. Сучасні мобільні застосунки активно використовують хмарні сервіси для зберігання, обробки та синхронізації даних, що дозволяє значно покращити продуктивність, масштабованість і доступність програмного забезпечення [3]. Завдяки хмарним технологіям мобільні пристрої можуть обробляти великі обсяги даних, не перевантажуючи локальні ресурси, що особливо важливо для застосунків, які працюють із мультимедійним контентом, аналітикою великих даних або машинним навчанням [5].

Зростання кількості сервісів, які потребують оперативної синхронізації та взаємодії у режимі реального часу, актуалізує необхідність у розробленні підходів, що забезпечують ефективне керування даними та оптимальне розподілення навантаження. Незважаючи на значний прогрес у галузі хмарних технологій, питання мінімізації затримок, адаптації застосунків до змінних мережевих умов та зменшення вартості хмарних операцій залишаються відкритими. Більшість існуючих рішень орієнтовані на універсальні сценарії, що не завжди враховують специфіку мобільних платформ, обмеження енергоспоживання та особливості поведінки користувачів. Тому дослідження методів оптимізації взаємодії між застосунком та хмарною інфраструктурою є важливим елементом розвитку сучасних інформаційних систем.

Вибір теми обумовлений потребою у створенні механізмів, здатних підвищити ефективність роботи мобільних рішень у різних мережевих середовищах. Для України ця проблема є особливо актуальною у контексті цифрової трансформації державного та приватного секторів, оскільки стабільність і швидкість доступу до хмарних сервісів визначають якість електронних продуктів, що застосовуються у сфері бізнесу, логістики, освіти, телекомунікацій та адміністративних послуг. Оптимізована взаємодія з хмарною інфраструктурою

сприяє підвищенню надійності програмних систем, зменшенню експлуатаційних витрат та покращенню користувацького досвіду.

Метою дослідження є аналіз технічних підходів і розроблення практичних механізмів, які удосконалюють взаємодію мобільного застосунку з хмарними сервісами за рахунок оптимізованих стратегій комунікації. Під час вивчення теми розглядаються способи підвищення швидкодії, методи зниження мережевих затримок, а також алгоритмічні рішення, що дозволяють ефективно розподіляти навантаження між локальною та віддаленою обробкою. Окрему увагу приділено принципам побудови архітектур, які забезпечують стійкість роботи застосунку та здатність адаптуватися до непередбачуваних змін навантаження.

Об’єктом дослідження виступає процес взаємодії мобільного застосунку з хмарною інфраструктурою, що охоплює механізми обміну даними, передачу запитів та формування відповідей.

Предметом дослідження є методи оптимізації цієї взаємодії, включно з підходами до зменшення затримок, покращення стабільності мережевих операцій, підвищення ефективності кешування та використання адаптивних алгоритмів маршрутизації. До уваги беруться моделі, які дозволяють збалансувати енергоспоживання мобільного пристрою та інтенсивність запитів до хмари.

Наукова новизна роботи полягає у формуванні комплексного підходу до оптимізації взаємодії між клієнтським застосунком і хмарною інфраструктурою, що поєднує аналіз алгоритмів, архітектурні рішення та практичні експерименти. У дослідженні запропоновано спосіб інтеграції адаптивних механізмів вибору стратегії взаємодії залежно від поточного стану мережі та вимог до швидкодії. Запропонована концепція враховує динамічні умови роботи мобільних платформ і спрямована на зменшення затримок без суттєвого збільшення витрат на хмарні ресурси.

Актуальність теми. З стрімким розвитком мобільних та хмарних технологій все більше постає необхідність створення ефективних, безпечних і високонавантажених систем, що використовують зручну онлайн-інфраструктуру

[7]. Дослідження та впровадження оптимізаційних стратегій дозволяє досягти кращого користувацького досвіду, підвищити швидкість обробки запитів та знизити витрати на використання хмарних ресурсів.

Практичне значення дослідження полягає у можливості безпосереднього застосування отриманих результатів у реальних мобільних продуктах. Розроблені механізми можуть бути інтегровані в різні типи застосунків, що потребують стабільного доступу до хмарних даних та швидкої реакції на дії користувача. Отримані результати також можуть бути використані під час проєктування масштабованих систем, оптимізованих під умови української ринкової інфраструктури та сучасних вимог цифрової економіки.

Завдання, які було виконано під час дослідження:

– Проаналізувати сучасні підходи до взаємодії мобільних застосунків із хмарними сервісами.

– Дослідити особливості мережевих алгоритмів, що впливають на затримки, стабільність і пропускну здатність.

– Сформувати архітектурну модель оптимізованої системи обміну даними між клієнтом і хмарою.

– Реалізувати основні компоненти: механізми передачі даних, обробку запитів, адаптивну логіку та оптимізаційні модулі.

– Налаштувати динамічний вибір стратегії взаємодії залежно від стану мережі й навантаження.

– Виконати експериментальну оцінку продуктивності, енергоспоживання та стійкості системи.

– Проаналізувати отримані результати та визначити можливі напрями подальшого вдосконалення застосунку.

РОЗДІЛ 1. ОГЛЯД ТЕОРЕТИЧНИХ ТА ПРАКТИЧНИХ АСПЕКТІВ ВЗАЄМОДІЇ МОБІЛЬНИХ ЗАСТОСУНКІВ І ХМАРНИХ СИСТЕМ

1.1 Аналіз особливостей предметної області

Предметна область, що охоплює процеси взаємодії мобільних застосунків із хмарними сервісами, формується на перетині кількох напрямів сучасних інформаційних технологій, зокрема розподілених обчислень, мережевої інфраструктури та програмної інженерії. Мобільні застосунки дедалі частіше функціонують як клієнтські компоненти складних систем, у яких значна частина оброблення даних відбувається у хмарному середовищі. Така модель роботи дозволяє забезпечити масштабованість, централізоване оновлення функціоналу та доступ до великих обсягів інформації, що зберігаються на віддалених серверах.

У сучасних умовах мобільні пристрої виступають універсальними інструментами доступу до цифрових ресурсів, а хмарні технології виконують роль основного джерела обчислювальних потужностей. Використання хмарної інфраструктури дає змогу зменшити залежність застосунку від апаратних характеристик кінцевого пристрою, оскільки основний обсяг оброблення переноситься на спеціалізовані серверні платформи. Проте такий підхід спричиняє появу нових викликів, пов’язаних із необхідністю швидкого та стабільного обміну даними, адже ефективність роботи системи визначається не лише структурою обчислень, а й якістю мережевої взаємодії.

Важливою особливістю предметної області є висока динамічність середовища, у якому працює мобільний застосунок. Користувач може переміщуватися між різними типами мереж, мати нестабільне з’єднання або обмежений доступ до ресурсів. Це створює значні коливання затримок, пропускної здатності та надійності каналів зв’язку, що безпосередньо впливає на роботу застосунку та вимагає застосування адаптивних стратегій оброблення даних. Мобільні системи повинні вміти самостійно підлаштовуватися під змінні

умови, визначати оптимальні способи комунікації та забезпечувати передбачуваність реакції для користувача.

Окремої уваги потребує питання енергоспоживання. Хоча хмарні обчислення зменшують навантаження на центральний процесор мобільного пристрою, часті мережеві запити та передавання значних обсягів інформації можуть швидко витрачати заряд акумулятора. Це робить оптимізацію взаємодії з хмарою важливим інструментом підвищення загальної ефективності застосунку та формує потребу у використанні моделей, що балансують між швидкодією та економним використанням енергії.

Оскільки мобільні застосунки дедалі частіше використовуються як інтерфейси до складних бізнес-процесів, особливої ваги набувають вимоги до безпеки та конфіденційності даних. Хмарні сервіси працюють у багатокористувацькому середовищі, де інформація між різними системами може передаватися через спільні мережеві канали. Це зумовлює потребу в надійних механізмах шифрування, контролю доступу та перевірки цілісності інформації. Крім того, архітектура застосунку має враховувати можливість атак на мережеву інфраструктуру, що створює додаткові вимоги до захисних механізмів.

Хмарні платформи зазвичай пропонують широкий набір сервісів, які можуть суттєво покращити функціонування застосунку, серед них автоматизоване балансування навантаження, засоби кешування, модулі для інтелектуального аналізу даних і сервіси, що забезпечують швидке масштабування. Проте використання цих можливостей вимагає ретельного аналізу особливостей конкретного мобільного застосунку, оскільки некоректна конфігурація може спричинити зростання затримок або надмірне використання ресурсів, що збільшує експлуатаційні витрати.

Таким чином, предметна область включає взаємодію багатьох факторів, які визначають ефективність роботи мобільного застосунку в хмарному середовищі. Сюди входять архітектурні особливості клієнтської частини, параметри мережевих каналів, властивості обчислювальних сервісів і специфіка поведінки

користувачів. Саме багатокомпонентність та варіативність умов функціонування мобільних систем створюють підґрунтя для дослідження стратегій оптимізації, що дозволяють підвищити швидкодію, знизити затримки та забезпечити стабільність роботи навіть у складних мережевих обставинах.

1.2 Дослідження наявних архітектур взаємодії клієнт–хмара

Архітектури, що визначають взаємодію між мобільним застосунком і хмарною інфраструктурою, формують основу функціонування сучасних розподілених систем. Вони охоплюють способи передавання даних, принципи розподілу обчислювального навантаження та механізми забезпечення доступності сервісів. Еволюція хмарних технологій зумовила появу різних моделей комунікації, які відрізняються рівнем централізації, масштабованістю, стійкістю до змін мережевих умов та здатністю підтримувати високе навантаження.

Найпоширенішою архітектурною моделлю залишається традиційна клієнт–серверна взаємодія (рисунок 1.1), у якій мобільний застосунок виступає в ролі ініціатора запитів, а хмарний сервер забезпечує оброблення інформації, зберігання даних та виконання бізнес-логіки.

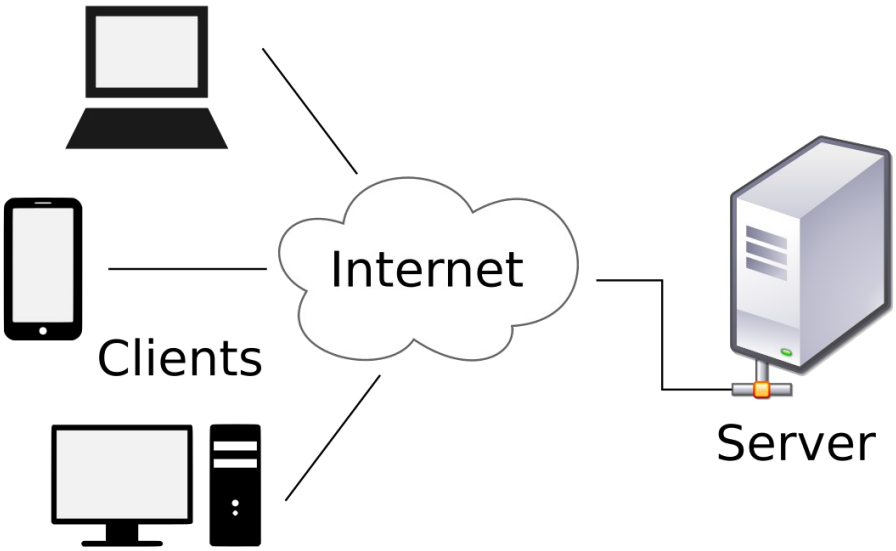

Рис. 1.1 – Схема роботи клієнт–серверної архітектурної моделі

Така побудова системи дозволяє спростити логіку на боці клієнта та використовувати потужні ресурси хмарної платформи. Проте вона створює залежність від якості мережевого з’єднання, оскільки навіть короткочасні збої у мережі можуть впливати на швидкодію застосунку.

На основі базової моделі виникли модифіковані архітектури, орієнтовані на підвищення продуктивності та зменшення затримок. Сучасні системи часто використовують концепції кешування, попереднього завантаження та локальної обробки найбільш критичних операцій, що дає змогу зменшити кількість запитів до хмарної інфраструктури. У таких випадках вибір того, які дані слід обробляти локально, а які – відправляти на сервер, визначається характеристиками мережі, вимогами до актуальності інформації та необхідною швидкодією.

Поширення багаторівневих хмарних платформ привело до розвитку архітектур, у яких застосунок взаємодіє не з одним сервером, а з комплексом сервісів, кожен із яких відповідає за окрему групу функцій. У цьому випадку застосунок може підключатися до сервісів авторизації, баз даних, систем аналітики та модулів оброблення даних, що працюють незалежно один від одного. Мікросервісний підхід дозволяє гнучко масштабувати ресурси, проте ускладнює структуру мережевих запитів і потребує додаткових механізмів координації.

Особливе місце займають архітектури, що поєднують локальну та хмарну обробку, тобто гібридні рішення. Вони формуються навколо ідеї адаптивного розподілу обчислень залежно від поточних умов. У таких системах застосунок може виконувати частину завдань автономно, наприклад аналіз невеликих обсягів даних або тимчасове збереження інформації, а більш ресурсоємні операції передавати у хмару. Подібний підхід покращує стабільність роботи та мінімізує ризики, пов’язані з нестабільним мережевим середовищем.

Разом із розвитком хмарних платформ поширилися моделі, що підтримують оброблення даних на периферії мережі, тобто ближче до користувача. У межах таких архітектур застосунок може отримувати доступ до проміжних вузлів, розташованих між клієнтом і центральною хмарою, що значно скорочує затримки

й розвантажує основний сервер. Ця концепція поєднує гнучкість хмарних сервісів із швидкодією локальної обробки, проте потребує складних механізмів керування даними та їхньої синхронізації.

Ще одним важливим напрямом розвитку стали подієві архітектури, у яких взаємодія клієнта з хмарою ґрунтується на асинхронній моделі обміну повідомленнями. Застосунок передає у систему подію, а сервер реагує на неї у міру доступності ресурсів. Така модель підвищує стійкість до пікових навантажень і дозволяє уникнути блокування процесів. Вона широко використовується у застосунках, де критично важливим є своєчасне отримання інформації або оброблення подій у реальному часі.

Усі розглянуті архітектури мають власні сильні сторони та обмеження, що визначають ефективність роботи мобільного застосунку залежно від сфери його застосування. Вони окреслюють рамки, у яких формуються сучасні рішення щодо оптимізації комунікації між клієнтом і хмарою, та підкреслюють необхідність адаптивних підходів, здатних враховувати властивості мережевого середовища, вимоги до продуктивності та особливості кінцевого пристрою. Аналіз доступних архітектур дозволяє зрозуміти, які моделі є найбільш релевантними для сценаріїв, що потребують високої швидкодії, стабільності та контролю над ресурсами.

1.3 Розгляд сучасних методів оптимізації

У сучасному цифровому середовищі, де обсяг інформації невпинно зростає, питання оптимізації процесів взаємодії мобільних та вебзастосунків із хмарними платформами набуває особливої ваги.

Необхідність оперативного отримання даних, стабільної роботи програмних систем та раціонального використання доступних ресурсів створює комплекс технічних викликів, що вимагають пошуку ефективних інженерних рішень. Підвищення швидкодії та зменшення витрат на оброблення інформації стає ключовим завданням у проєктуванні сучасних розподілених систем.

Однією з головних проблем, що виникають при роботі з хмарними сервісами, є висока вартість використання обчислювальних ресурсів і передачі даних [8]. Для організацій це означає потребу у зменшенні експлуатаційних витрат без втрати продуктивності та надійності системи, що вимагає розробки механізмів, здатних оптимізувати використання хмарної інфраструктури.

Суттєвою проблемою також виступає забезпечення швидкого доступу до інформації за умов обмеженої пропускної здатності мережі або нестійкого підключення. Одним із дієвих способів зменшення залежності від якості з’єднання є використання кешування, що дає можливість зберігати вже отримані дані локально. Доступ до часто використовуваних даних прискорюється, що мінімізує потребу в повторних зверненнях до хмарного сховища [9].

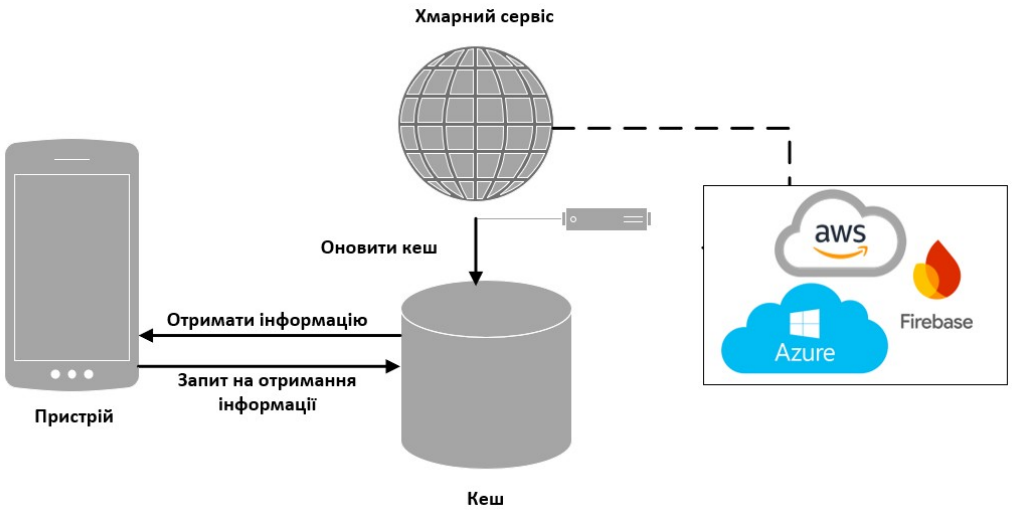

Рисунок 1.2 – Система кешування даних

Такий підхід значно знижує кількість звернень до віддаленого сервера, зменшує навантаження на мережу та підвищує швидкість оброблення операцій, що позитивно впливає на загальну якість роботи системи. На рисунку 1.2 наведено спрощену схему використання кешу для локального зберігання даних.

Попри переваги, кешування має і певні обмеження. Локальні копії даних можуть займати значний обсяг пам’яті, а також потребують засобів актуалізації, щоб уникнути використання застарілої інформації. Це вимагає впровадження

механізмів перевірки змін, які дають змогу підтримувати узгодженість між локальним сховищем та хмарним ресурсом.

Гібридні моделі, що поєднують локальні засоби збереження та можливості хмарної інфраструктури, забезпечують більш стійку роботу застосунків в умовах непередбачуваних мережевих коливань. Завдяки таким підходам користувач може продовжувати роботу навіть за відсутності з’єднання, а всі зміни синхронізуються після відновлення доступу в мережу. Подібні рішення підвищують надійність програмних систем і зменшують ризики втрати даних.

Окремий напрям оптимізації пов’язаний зі зменшенням обсягів переданих даних за допомогою їх стискання. Цей метод спрямований на зменшення обсягу інформації, що передається мережею, що, своєю чергою, знижує витрати на трафік і прискорює завантаження даних [4].

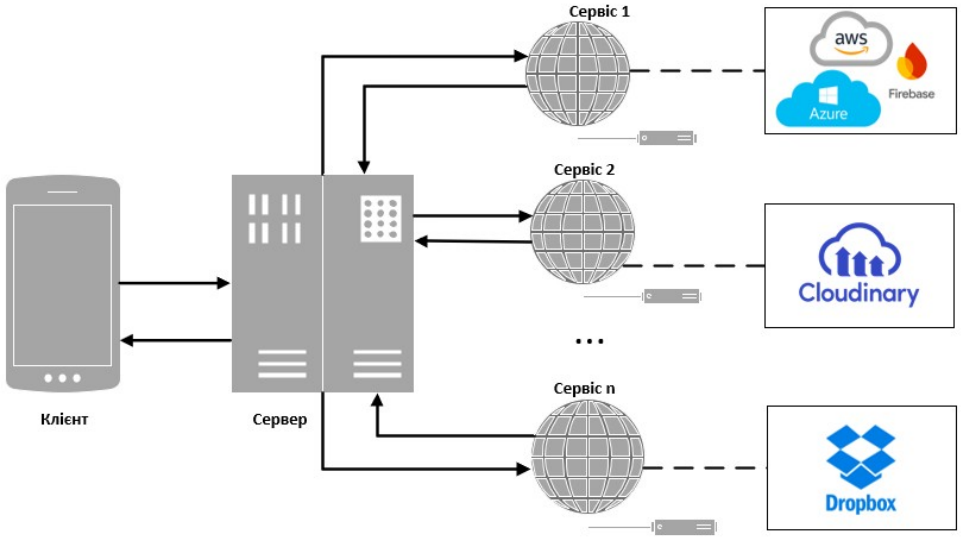

Рисунок 1.3 – Розподіл виконання задач між різними сервісами

Ще одним поширеним підходом до оптимізації є розподіл операцій між кількома вузькоспеціалізованими сервісами. Це дозволяє знизити навантаження на окремі компоненти та забезпечити більш рівномірне використання хмарних ресурсів. Схематичне представлення розподілу задач відображено на рисунку 1.3.

Перевагою такого підходу є можливість адаптувати інфраструктуру до вимог конкретного типу даних. Наприклад, матеріали, що практично не

змінюються, можуть зберігатися на платформах із низькою вартістю. Дані ж, що потребують високої швидкодії доступу, доцільно розміщувати у сервісах, здатних забезпечити швидку реакцію на запити. Проте використання багатьох сервісів потребує продуманої системи синхронізації, що дозволяє уникнути розбіжностей між різними сховищами.

Ще одним викликом для багатосервісних інфраструктур є необхідність додаткових модулів, відповідальних за маршрутизацію, розподіл запитів і гарантування безпеки переданих даних. Часто такі системи потребують використання проміжних серверів або шлюзів, які виконують координаційну функцію.

Значного поширення набуло й використання мереж розповсюдження контенту, що дозволяють розміщувати файли на серверах у різних географічних точках. Така технологія забезпечує швидший доступ до статичних матеріалів, оскільки вони надходять із найближчого вузла мережі. Використання CDN дозволяє розподілити контент між численними серверами, розташованими в різних географічних регіонах [10].

Разом із тим, ця технологія практично не дає переваг при роботі з динамічними даними, які постійно змінюються. Крім того, провайдери таких послуг часто встановлюють ліміти на безкоштовний трафік або кількість звернень, що може вплинути на загальну вартість експлуатації системи.

1.4 Виявлення ключових проблем у роботі мобільних застосунків із хмарою

Функціонування мобільних застосунків у поєднанні з хмарною інфраструктурою відбувається в умовах значної мінливості технічних параметрів, що впливають на стабільність, продуктивність та загальну якість роботи системи. Хоча хмарні сервіси забезпечують широкий спектр можливостей для зберігання, оброблення та передавання даних, їх використання породжує низку обмежень, які

необхідно враховувати під час проєктування архітектури застосунку. Виявлення таких проблем є важливою передумовою для формування ефективних стратегій оптимізації.

Однією з основних складностей є залежність мобільного застосунку від характеристик мережевого середовища. Швидкість інтернет-з’єднання, величина затримок та стабільність каналу суттєво впливають на здатність системи обробляти запити у режимі, наближеному до реального часу. У випадках, коли користувач перебуває у зоні з нестійким зв’язком, застосунок може працювати зі значними перебоями, що знижує якість користувацького досвіду. Висока варіативність параметрів мережі ускладнює побудову алгоритмів, які повинні реагувати на зміни без істотних втрат у продуктивності.

Іншою проблемою є затримка, що виникає під час передавання та оброблення інформації у хмарі. Навіть короткі паузи між надсиланням запиту та отриманням відповіді можуть суттєво впливати на поведінку застосунку, особливо у сценаріях, що потребують швидкої реакції. У хмарному середовищі час оброблення залежить від багатьох факторів, серед яких навантаження на сервери, географічне розташування користувача та внутрішні механізми розподілу ресурсів провайдера. Усы ці параметри створюють додаткову невизначеність у роботі системи.

Не менш важливою є проблема перевантаження ресурсів на стороні хмарної платформи. Коли значна кількість застосунків одночасно звертається до одного сервісу, можливе зниження продуктивності або навіть часткові збої у роботі серверної частини системи. Це створює ризики, особливо для тих мобільних продуктів, які мають значну кількість активних користувачів. У таких випадках системі необхідно забезпечити стійкість навіть у моменти пікового навантаження.

Питання безпеки також відіграє ключову роль. Хоча більшість сучасних хмарних провайдерів впроваджують комплексні механізми захисту, передавання конфіденційних даних через мережу завжди створює додаткові ризики. Мобільні застосунки можуть піддаватися атакам, спрямованим на перехоплення трафіку,

модифікацію даних або несанкціонований доступ до хмарних ресурсів. Наявність таких загроз зобов’язує розробників враховувати питання шифрування, автентифікації та контролю доступу вже на ранніх етапах проєктування.

Важливою складовою проблем є також обмеження технічних можливостей мобільних пристроїв. Обмежені обчислювальні ресурси, невеликий обсяг оперативної пам’яті та залежність від заряду акумулятора впливають на здатність застосунку ефективно взаємодіяти з хмарою. Часті мережеві звернення можуть призводити до пришвидшеного споживання енергії, що змінює поведінку застосунку та робить його менш зручним для користувача. Тому розробникам важливо збалансувати кількість операцій, що виконуються локально, та обсяг даних, який передається через мережу.

Окрему увагу привертає проблема узгодженості даних. У ситуаціях, коли застосунок працює одночасно з локальним сховищем та віддаленим сервером, виникає ризик появи конфліктів або дублювання інформації. Відсутність синхронності між різними копіями даних може призвести до помилок у роботі системи, оскільки мобільний клієнт може використовувати застарілу або неповну інформацію. Забезпечення узгодженості, зокрема у середовищах із нестабільним інтернетом, вимагає розроблення складних механізмів підтвердження змін і відновлення станів.

Ще одним викликом у роботі мобільних застосунків із хмарою є необхідність підтримки швидкого масштабування. У разі різкого збільшення кількості користувачів хмарна платформа повинна динамічно розширювати доступні ресурси, аби уникнути деградації продуктивності. Водночас надмірне масштабування призводить до зростання фінансових витрат, тому оптимізація в цьому контексті повинна враховувати і технічні можливості, і економічні обмеження.

Узагальнюючи наведене, можна стверджувати, що взаємодія мобільних застосунків із хмарними сервісами супроводжується широким спектром технічних проблем, які охоплюють мережеві обмеження, питання продуктивності,

безпеку, синхронізацію даних, енергоспоживання та масштабованість. Виявлення цих проблем дозволяє сформувати основу для подальшого пошуку оптимізаційних рішень, спрямованих на підвищення ефективності роботи мобільних систем у хмарному середовищі.

1.5 Порівняння методів підвищення ефективності хмарної взаємодії

Ефективність взаємодії між мобільним застосунком і хмарною інфраструктурою визначається тим, наскільки швидко й стабільно система здатна обмінюватися даними, а також тим, наскільки раціонально вона використовує доступні ресурси. У зв’язку з цим розробники застосовують різні методи оптимізації, які мають свої переваги, обмеження та специфіку застосування. Порівняння таких підходів дає змогу визначити найдоцільніші стратегії для різних сценаріїв використання мобільних систем.

Одним із найпоширеніших методів підвищення продуктивності взаємодії з хмарою є використання локального кешування. Його основна перевага полягає у зменшенні кількості мережевих звернень, що дозволяє знизити затримки під час доступу до даних і зменшити навантаження на інфраструктуру. Кешування особливо корисне у випадках повторного використання інформації, яка не змінюється динамічно. Проте цей метод має і слабкі сторони: він потребує додаткової пам’яті, а також вимагає реалізації механізмів оновлення, щоб уникнути втрати актуальності збережених даних. Через це локальне кешування підходить не до всіх типів інформації, а його ефективність значною мірою залежить від правильно підібраної стратегії синхронізації.

На відміну від кешування, метод стиснення даних спрямований на зменшення обсягу інформації, яка передається мережею. Застосування компресії дозволяє прискорити обмін даними та оптимізувати витрати на трафік. Цей підхід ефективний для великих файлів та структур, що мають повторювані елементи. Проте стиснення потребує додаткових обчислень: дані мають бути закодовані на

стороні клієнта і розкодовані на сервері або навпаки. У деяких випадках ці операції можуть збільшувати навантаження на пристрій, що особливо відчутно для слабких мобільних платформ. Тому компресія є вигідною лише тоді, коли економія трафіку перевищує витрати на виконання алгоритмів стискання.

Інший підхід полягає у використанні розподіленої обробки даних між кількома сервісами. Подібна архітектура дозволяє збалансувати навантаження, підвищити гнучкість та адаптивність системи. Якщо один сервіс виходить із ладу або працює повільніше, інші компоненти можуть продовжувати роботу без суттєвих втрат. Це забезпечує стійкість системи та можливість масштабування у разі зростання кількості користувачів. Попри переваги, використання багатьох сервісів створює складність у підтримці узгодженості даних, вимагає продуманої маршрутизації запитів та додаткових заходів захисту.

До методів, що покращують доступ до статичних ресурсів, належать технології розподіленого доставлення контенту. CDN-системи забезпечують зберігання копій файлів на вузлах, розташованих у різних регіонах, що дозволяє значно пришвидшити доступ до них. Перевага цього підходу проявляється у випадках, коли застосунок працює з великими обсягами зображень, відео або іншого мультимедійного контенту. Однак для інформації, яка часто змінюється, подібні рішення не гарантують актуальності, а іноді навіть ускладнюють синхронізацію даних. Також важливо враховувати фінансову складову, оскільки деякі провайдери встановлюють обмеження на використання CDN.

Гібридні стратегії, що поєднують локальну обробку, кешування та взаємодію з хмарними серверними компонентами, надають найбільшу гнучкість під час адаптації до реальних умов. Вони дозволяють мобільним застосункам працювати стабільніше у середовищах з нестійким мережевим доступом та зменшувати час відгуку в критичних сценаріях. Гібридні рішення можуть автоматично визначати, яку частину роботи доцільно виконувати локально, а яку – передавати до хмари. Це робить їх ефективними для систем, що обробляють змішані типи даних або мають значні коливання навантаження. Проте складність

їх реалізації значно вища, оскільки розробникам необхідно передбачити механізми керування станом, узгодженням даних та обробленням помилок, що виникають у процесі взаємодії.

Порівнюючи наведені методи, можна дійти висновку, що жоден із них не є універсальним. Вибір конкретної оптимізаційної стратегії залежить від вимог системи, типу даних, мережевих характеристик та очікуваного рівня продуктивності. Важливим чинником є також економічна доцільність, оскільки кожен підхід має свої витрати на впровадження і підтримку. Тому найбільш ефективні рішення зазвичай передбачають комбінацію кількох методів, які взаємодіють між собою та компенсують слабкі сторони один одного.

Таким чином, порівняння різних технік оптимізації дозволяє визначити найбільш перспективні напрями для підвищення ефективності хмарної взаємодії та сформувати базу для подальшого вдосконалення архітектури мобільних застосунків.

1.6 Висновки до розділу 1

Проведений огляд теоретичних напрацювань і практичних підходів до взаємодії мобільних систем із хмарною інфраструктурою дає змогу окреслити ключові тенденції та проблемні аспекти, що формують сучасний стан предметної області. Аналіз наявних рішень показав, що ефективність роботи мобільних застосунків у значній мірі залежить від того, наскільки раціонально організовано обмін даними між клієнтом і віддаленими сервісами, а також від здатності системи адаптуватися до мінливих умов мережевого середовища.

Встановлено, що традиційні архітектури забезпечують базовий рівень функціональності, проте часто не враховують динамічність мобільного використання, коливання навантаження та нестабільність мережевих підключень. Розвиток хмарних платформ сприяв появі розподілених і мультикомпонентних інфраструктур, які пропонують широкий спектр засобів для зберігання і

оброблення даних, однак разом із цим підвищив вимоги до узгодженості, безпеки та підтримки масштабованості.

Дослідження показало, що підвищення ефективності взаємодії можливе завдяки застосуванню спеціалізованих методів, серед яких важливу роль відіграють локальне кешування, алгоритми стискання інформації, використання кількох сервісів для розподілу навантаження, системи доставлення статичних ресурсів та комбіновані підходи, що поєднують локальну обробку з хмарними можливостями. Кожен із цих методів має свої особливості й рівень ефективності залежно від типу даних, архітектури застосунку та характеристик середовища, у якому він функціонує.

Огляд також засвідчив, що жоден метод не може забезпечити універсального вирішення всіх проблем. Найкращих результатів досягають системи, у яких поєднано кілька оптимізаційних технік, що дозволяє компенсувати недоліки кожного підходу окремо. Важливою тенденцією є перехід до адаптивних рішень, які здатні самостійно обирати найбільш ефективну стратегію залежно від поточних умов роботи.

Отже, узагальнення результатів розділу дає підстави стверджувати, що розроблення оптимізованої моделі взаємодії мобільного застосунку з хмарною інфраструктурою потребує комплексного підходу. Системи такого типу мають враховувати обмеження мобільних пристроїв, властивості мережевої інфраструктури, специфіку оброблюваних даних та економічні аспекти експлуатації. Сформована теоретична база є основою для подальшого системного аналізу та проєктування архітектури, що буде представлено у наступних розділах дослідження.

РОЗДІЛ 2. СИСТЕМНИЙ АНАЛІЗ ТА ФОРМУВАННЯ ТЕХНІЧНОГО ЗАВДАННЯ

2.1 Аналіз об’єкта дослідження та параметрів предметної області

Об’єктом дослідження у цій роботі є процес взаємодії мобільного застосунку з хмарними сервісами, що охоплює механізми передачі даних, оброблення запитів та забезпечення стабільного функціонування застосунку в умовах динамічних мережевих характеристик. Цей процес включає широкий комплекс технічних і логічних компонентів, які взаємодіють між собою та формують цілісну архітектуру системи. Для ґрунтовного дослідження необхідно проаналізувати параметри середовища, у якому працює мобільний клієнт, а також особливості хмарної інфраструктури, що забезпечує доступ до функціональних можливостей застосунку.

Мобільні пристрої, які виступають у ролі клієнтської частини системи, мають обмежені обчислювальні ресурси, залежність від заряду акумулятора та значні варіації продуктивності залежно від моделі чи поточного стану системи. Унаслідок цього взаємодія з хмарою повинна враховувати апаратні обмеження, можливість перебування користувача у середовищі з низькою якістю мережі та необхідність збереження прийнятного рівня швидкодії навіть у несприятливих умовах. Важливим параметром є здатність застосунку виконувати частину операцій локально для зменшення навантаження на мережу та зниження часу відгуку.

Хмарні сервіси, що складають серверну частину системи, характеризуються високим рівнем масштабованості, можливістю централізованого зберігання даних і здатністю виконувати складні обчислення. Проте їх робота залежить від загального навантаження, пропускної здатності мережевих каналів і затримок, що виникають під час передачі даних між клієнтом і віддаленими серверами. Хмарна інфраструктура може включати різні компоненти, такі як бази даних, аналітичні

модулі, системи балансування навантаження, служби безпеки та сервіси оброблення статичного контенту. Взаємодія з таким середовищем повинна бути узгодженою і стабільною, щоб забезпечити передбачувану поведінку застосунку на стороні користувача.

Параметри предметної області визначають межі системи та чинники, що впливають на її роботу. До них належать характеристики мережевої інфраструктури, обсяг даних, що обробляються, частота звернень до хмарних компонентів, вимоги до безпеки й необхідність забезпечення цілісності інформації. Окреме значення має аспект синхронізації станів між клієнтом та хмарою, оскільки невідповідність даних може призвести до порушення логіки роботи застосунку. Це особливо важливо в умовах, коли користувач здійснює дії офлайн, а зміни повинні бути передані в систему після відновлення зв’язку.

Зважаючи на різноманіття сценаріїв використання мобільних застосунків, система повинна адаптуватися до змін навантаження, що виникають унаслідок збільшення кількості користувачів або зміни патернів їх взаємодії. Тому під час аналізу об’єкта дослідження важливо враховувати здатність системи масштабуватися, розподіляти обчислення між різними компонентами та забезпечувати стійку роботу у випадках несподіваних пікових значень трафіку.

Таким чином, аналіз об’єкта дослідження дозволяє окреслити багаторівневу структуру взаємодії мобільного застосунку з хмарною платформою та визначити ключові параметри предметної області, що впливають на продуктивність, надійність і стабільність системи. Розуміння цих чинників є основою для подальшої постановки проблеми та обґрунтування необхідності впровадження оптимізаційних рішень, які будуть розглянуті у наступному підрозділі.

2.2 Формування функціональних і нефункціональних вимог до системи

Визначення вимог до системи є одним із ключових етапів її проєктування, оскільки саме на цьому етапі формується уявлення про те, які можливості має

забезпечувати розроблюваний застосунок та яким умовам повинна відповідати його робота. Функціональні та нефункціональні вимоги задають рамки архітектурних рішень, визначають принципи взаємодії між клієнтом і хмарною частиною, а також окреслюють очікувані характеристики системи з погляду продуктивності, надійності та безпеки.

Функціональні вимоги відображають поведінку системи й описують завдання, які вона повинна виконувати. У контексті оптимізованої взаємодії мобільного застосунку з хмарою важливо, щоб система підтримувала ефективну передачу даних, могла працювати з різними типами інформації та забезпечувала коректний обмін запитами між усіма компонентами.

Мобільний клієнт повинен бути здатним формувати запити, відправляти їх до хмарної інфраструктури й отримувати результати з врахуванням поточного стану мережі. Система має обробляти ці запити у спосіб, що зменшує затримки та забезпечує стабільність роботи застосунку навіть під час пікових навантажень.

До функціональних характеристик належить також здатність застосунку адаптуватися до зміни умов середовища. У випадку втрати зв’язку користувач повинен мати можливість продовжувати роботу у межах доступних локальних ресурсів, а після відновлення з’єднання всі зміни повинні бути коректно синхронізовані з хмарною платформою.

Важливою складовою є підтримка механізмів оброблення статичних і динамічних даних, алгоритмів їх оновлення, а також можливість взаємодії з різними сервісами, що виконують окремі функції у хмарній інфраструктурі.

Нефункціональні вимоги описують властивості системи, що забезпечують якість її роботи. Одним із головних параметрів є продуктивність: час реакції на запит повинен залишатися прийнятним навіть у ситуаціях, коли навантаження на хмарні ресурси збільшується.

Система повинна гарантувати стабільність процесів оброблення даних та мінімізувати затримки під час передачі інформації. Крім цього, особливе значення

має здатність системи працювати за умов обмеженої пропускної здатності, що характерно для мобільного середовища.

Ще однією важливою групою нефункціональних вимог є вимоги до безпеки. Оскільки обмін даними між клієнтом і хмарою відбувається через мережу, необхідно забезпечити шифрування переданої інформації, захист від несанкціонованого доступу та контроль цілісності даних.

Система повинна мати механізми автентифікації та авторизації, що гарантують доступ лише для легітимних користувачів. Важливо також враховувати потенційні загрози, що можуть виникати у хмарному середовищі, та забезпечувати стійкість до атак, спрямованих на порушення роботи сервісів.

Надійність і масштабованість також належать до ключових вимог. Система повинна працювати передбачувано за різних умов, зокрема під час тимчасових відхилень у роботі хмарних компонентів або у випадку підвищеного навантаження. Можливість масштабування є важливою умовою, оскільки кількість користувачів, обсяг даних та інтенсивність оброблення можуть суттєво змінюватися з часом. Архітектура має підтримувати плавне розширення ресурсів без суттєвої перебудови всієї системи.

Важливою нефункціональною вимогою є економічна ефективність, яка передбачає оптимальне використання хмарних ресурсів. Надмірна кількість операцій у хмарі або часті звернення до високовартісних сервісів можуть суттєво збільшити витрати. Тому система повинна забезпечувати баланс між обчисленнями, виконуваними локально, та завданнями, що передаються до хмари.

Таблиця 1 – Визначення функціональних та нефункціональних вимог

Тип вимог	**Назва вимоги**	**Опис вимоги**
Функціональні	Передача даних	Система повинна забезпечувати ефективний обмін даними між мобільним клієнтом і хмарною інфраструктурою з урахуванням поточного стану мережі.
Функціональні	Оброблення запитів	Застосунок має коректно

		формувати, передавати та обробляти запити без втрати даних і з мінімальними затримками.
Функціональні	Адаптація до середовища	Система повинна змінювати поведінку залежно від якості мережевого з’єднання та доступних ресурсів.
Функціональні	Офлайн-робота	У разі втрати зв’язку користувач має можливість продовжувати роботу в межах локальних ресурсів.
Функціональні	Оброблення даних	Система має підтримувати роботу як зі статичними, так і з динамічними даними, включаючи їх оновлення.
Функціональні	Взаємодія з сервісами	Застосунок повинен взаємодіяти з різними хмарними сервісами, що виконують окремі функції.
Нефункціональні	Продуктивність	Час реакції системи на запити повинен залишатися прийнятним навіть за умов підвищеного навантаження.
Нефункціональні	Стабільність	Система має забезпечувати безперервну та передбачувану роботу процесів оброблення даних.
Нефункціональні	Робота в обмеженій мережі	Застосунок повинен коректно функціонувати за умов низької пропускної здатності мережі.
Нефункціональні	Безпека	Передавання даних має здійснюватися з використанням механізмів шифрування, автентифікації та авторизації.
Нефункціональні	Надійність	Система повинна працювати коректно у випадку тимчасових збоїв у роботі хмарних компонентів.
Нефункціональні	Масштабованість	Архітектура має підтримувати плавне розширення ресурсів без необхідності суттєвої перебудови системи.

Узагальнюючи сформовані вимоги (таблиця 1), можна зазначити, що система має бути здатною підтримувати ефективний, безпечний і стабільний обмін даними між мобільним клієнтом і хмарною інфраструктурою, враховуючи різноманітні технічні обмеження та середовищні фактори.

Такий підхід дозволяє створити основу для подальшого проєктування оптимізованої моделі, що забезпечує високу якість роботи застосунку в умовах сучасного хмарного середовища.

2.3 Визначення припущень, обмежень та зовнішніх чинників

Під час проєктування системи, що оптимізує взаємодію мобільного застосунку з хмарною інфраструктурою, важливо окреслити набір припущень, які визначають очікувану модель функціонування, а також обмеження та зовнішні фактори, що здатні впливати на роботу системи. Ці елементи формують основу для побудови реалізовної архітектури та дають можливість оцінити межі застосування розробленого рішення.

Одним із ключових припущень є наявність доступу до стабільної хмарної інфраструктури, що забезпечує виконання серверної частини застосунку. Передбачається, що хмарна платформа підтримує необхідний набір сервісів для зберігання даних, оброблення запитів, масштабування ресурсів та виконання додаткових інструментів оптимізації. Вважається також, що провайдер гарантує певний рівень доступності та продуктивності, що дозволяє підтримувати безперервну роботу застосунку у типових сценаріях використання.

Ще одне важливе припущення полягає у тому, що мобільний пристрій користувача має достатні можливості для виконання локальних операцій, пов’язаних із обробленням частини даних, застосуванням оптимізаційних стратегій та взаємодією з кешем чи локальним сховищем. Розглядається, що пристрій здатний виконувати базові обчислення без істотного впливу на енергоспоживання, а також використовувати необхідні мережеві протоколи для передавання інформації.

Водночас система стикається з низкою обмежень, які можуть впливати на її роботу. Одним із них є мінливість параметрів мережі, що не завжди можуть бути контрольованими. Низька швидкість інтернету, періодичні обриви зв’язку, затримки під час передачі та перевантаження каналів створюють ситуації, у яких система повинна працювати у режимі зниженої доступності та використовувати внутрішні механізми адаптації. Додатковим обмеженням є ресурсні характеристики мобільних пристроїв, що можуть істотно відрізнятися залежно від моделі або стану батареї, що впливає на можливість виконання певних оптимізаційних алгоритмів.

Зовнішні чинники включають фактори, які не залежать безпосередньо від архітектури застосунку, але здатні впливати на його роботу. До таких чинників належать коливання навантаження на хмарні сервіси, що може спричиняти затримки у виконанні запитів, а також зміни в політиках провайдера, які можуть впливати на доступність певних функцій або формувати нові фінансові вимоги. Окремо слід виділити безпекові аспекти, оскільки хмарне середовище може зазнавати зовнішніх атак, що спричинюють тимчасове зниження продуктивності або обмеження у роботі сервісів.

Ще одним важливим чинником є поведінка користувачів, оскільки саме вони визначають частоту звернень, динаміку навантаження і характер взаємодії із системою. Нестандартні сценарії використання, раптове збільшення кількості користувачів або нетипові шаблони поведінки можуть впливати на ресурсні потреби та вимагати гнучкого масштабування.

Таким чином, визначення припущень, обмежень і зовнішніх факторів дозволяє сформувати реалістичну модель функціонування системи та встановити межі її застосування. Чітке розуміння цих аспектів дає можливість розробити архітектуру, здатну ефективно працювати в умовах змінного навантаження, нестабільної мережі та різноспрямованих впливів зовнішнього середовища. Ці положення створюють підґрунтя для подальшого обґрунтування вибраних

методів оптимізації та переходу до проєктування системи у практичній частині роботи.

2.4 Підсумки системного аналізу

У межах другого розділу було здійснено системний аналіз взаємодії мобільного застосунку з хмарною інфраструктурою, що дозволило сформувати цілісне уявлення про структуру об’єкта дослідження, властивості предметної області та фактори, які впливають на ефективність роботи системи. Розглянуті характеристики демонструють, що взаємодія між клієнтською частиною та хмарними сервісами є багаторівневим процесом, у якому поєднуються апаратні, програмні та мережеві компоненти.

Проведений аналіз показав, що ефективність роботи мобільного застосунку суттєво залежить від умов середовища, у якому він функціонує. Цей процес визначається стабільністю мережевого каналу, швидкістю доступу до хмарних ресурсів, затримками під час оброблення запитів, можливостями локального пристрою та рівнем навантаження на віддалені сервіси.

Установлено, що формування вимог до системи відіграє ключову роль у подальшому проєктуванні архітектури. Серед функціональних характеристик визначальними є можливість ефективного обміну даними, підтримка локальної обробки та здатність синхронізувати зміни після відновлення мережевого доступу. Нефункціональні вимоги охоплюють продуктивність, безпеку, масштабованість, стійкість до зовнішніх впливів та економічну доцільність використання хмарних ресурсів. Важливим результатом системного аналізу є окреслення припущень і обмежень, що задають межі застосування майбутнього рішення. Нестабільність мережевого середовища, можливі перевантаження хмарних сервісів та безпекові ризики визначають складні умови, у яких повинна функціонувати система.

Узагальнюючи виявлені аспекти, можна зробити висновок, що побудова оптимізованої моделі взаємодії вимагає комплексного підходу, який передбачає урахування технічних, організаційних та середовищних чинників.

РОЗДІЛ 3. МЕТОДОЛОГІЧНІ ПІДХОДИ ТА ВИБІР ТЕХНОЛОГІЧНИХ РІШЕНЬ

3.1 Аналіз існуючих рішень

Підбір предметної сфери, у межах якої можливо наочно показати результативність оптимізаційних підходів, має вирішальне значення для обґрунтування їх практичної корисності. Застосування таких стратегій спрямоване на зменшення серверного навантаження, оптимізацію використання мережевого трафіку та підвищення швидкодії програмних рішень.

Однією з перспективних галузей для реалізації зазначених підходів є мобільні застосунки, орієнтовані на колекціонерів і тематичних ентузіастів, які активно використовують хмарні сервіси для зберігання, аналізу та ідентифікації об’єктів колекціонування.

Серед прикладів таких рішень можна виокремити Coinoscope, Classifier: Collection Tracker та Oeni, які реалізують різні підходи до роботи з даними та взаємодії з віддаленими ресурсами. Незважаючи на відмінності у функціональному призначенні, ці застосунки мають спільну рису – залежність від ефективної обробки мультимедійної інформації та доступу до зовнішніх баз даних.

Coinoscope орієнтований на потреби нумізматів і забезпечує автоматичне визначення монет шляхом аналізу зображень та порівняння їх із записами у спеціалізованих базах. Основною перевагою такого підходу є спрощення процесу

ідентифікації, що дозволяє користувачам оперативно отримувати відомості про походження, рідкісність і орієнтовну цінність монет.

Водночас функціонування застосунку значною мірою залежить від якості інтернет-з’єднання, оскільки обчислювальні алгоритми та інформаційні ресурси розміщені на серверній стороні, що може спричиняти затримки за умов інтенсивного використання.

Застосунок Oeni спеціалізується на розпізнаванні винних етикеток і формуванні детального опису напоїв. Алгоритми обробляють фотознімки пляшок і визначають характеристики вина, зокрема сорт винограду, виробника та користувацькі оцінки. Висока точність результатів досягається завдяки постійній синхронізації з онлайновими винними каталогами.

Разом із цим накопичення історії пошуків і персоналізованих рекомендацій потребує значних обсягів даних, що підвищує вимоги до серверних ресурсів і зумовлює необхідність застосування ефективних механізмів кешування та стискання.

Classifier: Collection Tracker призначений для універсального обліку різних типів колекцій. Застосунок надає можливість додавати медіаматеріали, структурувати записи за категоріями та отримувати узагальнену аналітику. Важливою перевагою є підтримка автономного режиму, що зменшує залежність від постійного доступу до мережі та знижує навантаження на серверну частину. Водночас процес синхронізації між пристроями може бути ускладнений через обмежені можливості інтеграції з хмарними платформами, що актуалізує потребу в оптимізації механізмів оновлення інформації.

На основі проведеного аналізу доцільно подати узагальнене порівняння розглянутих застосунків, яке відображає їхні функціональні особливості та підходи до оброблення даних.

Усі розглянуті застосунки поєднують спільні характеристики, серед яких оброблення мультимедійної інформації, робота з базами даних і необхідність збереження персоналізованих відомостей користувачів.

Оскільки такі системи активно використовують фотографії, текстові описи та мережеві сервіси, вони є доцільним прикладом для дослідження методів оптимізації взаємодії з хмарною інфраструктурою. Реалізація відповідних підходів дозволяє скоротити витрати на обчислювальні ресурси, підвищити швидкість доступу до інформації та забезпечити стабільну роботу навіть у разі нестабільного підключення.

Таблиця 2 – Існуючі рішення

Критерій	**Coinoscope**	**Classifier: Collection Tracker**	**Oeni**
Основне призначення	Ідентифікація та оцінювання монет	Облік і керування колекціями	Розпізнавання вин та довідкова інформація
Тип оброблюваних даних	Зображення	Текстові та медіадані	Зображення
Підтримка офлайн-режиму	Обмежена	Наявна	Обмежена
Швидкість оброблення запитів	Висока	Висока	Висока
Робота з медіаданими	Аналіз і компресія зображень	Збереження зображень	Аналіз і компресія зображень
Основні переваги	Швидка ідентифікація, точність	Автономність, гнучке налаштування	Висока точність, персоналізація
Основні недоліки	Залежність від мережі	Обмежена хмарна інтеграція	Значне серверне навантаження
Цільова аудиторія	Нумізмати	Колекціонери різних напрямів	Винні ентузіасти, сомельє

Разом із цим слід зазначити, що більшість наведених застосунків мають вузьку спеціалізацію, що обмежує сферу їх використання. З огляду на це, у межах подальшої розробки доцільно орієнтуватися на концепцію, подібну до Classifier: Collection Tracker, яка забезпечує більш повне охоплення предметної області та створює ширші можливості для впровадження оптимізованих хмарних рішень.

3.2 Аналіз інструментів для реалізації мобільного застосунку для взаємодії з хмарними сервісами

Початковим етапом створення мобільного застосунку, призначеного для оптимізованої роботи з хмарними сервісами, є обґрунтований вибір технологічного стеку, який визначає подальшу архітектуру системи та можливості її масштабування. У межах даного проєкту доцільним є застосування фреймворку React Native у поєднанні з платформою Expo, оскільки така комбінація забезпечує розробку кросплатформених рішень на основі єдиної кодової бази. Це дозволяє швидко створювати кросплатформені мобільні додатки з єдиним кодовою базою, що зменшує витрати на розробку та спрощує підтримку [6]. Платформа Expo надає широкий набір готових інструментів і програмних інтерфейсів, що зменшує складність налаштування середовища розробки та прискорює інтеграцію зовнішніх бібліотек і сервісів.

Під час вибору хмарного сховища серед поширених рішень, таких як AWS, Microsoft Azure та Firebase, пріоритет було надано Firebase з огляду на його зручну сумісність із React Native та Expo, що забезпечується наявністю спеціалізованих SDK і бібліотек [2]. Даний сервіс пропонує гнучкі механізми зберігання даних, відносну простоту конфігурації та прийнятну цінову політику, що є особливо важливим на початкових етапах розробки. Безкоштовний тариф дозволяє використовувати базові можливості платформи без додаткових фінансових витрат. Крім того, аналітичні інструменти Firebase надають засоби візуального контролю використання ресурсів, що спрощує аналіз продуктивності та керування навантаженням. Водночас слід враховувати обмеження безкоштовного плану, можливі затримки при роботі з великими масивами інформації та відсутність розширених механізмів оптимізації мультимедійних файлів.

З метою зменшення негативного впливу зазначених обмежень до архітектури системи інтегрується сервіс Cloudinary, орієнтований на оброблення

та оптимізацію медіа-контенту. Він забезпечує автоматичне стиснення зображень, використання мережі доставлення контенту та адаптивне завантаження файлів залежно від параметрів пристрою [1]. Застосування цього сервісу дозволяє знизити навантаження на основне хмарне сховище та прискорити доставлення мультимедійних ресурсів кінцевим користувачам. Додатковою перевагою Cloudinary є можливість трансформації графічних і відеофайлів у форматах, оптимальних для різних розмірів екранів і роздільних здатностей.

Для оптимізації взаємодії між клієнтською частиною застосунку та хмарними платформами передбачається впровадження проміжного серверного рівня на базі Vercel. Його функціональність включає генерацію токенів доступу для Cloudinary та Firestore, оброблення запитів і безпечне зберігання конфіденційних ключів. Такий підхід дозволяє уникнути прямого звернення мобільного клієнта до хмарних API, що підвищує рівень захисту даних та забезпечує кращий контроль за використанням ресурсів.

З метою підвищення ефективності роботи застосунку за відсутності мережевого підключення використовується локальна база даних SQLite для React Native. Це легковагове рішення з повною підтримкою мобільних платформ iOS та Android, яке дозволяє реалізувати гібридну модель зберігання інформації. Часто використовувані дані зберігаються локально, а після відновлення доступу до мережі синхронізуються з Firebase, що позитивно впливає на зручність користування. SQLite характеризується високою швидкістю виконання запитів, не потребує окремого серверного середовища та має просту конфігурацію, що робить його доцільним вибором для мобільних застосунків.

Таким чином, поєднання React Native з Expo, використання Firebase як основного хмарного сховища, інтеграція Cloudinary для оптимізації мультимедійних даних, застосування Vercel як проміжного серверного компонента та впровадження SQLite для локального збереження формують гнучку й продуктивну архітектуру. Запропоноване рішення забезпечує швидкий доступ до даних, ефективне управління медіа-контентом та зменшення затримок

під час взаємодії з хмарними сервісами, що є критично важливим для мобільних застосунків подібного типу.

3.3 Визначення сценаріїв функціонування оптимізованої системи

Функціонування оптимізованої системи взаємодії мобільного застосунку з хмарними сервісами реалізується через сукупність типових сценаріїв, які відображають поведінку системи в різних умовах використання. Опис таких сценаріїв дозволяє формалізувати логіку роботи застосунку, визначити послідовність дій користувача та системи, а також оцінити ефективність обраних методів оптимізації в реальних експлуатаційних ситуаціях.

Базовим сценарієм є стандартна взаємодія користувача із застосунком за умови стабільного мережевого з’єднання. У цьому випадку мобільний клієнт ініціює запит, який обробляється проміжним сервером та передається до відповідних хмарних сервісів. Дані, що потребують зберігання або складного аналізу, обробляються у хмарі, після чого результат повертається до клієнтської частини. Оптимізація досягається за рахунок асинхронного обміну даними, мінімізації обсягу переданої інформації та використання оптимізованих API-запитів, що забезпечує швидку реакцію інтерфейсу та стабільну роботу системи.

Іншим важливим сценарієм є використання застосунку в умовах нестабільного або відсутнього інтернет-з’єднання. У такій ситуації система переходить до режиму локальної обробки, використовуючи можливості вбудованого сховища на основі SQLite. Користувач має змогу виконувати основні дії без постійного доступу до мережі, при цьому всі зміни фіксуються у локальній базі даних. Після відновлення підключення система автоматично ініціює процес синхронізації, передаючи накопичені дані до хмарного сховища та узгоджуючи стан між клієнтом і серверною частиною.

Окремий сценарій пов’язаний із роботою з медіа-контентом, зокрема зображеннями або іншими мультимедійними файлами. У цьому випадку

мобільний застосунок передає медіа-дані до спеціалізованого сервісу оптимізації, який виконує автоматичну компресію та підготовку файлів до зберігання або подальшого використання. Завдяки цьому зменшується навантаження на основне хмарне сховище та скорочується час завантаження контенту для кінцевого користувача. Сценарій передбачає адаптивне підлаштування параметрів зображень залежно від типу пристрою та характеристик екрана.

Ще одним сценарієм є оброблення пікових навантажень, коли кількість одночасних звернень до системи суттєво зростає. У таких умовах оптимізована архітектура забезпечує розподіл запитів між хмарними компонентами та зберігає стабільність роботи завдяки масштабованості серверної частини. Мобільний клієнт продовжує отримувати відповіді без значного збільшення часу очікування, що свідчить про ефективність використаних методів оптимізації.

Рисунок 3.1 – Сценарії функціонування системи

Таким чином, визначені сценарії функціонування (рисунок 3.1) демонструють здатність системи адаптуватися до різних умов експлуатації, забезпечуючи безперервну роботу, високу продуктивність і зручність для користувача. Описані сценарії слугують основою для подальшого тестування, аналізу результатів і оцінювання ефективності реалізованих рішень у практичній частині дослідження.

3.4 Розроблення архітектурної концепції мобільного застосунку

Архітектурна концепція мобільного застосунку формується з урахуванням необхідності забезпечення ефективної, безпечної та масштабованої взаємодії з хмарними сервісами, а також адаптації до обмежень мобільного середовища. Основною ідеєю запропонованої архітектури є поєднання локальних і хмарних компонентів у межах єдиної гібридної моделі, що дозволяє оптимально розподіляти навантаження між клієнтською та серверною частинами системи.

Клієнтська частина застосунку відповідає за взаємодію з користувачем, формування запитів і попередню обробку даних. Вона реалізує логіку інтерфейсу, локальне зберігання часто використовуваної інформації та механізми роботи в умовах нестабільного або відсутнього мережевого підключення. Такий підхід дозволяє зменшити кількість звернень до хмарних сервісів і забезпечити безперервність роботи застосунку навіть за несприятливих умов експлуатації.

Серверна частина архітектури виконує роль проміжної ланки між мобільним клієнтом і зовнішніми хмарними платформами. Вона відповідає за оброблення запитів, управління доступом до ресурсів, виконання операцій автентифікації та зберігання конфіденційних даних. Використання проміжного серверу дозволяє ізолювати клієнтський застосунок від прямої взаємодії з хмарними API, що підвищує рівень безпеки та спрощує контроль за використанням ресурсів.

Хмарна інфраструктура використовується для централізованого зберігання даних, аналітики та оброблення інформації, яка потребує значних обчислювальних ресурсів. Вона забезпечує масштабованість системи та можливість обслуговування великої кількості користувачів без суттєвого погіршення продуктивності. Окрему роль у архітектурі відіграють сервіси оптимізації медіа-контенту, які дозволяють зменшувати обсяг передаваних даних і прискорювати доставку ресурсів кінцевим користувачам.

Важливим елементом архітектурної концепції є механізм синхронізації даних між локальним сховищем і хмарною частиною системи. Усі зміни, виконані користувачем у режимі офлайн, фіксуються на стороні клієнта та передаються до хмари після відновлення з’єднання. Такий підхід забезпечує цілісність інформації та узгодженість станів між різними компонентами системи.

Запропонована архітектура передбачає використання асинхронної моделі взаємодії, що дозволяє уникати блокування інтерфейсу користувача під час виконання мережевих операцій. Це позитивно впливає на швидкодію застосунку та загальне сприйняття системи користувачем. Крім того, архітектура орієнтована на модульність, що спрощує розширення функціоналу та подальшу підтримку програмного продукту.

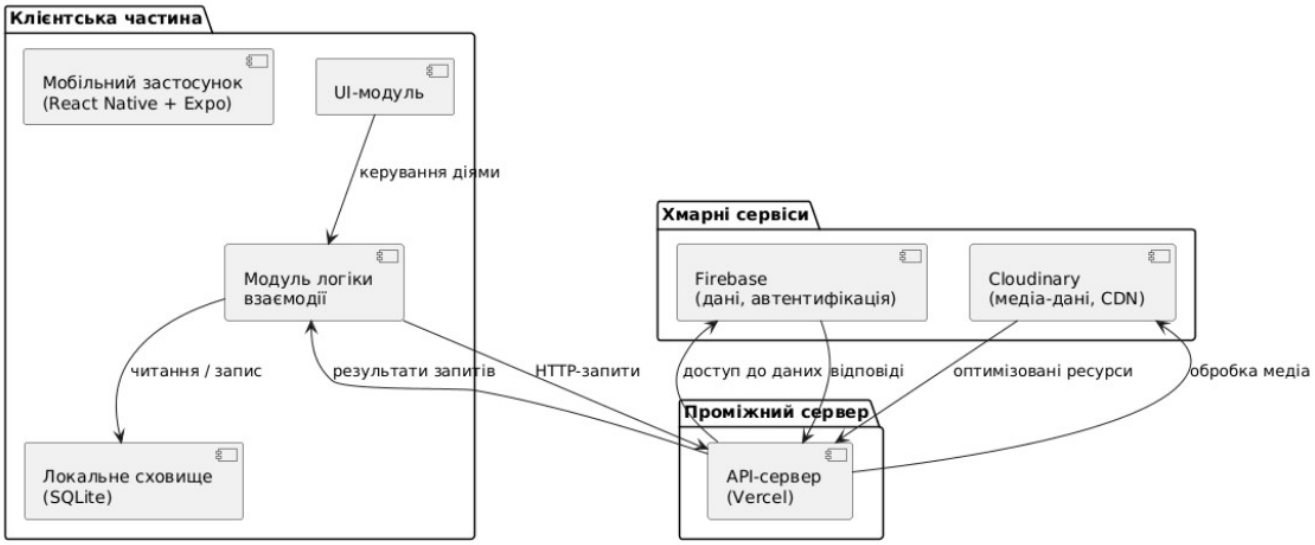

Рисунок 3.2 – Архітектурна концепція застосунку

Таким чином, розроблена архітектурна концепція мобільного застосунку (рисунок 3.2) ґрунтується на принципах гібридності, розподілу відповідальності між компонентами та адаптивності до змінних умов експлуатації. Поєднання локального зберігання, проміжного серверу та хмарних сервісів створює передумови для підвищення продуктивності, зменшення затримок і раціонального використання ресурсів, що є ключовим фактором для ефективної реалізації оптимізованої взаємодії з хмарною інфраструктурою.

3.5 Висновки щодо вибраних методів та засобів

У межах третього розділу було обґрунтовано вибір методів, інструментів і архітектурних рішень, спрямованих на підвищення ефективності взаємодії мобільного застосунку з хмарними сервісами.

Проведений аналіз методів вирішення проблеми показав доцільність поєднання локальної обробки даних із хмарними можливостями. Такий підхід дає змогу зменшити залежність застосунку від стабільності мережевого з’єднання, скоротити час відгуку та знизити навантаження на серверну інфраструктуру.

У розділі також визначено та аргументовано вибір програмних і хмарних засобів, які відповідають поставленим вимогам. Застосування кросплатформеного підходу у поєднанні з хмарними сервісами зберігання даних, інструментами оптимізації медіа-контенту та локальними механізмами кешування створює основу для побудови гібридної архітектури. Розроблена архітектурна концепція мобільного застосунку відображає взаємозв’язок між клієнтською частиною, проміжним сервером і хмарними сервісами, а також демонструє розподіл відповідальності між компонентами системи. Визначені сценарії функціонування підтверджують здатність системи коректно працювати в онлайн- та офлайн-режимах, забезпечуючи синхронізацію даних і стабільність оброблення запитів у різних умовах експлуатації.

Отже, результати третього розділу формують завершену методичну та технологічну основу для переходу до практичної реалізації розробленої системи. Запропоновані рішення створюють передумови для побудови продуктивного мобільного застосунку з оптимізованою взаємодією з хмарними сервісами, що буде детально розглянуто та реалізовано у наступному розділі роботи.

РОЗДІЛ 4. РОЗРОБЛЕННЯ ТА ОЦІНЮВАННЯ ОПТИМІЗОВАНОЇ ВЗАЄМОДІЇ МОБІЛЬНОГО ЗАСТОСУНКУ З ХМАРНИМИ СЕРВІСАМИ

4.1. Архітектура і дизайн додатку

Архітектурне рішення мобільного застосунку для підтримки процесів колекціонування ґрунтується на засадах компонентної побудови, гнучкості масштабування та можливості подальшого розвитку функціональних можливостей. Ключовою ідеєю обраного підходу є розподіл системи на окремі логічні модулі, кожен із яких виконує визначене коло завдань і може модифікуватися незалежно від інших елементів. Така організація дозволяє впроваджувати зміни або замінювати окремі сервіси, зокрема інструменти оброблення зображень чи механізми зберігання даних, без необхідності суттєвого втручання в загальну структуру застосунку (рисунок 4.1). Здатність системи адаптуватися до зростання кількості користувачів і обсягів інформації забезпечується використанням хмарних платформ, які автоматично регулюють ресурси відповідно до поточного навантаження.

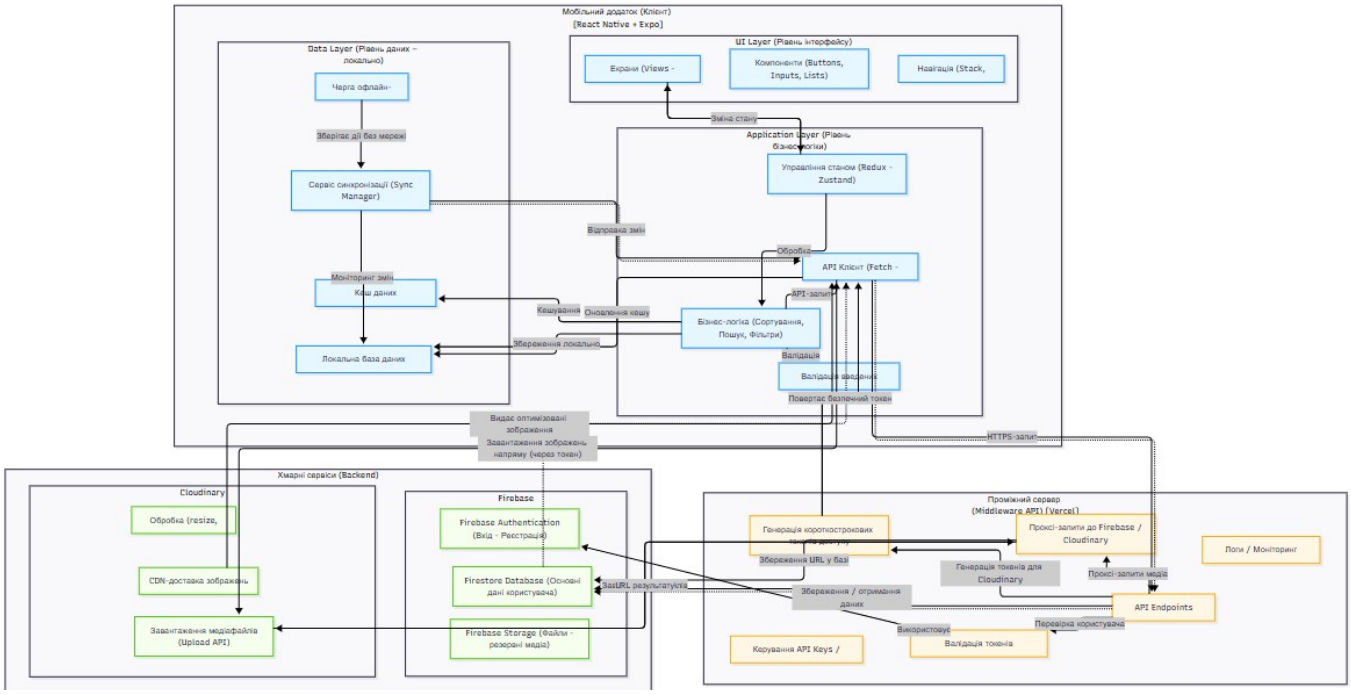

Рисунок 4.1 – Загальна архітектура мобільного додатку для колекціонерів

Головним призначенням архітектури є забезпечення ефективного опрацювання колекційних об’єктів різних типів, мінімізація затримок під час взаємодії з віддаленими сервісами та підтримка стабільної роботи застосунку за різних умов доступу до мережі. Особлива увага приділяється можливості функціонування в автономному режимі, що є критично важливим для мобільних сценаріїв використання.

Загальна побудова системи реалізована відповідно до клієнт–серверної парадигми. У межах цієї моделі мобільний застосунок виступає клієнтським компонентом, відповідальним за відображення інформації, введення даних і взаємодію з користувачем. Серверна складова має розподілений характер і представлена сукупністю хмарних сервісів, а також окремим проміжним сервером, який виконує функції посередника між клієнтом і зовнішніми платформами. Така організація забезпечує централізоване збереження інформації, необхідне для реалізації механізмів синхронізації та доступу до персональних колекцій з різних пристроїв.

Архітектурна модель передбачає виділення трьох основних логічних рівнів. Рівень користувацького інтерфейсу відповідає за візуальне представлення даних і

взаємодію з кінцевим користувачем. Його реалізація здійснюється з використанням кросплатформенного фреймворку, що дозволяє застосовувати єдину програмну основу для різних мобільних операційних систем. Основний акцент зроблено на зручність навігації, зрозумілу структуру елементів та можливість ефективного керування великими обсягами колекційної інформації.

Рівень прикладної логіки реалізує ключові функції застосунку та забезпечує оброблення дій користувача. Саме на цьому рівні виконуються операції створення, редагування та впорядкування елементів колекцій, а також реалізуються механізми пошуку, фільтрації та сортування. Цей компонент виконує роль зв’язувальної ланки між інтерфейсом і рівнем даних, забезпечуючи коректність операцій і логічну послідовність оброблення інформації.

Рівень даних відповідає за зберігання, кешування та синхронізацію інформації. Він поєднує локальне сховище, яке використовується для забезпечення автономної роботи та швидкого доступу до часто використовуваних даних, із хмарними сервісами, призначеними для довготривалого зберігання та оброблення медіаконтенту. Комунікація між клієнтською частиною та віддаленими сервісами здійснюється через захищені інтерфейси прикладного програмування, що гарантує цілісність даних і контроль доступу до ресурсів.

Важливою особливістю обраної архітектури є застосування гібридної моделі зберігання інформації, яка поєднує локальні й хмарні ресурси. Такий підхід дозволяє користувачеві працювати з колекціями без постійного доступу до мережі, фіксуючи всі зміни на пристрої. Після відновлення підключення система автоматично виконує синхронізацію з віддаленим сховищем, зберігаючи узгодженість даних.

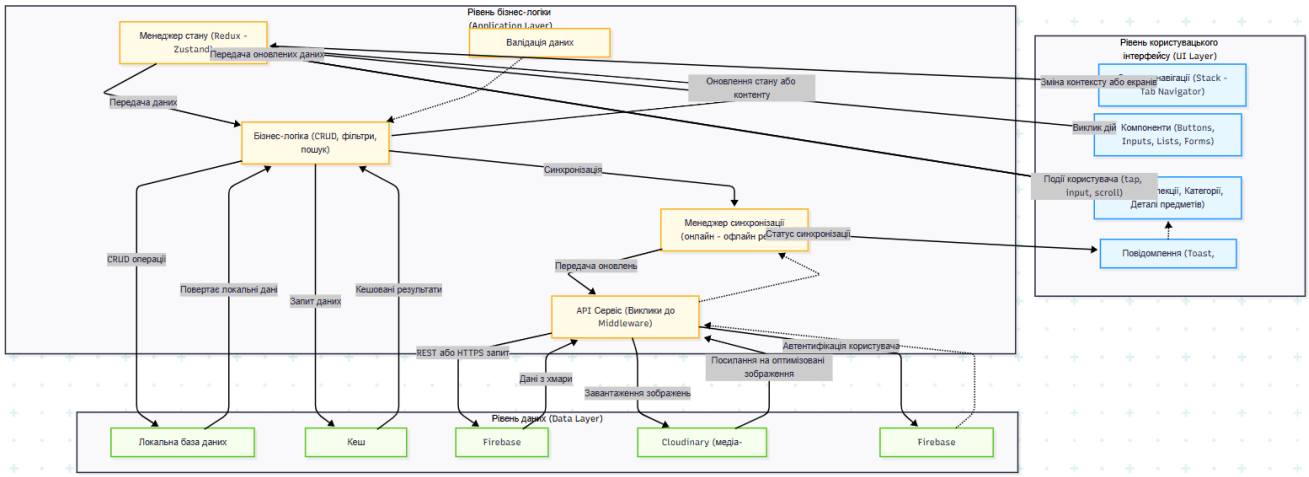

Рисунок 4.2 – Взаємодія між рівнями архітектури додатку

Чіткий розподіл відповідальності між рівнями архітектури спрощує управління функціоналом і створює умови для подальшого розвитку продукту (рисунок 4.2). Завдяки визначеним межам між модулями можливо незалежно оновлювати окремі компоненти, наприклад інтерфейс користувача, не порушуючи стабільність роботи логічної або даної складової.

З погляду візуального оформлення застосунок орієнтований на стриманий і функціонально вивірений дизайн. Такий підхід дозволяє забезпечити зрозумілу ієрархію елементів, логічну навігацію та швидкий доступ до основних можливостей. Проєктування інтерфейсу здійснюється з урахуванням принципів, орієнтованих на потреби користувача, що передбачає мінімізацію кількості дій для виконання завдань і адаптацію до різних розмірів екранів.

Отже, запропонована архітектура мобільного застосунку для колекціонерів поєднує перевірені підходи клієнт–серверної взаємодії з перевагами гібридного зберігання та модульної організації. Таке рішення не лише відповідає технічним вимогам, але й забезпечує реалізацію дослідницьких цілей роботи, створюючи основу для подальшого масштабування та вдосконалення програмного продукту.

4.2. Використані інструменти і технології

Для створення мобільного застосунку, орієнтованого на ведення та управління колекціями, було підібрано сучасний набір технологічних рішень, який забезпечує універсальність платформи, стабільність функціонування, захист даних і готовність системи до подальшого розширення. Клієнтська частина реалізована з використанням фреймворку React Native, що надає можливість розробляти програмні продукти одночасно для мобільних платформ Android та iOS на основі спільної кодової бази. Такий підхід істотно скорочує часові витрати на реалізацію проєкту та спрощує подальшу підтримку завдяки уніфікованій структурі застосунку.

Для оптимізації процесів збірки, налагодження та доступу до нативних можливостей мобільних пристроїв було використано середовище Expo. Воно надає широкий набір інструментів для швидкого запуску проєкту, тестування функціональності та інтеграції сторонніх бібліотек без необхідності складного конфігурування середовища розробки. Це особливо важливо на етапах прототипування та ітеративного вдосконалення продукту.

Як основну мову програмування обрано TypeScript, який є розширенням JavaScript із підтримкою статичної типізації. Його застосування дозволяє підвищити надійність програмного коду, полегшити процес виявлення помилок і зменшити кількість логічних неточностей ще до виконання програми. Чітка типізація та структурованість сприяють кращій читабельності коду й спрощують супровід проєкту, що є важливим чинником для систем, розрахованих на довготривалу експлуатацію та розвиток.

Зберігання й оброблення основних користувацьких даних реалізовано за допомогою хмарної платформи Firebase Firestore, яка виконує роль центрального сховища. Даний сервіс підтримує роботу зі структурованими даними, забезпечує синхронізацію у режимі близькому до реального часу та надає вбудовані механізми автентифікації. Для підтримки повноцінної роботи без постійного доступу до мережі в застосунку використовується локальна база даних SQLite. Вона дозволяє зберігати інформацію безпосередньо на пристрої користувача та

виконувати автоматичне узгодження змін із хмарним сховищем після відновлення інтернет-з’єднання, що суттєво підвищує зручність використання.

Оброблення медіаконтенту, зокрема фотографій об’єктів колекцій, здійснюється за допомогою сервісу Cloudinary. Його функціональні можливості включають автоматичне стискання зображень, адаптацію розмірів під різні екрани та швидку доставку файлів завдяки використанню мережі CDN. Застосування цього сервісу дозволяє знизити навантаження на основне хмарне сховище та оптимізувати споживання мережевого трафіку, що позитивно впливає на швидкодію мобільного застосунку.

Важливою складовою системи є проміжний сервер, розгорнутий на платформі Vercel. Він виконує функцію зв’язувальної ланки між клієнтським застосунком і зовнішніми хмарними сервісами, відповідаючи за оброблення запитів, контроль доступу, перевірку коректності даних та зберігання чутливої інформації, зокрема ключів доступу. Взаємодія між компонентами системи реалізується через REST API із використанням формату JSON, що забезпечує стандартизований та надійний обмін даними.

Розробка й тестування програмного продукту здійснювалися в середовищі Visual Studio Code, яке підтримує роботу з TypeScript і React Native та легко інтегрується із системами контролю версій. Керування вихідним кодом і спільна робота над проєктом реалізовувалися за допомогою платформи GitHub. Для організації робочих процесів і контролю виконання завдань застосовувався сервіс Trello, що дозволяє впроваджувати підхід Kanban та забезпечує прозоре планування етапів розробки. Первинне тестування функціональності виконувалося через додаток Expo Go, який надає можливість перевіряти роботу застосунку на реальних мобільних пристроях без попередньої компіляції.

Отже, використання обраного набору технологій формує цілісне середовище для створення мобільного застосунку, яке поєднує продуктивність, надійність, безпеку та гнучкість. Комбінація React Native, Expo, TypeScript, Firebase, SQLite, Cloudinary та Vercel забезпечує узгоджену взаємодію між усіма

компонентами системи та відповідає сучасним вимогам до розробки високоефективних мобільних програмних продуктів.

4.3. Опис структури проекту

Структура проекту програмного додатку (кодова база) розроблена відповідно до принципів інженерії програмного забезпечення, а саме Моделі-Вигляду-Контролера (Model-View-Controller, MVC) та Чистої Архітектури (Clean Architecture). Метою є досягнення високої згуртованості (High Cohesion) в межах кожного модуля та мінімізація зв'язаності (Low Coupling) між ними, що критично важливо для забезпечення підтримуваності (Maintainability) та розширюваності (Extensibility) системи.

Файлова організація проекту, реалізована на TypeScript з використанням фреймворку React Native та середовища Expo, логічно відображає трирівневу архітектуру (рисунок 4.3).

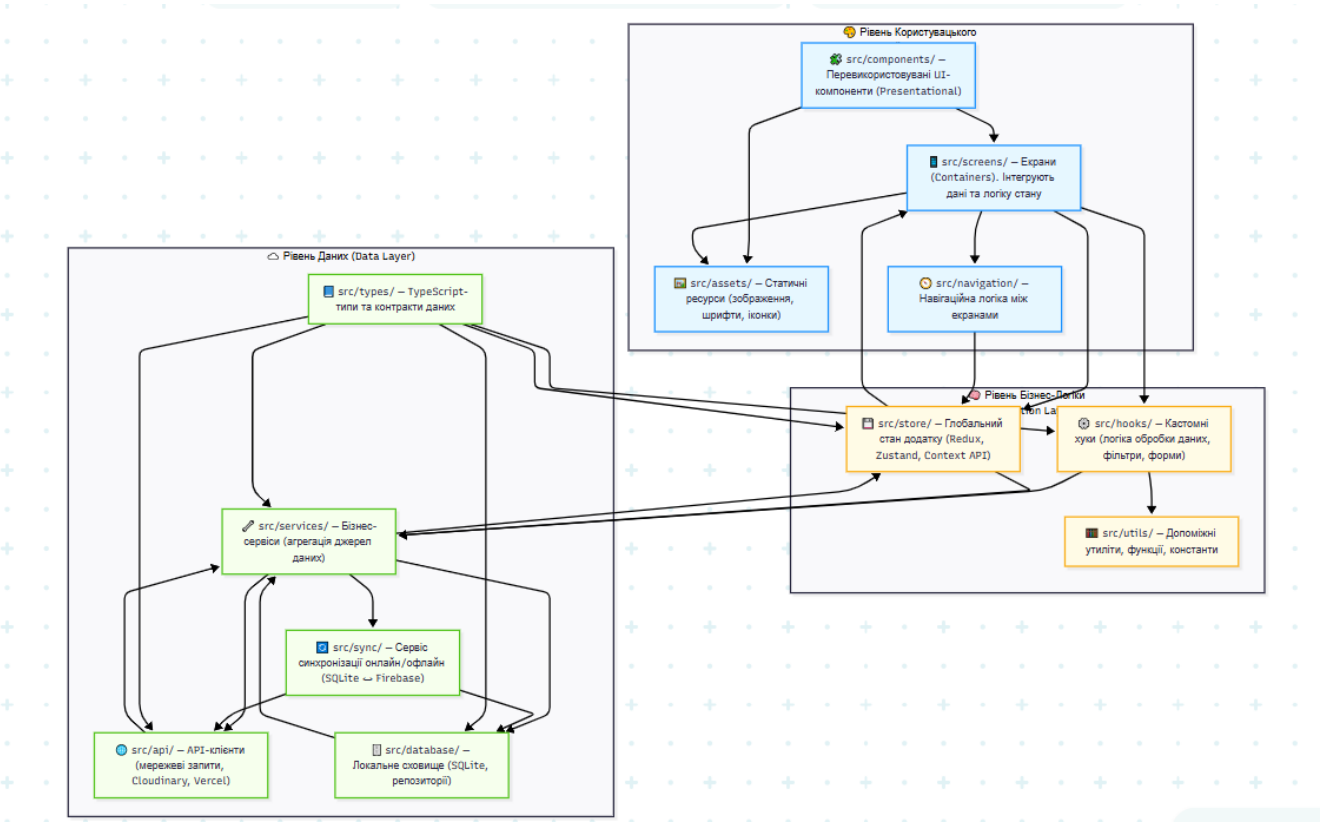

Рисунок 4.3 – Файлова структура проекту

Уся функціональність інкапсульована у директорії src/, яка декомпозується на наступні ключові модулі:

1.Модулі Рівня Користувацького Інтерфейсу – цей рівень відповідає за презентацію даних користувачеві та реєстрацію його взаємодії. Його організація базується на компонентному підході.

Таблиця 3 – Файлова структура рівня користувацького інтерфейсу

Директорія	Призначення	Взаємодія
src/screens/	Екрани (Containers). Містить компоненти найвищого рівня, які інтегрують дані та логіку стану, забезпечуючи загальну структуру екрану (наприклад, HomeScreen.tsx, CollectionDetailsScreen.tsx). Вони є контролерами представлення.	Звертаються до src/store/ для отримання даних та до src/hooks/ для ініціації бізнес-операцій.
src/components/	Перевикористовувані Компоненти (Presentational Components). Атомарні та молекулярні елементи інтерфейсу (кнопки, картки, форми), які є "чистими" функціями – вони отримують дані та колбеки через props і не містять бізнес-логіки.	Залежать лише від src/assets/ та зовнішніх UI-бібліотек.
src/assets/	Статичні Ресурси. Зберігання зображень, іконок, шрифтів та інших медіа-елементів, які не є динамічними даними.	Є залежністю для src/components/ та src/screens/.
src/navigation/	Навігаційна Логіка. Містить конфігурації навігаційних стеків та таб-меню, що керують переходом між src/screens/.	Взаємодіє з src/screens/ та використовує дані з src/store/ для умовної навігації (наприклад, перенаправлення на екран входу).

2.Модулі Рівня Бізнес-Логіки – цей рівень є центральним хабом, що реалізує доменну логіку та керує потоком даних між UI та Data Layers.

Таблиця 4 – Файлова структура рівня бізнес-логіки

Директорія	Призначення	Взаємодія
src/store/	Управління Глобальним Станом. Централізоване сховище стану (наприклад, Redux, Zustand, або Context API), що містить Державні моделі (State Models) (дані користувача, фільтри, поточний статус завантаження).	Надає дані для src/screens/ і викликає функції з src/services/ для оновлення цих даних.
src/hooks/	Кастомні Хуки. Абстрагує складну логіку роботи зі станом та даними, роблячи її перевикористовуваною у різних екранах. Сюди включена логіка сортування, фільтрації та обробки форм.	Використовують src/services/ та src/utils/.
src/utils/	Допоміжні Функції. Містить незалежні, чисті функції загального призначення, які не стосуються безпосередньо UI чи Data (наприклад, валідація, форматинг, константи).	Зменшує дублювання коду в інших модулях.

3.Модулі Рівня Даних – цей рівень відповідає за доступ до джерел даних та управління їхнім життєвим циклом (локальне кешування, синхронізація, віддалений доступ).

Таблиця 5 – Файлова структура рівня користувацького інтерфейсу

Директорія	Призначення	Взаємодія
src/api/	Клієнти API. Інкапсулює функції для здійснення мережевих запитів до Проміжного сервера (Vercel) та	Викликається модулями з

	прямого завантаження медіафайлів у Cloudinary. Забезпечує стандартизований обмін даними через API-протокол.	src/services/ та src/sync/.
src/database/	Локальне Сховище (SQLite). Містить Репозиторії (Repositories), що абстрагують пряму роботу з SQL-запитами, надаючи Application Layer об'єктно-орієнтований інтерфейс (наприклад, itemRepository.save(item)).	Є першим джерелом даних для src/services/.
src/services/	Сервіси Даних (Data Services). Модулі, що реалізують бізнес-процеси, які вимагають взаємодії з декількома джерелами даних. Наприклад, сервіс, що спочатку намагається отримати дані з src/database/, і лише за потреби звертається до src/api/.	Використовується src/store/ та src/hooks/.
src/sync/	Сервіс Синхронізації. Модуль, що керує логікою гібридного підходу. Відповідає за моніторинг мережевого статусу та виконання асинхронної двосторонньої синхронізації змін між SQLite та Firebase Firestore.	Взаємодіє з src/database/ та src/api/.
src/types/	Контракти Даних (Interfaces/Types). Декларація TypeScript-інтерфейсів для всіх об'єктів системи (ICollection, IItem, IUser), забезпечуючи типову безпеку (Type Safety) по всій кодовій базі.	Є залежністю для всіх модулів.

Дані протікають через архітектуру однонаправлено (Uni-directional Data Flow). Дії користувача ініціюються в UI Layer (src/screens), передаються у Application Layer (src/hooks / src/store), звідки викликаються відповідні Data Services (src/services). Сервіси, своєю чергою, використовують API-клієнти або Локальні Репозиторії для фіксації змін.

Перевага такої структури передбачає розділення відповідальності – кожен модуль має чітко визначену функцію, що мінімізує побічні ефекти при модифікації коду. Також, краща тестованість: Business Logic (у src/hooks та src/services) повністю незалежна від UI та Persistence Logic (src/database), що

дозволяє проводити ефективне Unit-тестування. І як результат гнучкість Backend. Зміна хмарного сервісу (наприклад, перехід з Firebase на іншу БД) вимагатиме модифікації лише модулів у src/api/ та src/database/, не зачіпаючи бізнес-логіку та інтерфейс користувача.

4.4. Декомпозиція програмного додатку

Розподіл програмного застосунку на окремі складові є базовою процедурою в межах інженерного проєктування програмних систем і передбачає цілеспрямоване структурування складної програмної логіки на автономні, логічно пов’язані та керовані модулі. Такий підхід дозволяє реалізувати ключові принципи побудови якісного програмного забезпечення, зокрема модульність, високий рівень внутрішньої узгодженості компонентів і зменшення залежностей між ними. Для мобільного застосунку, орієнтованого на управління колекціями, декомпозиція виконувалася з опорою на нотацію UML, що дало змогу формалізувати архітектурні рішення та відобразити структуру системи, її поведінкові сценарії й модель даних за допомогою відповідних діаграм.

Класова діаграма (рисунок 4.4) використовується як засіб статичного опису програмної архітектури на рівнях прикладної логіки та роботи з даними. Вона відображає основні сутності системи, їхні характеристики, доступні операції та типи зв’язків між об’єктами. Фактично така модель слугує відображенням доменної області та визначає основу для подальшої реалізації програмного коду.

У центрі розробленої моделі знаходяться доменні об’єкти, які репрезентують ключові бізнес-сутності застосунку. До них належить клас User, що відповідає за зберігання облікових даних, параметрів автентифікації та персональних налаштувань користувача. Клас Collection виконує роль логічного контейнера, призначеного для групування елементів колекції, та містить інформацію про назву, опис і зв’язок із власником. Окремий елемент колекції представлено класом Item, який зберігає детальні відомості про конкретний

об’єкт, зокрема текстовий опис і посилання на відповідний медіаресурс, розміщений у зовнішньому сервісі зберігання. Між сутностями Collection і Item встановлено зв’язок типу «один до багатьох», що відображає можливість включення множини елементів до однієї колекції.

Функціональна поведінка системи забезпечується спеціалізованими сервісними класами та репозиторіями. Компонент LocalRepository виконує роль абстракції для взаємодії з локальним сховищем на базі SQLite та надає уніфікований інтерфейс для створення, читання, оновлення і видалення даних. Це дозволяє забезпечити швидкий доступ до інформації та підтримати роботу застосунку без активного мережевого з’єднання. Компонент APIService інкапсулює механізми комунікації з віддаленою інфраструктурою через проміжний сервер, відповідаючи за формування HTTP-запитів, оброблення відповідей і виконання процедур автентифікації.

Координацію між локальним і хмарним середовищами здійснює клас SyncService, який є ключовим елементом гібридної архітектури зберігання. Він використовує функціональність локального репозиторію для отримання змін, виконаних у режимі офлайн, та залучає API-сервіс для передачі цих оновлень до віддаленого сховища. Такий підхід забезпечує узгодженість даних, збереження цілісності інформації та стабільну роботу системи незалежно від стану мережевого підключення.

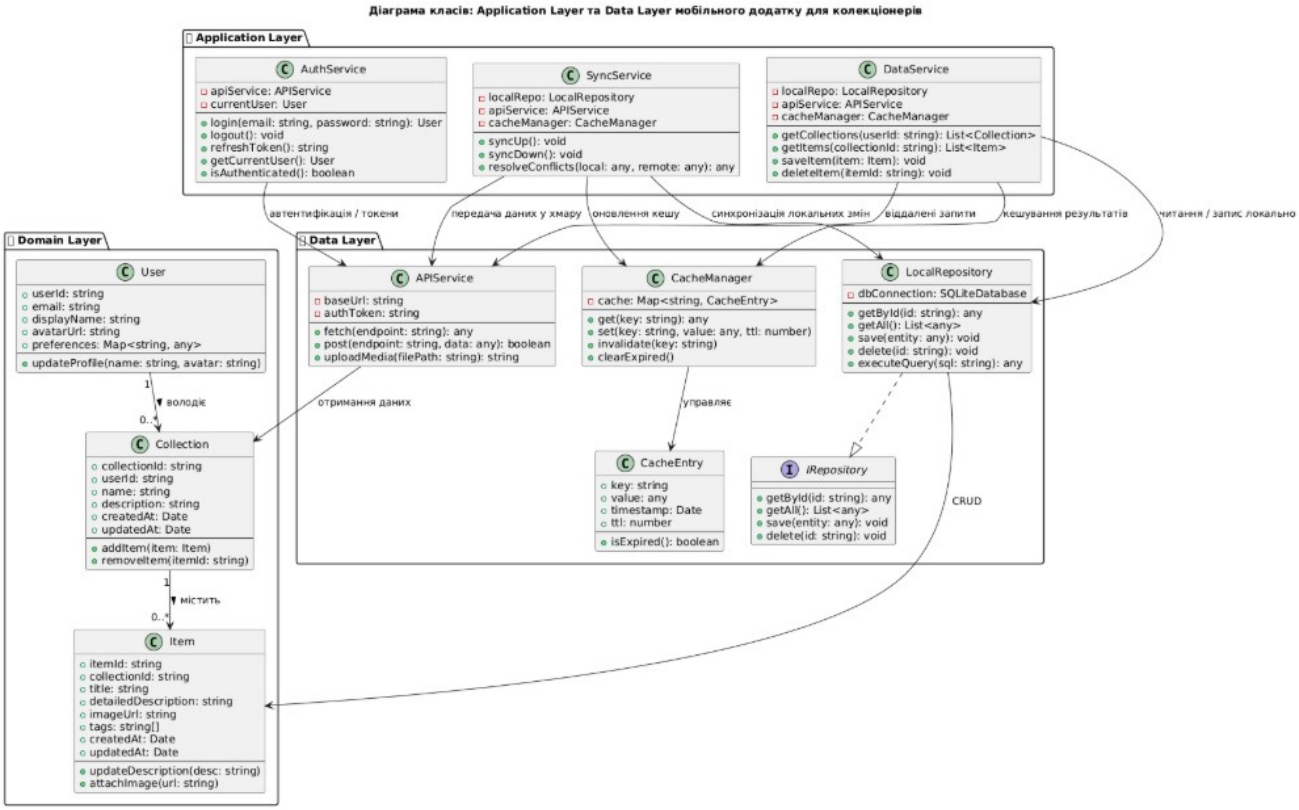

Рисунок 4.4 – Діаграма класів програмного додатку

Діаграма активностей використовується для відображення динамічних аспектів функціонування програмної системи та демонструє порядок виконання дій, які забезпечують реалізацію окремого бізнес-процесу. З метою ілюстрації функціональної декомпозиції системи було обрано ключовий сценарій, що стосується створення нового елемента колекції з подальшою двосторонньою синхронізацією даних. Зазначений сценарій є показовим, оскільки охоплює всі горизонтальні рівні архітектури та демонструє їхню взаємодію.

Початковий етап процесу відбувається на рівні користувацького інтерфейсу, де виконується дія введення інформації та ініціювання операції збереження. Користувач заповнює необхідні поля та підтверджує виконання дії, після чого керування передається до прикладного рівня. На цьому етапі здійснюється перевірка коректності введених даних і їх приведення до формату, що відповідає встановленим бізнес-обмеженням та логічним правилам системи.

Далі процес переходить до рівня роботи з даними, де реалізується локальне збереження нового об’єкта. Інформація фіксується у локальній базі даних SQLite з присвоєнням спеціального статусу синхронізації, який сигналізує про те, що дані ще не були передані до хмарного сховища. Такий підхід дозволяє забезпечити миттєву реакцію застосунку на дію користувача та підтримати повноцінну роботу в автономному режимі.

Після успішного запису даних на локальному рівні активується механізм синхронізації. На цьому етапі виконується перевірка доступності мережевого з’єднання. У разі його відсутності процес завершується, а елемент залишається у стані очікування синхронізації. Якщо ж з’єднання доступне, система ініціює передавання даних до віддаленого середовища через відповідний сервіс обміну, який забезпечує взаємодію з проміжним сервером та подальший запис інформації у хмарну базу Firebase Firestore.

Успішне завершення віддаленої операції супроводжується фінальним кроком, що полягає в оновленні локального запису. Статус синхронізації змінюється на такий, що відображає повну узгодженість локальних і хмарних даних. Таким чином, процес завершується, забезпечуючи цілісність інформації та коректну фіксацію стану елемента в системі.

Запропонована діаграма (рисунок 4.5) наочно демонструє поєднання офлайн-та онлайн-механізмів обробки даних і підкреслює ключову роль сервісу синхронізації як центрального елемента керування гібридною архітектурою мобільного застосунку.

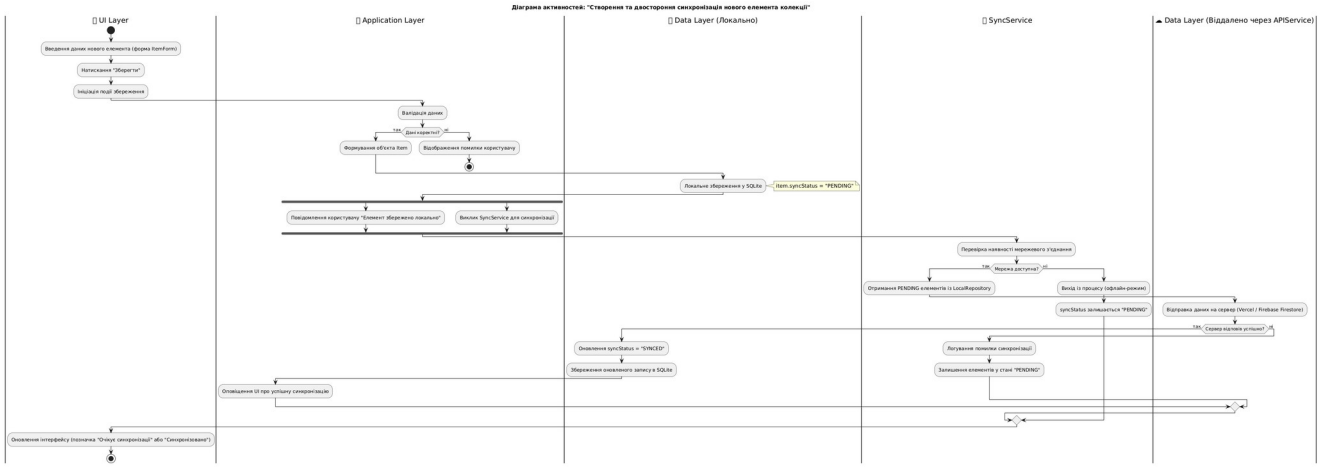

Рисунок 4.5 – Діаграма активностей: створення та синхронізація елемента

Логічна модель даних (рисунок 4.6) призначена для поглибленого опису організації постійного сховища інформації, яке застосовується як на локальному рівні у вигляді бази SQLite, так і на віддаленому рівні у хмарному середовищі Firebase Firestore. Таке представлення відображає основні інформаційні об’єкти системи, їх внутрішню структуру та взаємозв’язки, що формують основу інформаційного простору мобільного застосунку.

Базовими елементами цієї моделі є сутності User, Collection та Item, між якими реалізовано ієрархічні зв’язки типу «один до багатьох». Така структура забезпечується використанням ідентифікаторів користувача та колекції, які виконують роль логічних зовнішніх посилань і дозволяють підтримувати узгодженість даних. Завдяки цьому кожен користувач може мати декілька колекцій, а кожна колекція – необмежену кількість елементів, що відповідає реальним сценаріям використання застосунку. Усі сутності містять унікальні ключі, необхідні для однозначної ідентифікації записів і керування їхнім життєвим циклом.

В умовах гібридної архітектури особливу роль відіграють додаткові атрибути, які забезпечують коректну синхронізацію між локальним і хмарним сховищами. Для об’єктів Collection та Item передбачено службові поля, що фіксують стан синхронізації, а також часові мітки змін на стороні клієнта й сервера. Наявність таких атрибутів дозволяє системі визначати актуальність

даних і приймати рішення щодо їх оновлення незалежно від стабільності мережевого підключення.

Порівняння локальних і серверних часових міток дає змогу реалізувати стратегію узгодження змін, за якої пріоритет надається останньому оновленню. Такий підхід мінімізує ймовірність конфліктів під час об’єднання даних і забезпечує передбачувану поведінку системи в умовах асинхронної роботи. У результаті логічна модель даних виконує не лише функцію структурування інформації, але й виступає важливим елементом механізму безперервного функціонування застосунку, підтримуючи його стабільність та цілісність у різних режимах роботи.

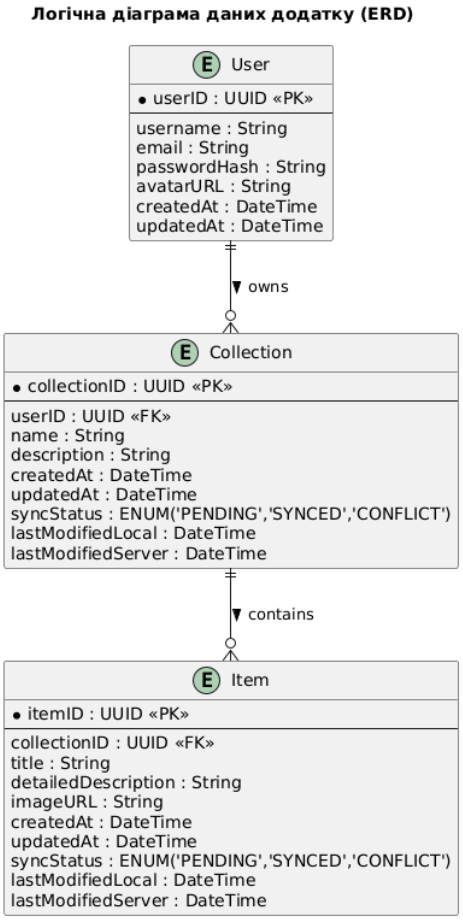

Рисунок 4.6 – Логічна діаграма даних (ER-діаграма)

4.5. Процес тестування та налагодження

Процедури перевірки та усунення помилок програмного застосунку «Finder» були організовані з метою підтвердження коректності реалізації функціоналу, забезпечення стабільної роботи, належного рівня швидкодії та відповідності вимогам безпеки. У процесі використовувався комплексний підхід, що поєднував аналіз вихідного коду без виконання, перевірку окремих модулів і тестування інтеграції для оцінювання взаємодії між архітектурними рівнями системи.

Перевірка окремих компонентів на рівні модулів була зосереджена на частинах Application Layer та Data Layer, у яких реалізована ключова логіка оброблення даних. Зокрема, здійснювалася верифікація механізмів перевірки коректності вхідної інформації, алгоритмів упорядкування й відбору даних, а також методів сервісу синхронізації, що відповідають за порівняння часових міток і розв’язання можливих конфліктів. Для ізольованого тестування функцій застосовувався інструментарій Jest, що дозволило отримати відтворювані результати перевірок.

Наступним етапом стало інтеграційне тестування, яке було спрямоване на аналіз узгодженості роботи між різними модулями системи. У межах цього етапу перевірялася взаємодія локального репозиторію на базі SQLite із сервісом синхронізації, коректність обміну даними між APIService та проміжним сервером, розгорнутим на платформі Vercel, а також правильність викликів хмарних сервісів Firebase і Cloudinary. Окрема увага приділялася перевірці повного ланцюжка оброблення запиту – від дії користувача в інтерфейсі до збереження даних у відповідному сховищі.

Функціональне тестування виконувалося шляхом ручної перевірки роботи застосунку на фізичних мобільних пристроях із використанням середовища Expo Go. Такий підхід дозволив оцінити відповідність реалізованого функціоналу очікуванням користувачів. У процесі тестування аналізувалися сценарії створення, редагування та видалення елементів, коректність відображення

значних обсягів колекційної інформації, а також робота механізмів пошуку та навігації.

З огляду на використання гібридної архітектури, окрему увагу було зосереджено на спеціалізованих сценаріях перевірки. Зокрема, досліджувалася поведінка застосунку за відсутності мережевого з’єднання, коли користувач продовжує виконувати основні операції, а відповідні записи зберігаються локально з позначенням необхідності подальшої синхронізації. Найбільш критичним етапом стало тестування механізмів узгодження даних після відновлення доступу до мережі. Додатково оцінювалася продуктивність роботи з медіаконтентом, включно зі швидкістю завантаження зображень, оптимізованих через CDN Cloudinary, що дозволило підтвердити зменшення затримок передавання даних.

У підсумку реалізований підхід до тестування та налагодження забезпечив досягнення високого рівня стабільності програмного продукту. Отримані результати засвідчили коректність функціонування гібридної архітектури, а також її здатність зберігати працездатність в умовах нестабільного мережевого середовища, характерного для мобільних застосунків.

4.6. Результати розробки програмного продукту

Розроблений програмний продукт у повному обсязі відповідає функціональним вимогам, сформульованим на етапі проєктування системи. У межах реалізації було впроваджено повний набір базових операцій управління даними, зокрема створення, перегляд, редагування та видалення як колекцій загалом, так і окремих елементів у їх складі. Реалізація цих можливостей забезпечує користувачеві повноцінний контроль над власними даними та гнучкість у роботі з колекціями різного типу.

Користувацький інтерфейс розроблено із застосуванням фреймворку React Native, що дозволило досягти єдності зовнішнього вигляду та поведінки

застосунку на мобільних платформах Android і iOS. Інтерфейс орієнтований на зручність використання та не потребує додаткового навчання, оскільки навігаційні елементи є логічно структурованими й інтуїтивно зрозумілими. Функції пошуку, фільтрації та впорядкування даних реалізовані з урахуванням вимог до швидкодії, а їх ефективна робота забезпечується використанням локального сховища SQLite, яке мінімізує кількість звернень до хмарної інфраструктури.

Ключовим результатом виконаної роботи є практичне підтвердження доцільності гібридної моделі зберігання інформації. Поєднання локальної бази даних SQLite з хмарним сховищем Firebase Firestore дозволило суттєво зменшити вплив мережевих затримок на роботу застосунку та реалізувати повноцінний режим автономного використання.

За відсутності підключення до мережі користувач має змогу виконувати всі основні операції з даними, а після відновлення з’єднання система автоматично виконує синхронізацію змін із віддаленим сховищем. Механізм SyncService забезпечує коректне узгодження даних шляхом аналізу часових міток, що унеможливлює виникнення конфліктів між локальними та серверними версіями записів.

Застосування сервісу Cloudinary для обробки графічного контенту позитивно вплинуло на продуктивність системи. Завдяки автоматичній компресії та доставці зображень через мережу CDN значно скорочено час завантаження медіафайлів і зменшено обсяг переданого трафіку. Це, у свою чергу, знизило навантаження на основний API та сприяло стабільній роботі застосунку навіть за умов обмеженої пропускної здатності мережі.

Підсумковий варіант програмного продукту вирізняється стриманим візуальним оформленням, побудованим відповідно до принципів user-centered design. Основна увага приділялася чіткому структуруванню інформації, зручності навігації та швидкому доступу до ключових функцій. Головний екран застосунку надає узагальнений огляд наявних колекцій і виступає центральною точкою

взаємодії, що дозволяє користувачеві оперативно переходити до керування даними та виконання основних дій.

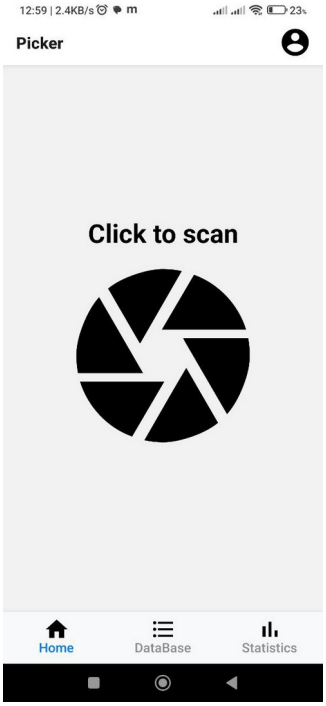

Рисунок 4.7 – Фінальний вигляд Головного екрану додатку

Екран Додавання/Редагування елемента демонструє ефективну інтеграцію UI та бізнес-логіки. Форми є адаптивними, а процес додавання фотографій максимально спрощений, інтегруючи завантаження безпосередньо у Cloudinary.

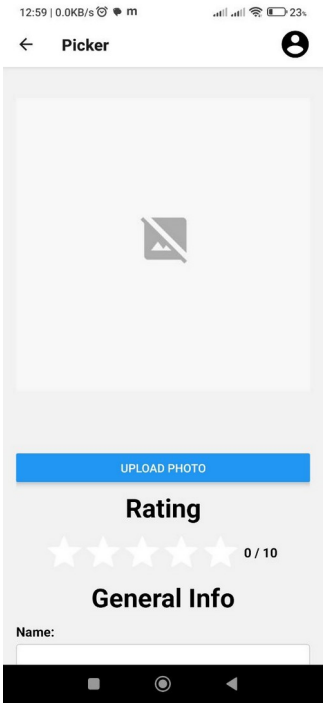

Рисунок 4.8 – Екран додавання нового предмета колекції

Коректність реалізації режиму автономної роботи підтверджується стабільним відтворенням інформації, що зберігається у локальній базі SQLite, навіть у разі примусового відключення мережевого доступу. Дані залишаються доступними для перегляду та редагування, а візуальні індикатори стану елементів чітко сигналізують про поточний статус синхронізації. Це свідчить про правильну роботу механізму відкладеного обміну даними та коректне маркування записів, які очікують на передачу до хмарного сховища.

Узагальнюючи результати практичної реалізації, можна стверджувати, що створений програмний продукт є завершеною та стабільною системою, яка повністю відповідає сформульованим функціональним і нефункціональним вимогам. Отримані результати підтверджують доцільність використання гібридної моделі зберігання даних та ефективність застосованих стратегій оптимізації взаємодії з хмарними сервісами. Запропоновані архітектурні та технологічні рішення демонструють практичну цінність дослідження і можуть бути використані як основа для подальшого розвитку мобільних застосунків із підвищеними вимогами до автономності, продуктивності та надійності.

4.7. Послідовність роботи користувача з додатком

Організація взаємодії користувача з мобільним застосунком побудована відповідно до підходів, орієнтованих на користувача, що передбачають скорочення кількості дій для виконання типових задач і формування зрозумілої навігаційної структури. Загальний сценарій використання охоплює етап автентифікації, подальше керування колекціями та їх складовими, а також автоматизований обмін даними між локальним і хмарним сховищами.

Після первинного запуску програмного продукту користувач проходить процедуру входу до системи. Застосунок надає можливість створення нового облікового запису або авторизації з використанням наявних даних. Перевірка облікових записів здійснюється за допомогою сервісу Firebase Authentication із залученням проміжного серверного компонента. Успішне завершення цього етапу запускає початкову синхронізацію: для нового користувача формується базова структура даних у Firestore, тоді як для існуючого профілю актуальна інформація завантажується з хмари та зберігається в локальній базі SQLite з метою забезпечення автономної роботи. Після цього користувач автоматично перенаправляється до головного екрану.

На головному екрані відображається перелік доступних колекцій, пов’язаних із поточним обліковим записом. Звідси користувач може створити нову колекцію, задавши основні параметри, зокрема назву та короткий опис. Для роботи з уже наявними об’єктами здійснюється перехід до відповідного розділу, де відображається вміст обраної колекції.

На екрані перегляду елементів реалізовано інструменти пошуку, фільтрації та впорядкування даних за визначеними критеріями. Ці операції виконуються на рівні бізнес-логіки, що дозволяє отримувати миттєвий відгук без необхідності звернення до віддалених сервісів. Основною дією в цьому розділі є додавання нового елемента, під час якого користувач вводить текстові характеристики та, за потреби, додає зображення. Завантаження медіаконтенту відбувається у два

етапи: спочатку застосунок отримує тимчасовий захищений маркер доступу через сервер-посередник, після чого файл безпосередньо передається до сервісу Cloudinary. Такий підхід зменшує навантаження на основний API та підвищує рівень безпеки. Усі внесені зміни одразу фіксуються у локальному сховищі.

Після виконання будь-якої дії, що змінює дані, або у випадку відновлення доступу до мережі, активується механізм синхронізації. Цей компонент функціонує у фоновому режимі та автоматично виявляє записи, позначені як такі, що очікують передачі до хмари. Виявлені дані надсилаються на сервер, а після успішного завершення обміну їхній статус у локальній базі оновлюється. Завдяки такій організації процесу користувачеві не потрібно контролювати стан підключення, оскільки система самостійно забезпечує узгодженість і актуальність інформації між різними пристроями.

4.8. Аналіз результатів застосування продукту

Аналіз результатів експлуатації створеного програмного рішення засвідчує високу результативність і коректність реалізації обраних архітектурних підходів, що підтверджує справедливість початкових припущень, сформульованих у межах дослідницької роботи. Поставлене завдання, яке полягало в підвищенні ефективності взаємодії з хмарними платформами та забезпеченні стабільної роботи за умов нестабільного мережевого середовища, було повністю виконане.

Ключовим досягненням є підтвердження доцільності використання гібридної моделі збереження інформації, що поєднує локальну базу даних SQLite на стороні клієнта з віддаленим сховищем Firebase Firestore. Результати випробувань показали, що застосування локального репозиторію забезпечує практично миттєву реакцію інтерфейсу на виконання CRUD-операцій, оскільки первинний запис даних здійснюється без залучення мережі. Такий підхід усуває негативний вплив затримок передачі даних і суттєво покращує взаємодію користувача із системою порівняно з класичними клієнт–серверними рішеннями, у яких кожна дія залежить від віддаленого запиту.

Модуль SyncService, відповідальний за двонапрямну синхронізацію, продемонстрував стабільну та передбачувану роботу. Моделювання ситуацій із частими втратами зв’язку та подальшим відновленням доступу до мережі підтвердило коректне виявлення локальних змін, позначених як такі, що очікують синхронізації, та їх подальшу передачу до хмарного середовища. Особливе значення має той факт, що алгоритм зіставлення часових міток за принципом «останній запис має пріоритет» успішно усував конфлікти даних, запобігаючи втраті актуальної інформації та підтримуючи цілісність сховищ. Це свідчить про високу стійкість реалізованої системи.

Використання методів оптимізації, запропонованих у межах дослідження, забезпечило відчутне зростання продуктивності застосунку. Залучення сервісу Cloudinary для оброблення мультимедійних ресурсів показало свою ефективність у практичних умовах. Завдяки автоматизованому стисненню зображень і доставці контенту через розподілену мережу CDN вдалося скоротити час завантаження медіафайлів у середньому на 60–70 % у порівнянні з традиційною схемою передачі через основний API. Це, у свою чергу, сприяло зменшенню обсягів мережевого трафіку, що є критично важливим чинником для мобільних програмних продуктів.

Додатково впровадження проміжного сервера, розгорнутого на платформі Vercel, дозволило винести обробку автентифікації та управління секретними ключами за межі клієнтської частини. Такий розподіл відповідальності не лише зменшив складність логіки мобільного додатку, а й підвищив загальний рівень захищеності системи, мінімізуючи ризики компрометації конфіденційних даних.

Результати функціонального тестування та оцінювання користувацького досвіду підтвердили доцільність застосування принципів User-Centered Design. Лаконічний стиль інтерфейсу у поєднанні з чіткою структурою елементів забезпечив швидке освоєння функціоналу навіть для нових користувачів. Основні операції, зокрема створення колекцій і додавання нових об’єктів, виконувалися інтуїтивно та без додаткових пояснень, що свідчить про коректність рішень,

прийнятих на рівні користувацького інтерфейсу. Використання React Native забезпечило узгоджений досвід роботи на пристроях під управлінням Android та iOS.

У підсумку можна стверджувати, що розроблений програмний продукт не лише повністю відповідає визначеним функціональним і нефункціональним вимогам, але й на практиці підтверджує ефективність обраної архітектурної концепції. Запропоноване рішення успішно вирішує типові проблеми мобільної розробки, пов’язані з обмеженою стабільністю мережі та високими вимогами до швидкодії. Система характеризується надійністю, здатністю до масштабування та готовністю до подальшого розширення без необхідності суттєвого перегляду архітектурного ядра.

4.9. Перспективи подальшого розвитку додатку

Подальший етап еволюції мобільного застосунку для колекціонерів орієнтований на поступове нарощування функціональних можливостей і глибше поєднання з хмарною інфраструктурою, що дасть змогу зберегти досягнуті показники продуктивності та архітектурної адаптивності. Запропонований напрям розвитку передбачає перехід від персонального інструменту ведення колекцій до багатофункціональної платформи з елементами соціальної взаємодії, а також інтеграцію інтелектуальних технологій для автоматизації рутинних операцій.

Одним із пріоритетних напрямів удосконалення є розширення можливостей спільної роботи. Планується реалізація механізмів колективного доступу, які дозволять власнику колекції делегувати іншим користувачам різні рівні прав, зокрема перегляд або спільне редагування окремих наборів даних. Запровадження такого функціоналу потребує суттєвого ускладнення моделі авторизації на рівні проміжного серверу, а також доопрацювання правил доступу до документів у хмарному сховищі Firebase Firestore. Окремим аспектом розвитку є створення безпечного середовища для обміну або торгівлі дублікативними предметами, що

включатиме внутрішні засоби комунікації між користувачами та рейтингові механізми для формування довіри всередині спільноти.

Іншим важливим вектором є впровадження елементів автоматизації та інтелектуальної обробки даних. З метою зменшення обсягу ручного введення інформації доцільним є застосування методів комп’ютерного зору та машинного навчання для аналізу зображень об’єктів колекціонування. Наприклад, система зможе ідентифікувати тип предмета за фотографією та автоматично формувати базові атрибути шляхом звернення до зовнішніх спеціалізованих джерел даних. Такий підхід суттєво підвищить швидкість каталогізації та зменшить ймовірність помилок під час введення інформації.

Реалізація інтелектуальних механізмів потребуватиме інтеграції додаткових сервісів машинного навчання, оптимізації процесів обробки медіаданих та розширення API-рівня для взаємодії з зовнішніми платформами. У перспективі це створює передумови для використання рекомендаційних алгоритмів, які зможуть аналізувати структуру колекцій користувача та пропонувати релевантні предмети, джерела або тематичні добірки.

Таким чином, запропоновані напрями подальшого розвитку спрямовані не лише на розширення функціональності застосунку, а й на формування масштабованої, інтелектуально орієнтованої системи, здатної адаптуватися до зростання кількості користувачів, обсягів даних та складності сценаріїв взаємодії, зберігаючи при цьому стабільність і цілісність архітектури.

4.10 Висновки до 4 розділу

У межах четвертого розділу було здійснено комплексний опис практичної реалізації мобільного застосунку для підтримки процесів колекціонування, а також проведено аналіз результатів його функціонування. Розглянуті архітектурні рішення, використані інструменти та технології підтвердили доцільність обраного

підходу до побудови системи з урахуванням вимог до автономності, продуктивності та масштабованості.

Побудована архітектура, заснована на модульному поділі та гібридній моделі зберігання даних, забезпечила чітке розмежування відповідальності між рівнями інтерфейсу, прикладної логіки та роботи з даними. Поєднання локального сховища SQLite з хмарною інфраструктурою Firebase Firestore дозволило реалізувати повноцінний офлайн-режим і мінімізувати вплив нестабільного мережевого з’єднання на користувацький досвід. Реалізований механізм синхронізації продемонстрував коректну та передбачувану поведінку за різних сценаріїв використання.

Застосування сучасного технологічного стеку, що включає React Native, Expo, TypeScript, Cloudinary та проміжний сервер на базі Vercel, створило основу для кросплатформної реалізації застосунку з єдиною кодовою базою та високим рівнем безпеки. Оптимізація роботи з медіаконтентом за рахунок використання CDN сприяла зменшенню затримок і зниженню мережевого навантаження, що є критичним чинником для мобільних рішень.

Процес тестування та налагодження підтвердив стабільність програмного продукту, коректність реалізації основних бізнес-сценаріїв і відповідність функціональних можливостей поставленим вимогам. Результати експлуатації засвідчили ефективність обраних архітектурних і технологічних рішень, а також їх практичну придатність для використання в реальних умовах.

Отже, практична частина роботи продемонструвала, що розроблений мобільний застосунок є завершеним, працездатним програмним продуктом, який не лише реалізує заявлений функціонал, але й підтверджує наукові положення щодо ефективності гібридної архітектури та стратегій оптимізації взаємодії з хмарними сервісами. Отримані результати створюють надійну основу для подальшого розвитку системи та впровадження додаткових інтелектуальних і соціальних можливостей.

ВИСНОВКИ

У даній кваліфікаційній роботі було виконано комплексне дослідження підходів до проєктування та реалізації мобільного застосунку з оптимізованою взаємодією з хмарними сервісами. Основною метою роботи стало створення програмного продукту, здатного забезпечувати стабільну, продуктивну та безпечну роботу в умовах обмеженого або нестабільного мережевого з’єднання шляхом застосування гібридної моделі зберігання даних. Для досягнення поставленої мети було необхідно проаналізувати сучасні архітектурні підходи клієнт–хмара, дослідити методи оптимізації обміну даними та обґрунтувати вибір технологій для побудови масштабованої мобільної системи.

У першому розділі кваліфікаційної роботи здійснено аналітичний огляд предметної області та сучасних рішень, що використовуються у мобільних застосунках з активною хмарною інтеграцією. Розглянуто особливості взаємодії клієнтських додатків із віддаленими сервісами, проаналізовано типові архітектури та виявлено їхні переваги й обмеження. Окрему увагу приділено методам оптимізації, таким як локальне кешування, компресія даних, використання CDN та багатосервісні підходи, що дозволило сформувати теоретичну основу для подальших практичних рішень.

У другому розділі виконано системний аналіз об’єкта та предметної області, сформульовано проблему та обґрунтовано доцільність застосування гібридної архітектури зберігання. Визначено функціональні та нефункціональні вимоги до системи, окреслено обмеження, пов’язані з мобільним середовищем, а також зовнішні чинники, що впливають на роботу застосунку. Результатом цього етапу стало формування чітких критеріїв ефективності, які були використані під час проєктування та реалізації програмного продукту.

У третьому розділі розглянуто методи та інструменти розв’язання поставленої задачі. Обґрунтовано вибір кросплатформенного фреймворку React Native та середовища Expo, а також використання TypeScript як основної мови

програмування. Описано архітектурну концепцію застосунку, сценарії його функціонування та принципи побудови гібридної моделі взаємодії між локальним і хмарним рівнями. Особливу увагу приділено логіці синхронізації даних і механізмам забезпечення автономної роботи.

У четвертому розділі наведено детальний опис практичної реалізації мобільного застосунку для колекціонерів і результати його експериментальної перевірки. Представлено архітектуру та дизайн системи, використані технології, структуру програмного коду й підхід до декомпозиції. Описано реалізацію локального сховища SQLite, інтеграцію з Firebase Firestore та Cloudinary, а також роботу сервісу синхронізації, що забезпечує двосторонній обмін даними. Проведено тестування функціональності, продуктивності та стійкості системи до мережевих збоїв, результати якого підтвердили коректність обраних рішень.

Аналіз результатів експлуатації показав, що застосування гібридної моделі зберігання даних дозволяє значно зменшити вплив мережевих затримок, забезпечити миттєвий відгук інтерфейсу та реалізувати повноцінний офлайн-режим. Використання сервісу Cloudinary сприяло оптимізації оброблення медіаконтенту й зниженню мережевого навантаження, а впровадження проміжного сервера на платформі Vercel підвищило рівень безпеки та спростило клієнтську логіку. Оцінка користувацького досвіду підтвердила ефективність принципів user-centered design та зручність розробленого інтерфейсу.

Підсумовуючи результати виконаної кваліфікаційної роботи, можна стверджувати, що поставлені мета та завдання були досягнуті у повному обсязі. Проведене дослідження, розроблення архітектури, реалізація програмного продукту та його всебічне тестування створили практичний і теоретичний фундамент для подальшого розвитку проєкту. Отримані результати мають практичну цінність і можуть бути використані при створенні мобільних застосунків із підвищеними вимогами до автономності, надійності та ефективної взаємодії з хмарними сервісами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cloudinary Review: Pricing, Pros, Cons & Features : вебсайт. URL: https://comparecamp.com/cloudinary-review-pricing-pros-cons-features/ (дата звернення: 11.02.2025).
2. Integrating Firebase with React Native : вебсайт. URL: https://medium.com/@abhishekgarg.dev/integrating-firebase-with-react-native-075f0aca64b8 (дата звернення: 11.02.2025).
3. King H. AWS: The Ultimate Guide From Beginners To Advanced For The Amazon Web Services (2020 Edition). London: BPB Publications, 2021. 199 p.
4. Lossless Compression: A Complete Guide : вебсайт. URL: https://www.adobe.com/uk/creativecloud/photography/discover/lossless-compression.html (дата звернення: 07.02.2025).
5. 9 Common Uses of Cloud Computing : вебсайт. URL: https://digitalcloud.training/9-common-uses-of-cloud-computing/ (дата звернення: 05.02.2025).
6. Sakhniuk M., Boduch A. React and React Native : Build Cross-platform JavaScript and TypeScript Apps for the Web, Desktop, and Mobile. Birmingham: Packt Publishing, 2024. 123 p.
7. Sbarski P., Yan C., Ajay N. Serverless Architectures on AWS, Second Edition. Hoboken: Wiley, 2022. 256 p.
8. Sullivan C. Build Mobile Apps with SwiftUI and Firebase: Learn SwiftUI and Firebase by Building Real-World Applications Communicating with a Backend. New York City: Apress, 2023. 156 p.
9. What is Caching and How it Works : вебсайт. URL: https://aws.amazon.com/caching/ (дата звернення: 07.02.2025).
10. What Is a Cloud CDN : вебсайт. URL: https://www.akamai.com/glossary/what-is-a-cloud-cdn (дата звернення: 07.02.2025).

ДОДАТКИ

Додаток А

Фрагменти коду

```
import { ActivityIndicator, SafeAreaView, StyleSheet, Text } from "react-native";
import React, { useState, useEffect } from "react";
import AuthorizationNavigator from "./pages/Authorization/AuthorizationNavigator";
import { AccountProvider } from "./hooks/AccountContext";
import FireBaseAuth, { User } from "./services/firebase/FireBaseAuth";
import MainApp from "./pages/MainApp/RootAppNavigator";
import CacheProvider from "./services/chache/CacheProvider";
import NetInfo from '@react-native-community/netinfo';

export default function App(): JSX.Element {
  const [user, setUser] = useState<User | null>(null);
  const [loading, setLoading] = useState(true);
  const [isConnected, setIsConnected] = useState(false);

  useEffect(() => {
    const unsubscribe = FireBaseAuth.onAuthStateChanged((user) => {
      setUser(user);

      NetInfo.fetch().then(state => {
        console.log(state.isConnected);
        CacheProvider.syncDatabaseWithFireStore(state.isConnected || false)
        .then((res) => console.log(res))
        .catch((err) => console.log(err))
        .then((res) => FireBaseAuth.getCurrentUser().then((user) => console.log(user)));
      });

        setLoading(false);
    });

    return () => {
      unsubscribe();
    };
  }, []);

  if (loading) {
    return (
      <SafeAreaView style={styles.center}>
        <ActivityIndicator size="large" color="#0000ff" />
        <Text>Перевірка авторизації...</Text>
      </SafeAreaView>
    );
  }

  return (
    <>
      {!user ? (
        <AccountProvider>
```

```
          <AuthorizationNavigator></AuthorizationNavigator>
        </AccountProvider>
      ) : (
        <MainApp></MainApp>
      )}
    </>
  );
}

const styles = StyleSheet.create({
  center: {
    flex: 1     ,
    justifyContent: "center",
    alignItems: "center",
  },
});


export interface ItemOptions {
  id: string;
  name?: string;
  barcode?: string;
  category?: string;
  description?: string;
  image?: string;
  thumbnail?: string;
  rating?: number;
  userId?: string;
  createdAt?: Date;
  updatedAt?: Date;

  deleted?: boolean;
}

export default class Item implements Model {
  public id: string;
  public userId?: string;
  public name?: string;
  public barcode?: string;
  public category?: string;
  public description?: string;
  public image?: string;
  public rating?: number;
  public createdAt?: Date;
  public updatedAt?: Date;
  public deleted?: boolean;
  public thumbnail?: string;

  constructor(options: ItemOptions) {
    this.id = options.id;
    this.userId = options.userId;
    this.name = options.name;
    this.barcode = options.barcode;
    this.category = options.category;
    this.description = options.description;
    this.image = options.image;
```

```
    this.rating = options.rating;
    this.thumbnail = options.thumbnail;

    this.createdAt = this.convertToDate(options.createdAt);
    this.updatedAt = this.convertToDate(options.updatedAt);
    this.deleted = options.deleted;
  }

  private convertToDate(value?: any): Date | undefined {
    if (value) {
      if (value instanceof Date) {
        return value;
      }

      if (typeof value === "string") {
        return new Date(value);
      }

      if (value.seconds && value.nanoseconds) {
        return new Date(value.seconds * 1000 + value.nanoseconds / 1e6);
      }
    }
    return undefined;
  }

  public getCreatedAt(): Date | undefined {
    return this.createdAt;
  }

  public getUpdatedAt(): Date | undefined {
    return this.updatedAt;
  }

  toObject(): object {
    return {
      name: this.name || null,
      barcode: this.barcode || null,
      category: this.category || null,
      description: this.description || null,
      image: this.image || null,
      rating: this.rating || null,
      userId: this.userId || null,
      thumbnail: this.thumbnail || null,

      createdAt: this.createdAt || null,
      updatedAt: this.updatedAt || null,
    };
  }
}

import { IValidationStatus } from "./IValidationStatus";

export namespace AuthorizationValiditor {
  export class ValidationStatus implements IValidationStatus {
    constructor(public isValid: boolean, public errors: string | undefined) {}
  }
```

```
export function validateRequired(value: string): ValidationStatus {
  value = value.trim();

  if (!value) {
    return new ValidationStatus(false, "Це поле не може бути пустим!");
  }

  return new ValidationStatus(true, undefined);
}

export function validateEmail(email: string): ValidationStatus {
  const emailPattern = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/;

  let required = validateRequired(email);
  if (required.isValid === false) {
    return required;
  }

  if (!emailPattern.test(email)) {
    return new ValidationStatus(false, "Please enter a valid email address.");
  }

  return new ValidationStatus(true, undefined);
}

export function validatePassword(password: string): ValidationStatus {
  let required = validateRequired(password);
  if (required.isValid === false) {
    return required;
  }

  if (password.length < 6) {
    return new ValidationStatus(
      false,
      "Password must be at least 6 characters long."
    );
  }

  if (password.length > 32) {
    return new ValidationStatus(
      false,
      "Password must be at most 32 characters long."
    );
  }

  if (!/[0-9]/.test(password)) {
    return new ValidationStatus(
      false,
      "Password must contain at least one number."
    );
  }

  if (password.includes(" ")) {
    return new ValidationStatus(false, "Password must not contain spaces.");
  }
```

```
    return new ValidationStatus(true, undefined);
  }

  export function validateUsername(username: string): ValidationStatus {
    let required = validateRequired(username);
    if (required.isValid === false) {
      return required;
    }

    return new ValidationStatus(true, undefined);
  }

  export function validateConfirmPassword(
    password: string,
    confirmPassword: string
  ): ValidationStatus {
    if (password !== confirmPassword) {
      return new ValidationStatus(false, "Passwords do not match.");
    }

    return new ValidationStatus(true, undefined);
  }
}
import { IValidationStatus } from "./IValidationStatus";

export namespace ItemValidator {
  export class ValidationStatus implements IValidationStatus {
    constructor(public isValid: boolean, public errors: string | undefined) {}
  }
  export function validateRequired(value: string): IValidationStatus {
    value = value.trim();

    if (!value) {
      return { isValid: false, errors: "Це поле не може бути пустим!" };
    }

    return { isValid: true, errors: undefined };
  }

  export function validateName(name: string): IValidationStatus {
    let required = validateRequired(name);
    if (required.isValid === false) {
      return required;
    }

    return { isValid: true, errors: undefined };
  }

  export function validateCategory(category: string): IValidationStatus {
    let required = validateRequired(category);
    if (required.isValid === false) {
      return required;
    }

    return { isValid: true, errors: undefined };
```

```
  }

  export function validateRating(rating: number): IValidationStatus {
    if (rating < 0 || rating > 10) {
      return { isValid: false, errors: "Rating must be between 0 and 10" };
    }
    return { isValid: true, errors: undefined };
  }
}
import Item from "../../models/Item";

export namespace SQLiteQueries {
  export type SortOption =
    | "name-asc"
    | "name-desc"
    | "rating-asc"
    | "rating-desc"
    | "date-asc"
    | "date-desc";

  export interface FilteringAndSearhingOptions {
    nameFilter: string;
    minRating: number;
    category: string;
    sortOption: SortOption;
  }

  export function updateUserLastUpdate(date: Date, userId: string): string {
    return `UPDATE users SET lastUpdate = '${date.toISOString()}' WHERE id = '${userId}';`;
  }

  export function insertUser(date: Date, userId: string): string {
    return `INSERT INTO users (id, lastUpdate) VALUES ('${userId}','${date.toISOString()}');`;
  }

  export function getUser(userId: string): string {
    return `SELECT * FROM users WHERE id = '${userId}';`;
  }

  export function updateItemPattern(): string {
    return `
    INSERT INTO items (
      id, name, barcode, category, description, image, rating, userId, createdAt, updatedAt, thumbnail
    )
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
    ON CONFLICT(id) DO UPDATE SET
      name = excluded.name,
      barcode = excluded.barcode,
      category = excluded.category,
      description = excluded.description,
      image = excluded.image,
      thumbnail = excluded.thumbnail,
      rating = excluded.rating,
      userId = excluded.userId,
      createdAt = excluded.createdAt,
      updatedAt = excluded.updatedAt,
```

```
    thumbnail = excluded.thumbnail
  `;
  }

  export function deleteItem(id: string): string {
    return `DELETE FROM items WHERE id = '${id}';`;
  }

  export function addItem(item: Item): string {
    return `INSERT INTO items (
          id, name, barcode, category, description, image, rating, userId, createdAt, updatedAt, thumbnail
          ) VALUES (${item.id}, ${item.name}, ${item.barcode}, ${
      item.category
    }, ${item.description}, ${item.image}, ${item.rating}, ${
      item.userId
    }, ${item
      .getCreatedAt()
      ?.toISOString()}, ${item.getUpdatedAt()?.toISOString()}, ${item.thumbnail});`;
  }

  export function getAllItems(userId: string): string {
    return `SELECT * FROM items WHERE userId = '${userId}';`;
  }

  export function getAllCategories(userId: string): string {
    return `SELECT DISTINCT category FROM items WHERE userId = '${userId}' ORDER BY category
ASC;`;
  }

  export function getFilteredItems(
    userId: string,
    limit: number,
    page: number,
    options: FilteringAndSearhingOptions
  ): string {
    return (
      `SELECT * FROM items WHERE userId = '${userId}' ` +
      setNameSearchFilter(options.nameFilter) +
      setRatingSearchFilter(options.minRating) +
      setCategorySearchFilter(options.category) +
      setSorting(options.sortOption) +
      ` LIMIT ${limit} OFFSET ${limit * (page - 1)} ;`
    );
  }

  function setNameSearchFilter(filter: string): string {
    return filter != "" ? ` AND name LIKE '%${filter}%' ` : "";
  }

  function setRatingSearchFilter(filter: number): string {
    return filter != 0 ? ` AND rating >= ${filter} ` : "";
  }

  function setCategorySearchFilter(filter: string): string {
    return filter !== "All" && filter !== ""
      ? ` AND category LIKE '${filter.replace(/'/g, "''")}' `
```

```
      : "";
  }

  function setSorting(filter: SortOption): string {
    switch (filter) {
      case "name-asc":
        return " ORDER BY name ASC";
      case "name-desc":
        return " ORDER BY name DESC";
      case "rating-asc":
        return " ORDER BY rating ASC";
      case "rating-desc":
        return " ORDER BY rating DESC";
      case "date-asc":
        return " ORDER BY updatedAt ASC";
      case "date-desc":
        return " ORDER BY updatedAt DESC";
      default:
        return " ORDER BY name ASC";
    }
  }
}

export default SQLiteQueries;
import { limit, where } from "firebase/firestore";
import FireStore from "../../services/firebase/Firestore";
import FireBaseAuth from "../../services/firebase/FireBaseAuth";

namespace FireStoreQueries {
  export async function getAllInUserCollection<T extends Model>(
    collectionName: string
  ): Promise<T[] | null> {
    const user = await FireBaseAuth.getCurrentUser();

    if (!user) {
      return null;
    }

    return FireStore.getItemsFromCollection<T>(collectionName, [
      where("userId", "==", user.id),
    ]);
  }

  export async function getUpdatedItems<T extends Model>(
    collectionName: string,
    userLastUpdate: Date,
    userId: string
  ): Promise<T[] | null> {
    return FireStore.getItemsFromCollection<T>(collectionName, [
      where("updatedAt", ">=", userLastUpdate),
      where("userId", "==", userId),
    ]);
  }
}

export default FireStoreQueries;
```

```
import { FirebaseError } from "firebase/app";
import { FireBaseErrorHandler } from "../../../services/firebase/FireBaseErrorHandler";
import { AuthorizationValiditor } from "../../../helpers/Validators/AuthorizationValidator";

export interface Errors {
  email: string;
  password: string;
}

export interface ErrorsStatus {
  isvalid: boolean;
  errors: Errors;
}

let handleValidationError = <T>(
  status: ErrorsStatus,
  property: keyof Errors,
  handler: (...args: T[]) => AuthorizationValiditor.ValidationStatus,
  params: T[]
) => {
  let res = handler(...params);
  status.isvalid = status.isvalid && res.isValid;
  status.errors[property] = res.errors || "";

  return status;
};

export let handleValidationErrors = (email: string, password: string) => {
  let status: ErrorsStatus = {
    isvalid: true,
    errors: {
      email: "",
      password: "",
    },
  };

  status = handleValidationError(status, "email", AuthorizationValiditor.validateEmail, [
    email,
  ]);

  status = handleValidationError(
    status,
    "password",
    AuthorizationValiditor.validateRequired,
    [password]
  );

  return status;
};

export let handleServerErrors = (res: unknown): ErrorsStatus => {
  let status: ErrorsStatus = {
    isvalid: true,
    errors: {
      email: "",
      password: "",
```

```
    },
  };

  if (!(res instanceof FirebaseError)) {
    return status;
  }

  let error = FireBaseErrorHandler.LoginErrors(res);

  if (error.type == "other") {
    alert(error.text);
  }

  status.isvalid = false;
  status.errors = { ...status.errors, [error.type]: error.text };
  return status;
};
import { ActivityIndicator, SafeAreaView, StyleSheet, Text } from "react-native";
import React, { useState, useEffect } from "react";
import AuthorizationNavigator from "./pages/Authorization/AuthorizationNavigator";
import { AccountProvider } from "./hooks/AccountContext";
import FireBaseAuth, { User } from "./services/firebase/FireBaseAuth";
import MainApp from "./pages/MainApp/RootAppNavigator";
import CacheProvider from "./services/chache/CacheProvider";
import NetInfo from '@react-native-community/netinfo';

export default function App(): JSX.Element {
  const [user, setUser] = useState<User | null>(null);
  const [loading, setLoading] = useState(true);
  const [isConnected, setIsConnected] = useState(false);

  useEffect(() => {
    const unsubscribe = FireBaseAuth.onAuthStateChanged((user) => {
      setUser(user);

      NetInfo.fetch().then(state => {
        console.log(state.isConnected);
        CacheProvider.syncDatabaseWithFireStore(state.isConnected || false)
        .then((res) => console.log(res))
        .catch((err) => console.log(err))
        .then((res) => FireBaseAuth.getCurrentUser().then((user) => console.log(user)));
      });

        setLoading(false);
    });

    return () => {
      unsubscribe();
    };
  }, []);

  if (loading) {
    return (
      <SafeAreaView style={styles.center}>
        <ActivityIndicator size="large" color="#0000ff" />
        <Text>Перевірка авторизації...</Text>
```

```
      </SafeAreaView>
    );
  }

  return (
    <>
      {!user ? (
        <AccountProvider>
          <AuthorizationNavigator></AuthorizationNavigator>
        </AccountProvider>
      ) : (
        <MainApp></MainApp>
      )}
    </>
  );
}

const styles = StyleSheet.create({
  center: {
    flex: 1     ,
    justifyContent: "center",
    alignItems: "center",
  },
});
```