Plag

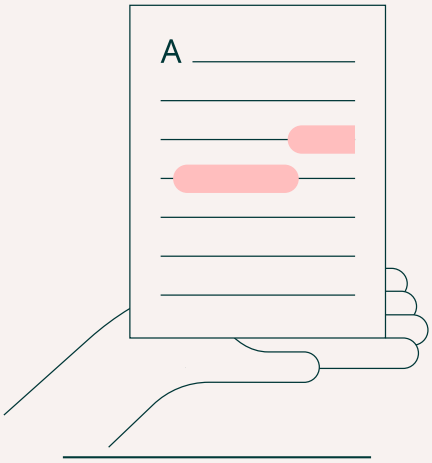
Звіт про оригінальність

● Оцінка схожості 4%

● Ризик плагіату СЕРЕДНІЙ

Ігор Кагало 2025-06-10 22:44

Посилання на звіт: 103Aj / Посилання користувача: qfC8



Ось вона –
Ваша звіт про оригінальність!

Ми раді повідомити, що перевірка вашого документа завершена, і результати вже готові! Наші алгоритми старанно працювали, щоб знайти збіги в наших базах даних.

На наступних сторінках ви знайдете результати перевірки:

Бали
Збіги
Посилання

Ваш документ було перевірено за такими джерелами:

- ✓ База даних інтернет-джерел
- ✓ База даних наукових статей
- ○ Глибока перевірка (наш вдосконалений алгоритм)

Бали

Збіги тексту	4%
Перефразування	0%
Цитований текст	0%
Неправильне цитування	0%
Збігів не знайдено	96%

Ризик плагіату **СЕРЕДНІЙ**

Ризик плагіату вказує, як збіги тексту розподілені по документу. Вищий ризик виникає, коли збіги з'являються близько один до одного, наприклад, у тому самому абзаці або розділі.

Оцінка схожості 4%

Оцінка схожості показує, скільки слів або символів у вашому документі збігаються з текстами інших документів, включаючи перефразовані тексти або неправильні цитати.

Збіги

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

4 ВІДОКРЕМЛЕНИЙ СТРУКТУРНИЙ ПІДРОЗДІЛ

4 «ФАХОВИЙ КОЛЕДЖ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ
4 «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

14 ПОЯСНЮВАЛЬНА 14 ЗАПИСКА

14 до дипломного проєкту

15 Фаховий молодший бакалавр

15 (освітньо-професійний 15 ступінь)

10 на тему: Розробка та впровадження мобільного додатку для внутрішньої комунікації в ІТ коледжі Львівської Полетіхенки

14 Виконав студент

14 IV

14 курсу, групи

14 ОК- 14 42

3 ОПП «Обслуговування 3 комп’ютерних систем та мереж»

3 Спеціальності 3 123 Комп'ютерна інженерія

3 (прізвище, ім'я по батькові)

3 Керівник

Мирослава Шеремета

(підпис) (ім'я прізвище)

Нормоконтролер

Любомира Кужій

(підпис) (ім'я 10 прізвище)

10 Рецензент

10 (підпис) (ім'я прізвище)

Голова ЕК

Олег Гіщак

(підпис) (ім'я прізвище)

Члени ЕК

Любомира Кужій

(підпис) (ім'я прізвище)

Андрій Селемонавічус

(підпис) (ім'я прізвище)

Дипломний проєкт захищений в ЕК « __ » _____2025 р.

з оцінкою « _____ »

Львів 2025

15 НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА» 4 ВІДОКРЕМЛЕНИЙ СТРУКТУРНИЙ ПІДРОЗДІЛ

4 «ФАХОВИЙ КОЛЕДЖ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

4 НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ 15 «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

4 Циклова комісія

3 Комп'ютерних систем і мереж

3 Освітньо-професійний 4 ступінь

4 Фаховий молодший бакалавр

4 Освітньо-професійна 4 програма

3 Обслуговування 3 комп’ютерних систем та мереж

3 Спеціальність

3 123 Комп'ютерна інженерія

3 ЗАТВЕРДЖУЮ

3 Завідувач відділення

3 «Комп’ютерних систем і мереж»

3 _____ Володимир СТАХІВ

«____» _____ 2025 року

ЗАВДАННЯ

НА ДИПЛОМНИЙ ПРОЄКТ СТУДЕНТУ

Сабірову Матвію Олеговичу

(прізвище, 10 ім'я та 1 по батькові)

1 1. Тема проєкту

1 Розробка та впровадження мобільного додатку для

внутрішньої комунікації в IT коледжі Львівської Політехніки

керівник проєкту

Шеремета Мирослава Ярославівна

(1 прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

1 затверджені наказом директора від «20» березня 2025 року № 20 - ст

1 2. Строк подання студентом проєкту «10» червня 2025 10 року

1 Вихідні дані до проєкту

1 3.1 Операційна система Android

3.2 Середовище Android SDK

3.3 Мова програмування Kotlin

1 Зміст розрахунково-пояснювальної записки

1 4.1 Теоретичні основи проєктування мобільних застосунків

4.2 Проєктування мобільних застосунків засобами Kotlin

4.3 Реалізація проєкту

4.4 Техніко-економічне обгрунтування

4.5 Охорона праці та безпека 25 життєдіяльності

1 5. Перелік графічного матеріалу

1 5.1.

Лист 1 Схема взаємодії модулів додатку

5.2.

Лист 2 Схематичне зображення бази даних

5.3.

Лист 3 Схематичне зображення архітектури

5.4.

Лист 4 Кошторис витрат на проєкт

1 6 Консультанти розділів проєкту

6 Розділ

6 Ім'я, прізвище 1 та посада консультанта

1 Підпис, дата

1 Завдання видав

Завдання отримав

Техніко-економічне обґрунтовування

Тетяна Підкуймуха

Охорона праці та безпека життєдіяльності

Роман Томків

1 7. Дата видачі завдання « 01»квітня 2025 6 року

6 КАЛЕНДАРНИЙ ПЛАН

6 №

6 з/п

25 Назва етапів дипломного проєкту

25 Термін

виконання

Примітка

1

Огляд існуючого завдання

02.04.2025

2

Вибір засобів проєктування

05.04.2025

3

Створення архітектури додатку

08.04.2025

4

Реалізація механік додатку

15.05.2025

5

Техніко-економічне обгрунтування

17.05.2025

6

Охорона праці та безпека життєдіяльності

20.05.2025

7

Розробка переліку умовних позначень, реферату, змісту, вступу та висновку

25.05.2025

8

Розробка графічного змісту димломного проєкту

27.05.2025

Студент

Матвій Сабіров

(6 підпис)

6 (ім'я, прізвище)

6 Керівник проєкту

Мирослава Шеремета

(підпис)

(ім'я, прізвище)

РЕФЕРАТ

Текстова частина дипломного проєкту : 72 ст., 13 рис., 3 табл., 18 посилань.

Об'єкт проєктування – розробка логіки взаємодії та поведінки елементів користувацького інтерфейсу мобільного застосунку на платформі Android із використанням мови програмування Kotlin у середовищі Android Studio.

Мета 6 виконання дипломного проєкту полягає у створенні повноцінної логіки мобільного застосунку для Android, яка включає управління елементами інтерфейсу,

обробку подій користувача, реалізацію навігації між екранами, збереження даних, інтеграцію з локальними базами даних та використання можливостей апаратного забезпечення смартфонів. Усі компоненти реалізовано з використанням Kotlin та Android SDK у середовищі Android Studio.

Галузь використання – розробка мобільних застосунків для Android, зокрема програм з розширеною логікою взаємодії з користувачем, де необхідно забезпечити надійне управління даними, чуйний UI/UX та інтеграцію з системними сервісами пристрою.

У дипломному проєкті проаналізовано сучасні підходи до створення мобільних застосунків, розглянуто популярні архітектурні шаблони (MVVM, MVI), бібліотеки та інструменти Android SDK. Визначено переваги використання Kotlin у порівнянні з Java для Android-розробки, а також обґрунтовано вибір Android Studio як основного середовища розробки.

ANDROID, 16 KOTLIN, ANDROID SDK, ANDROID STUDIO, МОБІЛЬНИЙ ЗАСТОСУНОК, MVVM, FIREBASE, FIRESTORE, UI/UX, МОБІЛЬНА РОЗРОБКА

ВСТУП

У сучасних умовах цифрової трансформації ефективна внутрішня комунікація є ключовим чинником 11 успішної діяльності будь-якої організації чи навчального закладу. Особливо це стосується освітніх установ, де швидкий обмін інформацією між студентами, викладачами та адміністрацією є надзвичайно важливим. З огляду на це, розробка спеціалізованих мобільних додатків, орієнтованих на потреби конкретного закладу, набуває особливої актуальності.

Цей дипломний проєкт присвячено розробці та впровадженню 12 мобільного застосунку для внутрішньої комунікації у Фаховому коледжі інформаційних технологій НУ "Львівська політехніка" (ФКІТ). Основною метою є створення зручного, функціонального та безпечного інструменту для обміну повідомленнями, оголошеннями, матеріалами та іншою інформацією між учасниками навчального процесу.

У розробці використано сучасний стек технологій: 16 мову програмування Kotlin, середовище розробки Android Studio, а також хмарну платформу Firebase, зокрема її компонент Firestore, який забезпечує зберігання та синхронізацію даних 11 у режимі реального часу. Цей вибір зумовлений потребою у швидкій обробці повідомлень, зручному адмініструванні даних та масштабованості застосунку.

У межах роботи проаналізовано існуючі рішення для внутрішньої комунікації та обґрунтовано доцільність створення власного мобільного додатку з урахуванням

специфіки ФКІТ. Проєкт охоплює проєктування архітектури застосунку, реалізацію користувацького інтерфейсу, розробку логіки взаємодії з базою даних Firestore, обробку повідомлень та сповіщень, а також тестування функціоналу.

12 Особливу увагу приділено забезпеченню зручності користування, безпеці даних і можливості легкого масштабування функціоналу у майбутньому. Застосунок також передбачає рольову модель доступу для студентів , що дає змогу гнучко організувати взаємодію із навчанням в коледжі.

1 ТЕОРЕТИЧНІ ОСНОВИ ПРОЄКТУВАННЯ МОБІЛЬНИХ ЗАСТОСУНКІВ

Особливості мобільної розробки для Android

Мобільна розробка є однією з найдинамічніших і затребуваних галузей сучасного програмування. Створення застосунків для операційної системи Android вимагає від розробника не лише знань мови програмування Kotlin, а й глибокого розуміння архітектури платформи, принципів UI/UX-дизайну та взаємодії з хмарними сервісами. Особливості Android-розробки суттєво впливають на всі етапи створення застосунку – від проєктування до впровадження та тестування.

Щоб зрозуміти особливості механіки розробки, варто розглянути її основні аспекти. Платформа Android є однією з найпоширеніших у світі, охоплюючи широкий спектр пристроїв – від бюджетних смартфонів до флагманських моделей. Серед основних переваг Android-розробки – відкритість платформи, велика база користувачів, потужний інструментарій SDK, регулярне оновлення фреймворків та велика кількість бібліотек з відкритим кодом. Kotlin, як офіційно підтримувана мова для Android, забезпечує лаконічність синтаксису, безпечну роботу з null-значеннями та хорошу інтеграцію з інструментами Google.

Разом з тим, розробка для Android має свої виклики. Основним є фрагментація пристроїв – різноманіття розмірів екранів, версій Android та апаратного забезпечення потребує ретельного тестування застосунку. Також важливо враховувати обмеження продуктивності, енергоспоживання та використання мобільного трафіку. Розробник має знаходити баланс між функціональністю застосунку та оптимізацією його роботи для забезпечення якісного користувацького досвіду.

Операційна система Android побудована на багаторівневій архітектурі, яка включає в себе ядро Linux, бібліотеки та runtime-середовище, фреймворк Android та рівень застосунків. Ця структура дозволяє розробникам отримати доступ до низькорівневих системних функцій при необхідності, одночасно працюючи на високому рівні абстракції за допомогою API Android SDK.

Ключовим елементом у створенні застосунків є Android Application Framework, який

надає набір сервісів для обробки UI, доступу до бази даних, управління ресурсами, локальними файлами, мережевими запитами тощо. Знання архітектури системи допомагає ефективно будувати взаємодію між компонентами застосунку, а також правильно управляти ресурсами пристрою, щоб уникнути витоків пам’яті, зависань чи аварійного завершення процесу.

У мобільній розробці для Android розуміння життєвого циклу компонентів є критично важливим для стабільності, продуктивності та передбачуваної поведінки застосунку. Кожен основний компонент має свій набір подій життєвого циклу, які розробник повинен враховувати для керування ресурсами, збереження стану інтерфейсу та обробки фонових задач. Окрім класичних компонентів (Activity, Fragment, Service), варто враховувати й новітні підходи Jetpack Compose, ViewModel, та життєвий цикл самого застосунку через Application.

Activity — основний компонент, що представляє один екран з інтерфейсом користувача. Його життєвий цикл включає наступні методи:

onCreate() — викликається при створенні активності. Тут ініціалізуються UI-компоненти, ViewModel, прив’язки до layout або Compose;

onStart() — активність стає видимою;

onResume() — активність у фокусі й взаємодіє з користувачем;

onPause() — активність втрачає фокус (наприклад, при відкритті іншої активності або діалогу);

onStop() — активність більше не видима;

onRestart() — викликається після onStop() перед повторним onStart();

onDestroy() — фінальне завершення, перед очищенням пам’яті.

На рисунку 1.1 представлене схематичне зображення життєвого циклу Activity:

Рисунок 1.1 – Життєвий цикл Activity

У Jetpack Compose ці методи все ще використовуються для керування життєвим циклом, але сама UI-логіка не прив'язується до XML.

Fragment — гнучкий компонент, який є частиною UI або поведінки активності. Має складніший життєвий цикл:

onAttach();

onCreate();

onCreateView();

onViewCreated();

onStart();

onResume();

onPause();

onStop();

onDestroyView();

onDestroy();

onDetach().

Загалом логіка методів життєвого циклу у Fragment та Activity схожа. Фактично з нового лише onAttach() та onDetach() – методи, які необхідні, щоб Fragment “прикріпився” до батьківської Activity. Особливо важливо розділяти onCreate() (логіка без UI) і onCreateView() (створення UI), а також правильно очищати ресурси в onDestroyView().

Service — компонент для виконання довготривалих операцій у фоні (наприклад, синхронізація, музика). Методи життєвого циклу Service:

onCreate() — створення сервісу;

onStartCommand() — запуск виконання;

onBind() / onUnbind() — для BoundService, якщо треба прив'язати клієнта;

onDestroy() — завершення роботи сервісу.

З появою WorkManager та JobScheduler, класи Service використовуються переважно для специфічних випадків (наприклад, Foreground Service).

Application — глобальний життєвий цикл застосунку, з методом onCreate() — викликається при старті процесу. Тут можна ініціалізувати бібліотеки (Firebase, DI, аналітику). Це точка входу до всього застосунку, працює до завершення процесу.

ViewModel — частина MV* архітектури, яка призначена для зберігання даних та бізнес-логіки незалежно від життєвого циклу UI. Існує доти, доки живе Activity або Fragment, що його створило. Має метод onCleared() — викликається, коли ViewModel знищується

(наприклад, при завершенні Activity). Це дозволяє зберігати дані при повороті екрана, повторному створенні UI тощо.

У Jetpack Compose немає класичного життєвого циклу View, оскільки UI формується декларативно. Але є свої правила:

Recomposition: коли змінюється State, функція Composable викликається заново;

LaunchedEffect: дозволяє запускати побічні ефекти (наприклад, запити) при певних умовах життєвого циклу;

remember{} і rememberSaveable{}: для збереження стану між recomposition;

DisposableEffect: виконується при виході з Composition (аналог onDestroy для UI);

SideEffect, SnapshotFlow, LifecycleOwner.current — додаткові механізми для реакції на зміни в життєвому циклі.

Також можна використовувати LifecycleEventObserver з LocalLifecycleOwner.current, щоб відслідковувати життєвий цикл Activity у Compose.

Таким чином, мобільна розробка для Android є складним, але надзвичайно гнучким процесом, який поєднує роботу з багаторівневою системною архітектурою, специфікою життєвих циклів компонентів і сучасними підходами до створення інтерфейсу. Розуміння життєвих циклів таких компонентів, як Activity, Fragment, Service, ViewModel, а також особливостей Jetpack Compose, дає змогу ефективно управляти ресурсами, зберігати стан застосунку і забезпечувати стабільну взаємодію з користувачем. Крім того, правильне проєктування структури застосунку з урахуванням фрагментації Android-пристроїв, роботи в умовах обмежених ресурсів і інтеграції з хмарними сервісами, такими як Firebase, є критично важливим для досягнення високої якості, продуктивності та зручності використання мобільного застосунку.

Вибір 22 мови програмування та середовища розробки

22 Java історично є основною мовою для створення Android-застосунків. Вона з’явилася в Android SDK ще з перших версій системи, має розвинену екосистему, стабільну документацію та тисячі бібліотек. Java базується на об'єктно-орієнтованій парадигмі, що дозволяє будувати масштабовані системи з чітким розподілом обов'язків. Разом з тим, через роки активного використання виявилися і її недоліки, зокрема:

Надмірна багатослівність синтаксису;

Відсутність вбудованого механізму роботи з null-безпечністю;

Обмежені можливості функціонального програмування;

Уповільнене впровадження нових мовних конструкцій.

Для вирішення цих проблем компанія JetBrains у 2011 році анонсувала мову Kotlin — сучасну статично типізовану мову, сумісну з Java Virtual Machine (JVM), яка у 2017 році була офіційно визнана Google як рекомендована мова для Android-розробки.

Kotlin було спроєктовано як ефективну, лаконічну та безпечну альтернативу Java. Серед ключових переваг Kotlin:

Коротший код: завдяки конструкціям, як data class, when, extension functions, type inference, Kotlin дозволяє писати менше коду з тим же функціоналом;

Безпечна робота з null: Kotlin реалізує систему null-safety на рівні компілятора, що запобігає помилкам типу NullPointerException, які є одними з найпоширеніших у Java;

Функціональне програмування: підтримка лямбда-виразів, inline-функцій, map/filter/reduce-операцій дає можливість писати декларативний, чистий і читабельний код;

Повна сумісність із Java: Kotlin компілюється у JVM-байткод і може працювати поруч із Java-класами без конфліктів. Це дозволяє поступово переводити проєкти з Java на Kotlin без повного переписування;

Покращена підтримка Android API: завдяки Kotlin Android Extensions, Jetpack Compose, DSL-підходам, розробка Android-застосунків стає зручнішою, ніж у Java;

Kotlin швидко завоював популярність серед Android-розробників завдяки своїй зручності, продуктивності та гнучкості. За останні роки практично всі нові Android-проєкти в Google створюються саме на Kotlin, а більшість навчальних матеріалів, туторіалів і документації тепер орієнтовані на Kotlin-код.

При розробці мобільного застосунку для внутрішньої комунікації студентів ФКІТ вибір на користь Kotlin був абсолютно обґрунтованим. Серед практичних переваг цієї мови в межах даного проєкту можна виділити багато плюсів.

Kotlin чудово інтегрується з бібліотеками Firebase, включно з Firestore, Authentication, Realtime Database, Cloud Messaging тощо. Завдяки Kotlin Coroutines можна реалізовувати асинхронні виклики Firestore API просто та читабельно, без вкладених callback-структур.

Замість десятків рядків коду, як це буває в Java (наприклад, при створенні класів-

моделей, парсингу даних чи реалізації callback-інтерфейсів), у Kotlin завдяки data class, sealed class та inline-конструкціям усе це реалізується набагато коротше.

Завдяки можливості чітко контролювати scope життєвого циклу (наприклад, через viewModelScope, lifecycleScope) і обробку помилок (через try-catch, Result, runCatching) код стає більш стабільним і передбачуваним.

Сучасна декларативна система побудови UI в Android (Jetpack Compose) підтримується лише в Kotlin, що робить її єдиним вибором для розробників, які хочуть працювати з найновішими фреймворками Android.

Таким чином, Kotlin не лише 9 дозволяє зменшити кількість коду, а й підвищує його читабельність, стабільність та відповідність сучасним стандартам мобільної розробки.

У 2 розробці програмного забезпечення важливо мати 2 ефективне, надійне та потужне 2 інтегроване середовище розробки (IDE), яке забезпечує повний цикл створення застосунку — від проєктування інтерфейсу до складання фінального APK-файлу. Для розробки Android-застосунків таким середовищем є Android Studio — офіційне IDE, що підтримується компанією Google та побудоване на основі платформи IntelliJ IDEA від JetBrains.

Android Studio є комплексним рішенням, яке включає в себе всі необхідні інструменти для повноцінної мобільної розробки: підтримку мови Kotlin, управління залежностями, UI-редактори, емулятори пристроїв, засоби тестування, профілювання, налагодження, публікації, а також інтеграцію з хмарними сервісами, зокрема Firebase.

Android Studio 2 має вбудовану підтримку мови Kotlin, що дозволяє розробляти сучасні, лаконічні та безпечні застосунки. IDE автоматично надає підказки щодо синтаксису, дозволяє швидко переходити між класами, реалізовує інтелектуальне завершення коду (smart autocomplete), проводить рефакторинг змін на рівні всього проєкту. Окрім того, Android Studio дозволяє розробляти інтерфейс користувача двома основними способами:

Класичний XML-редактор, що включає інтерактивний візуальний редактор ("Design Mode") та можливість ручного редагування розмітки;

Jetpack Compose Preview — сучасний декларативний підхід до UI, який дає змогу переглядати вигляд компонента без запуску застосунку. Це значно прискорює цикл розробки та тестування UI-компонентів.

Завдяки цим засобам розробник може швидко створювати 7 адаптивні інтерфейси для різних типів екранів і конфігурацій пристроїв.

Android Studio включає Android Virtual Device (AVD) Manager, який дозволяє створювати емулятори з різними версіями Android, розмірами екранів, об’ємом пам’яті тощо. Це дає змогу швидко тестувати застосунок без фізичного пристрою.

Крім того, IDE підтримує підключення реального пристрою через USB або Wi-Fi, з відлагодженням 7 у реальному часі через Logcat, Device Explorer, Layout Inspector та інші інструменти.

Android Studio базується на Gradle — потужній системі збирання, яка автоматизує компіляцію проєкту, керування залежностями, створення кількох варіантів складання (debug/release) та CI/CD-інтеграцію. Gradle дозволяє використовувати різні бібліотеки (наприклад, Material3, Coroutine Flow, Room, Firebase SDK) без ручного налаштування jar-файлів.

Крім того, Gradle забезпечує оптимізацію розміру APK, обфускацію коду (через ProGuard або R8), перевірку підпису та налаштування для публікації в Google Play.

Android Studio надає широкі можливості для дебагу і профілювання продуктивності:

Logcat — перегляд журналу подій застосунку 7 в реальному часі;

7 Debugger — можливість ставити breakpoints, переглядати змінні, виклики функцій та стек;

Memory Profiler — аналіз використання оперативної пам’яті, виявлення витоків;

CPU Profiler — моніторинг обчислювального навантаження застосунку;

Network Profiler — аналіз HTTP-запитів, що дозволяє перевіряти правильність роботи з Firestore.

Ці інструменти дозволили провести ефективне тестування застосунку, виявити "вузькі місця" в логіці обміну даними та оптимізувати роботу з мережею.

Архітектурні шаблони в Android-розробці

У сучасній мобільній розробці архітектура застосунку є критично важливим аспектом, який визначає зручність масштабування, підтримки, повторного використання коду, а також забезпечує передбачувану поведінку інтерфейсу. Без чіткої архітектурної моделі код швидко стає неструктурованим, а зміни призводять до помилок і дублювання логіки. Саме тому в Android-розробці активно використовуються архітектурні патерни, які розділяють обов’язки між логікою представлення, бізнес-логікою та доступом до даних.

Серед найбільш поширених архітектурних шаблонів можна виділити MVVM (Model–View–ViewModel) — офіційно рекомендований Google, а також MVI (Model–View–Intent), який набирає популярності, особливо у зв'язці з Jetpack Compose і StateFlow.

MVVM (Model–View–ViewModel) — це архітектурна модель, яка розділяє відповідальність між трьома основними компонентами:

Model — містить бізнес-логіку, роботу з репозиторіями, мережевими або локальними джерелами даних (наприклад, Firebase Firestore, Room);

View — відповідає за відображення інформації, UI-елементи та реакцію на дії користувача;

ViewModel — зв'язує Model та View. Отримує дані з Model, трансформує їх у форму, зручну для відображення, і повідомляє View про зміни через реактивні потоки (LiveData, StateFlow тощо).

На рисунку 1.2 зображена схема паттерну MVVM:

Рисунок 1.2 – Схема паттерну MVVM

У традиційній XML-базованій розробці зв'язок між ViewModel і View здійснюється через LiveData, яку View (наприклад, Activity або Fragment) підписується на спостереження (observe). У сучасних реалізаціях, особливо з Jetpack Compose, на зміну LiveData дедалі частіше приходить StateFlow — більш передбачуваний та декларативний API 2 для роботи з потоками станів.

Переваги MVVM:

Чітке розділення обов'язків;

Можливість повторного використання логіки у ViewModel;

Зменшення залежності View від джерел даних;

Проста перевірка бізнес-логіки через unit-тести ViewModel.

У зв'язці з Jetpack Compose, дедалі більшого поширення набуває MVI (Model–View–Intent) — архітектурна модель, яка особливо добре підходить до декларативного підходу побудови інтерфейсу.

Ключові компоненти MVI:

Model / State — єдине джерело правди, яке зберігає повний стан інтерфейсу у вигляді

data-класу;

View — Composable-функція, яка реагує 13 на зміни стану і відображає UI;

Intent / Event — події, які ініціює користувач або система (наприклад, клік кнопки, завантаження екрана, зміна тексту);

Reducer — логіка, яка перетворює старий стан і подію на новий стан;

Effect — одноразові побічні ефекти, як-от показ діалогу, навігація або тост.

На рисунку 1.3 зображена схема паттерну MVI:

Рисунок 1.3 – Схема паттерну MVI

Особливості MVI:

Односпрямований 13 потік даних (Unidirectional Data Flow) — стан оновлюється лише через Intent Reducer State View.

Immutable State — стан не змінюється напряму, а завжди створюється новий.

Простота відтворення — будь-який стан можна зберегти і "відмотати", що корисно для тестування і відладки.

У моєму проєкті, де використовується Jetpack Compose із StateFlow, саме MVI виявився найбільш зручним і логічним способом реалізації архітектури. Завдяки цьому підходу 9 кожен екран має чітко визначений стан (UiState), набір подій (UiEvent) та побічних ефектів (UiEffect). ViewModel відповідає за обробку подій та генерацію нового стану, а сам інтерфейс автоматично оновлюється при кожній зміні стану.

Чітке розділення відповідальностей — основа якісної архітектури. У межах MVVM або MVI, типова структура застосунку дотримується таких принципів:

View (UI/Compose): лише відображає дані. Не містить бізнес-логіки, не звертається до репозиторіїв напряму;

ViewModel: керує станом UI, обробляє події (Intent/Events), викликає відповідні методи в Model/Repository, трансформує результат у State;

Repository / UseCase: є посередником між ViewModel та джерелами даних (Firestore, Room, API);

Model / Data Source: безпосередня реалізація доступу до даних, робота з Firestore,

кешем, SharedPreferences тощо.

Такий поділ забезпечує тестованість, масштабованість і гнучкість у подальшій розробці.

ViewModel — основний компонент архітектури, який зберігає стан незалежно від життєвого циклу View. У ViewModel реалізується бізнес-логіка, обробка подій, виклики до репозиторіїв. Працює разом із viewModelScope для безпечного виконання корутин.

LiveData — класичний спосіб передавати дані з ViewModel у View. Автоматично враховує життєвий цикл, що спрощує підписку у Fragment/Activity. Проте має обмеження при роботі з Compose.

StateFlow — сучасна альтернатива LiveData на базі Kotlin Coroutines. Переваги:

Підтримка flow-операторів (map, combine, debounce);

Повністю сумісна з Jetpack Compose;

Просте збереження останнього стану;

Краще контрольована підписка і відписка.

У межах дипломного проєкту всі екрани реалізовано за допомогою MVI + StateFlow, що забезпечило чисту архітектуру, передбачувану реакцію на події та легкість масштабування.

Сучасна Android-розробка вимагає не лише володіння мовою програмування, а й правильного проєктування архітектури застосунку. Використання шаблонів MVVM і MVI забезпечує чітку структуру, розділення логіки та UI, стабільність і масштабованість застосунку. Завдяки ViewModel, StateFlow та односпрямованому потоку даних, можна ефективно керувати станом інтерфейсу, спростити обробку подій користувача та підвищити якість коду. У контексті Jetpack Compose саме MVI виявився найбільш зручним і відповідним архітектурним підходом, який дозволив реалізувати надійний та розширюваний інтерфейс застосунку.

Хмарні технології у мобільних застосунках: Firebase

З розвитком мобільних технологій дедалі актуальнішим стає використання хмарних сервісів, які дозволяють зберігати, обробляти та синхронізувати дані поза межами пристрою користувача. Завдяки хмарним технологіям мобільні застосунки можуть залишатися легкими, масштабованими, отримувати дані в режимі реального часу та забезпечувати узгоджений користувацький досвід на кількох пристроях. Одним із найпопулярніших рішень у цій галузі є платформа Firebase, розроблена компанією Google.

Firebase — це хмарна платформа для мобільної та веб-розробки, яка надає широкий набір сервісів: базу даних у реальному часі, систему автентифікації, хмарне сховище, аналітику, систему сповіщень, краш-репорти, хостинг тощо. Особливістю Firebase є його тісна інтеграція з Android SDK, простота впровадження та підтримка масштабованих серверless-архітектур.

Основні компоненти платформи Firebase:

Authentication — модуль авторизації, який дозволяє реєструвати та автентифікувати користувачів за допомогою email/пароля, соціальних мереж, анонімного входу тощо;

Cloud Firestore — гнучка, масштабована база даних NoSQL з підтримкою офлайн-доступу;

Realtime Database — альтернативна 2 база даних з меншою гнучкістю, але миттєвою синхронізацією;

Firebase Storage — зберігання файлів (зображення, відео, документи);

Firebase Cloud Messaging (FCM) — система push-сповіщень;

Firebase Crashlytics — інструмент для моніторингу збоїв у застосунку;

Firebase Analytics — збір поведінкових даних користувача для подальшої аналітики.

У межах даного дипломного проєкту головним сервісом була використана база даних Firestore, яка забезпечила централізоване зберігання повідомлень, оголошень та користувацьких профілів із синхронізацією в реальному часі. На рисунку 1.4 представлений вигляд адмін-панелі Firebase.

Рисунок 1.4 – Адмін панель Firebase

Cloud Firestore — це хмарна база даних NoSQL, яка використовує ієрархічну структуру "колекція документ підколекція". Основною одиницею є документ — набір пар "ключ–значення", що може містити прості типи даних (рядки, числа, масиви), вкладені об’єкти або навіть посилання на інші документи. Документи об’єднуються в колекції, які можуть містити інші підколекції, утворюючи дерево даних. Основні переваги Firestore:

Гнучкість схеми — відсутність жорсткого типізованого схематичного опису, як у реляційних базах;

Автоматична синхронізація даних між пристроями, коли Firestore підключено до інтернету.;

Локальний кеш з офлайн-підтримкою — усі дані зберігаються на пристрої користувача, що дозволяє працювати із застосунком навіть без доступу до мережі.;

Безпека — гнучкі правила доступу на рівні колекцій, документів або навіть окремих полів;

Масштабованість — розподілене хмарне зберігання, яке підтримує великі обсяги трафіку та користувачів.

5 Однією з ключових переваг Firestore у контексті мобільної розробки є підтримка роботи 5 в реальному часі. Це досягається через механізм підписок (addSnapshotListener), який дозволяє відслідковувати 9 зміни в базі даних миттєво та оновлювати інтерфейс користувача без необхідності ручного оновлення. Наприклад, коли один студент надсилає повідомлення, воно одразу з'являється на екрані іншого користувача, без перезавантаження екрану або додаткового запиту до сервера.

Ще однією важливою функцією є офлайн-синхронізація. Firestore автоматично зберігає копію даних у локальному кеші пристрою. 2 Це означає, що користувач може переглядати повідомлення або створювати нові 5 навіть без підключення до інтернету — зміни синхронізуються автоматично, щойно з'являється мережа.

Для дипломного проєкту це особливо важливо, оскільки в умовах навчального закладу інтернет-з'єднання може бути нестабільним. Завдяки офлайн-підтримці застосунок залишається функціональним у будь-який момент.

Хмарна платформа Firebase, а особливо її компонент Firestore, надала всі необхідні інструменти для реалізації стабільного, гнучкого та масштабованого мобільного застосунку. Завдяки простоті інтеграції, підтримці роботи 5 в реальному часі та офлайн-режимі, а також автоматичній синхронізації даних між пристроями, Firestore став ключовим елементом інфраструктури дипломного проєкту. У поєднанні з Kotlin та Jetpack Compose, Firebase значно спростив реалізацію логіки обміну повідомленнями та зберігання даних у застосунку.

Вимоги до користувацького інтерфейсу мобільних застосунків

Користувацький інтерфейс 21 (UI) та користувацький досвід (UX) відіграють ключову роль у сприйнятті мобільного застосунку. Навіть за наявності потужної логіки та багатого функціоналу, незручний або застарілий інтерфейс може суттєво знизити інтерес користувачів. Тому під час розробки Android-застосунків 24 особливу увагу приділяють дотриманню принципів сучасного UI/UX-дизайну, адаптивності та технічної реалізації інтерфейсу.

Google надає рекомендації щодо дизайну мобільних застосунків у рамках системи

Material Design. Вона охоплює такі аспекти:

Інтуїтивність — користувач має легко орієнтуватися у застосунку без потреби в інструкції;

Консистентність — усі екрани мають дотримуватися єдиного стилю, кольорової палітри, типографіки;

Реактивність — інтерфейс має оперативно 2 реагувати на дії користувача (анімації, візуальний зворотній зв'язок);

Простота — чітка ієрархія елементів, мінімалізм та фокус на головному;

Доступність — підтримка різних мов, масштабування шрифтів, читабельність.

Android-платформа охоплює 19 велику кількість пристроїв із різними технічними характеристиками, серед яких варто відзначити різні розміри дисплеїв, співвідношення сторін та щільність пікселів. У зв'язку з цим під час розробки інтерфейсу особливої уваги потребує адаптивність — здатність застосунку коректно виглядати й функціонувати на будь-якому екрані. Це досягається завдяки поєднанню кількох технічних підходів.

Одним із найбільш ефективних рішень у Jetpack Compose є використання модифікаторів, які дозволяють елементам займати всю доступну ширину або змінюватися відповідно до обмежень батьківського контейнера. Компонент BoxWithConstraints, наприклад, дає змогу враховувати габарити екрана під час побудови інтерфейсу, що особливо корисно для створення адаптивних макетів. Для більш класичних підходів у XML-розмітці застосовується ConstraintLayout — гнучкий інструмент, який 2 дозволяє створювати складні інтерфейси з точним позиціюванням елементів. На рисунку 1.5 представлено вигляд інтерфейсу XML-редактора:

Рисунок 1.5 – Вигляд інтерфейсу XML редактора

Крім того, важливим аспектом адаптації є використання файлів ресурсів, зокрема dimens.xml, де визначаються окремі розміри для різних категорій екранів. Це дозволяє масштабувати інтерфейс відповідно до характеристик конкретного пристрою. Ще одним елементом адаптивного підходу є використання векторної графіки, яка забезпечує якісне відображення іконок та зображень незалежно від роздільної здатності дисплея.

У сучасній Android-розробці існують два основні підходи до створення інтерфейсу користувача — імперативний і декларативний. Класичний імперативний підхід передбачає створення розмітки у форматі XML-файлів, де компоненти інтерфейсу

описуються окремо від логіки керування. Цей метод залишається широко поширеним і підтримується всіма версіями Android, проте він вимагає більше коду для динамічної побудови інтерфейсу та складнішої інтеграції зі станом застосунку.

Натомість Jetpack Compose представляє собою декларативну модель розробки інтерфейсу, у якій компоненти UI описуються безпосередньо у Kotlin-коді. Такий підхід забезпечує вищу швидкість розробки, зменшення обсягів коду та гнучкі можливості повторного використання елементів. Завдяки тісній інтеграції з реактивними потоками, зокрема з State і StateFlow, інтерфейс у Compose автоматично оновлюється при зміні стану. Це значно спрощує логіку побудови екранів та покращує підтримку архітектурних патернів на кшталт MVI.

Ще однією перевагою Jetpack Compose є можливість створення попереднього перегляду компонентів без запуску застосунку, що суттєво скорочує час ітерацій під час розробки. У межах цього проєкту весь користувацький інтерфейс було реалізовано саме з використанням Jetpack Compose. Такий підхід дозволив ефективно поєднати декларативну побудову UI з сучасною архітектурою, забезпечити адаптивність та просту інтеграцію зі станом у ViewModel, що особливо важливо в контексті застосування шаблону MVI.

ПРОЄКТУВАННЯ МОБІЛЬНИХ ЗАСТОСУНКІВ ЗАСОБАМИ KOTLIN

Особливості проєктування Android-застосунків мовою Kotlin

Android-застосунок складається з набору компонентів, які взаємодіють між собою в межах операційної системи. Основними з них є Activity, Fragment, Service, BroadcastReceiver та ContentProvider. Вони керуються системою Android і мають чітко визначений життєвий цикл, що вимагає від розробника враховувати особливості їх ініціалізації, збереження стану та звільнення ресурсів.

Структура сучасного Android-застосунку зазвичай реалізується на базі певної архітектурної моделі (MVVM або MVI) та поділяється на кілька рівнів:

UI-рівень, що містить елементи інтерфейсу користувача;

ViewModel-рівень, де зберігається логіка взаємодії між UI та даними;

Data/Repository-рівень, який відповідає за доступ до джерел інформації (локальних або віддалених).

Застосунок організовується у вигляді модулів або пакунків за функціональністю, що дозволяє забезпечити логічну ізоляцію частин програми, полегшити масштабування та підтримку. У такій структурі важливо чітко розмежовувати обов’язки між шарами,

уникати дублювання логіки та забезпечувати тестованість компонентів. На рисунку 2.1 представлена схема базової структури Android:

Рисунок 2.1 – Базова структура Android

Мова Kotlin значною мірою спрощує архітектурне проєктування Android-застосунків. По-перше, вона надає зручний синтаксис, який дозволяє скоротити обсяг коду без втрати функціональності. Наприклад, замість створення повноцінного класу з геттерами, конструкторами та перевизначенням equals() достатньо використати одну конструкцію data class.

По-друге, Kotlin забезпечує безпеку типів, зокрема через систему null-безпечності. Це знижує ймовірність виникнення критичних помилок, пов’язаних із посиланням на об’єкти, які не були ініціалізовані, що є особливо важливим у багатопотокових або асинхронних сценаріях взаємодії з мережею чи базою даних.

Ще одним важливим інструментом є Kotlin Coroutines — легкий механізм управління асинхронними викликами, який дозволяє уникати складних callback-структур. Завдяки цьому бізнес-логіку застосунку можна реалізувати у прямолінійному стилі, що робить її легшою для читання, тестування та підтримки.

Крім того, Kotlin має потужну підтримку функціонального програмування — лямбда-вирази, вищі порядки функцій, розширення (extensions functions), sealed-класи. Усі ці елементи дозволяють розробникам впроваджувати архітектурні патерни на рівні мови без потреби у надмірній кількості шаблонного коду.

У проєктуванні інтерфейсів користувача в Android тривалий час домінував імперативний стиль, за якого елементи UI описуються у XML-файлах, а логіка їхнього управління реалізується у відповідних класах (Activity, Fragment). У цьому підході View безпосередньо контролює логіку, змінює стан елементів і реагує на дії користувача. Це ускладнює підтримку та тестування коду, особливо при роботі зі складними формами або динамічним контентом.

Сучасною альтернативою є декларативний стиль, який у Kotlin реалізується через Jetpack Compose. У цьому підході інтерфейс описується як функція від стану, а кожна зміна стану автоматично призводить до оновлення UI. Це суттєво спрощує логіку керування екраном, усуває потребу в ручному оновленні елементів та забезпечує передбачувану реакцію інтерфейсу на події користувача.

Крім того, декларативний підхід сприяє використанню сучасних архітектур/паттернів, таких як MVI (Model–View–Intent), де логіка застосунку організована через потоки подій, а View є "підписаним" спостерігачем на зміну стану. Kotlin забезпечує всі необхідні інструменти для реалізації цього патерну — зокрема, за допомогою StateFlow,

SharedFlow і collectAsState() у Compose.

Етапи проєктування мобільного застосунку

Проєктування мобільного застосунку є поетапним процесом, що включає як аналітичну, так і технічну підготовку до розробки. Якісне проєктування дозволяє уникнути архітектурних помилок, логічних суперечностей та забезпечити ефективну реалізацію майбутнього програмного продукту. Особливої важливості набуває чітке проходження ключових етапів — від вивчення потреб користувача до створення прототипів і вибору технічного стеку.

Початковим етапом проєктування будь-якого застосунку є аналіз майбутніх користувачів. Визначення цільової аудиторії дозволяє сформувати чітке розуміння функціональності, інтерфейсних рішень і технічних вимог. У випадку дипломного проєкту основними користувачами є студенти навчального закладу, які потребують простого та зручного інструменту для отримання оголошень, спілкування у межах навчального середовища та доступу до базової інформації про курси чи події.

Цей етап передбачає формування користувацьких сценаріїв (user stories): як саме користувач взаємодіятиме із застосунком, у яких контекстах (у дорозі, між парами, вдома), і якими функціями користуватиметься найчастіше. Розуміння цих потреб закладає основу для формування вимог.

На основі потреб аудиторії формуються функціональні вимоги — перелік можливостей, які має реалізовувати застосунок. До прикладу, функції перегляду оголошень, авторизації, відправки повідомлень, отримання push-сповіщень, оновлення профілю тощо. Кожна з цих вимог описується у вигляді задач, що лягають в основу майбутньої реалізації.

Паралельно з цим формуються нефункціональні вимоги, які стосуються не поведінки, а якості застосунку. Це, зокрема:

Продуктивність — швидке завантаження контенту, плавність інтерфейсу;

Надійність — стійкість до обриву інтернет-з’єднання;

Безпека — захист персональних даних користувачів;

Доступність — коректне відображення на різних пристроях;

Масштабованість — можливість розширення функцій у майбутньому.

Правильна постановка вимог є критично важливою, адже вона визначає технічні

рішення та фокусує розробника на пріоритетних аспектах системи.

Наступним етапом є вибір архітектурного підходу та інструментів реалізації. У сучасній Android-розробці найчастіше використовуються архітектурні шаблони MVVM або MVI, які забезпечують розділення логіки інтерфейсу, стану та джерел даних. У проєкті було обрано MVI-архітектуру, що найкраще поєднується з декларативним підходом Jetpack Compose та Kotlin StateFlow.

Серед технологій, які було використано:

Kotlin — мова розробки з підтримкою корутин;

Jetpack Compose — інструмент побудови UI;

Firebase Firestore — хмарна база даних у режимі реального часу;

StateFlow та coroutines — реактивна логіка управління станом;

Hilt — ін’єкція залежностей.

Також на цьому етапі визначається структура проєкту — поділ на модулі, пакунки, компоненти. Це дозволяє закласти основу для масштабованості, тестованості та легшої підтримки застосунку.

Завершальним етапом проєктування мобільного застосунку є створення макетів інтерфейсу користувача, які відображають, як виглядатимуть основні екрани застосунку та як користувач буде з ними взаємодіяти. Цей етап часто реалізується у вигляді графічних прототипів, створених у спеціалізованих інструментах, таких як Figma, або безпосередньо у середовищі розробки за допомогою Jetpack Compose, де створюються первинні версії екранів із демонстраційним станом. Такий підхід дозволяє не лише побачити загальну візуальну концепцію, а й одразу протестувати її на реальному пристрої. На рисунку 2.2 представлений вигляд прототипів екранів, які створені в Figma:

Рисунок 2.2 – Прототипи екранів в Figma

Одним із найважливіших завдань під час створення макетів є продумування логіки навігації між екранами. Користувач повинен мати змогу легко переходити з одного розділу застосунку до іншого — наприклад, з головного екрану до переліку оголошень, чату чи профілю. При цьому важливо забезпечити зручність повернення до попереднього стану, уникати надлишкових дій і забезпечити логічну послідовність переходів. У випадку Jetpack Compose, навігація реалізовується через бібліотеку Navigation Compose, яка дозволяє гнучко керувати маршрутами, параметрами екранів і back stack-ом.

Також під час побудови макетів визначається розташування елементів інтерфейсу — кнопок, текстових полів, списків, меню тощо. Це розташування має відповідати логіці використання: найбільш важливі дії повинні бути розташовані у зоні зручного доступу (наприклад, у нижній частині екрана для смартфонів), а вторинні функції — винесені в окремі панелі або випадаючі меню. Вдале розміщення елементів не лише покращує зовнішній вигляд застосунку, а й сприяє підвищенню ефективності взаємодії користувача з програмою.

Не менш важливою є відповідність інтерфейсу принципам Material Design, які диктують сучасні підходи до візуального оформлення, анімацій, типографіки та кольорової палітри. Material Design не лише формує стиль, а й визначає правила використання компонентів, їх поведінку у відповідь на дії користувача, відступи, тіні, закруглення тощо. Дотримання цих рекомендацій забезпечує послідовність в інтерфейсі, знижує когнітивне навантаження та робить взаємодію більш передбачуваною.

Особливу увагу під час проєктування слід приділяти адаптації інтерфейсу до різних розмірів екранів. Android-пристрої бувають дуже різноманітними — від невеликих бюджетних смартфонів до планшетів із високою щільністю пікселів. Тому макети повинні коректно масштабуватися, підтримувати відносні розміри компонентів, використовувати векторну графіку та масштабовану типографіку. У Jetpack Compose адаптивність досягається завдяки модифікаторам типу Modifier.fillMaxWidth(), BoxWithConstraints, а також через динамічне обчислення розмірів залежно від конфігурації екрана.

Нарешті, ключовим аспектом є взаємодія користувача з інтерфейсом — те, як він натискає кнопки, вводить текст, перегортає списки, отримує візуальний зворотний зв’язок. Макети мають передбачати, як елементи реагуватимуть на дії користувача, які анімації будуть застосовані, як виглядатиме завантаження контенту чи обробка помилок. Завдяки можливостям Jetpack Compose, подібна взаємодія описується декларативно, через зміну стану, що дозволяє зробити її послідовною, передбачуваною та легкою у підтримці.

На рисунку 2.3 представлено готовий створений дизайн на основі прототипів та врахування Material3 принципів:

Рисунок 2.3 – Дизайн сторони студента

Таким чином, побудова макетів інтерфейсу є завершальним, але дуже важливим етапом проєктування мобільного застосунку. Вона перетворює абстрактні вимоги на візуальну та інтерактивну форму, що дозволяє протестувати логіку до початку реалізації й забезпечити позитивний користувацький досвід.

Проєктування архітектури застосунку за допомогою Kotlin

Якісна архітектура є основою будь-якого масштабованого та підтримуваного програмного забезпечення. У мобільній розробці, зокрема на платформі Android, архітектура визначає не лише внутрішню структуру застосунку, а й спосіб взаємодії компонентів, реакцію на події користувача, обробку даних та підтримку життєвого циклу екранів. Мова Kotlin, завдяки своїм сучасним можливостям, дозволяє реалізувати гнучку, чисту й ефективну архітектуру, яка відповідає потребам як невеликих, так і великих проєктів.

Проєктування застосунку передбачає поділ коду на логічні шари, 18 кожен з яких відповідає за свою частину відповідальності. Типова архітектура, яку підтримують більшість сучасних Android-проєктів, включає такі рівні:

UI-шар відповідає за відображення стану інтерфейсу. У випадку з Jetpack Compose він реалізується у вигляді Composable-функцій, які отримують стан і рендерять відповідний вигляд. Цей шар не містить бізнес-логіки і не працює безпосередньо з джерелами даних.

ViewModel-шар є посередником між інтерфейсом і бізнес-логікою. ViewModel зберігає стан екрана, обробляє події від користувача, викликає відповідні дії у бізнес-шарі та оновлює UI-стан. Саме тут переважно реалізується логіка MVI (Model–View–Intent) у Kotlin-застосунках.

Domain-шар (не завжди виділяється окремо у невеликих проєктах) включає бізнес-правила, інтерфейси репозиторіїв, use case-и — тобто, логіку, яка не залежить від джерела даних чи UI.

Data-шар реалізує доступ до зовнішніх джерел інформації, таких як Firebase Firestore, локальні бази даних (Room), сховища налаштувань (DataStore), REST-API тощо. Тут розміщується вся логіка зчитування, запису та обробки помилок, а також мапінг DTO-моделей у domain-моделі. На рисунку 2.4 представлено схематичне представлення архітектури:

2.3 – Схематичне представлення архітектури застосунку

Такий розподіл дозволяє досягти розділення обов’язків, що спрощує тестування, повторне використання коду і забезпечує хорошу масштабованість проєкту.

У середовищі Android розробники традиційно використовують шаблон MVVM (Model–View–ViewModel), що був рекомендований Google у складі бібліотеки Android Jetpack. Цей патерн передбачає, що ViewModel є джерелом даних для View і не має прямої залежності від UI-компонентів. У Kotlin-проєктах MVVM реалізується легко і

природно, з використанням LiveData або StateFlow, що дозволяє View реагувати на зміни без потреби вручну оновлювати інтерфейс.

Разом із поширенням Jetpack Compose дедалі більше проєктів переходить до MVI (Model–View–Intent) — декларативного архітектурного патерну, який передбачає односпрямований потік даних. У цій моделі View надсилає Intent (події користувача), ViewModel перетворює їх на новий стан (State), і цей стан передається назад у View для візуалізації. MVI дозволяє реалізувати повністю передбачувану логіку, де будь-який стан є результатом певної події, а інтерфейс ніколи не залежить від сторонніх змін. Для Kotlin ця модель є надзвичайно зручною завдяки наявності sealed-класів, data-класів і підтримці Flow.

Окремо варто відзначити роль Kotlin Coroutines у сучасному підході до реалізації логіки Android-застосунку. Coroutines надають зручний і легкий для читання спосіб обробки асинхронних операцій — таких як запити до Firestore, завантаження профілю користувача, читання даних із DataStore тощо. Замість callback-підходу, який був типовим у Java, coroutines дозволяють писати лінійний, послідовний код у ViewModel чи репозиторіях.

У зв’язці з coroutine використовуються Flow — потоки даних, які можуть емінувати значення з часом. Для цілей відображення стану в UI найчастіше використовується StateFlow — спеціалізований тип Flow, який завжди зберігає останнє значення. Це дозволяє ViewModel створювати публічний стан (val uiState: StateFlow<UiState>) і оновлювати його через update {} або emit() при зміні даних чи подій.

Завдяки такій зв’язці (coroutines + StateFlow), архітектура застосунку стає реактивною, передбачуваною та стабільною. Jetpack Compose безпосередньо підтримує StateFlow через функцію collectAsState(), що дозволяє автоматично оновлювати інтерфейс при кожній зміні стану — без зайвого boilerplate-коду або необхідності вручну відстежувати зміни. На рисунку 2.4 представлена схематична робота Kotlin Flow:

Рисунок 2.4 – Схематичне прадставлення моделі роботи Kotlin Flow

Інструменти та бібліотеки Kotlin для проєктування застосунків

Ефективна розробка мобільних застосунків потребує використання не лише мови програмування, а й відповідних інструментів та бібліотек, які забезпечують зручність побудови інтерфейсу, керування залежностями, навігацію між екранами та роботу з даними. Мова Kotlin має глибоку інтеграцію з екосистемою Android, а також підтримує сучасні інструменти Google і JetBrains, що 8 дозволяє значно підвищити продуктивність та якість розробки.

Jetpack Compose — це сучасний декларативний фреймворк для побудови інтерфейсу користувача на Android, створений з нуля з урахуванням особливостей Kotlin. На відміну від класичного підходу на основі XML, Compose дозволяє описувати UI безпосередньо у коді за допомогою функцій @Composable, які реагують на зміну стану.

Перевагою Compose є висока гнучкість, можливість створення повторно використовуваних компонентів, простота анімацій та повна інтеграція з State і StateFlow. Компоненти автоматично оновлюються при зміні стану, що спрощує підтримку та тестування. Compose ідеально поєднується з патернами MVVM і MVI та забезпечує високу швидкість розробки навіть для складних екранів.

Керування залежностями є невід'ємною частиною архітектурного проєктування. У Kotlin-застосунках найчастіше використовуються фреймворки Koin або Hilt. Обидва дозволяють реалізувати ін'єкцію залежностей (Dependency Injection) — тобто автоматичне надання об'єктів у компоненти, що їх використовують.

Koin — це легкий DI-фреймворк, що не потребує генерації коду й легко інтегрується з Kotlin DSL. Його перевага — швидке налаштування та мінімалізм.

Hilt — фреймворк на базі Dagger, офіційно рекомендований Google для Android. Він інтегрується з життєвим циклом компонентів Android, автоматично створює граф залежностей та добре працює у великих проєктах.

Обидва підходи дозволяють уникнути ручного створення об'єктів, спростити тестування та реалізувати слабке зв'язування між модулями застосунку.

У Jetpack Compose навігація реалізується через бібліотеку Navigation Compose, яка є частиною Jetpack Navigation. Вона дозволяє створювати маршрути між екранами (composable) та керувати переходами за допомогою NavController.

Navigation Compose підтримує передачу параметрів між екранами, контроль back stack-у, обробку deep link-ів та інші стандартні сценарії. Бібліотека тісно інтегрується з ViewModel і StateFlow, що дозволяє управляти станом екрану під час переходів. У рамках застосунку це забезпечує чисту архітектуру та зменшує кількість boilerplate-коду порівняно з класичними Fragment-транзакціями.

У сучасних Android-застосунках дані можуть зберігатися як локально, так і в хмарі. Для локального зберігання найчастіше використовується бібліотека Room, яка є обгорткою над SQLite. Вона забезпечує типобезпечний доступ до даних, генерацію DAO-інтерфейсів, підтримку міграцій та роботу з Flow/LiveData. Room ідеально підходить для кешування інформації або роботи з локальними таблицями.

Для хмарного зберігання даних часто використовується Firebase Firestore — NoSQL-база

даних із підтримкою синхронізації 8 в реальному часі та офлайн-режиму. У Kotlin-проєктах Firestore працює разом із coroutines (.await()) та дозволяє інтегруватися з StateFlow, щоб автоматично оновлювати інтерфейс при зміні даних. Це особливо важливо у чатах, оголошеннях та будь-яких сценаріях, де важлива миттєва реакція на оновлення.

Таким чином, поєднання Kotlin з такими інструментами, як Jetpack Compose, Hilt/Koin, Navigation Compose, Room і Firebase, забезпечує повний стек для ефективного проєктування та реалізації Android-застосунку. Використання цих бібліотек дозволяє дотримуватися принципів чистої архітектури, забезпечити масштабованість, адаптивність та високу продуктивність мобільного застосунку.

Кращі практики проєктування Android-застосунків на Kotlin

Після вибору архітектурного підходу, інструментів і структури проєкту постає питання про дотримання кращих практик, які забезпечують стабільність, зрозумілість і підтримуваність коду. Kotlin надає розробникам широкий набір інструментів, які дозволяють реалізовувати ці практики ефективно та природно, але лише за умови правильного застосування підходів до організації логіки, управління залежностями й структурування проєкту.

Одним із головних викликів у проєктуванні є уникнення антипатернів, які виникають при порушенні розмежування відповідальностей між компонентами. Наприклад, поява так званих "God Object" — надмірно універсальних класів, які об’єднують UI, бізнес-логіку, доступ до даних і реагування на події — призводить до того, що такі класи важко тестувати, масштабувати або модифікувати без ризику зламати функціональність. У контексті Android це часто трапляється з ViewModel або Fragment, якщо вони містять занадто багато логіки.

Ще один поширений антипатерн — це "Logic in View", коли ключова логіка, наприклад перевірка введення або обробка відповіді з сервера, реалізується безпосередньо в UI-компонентах. Такий підхід суперечить ідеології сучасних архітектурних патернів і ускладнює тестування. Правильна практика передбачає винесення логіки у ViewModel або відповідні UseCase-и, залишаючи Composable-функціям лише функцію візуалізації.

Крім дотримання принципів розмежування відповідальностей, надзвичайно важливими є розширюваність, тестованість і підтримка коду. Розширюваність забезпечується через чітку декомпозицію застосунку на незалежні фічі та модулі, використання інтерфейсів і принципів інверсії залежностей. Тестованість — через мінімізацію побічних ефектів у ViewModel, використання Mock- або Fake-репозиторіїв і спрощення логіки до окремих функцій із передбачуваною поведінкою. А підтримка коду

залежить від чистоти структури, консистентності найменувань, наявності внутрішньої документації та прозорої логіки розподілу обов'язків.

У Kotlin-проєктах важливо правильно організовувати файлову структуру, яка має бути логічно зрозумілою для всієї команди або майбутніх розробників. Найбільш поширеним є функціональний підхід (за фічами): коли структура поділяється за екранами або модулями, на рисунку 2.5 представлено приклад організації структури модулів:

Рисунок 2.5 –Модель організації структури модулів

Такий підхід дозволяє чітко розділяти відповідальність, забезпечити ізольованість функціональності та спрощує інтеграцію нових фіч. У великих проєктах застосовується багатомодульна архітектура, де кожен модуль є незалежним артефактом (наприклад, core, auth, home), що дозволяє пришвидшити збірку та зменшити час компіляції.

РЕАЛІЗАЦІЯ ПРОЄКТУ

Постановка завдання та технічні вимоги

У межах дипломного проєкту було поставлено завдання розробити мобільний застосунок для внутрішньої комунікації студентів Фахового коледжу інформаційних технологій (ФКІТ). Метою створення застосунку є цифровізація навчального процесу, спрощення доступу студентів до важливої академічної інформації, підвищення ефективності взаємодії між студентами та викладачами, а також підтримка інклюзивності навчання.

Запропонований мобільний застосунок має стати сучасною, зручною та доступною альтернативою традиційним каналам комунікації, таким як електронна пошта, групи в месенджерах або сторонні сервіси на кшталт Google Classroom. Однією з основних вимог до застосунку є забезпечення легкості використання — інтерфейс має бути інтуїтивно зрозумілим для студентів різного віку і рівня цифрової грамотності. Важливим акцентом стала також доступність застосунку для різних категорій користувачів, включаючи підтримку масштабування шрифтів, адаптацію під різні розміри екранів, можливість роботи при слабкому або нестабільному інтернет-з'єднанні.

Крім того, проєкт передбачає підтримку офлайн-можливостей: застосунок має працювати навіть без активного підключення до мережі, зберігаючи локальні копії розкладу, курсів і завдань, а при відновленні з'єднання автоматично синхронізувати зміни. Це забезпечить безперервний доступ до навчальної інформації в будь-яких умовах.

Застосунок має не лише полегшити організацію навчального процесу для студентів, а й сприяти загальній цифровій трансформації коледжу, підтримуючи новітні стандарти у сфері освіти та комунікацій.

На момент розробки дипломного проєкту реалізовано функціонал для студентської ролі, який включає наступні можливості:

Перегляд журналу оцінок: студент має змогу в будь-який момент переглянути свої поточні академічні досягнення за курсами та окремими завданнями;

Нагадування про дедлайни: окремий екран нагадувань інформує студента про наближення термінів здачі матеріалів, контрольних завдань або інших важливих подій;

Перегляд новин та івентів коледжу: студенти можуть швидко ознайомитися з актуальними подіями та новинами, при потребі переходячи на офіційний сайт ФКІТ для детальнішої інформації;

Список курсів: у застосунку відображається перелік доступних студенту навчальних курсів;

Доступ до курсів і матеріалів: у межах кожного курсу студент має можливість переглядати доступні матеріали, виконувати завдання, надсилати відповіді на перевірку та вести діалог із викладачем у межах курсу;

Перегляд розкладу занять: студент отримує динамічно оновлюваний розклад занять, який адаптується залежно від змін в академічному календарі;

Профіль користувача: студенти мають змогу редагувати своє біо, змінювати фото профілю та переглядати персональні дані, пов'язані з їхнім навчанням.

Оскільки навчальний процес потребує постійної актуальності даних, застосунок має підтримувати роботу 8 в реальному часі. Це означає, що зміни в курсах, дедлайнах, оцінках або розкладі повинні миттєво відображатися на пристроях студентів. Для реалізації цієї функціональності було використано можливості Firebase Firestore, який забезпечує синхронізацію даних із сервером без необхідності ручного оновлення.

Водночас, для забезпечення безперебійного доступу до навчальної інформації навіть у випадку відсутності інтернет-з'єднання, реалізовано підтримку офлайн-режиму. Застосунок автоматично зберігає локальні копії необхідних даних, дозволяючи студенту переглядати розклад, курси, дедлайни та оцінки без підключення до мережі. Після відновлення інтернет-з'єднання всі зміни синхронізуються з сервером. Такий підхід гарантує, що користувач завжди матиме доступ до актуальної інформації незалежно від технічних обставин, що є критичним для навчального середовища.

Структура проєкту та організація коду

Архітектура мобільного застосунку реалізована з урахуванням принципів модульності, чистої архітектури та сучасних патернів Android-розробки. Це забезпечує гнучкість, масштабованість, зручність підтримки та розширення функціоналу.

Основою проєкту є головний модуль app, у якому відбувається складання застосунку та ініціалізація всіх функціональних модулів. Всі основні можливості розділено на окремі feature-модулі, кожен з яких відповідає за конкретну частину функціоналу:

assignment-details — модуль для роботи з деталями завдань;

auth — автентифікація користувачів;

course-materials та courses — робота з курсами та матеріалами;

profile — управління профілем користувача;

semester та time-table — модулі для роботи з розкладом і семестрами;

files — обробка файлів (завантаження, видалення, надсилання);

student — студентський функціонал та екрани;

user — управління користувачами, розподіл ролей;

main — головні екрани для студентів, система визначає роль і показує відповідну навігацію.

На рисунку 3.1 представлено схему взаємодії модулів:

Рисунок 3.1 – Схема взаємодії модулів

Для забезпечення перевикористовуваності компонентів і спрощення UI-розробки створено окремий модуль common-ui, який містить спільні компоненти інтерфейсу та тему оформлення застосунку.

Модуль common включає спільні специфікації, data sources та логіку роботи з Firestore й іншими зовнішніми даними.

Також виділено окремий модуль files для роботи з файлами та user для централізованої роботи з користувачами.

В межах кожного feature-модуля застосовано шарову архітектуру:

data — джерела даних (Firestore, локальні джерела, репозиторії);

domain — бізнес-логіка та моделі;

presentation — екрани (UI), ViewModel, UI-моделі;

di — залежності та їх ініціалізація (Hilt/Koin).

Такий підхід забезпечує чітке розділення відповідальностей, спрощує тестування й обслуговування проєкту. Екрани реалізовані за допомогою Jetpack Compose, що дозволяє декларативно описувати інтерфейс та гнучко реагувати на зміни стану. ViewModel відповідає за підготовку даних для відображення та обробку подій від користувача. Репозиторії є єдиним джерелом даних для доменного шару, інкапсулюючи доступ до Firestore та інших сервісів.

У проєкті комбінуються два архітектурних патерни: MVVM та MVI.

MVVM використовується на більшості екранів, особливо там, де потрібна робота зі списками, фільтрацією та пошуком. Цей підхід забезпечує простоту реалізації та підтримки таких сценаріїв.

MVI (Model-View-Intent) застосовано в тих частинах застосунку, де важливо забезпечити єдиний джерело стану (Single Source of Truth) і уніфіковане управління подіями. Зокрема, MVI використовується в тих екранах, де є складна бізнес-логіка, багатоступеневі стани або необхідність чітко обробляти наміри користувача.

В основі MVI-реалізації лежить окремий модуль mvi, який містить загальні класи для роботи з інтенціями, станами та редьюсерами. Це дозволяє уникнути дублювання коду та забезпечує однаковий підхід до побудови UI в усіх частинах застосунку.

Для керування асинхронними потоками використовується Kotlin Coroutines та StateFlow, що дозволяє ефективно оновлювати стан інтерфейсу в режимі реального часу та оптимально працювати з Firestore.

Така організація структури та коду дозволяє забезпечити чисту архітектуру застосунку, полегшує підтримку та додавання нових можливостей. Використання сучасних підходів (MVVM + MVI, StateFlow, Jetpack Compose) та розподіл коду на модулі забезпечують гнучкість, стабільність і готовність до масштабування в майбутньому.

Реалізація інтерфейсу користувача в Jetpack Compose

Враховуючи сучасні тенденції Android-розробки, інтерфейс користувача мобільного застосунку реалізовано за допомогою Jetpack Compose — декларативного UI-фреймворку, що дозволяє спростити побудову інтерфейсів і забезпечити гнучке керування станами.

Щоб уникнути дублювання коду та забезпечити уніфікований вигляд елементів інтерфейсу, всі основні UI-компоненти були винесені до окремого модуля common-ui. У цьому модулі реалізовано:

Кнопки (Button, IconButton) із загальним стилем, що відповідає кольоровій палітрі та типографіці застосунку;

Поля введення (TextField, OutlinedTextField) з підтримкою валідації та відображення помилок;

Базові вью-компоненти (наприклад, Card, ListItem), які використовуються для створення списків курсів, оцінок, дедлайнів тощо;

Тему застосунку, яка включає колірні схеми для світлої теми, розміри шрифтів та відступів.

Цей підхід дозволив забезпечити стилістичну єдність всього застосунку та спростити підтримку UI при майбутніх змінах дизайну.

Інтерфейс користувача мобільного застосунку було спроєктовано так, щоб забезпечити максимально зрозумілу навігацію та доступ до основних функцій для студентів. Основними екранами є головна сторінка, екран курсів, екран розкладу та профіль користувача. Головна сторінка відображає коротку інформацію про студента, поточні дедлайни, журнал оцінок та новини ФКІТ. Екран курсів надає можливість переглядати всі доступні навчальні курси, ознайомлюватися з матеріалами та дедлайнами, що дозволяє студенту планувати навчальний процес. Екран розкладу інтегрує динамічний розклад, який автоматично оновлюється з бази даних Firestore, забезпечуючи актуальність інформації. У профілі користувача реалізовано функціонал зміни особистої інформації, такої, як біографії та фотографії, а також перегляд основних даних акаунту.

Усі екрани побудовано за принципом presentation layer із чітким розподілом відповідальностей. Компоненти інтерфейсу відповідають виключно за відображення даних і реагування на події користувача, тоді як логіка обробки станів та взаємодії з даними винесена у ViewModel та інші архітектурні шари. Для забезпечення реактивності інтерфейсу у проєкті використано підхід із використанням StateFlow у ViewModel. Це дозволяє користувацькому інтерфейсу автоматично оновлюватися при зміні стану без необхідності додаткового ручного втручання з боку розробника. Наприклад, дані про курси або розклад підписані на StateFlow у ViewModel. При оновленні інформації у Firestore відповідний потік даних у Data Layer транслює зміни до Domain Layer, а ViewModel приймає новий стан і оновлює StateFlow. У свою чергу, ці зміни автоматично відображаються у користувацькому інтерфейсі. Такий підхід дозволив уникнути імперативного управління станами і мінімізувати ризик виникнення

помилок, пов’язаних із відображенням неактуальної інформації.

Особливу увагу було приділено реалізації адаптивності інтерфейсу. Застосунок підтримує адаптивний дизайн, що забезпечує коректне відображення на пристроях із різними розмірами екранів та щільністю пікселів. Для цього використовувалися засоби Jetpack Compose, зокрема Modifier.fillMaxWidth(), BoxWithConstraints та інші API, які дозволяють гнучко масштабувати компоненти. Графічні елементи реалізовано за допомогою векторної графіки, що дає змогу зберігати високу якість зображень на будь-якому розширенні екрана. Передбачено підтримку змін масштабування шрифтів користувачем, що відповідає сучасним стандартам доступності (Accessibility). За необхідності застосовано механізм WindowSizeClass для оптимізації відображення на планшетах та складних (foldable) пристроях. Крім того, інтерфейс сумісний із темною темою та висококонтрастним режимом, що підвищує інклюзивність і забезпечує зручність користування для людей із вадами зору.

Комплексне поєднання цих підходів дало змогу створити інтерфейс, який є сучасним, гнучким, зручним у користуванні та доступним для широкого кола користувачів.

Реалізація навігації та управління станами

У мобільному застосунку для внутрішньої комунікації студентів ФКІТ важливу роль відіграє зручна навігація між екранами та ефективне управління станами. Для реалізації маршрутизації між екранами було використано Navigation Compose — сучасний компонент Android Jetpack, який дозволяє створювати граф переходів у декларативному стилі без потреби використання XML.

Налаштування Navigation Compose передбачало створення центрального навігаційного графа, у якому визначено всі основні маршрути: головна сторінка, курси, розклад, профіль, а також допоміжні екрани, наприклад, екран деталей курсу або деталей завдання. Навігаційний контролер (NavController) було інтегровано в основну активність застосунку, що дозволило управляти переходами у відповідності до бізнес-логіки та ролі користувача.

Для управління станами застосовано комбінацію патернів MVVM та MVI. Екрани, що відображають списки та мають прості сценарії взаємодії, реалізовані з використанням MVVM, де ViewModel виступає посередником між UI та бізнес-логікою. Для екранів із більш складною взаємодією, які потребують обробки різних станів і подій, застосовано патерн MVI. У цьому патерні кожна подія користувача (Intent) обробляється та трансформується у відповідний стан (State) або побічний ефект (Effect), наприклад, відображення повідомлення про помилку або успішне виконання дії.

Особливу увагу приділено управлінню стандартними сценаріями станів, такими як

завантаження даних, порожні стани та обробка помилок. Для відображення процесу завантаження на UI було створено спільні компоненти у модулі common-ui, що демонструють прогрес-індикатор або повідомлення про відсутність даних. Помилки, які виникають під час взаємодії з Firestore чи мережею, обробляються на рівні ViewModel і транслюються у вигляді ефектів до UI для відображення сповіщень або діалогів з поясненнями.

Управління потоками даних здійснюється за допомогою StateFlow і SharedFlow, що дозволяє ефективно оновлювати стани в інтерфейсі та реагувати на одноразові події без ризику дублювання або втрати інформації. Такий підхід забезпечив стабільну роботу застосунку навіть у випадках повільного інтернет-з'єднання або тимчасової відсутності мережі.

Інтеграція з Firebase Firestore

Для зберігання, обробки та синхронізації даних у мобільному застосунку використано хмарну NoSQL базу даних Firebase Firestore. Це рішення обрано через його простоту інтеграції, підтримку роботи у реальному часі, офлайн-синхронізацію та масштабованість під майбутні потреби проєкту.

Структура Firestore організована за принципом колекцій і документів, що забезпечує гнучкість та дозволяє легко масштабувати дані. Основними колекціями в базі є:

courses — містить інформацію про всі курси, доступні студентам. Усередині курсів можуть зберігатися підколекції матеріалів, які потрібні в рамках навчального процесу;

files — колекція для зберігання метаданих файлів, що використовуються в курсах або завданнях (посилання на Cloud Storage);

group — інформація про навчальні групи, яка допомагає формувати розклади та доступність курсів для конкретних студентів;

students — дані студентів, включаючи їхні оцінки, персональну інформацію;

teachers — профілі викладачів та дані для ідентифікації їхніх курсів;

users — базова інформація для авторизації та профілів користувачів

17 Така структура дозволяє чітко розподілити обов'язки між модулями застосунку та спростити побудову запитів для отримання необхідної інформації. На рисунку 3.2 представлено схематичне зображення бази даних:

Рисунок 3.2 – схематичне зображення бази даних

Використовуючи Firestore SDK для Android, застосунок реалізує отримання та збереження даних у реальному часі. Для більшості колекцій (курси, студенти, дедлайни) підключено слухачі змін (snapshot listeners), які дозволяють автоматично оновлювати UI при будь-яких змінах у базі даних без потреби ручного оновлення. Це особливо важливо для таких сценаріїв, як оновлення розкладу, додавання нових матеріалів або змінення дедлайнів.

Усі запити виконуються в межах Data Layer через спеціалізовані дата-сорси, які інкапсулюють доступ до Firestore. Ці дата-сорси передають дані в Domain Layer, де бізнес-логіка обробляє їх перед передачею в ViewModel та відображенням у UI.

Firestore автоматично підтримує офлайн-синхронізацію, що є критично важливою для студентів, які можуть тимчасово втрачати підключення до Інтернету (наприклад, перебуваючи в транспорті чи місцях зі слабким сигналом). Навіть у разі відсутності мережі застосунок продовжує відображати останні доступні дані з локального кешу. При відновленні з’єднання всі зміни автоматично синхронізуються із сервером.

Для більшої стабільності в Data Layer додатково реалізовано обробку помилок при втраті з’єднання та сповіщення користувача у випадках, коли певна інформація не може бути оновлена або отримана.

Інтеграція Firestore дозволила реалізувати надійну та масштабовану модель роботи з даними, яка відповідає сучасним вимогам до мобільних застосунків і забезпечує якісний користувацький досвід навіть у складних мережевих умовах.

Функціональність для викладача

У рамках дипломного проєкту було реалізовано повноцінну функціональність для викладачів, яка забезпечує цифровізацію основних процесів навчання, взаємодії зі студентами, оцінювання та керування освітнім контентом. Вся логіка побудована відповідно до архітектурної моделі MVVM (Model–View–ViewModel), що дозволяє чітко відокремити представлення, логіку і доступ до даних.

Основні функціональні можливості викладача:

Перегляд і редагування профілю. Кожен викладач має власний профіль, у якому відображається ім’я, фото, електронна пошта, біографічна інформація (біо). Застосунок дозволяє змінювати фото профілю (з вибором із галереї) та біографію. Ці зміни зберігаються у Firestore, а зображення — у Firebase Storage. ViewModel профілю працює з UserRepository(TeacherRepository) і застосовує специфікації для оновлення даних користувача.

Управління курсами та матеріалами. Викладач має змогу створювати нові курси: вказувати назву, добавляти групи, добавляти викладачів, додавати обкладинку. Реалізована можливість редагувати і видаляти власні курси. Для кожного курсу можна створювати навчальні матеріали — лекції, лабораторні, контрольні, домашні/практичні завдання. У матеріалах задається тема, опис, дедлайн, максимальна оцінка, а також можуть бути прикріплені файли. Дані синхронізуються з Firestore і обробляються через MaterialRepository та відповідні специфікації

Оцінювання та чат зі студентами. На основі створених матеріалів викладач може виставляти оцінки для кожного студента окремо. Інтерфейс для оцінювання реалізовано у вигляді електронного журналу: викладач бачить список студентів, матеріал і вводить бали та коментарі. Дані зберігаються у Firestore в структурі students/{studentId}/results/{materialId}. Також реалізовано можливість спілкування зі студентами через чат: як у форматі «викладач студент», так і «студент викладач»,. Використано Firestore та GroupRepository.

Перегляд журналу оцінок по групах. Викладач має доступ до перегляду повного журналу оцінок за кожною групою: список студентів, список матеріалів, оцінки, максимальні бали. Реалізовано табличне представлення у форматі матеріал - студент, що дозволяє візуально оцінити успішність групи загалом. ViewModel об’єднує дані з кількох джерел: студентів групи, матеріалів курсу та відповідних оцінок.

Перегляд розкладу. Застосунок підтримує відображення актуального розкладу занять для викладача. Дані отримуються з Firestore через GroupRepository та ObserveGroupSemesterSessionSpec, з урахуванням поточного семестру та прив’язаної групи. Інтерфейс реалізовано з можливістю перемикання між місяцями, перегляду пар на день, та швидкого доступу до деталей заняття.

Аналіз досягнутих цілей і подальші кроки розвитку застосунку

У ході розробки мобільного застосунку для внутрішньої комунікації студентів ФКІТ було досягнуто основних поставлених цілей. Створено сучасний багатофункціональний застосунок, 20 який дозволяє студентам отримувати доступ до своїх курсів, переглядати навчальні матеріали, слідкувати за розкладом, отримувати нагадування про дедлайни, переглядати особистий профіль і оцінки. Архітектура застосунку була побудована із застосуванням сучасних підходів — розподілення на модулі, використання патернів MVVM і MVI, реактивне управління станами через StateFlow, а також використання декларативного підходу до побудови UI завдяки Jetpack Compose. Це забезпечило гнучкість, розширюваність та легкість супроводу коду.

Інтеграція з Firebase Firestore дала змогу організувати зберігання та обробку даних у реальному часі із підтримкою офлайн-синхронізації. Це гарантує стабільну роботу

застосунку навіть за умов нестабільного інтернет-з'єднання. Структура Firestore була спроєктована таким чином, щоб забезпечити гнучкість масштабування у разі розширення функціоналу застосунку у майбутньому.

Серед перспектив подальшого розвитку застосунку планується інтеграція push-сповіщень для оперативного інформування студентів про новини, дедлайни та зміни в розкладі. Крім того, можливе додавання аналітичного модуля, що дозволить студентам і викладачам відстежувати прогрес і успішність у навчанні.

Таким чином, розроблений мобільний застосунок не лише успішно реалізував поставлені задачі, але й заклав надійну основу для майбутнього вдосконалення та розширення функціональних можливостей відповідно до потреб навчального процесу та користувачів.

ВИСНОВКИ

Даний дипломний проєкт є комплексним дослідженням процесу розробки мобільного застосунку для внутрішньої комунікації в навчальному середовищі на прикладі ФКІТ. У роботі було детально розглянуто як теоретичні основи побудови сучасних Android-застосунків, так і практичні аспекти реалізації ключових функцій з використанням актуального технологічного стеку.

У теоретичній частині було проаналізовано особливості мобільної розробки для Android-платформи, переваги використання мови Kotlin, архітектурні підходи з різними паттернами (зокрема MVVM та MVI), а також можливості хмарної платформи Firebase, зокрема Firestore, як гнучкого та масштабованого інструменту для зберігання та синхронізації даних у реальному часі.

У практичній частині було реалізовано базовий функціонал мобільного застосунку, орієнтований на студентів. Серед основних можливостей: авторизація користувача, перегляд і отримання оголошень у режимі реального часу, зручна структура інтерфейсу та адаптивний дизайн, що відповідає сучасним вимогам до мобільних застосунків.

Використання Firebase Firestore забезпечило ефективне управління даними та спрощену синхронізацію між користувачами. Крім того, у роботі було враховано питання масштабованості, що дозволяє у майбутньому розширити функціонал застосунку — зокрема, реалізувати додаткові ролі користувачів, систему приватних повідомлень, push-сповіщення тощо.

Загалом, 23 результати дипломної роботи підтверджують доцільність використання Kotlin, Android SDK та Firebase як ефективного інструментарію для створення функціональних, надійних і масштабованих мобільних застосунків. Реалізований

застосунок може бути основою для подальшого розвитку цифрової інфраструктури ФКІТ, а також корисним прикладом для студентів, які цікавляться мобільною розробкою

Посилання

Це джерела виділених збігів у вашому документі. Кожен збіг позначено темно-зеленим числом, яке відповідає вказаному тут джерелу. Джерела впорядковані за схожістю — чим вищий бал, тим сильніше збіг.

#	Джерело	%
1	repository.hneu.edu.ua	0.6%
2	ela.kpi.ua	0.4%
3	er.nau.edu.ua	0.3%
4	it-college.khai.edu	0.3%
5	moodle2.snu.edu.ua	0.2%
6	ela.kpi.ua	0.2%
7	metod.vntu.edu.ua	0.1%
8	knsa.chdtu.edu.ua	0.1%
9	dspace.znu.edu.ua	0.1%
10	dspace.nuft.edu.ua	0.1%
11	confcontact.com	0.1%
12	repository.lnup.edu.ua	0.1%
13	ela.kpi.ua	0.1%
14	46.219.2.151	0.1%
15	itcollege.lviv.ua	0.1%
16	jarch.donnu.edu.ua	0.1%
17	ir.polissiauniver.edu.ua	0.1%
18	visnyk-juris.uzhnu.uz.ua	0.1%
19	eprints.library.odeku.edu.ua	0.1%
20	uk.aulapro.co	0.1%
21	blog.lebara.co.uk	0.1%
22	metod.vntu.edu.ua	0.1%
23	duikt.edu.ua	0.1%

#	Джерело	%
24	donnu.edu.ua	0.1%
25	dspace.nuft.edu.ua	0.1%

Plag

Дякуємо, що перевірили
свій документ за допомогою
Plag!