Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

РОЗРОБКА МОБІЛЬНОГО ДОДАТКУ ДЛЯ ОПЕРАТИВНОЇ ПУБЛІКАЦІЇ ТА ЧИТАННЯ СТАТЕЙ З ВИКОРИСТАННЯМ KOTLIN ТА REALTIME DATABASE

DEVELOPMENT OF A MOBILE APPLICATION FOR RAPID ARTICLE PUBLICATION AND READING USING KOTLIN AND REALTIME DATABASE

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗс-21
Мороз Іван Степанович

Керівник:
Падалко Анатолій Михайлович

Кваліфікаційну роботу
допущено до захисту
«10» 06 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

_____

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *інженерії програмного забезпечення*
Ступінь вищої освіти *бакалавр*
Галузь знань: *12 «Інформаційні технології»*
Спеціальність: *121 «Інженерія програмного забезпечення»*
Освітня програма: «*Інженерія програмного забезпечення»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____
«20» 12 2024 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Морозу Івану Степановичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: *«Розробка мобільного додатку для оперативної публікації та читання статей з використанням Kotlin та Realtime Database».*
Керівник роботи: *к. ф. -м. н., доцент Падалко А. М.*
затверджені наказом закладу вищої освіти від *«19» грудня 2024 р.* № *474/01-02.*
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра *«10» червня 2025 р.*
3. Вихідні дані до роботи: *Kotlin (JDK 11+), Android Studio (Gradle), Firebase Authentication & Realtime Database, Moxy (MVP), RxJava3, Glide, Material Design Components, Git / GitHub Actions, Firebase Crashlytics, Firebase Performance Monitoring, R8/ProGuard, методичні вказівки до виконання кваліфікаційної роботи бакалавра.*
4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): *аналіз предметної області, аналіз існуючих рішеннь; вимоги та модельне проектування; архітектура системи та структура даних; технологічний стек; розробка ключових функціональних модулів; тестування та налагодження системи; підготовки інсталяційного пакету.*
5. Перелік графічного матеріалу: *14 рисунків, 3 додатки.*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	*Падалко А. М.*		
Специфікація вимог до розробленої системи	*Падалко А. М.*		
Розробка об'єкта проектування	*Падалко А. М.*		
Нормоконтроль	*Вознюк А. В.*		
Гарант ОП	*Ліщина Н. М.*		
Показник запозичень тексту	2,74 %		
Академічна доброчесність	*Падалко А. М.*		

7. Дата видачі завдання *«20» грудня 2024 р.*

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 04.02.2025 р.	
2	Аналіз проблеми розробки та впровадження об'єкту проектування	до 01.03.2025 р.	
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 15.03.2025 р.	
4	Розробка функціонально-структурної схеми роботи об'єкта проектування та проектування бази даних	до 29.03.2025 р.	
5	Практична реалізація об'єкта проектування та розробка бази даних	до 26.04.2025 р.	
6	Тестування та налагодження об'єкта проектування	до 03.05.2025 р.	
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру	до 10.06.2025 р.	

Здобувач вищої освіти Іван МОРОЗ

Керівник кваліфікаційної роботи Анатолій ПАДАЛКО

АНОТАЦІЯ

Мороз І. С. Розробка мобільного додатку для оперативної публікації та читання статей з використанням Kotlin та Realtime Database. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2025.

Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків.

У першому розділі проведено аналіз предметної області мобільних додатків для публікації та читання статей, вивчено існуючі рішення і визначено доцільність створення нативного Android-додатку із підтримкою offline-режиму. У другому розділі сформульовано функціональні та нефункціональні вимоги, виконано модельне проєктування (UML-діаграми прецедентів, класів, послідовностей, активностей, компонентів та розгортання) і спроєктовано NoSQL-модель даних для Firebase Realtime Database. У третьому розділі реалізовано практичну частину: налаштовано середовище розробки в Android Studio із Gradle, створено механізм аутентифікації через Firebase Auth, побудовано стрічку статей і CRUD-інтерфейс із Moxy+RxJava, організовано обробку зображень Glide і застосовано Material Components для UI, а також проведено статичний аналіз, модульні та UI-тести й підготовлено релізні APK/AAB. У висновках підкреслено, що реалізоване рішення є надійним, продуктивним і готовим до публікації та подальшого масштабування.

Ключові слова: Kotlin, Android Studio, Firebase Realtime Database, Moxy, RxJava, Glide, Material Design, MVP, offline.

ABSTRACT

Moroz S. Development of a Mobile Application for Rapid Article Publication and Reading Using Kotlin and Realtime Database. Manuscript. Qualification work of the bachelor's program "Software engineering" in the specialty "Software engineering". Lutsk National Technical University. Lutsk, 2025.

The bachelor's qualification thesis consists of an introduction, three chapters, conclusions, a list of references, and appendices.

In the first chapter, an analysis of the domain of mobile applications for publishing and reading articles was conducted, existing solutions were reviewed, and the feasibility of developing a native Android application with offline support was established. In the second chapter, functional and non-functional requirements were defined, model design was carried out (including UML use-case, class, sequence, activity, component, and deployment diagrams), and a NoSQL data model for Firebase Realtime Database was architected. In the third chapter, the practical implementation was completed: the development environment was configured in Android Studio with Gradle; an authentication mechanism using Firebase Auth was implemented; an article feed and CRUD interface leveraging Moxy + RxJava were built; image handling via Glide and UI components following Material Design guidelines were integrated; static analysis, unit and UI testing were performed; and release-ready APK and AAB packages were prepared. The conclusions highlight that the resulting solution is robust, performant, and ready for publication and future scaling.

Keywords: Kotlin, Android Studio, Firebase Realtime Database, Moxy, RxJava, Glide, Material Design, MVP, offline.

ЗМІСТ

ВСТУП

Актуальність теми. Сучасне суспільство дедалі більше орієнтується на мобільні пристрої як на основний канал споживання інформації. Кількість користувачів смартфонів в Україні та світі зростає щороку, а взаємодія зі цифровим контентом у режимі «тут і зараз» стає нормою. У таких умовах оперативна публікація та миттєвий доступ до актуальних статей є критично важливими як для медіа, так і для спеціалізованих галузевих порталів: від новинних агентств до освітніх та наукових спільнот.

Використання традиційних підходів із періодичними оновленнями контенту часто не може задовольнити вимоги користувачів, які очікують миттєвого доступу до нової інформації. Реалізація мобільного додатку із синхронізацією через Realtime Database дозволяє забезпечити безперервний потік даних, мінімізуючи затримки між публікацією матеріалу та його відображенням у клієнтському інтерфейсі.

Мова Kotlin, що офіційно підтримується Google як основна для Android-розробки, надає розробникам зручні сучасні конструкції та високу продуктивність. У поєднанні з хмарними сервісами Firebase Realtime Database, яка забезпечує автоматичне оновлення та зберігання даних у реальному часі, це створює потужний інструментарій для розробки кросплатформних додатків із мінімальними затратами часу на налаштування серверної інфраструктури.

Метою кваліфікаційної роботи бакалавра є розробка кросплатформного мобільного додатку для оперативної публікації та читання статей із використанням мови програмування Kotlin та Firebase Realtime Database.

Об’єктом кваліфікаційної роботи є мобільний додаток, який забезпечує взаємодію авторів і читачів через мобільний інтерфейс для створення, публікації та перегляду текстових матеріалів у режимі реального часу.

Предметом кваліфікаційної роботи бакалавра є практична реалізація функціоналу мобільного додатку на Kotlin, інтеграція з Firebase Realtime

Database для зберігання та миттєвої синхронізації контенту, а також розробка інтерфейсу користувача для швидкого доступу до статей.

Для досягнення поставленої мети були сформульовані такі завдання:

– проаналізувати предметну область і існуючі рішення;

– визначити вимоги та виконати модельне проектування;

– спроектувати архітектуру системи та структуру даних;

– обґрунтувати вибір технологічного стеку;

– розробити ключові функціональні модулі;

– провести тестування та налагодження;

– підготувати інсталяційний пакет.

Отже, розробка мобільного додатку для оперативної публікації та читання статей із використанням Kotlin та Realtime Database є актуальним напрямом, що відповідає сучасним вимогам до швидкості, зручності та надійності інформаційних сервісів. Такий додаток надає можливість авторам оперативно ділитися новинами й аналітикою, а читачам – миттєво отримувати найсвіжішу інформацію прямо на свій смартфон.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

У сучасному мобільному середовищі смартфони стали основним каналом доступу до цифрового контенту. За даними DataReportal, щоденний рівень інтернет-активності в Україні у 2024 році перевищив 80 %, із помітним зростанням частки користувачів, які споживають новини саме через мобільні пристрої [1]. При цьому StatCounter фіксує, що понад 28 % веб-трафіку в Україні припадає на мобільні екрани з роздільною здатністю 393×873 і подібними, що свідчить про різноманіття сегментів смартфонів і необхідність адаптивного дизайну під різні форм-фактори [2].

Ринок мобільних новинних додатків демонструє стійке зростання: за оцінками Market.US, глобальний обсяг цього ринку збільшиться з $ 1,3 млрд у 2023 році до $ 4,2 млрд до 2033 року при CAGR 12,5 % (рис. 1.1) [3].

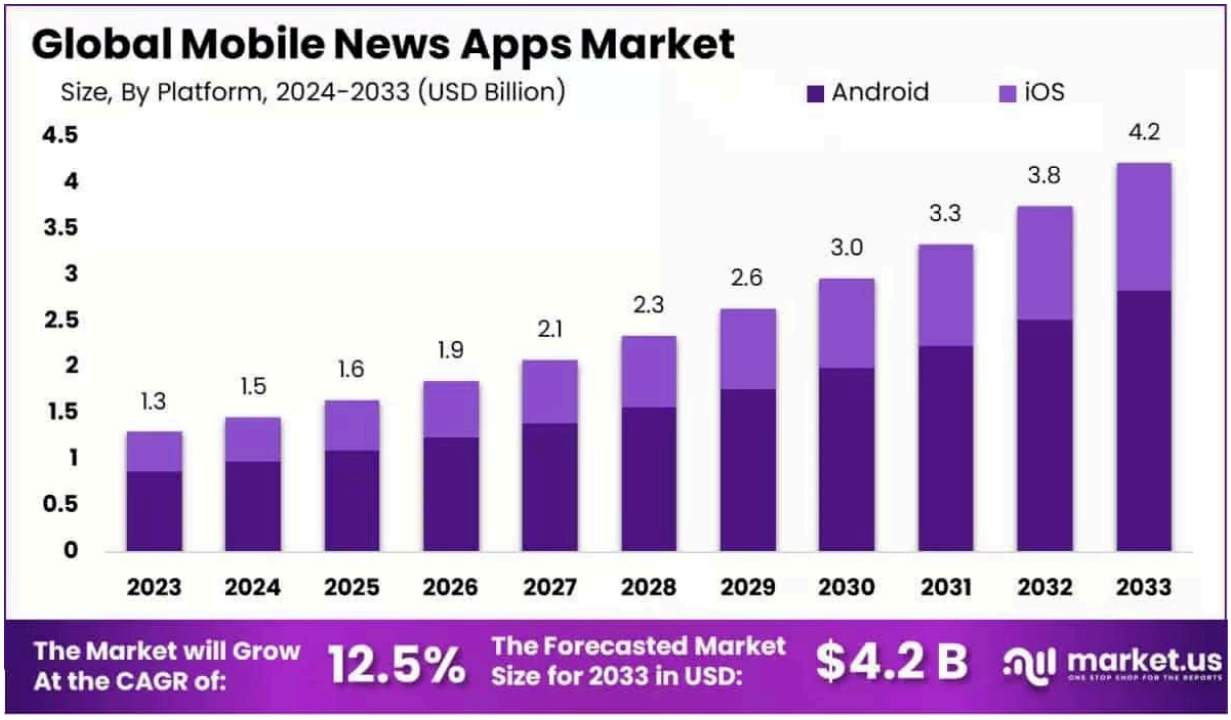

Рисунок 1.1 – Глобальний ринок мобільних новинних додатків [3]

Аналогічно, OpenPR прогнозує, що доходи індустрії лавиноподібно зростуть із $ 14,14 млрд у 2024 році до $ 15,51 млрд у 2025-му [4]. Додатки типу Google News, Flipboard чи Medium утримують високу залученість: середній час сесії в таких програмах перевищує 30 хвилин щодня [5].

Серед існуючих аналогів заслуговують уваги рішення, реалізовані в The Guardian: оновлена мобільна версія містить «текст-у-мову», подкасти та інтерактивні елементи, проте важить понад 80 МБ і потребує значних ресурсів пам’яті й батареї. Інші популярні додатки, такі як BBC News або CNN, також пропонують оновлення у режимі реального часу й складну ієрархію категорій, що іноді ускладнює навігацію й знижує швидкість завантаження на слабких пристроях.

Водночас багато нещодавніх проектів використовують гібридні чи веб-базовані підходи (PWA) для дружності до різних платформ, але вони страждають від обмежень кешування та не забезпечують миттєвого оновлення на рівні natve-рішень. Складність налаштування push-сповіщень і offline-модулів у власноруч збудованих рішеннях може перевищувати технічні можливості невеликих команд, особливо коли потрібно швидко виводити нові матеріали.

На цьому тлі використання Firebase Realtime Database виглядає доцільним: вона забезпечує миттєву двонаправлену синхронізацію даних та вбудовану підтримку offline-режиму без необхідності розгортання власної серверної інфраструктури. Об’єднання Kotlin і Realtime Database дозволяє створювати легкі, але потужні Android-додатки, які автоматично отримують оновлення та економлять ресурси пристрою. Це обґрунтовує необхідність розробки нового мобільного додатку, що поєднує простоту розгортання й високу швидкість реакції на зміну контенту, задовольняючи запити сучасної аудиторії.

У ході аналізу літературних джерел було виявлено кілька ключових публікацій, що висвітлюють особливості використання Kotlin та Firebase Realtime Database для розробки мобільних додатків. У статті «Kotlin with

Firebase: Building a Simple Database App» [6] на платформі Medium автор демонструє, як всього кількома рядками коду можна налаштувати запис і читання даних у Realtime Database, обробити події оновлення й забезпечити двонаправлену синхронізацію між клієнтом і сервером. Зокрема, увагу приділено використанню ValueEventListener для миттєвого реагування на зміни в БД і оптимізації запитів через запити з фільтрацією даних прямо в Kotlin-коді.

Більш широке бачення надає оглядова стаття «A Review on Firebase (Backend as A Service) for Mobile Application Development» [7] на ResearchGate, де автори розглядають архітектуру Firebase як сервісу, аналізують його компоненти (Auth, Realtime Database, Storage) та порівнюють із традиційними серверними рішеннями. У роботі виділено переваги Realtime Database – відсутність необхідності в розгортанні власних серверів до вбудованого offline-підтримки – і доведено, що для невеликих проєктів та прототипів Firebase є оптимальним вибором завдяки швидкому старту і масштабуванню по вимозі.

У кейс-дослідженні «Implementation of Firebase in the Development of Android-based Queue Reservation and Treatment Record Applications» [8] також на ResearchGate розглянуто практичну реалізацію медичного додатку з використанням Realtime Database і Firebase Auth. Автори описують, як завдяки синхронізації в реальному часі вдалося організувати чергу пацієнтів без затримок і забезпечити коректну обробку подій у середовищі з високим рівнем паралелізму запитів.

У статті «Firebase, Firebase Realtime Database and Firestore» [9] на Medium наведено порівняння двох NoSQL-рішень від Firebase: класичної Realtime Database та новішої Cloud Firestore. Автор розкриває сценарії, у яких доцільніше використовувати кожну з баз: Realtime Database краще підходить для простих деревоподібних даних із високими вимогами до latency, тоді як Firestore забезпечує більш гнучкі запити та кращу масштабованість на великі обсяги даних.

Нарешті, гайд «Building Real-Time Apps with Firebase Realtime Database and Kotlin» [10] від Stackademic Blog надає глибокий технічний аналіз: описано оптимізацію запитів, організацію потоків даних через корутини Kotlin, налаштування кешу й offline-режиму, а також особливості обробки конфліктів при одночасному редагуванні даних на різних пристроях. Цей матеріал слугує відправною точкою для вибудови складних реальних застосунків із мінімальною затратою часу на налаштування інфраструктури.

Таким чином, існуючі публікації підтверджують ефективність поєднання Kotlin і Firebase Realtime Database для створення високопродуктивних і користувацько-орієнтованих мобільних додатків, що оновлюють контент у режимі реального часу. Накопичений досвід і показані в статтях методики лягають в основу розробки власного Android-додатку для оперативної публікації й читання статей.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

Метою кваліфікаційної роботи бакалавра є розробка кросплатформного мобільного додатку для оперативної публікації та читання статей із використанням мови програмування Kotlin та Firebase Realtime Database.

Для досягнення поставленої мети в роботі було сформульовано такі завдання:

– проаналізувати предметну область і існуючі рішення;

– визначити вимоги та виконати модельне проектування;

– спроектувати архітектуру системи та структуру даних;

– обґрунтувати вибір технологічного стеку;

– розробити ключові функціональні модулі;

– провести тестування та налагодження;

– підготувати інсталяційний пакет.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОГО МОБІЛЬНОГО ДОДАТКУ

2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення

Визначення вимог до програмного забезпечення є одним із найважливіших етапів життєвого циклу проєкту, оскільки саме на цьому кроці формуються чіткі очікування від системи, закладається фундамент для архітектурного проєктування та тестування. У мобільному додатку для публікації й читання статей необхідно з одного боку забезпечити повний набір функціональних можливостей – від реєстрації користувачів до створення й відображення статей у реальному часі, а з іншого – дотриматися жорстких критеріїв якості: продуктивності, безпеки та зручності використання в умовах мобільних пристроїв.

Функціональні вимоги:

– реєстрація та аутентифікація. Додаток реалізує створення облікового запису через Firebase Auth (email/пароль) та вхід із перевіркою достовірності даних;

– управління ролями. Система розрізняє ролі «автора» та «читача» – автор отримує доступ до створення, редагування й видалення власних статей, тоді як читач може лише переглядати контент;

– CRUD-операції зі статтями. Автор може створювати нові публікації (заголовок, текст, зображення), редагувати й видаляти їх через інтуїтивні інтерфейси;

– перегляд списків статей. Інтерфейс відображає загальну стрічку всіх статей із мінімальною інформацією (заголовок, автор, дата), а також окремий розділ «Мої статті» для авторів;

– оперативна синхронізація. Зміни в базі (додавання, редагування, видалення) мають миттєво відображатися в інтерфейсі без перезапуску додатку завдяки Firebase Realtime Database;

– детальний перегляд. При виборі статті відкривається екран із повним текстом, метаданими та кнопкою «Назад» до списку;

– пошук і фільтрація. Користувачі можуть шукати матеріали за заголовком або ключовими словами в тілі статті;

– профіль користувача. Користувач може переглядати та редагувати власний профіль: змінювати аватар, ім’я й опис;

– offline-режим. Додаток зберігає останні отримані дані в локальному кеші й відображає їх при відсутності мережі, а після відновлення зв’язку автоматично синхронізує зміни.

Нефункціональні вимоги.

– продуктивність. Сторінка зі списком статей завантажується за не більше ніж 2 секунди, оновлення даних виконується за 1 секунду;

– надійність. Після втрати зв’язку додаток відновлює роботу без втрати даних, використовуючи локальний кеш і автоматичне повторне підключення до бази;

– безпека. Весь трафік шифрується через HTTPS; правила доступу реалізовані в Firebase Security Rules, гарантують, що автор може змінювати тільки свої статті;

– сумісність. Додаток підтримує Android 6.0 (API 23) і вище та адаптується до екранів різних розмірів;

– Usability. Інтерфейс виконано за гайдлайном Material Design із чіткими кнопками, зрозумілими підказками та підтримкою темної/світлої тем;

– масштабованість. Використання MVVM-патерну і Firebase дозволяє легко додавати нові функції та обробляти зростаючу кількість користувачів;

– підтримуваність. Код організовано модульно з застосуванням Dependency Injection (Hilt); проект супроводжується документацією та юніт-тестами для ViewModel і моделей;

– доступність. Підтримка збільшення шрифту відповідно до налаштувань пристрою та зрозумілі альтернативні підписи для зображень;

– моніторинг. Інтеграція Firebase Crashlytics для збору даних про збої та Firebase Analytics для відстеження користувацької активності.

Ці вимоги лягли в основу подальшого проектування UML-діаграм, інформаційних потоків і UX/UI-прототипів, що забезпечить високу якість, безпеку та зручність майбутнього додатку.

Діаграма прецедентів (Use-Case) відображає усі основні сценарії взаємодії двох типів користувачів – «Автор» і «Читач» – з системою: реєстрація/вхід, створення/редагування/видалення статей, перегляд стрічки, пошук, управління профілем, робота в offline-режимі (рис. 2.1).

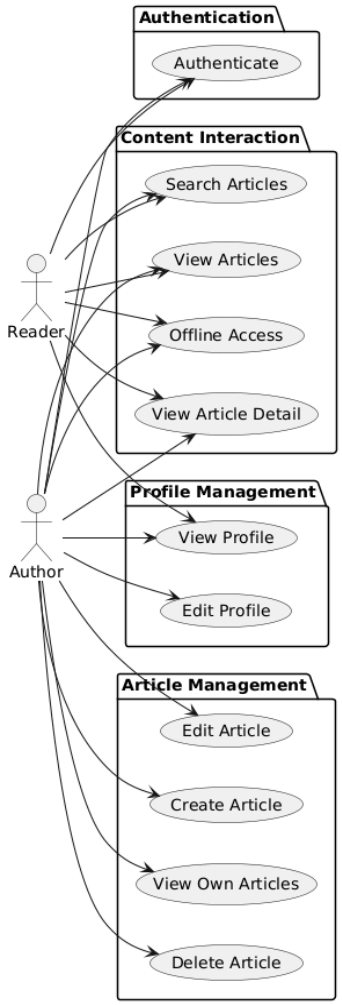

Рисунок 2.1 – Діаграма прецедентів (Use-Case)

Діаграма прецедентів відображає основні сценарії взаємодії двох категорій користувачів із мобільним додатком.

Reader (Читач) може:

– Authenticate (UC1) – пройти реєстрацію та вхід через Firebase Auth;

– View Articles (UC2) – переглянути список усіх доступних статей;

– Search Articles (UC3) – здійснити пошук за заголовком або ключовими словами;

– View Article Detail (UC4) – відкрити повний текст конкретної статті;

– Offline Access (UC5) – працювати з останнім кешованим набором статей без мережі;

– View Profile (UC6) – переглянути власний профіль.

Author (Автор), окрім того, що має всі дії читача, також може:

– Edit Profile (UC7) – редагувати свої персональні дані;

– Create Article (UC8) – додавати нові публікації з текстом та зображеннями;

– Edit Article (UC9) – змінювати власні статті після публікації;

– Delete Article (UC10) – видаляти свої пости;

– View Own Articles (UC11) – переглядати відособлений список лише власних матеріалів.

Сценарії організовані в чотири логічні блоки:

– Authentication – вся взаємодія, пов’язана з реєстрацією та вхідними даними;

– Content Interaction – дії із читанням і пошуком матеріалів у реальному та офлайн-режимах;

– Profile Management – перегляд і редагування особистої інформації;

– Article Management – повний цикл CRUD-операцій для авторів.

Клас-діаграма (Class Diagram) описує ключові сутності: User (з атрибутами і ролями), Article (заголовок, текст, зображення, таймстемп), Profile, а також сервіси/компоненти для кешування (OfflineCache) та синхронізації (RealtimeSync) (рис. 2.2).

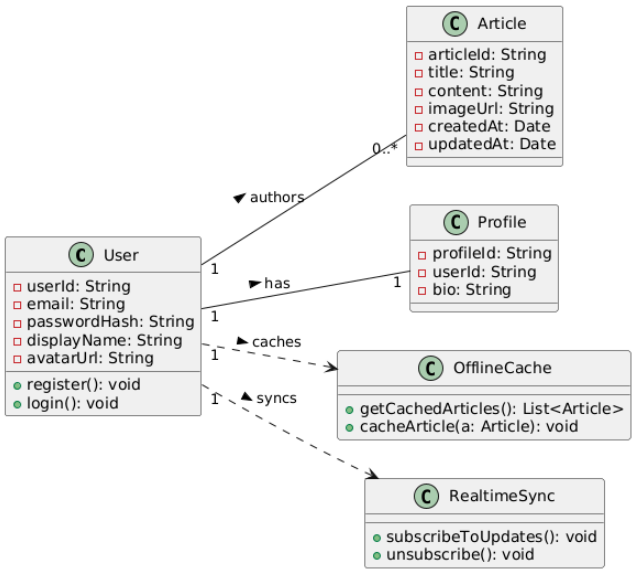

Рисунок 2.2 – Клас-діаграма (Class Diagram)

Діаграма класів відображає ключові сутності мобільного додатку та їх взаємозв’язки:

– User – центральний клас, що містить інформацію про користувача: ідентифікатор, email, хеш пароля, відображуване ім’я та посилання на аватар. У ньому визначені методи register() і login(), які відповідають за створення облікового запису та аутентифікацію;

– Article – клас статті з атрибутами articleId, title, content, imageUrl, createdAt та updatedAt. Зв’язок 1 – 0..* між User та Article вказує, що один користувач (Author) може бути автором багатьох статей;

– Profile – додаткова інформація про користувача (біографія), пов’язана з класом User через відношення 1 – 1. Це окремий об’єкт, що розвантажує клас User і містить лише поля профілю;

– OfflineCache – сервісний клас без стану користувача, який забезпечує методи getCachedArticles() для отримання раніше збережених статей і cacheArticle(a: Article) для зберігання нових даних локально. Взаємодіє з User через позиційну залежність, показуючи, що кеш належить поточному користувачу;

– RealtimeSync – клас, що відповідає за підписку (subscribeToUpdates()) та відписку (unsubscribe()) від оновлень у Realtime Database. Аналогічно до кешу, пов’язаний із User залежністю «syncs», що ілюструє механізм автоматичної синхронізації даних.

Ця структура підтримує розділення відповідальностей: класи даних (User, Article, Profile) містять лише атрибути й базові методи, а класи сервісів (OfflineCache, RealtimeSync) інкапсулюють логіку зберігання та синхронізації. Такий підхід полегшує розширення функціоналу і забезпечує чисту архітектуру MVVM у мобільному додатку.

Діаграма послідовностей (Sequence Diagram) деталізує протікання конкретних сценаріїв, наприклад публікацію нової статті або оновлення стрічки: виклик UI → ViewModel → Firebase SDK → Realtime Database → колбеки оновлення (рис. 2.3).

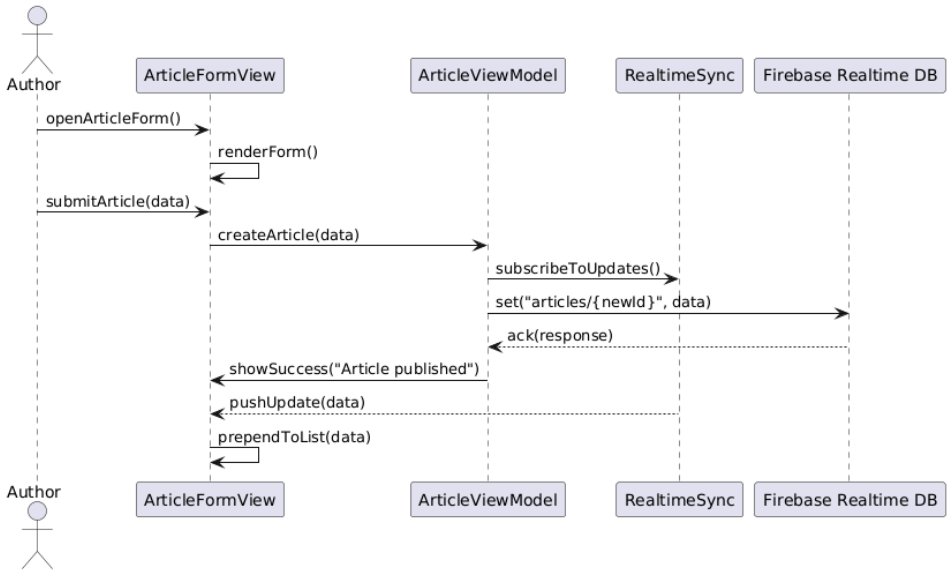

Рисунок 2.3 – Діаграма послідовностей (Sequence Diagram)

Дана діаграма послідовностей ілюструє процес публікації нової статті автором у мобільному додатку. Спочатку автор активує екран форми статті, викликавши openArticleForm() у компоненті ArticleFormView, після чого інтерфейс рендериться (renderForm()). Після заповнення полів автор надсилає

дані через метод submitArticle(data), який передає інформацію до ArticleViewModel за допомогою виклику createArticle(data).

Далі ViewModel ініціює підписку на оновлення, викликаючи сервіс RealtimeSync.subscribeToUpdates(), а потім записує нову статтю в Firebase Realtime Database через set("articles/{newId}", data). Після успішного збереження база відсилає підтвердження (ack(response)), і ViewModel повідомляє компонент View про результат виконання, викликаючи showSuccess("Article published").

Одночасно сервіс RealtimeSync передає оновлені дані назад до ArticleFormView за допомогою pushUpdate(data), що дозволяє відобразити щойно опубліковану статтю у стрічці. Компонент оновлює список викликом prependToList(data), гарантуючи, що новий запис з’явиться на початку переліку без необхідності перезавантаження додатку.

Така послідовність операцій забезпечує безшовну інтеграцію між UI, бізнес-логікою та базою даних, дозволяє користувачам миттєво бачити результати власних дій і підтримує актуальність контенту в реальному часі.

Діаграма активності (Activity Diagram) ілюструє алгоритм синхронізації в offline-режимі або логіку CRUD-операцій із перевіркою ролі та мережевого статусу (рис. 2.4).

Спочатку відображається екран із переліком статей, куди завантажуються дані з локального кешу та Firebase Realtime Database. При наявності мережевого з’єднання ініціюється підписка на оновлення (Sync.subscribeToUpdates()), яка увімкнена у циклі очікування нових даних. Коли з’являється оновлення, локальний кеш і інтерфейс одразу оновлюються, забезпечуючи миттєвий доступ до нових матеріалів. Якщо мережа відсутня, користувач бачить лише кешовані статті без блокувань. Після цього користувач може обрати конкретну статтю, і система переведе його на екран детального перегляду. Такий підхід гарантує безперебійну роботу додатка в різних умовах мережі та підтримує швидкий доступ до контенту.

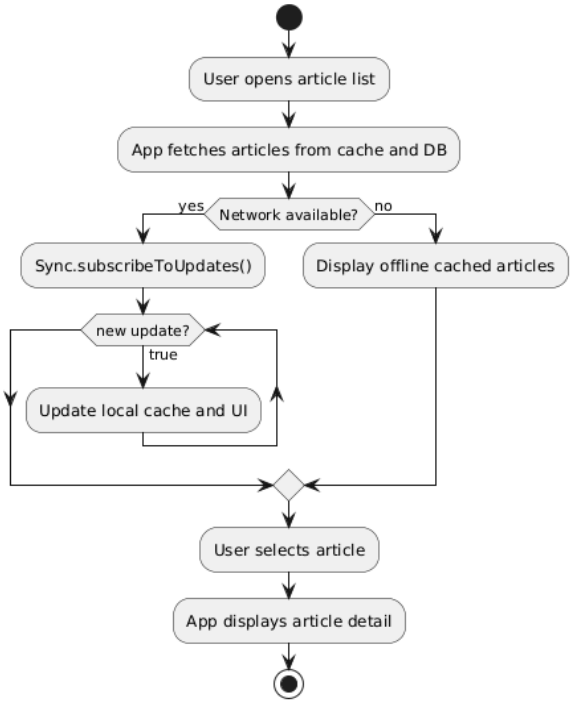

Рисунок 2.4 – Діаграма активності (Activity Diagram)

Діаграма компонентів (Component Diagram) показує високорівневу архітектуру: мобільний клієнт на Kotlin із модулями UI, ViewModel, DataLayer, а також зовнішню службу Firebase (Auth, Realtime DB, Crashlytics) (рис. 2.5).

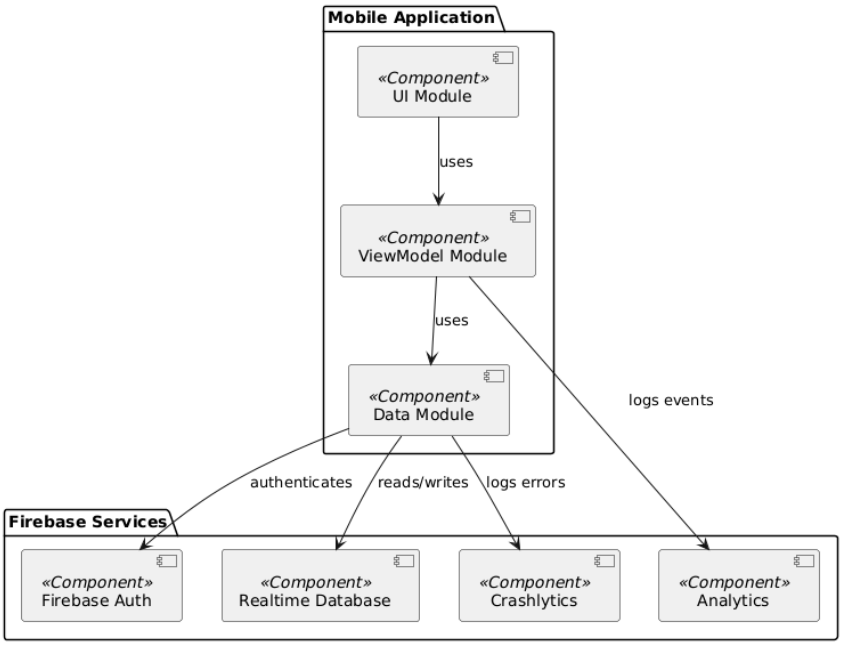

Рисунок 2.5 – Діаграма компонентів (Component Diagram)

Діаграма компонентів ілюструє розподіл основних модулів мобільного застосунку та їхню взаємодію з сервісами Firebase. Всередині пакету Mobile Application розміщені три ключові компоненти:

– UI Module – відповідає за інтерфейс користувача, рендеринг екранів та обробку подій натискання;

– ViewModel Module – забезпечує бізнес-логіку, трансформацію даних для UI та передачу подій у шари даних;

– Data Module – відповідає за роботу з джерелами даних: звернення до Firebase Auth для автентифікації, до Realtime Database для зчитування й запису статей, а також інтеграцію з Crashlytics для логування помилок.

У пакеті Firebase Services представлені зовнішні сервіси, що надають функціональні можливості бекенду:

– Firebase Auth для реєстрації й аутентифікації користувачів;

– Realtime Database як основна система зберігання й синхронізації даних у реальному часі;

– Crashlytics для збору та аналізу аварійних подій у додатку;

– Analytics для відстеження ключових подій користувацької взаємодії.

Взаємодія між модулями реалізована стрілками «uses», які вказують на напрямок виклику: UI Module звертається до ViewModel Module, ViewModel – до Data Module, а Data Module взаємодіє з Firebase сервісами для виконання CRUD-операцій, автентифікації та логування. Така архітектура забезпечує модульність, полегшує тестування окремих компонентів та дозволяє розширювати систему новими сервісами без суттєвих змін у коді клієнта.

Діаграма розгортання (Deployment Diagram) відображає розміщення компонентів: Android-пристрій (додаток), Firebase Cloud (сервер Auth, Realtime DB), мережеві з’єднання (HTTPS) (рис. 2.6).

На боці клієнта розміщено Android Device, на якому працює MChat App. Додаток взаємодіє через захищене HTTPS-з’єднання з сервісами Firebase Auth Server для автентифікації та отримання JWT-токена, а також з Realtime Database Server для зчитування, запису та підписки на оновлення статей у реальному

часі. Така схема забезпечує відокремлення клієнтської логіки від хмарних сервісів, що спрощує масштабування і гарантує безпеку даних при передачі.

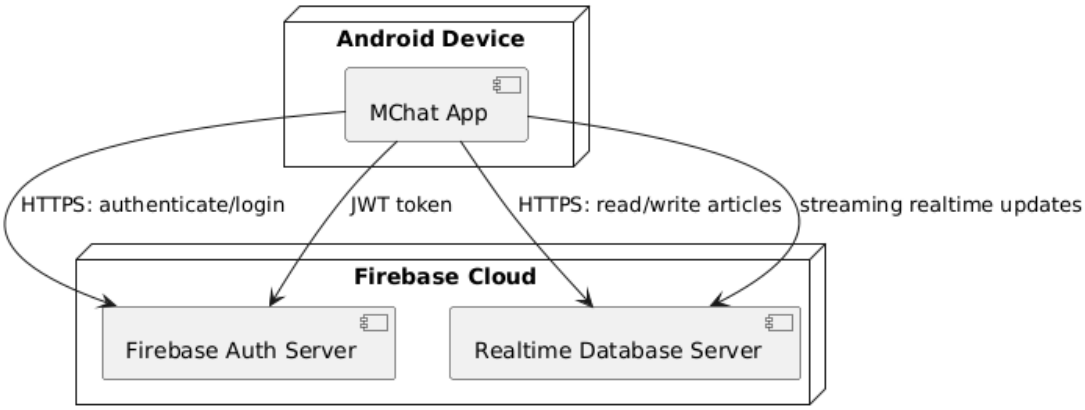

Рисунок 2.6 – Діаграма розгортання (Deployment Diagram)

При розробці UX/UI було проведено детальний аналіз потреб користувачів і побудовано інформаційні потоки між ключовими екранами додатку (екран автентифікації, список статей, детальний перегляд, форма публікації, профіль). Для кожного екрану визначено набір функціональних можливостей – від основних елементів навігації та форм введення до відображення станів завантаження й помилок – та описано, як користувачі переходять від однієї дії до іншої. Інформаційні потоки спроектовано з урахуванням двонаправленої синхронізації: дії користувача в UI передаються через ViewModel до Data Module, де відбуваються операції з Realtime Database, а отримані оновлення в реальному часі миттєво повертаються на екран. Технологічно для інтерфейсу застосовано гайдлайни Material Design із акцентом на простоту, контрастність та читабельність тексту, а для інформаційного обслуговування – механізми локального кешу і push-сповіщень, що гарантують доступ до контенту за будь-яких умов мережі. Такий підхід забезпечує інтуїтивну навігацію, швидкий доступ до статей та відповідність сучасним очікуванням користувацького досвіду.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Вибір технологічного стеку є ключовим етапом у розробці будь-якого програмного продукту, оскільки від нього залежить швидкість реалізації, продуктивність, підтримуваність та масштабованість рішення. У випадку мобільного додатку для оперативної публікації та читання статей правильне поєднання мови програмування, середовища розробки, бази даних і бібліотек визначає не лише зручність роботи команди розробників, але й кінцевий досвід користувачів. Невдалий вибір може призвести до затримок із випуском оновлень, збоїв під навантаженням або складнощів у підтримці старих версій.

Kotlin остаточно утвердився як провідна мова для розробки Android-додатків: за офіційними даними Kotlinlang.org, більш ніж 50 % професійних Android-розробників сьогодні використовують Kotlin як основну мову, тоді як Java утримує лише близько 30 % ринку [11]. Цей показник демонструє не лише швидку міграцію ком’юніті до нових підходів, а й визнання переваг Kotlin у продуктивності та безпеці.

Детальніше, у сегменті мобільної розробки поширеність Kotlin досягає 85 % у порівнянні з 60 % у Java, що свідчить про чітку перевагу Kotlin серед інструментів саме для Android-екосистеми (рис. 2.7) [12]. Це зростання відбулося після того, як Google у 2019 році офіційно рекомендувала Kotlin як «першокласну» мову для Android, що дало потужний імпульс до її впровадження у проєктах будь-якого масштабу.

З іншого боку, аналіз репозиторіїв GitHub у рамках Octoverse 2024 показав, що приблизно 60-70 % нових Android-проєктів використовують Kotlin у своїй кодовій базі, що підтверджує тренд на поступову відмову від Java в новобудовах і стартових шаблонах [13]. Така ситуація створює великий пул готових бібліотек і навчальних матеріалів, підтримуваних спільнотою.

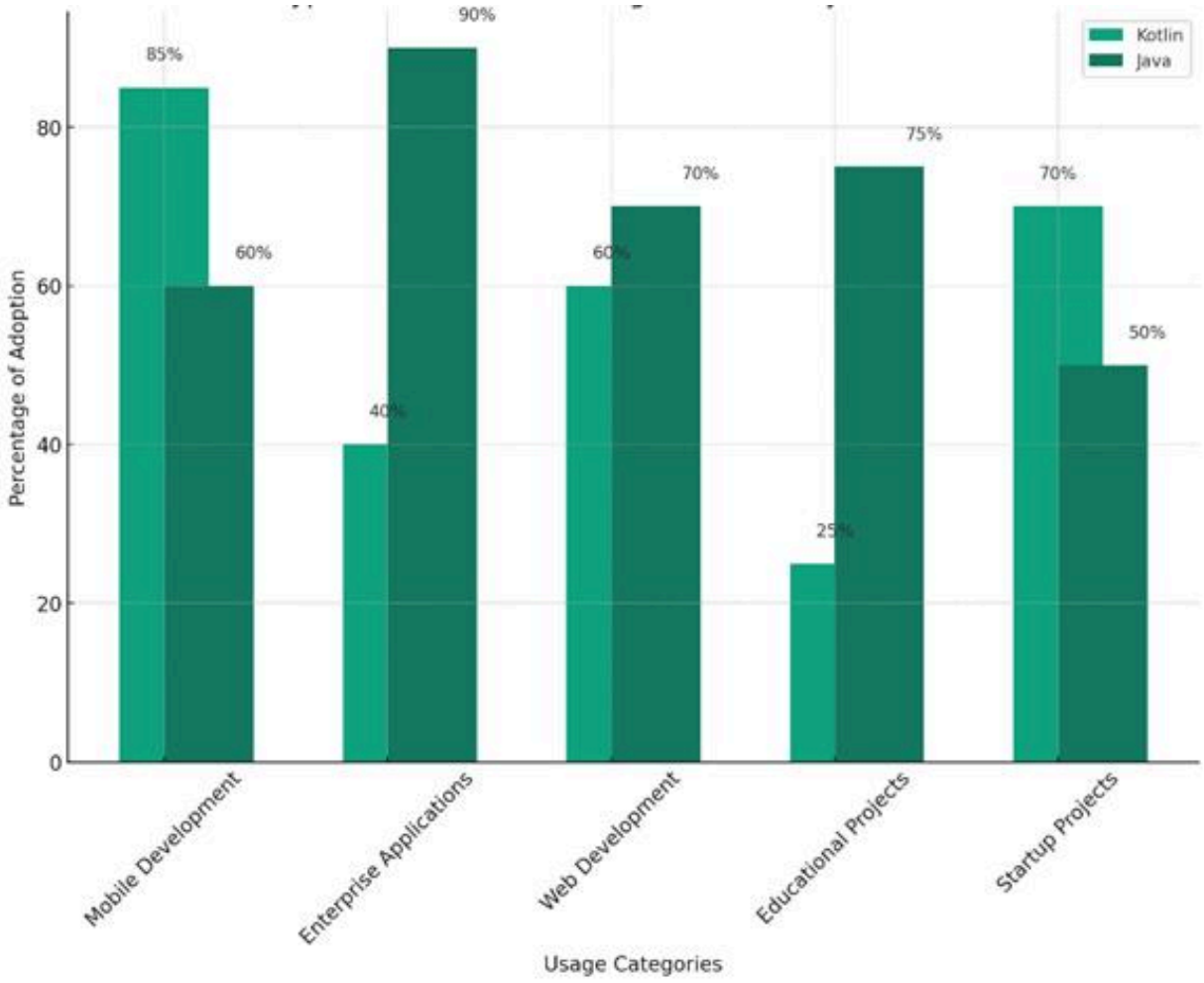

Рисунок 2.7 – Використання мов програмування Java та Kotlin у розробці програмного забезпечення [12]

Головними перевагами Kotlin вважають вбудовану «null-safety», можливість «корутин» для зручної асинхронної обробки, лаконічний синтаксис із меншою кількістю «шаблонного» коду та повну «інтероперабельність із Java». У поєднанні з багатою екосистемою Android Jetpack це дозволяє розробникам швидше впроваджувати нові фічі, знижуючи ймовірність помилок і прискорюючи рефакторинг.

У порівнянні з іншими кросплатформенними рішеннями, такими як Flutter (Dart) чи React Native (JavaScript), Kotlin/Android Studio забезпечує нативну продуктивність та меншу вагу APK: Flutter часто критикують за розмір бінарника (часто понад 8 MB) і складність інтеграції з нативними SDK, а React

Native – за додатковий шар «бриджу» між JS і нативним кодом, що може призводити до латентності в UI-реакціях. Використання Kotlin дає змогу уникнути цих компромісів, зберігаючи високу швидкість розробки й відгуку додатку без додаткових абстракцій.

Android Studio обрано як основне середовище розробки завдяки його офіційному статусу та глибокій інтеграції з Android SDK і Gradle. Побудоване на базі IntelliJ IDEA, воно забезпечує інструменти для всіх етапів розробки: візуальний редактор макетів із підтримкою ConstraintLayout, емулятори із різними конфігураціями пристроїв, профайлери CPU і пам’яті та потужну систему налагодження з можливістю «hot swap» коду [14].

Завдяки плагіну Kotlin Support, Android Studio надає повноцінну IDE-підтримку мови: миттєве автодоповнення, рефакторинг, перевірка null-safety і вбудовану систему шаблонів для коду, що прискорює написання й вдосконалення бізнес-логіки. Вбудована екосистема плагінів (Firebase Assistant, Hilt, SQLDelight тощо) дозволяє безшовно додавати функціонал без виходу з IDE і налаштовувати робоче місце під індивідуальні вимоги проєкту.

Система збірки Gradle, інтегрована в Android Studio, керує залежностями та конфігурацією збірок (build variants, product flavors), підтримує інкрементальні збірки та паралельне виконання задач, що значно скорочує час компіляції великих кодових баз. Крім того, Gradle дозволяє застосовувати ProGuard/R8 для обфускації й мінімізації APK при підготовці release-версій.

Використання Android Studio + Gradle гарантує уніфіковане середовище для всієї команди: просте налаштування CI/CD за допомогою GitHub Actions або Jenkins, стандартизовані скрипти для запуску юніт- і інтеграційних тестів, а також прозоре керування SDK-компонентами та плагінами через Gradle Wrapper. Це поєднання забезпечує швидкий цикл розробки «код-збірка-налагодження» та дозволяє підтримувати високий рівень продуктивності й якості Android-додатку.

Вибір Firebase Realtime Database обґрунтовано її можливостями для миттєвої двонаправленої синхронізації даних та вбудованої підтримки

offline-режиму без необхідності розгортання власних серверів. На відміну від Cloud Firestore, Realtime Database оптимізована під прості деревоподібні схеми даних і забезпечує нижчу латентність оновлень, що критично для стрічки статей і чату в режимі реального часу [15].

Вбудовані Firebase Security Rules дають можливість строго регламентувати доступ: користувачі можуть читати тільки дозволені записи, а автори – змінювати лише свої статті, що підвищує рівень безпеки даних без додаткових серверних налаштувань . Крім того, автоматичне кешування на клієнті та механізм повторного підключення забезпечують стабільність роботи додатка при втраті зв'язку і прозору синхронізацію після відновлення мережі.

Завдяки цим властивостям Firebase Realtime Database дозволяє скоротити час розробки, зосередившись на бізнес-логіці та UI, і гарантує високу швидкість та надійність оновлення контенту в мобільному додатку.

Для реалізації аутентифікації користувачів було обрано Firebase Authentication – готове рішення «Identity as a Service», яке забезпечує повний цикл реєстрації та входу без необхідності створювати власний серверний модуль. Firebase Auth підтримує стандартні методи входу через email/пароль, телефонні номери та популярні соціальні провайдери (Google, Facebook, Twitter, GitHub) за допомогою інтеграції OAuth 2.0 й OpenID Connect [16].

SDK Firebase Auth надає:

– бекенд-сервіси та UI-компоненти для швидкого старту. Drop-in рішення FirebaseUI Auth обробляє потоки реєстрації, відновлення пароля та входу через провайдерів (із підтримкою стандартних сценаріїв та обробкою помилок);

– безпечне зберігання токенів. Після успішного входу клієнт отримує ID-токен (JWT), який автоматично додається до кожного запиту до Realtime Database, забезпечуючи авторизацію на рівні правил доступу;

– розширені можливості (по мірі масштабування проєкту). Багатофакторна аутентифікація, блокування функцій (blocking functions), аудит логів, SAML/OpenID Connect для корпоративної інтеграції та мультиорендність – усе це стає доступним із модернізацією до Identity Platform.

Завдяки тісній інтеграції з Firebase Realtime Database правила безпеки (Firebase Security Rules) можуть обмежувати читання/запис на рівні користувача (auth.uid), що гарантує:

1) лише аутентифіковані користувачі можуть працювати з даними;

2) автори змінюють тільки свої статті;

3) дані інших користувачів залишаються недоступними.

Отже, Firebase Auth мінімізує витрати часу на розробку кастомних модулів безпеки, забезпечуючи при цьому надійну, масштабовану та добре документовану систему аутентифікації, що повністю відповідає вимогам нашого мобільного додатка.

Було обрано поєднання MVP-фреймворку Moxy та реактивної бібліотеки RxJava для організації чистої, масштабованої архітектури та зручної обробки асинхронних потоків.

Було обрано Moxy як інструмент реалізації патерну «Model-View-Presenter», оскільки він автоматично зберігає стан презентерів при зміні конфігурації екранів (наприклад, поворот мобільного пристрою) і делегує виклики View через проксі-інтерфейси. Це забезпечує чітке розділення бізнес-логіки (Presenter) від відображення (View), полегшує модульне тестування та знижує залежності між компонентами інтерфейсу.

Для реактивної обробки даних було обрано RxJava, яка пропонує велику кількість операторів (map, filter, debounce, flatMap тощо) для трансформації та комбінації стрімів подій. Використання RxJava у поєднанні з Kotlin дає змогу легко реалізувати асинхронні виклики до Firebase Realtime Database, обробку UI-подій і таймерів у єдиному стилі, гарантуючи високий рівень контролю над backpressure і продуктивністю.

Незважаючи на те, що в екосистемі Kotlin дедалі частіше застосовують корутини та Flow, RxJava було обрано через її зрілу спільноту, перевірені практики й широку інтеграцію з багатьма бібліотеками, що забезпечує готовність до складних сценаріїв обробки даних. Такий вибір гарантує

збереження стану UI при зміні конфігурації, простоту масштабування та можливість широкого покриття коду тестами.

Використання Material Design Components (MDC) обґрунтовано широкою підтримкою Google та приналежністю до одного з найвпізнаваніших дизайн-систем у світі. Завдяки чітким рекомендаціям із визуального стилю, анімацій і взаємодії MDC дозволяє створювати інтерфейси, які миттєво знайомі користувачам Android-пристроїв та відповідають очікуванням зручності й передбачуваності [17]. За даними дослідження Bardia Doosti et al., застосування компонентів Material Design корелює з вищими оцінками додатків і збільшеною кількістю встановлень, що підтверджує їхню ефективність у реальному середовищі [18].

У проєкті було обрано офіційну бібліотеку MDC Android, яка забезпечує готові реалізації таких елементів, як App Bar, Floating Action Button, Snackbar, а також універсальні компоненти вводу й навігації. Завдяки цій підходу інтерфейс додатку залишається однорідним і легким у підтримці, а команда розробників може швидко впроваджувати нові екрани, спираючись на перевірені патерни й оптимізовані компоненти.

Було обрано Git як систему контролю версій завдяки його статусу де-факто стандарту у світі розробки програмного забезпечення: за даними Eclipse Foundation та щорічних опитувань Stack Overflow, у 2022 році ним користувалися 93,9 % професійних розробників [19]. Git дозволяє зберігати повну історію змін, працювати з гілками та злиттями (merge) із мінімальною ймовірністю конфліктів і легко інтегрується з хостингом репозиторіїв на GitHub, GitLab чи Bitbucket. Завдяки власному механізму репозиторіїв із розподіленою архітектурою він забезпечує швидкі локальні операції (commit, diff, log) та надійний захист даних через криптографічні хеші. Використання Git у поєднанні з GitHub Actions дає змогу автоматизувати CI/CD-процеси: запускати юніт- і інтеграційні тести, лінтинг та деплой у єдиному конвеєрі, що суттєво підвищує якість і стабільність релізів.

Була обрана система Firebase Crashlytics для централізованого збирання та аналізу аварійних збоїв у додатку. Crashlytics – легковаговий інструмент від Google, який миттєво групує й пріоритизує краші, надаючи докладні стек-трейси та умови, що призвели до збою. Завдяки реальному часу оновлення дашборду можна оперативно виявляти найкритичніші проблеми: показник «crash-free users» відображає частку користувачів, що не зазнали збоїв за обраний період, що є ключовим параметром здоров'я додатку [20].

Для оцінки продуктивності використано Firebase Performance Monitoring, який автоматично збирає метрики часу запуску додатку, тривалості мережевих запитів, рендерингу екранів і кастомних трас. Отримані дані відображаються у вигляді графіків у консолі Firebase, що дозволяє швидко виявляти вузькі місця (наприклад, повільні HTTP-запити чи затримки при ініціалізації компонентів) та вимірювати вплив оптимізацій у реальному середовищі [21].

Комбіноване застосування Crashlytics і Performance Monitoring дає змогу:

– відстежувати стабільність випусків через метрику «crash-free users/sessions» і швидко реагувати на регресії;

– аналізувати час відкриття ключових екранів і вузькі місця в мережевих операціях;

– планувати пріоритети виправлень і покращень на основі об'єктивних даних з реального користувацького середовища.

Ці інструменти інтегруються безпосередньо в Android Studio через Firebase Assistant і не потребують додаткового серверного коду, що значно скорочує час налаштування моніторингу й дозволяє зосередитися на розвитку функціоналу додатку.

Для захисту й оптимізації кінцевого бінарника було застосовано R8 (який замінив ProGuard як стандартний інструмент у Android Gradle Plugin з версії 3.4) для одночасного виконання десагерінгу, обфускації, стискання коду (tree shaking) і dex-компіляції в один крок. R8 автоматично видаляє невикористані класи й методи, переписує код для підвищення продуктивності та

скорочує розмір APK, що призводить до швидшого старту додатку, менше ANR і кращого управління пам’яттю.

Для сумісності з існуючими налаштуваннями ProGuard зберігається підтримка файлу «proguard-rules.pro», у якому можна вказувати правила «-keep», щоб захистити від видалення або обфускації критичні елементи коду (наприклад, моделі даних та API-кілери). Дослідження показують, що застосування обфускації в Android-додатках зростає: відсоток застосування ProGuard/R8 у топових APK перевищив 70 % у 2023-2024 роках, що демонструє її ефективність у боротьбі з реверс-інжинірингом.

Таким чином, інтеграція R8/ProGuard у конвеєр збірки гарантує не лише захист інтелектуальної власності, але й підвищену продуктивність і зменшення розміру додатку без значних додаткових зусиль під час налаштування.

РОЗДІЛ 3
РОЗРОБКА МОБІЛЬНОГО ДОДАТКУ ДЛЯ ОПЕРАТИВНОЇ ПУБЛІКАЦІЇ ТА ЧИТАННЯ СТАТЕЙ

3.1 Практична реалізація об’єкта проектування

Першим кроком було встановлено останню версію Android Studio Arctic Fox із офіційного сайту Google, перевіривши при цьому наявність JDK 11+ та Android SDK 31. Після запуску IDE у налаштуваннях проекту активовано плагін Kotlin – це дозволило писати код на Kotlin без додаткових конфігурацій.

Далі у файлі app/build.gradle було вказано мінімальну підтримувану версію Android (minSdkVersion 24) та цільову платформу (targetSdkVersion 31), увімкнено ViewBinding і підключено плагін com.google.gms.google-services для інтеграції з Firebase (ліст. 3.1).

Лістинг 3.1 – Файлі app/build.gradle (початкова конфігурація)

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'com.google.gms.google-services'
    id 'kotlin-kapt'
}

android {
    namespace "com.morozione.roboblog"
    compileSdk 34

    defaultConfig {
        applicationId "com.morozione.roboblog"
        minSdk 24
        targetSdk 34
        versionCode 1
        versionName "1.0"
        multiDexEnabled true
    }

    buildTypes {
        release {
            minifyEnabled true
                                                    proguardFiles
getDefaultProguardFile('proguard-android-optimize.txt'),
'proguard-rules.pro'
```

```
        }
    }

    buildFeatures {
        viewBinding true
    }

    compileOptions {
        sourceCompatibility JavaVersion.VERSION_17
        targetCompatibility JavaVersion.VERSION_17
    }

    kotlinOptions {
        jvmTarget = "17"
    }
}

kapt {
    correctErrorTypes = true
}
```

кінець лістингу 3.1

Файл google-services.json, наданий з консолі Firebase, поміщено в корінь модуля app/, після чого Gradle автоматично імпортував усі налаштування для Firebase Auth і Realtime Database (ліст. 3.2).

Лістинг 3.2 – Файл google-services.json

```
{
  "project_info": {
    "project_number": "566758721581",
    "firebase_url": "https://roboblog-d8ccb.firebaseio.com",
    "project_id": "roboblog-d8ccb",
    "storage_bucket": "roboblog-d8ccb.appspot.com"
  },
  "client": [
    {
      "client_info": {
                                                    "mobilesdk_app_id":
"1:566758721581:android:8d0c5bddcec05a8e",
        "android_client_info": {
          "package_name": "com.morozione.roboblog"
        }
      },
      "oauth_client": [
        {
```

```
                                                                "client_id":
"566758721581-ltcfqruo0h7n4b9f7t64oojjefff5qfk.apps.googleusercont
ent.com",
          "client_type": 1,
          "android_info": {
            "package_name": "com.morozione.roboblog",
                                                        "certificate_hash":
"9a19f15592767c1eed3b32cc41da5d17941fc253"
          }
        },
        {
                                                                "client_id":
"566758721581-5r57o7efkihbidrr4h76p7vscgccfqqk.apps.googleusercont
ent.com",
          "client_type": 3
        }
      ],
      "api_key": [
        {
          "current_key": "AIzaSyCt2zqkLtXFt05qOFwqXUSb5UMdC41J4_I"
        }
      ],
      "services": {
        "analytics_service": {
          "status": 1
        },
        "appinvite_service": {
          "status": 2,
          "other_platform_oauth_client": [
            {
                                                                "client_id":
"566758721581-5r57o7efkihbidrr4h76p7vscgccfqqk.apps.googleusercont
ent.com",
              "client_type": 3
            }
          ]
        },
        "ads_service": {
          "status": 2
        }
      }
    }
  ],
  "configuration_version": "1"
}
```

кінець лістингу 3.2

Перший запуск на емуляторі Nexus 5X підтвердив правильність налаштувань – на екрані відобразився заголовок і кнопка входу, що свідчить про успішну підготовку середовища для подальшої розробки.

Далі буде детально розглянуто структуру проєкту (рис. 3.1): організацію папок і пакетів у модулі app, їх призначення та взаємозв'язки. Така архітектура дозволяє чітко розподілити відповідальність між елементами UI, бізнес-логікою та роботою з даними, полегшуючи навігацію в коді, масштабування функціоналу та підтримку розробленого рішення.

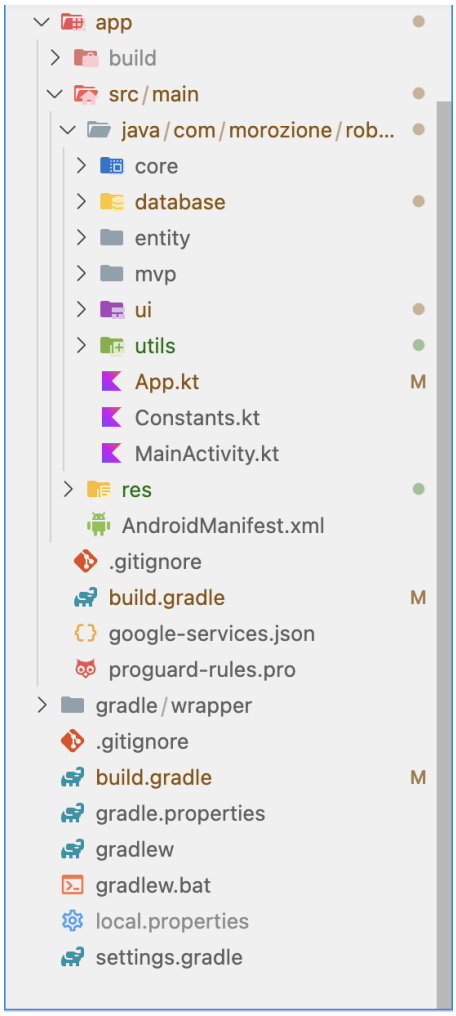

Рисунок 3.1 – Структура проєкту та організація папок

У корені модуля app під папкою src/main/java/com/morozione/roboblog реалізовано чітку модульну структуру, яка відповідає принципам чистої архітектури й полегшує навігацію в коді:

– ui/activity. Містить класи Activity – основні екрани додатку, такі як MainActivity (точка входу) та LoginActivity. Вони відповідають за управління життєвим циклом екрану і делегують бізнес-логіку презентерам;

– ui/fragment. У цій папці лежать Fragment-компоненти для окремих екранів (стрічка статей, детальний перегляд публікацій, створення/редагування посту тощо). Фрагменти забезпечують більш гнучке розташування UI-блоків і дозволяють повторно використовувати макети;

– ui/adapter. Тут зібрані класи для адаптерів RecyclerView, які відображають списки статей та інших колекційних даних. Кожен адаптер інкапсулює логіку «прив’язки» моделі до елементів списку й обробки кліків;

– ui/dialog. Містить діалоги (наприклад, підтвердження видалення або вибір зображення), що виносять повторювані UI-шаблони в окремі класи для зручності та консистентності;

– data. Відповідає за роботу з джерелами даних: тут знаходяться сервіси для звернення до Firebase Realtime Database, класи для локального кешу (через Room або SharedPreferences) і маппери між мережевими об’єктами та внутрішніми моделями;

– model. Сховище доменних моделей (User, Article, Profile), які описують структуру даних додатку. Ці класи не залежать від UI і можуть використовуватися у різних контекстах – від мережевих викликів до кешу;

– di. Пакет із компонентами та модулями для ін’єкції залежностей (Hilt або Dagger), де описано, як створюються екземпляри ViewModel, сервісів і репозиторіїв.

Додаткові каталоги, такі як util (утиліти для конвертації дат, перевірки мережі тощо) і common (загальні інтерфейси і константи), об’єднані в окремі пакети, щоб не засмічувати основну папку з бізнес-логікою. Така організація коду дозволяє швидко знайти відповідний клас, розділити відповідальність між компонентами і забезпечує простоту масштабування та підтримки проєкту.

У проєкті реалізовано власний клас-наступник Application – App.kt, який виступає центральною точкою ініціалізації всіх глобальних сервісів ще до запуску будь-якого Activity чи Fragment (ліст. 3.3).

Лістинг 3.3 – Власний клас-наступник Application – App.kt

```
package com.morozione.roboblog

import android.app.Application
import com.google.firebase.Firebase
import com.google.firebase.database.FirebaseDatabase
import com.google.firebase.database.database
import com.morozione.roboblog.core.OffirentPresenterStorage

class App : Application() {

    companion object {
        val presenterStorage = OffirentPresenterStorage()
    }

    override fun onCreate() {
        super.onCreate()
        // Initialize Firebase using KTX syntax
        Firebase.database.setPersistenceEnabled(true)
    }
}
```

кінець лістингу 3.3

У цьому фрагменті продемонстровано клас App, що наслідує від стандартного Application і виступає єдиною точкою входу в нашу Android-аплікацію. Всередині companion object оголошено єдиний екземпляр presenterStorage, який створюється за допомогою кастомного класу OffirentPresenterStorage; це дозволяє зберігати стан усіх Presenter-ів у єдиній глобальній області видимості без прив’язки до конкретного екрана чи життєвого циклу Activity.

Метод onCreate() викликається відразу після запуску процесу додатка, тож саме тут ми ініціалізуємо ключові сервіси. Використовуючи розширення KTX – звернення до Firebase.database – ми вмикаємо опцію setPersistenceEnabled(true), що активує локальне збереження даних Realtime Database на пристрої. Це

означає, що будь-які записи, які надсилаються чи отримуються через Firebase, автоматично кешуються в локальній базі, а після втрати з’єднання користувач продовжить бачити останню відомості з бази. Як тільки мережа відновлюється, всі зміни синхронізуються «за кадром», не викликаючи перезавантаження UI.

Таким чином, App.kt забезпечує центральну ініціалізацію корпоративних сервісів: з одного боку – гарантоване збереження Presenter-ів у нашому кастомному сховищі, а з іншого – стабільну й відмовостійку роботу Firebase Realtime Database, що суттєво підвищує комфорт користувача під час перегляду та публікації статей.

Перед запуском будь-якої частини додатку Android спочатку читає файл AndroidManifest.xml, де вказано головний клас Application, перелік Activity і необхідні дозволи (ліст. 3.4). У цьому проєкті атрибут android:name=".App" у блоці <application> повідомляє системі Android, що для глобальної ініціалізації слід використовувати кастомний клас App. Далі у списку Activity представлені три екрани: LoginActivity із LAUNCHER-інтенцією, яка стартує додаток, головний MainActivity, що відповідає за відображення стрічки публікацій, та BlogDetailsActivity для перегляду окремого запису. Наприкінці файлу зазначено перелік дозволів – від доступу до Інтернету до камерних та файлових операцій – які потрібні для коректної роботи функцій створення та перегляду багатомедійного контенту.

Лістинг 3.4 – Файл AndroidManifest.xm

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.morozione.roboblog">

    <application
        android:name=".App"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity" />
```

```
          <activity android:name=".ui.activity.BlogDetailsActivity"
/>
        <activity
            android:name=".ui.activity.LoginActivity"
            android:exported="true">
            <intent-filter>
                  <action android:name="android.intent.action.MAIN"
/>

                                                                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-feature
        android:name="android.hardware.camera"
        android:required="false" />

                                                    <uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"
        android:maxSdkVersion="32" />
    <uses-permission android:name="android.permission.CAMERA" />
                                                    <uses-permission
android:name="android.permission.POST_NOTIFICATIONS" />

    <uses-permission android:name="android.permission.INTERNET" />

</manifest>
```

кінець лістингу 3.4

У лістингу 3.4 manifest’у перший атрибут android:name=".App" у тезі <application> гарантує, що перед створенням будь-якого Activity чи Fragment викликається метод onCreate() нашого класу App, де відбувається ініціалізація Firebase та налаштування Presenter-Storage. Додаткові атрибути android:icon і android:label задають піктограму та назву додатку, а supportsRtl і theme визначають підтримку RTL-мов та глобальну тему. Секція <activity> для LoginActivity містить інтенційний фільтр із категоріями MAIN і LAUNCHER, що робить цей екран початковим при запуску програми. Інші Activity без інтенційних фільтрів можуть бути викликані лише з коду додатку. Наприкінці файлу розташовано блок <uses-permission>, який запитує у користувача доступ до камери, зовнішнього сховища, push-повідомлень і мережі – усе це необхідно для завантаження, зберігання й відображення зображень у статтях, а також для

синхронізації даних із Firebase. Така конфігурація manifest'у забезпечує правильний старт і роботу всіх компонентів мобільного додатку.

Після централізованої ініціалізації Firebase і налаштування Presenter Storage у класі App можна переходити до розробки механізму автентифікації користувачів. Доцільно описати як реалізовано екран входу (LoginActivity) з його View-Presenter взаємодією через Moxy, а також як Presenter викликає методи FirebaseAuth і обгортає їх у RxJava-потоки.

У модулі авторизації реалізовано активність LoginActivity, яка наслідує від MvpAppCompatActivity і реалізує інтерфейс LoginView. На екрані розміщено поля для вводу email і пароля, а також дві кнопки – «Login» та «Registration» (рис. 3.2).

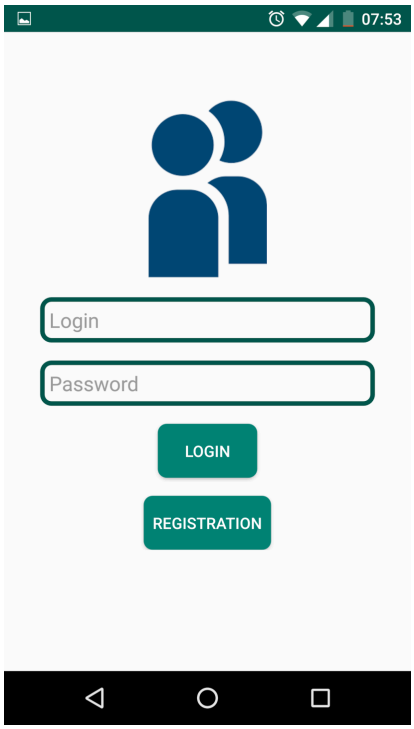

Рисунок 3.2 – Екран входу

Клас LoginActivity (додаток А) відповідає за весь процес аутентифікації та реєстрації користувача в додатку і побудований за патерном MVP із використанням Moxy. Він наслідується від MvpAppCompatActivity та реалізує

інтерфейс LoginView, що визначає колбеки для результатів авторизації та реєстрації.

При запуску активності в методі onCreate() відбувається двоетапна підготовка: спочатку викликається setContentView() із розміткою екрану, потім – setListeners(), яка прив’язує обробники натискань до кнопок «Login» і «Registration». Після цього метод checkOnLoginning() перевіряє через presenter.currentUser, чи вже є залогінений користувач; у разі позитивної відповіді відразу стартує MainActivity, а LoginActivity завершує своє життя, щоб не показувати екран логіну повторно.

Для отримання посилань на UI-елементи застосовано власне розширення bind<T>(R.id.xyz), що приховує boilerplate з findViewById. Це дає змогу оголосити змінні mLogin, bRegistration, mEmail і mPassword у вигляді властивостей класу, а їх ініціалізація відбувається «на вимогу» при першому зверненні.

Методи singIn() і singUp() виконують елементарну валідацію – переконуються, що поля не порожні, і потім делегують виклик у Presenter (presenter.singIn(login, password) або presenter.singUp(login, password)). Якщо ж користувач не ввів обидва значення, показується Toast із проханням заповнити дані.

Presenter, у свою чергу, через RxJava-або прогресивні Task-методи звертається до FirebaseAuth і викликає зворотні методи onAuthorizationResult() та onRegistrationResult(). Саме ці методи реалізовано в LoginActivity: при успішному вході (isSuccess = true) відразу запускається MainActivity, а при успішній реєстрації викликається presenter.saveUser(getFilledUser()), який записує новий об’єкт User у Realtime Database.

Після збереження профілю користувача Presenter викликає onSavedUserSuccess(), що знову переводить додаток у головний екран і завершить LoginActivity. У разі будь-якої помилки (onError()), наприклад через неправильний пароль або проблеми з мережею, користувач отримує просте повідомлення «Error» у вигляді Toast.

Метод getFilledUser() створює новий об’єкт User, заповнює лише поле email (решту атрибутів можна доповнити пізніше) і повертає його в Presenter. Така організація дозволяє чітко розділити відповідальність: Activity – за роботу з UI і навігацію, Presenter – за комунікацію з бекендом і бізнес-логіку, а View – за абстрактні зворотні виклики, які Activity перетворює на конкретні дії. Це гарантує, що код залишається модульним, легким для тестування та підтримки.

На скріншоті з консолі Firebase (рис. 3.3) відображено розділ Authentication → Users, де можна бачити список зареєстрованих облікових записів. Інтерфейс надає пошукове поле для фільтрації користувачів за email, номером телефону або UID, а також показує провайдери входу (в даному випадку – email), дату створення облікового запису й останній вхід. Цей інструмент дозволяє адміністраторам вручну додавати, видаляти або блокувати користувачів, перевіряти статус їх аутентифікації та отримувати детальні діагностичні дані про активність, що корисно для відлагодження і підтримки безпеки додатку.

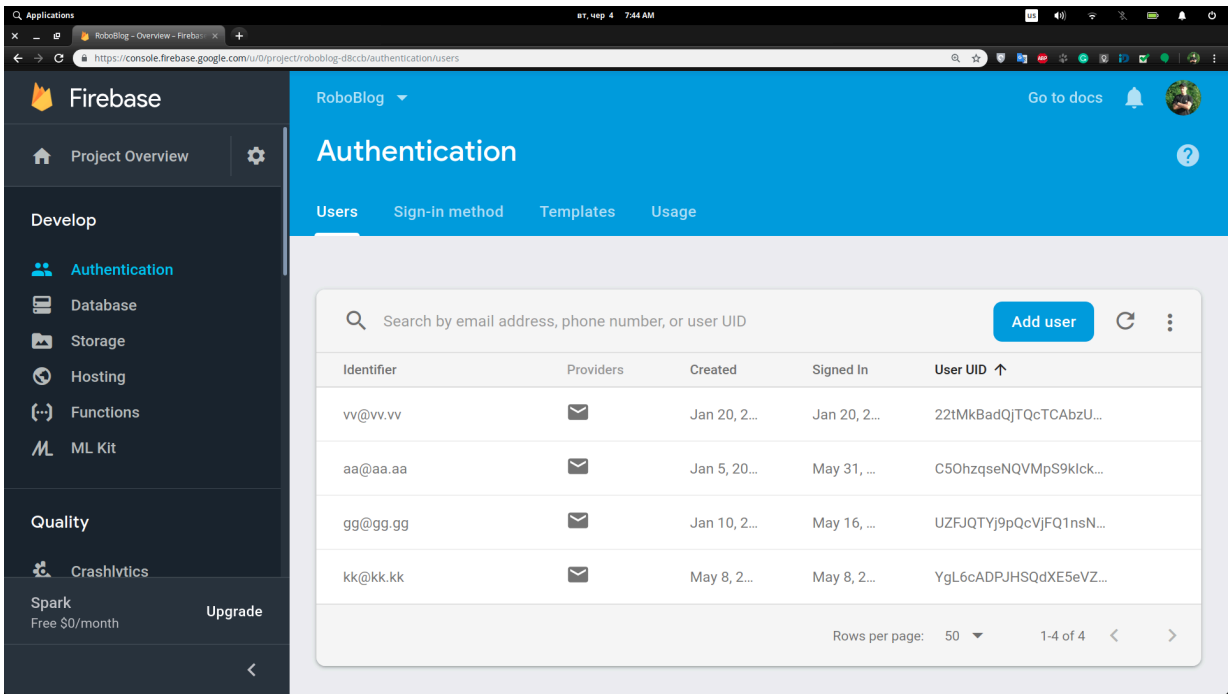

Рисунок 3.3 – Сторінка управління користувачами

Після успішної авторизації користувач потрапляє на головний екран із стрічкою публікацій, реалізовану у вигляді фрагмента FeedFragment. Цей фрагмент відповідає за динамічний показ списку статей, підвантажуючи їх з Firebase Realtime Database та оновлюючи інтерфейс одразу після появи нових даних (рис. 3.4).

Рисунок 3.4 – Список всіх статей

Клас BlogsFragment (додаток Б) виступає універсальним базовим фрагментом для відображення списку публікацій. Він наслідується від MvpAppCompatFragment, що дозволяє йому працювати за патерном MVP із Moxy, і використовує ViewBinding для зв’язку з макетом fragment_blogs.xml. У конструкторі фрагмента ініціалізується екземпляр BlogsAdapter, який залежно від типу блогу (BlogType.GLOBAL або інших) виводить відповідний набір даних.

У методі initView() встановлюється RecyclerView із лінійним лейаутом і підключається адаптер, а в setListener() налаштовується обробник свайпу для оновлення стрічки та клік-лісенер, що відкриває екран деталей публікації через

BlogDetailsActivity. Методи setBlogs(), showProgress(), hideProgress() і showError() забезпечують керування станом UI, показуючи індикатор завантаження, оновлюючи дані адаптера або виводячи повідомлення про помилку. Така організація коду гарантує чітке розділення бізнес-логіки (Presenter) від представлення (View), спрощує повторне використання компонента для різних типів стрічок та полегшує тестування і підтримку

Після перегляду наявних публікацій у стрічці користувачеві надається можливість не лише читати, а й створювати або змінювати власні записи. Для цього передбачено окремий екран редагування – фрагмент CreateBlogFragment, де поля заголовка та опису автоматично заповнюються поточними значеннями, а за допомогою кнопки «Зберегти» зміни відправляються на сервер, оновлюючи існуючу статтю або додаючи нову (рис. 3.5).

Рисунок 3.5 – Екран створення статті

Клас CreateBlogFragment (додаток В) реалізує екран створення та редагування публікацій у стилі MVP із Moxy і служить мостом між UI та бізнес-логікою. Він наслідується від BaseImageFragment, що додає можливість

вибору й обробки зображень, та реалізує інтерфейс CreateBlogView, через який Presenter сповіщає про результати операцій. При створенні фрагмента в методі onCreate активується меню з пунктами збереження та додавання фото, а в onCreateView за допомогою ViewBinding створюється прив’язка до XML-макета FragmentCreateBlogBinding. Дії меню обробляються в onOptionsItemSelected: вибір «Зберегти» запускає метод creteBlog(), який формує або оновлює модель Blog і передає її в Presenter, а «Додати фото» викликає діалог камери чи галереї.

Коли Presenter успішно створює або оновлює публікацію, відповідні колбеки onBlogCreated() та onBlogUpdated() показують користувачеві підтвердження через Snackbar і очищають поля введення, готуючи фрагмент до наступного використання. У разі помилки onError() виводить повідомлення про невдачу. Обробник imageMade() оновлює прев’ю фото, перекодовуючи URI у Bitmap для економного завантаження. Завдяки такій організації коду CreateBlogFragment чітко розмежовує обов’язки: UI-елементи змінюються тут, а всі операції з мережею та збереженням моделей делегуються Presenter-ові, що забезпечує чисту, тестовану й гнучку архітектуру мобільного додатку.

Після того як користувач заповнює заголовок та текст, він може додати ілюстрацію до своєї публікації. Для цього в CreateBlogFragment використовується Glide: у момент, коли користувач вибирає або знімає фото, URI передається в Presenter, а потім у View (через метод imageMade), де ми викликаємо GlideApp.with(context).load(uri)…into(binding.mIcon). Цей простий рядок коду автоматично завантажує зображення у фоновому потоці, кешує його в пам’яті та на диску, а при повторному відображенні підтягує вже закешовану копію. Завдяки Glide вдається уникнути «мигання» інтерфейсу, скоротити кількість мережевих запитів і забезпечити плавну роботу списку статей, коли публікації містять великі зображення. Таким чином, обробка мультимедійного контенту відбувається цілком прозоро для користувача, поєднуючи зручний UI та високий рівень продуктивності.

Усі екрани додатку оформлені відповідно до принципів Material Design із використанням офіційних компонентів із бібліотеки MDC Android. У нижній

частині екрану розташовано BottomNavigationView, що дозволяє швидко перемикатися між головною стрічкою публікацій, власними постами та профілем користувача без перевантаження інтерфейсу.

Кожного разу, коли користувач виконує дію (успішно зберігає публікацію, стикається з помилкою чи потребує підтвердження), у нижній частині екрану з’являється Snackbar. Він надає стислу зворотну інформацію та може містити кнопку дії (наприклад, «Спробувати ще раз» при невдалій операції), що дозволяє швидко виправити помилку. Такий підхід гарантує однорідність стилю, передбачувані поведінку та анімації, а також допомагає користувачам легко орієнтуватися в додатку та виконувати основні сценарії без зайвих дій.

У ході практичної реалізації ключовими рішеннями стали обрані Kotlin для написання компактного та читабельного коду, Moxy + RxJava для чіткого розділення UI і бізнес-логіки з реактивною обробкою асинхронних подій, а також Firebase Realtime Database із вбудованим кешем для миттєвої синхронізації та offline-підтримки без додаткових серверних компонентів. Використання Glide забезпечило зручне завантаження й кешування ілюстрацій, а Material Components (BottomNavigationView, FAB, Snackbar) надали інтерфейсу сучасний і передбачуваний вигляд. Завдяки такій комбінації стеку розробка пройшла значно швидше, уникнувши «болючих» місць із налаштуванням серверної інфраструктури й управлінням життєвим циклом Presenter-ів, а користувачі одразу отримали плавний та стабільний досвід навіть за нестабільного інтернету. Для наступного релізу залишаються відкритими завдання з додавання персонального кабінету, push-сповіщень і аналітики, а також оптимізації сховища зображень та розширених можливостей фільтрації контенту.

3.2 Тестування та налагодження інформаційно-комп’ютерної системи

Умова стабільності та коректності роботи мобільного додатку досягається системним підходом до тестування та налагодження на всіх рівнях архітектури.

Спочатку проводиться статичний аналіз коду за допомогою вбудованих у Android Studio інструментів (Lint, Detekt) та Kotlin Compiler Checks – це дозволяє виявити потенційні помилки форматування, невикористані змінні, ризик null-посилань тощо ще до компіляції.

Для перевірки бізнес-логіки застосовуються модульні тести з використанням JUnit і Mockito. Presenter-и та утиліти тестуються із замоканими залежностями на FirebaseAuth і DatabaseReference, що дає змогу імітувати успішні й помилкові відповіді сервера без мережі. Модульні тести охоплюють сценарії входу, реєстрації, створення й оновлення публікацій, валідацію введення даних та обробку виключень.

Нижче наведено приклад модульного тесту для LoginPresenter, який демонструє два базові сценарії: успішний вхід і обробку помилки при неправильних облікових даних (ліст. 3.5).

У тесті за допомогою Mockito створюється фейковий екземпляр FirebaseAuth, на який підмінюється повернення успішного або невдалого Task<AuthResult>. Після виклику presenter.login(email, password) перевіряється, що у випадку успіху викликається view.showLoginSuccess(), а при помилці – view.showError(message).

Лістинг 3.5 – Модульний тест для LoginPresenter

```
class LoginPresenterTest {

    private lateinit var presenter: LoginPresenter
    private val auth = mock(FirebaseAuth::class.java)
    private val view = mock(LoginView::class.java)

    @Before
    fun setUp() {
        // Підмінюємо повернення FirebaseAuth.getInstance()
        FirebaseAuth.setInstance(auth)
        presenter = LoginPresenter()
        presenter.attachView(view)
    }

    @Test
    fun `login success invokes showLoginSuccess`() {
        // Підготовка успішного результату
```

```
        val task = Tasks.forResult(mock(AuthResult::class.java))
         `when`(auth.signInWithEmailAndPassword("user@example.com",
"password"))
            .thenReturn(task)

        presenter.login("user@example.com", "password")

        verify(view).showLoginSuccess()
        verify(view, never()).showError(any())
    }

    @Test
    fun `login failure invokes showError`() {
        // Підготовка помилки аутентифікації
           val exception = FirebaseAuthException("ERROR", "Invalid
credentials")
        val task = Tasks.forException<AuthResult>(exception)
         `when`(auth.signInWithEmailAndPassword("user@example.com",
"wrong"))
            .thenReturn(task)

        presenter.login("user@example.com", "wrong")

        verify(view).showError("Invalid credentials")
        verify(view, never()).showLoginSuccess()
    }
}
```

кінець лістингу 3.5

Нижче наведено приклад модульного тесту для CreateBlogPresenter, що демонструє два базові сценарії: успішне створення статті та обробку помилки при записі в базу (ліст. 3.6).

За допомогою Mockito підмінюється DatabaseReference, а результат операції setValue() моделюється через Tasks.forResult() або Tasks.forException().

Лістинг 3.6 – Модульний тест для CreateBlogPresenterTest

```
class CreateBlogPresenterTest {

    private lateinit var presenter: CreateBlogPresenter
    private val dbRef = mock(DatabaseReference::class.java)
    private val childRef = mock(DatabaseReference::class.java)
    private val view = mock(CreateBlogView::class.java)

    @Before
    fun setUp() {
        // Підмінюємо кореневий DatabaseReference на наш мок
```

```
        Firebase.databaseRef = dbRef
        presenter = CreateBlogPresenter()
        presenter.attachView(view)

        // Загальна настройка: push() → childRef
        whenever(dbRef.child("blogs")).thenReturn(childRef)
        whenever(childRef.push()).thenReturn(childRef)
    }

    @Test
    fun `createBlog success invokes onBlogCreated`() {
        // Моделюємо успішний запис у базу

whenever(childRef.setValue(any<Blog>())).thenReturn(Tasks.forResul
t(null))

        presenter.createBlog(Blog().apply {
            id = "123"; title = "Test"; descrption = "Desc"
        }, imageUri = null, activity = mock(Activity::class.java))

        verify(view).onBlogCreated()
        verify(view, never()).onError()
    }

    @Test
    fun `createBlog failure invokes onError`() {
        // Моделюємо помилку запису
        val exception = DatabaseException("Write failed")

whenever(childRef.setValue(any<Blog>())).thenReturn(Tasks.forExcep
tion(exception))

          presenter.createBlog(Blog(), imageUri = null, activity =
mock(Activity::class.java))

        verify(view).onError()
        verify(view, never()).onBlogCreated()
    }
}
```

кінець лістингу 3.6

У цих тестах:

– перш ніж викликати presenter.createBlog(), кореневий DatabaseReference підмінюється на мок-об’єкт, і для шляху "blogs" та методу push() налаштовано повернення того ж мок-референсу;

– у тесті успіху метод setValue() повертає Tasks.forResult(null), що призводить до виклику view.onBlogCreated();

– у тесті невдачі setValue() повертає Tasks.forException(...), тому Presenter викликає view.onError(), і жодного успішного колбека не відбувається.

Ці приклади демонструють, як можна ізольовано перевіряти Presenter, не залучаючи реальний Firebase, та гарантувати коректну обробку обох сценаріїв.

Інтеграційні тести в середовищі Robolectric виконують перевірку взаємодії UI-шару з ViewBinding і PresenterStore Moxy у JVM, гарантуючи, що фрагменти коректно створюють макети, реактивні підписки на потоки даних ініціюються та стан Presenter-а відновлюється після зміни конфігурації екрана.

Для підтвердження правильності роботи інтерфейсу та навігації застосовуються end-to-end UI-тести з Espresso. Автоматизовані сценарії проганяються як локально, так і в хмарній лабораторії Firebase Test Lab на різних емуляторах і пристроях. Перевіряються ключові шляхи користувача: вхід у систему, відображення стрічки, створення та редагування статей, а також відмова через відсутність мережі.

Моніторинг продуктивності та виявлення вузьких місць здійснюється за допомогою Android Profiler і Firebase Performance Monitoring. Фіксуються метрики часу запуску, latency мережевих запитів, використання пам’яті й CPU під час масивних оновлень стрічки. Аварійні збої агрегуються у Firebase Crashlytics із докладними стек-трейсами та контекстом, що дозволяє оперативно локалізувати та виправляти дефекти.

Для роботи програми визначено мінімальні системні вимоги: Android 6.0 (API 23) або вище, 2 GB RAM та доступ до Інтернету (HTTP/HTTPS). Рекомендовані параметри – Android 9.0+ із 4 GB RAM, сучасний процесор і стабільне 3G/4G/5G-з’єднання для оптимальної продуктивності та швидкої синхронізації даних у реальному часі.

3.3 Розробка бази даних

Реалізовано фізичну модель бази даних (рис. 3.6) відповідно до раніше побудованих UML- і ER-діаграм. У представленій ER-схемі визначено дві

головні сутності – users і blogs, які відображають дві ключові колекції в даній NoSQL-моделі (Firebase Realtime Database).

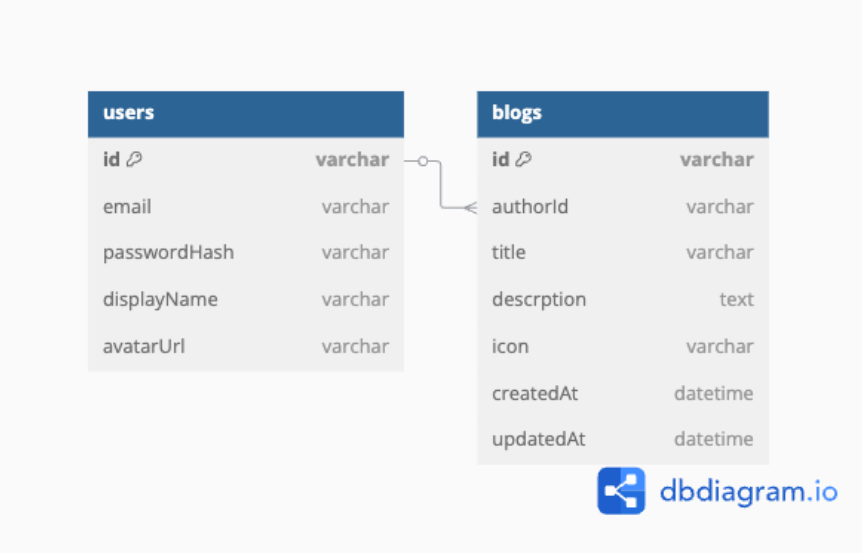

Рисунок 3.6 – Схема БД

Таблиця users:

– id (varchar, PK) – унікальний ідентифікатор користувача, що відповідає UID із Firebase Auth. Використовується як первинний ключ у всіх зв’язках із публікаціями;

– email (varchar) – адреса електронної пошти для входу та комунікації;

– passwordHash (varchar) – хеш пароля, якщо ви реалізуєте власну схему збереження паролів (в разі використання тільки Firebase Auth це поле може бути неактивним);

– displayName (varchar) – видиме ім’я користувача, яке відображається поруч із його публікаціями;

– avatarUrl (varchar) – URL зображення профілю, що використовується для персоналізації.

Таблиця blogs:

– id (varchar, PK) – ключ, який генерується методом push() в Realtime Database. Унікально ідентифікує кожну публікацію;

– authorId (varchar, FK → users.id) – зовнішній ключ на автора статті, що задає зв’язок «багато до одного»: один користувач може мати багато статей;

– title (varchar) – заголовок публікації, короткий опис теми;

– descrption (text) – детальний вміст статті;

– icon (varchar) – URL зображення, прикріпленого до публікації;

– createdAt (datetime) – мітка часу створення публікації;

– updatedAt (datetime) – мітка часу останнього редагування.

Зв’язок між таблицями реалізовано через поле authorId, яке вказує на users.id. Це означає, що кожна публікація blogs належить одному автору із users, а в користувача може бути нуль чи більше пов’язаних статей. Така схема дозволяє ефективно вибирати всі пости певного автора, а також одразу отримувати інформацію про автора при читанні статті.

3.4 Розробка інсталяційного пакета

Розробка інсталяційного пакета є завершальним етапом життєвого циклу мобільного додатку – саме цей артефакт потрапляє в руки кінцевого користувача та визначає, наскільки просто його встановити, оновити й експлуатувати. У контексті Android-системи процес формування інсталяційного пакета базується на декількох взаємопов’язаних концепціях: розділенні конфігурацій на debug- і release-збірки, цифровому підписі, оптимізації коду (мініфікації та обфускації), а також створенні Android App Bundle для публікації в Google Play.

По-перше, debug-збірка призначена для налагодження: вона містить розширені логи, має увімкнене відлагодження й не обфускує код, що полегшує виявлення помилок у ранніх стадіях розробки. Release-збірка навпаки максимально оптимізована – код стискається й обфускується (ProGuard/R8), щоб зменшити розмір APK і ускладнити реверс-інжиніринг; також вона підписується сертифікатом розробника, що гарантує цілісність і авторство пакета.

Діджитал-підпис є обов’язковою умовою встановлення будь-якого Android-додатку: він формує криптографічний зв’язок між вмістом пакета та його власником, дозволяючи системі безпеки перевірити, чи не було змінено код після збірки. Правильне керування ключами підпису – одна з критичних процедур у DevOps-конвеєрі, оскільки компрометація keystore відкриває шлях до розповсюдження шкідливого ПЗ під виглядом легітимного оновлення.

Ще один сучасний тренд – перехід від простого APK до Android App Bundle (AAB), який дозволяє магазинному серверу динамічно збирати оптимізовані APK для конкретного пристрою (за архітектурою, локалізацією, ресурсами). Це знижує об’єм завантажуваного пакета – користувач отримує тільки ті ресурси, що йому дійсно потрібні.

Нарешті, питанням розробки пакета слідує налаштування безперервної інтеграції: автоматизовані збірки в CI/CD середовищі забезпечують регулярне формування підписаних release-артефактів після проходження всіх тестів, а системи моніторингу версій гарантують коректність чисел versionCode і versionName. Завдяки цьому вихідний процес поєднує в собі належну якість, безпеку й зручність доставки оновлень до кінцевих користувачів.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи були послідовно реалізовані всі етапи розробки мобільного застосунку для публікації та читання статей: від детального аналізу предметної області й формулювання вимог до побудови архітектури, вибору технологічного стеку й практичної імплементації ключових модулів до комплексного тестування й підготовки релізного інсталяційного пакета. Кожне з поставлених завдань було вирішене із застосуванням сучасних інструментів та методів, що забезпечило створення стабільного, безпечного і зручного в користуванні рішення.

Проведено ґрунтовний аналіз предметної області мобільних додатків для публікації та читання статей: вивчено популярні нативні й гібридні рішення, виявлено їхні сильні й слабкі сторони, а також обґрунтовано необхідність створення спеціалізованого Android-додатку з підтримкою реального часу та offline-режиму.

Визначено та задокументовано функціональні вимоги (реєстрація, CRUD-операції зі статтями, пошук, offline-доступ тощо) і нефункціональні вимоги (продуктивність, безпека, сумісність, юзабіліті), а також на їхній основі виконано модельне проєктування: побудовано UML-діаграми прецедентів, класів, послідовностей, активностей, компонентів і розгортання.

Спроєктовано архітектуру системи з урахуванням патерну MVP (Moxy) і реактивного шару RxJava, розроблено компоненти UI, бізнес-логіки та Data Layer, а також створено ER-схему для Firebase Realtime Database у вигляді DBML для візуалізації на dbdiagram.io.

Обґрунтовано вибір технологічного стеку: Kotlin як основна мова, Android Studio + Gradle для збірки, Firebase Auth і Realtime Database для безпечної та миттєвої синхронізації, Glide для роботи з зображеннями, Material Components для UI, Git/CICD, Crashlytics і R8/ProGuard для захисту й оптимізації.

Реалізовано ключові функціональні модулі: екран логіну та реєстрації через FirebaseAuth і Moxy+RxJava; стрічку статей із RecyclerView/Adapter та

реактивною підпискою на Realtime Database; екрани створення й редагування публікацій із зображеннями; offline-підтримку через локальний кеш Firebase; а також UI-компоненти з Material Design (BottomNavigationView, FAB, Snackbar).

Проведено комплексне тестування й налагодження: статичний аналіз коду (Lint, Detekt), модульні тести Presenter-ів із JUnit/Mockito, інтеграційні тести Robolectric, UI-тести Espresso в Firebase Test Lab, моніторинг продуктивності через Android Profiler і Firebase Performance та збір аварійних звітів за допомогою Firebase Crashlytics.

Підготовлено інсталяційний пакет: налаштовано debug- і release-збірки з окремими signingConfig та оптимізацією R8/ProGuard, сформовано APK і AAB через Gradle, перевірено цифровий підпис і цілісність пакета, а також інтегровано автоматичну збірку та тестування в CI/CD-конвеєрі.

Отже, розроблений Android-додаток відповідає всім вимогам завдання й демонструє високу якість реалізації: архітектура на базі Kotlin, Moxy+RxJava і Firebase забезпечує відмовостійкість та гнучкість, а Material Design і Glide гарантують інтуїтивний інтерфейс і оптимальне відображення мультимедіа. Підсумком роботи стало створення готового до публікації пакета, що може бути легко масштабованим та подальше обслуговуватися, що підтверджує практичну цінність отриманого рішення і дає змогу рекомендувати його для використання в реальних умовах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ukrainians begin using Internet more, with 80 % online every day, social survey finds. UNDP. URL: https://www.undp.org/ukraine/press-releases/ukrainians-begin-using-internet-more-80-online-every-day-social-survey-finds (дата звернення: 04.03.2025).

2. Mobile Screen Resolution Stats Ukraine. Statcounter Global Stats. URL: https://gs.statcounter.com/screen-resolution-stats/mobile/ukraine/2025 (дата звернення: 04.03.2025).

3. Mobile News Apps Market. Market.us. URL: https://market.us/report/mobile-news-apps-market/ (дата звернення: 05.03.2025).

4. Mobile News Apps Market Size Forecasted To Achieve 22.29 Billion By 2029 With Steady Growth. URL: https://www.openpr.com/news/4071205/mobile-news-apps-market-size-forecasted-to-achieve-22-29-billion (дата звернення: 06.03.2025).

5. 2024's Essential Mobile App Statistics. Trends, Growth, and User Insights. Software Development Company. Netguru. URL: https://www.netguru.com/blog/mobile-app-statistics (дата звернення: 09.03.2025).

6. Kotlin with Firebase. Building a Simple Database App. URL: https://medium.com/@juricavoda/kotlin-with-firebase-building-a-simple-database-app-d29aa7ebdc52 (дата звернення: 13.03.2025).

7. A Review on Firebase (Backend as A Service) for Mobile Application Development. ResearchGate. URL: https://www.researchgate.net/publication/358244355_A_Review_on_Firebase_Backend_as_A_Service_for_Mobile_Application_Development (дата звернення: 14.03.2025).

8. Implementation of Firebase in the Development of Android-based Queue Reservation and Treatment Record Applications. ResearchGate. URL: https://www.researchgate.net/publication/390094294_IMPLEMENTATION_OF_FIREBASE_IN_THE_DEVELOPMENT_OF_ANDROID-BASED_QUEUE_RESERVA

TION_AND_TREATMENT_RECORD_APPLICATIONS (дата звернення: 15.03.2025).

9. Firebase Realtime Database and Firestore. URL: https://medium.com/@ramadan123sayed/firebase-firebase-realtime-database-and-firestore-8b4fcddb1059 (дата звернення: 16.03.2025).

10. Building Real-Time Apps with Firebase Realtime Database and Kotlin. URL: https://blog.stackademic.com/building-real-time-apps-with-firebase-realtime-database-and-kotlin-fd368a396e96 (дата звернення: 18.03.2025).

11. Kotlin for Android. Kotlin Help. URL: https://kotlinlang.org/docs/android-overview.html (дата звернення: 02.04.2025).

12. Kotlin vs Java. Key Differences & Best Choice for Developers. Software Development Outsourcing Services. Hire Offshore Developers. URL: https://www.clariontech.com/blog/kotlin-vs-java (дата звернення: 05.04.2025).

13. Top Programming Languages in 2025. A Data-Driven Ranking Based on Industry Demand. FutureShield. AI, Security & Automation. URL: https://futureshield.substack.com/p/top-programming-languages-in-2025 (дата звернення: 09.04.2025).

14. Contributors to Wikimedia projects. Android Studio. Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Android_Studio (дата звернення: 11.04.2025).

15. Choose a Database. Cloud Firestore or Realtime Database. Firebase Realtime Database. URL: https://firebase.google.com/docs/database/rtdb-vs-firestore (дата звернення: 14.04.2025).

16. Firebase Authentication. Firebase. URL: https://firebase.google.com/docs/auth (дата звернення: 17.04.2025).

17. Material Design. Version 2. URL: https://m2.material.io/design/introduction (дата звернення: 18.04.2025).

18. A Computational Method for Evaluating UI Patterns. URL: https://arxiv.org/abs/1807.04191 (дата звернення: 20.04.2025).

19. Contributors to Wikimedia projects. Git – Wikipedia. Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Git (дата звернення: 21.04.2025).

20. Firebase Crashlytics. Firebase. URL: https://firebase.google.com/docs/crashlytics (дата звернення: 23.04.2025).

21. Firebase Performance Monitoring. Firebase. URL: https://firebase.google.com/docs/perf-mon (дата звернення: 24.04.2025).

ДОДАТКИ

Додаток А

Клас LoginActivity

```
package com.morozione.roboblog.ui.activity

import android.content.Intent
import android.os.Bundle
import android.widget.Button
import android.widget.EditText
import android.widget.Toast
import moxy.MvpAppCompatActivity
import moxy.presenter.InjectPresenter
import com.morozione.roboblog.MainActivity
import com.morozione.roboblog.R
import com.morozione.roboblog.entity.User
import com.morozione.roboblog.mvp.presenter.LoginPresenter
import com.morozione.roboblog.mvp.view.LoginView
import com.morozione.roboblog.utils.bind

class LoginActivity : MvpAppCompatActivity(), LoginView {

    private val mLogin by bind<Button>(R.id.login)
    private val bRegistration by bind<Button>(R.id.registration)
    private val mEmail by bind<EditText>(R.id.email)
    private val mPassword by bind<EditText>(R.id.password)

    @InjectPresenter
    lateinit var presenter: LoginPresenter

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_login)

        setListeners()
        checkOnLoginning()
    }

    private fun checkOnLoginning() {
        if (presenter.currentUser != null) {
            startActivity(Intent(this, MainActivity::class.java))
            finish()
        }
    }

    private fun setListeners() {
        mLogin.setOnClickListener {
            singIn()
        }
        bRegistration.setOnClickListener {
            singUp()
        }
    }
```

```
    private fun singIn() {
        val login = mEmail.text.toString()
        val password = mPassword.text.toString()
        if (!login.isEmpty() && !password.isEmpty()) {
            presenter.singIn(login, password)
        } else {
                    Toast.makeText(this, "Input available data",
Toast.LENGTH_SHORT).show()
        }
    }

    private fun singUp() {
        val login = mEmail.text.toString()
        val password = mPassword.text.toString()
        if (!login.isEmpty() && !password.isEmpty()) {
            presenter.singUp(login, password)
        } else {
                    Toast.makeText(this, "Input available data",
Toast.LENGTH_SHORT).show()
        }
    }

    override fun onAuthorizationResult(isSuccess: Boolean) {
        if (isSuccess) {
            startActivity(Intent(this, MainActivity::class.java))
            finish()
        }
    }

    override fun onRegistrationResult(isSuccess: Boolean) {
        if (isSuccess) {
            presenter.saveUser(getFilledUser())
        }
    }

    override fun onSavedUserSuccess(isSuccess: Boolean) {
        if (isSuccess) {
            startActivity(Intent(this, MainActivity::class.java))
            finish()
        }
    }

    override fun onError() {
        Toast.makeText(this, "Error", Toast.LENGTH_SHORT).show()
    }

    private fun getFilledUser(): User {
        val user = User()
        user.email = mEmail.text.toString()

        return user
    }
}
```

Додаток Б

Клас BlogsFragment

```
package com.morozione.roboblog.ui.fragment

import android.content.Intent
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.recyclerview.widget.LinearLayoutManager
import androidx.recyclerview.widget.RecyclerView
import com.google.android.material.snackbar.Snackbar
import com.morozione.roboblog.R
import com.morozione.roboblog.core.BlogType
import com.morozione.roboblog.databinding.FragmentBlogsBinding
import com.morozione.roboblog.entity.Blog
import com.morozione.roboblog.ui.activity.BlogDetailsActivity
import com.morozione.roboblog.ui.adapter.BlogsAdapter
import com.morozione.roboblog.utils.showSnackbar
import moxy.MvpAppCompatFragment

abstract class BlogsFragment : MvpAppCompatFragment() {

    private lateinit var binding: FragmentBlogsBinding
    protected var blogType = BlogType.GLOBAL
    protected lateinit var adapter: BlogsAdapter

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        adapter = BlogsAdapter(blogType, arrayListOf())
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
                binding = FragmentBlogsBinding.inflate(inflater,
container, false)
        initView(binding.root)

        return binding.root
    }

      override fun onViewCreated(view: View, savedInstanceState:
Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        setListener()
    }

    private fun initView(rootView: View) {
```

```
                                        val     mRecyclerView     =
rootView.findViewById<RecyclerView>(R.id.recycler_view)
        mRecyclerView.adapter = adapter
        mRecyclerView.setHasFixedSize(true)
        mRecyclerView.layoutManager = LinearLayoutManager(context)
    }

    private fun setListener() {
                        adapter.onBlogClickListener   =   object   :
BlogsAdapter.OnBlogClickListener {
            override fun onBlogClick(blog: Blog) {
                openBlogDetails(blog)
            }
        }
        binding.mSwipeRefresh.setOnRefreshListener {
            onUpdate()
        }
    }

    private fun openBlogDetails(blog: Blog) {
                               val    intent    =    Intent(context,
BlogDetailsActivity::class.java)

startActivity(BlogDetailsActivity.createBundleForBlog(intent,
blog.id, blogType))
    }

     protected fun setBlogs(blogs: List<Blog>, isLoading: Boolean)
{
        hideProgress()

        if (isLoading) {
            adapter.addData(ArrayList(blogs))
        } else {
            adapter.swapData(ArrayList(blogs))
        }
    }

    protected fun showError(messge: String) {
         view?.let { showSnackbar(it, messge, Snackbar.LENGTH_LONG)
}
    }

    protected fun showProgress() {
        binding.mSwipeRefresh.isRefreshing = true
    }

    protected fun hideProgress() {
        binding.mSwipeRefresh.isRefreshing = false
    }

    abstract fun onUpdate()
}
```

Додаток В

Клас CreateBlogFragment

```
package com.morozione.roboblog.ui.fragment

import android.os.Bundle
import com.google.android.material.snackbar.Snackbar
import android.text.TextUtils
import android.view.*
import moxy.presenter.InjectPresenter
import com.morozione.roboblog.R
import
com.morozione.roboblog.databinding.FragmentCreateBlogBinding
import com.morozione.roboblog.entity.Blog
import com.morozione.roboblog.mvp.presenter.CreateBlogPresenter
import com.morozione.roboblog.mvp.view.CreateBlogView
import com.morozione.roboblog.utils.GlideApp
import com.morozione.roboblog.utils.ImageUtil
import com.morozione.roboblog.utils.showSnackbar

class CreateBlogFragment : BaseImageFragment(), CreateBlogView {

    private lateinit var binding: FragmentCreateBlogBinding

    @InjectPresenter
    lateinit var presenter: CreateBlogPresenter

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setHasOptionsMenu(true)
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
       ) = FragmentCreateBlogBinding.inflate(inflater, container,
false).apply {
        binding = this
        root
    }.root

         override fun onCreateOptionsMenu(menu: Menu, inflater:
MenuInflater) {
        super.onCreateOptionsMenu(menu, inflater)
        inflater.inflate(R.menu.main_menu, menu)
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        when (item.itemId) {
            R.id.m_save -> creteBlog()
            R.id.m_image -> makePhoto()
        }
        return super.onOptionsItemSelected(item)
```

```
    }

    private fun creteBlog() {
        if (presenter.blog == null) {
            activity?.let { activity ->
                val blog = constructBlog(Blog())
                presenter.createBlog(blog, blog.icon, activity)
            }
        } else {
            presenter.blog?.let { blog ->
                presenter.updateBlog(constructBlog(blog))
            }
        }
    }

    private fun constructBlog(blog: Blog): Blog {
        blog.title = binding.mTitle.text.toString()
        blog.descrption = binding.mDescription.text.toString()
        blog.icon = imageUri.toString()
        imageUri = null
        return blog
    }

    override fun onError() {
        view?.let {
            showSnackbar(
                it,
                getString(R.string.something_was_wrong),
                Snackbar.LENGTH_SHORT
            )
        }
    }

    override fun onBlogCreated() {
           view?.let { showSnackbar(it, getString(R.string.saved),
Snackbar.LENGTH_SHORT) }
        imageUri = null
        emptyField()
    }

    override fun onBlogUpdated() {
          view?.let { showSnackbar(it, getString(R.string.updated),
Snackbar.LENGTH_SHORT) }
        emptyField()
        presenter.blog = null
    }

    private fun emptyField() {
        binding.mTitle.setText("")
        binding.mDescription.setText("")
        binding.mIcon.setImageBitmap(null)
    }
```

```
    fun setBlogForEdit(blog: Blog) {
        presenter.blog = blog

        binding.mTitle.setText(blog.title)
        binding.mDescription.setText(blog.descrption)
        if (imageUri == null) {
            context?.let {
                GlideApp.with(it)
                    .load(blog.icon)
                    .into(binding.mIcon)
            }
        }
    }

    override fun imageMade(imageUri: String?) {
                                if    (imageUri    !=    null    &&
!TextUtils.isEmpty(imageUri.toString())) {
                                                    val    bitmap    =
ImageUtil.decodeSampledBitmapFromResource(
                BaseImageFragment.imageUri.toString(),
                300,
                300
            )
            binding.mIcon.setImageBitmap(bitmap)
        }
    }
}
```