МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ, КОМП'ЮТЕРНОЇ
ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ
КАФЕДРА КОМП'ЮТЕРНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
_____ Аліна САВЧЕНКО
«___»_____2022 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ МАГІСТР
ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ
«ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ»

Тема: «Додатки на базі ОС Android з застосуванням сучасних архітектур розробки»

Виконавець: Михайло СЛОБОДЯНЮК

Керівник: к.т.н., доцент Сергій ВОДОП'ЯНОВ

Нормоконтролер: к.т.н., доцент Олена ТОЛСТІКОВА

КИЇВ 2022

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет *кібербезпеки, комп'ютерної та програмної інженерії*
Кафедра *Комп'ютерних інформаційних технологій*
Спеціальність *122 «Комп'ютерні науки»*
Освітньо-професійна програма *«Інформаційні технології проектування»*

ЗАТВЕРДЖУЮ:
завідувач кафедри КІТ
_____Аліна САВЧЕНКО
(підпис)
«____»_____2022 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи
Слободянюка Михайла Володимировича
(ПІБ випускника)

1. Тема роботи: «Додатки на базі ОС Android з застосуванням сучасних архітектур розробки» затверджена наказом ректора № 1774/ст від 28.09.2022р.
2. Термін виконання роботи: з 26 вересня 2022 року по 27 листопада 2022 року.
3. Вихідні дані до роботи: сучасні архітектурні підходи до розробки додатків на базі ОС Android.
4. Зміст пояснювальної записки: 1. Аналіз архітектурних рішень додатків ОС Android. 2. Проектування архітектури і роботи додатку. 3. Розробка та тестування додатка.
5. Перелік обов'язкового ілюстративного матеріалу: 1.Принципи SOLID. 2. Архітектурні рівні. 3. Детальний опис шарів архітектури Android. 4. Архітектурна структура модулів додатку. 5. Компоненти Android в рівневій архітектурі. 6. Структура реалізованого функціонального модуля. 7. Готовіі екрани функціонального модуля.

6. Календарний план-графік

№ з/п	Завдання	Термін виконання	Підпис керівника
1.	Підготовка необхідного матеріалу по темі для початку роботи. Формування літературних джерел та інформаційних ресурсів, щодо теми індивідуального завдання.	26.09.2022-08.10.2022	
2.	Написання 1 розділу. Представлення готового розділу керівнику	09.10.2022-18.10.2022	
3.	Розробка додатка за допомогою обраного стеку технологій.	19.10.2022-30.10.2022	
4.	Написання 2 розділу згідно виконаної роботи, представлення керівнику.	31.10.2022-06.11.2022	
5.	Написання 3 розділу, представлення керівнику	07.11.2022-11.11.2022	
6.	Форматування та оформлення пояснювальної записки, збір додаткової інформації.	12.11.2022-14.11.2022	
7	Проходження нормоконтролю, друк та перепліт пояснювальної записки.	14.11.2022 19.11.2022	
8	Розробка тексту доповіді. Оформлення графічного матеріалу для презентації	19.11.2022 22.11.2022	

7. Дата видачі завдання 26.09.2022р.

Керівник кваліфікаційної роботи _____ Сергій ВОДОП'ЯНОВ
(підпис керівника)

Завдання прийняв до виконання _____ Михайло СЛОБОДЯНЮК
(підпис випускника)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи на тему: «Додатки на базі ОС Android з застосуванням сучасних архітектур розробки» містить: 97 сторінок, 54 рисунки, 20 інформаційних джерел, 1 додаток.

Об'єкт дослідження – методи проектування та розробки додатків на базі ОС Android.

Предмет дослідження – ефективність сучасних архітектурних підходів до розробки та проектування Android додатків, їх оптимізоване поєднання

Мета кваліфікаційної роботи – створити Android додаток з найоптимальнішим підходом до розробки, що зробить її максимально швидкою, масштабованою та безпомилковою.

Методи дослідження – мова програмування Kotlin, Android studio**,** шаблони проектування, шаблони архітектур побудови Android додатків, Android OS, public API.

Матеріали магістерської роботи є наглядним прикладом оптимізованого підходу, який можна використовувати для розробки будь-якого Android додатку, оцінивши його масштабованість та складність підібрати найкращий підхід для розробки, або ж підхід з яким розроблявся проект у роботі, проетуванні додатку, вибору сучасних технологій для додатку, створення зручного додатку для роботи у команді.

ANDROID ДОДАТОК, АРХІТЕКТУРНІ РІШЕННЯ, ШАБЛОН ПРОЕКТУВАННЯ, ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ, KOTLIN, BUSINESS LOGIC, ТЕСТУВАННЯ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

JDK *(Java Development Kit,)*	Набір інструментів для розробці використовуючи Java Virtual Machine
JVM (*Java Virtual Machine*)	Віртуальна машина, яка дозволяє запускати програми, які компілюються у Java-bytecode
SDK *(Software development kit)*	Набір для розробки програмного забезпечення
IDE *(Integrated development environment)*	Інтегроване середовище розробки
API (Application programming interface)	Програмний інтерфейс
OS *(Operational System)*	Операційна система
MVP *(Model-View-Presenter)*	«Мінімальна» життєдіяльна архітектура додаткую.
YAGNI(You Are not going to need it)	Принцип проектування системи, в якому відсутнє закладання реалізацій на майбутнє.
UI *(User Interface)*	Інтерфейс користувача
DRY *(Don't Repeat Yourself)*	Принцип проектування системи, де за допомогою принципів ООП уникається дублікація кода
URL *(Uniform Resource Locator)*	Визначник місцезнаходження сайту в мережі Інтернет

ВСТУП

Ринок IT є одним з найбільших сегментів економіки світу, в якому існує величезний обіг грошей. Сучасний світ неможливо уявити без програмних продуктів, якими більшість користується щодня на свої телефонах. Частка користувачів що користуються Android телефонами приблизно 80%, а кількість додатків у офіційному магазині більше трьох мільйонів. Для того, щоб створювати нові продукти швидко та ефективно у великих командах, існують принципи проектування та архітектури Android додатків.

На основі пройденого шляху розробниками у проектуванні Android додатків було знайдено багато рішень популярних технічних проблем, що зробило значно зручнішим розробку, продукт створений використовуючи сучасні підходи буде досить оптимізованим та з мінімальними шансами недоліків використовуючи лише загальні правила і правильного дотримання циклу розробки!

Для того, щоб додатки успішно підтримувались та розвивались, їм потрібно зробити все, щоб привабити клієнта, та дуже часто, для того щоб аналізувати користувачів, та який він контент хоче бачити використовують штучний інтелект. Правильно спроектувавши систему, можна додавати будь-який функціонал швидко та без помилок, щоб залишити у користувача найкращий досвід, а бізнесу зберегти якомога більше грошей.

Проектування та підхід до розробки додатків на базі ОС Android є дуже актуальною темою, яка допомагає бізнесам розвиватись, користувачам залишатись задоволеними, а команді розробки робити все ефективно та злагоджено.

Актуальність теми кваліфікаційної роботи «Додатки на базі ОС Android з застосуванням сучасних архітектур розробки» є значною через актуальність ОС Android та стрімкий ріст як користувачів так і компаній, що починають або підтримують розробку програмних продуктів. У такому випадку саме швидкість, зручність, зрозумілість, масштабованість та тестування системи є основними критеріями розробки, та архітектурні підходи є критично важливими.

Метою кваліфікаційної роботи є розробка додатку з використанням сучасних підходів розробки та архітектурних рішень для ОС Android.

Відповідно до поставленої мети роботи визначено основні **завдання дослідження**:

– проаналізувати літературу, науково популярних статті, рішення проектів у відкритому доступі.

– обрати інструменти для реалізації архітектурних рішень.

– скласти повний цикл розробки програмного продукту та план реалізації його до першого релізу.

– розробити Android додаток, в якому буде реалізований деякий нативний функціонал, з можливістю розширення та додавання нового.

– провести тестування додатку на безпомилковість, описати принцип побудови та вибору архітектури та переваги які вона дає у розробці.

Наукова новизна роботи включає в себе дослідження ефективності правильного підходу до розробки програмного продукту, оцінка результатів використання архітектурних рішень та методів сучасного проектування кодової бази.

Через свою уніфікованість та актуальність, робота може використовуватись при проектуванні та побудові будь-яких додатків, як для бізнесу так і для розробки будь-яких корисних інструментів.

Розвитком застосунку є реалізація ще більшої кількості абстракцій та реалізація власних інструментів для написання програми та функціоналу швидко та без помилок. Також, можна будувати логіку, що буде відображати завжди правильно інтерфейс користувача на будь-яких розмірах екрану.

РОЗДІЛ 1
АНАЛІЗ АРХІТЕКТУРНИХ РІШЕНЬ ДОДАТКІВ ОС ANDROID

1.1. Структурування і кількість коду

Дейкстра зауважив, що розуміти програму цілком складно, а goto заважають розбивати її на дрібніші частини. При цьому прості керуючі конструкції – *if/else, do/while* – навпаки, полегшували роботу. З таких елементарних конструкцій, що управляють, може бути побудована будь-яка програма. [3]

– розуміти програму дуже складно, її треба розбивати;

– необдуманий прямий контроль (*goto*) – погано;

– тести допомагають знайти та довести, що баг є. Вони не доводять, що багів немає.

Структурне програмування дозволило розбивати програму на частини, які можна рекурсивно розбивати більш дрібні частини.

Також Дейкстра зазначив, що тести допомагають знайти та довести, що баг є. Але вони не доводять, що багів немає. У цьому програмування більше схоже фізику, ніж математику. Математика доводить, що твердження істинне, тоді як фізика цього зробити не може і намагається довести, що твердження хибне.

– знання одного компонента системи про інші має бути обмежене;

– межі повинні відокремлювати сутності, що мають значення (для бізнес-логіки) від тих, що не мають значення;

– основа – бізнес-логіка.

Слід уникати передчасних рішень. Рішення передчасне — якщо воно не стосується бізнес-вимог. Вибір бази даних, фреймворку, навіть мови програмування – рішення передчасні.

Межі повинні відокремлювати сутності, які мають значення (для бізнес-логіки) від тих, що не мають значення. Наприклад, бізнес-логіка має залежати ні від схеми БД, ні від мови запитів.

У хорошій архітектурі бізнес-логіка - це основа, а все інше: пристрої введення-виводу, БД і т. д. - плагіни до неї:

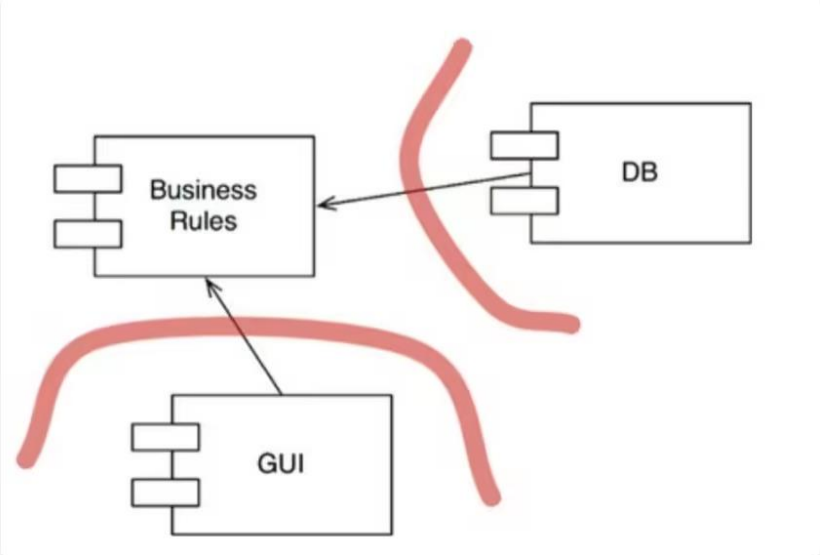


Рис. 1.1. Границі сутностів

1.2. Взаємовз'язок між класами і розуміння коду

Щоб визначити, до якого компоненту відноситься той чи інший клас, слід користуватися 3 принципами:

– принцип еквівалентності перевикористання і перевипуску;

– принцип загальної причини зміни;

– принцип спільного перевикористання.

Принцип еквівалентності перевикористання

Компонент не може включати просто набір класів та модулів, має бути спільна мета для цих класів та модулів. Усі класи та модулі одного компонента мають випускатися разом.

Принцип загальної причини зміни

В один компонент повинні включатися класи, які змінюються з однієї причини і одночасно. Для багатьох програм підтримуваність важливіша, ніж перевикористання. Якщо код повинен змінюватися, то буде зручніше, якщо зміна буде лише в одному компоненті, а не розмазано за програмою.

Принцип спільного перевикористання

У компоненті повинні міститися ті класи та модулі, які використовуються разом. Не примушуйте користувачів залежати від того, що вони не використовуватимуть.

Щоб визначити відносини між компонентами, слід користуватися 3 принципами:

– принцип ациклічності залежностей;

– принцип стабільних залежностей;

– принцип стабільності абстракцій.

Принцип ациклічності залежностей

Не допускайте зацикленості у графі залежностей компонента. Якщо в залежності є цикл, його можна розірвати одним з 2 способів:

– застосувати принцип інверсії залежностей;

– створити новий компонент, від якого залежатимуть компоненти, що викликають циклічність.

Принцип стабільних залежностей

Залежності мають бути спрямовані у бік стійкості. Деякі компоненти повинні бути змінені, щоб задовольняти зміни в бізнес-вимогах. І такі менш стабільні компоненти мають залежати від стабільніших.

Щоб дізнатися, наскільки компонент нестабільний, можна порахувати кількість вхідних та вихідних залежностей.

Нестабільність = Кількість вихідних / (У вхідних + Кількість вихідних)

Принцип стабільності абстракцій

Компонент має бути настільки ж абстрактним, наскільки він стабільним.

Абстрактність = Кількість абстрактних класів та інтерфейсів у компоненті / Загальна кількість класів у компоненті

По осі X – нестабільність, по осі Y – абстрактність. Слід дотримуватись лінії main sequence і уникати зон по кутах:

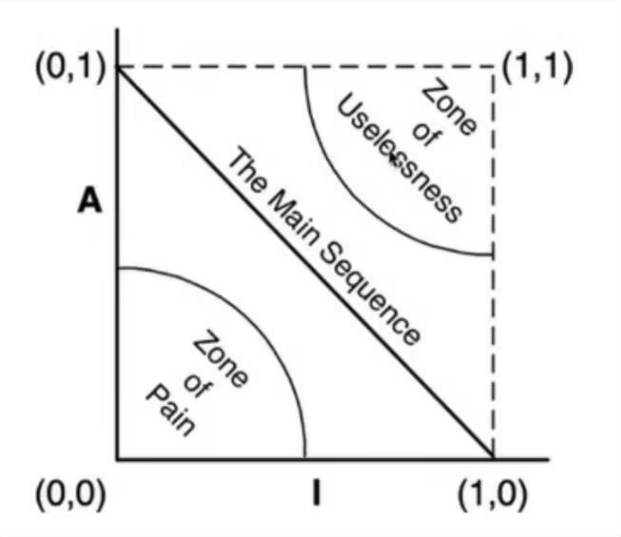


Рис. 1.2. Зони виключення

1.3. SOLID принципи

П’ять принципів допомагають нам зрозуміти необхідність певних шаблонів проектування та архітектури програмного забезпечення загалом. Тому я вважаю, що це тема, яку повинен вивчати кожен розробник.

Принципи SOLID були вперше введені відомим комп’ютерником Робертом Дж. Мартіном (він же Дядько Боб) у своїй роботі в 2000 році. Але абревіатуру SOLID ввів пізніше Майкл Фезерс.

Дядько Боб також є автором бестселерів Чистий код і Чиста архітектура, а також є одним з учасників «Agile Alliance».

Тому не дивно, що всі ці концепції чистого кода, об’єктно-орієнтованої архітектури та шаблонів проектування якимось чином пов’язані та доповнюють один одного: [4]

– принцип єдиної відповідальності;

– принцип відкритого-закритого;

– принцип заміщення Ліскова;

– принцип сегрегації інтерфейсу;

– принцип інверсії залежності.

Принцип єдиної відповідальності стверджує, що клас повинен робити одну справу, і тому він повинен мати лише одну причину для змін.

Щоб сформулювати цей принцип більш технічно: лише одна потенційна зміна (логіка бази даних, логіка реєстрації тощо) у специфікації програмного забезпечення повинна мати можливість впливати на специфікацію класу.

Це означає, що якщо клас є контейнером даних, як-от клас книги або клас студента, і він має деякі поля, що стосуються цієї сутності, він повинен змінюватися лише тоді, коли ми змінюємо модель даних.

Важливо дотримуватися принципу єдиної відповідальності. Перш за все, оскільки багато різних команд можуть працювати над одним проектом і редагувати один клас з різних причин, це може призвести до несумісних модулів.

По-друге, це полегшує контроль версій. Наприклад, припустимо, що у нас є клас збереження, який обробляє операції з базою даних, і ми бачимо зміни в цьому файлі в коммітах GitHub.

Принцип відкритості-закритості вимагає, щоб класи були відкритими для розширення і закритими для модифікації.

Модифікація означає зміну коду існуючого класу, а розширення означає додавання нових функцій.

Отже, цей принцип хоче сказати: ми повинні мати можливість додавати нові функції, не торкаючись існуючого коду для класу. Це тому, що щоразу, коли ми змінюємо існуючий код, ми ризикуємо створити потенційні помилки. Тому ми повинні уникати торкання перевіреного та надійного (переважно)

виробничого коду, якщо це можливо. Зазвичай це робиться за допомогою інтерфейсів і абстрактних класів.

Принцип підстановки Ліскова стверджує**,** що підкласи повинні бути замінними своїми базовими класами.

Це означає, що, враховуючи, що клас B є підкласом класу A, ми повинні мати можливість передати об'єкт класу B будь-якому методу, який очікує об'єкт класу A, і в цьому випадку метод не повинен давати ніяких дивних результатів.

Це очікувана поведінка, оскільки, коли ми використовуємо успадкування, ми припускаємо, що дочірній клас успадковує все, що має суперклас. Дочірній клас розширює поведінку, але ніколи не звужує її**.**

Сегрегація означає відокремлення речей, а принцип сегрегації інтерфейсу — це розділення інтерфейсів.

Принцип стверджує, що багато специфічних для клієнта інтерфейсів краще, ніж один інтерфейс загального призначення. Клієнтів не слід змушувати реалізовувати непотрібну їм функцію.

Принцип інверсії залежностей стверджує, що наші класи повинні залежати від інтерфейсів або абстрактних класів замість конкретних класів і функцій.

У своїй статті (2000) дядько Боб підсумовує цей принцип так:

«Якщо OCP визначає мету архітектури OO, то DIP визначає основний механізм».

Ці два принципи справді пов'язані, і ми застосовували цю модель раніше, коли обговорювали принцип відкритого-закритого.

Оскільки програми Android збільшуються, важливо визначити архітектуру, яка дозволить програмі масштабуватися, підвищить її надійність і полегшить тестування програми.

Архітектура програми визначає межі між частинами програми та обов'язки, які має мати кожна частина. Щоб задовольнити вищезгадані

потреби, ви повинні розробити архітектуру свого додатка відповідно до кількох конкретних принципів.

Стійкі моделі ідеально підходять з наступних причин:

Користувачі не втрачають дані, якщо ОС Android знищить вашу програму, щоб звільнити ресурси.

Додаток продовжує працювати у випадках, коли мережеве з'єднання нестабільне або недоступне.

Якщо архітектура програми базується на класах моделі даних, це робить додаток більш тестованим і надійним. [5]

Рівень інтерфейсу користувача, який відображає дані програми на екрані.

Рівень даних, який містить бізнес-логіку вашого додатка та надає дані програми.

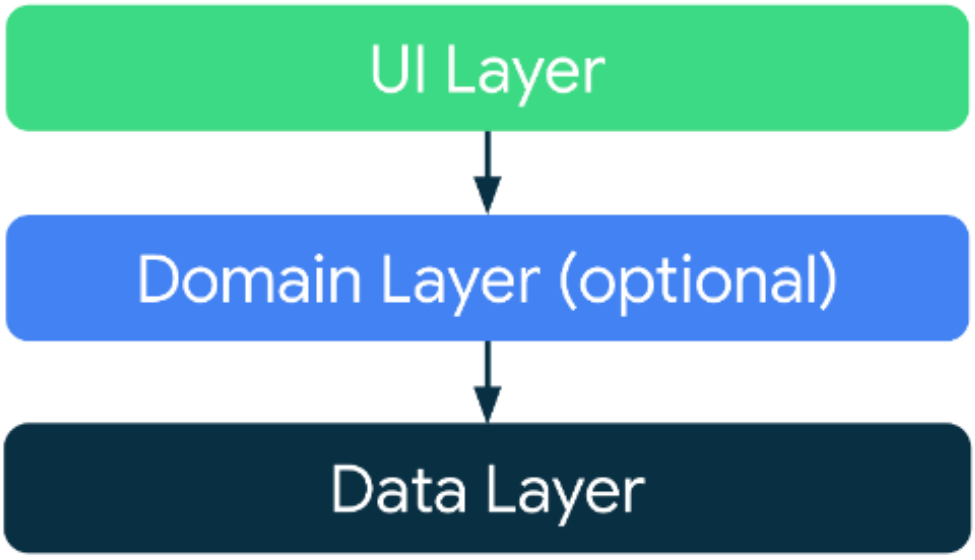


Рис. 1.3. Діаграма типової архітектури дордатка

Можна додати додатковий шар, який називається шаром домену, щоб спростити та повторно використовувати взаємодію між інтерфейсом користувача та рівнями даних.

Рівень інтерфейсу користувача

Роль рівня інтерфейсу користувача (або рівня презентації) полягає у відображенні даних програми на екрані. Щоразу, коли дані змінюються через взаємодію з користувачем (наприклад, натискання кнопки) або зовнішній вхід (наприклад, відповідь мережі), інтерфейс користувача має оновлюватися, щоб відобразити зміни.

Рівень UI складається з двох речей:

1. Елементи інтерфейсу користувача, які відображають дані на екрані. Ви створюєте ці елементи за допомогою функцій Views або Jetpack Compose.
2. Власники стану (наприклад, класи ViewModel), які зберігають дані, надають їх інтерфейсу та обробляють логіку.

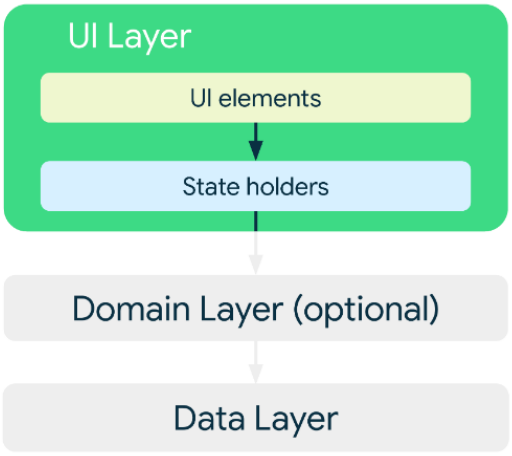


Рис. 1.4. Роль інтерфейсу користувача в архітектурі

Рівень домену програми містить бізнес-логіку. Бізнес-логіка — це те, що надає цінність вашому додатку — вона складається з правил, які визначають, як ваш додаток створює, зберігає та змінює дані.

Рівень даних складається із сховищ, кожне з яких може містити від нуля до багатьох джерел даних. Вам слід створити клас сховища для кожного різного типу даних, які ви обробляєте у своїй програмі. Наприклад, ви можете створити клас MoviesRepository для даних, пов’язаних з фільмами, або клас PaymentsRepository для даних, пов’язаних із платежами.

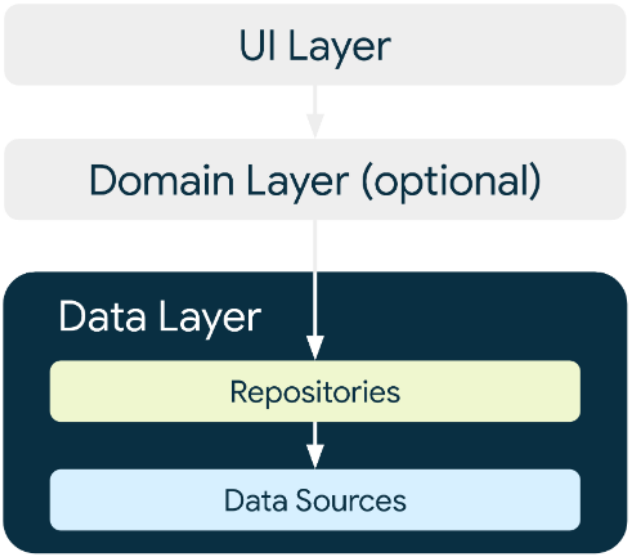


Рис. 1.5. Роль рівня даних в архітектурі

Класи репозитарію відповідають за такі завдання:

1. Відкриття даних для решти програми.
2. Централізація змін до даних.
3. Вирішення конфліктів між кількома джерелами даних.
4. Абстрагування джерел даних з решти програми.
5. Містить бізнес-логіку.

Кожен клас джерела даних повинен нести відповідальність за роботу лише з одним джерелом даних, яким може бути файл, мережеве джерело або локальна база даних. Класи джерел даних є містком між додатком і системою для операцій з даними.

Рівень домену – це необов’язковий рівень, який знаходиться між інтерфейсом користувача та рівнями даних.

Рівень домену відповідає за інкапсуляцію складної бізнес-логіки або простої бізнес-логіки, яка повторно використовується кількома моделями перегляду. Цей шар необов’язковий, оскільки не всі програми мають ці вимоги. Ви повинні використовувати його лише тоді, коли це необхідно — наприклад, щоб обробити складність або сприяти повторному використанню.

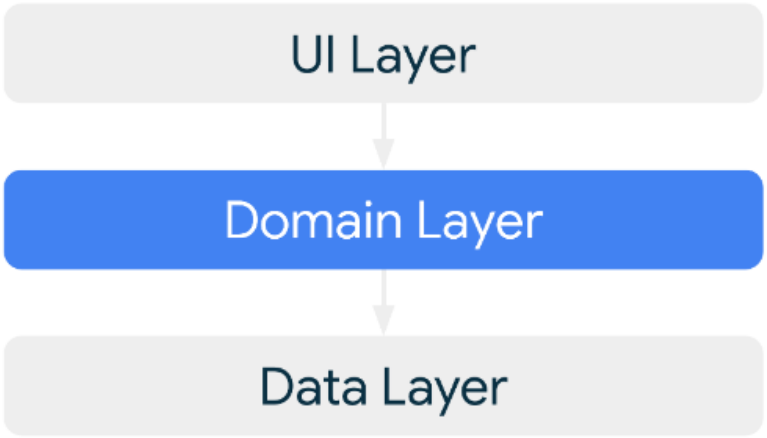


Рис. 1.6. Роль рівня домену в архітектурі

1.4. Model View Controller (MVC)

Архітектура MVC (модель – зображення – контролер).

Одним з найстаріших і найбільш широко використовуваних шаблонів проектування в архітектурі програмного забезпечення є MVC. У ньому є сильне розділення між представленням – як представляти дані, моделлю – як структурувати дані та контролером – як обробляти взаємодію користувача. Здебільшого Android розроблений так, щоб він міг слідувати шаблону MVC. Однак проблема з реалізацією MVC в Android полягає в тому, що Activity є одночасно представленням і контролером, що порушує принцип єдиної відповідальності, який є ключовим для цієї архітектури. [6]

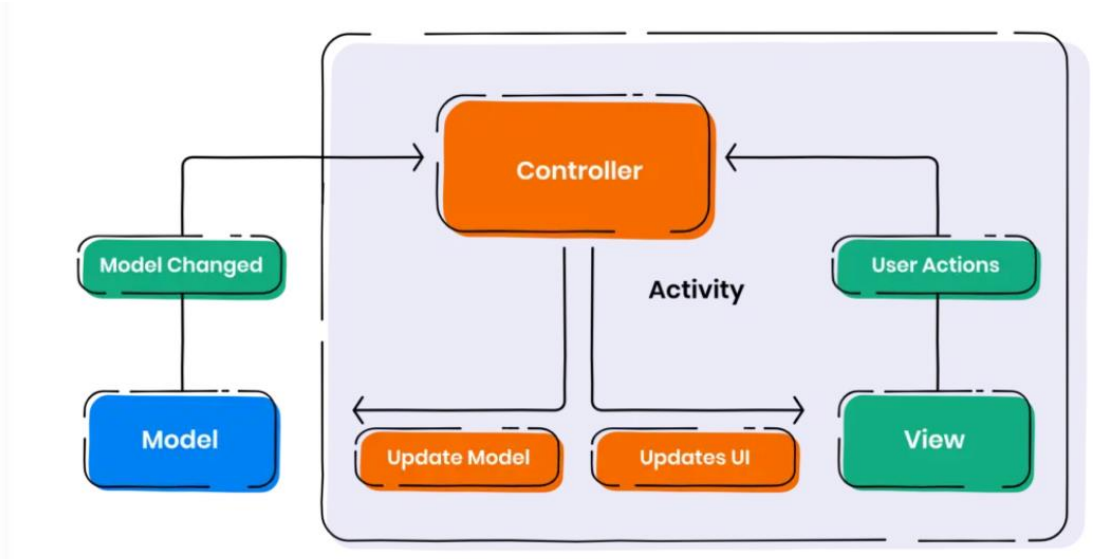


Рис. 1.7. Візуальне зображення MVC архітектури

1. Модель: шар для зберігання даних. Він відповідає за обробку логіки домену (реальних бізнес-правил) і зв'язку з базою даних і мережевими рівнями.

2. Зображення: шар інтерфейсу користувача. Він забезпечує візуалізацію даних, що зберігаються в моделі.

3. Контролер: рівень, який містить основну логіку. Він отримує інформацію про поведінку користувача та оновлює модель відповідно до потреб.

У схемі MVC і View, і Controller залежать від моделі. Дані програми оновлюються контролером, а View отримує дані. У цьому шаблоні модель можна було б тестувати незалежно від інтерфейсу користувача, оскільки вона відокремлена. Існує кілька можливих підходів до застосування шаблону MVC. Якщо Views дотримуються принципу єдиної відповідальності, то їхня роль полягає лише в тому, щоб оновлювати контролер для кожної події користувача та просто відображати дані з моделі, не впроваджуючи жодної бізнес-логіки. У цьому випадку тестів інтерфейсу користувача має бути достатньо, щоб охопити функціональні можливості View. [7]

Переваги:

1. Шаблон MVC підвищує тестованість коду та полегшує впровадження нових функцій, оскільки він дуже підтримує розділення проблем.

2. Модульне тестування моделі та контролера можливе, оскільки вони не розширюють і не використовують жодного класу Android.

3. Функціональні можливості View можна перевірити за допомогою тестів інтерфейсу користувача, якщо View дотримується принципу єдиної відповідальності (оновлювати контролер і відображати дані з моделі без реалізації логіки домену)

Недоліки:

1. Рівні коду залежать один від одного, навіть якщо MVC застосовано правильно.

2. Немає параметрів для обробки логіки інтерфейсу користувача, наприклад, як відображати дані.

1.5. Model View Presenter (MVP)

Спільнота Android все частіше почала використовувати шаблон MVP, де бізнес-логіка була визначена всередині класів, які називаються Presenters. Дія буде розширювати подання та взаємодіяти з доповідачем, який інформуватиме доповідача про дії користувача. Ця установка виявилася дуже ефективною, оскільки бізнес-логіка добре ізольована, а перегляд можна замінити незалежно від доповідачів.

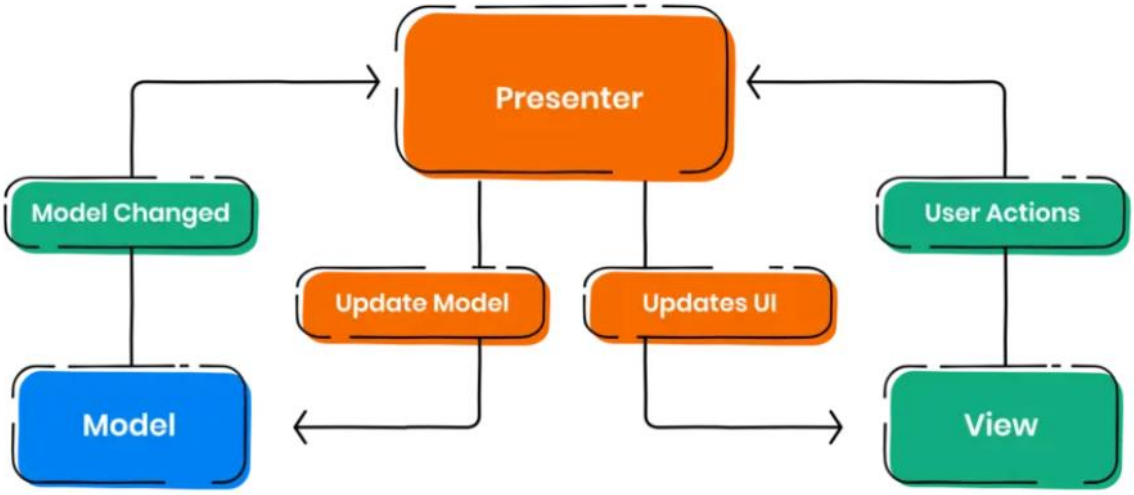


Рис. 1.8. Візуальне зображення MVP архітектури

Шаблон MVP — це друга ітерація архітектури додатків Android. Цей шаблон широко прийнято і все ще рекомендується для майбутніх розробників. Призначення кожного компонента легко дізнатися:

1. Модель: шар для зберігання даних. Він відповідає за обробку логіки домену (реальних бізнес-правил) і зв'язку з базою даних і мережевими рівнями.

2. Зображення: шар інтерфейсу користувача. Він забезпечує візуалізацію даних і відстежує дії користувача, щоб сповістити доповідача.

3. Доповідач: отримує дані з моделі та застосовує логіку інтерфейсу користувача, щоб вирішити, що відображати. Він керує станом View і виконує дії відповідно до сповіщень, які користувач вводить із View.

У схемі MVP View і Presenter тісно пов’язані між собою і мають посилання один на одного. Щоб зробити код читабельним і легшим для розуміння, для визначення зв’язку Presenter і View використовується клас інтерфейсу Contract. Подання є абстрактним і має інтерфейс, щоб увімкнути Presenter для модульного тестування.

Переваги:

1. Немає концептуальних зв’язків у компонентах Android.

2. Легке обслуговування та тестування коду, оскільки рівень моделі, представлення та презентатора програми розділені.

Недоліки:

1. Якщо розробник не дотримується принципу єдиної відповідальності, щоб зламати код, тоді рівень Presenter має тенденцію розширюватися до величезного всезнаючого класу.

1.6. Model View ViewModel (MVVM)

Архітектура MVVM (Model – View – ViewModel).

Виникла проблема з шаблоном MVP, він жодним чином не реагував. Довелося написати багато шаблонів, щоб підключити оновлення моделі до View. Шукаючи потенційні відповіді, спільнота Android зрозуміла, що реактивний підхід буде простішим і ефективнішим для мобільних архітектур. ViewModel — це модель, яку спостерігає View, і щоразу, коли модель змінюється, View оновлюватиметься самостійно. Тут на допомогу приходить

бібліотека зв’язування даних Google – вона автоматично передає LiveData від ViewModels до Views.

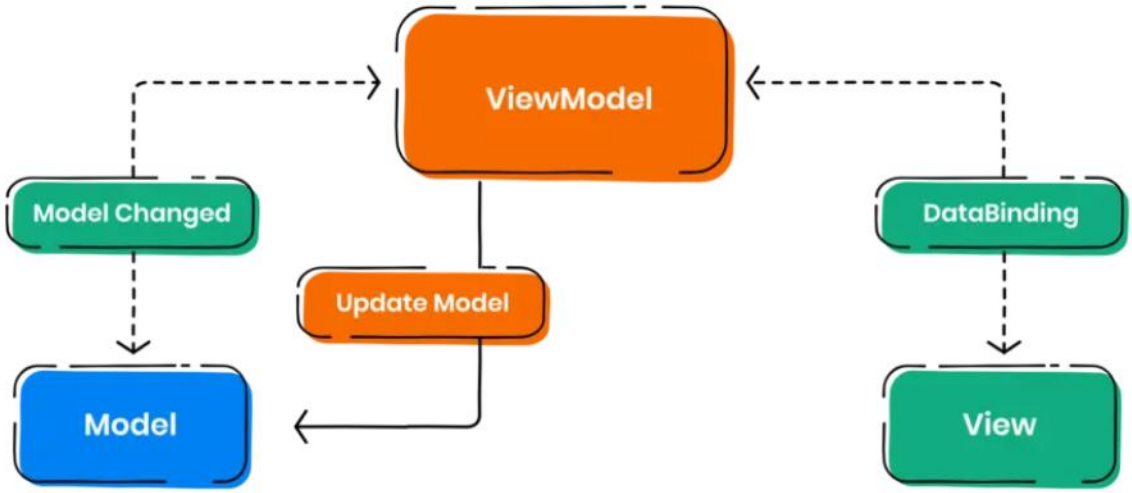


Рис. 1.9. Візуальне зображення MVVM архітектури

Третя ітерація архітектури Android — це шаблон MVVV. Під час випуску компонентів архітектури Android команда Android рекомендувала цей шаблон архітектури. Нижче наведено окремі шари коду:

Модель: цей рівень відповідає за абстракцію джерел даних. Модель і ViewModel працюють разом, щоб отримати та зберегти дані.

Перегляд: мета цього шару – інформувати ViewModel про дії користувача.

ViewModel: надає ті потоки даних, які мають відношення до View.

Шаблони MVVM і MVP дуже схожі, оскільки обидва ефективно абстрагують стан і поведінку шару перегляду.

У схемі MVVM View інформує ViewModel про різні дії. View має посилання на ViewModel, тоді як ViewModel не має інформації про View. Відношення «багато до одного», яке існує між View і ViewModel і MVVM, підтримує двостороннє прив’язування даних між обома.

1.7. Model View Intent (MVI)

Архітектура MVI (Модель – Вид – Намір).

Для більш детального підходу можна ввести концепцію наміру. Кожна взаємодія користувача є екземпляром набору намірів, які визначають екран програми. Кожна зміна на екрані інкапсулюється в інший Intent, який запускається з центрального місця, такого як Presenter, Controller або кінцевий автомат. Основна ідея тут полягає в тому, щоб забезпечити односпрямований потік даних (UDF), де дані та зміни на екрані надходять з одного місця в одному напрямку.

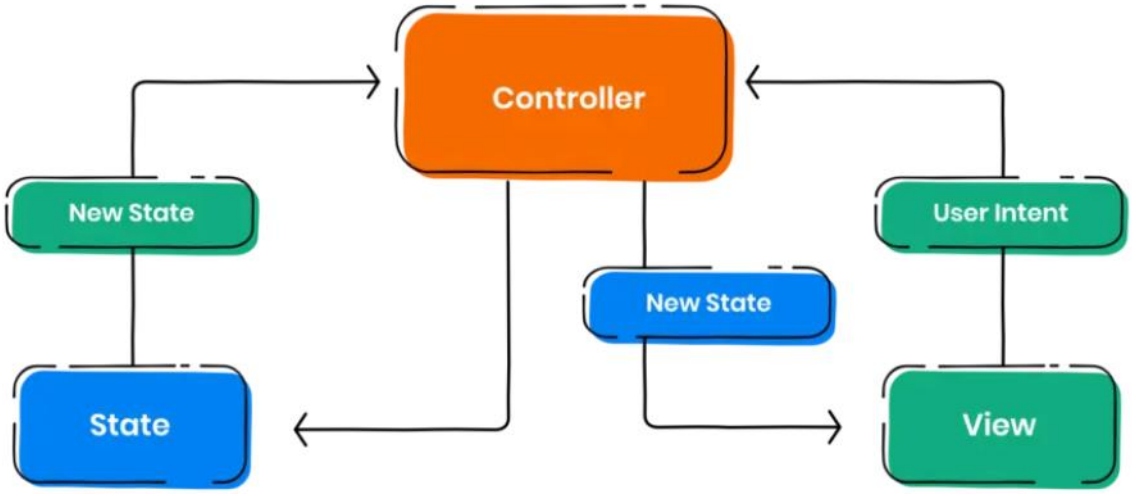


Рис. 1.10. Візуальне зображення MVI архітектури

У міру зростання програми або додавання незапланованих заздалегідь функціональних можливостей – без чіткого керування станом відображення подання разом із бізнес-логікою може бути трохи заплутаним.

Чим більш масштабований код програми, тим гнучкіший він до нових ідей та оновлень. Масштабованість, гнучкість і легкість тестування - ось що пропонує нам архітектура MVI. [8]

Основні переваги MVI:

– єдине джерело істини - один незмінний стан, загальний для всіх верст;

– як єдине джерело істини;

– односпрямований і циклічний потік даних;

– легкість виявлення та виправлення помилок;

– легкість тестування коду;

– можливість тестувати всі рівні програми за допомогою модульних тестів.

Кожне рішення, крім переваг, яке воно дає, має і свої недоліки. У випадку з MVI на даний момент я бачу дві істотні слабкості:

– більш стандартний, порівняно з іншими архітектурами;

– менш популярні, ніж MVP або MVVM, тому у разі проблем важче отримати допомогу чи дізнатися щось у спільноти.

1.8. Патерни проєктування

Шаблони проєктування (патерн, design pattern) - це багато разів застосовувана архітектурна конструкція, що надає рішення для загальної проблеми проектування в рамках конкретного контексту й описує значимість цього рішення. [9]

Патерн — це не закінчений приклад дизайну, який можна безпосередньо перевести в код, а опис або приклад того, як вирішити проблему таким чином, який можна використовувати в різних ситуаціях. Об'єктно-орієнтовані шаблони зазвичай показують зв'язки та взаємодію між класами або об'єктами, не вказуючи, які кінцеві класи або об'єкти програми будуть використані. Алгоритми не вважаються шаблонами, оскільки вони вирішують обчислювальні проблеми, а не проблеми проектування.

У 1970-х роках архітектор Крістофер Александер склав набір шаблонів дизайну. У сфері архітектури ця ідея не отримала розвитку, як це було пізніше в області розробки програмного забезпечення. Відповідно до визначення Крістофера Александера: «Кожне типове рішення описує певну проблему, що

повторюється, і ключ до її вирішення таким чином, що ви можете використовувати ключ кілька разів, ніколи не отримуючи того самого результату».

У 1987 році Кент Бек і Уорд Каннінгем перейняли ідеї Александера і розробили модель розробки програмного забезпечення графічної оболонки на основі мови Smalltalk.

У 1988 році Ерік Гамма почав писати свою докторську дисертацію в Цюрихському університеті про загальну застосовність методу для розробки програмного забезпечення.

Джеймс Коплін працював над розробкою ідіом програмування C++ у 1989-1991 роках і опублікував книгу Advanced C++ Idioms у 1991 році. Того ж року Ерік Гамма завершив свою докторську дисертацію та переїхав до Сполучених Штатів, де спільно з Річардом Хелмом, Ральфом Джонсоном і Джоном Влісідсом опублікував книгу «Патерни проектування — елементи багаторазового об’єктно-орієнтованого програмного забезпечення». У цій книзі описано 23 шаблони проектування. Крім того, група авторів цієї книги відома широкому загалу як Gang of Four (часто скорочено до GoF). Ця книга є причиною зростання популярності шаблонів проектування.

Іншою видатною фігурою в галузі проектування систем програмного забезпечення, що підтримує шаблони використання, є Мартін Фаулер, автор його 21-ї книги «Паттерни архітектури корпоративних додатків». Як зазначає Мартін Фаулер у своїй книзі: «Коли ви збираєтеся використовувати стандартні рішення, пам’ятайте, що вони є початковою, а не кінцевою точкою».

У книзі «UML and Applications of Design Patterns» Крейга Лармана описано 9 шаблонів GRASP (шаблони загального програмного забезпечення призначення відповідальності) — шаблони для об’єктно-орієнтованого проектування шляхом розподілу обов’язків між класами та об’єктів між об’єктами. Загальні завдання. Кожен із них допомагає вирішувати певні проблеми, які виникають під час об’єктно-орієнтованого аналізу та можуть

виникнути майже у будь-якому проекті розробки програмного забезпечення. Головна користь кожного окремого шаблону полягає в тому, що він описує рішення цілого класу абстрактних проблем. Також той факт, що кожен шаблон має своє ім'я, полегшує дискусію про абстрактні структури даних (ADT) між розробниками, так як вони можуть посилатися на відомі шаблони. Таким чином, за рахунок шаблонів проводиться уніфікація термінології, назв модулів і елементів проекту.

Правильно сформульований шаблон проектування дозволяє, відшукавши вдале рішення, користуватися ним знову і знову.

Однак іноді шаблони консервують громіздку і малоефективну систему понять, розроблену вузькою групою. Коли кількість шаблонів зростає, перевищуючи критичну складність, виконавці починають ігнорувати шаблони і всю систему, з ними пов'язану. Нерідко шаблонами замінюється відсутність або недоліки документації, яка складна програмному середовищі.

Є думка, що сліпе застосування шаблонів з довідника, без осмислення причин і передумов виділення кожного окремого шаблону, уповільнює професійне зростання програміста. Люди, які дотримуються цієї думки, вважають, що знайомитися зі списками шаблонів треба тоді, коли "доріс" до них в професійному плані - і не раніше. Хороший критерій потрібного ступеня професіоналізму - виділення шаблонів самостійно, на підставі власного досвіду. При цьому, зрозуміло, знайомство з теорією, пов'язаної з шаблонами, корисно на будь-якому рівні професіоналізму і направляє розвиток програміста в правильну сторону. Сумніву піддається тільки використання шаблонів "за довідником".

Шаблони можуть пропагувати погані стилі розробки додатків, і часто сліпо застосовуються.

Класифікація шаблонів проектування.

Патерни проектування класів / об'єктів:

– Адаптер (Adapter) – GoF;

– Декоратор (Decorator) або Оболонка (Wrapper) – GoF;

– Заступник (Proxy) або Сурогат (Surrogate) – GoF;

– Інформаційний експерт (Information Expert) – GRASP;

– Компонувальник (Composite) – GoF;

– Міст (Bridge), Handle (описувач) або Тіло (Body) – GoF;

– Низька зв'язаність (Low Coupling) – GRASP;

– Пристосуванець (Flyweight) – GoF;

– Стійкий до змін (Protected Variations) – GRASP;

– Фасад (Facade) – GoF Патерни проектування поведінки класів / об'єктів;

– Інтерпретатор (Interpreter) – GoF;

– Ітератор (Iterator) або Курсор (Cursor) – GoF;

– Команда (Command), Дія (Action) або Транзакція (Транзакція) – GoF;

– Спостерігач (Observer), Опублікувати - підписатися (Publish - Subscribe) або Delegation Event Model – GoF;

– Не розмовляйте з невідомими (Don't talk to strangers) – GRASP;

– Відвідувач (Visitor) – GoF;

– Посередник (Mediator) - GoF • Стан (State) – GoF;

– Стратегія (Strategy) – GoF;

– Зберігач (Memento) – GoF;

– Ланцюжок обов'язків (Chain of Responsibility) – GoF;

– Шаблонний метод (Template Method) – GoF;

– Високе зачеплення (High Cohesion) – GRASP;

– Контролер (Controller) – GRASP;

– Поліморфізм (Polymorphism) – GRASP;

– Штучний (Pure Fabrication) – GRASP;

– Перенаправлення (Indirection) – GRASP.

Твірні патерни проектування:

– Абстрактна фабрика (Abstract Factory, Factory), ін. назва Інструментарій (Kit) – GoF;

– Одинак (Singleton) – GoF;

– Прототип (Prototype) – GoF;

– Творець примірників класу (Creator) – GRASP;

– Будівельник (Builder) – GoF;

– Фабричний метод (Factory Method) або Віртуальний конструктор (Virtual Constructor) - GoF Архітектурні системні патерни.

Структурні патерни:

– Репозиторій;

– Клієнт / сервер;

– об'єктно - орієнтований, Модель предметної області (Domain Model), модуль таблиці (Data Mapper);

– Багаторівнева система (Layers) чи абстрактна машина;

– Потоки даних (конвеєр або фільтр) ⎫ Патерни управління;

– Патерни централізованого управління;

– Виклик - повернення (сценарій транзакції - окремий випадок);

– Диспетчер;

– Патерни управління, засновані на подіях;

– Передача повідомлень;

– Керування перериваннями;

– Патерни, що забезпечують взаємодію з базою даних;

– Активний запис (Active Record);

– Одиниця роботи (Unit Of Work);

– Завантаження на вимогу (Lazy Load);

– Колекція об'єктів (Identity Map);

– Безліч записів (Record Set);

– Успадкування з однією таблицею (Single Table Inheritance);
– Успадкування з таблицями для кожного класу (Class Table);
– Оптимістичне автономне блокування (Optimistic Offline Lock);
– Відображення з допомогою зовнішніх ключів;
– Відображення з допомогою таблиці асоціацій (Association Mapping);
– Песимістичне автономне блокування (Pessimistic Offline Lock);
– Перетворювач даних (Data Mapper);
– Шлюз запису даних (Row Data Gateway);
– Шлюз таблиці даних (Table Data Gateway);
– Патерни, призначені для представлення даних у Web;
– Двоетапне подання (Two Step View);
– Контролер додатка (Application Controller);

Патерни інтеграції корпоративних інформаційних систем

– Структурні патерни інтеграції;
– Взаємодія "крапка - крапка";
– Взаємодія "зірка" (інтегруюча середа);
– Змішаний спосіб взаємодії;
– Патерни за методом інтеграції;
– Інтеграція систем за даними (data-centric);
– Функціонально-центричний (function-centric) підхід;

1.9. Принципи програмування

KISS

Основна ідея принципу KISS полягає в тому, що чим простіше, тим краще, щоб кожен розробник міг зрозуміти, що відбувається в класах і методах.

З цього випливають два аргументи:

1. Будьте короткими.

2. Будьте простими та зрозумілими.

Хороший розробник може написати ту саму логіку так, щоб її зрозуміли всі. Хороший критерій потрібного ступеня професіоналізму - виділення шаблонів самостійно. Якщо ви створюєте логіку, яку розумієте лише ви, ви розумієте, що щось не так. [10]

Крім того, принцип *KISS* не тільки для програмування. Простота є метою будь-якого досвіду користувача — жодних непотрібних маніпуляцій і тертя користувача. Те ж саме стосується, наприклад, автомобілів, телефонів і дизайну загалом – просто і просто.

DRY

Коли ми створюємо певну функціональність, ми завжди повинні писати подібні або однакові методи та класи.

У якийсь момент хтось сказав, що хоче бачити в посиланнях двокрапки замість підкреслень. У цьому випадку вам доведеться перевірити всю програму і виправити її. Це займає багато часу. Крім того, зміна коду може випадково щось порушити.

Відповідно до принципу *DRY*, ви повинні не просто писати один і той же код один раз, а уникати дублювання знань.

Якщо ви десь бачите, що такий код був написаний, настав час його рефакторинг. Тим не менш, тут варто поглянути на спільне. Повний спільний доступ можна зробити у багаторазово використовувані класи, компоненти або методи.

Головне завдання – зменшити кількість повторюваних знань. Не повністю прибрати, а зменшити кількість. Фрагмент коду має бути реалізований лише в одному місці. Крім того, не забувайте про *SRP (Single Responsibility Principle)*, кожен модуль має виконувати одну справу. Ми обговоримо це пізніше.

YAGNI — не знадобиться

Перш ніж його реалізувати, потрібно подумати, чи потрібен він мені зараз. Так, ви можете думати про все, але коли ви сідаєте виконувати завдання, воно повинно виконувати саме те, що вам потрібно прямо тут, прямо зараз. Не потрібно створювати методи, які можуть знадобитися в майбутньому.

90% коду, написаного на майбутнє, не працює. Це лише погіршує розуміння вашого коду з часом, тому що інші люди не розуміють ваших методів і для чого вони потрібні. Ділові потреби швидко змінюються, і те, що здається необхідним, завтра може виявитися неважливим.

ВИСНОВКИ ДО РОЗДІЛУ 1

У перщому розділі розглянуто основні архітектурні шаблони разом з принципами розробки «чистої» архітектури. Проаналізовано переваги та недоліки кожного з підходів, виокремелі загальні рекомендації по написанню кода та структуруванню його у проекті.

Найбільш вагомий вплив на розробку при використанні правильного підходу з вдалою архітектурою – це швидкість та ефективність роботи у команді, коли всі притримуються одного стилю написання кода розробляти разом стає набагато легше. Історія розвитку архітектурних підходів є також важливою, адже в ній лежать приницпи ООП та їх найбільш вдалі використання так, щоб полегшити життя розробникам.

Весь проаналізований матеріал придатний до використання зараз, та обрана архітектура у роботі буде брати з кожної згаданої вище найкраще. Програмування ніколи не обмежує ні в чому розробника, все залежить від вмінь користуватись інструментами та вхідними даними, саме це дозволяє розвиватись програмним концепціям та підходам у розробці.

РОЗДІЛ 2

ПРОЕКТУВАННЯ АРХІТЕКТУРИ І РОБОТИ ДОДАТКУ

2.1. Найкращі існуючі практики для створення Android додатку

Android SDK

Для початку потрібно встановити Android SDK десь у своєму домашньому каталозі або в іншому місці, незалежному від програми. Деякі дистрибутиви IDE включають SDK після встановлення та можуть розміщувати його в тому самому каталозі, що й IDE. Це може бути погано, коли потрібно оновити (або перевстановити) IDE, оскільки можна втратити інсталяцію SDK, як наслідок довге повторне завантаження. [11]

Також потрібно уникайте розміщення SDK у системному каталозі, якому можуть знадобитися кореневі дозволи, щоб уникнути проблем із дозволами.

Побудова системи

Варіантом за замовчуванням має бути Gradle із використанням плагіна Android Gradle. Важливо, щоб процес збирання програми визначався написаними людиною файлами Gradle, а не залежав від конкретних конфігурацій IDE. Це забезпечує узгоджену збірку між інструментами та кращу підтримку систем безперервної інтеграції.

Структура проекту

Хоча Gradle пропонує високу гнучкість у структурі проекту, якщо немає вагомих причин робити інакше, бажано прийняти його структуру за замовчуванням, оскільки це спрощує ваші сценарії збірки.

Конфігурація *Gradle*

Загальна структура. Дотримуючись посібника *Google* щодо *Gradle* для *Android*. Мінімальна версія андроїд для встновлення додатку - *minSdkVersion*: 21. Рекомендовано переглянути таблицю використання версії *Android* перед визначенням мінімального необхідного *API*. Наведені статистичні дані є глобальними і можуть відрізнятися, якщо орієнтуватися на певний регіональний/демографічний ринок.

Варто зазначити, що деякі функції матеріального дизайну доступні лише на *Android* 5.0 (рівень API 21) і вище. Крім того, починаючи з *API* 21, бібліотека підтримки multidex більше не потрібна.

Рекомендовано також використовувати маленькі завдання. Замість сценаріїв (*shell, Python, Perl* тощо) можна створювати завдання в Gradle. Просто дотримуючись документації *Gradle* та *Google*, що також надає кілька корисних рецептів *Gradle*, специфічних для *Android*. [12]

Паролі

У *build.gradle* програми необхідно буде визначити *signingConfigs* для збірки випуску. Чого слід уникати при звичайній розробці додатку що виставляєтсья за замовчуванням наведено на рис. 2.1:

```
signingConfigs {
    release {
        // DON'T DO THIS!!
        storeFile file("myapp.keystore")
        storePassword "password123"
        keyAlias "thekey"
        keyPassword "password789"
    }
}
```

Рис. 2.1. Спосіб за замовчуваням налаштування релізних випусків

Натомість краще створити файл *gradle.properties*, який не слід додавати до системи контролю версій, зміст якого на рис. 2.2:

```
KEYSTORE_PASSWORD=password123
KEY_PASSWORD=password789
```

Рис. 2.2. Файл з інформацією про випуск додатку в окремому файлі

Цей файл автоматично імпортується *Gradle*, що дає змогу використовувати його в *build.gradle* як такий, що наведено на рис. 2.3:

```
signingConfigs {
    release {
        try {
            storeFile file("myapp.keystore")
            storePassword KEYSTORE_PASSWORD
            keyAlias "thekey"
            keyPassword KEY_PASSWORD
        }
        catch (ex) {
            throw new InvalidUserDataException("You should define KEYSTORE_PASSWORD and KEY_PASSWORD in gradl
        }
    }
}
```

Рис. 2.3. Використання файлу конфігурацій

Розв’язанню залежностей залишимо *Maven* перед імпортом файлів jar. Якщо явно включити файли *jar* у свій проект, вони будуть певною замороженою версією, наприклад 2.1.1. Завантаження *jar*-файлів і обробка оновлень громіздкі, і це проблема, яку Maven вже вирішує належним чином.

Уникнемо вирішення динамічних залежностей Maven наприклад 2.1.+, оскільки це може призвести до різних і нестабільних збірок або тонких, невідстежуваних відмінностей у поведінці між збірками. Використання статичних версій, таких як 2.1.1, допомагає створити більш стабільне, передбачуване та повторюване середовище розробки.

Використовуватимемо іншу назву пакета для невипускних збірок, а саме applicationIdSuffix для типу збірки налагодження, щоб мати можливість інсталювати як налагоджувальний, так і випускний apk на одному пристрої (також у майбутньому можна додати додаткові типи збірок із сваоїми налаштуваннями). Це буде особливо цінно після публікації програми.

```
android {
    buildTypes {
        debug {
            applicationIdSuffix '.debug'
            versionNameSuffix '-DEBUG'
        }

        release {
            // ...
        }
    }
}
```

Рис. 2.4 Використання різних типів збірки додатку

Надання доступу до файлу сховища ключів *APK* відлагодження через репозиторій програми економитиме час під час тестування на спільних пристроях і дозволятиме уникнути видалення/перевстановлення програми. Це також спрощує обробку роботи з деякими *Android SDK*, такими як *Facebook*, які вимагають реєстрації єдиного хешу сховища ключів. На відміну від файлу ключа випуску, файл ключа налагодження можна безпечно додати до свого сховища.

Спільний доступ до визначень форматування стилю коду

Надання спільного доступу до визначень стилю коду та форматування через репозиторій програм допомагає забезпечити візуально узгоджену базу коду та полегшує розуміння та перегляд коду.

Android Studio* як основна *IDE

Рекомендованою середою *IDE* для розробки Android є *Android Studio*, оскільки вона розроблена та постійно оновлюється *Google*, має гарну підтримку *Gradle*, містить низку корисних інструментів моніторингу та аналізу та повністю адаптована для розробки Android.

Уникатимемо у додатку додавання спеціальних файлів конфігурації *Android Studio*, таких як файли .iml, до системи керування версіями, оскільки

вони часто містять конфігурації, специфічні для вашої локальної машини, які не працюватимуть для колег, що будуть працювати над проектом.

Бібліотеки

Jackson — це бібліотека Java для серіалізації та десеріалізації *JSON*, вона має широкий і універсальний *API*, що підтримує різні способи обробки *JSON*: потокове передавання, модель дерева в пам'яті та традиційне зв'язування даних *JSON-POJO*.

Gson є ще одним популярним вибором, і оскільки бібліотека менша за Jackson, ви можете віддати перевагу їй, щоб уникнути обмеження методів у 65 тисяч.

Moshi, ще одна бібліотека Square з відкритим кодом, базується на досвіді розробки *Gson*, а також добре інтегрується з *Kotlin*.

Мережа, кешування та зображення. Є кілька перевірених у боях рішень для виконання запитів до серверів, які слід використовувати, а не впроваджувати власний клієнт. Рекомендовано базувати стек навколо *OkHttp* для ефективних *HTTP*-запитів і використовувати *Retrofit* для забезпечення типу безпечного рівня. Якщо ви обираєте ***Retrofit***, розгляньте Picasso для завантаження та кешування зображень.

Retrofit, *Picasso* та *OkHttp* створені однією компанією, тому вони гарно доповнюють один одного, а проблеми сумісності є рідкісними.

Glide — ще один варіант для завантаження та кешування зображень. Він підтримує анімовані *GIF*-файли, круглі зображення та твердження про кращу продуктивність, ніж *Picasso*, але також більшу кількість методів.

RxJava — це бібліотека для реактивного програмування, іншими словами, обробки асинхронних подій. Це потужна парадигма, але вона також має круту криву навчання. Потрібно бути обережним, перш ніж використовувати цю бібліотеку для розробки всієї програми. Існує багато публікацій про те як краще спроектувати систему з цією бібліотекою, щоб не нашкодити собі.

Використаємо *RxAndroid* для підтримки потоків *Android* і *RxBinding*, щоб легко створювати *Observable* з існуючих компонентів *Android*.

Retrolambda — це бібліотека Java для використання синтаксису лямбда-виразу в Android та інших платформах до *JDK8*. Це допомагає зберегти код щільним і читабельним, особливо якщо ви використовуєте функціональний стиль, наприклад у *RxJava*.

Android Studio пропонує підтримку коду лямбда *Java* 8. Будь-який інтерфейс із лише одним методом є «лямбда-дружнім» і може бути складений у більш жорсткий синтаксис. Або ж можна написати звичайний анонімний внутрішній клас, а потім дозволити *Android Studio* згорнути його в лямбда вираз автоматично.

Активіті та фрагменти

Серед спільноти розробників немає консенсусу щодо того, як найкраще організувати архітектури *Android* за допомогою фрагментів і активіті. *Square* навіть має бібліотеку для побудови архітектур переважно з *Views*, обходячи потребу у *Fragments*, але це все ще не вважається широко рекомендованою практикою в спільноті.

Через історію *Android API* можна вільно розглядати фрагменти як елементи інтерфейсу користувача на екрані. Іншими словами, фрагменти зазвичай пов’язані з інтерфейсом користувача. Активіті можна умовно вважати контролерами, вони особливо важливі для свого життєвого циклу та для управління станом. Однак, побачити різницю в цих ролях очевидно: активіті можуть мати ролі інтерфейсу користувача (забезпечення переходів між екранами), а фрагменти можуть використовуватися виключно як контролери. Рекомендації діяти обережно, приймаючи обґрунтовані рішення, оскільки вибір архітектури лише з фрагментами, або лише з активіті, або лише з представленнями має недоліки. На основі літератури виділемо, з чим слід бути обережним:

1) Уникати широкого використання вкладених фрагментів, оскільки можуть виникнути помилки матрьошки. Використовувати вкладені фрагменти лише тоді, коли це має сенс (наприклад, фрагменти в горизонтально ковзаючому *ViewPager* всередині екранного фрагмента) або якщо це добре обґрунтоване рішення.

2) Уникати надто багато коду в *Activities*. За можливості потрібно зберігати їх як легкі контейнери, які існують програмі переважно для життєвого циклу та інших важливих API інтерфейсу Android. Віддавайте перевагу однофрагментним діям замість звичайних дій – додайте код інтерфейсу користувача у фрагмент дії. Це робить його придатним для повторного використання на випадок, якщо вам знадобиться змінити його на макет із вкладками або на багатофрагментному екрані планшета. Уникайте діяльності без відповідного фрагмента, якщо тільки ви не приймаєте обґрунтоване рішення.

Структура пакетів Kotlin

Рекомендовано використовувати структуру пакета на основі функцій для вашого коду. Це має такі переваги:

1. Більш чітка залежність функцій і межі інтерфейсу.
2. Сприяє інкапсуляції.
3. Легше зрозуміти компоненти, які визначають функцію.
4. Зменшує ризик несвідомої зміни непов’язаного або спільного коду.
5. Простіша навігація: більшість пов’язаних класів буде в одному пакеті.
6. Легше видалити функцію.
7. Спрощує перехід до модульної структури збірки (кращий час збірки та підтримка миттєвих програм)

Альтернативний підхід визначення ваших пакетів за тим, як будується функція (шляхом розміщення пов’язаних дій, фрагментів, адаптерів тощо в окремих пакетах) може призвести до фрагментованої бази коду з меншою

гнучкістю впровадження. Найважливіше те, що це заважає здатності зрозуміти кодову базу з точки зору її основної ролі: надавати функції для програми.

Ресурси. Організація макета *XML*. Використовуватимо визначені нижче умови для правил форматування:

– один атрибут на рядок із відступом 4 пробіли;

– android:id як перший атрибут завжди;

– android:layout_**** атрибути вгорі;

– атрибут стилю внизу;

– позначте тег closer /> у окремому рядку, щоб полегшити впорядкування та додавання атрибутів;

– замість статичного кодування *android:text*, використання атрибутів *Designtime* буде кращим, доступних для Android Studio.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    >

    <TextView
        android:id="@+id/name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:text="@string/name"
        style="@style/FancyText"
        />

    <include layout="@layout/reusable_part" />

</LinearLayout>
```

Рис. 2.5. Приклад форматування XML документу

Як правило, атрибути *android:layout_**** мають бути визначені в макеті *XML*, тоді як інші атрибути *android*:**** мають залишатися в стилі *XML*. Це правило має винятки, але в цілому працює добре. Ідея полягає в тому, щоб зберегти лише макет (розташування, поле, розмір) і атрибути вмісту у файлах

макета, зберігаючи всі деталі зовнішнього вигляду (кольори, відступи, шрифт) у файлах стилів.

Винятками є:

1. *android:id*, очевидно, має бути у файлах макета.

2. *android:orientation* для *LinearLayout* зазвичай має більше сенсу у файлах макета.

3. *android:text* має бути у файлах макета, оскільки він визначає вміст.

4. Іноді має сенс створити загальний стиль, що визначає android:layout_width і android:layout_height, але за замовчуванням вони мають відображатися у файлах макета.

Використовувати стилі

Майже кожен проект потребує правильного використання стилів, тому що дуже часто повторюється поява для перегляду. Принаймні ви повинні мати загальний стиль для більшості текстового вмісту програми.

Безпека

Безпека швидко стає конкурентною відмінністю серед постачальників послуг розробки мобільних додатків. Здатність сучасних додатків, особливо на відносно відкритій платформі, такій як Android, захищати конфіденційність користувачів і дані піддається пильній перевірці. Безпека програм – це безперервний процес, який починається на початковому етапі розробки та розвивається разом із кіберзагрозами, якщо не випереджає їх. [13]

Безперервне тестування є одним із найкращих підходів для розробників, щоб захистити свої програми. Такі методи, як тестування на проникнення, можуть допомогти запобігти потенційним загрозам, висвітлюючи низку слабких місць безпеки. Розробка додатків для Android також має зосереджуватися на впровадженні надійних систем автентифікації та авторизації, які враховують такі ключові параметри, як конфіденційність, керування сеансами, керування ідентифікацією та безпека пристрою.

Уникайте використання загального чи загальнодоступного коду для розробки додатків і впроваджуйте найкращі методи кібербезпеки мобільних додатків під час написання коду.

Тестування

Краще всьо використовувати *JUnit* для модульного тестування. Звичайне модульне тестування без залежностей *Android* на *JVM* найкраще виконувати за допомогою *JUnit*. [14]

Тестування за допомогою *Robolectric* є неточним і неповним, оскільки воно працює, надаючи фіктивні реалізації платформи *Android*, що не надає жодних гарантій правильності. Замість цього використовуйте комбінацію модульних тестів на основі *JVM* і спеціальних інтеграційних тестів на пристрої.

Espresso полегшує написання тестів інтерфейсу користувача.

AssertJ-Android — бібліотека розширення *AssertJ*, що полегшує твердження під час тестування *Android*. *AssertJ* постачає модулі, які спрощують тестування специфічних компонентів Android, таких як бібліотеки служби підтримки *Android*, служби *Google Play* і *Appcompat*.

Тестове твердження виглядатиме так:

```
// Example assertion using AssertJ-Android
assertThat(layout).isVisible()
    .isVertical()
    .hasChildCount(5);
```

Рис. 2.6. Приклад тестового твердження

Конфігурація *Proguard*

ProGuard зазвичай використовується в проектах *Android* для зменшення та обфускації упакованого коду.

Використовування *ProGuard* залежить від конфігурації проекту. Зазвичай налаштовується Gradle на використання *ProGuard* під час створення випуску *APK*. [15]

```
buildTypes {
    debug {
        minifyEnabled false
    }
    release {
        signingConfig signingConfigs.release
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
    }
}
```

Рис. 2.7. Налаштування *ProGuard* файлу

Щоб визначити, який код потрібно зберегти, а який можна відкинути або завуалювати, потрібно вказати одну або кілька точок входу у свій код. Ці точки входу зазвичай є класами з основними методами, аплетами, мідлетами, активіті тощо. *Android Framework* використовує конфігурацію за замовчуванням, яку можна знайти в S*DK_HOME/tools/proguard/proguard-android.txt*. Використовуючи наведену вище конфігурацію, спеціальні правила *ProGuard* для конкретного проекту, визначені в my-project/app/proguard-rules.pro, будуть додані до конфігурації за замовчуванням.

Поширеною проблемою, пов'язаною з *ProGuard*, є збій програми під час запуску з *ClassNotFoundException* або *NoSuchFieldException* або подібним, навіть якщо команда збірки (тобто *assembleRelease*) виконана без попереджень. Це означає одне з двох:

– *ProGuard* видалив клас, enum, метод, поле або анотацію, вважаючи, що це не обов'язково.

– *ProGuard* затьмарив (перейменував) ім'я класу, переліку або поля, але його оригінальне ім'я використовується опосередковано, тобто через відображення Java.

Щоб запобігти видаленню ProGuard необхідних класів або членів класу, додамо параметри збереження до конфігурації ProGuard: «-keep class com.futurice.project.MyClass { *; }»

Зберігання даних

SharedPreferences механізм за замовчуванням для збереження простих значень, якщо програма працює в одному процесі, *SharedPreferences*, ймовірно, достатньо. Це хороший варіант за замовчуванням.

Є деякі ситуації, коли SharedPreferences не підходять:

1. Продуктивність: дані складні або їх багато.

2. Кілька процесів отримують доступ до даних: є віджети або віддалені служби, які працюють у власних процесах і потребують синхронізованих даних.

3. Реляційні дані окремі частини ваших даних є реляційними, і ви хочете забезпечити підтримку цих зв'язків.

4. Також можна зберігати складніші об'єкти, серіалізувавши їх у JSON для зберігання та десеріалізуючи їх під час отримання. Роблячи це, слід враховувати компроміси, оскільки це може бути не дуже продуктивним і непридатним для обслуговування.

ContentProviders використовують, якщо *SharedPreferences* недостатньо, стандартні платформи *ContentProviders*, є швидкими та безпечними.

Єдиною проблемою *ContentProviders* є кількість шаблонного коду, який потрібен для їх налаштування, а також низька якість навчальних посібників. Однак можна створити *ContentProvider* за допомогою бібліотеки, такої як *Schematic*, що значно зменшує зусилля.

Все ще потрібно самостійно написати код аналізу, щоб читати об'єкти даних зі стовпців *Sqlite* і навпаки. Можна серіалізувати об'єкти даних, наприклад, за допомогою *Gson*, і зберегти лише отриманий рядок. Таким чином ви втрачаєте продуктивність, але, з іншого боку, вам не потрібно оголошувати стовпець для всіх полів класу даних.

Використання *ORM*.

Загалом не рекомендовано використовувати бібліотеку відображення об'єктів-зв'язків, якщо у вас немає надзвичайно складних даних і ви маєте

гостру потребу. Вони, як правило, складні і вимагають часу для навчання. Якщо ви вирішите використовувати *ORM*, вам слід звернути увагу на те, чи безпечний процес, якщо цього вимагає ваша програма, оскільки багато існуючих рішень ORM, на диво, ні. [16]

Необхідність Stetho

Stetho — це міст для налагодження додатків *Android* від *Facebook*, який інтегрується з інструментами розробника браузера *Chrome* для робочого столу. За допомогою Stetho ви можете легко перевірити свою програму, особливо мережевий трафік. Це також дозволяє легко перевіряти та редагувати бази даних *SQLite* та спільні параметри у програмі. Однак потрібно переконатися, що *Stetho* увімкнено лише у збірці для налагодження, а не у варіанті збірки випуску.

Іншою альтернативою є *Chuck*, який, хоча й пропонує трохи спрощену функціональність, все ж корисний для тестувальників, оскільки журнали відображаються на пристрої, а не в складніших налаштуваннях підключеного браузера *Chrome*, які потрібні *Stetho*.

Необхідність *LeakCanary*

LeakCanary — це бібліотека, яка робить виявлення під час виконання та ідентифікацію витоків пам’яті більш рутинною частиною процесу розробки програми.

Необхідність постійної інтеграції**.** Системи безперервної інтеграції дозволяє автоматично створювати та тестувати свій проект кожного разу, коли ви надсилаєте оновлення для контролю версій. Безперервна інтеграція також запускає інструменти статичного аналізу коду, генерує файли APK і розповсюджує їх. *Lint* і *Checkstyle* — це інструменти, які забезпечують якість коду, тоді як Findbugs шукає помилки в коді.

Існує широкий вибір програмного забезпечення безперервної інтеграції, яке надає різні функції. Цінові плани можуть бути безкоштовними, якщо ваш проект має відкритий код. *Jenkins* — хороший варіант, якщо у вашому

розпорядженні є локальний сервер, з іншого боку, *Travis CI* — також рекомендований вибір, якщо ви плануєте використовувати службу безперервної інтеграції в хмарі.

2.2. Огляд основних архітектурних шарів. Вибір модулярного підходу

Діагарма основних шарів архітектури Android додатку зображена на рис. 2.8.

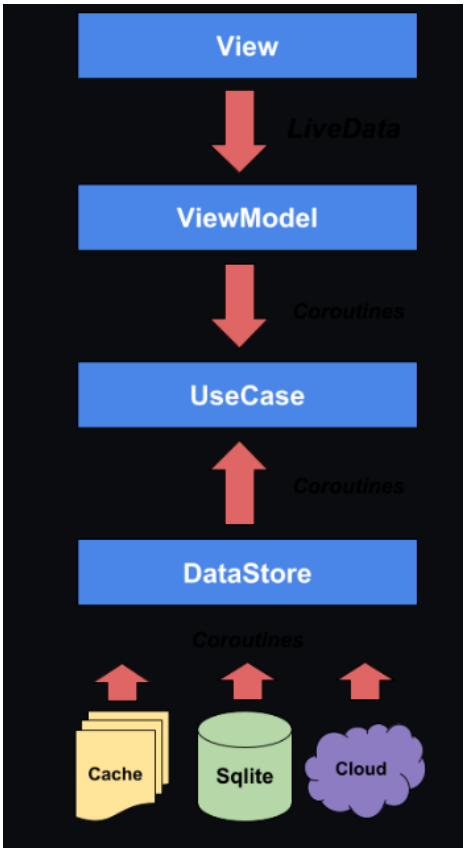


Рис. 2.8. Основні шари обраної архітектури Android

Рівень презентації. Презентаційний рівень було реалізовано за допомогою *MVVM* із *ViewModels*, що відкриває *LiveData*, які споживають *Views*. ViewModel нічого не знає про своїх споживачів. Він розкриває єдине джерело правди як *LiveData*, за яким споживачі можуть спостерігати, щоб отримати випущені події. Ці події складаються з сутностей, які загорнуті в клас *Result*, який використовує потужну функцію *Kotlin* із запечатаними

класами. Це дозволяє нам виражати різні стани окремих екранів програми в стислій та виразній формі. [17]

Рівень домену. Рівень домену містить випадки використання, які інкапсулюють одне й дуже конкретне завдання, яке можна виконати. Це завдання є частиною бізнес-логіки програми. *UseCase* розкриває веселий виклик оператора suspend, який повертає результат після виклику. *ViewModels* рівня презентації використовують *UseCases* для завершення бізнес-кейсу. Цей рівень також містить інтерфейси сховища та сутності домену. Це основні будівельні блоки програми та ті, які найменше змінюватимуться, коли щось зовнішнє змінюється

Рівень даних. Рівень даних реалізує інтерфейс сховища, який визначає рівень домену. Компоненти цього рівня забезпечують єдине джерело правдивих даних і приховують походження даних. Це забезпечує ефективне кешування за допомогою спеціальної реалізації кешу або/або бази даних *Sqlite*, не забруднюючи інші рівні деталями реалізації.

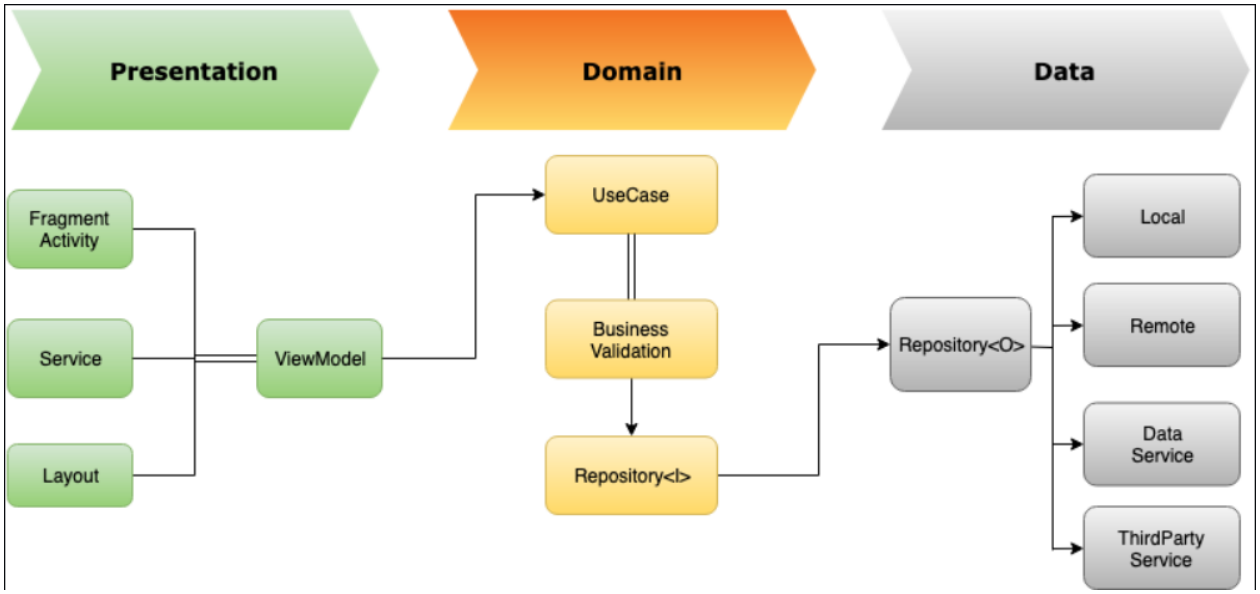


Рис. 2.9. Детальна репрезентація шарів архітектури Android

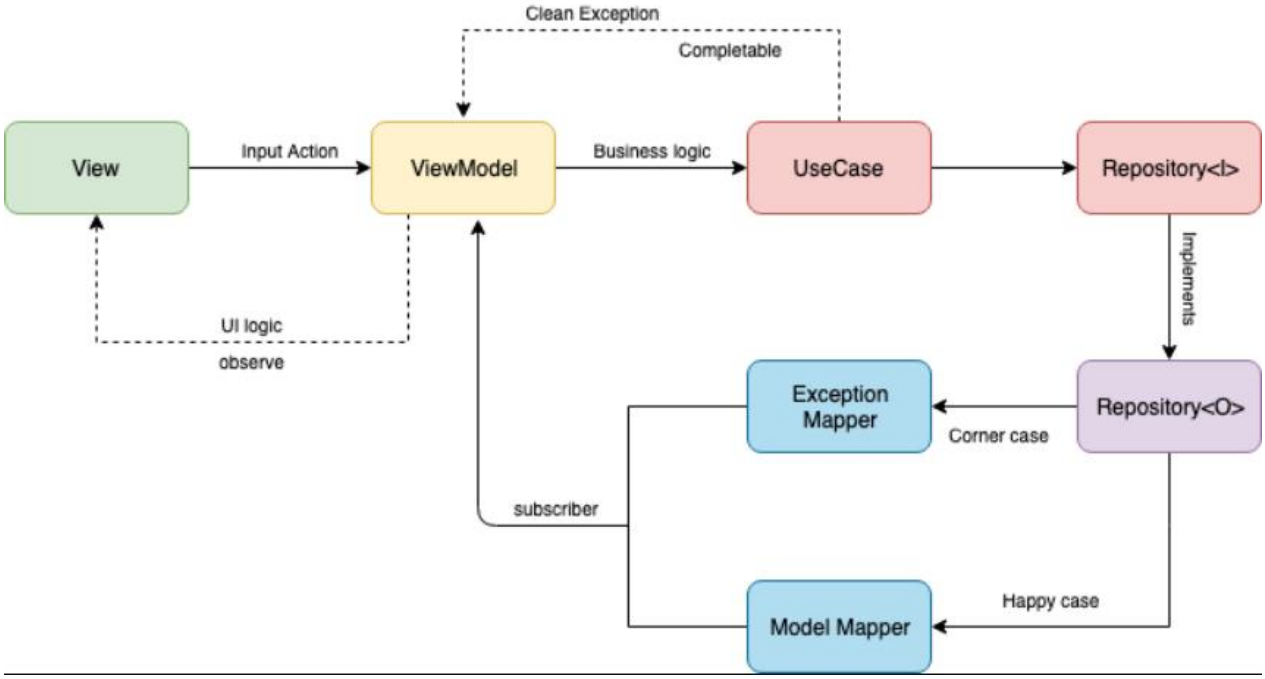


Рис. 2.10. Діаграма взаємодії архітектурних компонентів

Модулярність

Надважливу роль у розробці грає модульність і стратегія модулярності, яка використовується для створення модулів у програмі.

Модулярність — це практика розбиття концепції монолітної одномодульної кодової бази на слабко пов’язані самостійні модулі. [18]

Переваги модульності:

1. Це пропонує багато переваг, зокрема:

2. Масштабованість – у тісно пов’язаній кодовій базі одна зміна може викликати каскад змін. Належним чином модульний проект охоплюватиме принцип поділу проблем. Це, у свою чергу, надає учасникам більше автономії, а також забезпечує дотримання архітектурних шаблонів.

3. Увімкнення паралельної роботи. Модулярність допомагає зменшити конфлікти керування версіями та забезпечує ефективнішу паралельну роботу для розробників у великих командах.

4. Право власності – модуль може мати виділеного власника, який відповідає за підтримку коду та тестів, виправлення помилок і перегляд змін.

5. Інкапсуляція – ізольований код легше читати, розуміти, тестувати та підтримувати.

6. Скорочений час збірки. Використання паралельної та поступової збірки Gradle може скоротити час збірки.

7. Динамічна доставка. Модулярність є вимогою до Play Feature Delivery, яка дозволяє надавати певні функції вашої програми умовно або завантажувати їх на вимогу.

8. Багаторазове використання. Правильна модульність дає можливості для спільного використання коду та створення кількох програм на різних платформах на одній основі.

Підводні камені модулярізації:

1. Однак модульність — це шаблон, яким можна зловживати, і є деякі недоліки, про які слід пам’ятати під час модульування програми:

2. Забагато модулів – кожен модуль має накладні витрати, які виникають у вигляді збільшення складності конфігурації збірки. Це може призвести до збільшення часу синхронізації *Gradle* і призвести до витрат на поточне обслуговування. Крім того, додавання додаткових модулів збільшує складність налаштування проекту *Gradle* у порівнянні з одним монолітним модулем. Це можна пом’якшити, скориставшись стандартними плагінами, щоб вилучити багаторазово використовувану та придатну для компонування конфігурацію побудови в типово безпечний код *Kotlin*.

3. Недостатньо модулів – навпаки, якщо ваших модулів небагато, вони великі й щільно з’єднані, ви отримаєте ще один моноліт. Це означає, що ви втрачаєте деякі переваги модульності. Якщо ваш модуль роздутий і не має єдиної чітко визначеної мети, вам слід подумати про його розділення.

4. Занадто складний – тут немає жодної срібної кулі. Насправді не завжди має сенс модульувати ваш проект. Домінуючим фактором є розмір і відносна складність кодової бази. Якщо очікується, що ваш проект не

перевищить певного порогу, масштабованість і збільшення часу створення не застосовуватимуться.

Стратегія модулярізації

Важливо зазначити, що не існує єдиної стратегії модульного розвитку, яка б підходила для всіх проектів. Однак є загальні вказівки, дотримуючись яких можна отримати максимальну користь і мінімізувати недоліки.

Barebone-модуль — це просто каталог зі скриптом збірки Gradle всередині. Однак зазвичай модуль складатиметься з одного або кількох вихідних наборів і, можливо, колекції ресурсів або активів. Модулі можна створювати та тестувати незалежно. Завдяки гнучкості Gradle існує кілька обмежень щодо того, як ви можете організувати свій проект. Загалом, ви повинні прагнути до низького зчеплення та високої згуртованості.

Низький зв'язок – модулі повинні бути максимально незалежними один від одного, щоб зміни в одному модулі не мали або мали мінімальний вплив на інші модулі. Вони не повинні знати внутрішню роботу інших модулів.

Висока згуртованість. Модуль має містити набір коду, який діє як система. Він повинен мати чітко визначені обов'язки та залишатися в межах певної сфери знань.

2.3. Основні компоненти Android та їх безпека

Вектори атак на інформаційні системи на базі клієнтських додатків для ОС Android.

В рамках даного дослідження за модельну приймається така інформаційна система, що складається з додатку для *ОС Android* в якості клієнтського інтерфейсу, WEB-серверу, що представляє собою інтерфейс Backend, певної інформаційної інфраструктури в рамках backend та комунікаційного каналу за протоколом HTTP.Перш за все, слід окреслити основні напрями атак, що можуть бути здійснені на такі системи -їх перелік надається нижче:

1) На процеси серверної частини.

2) На клієнтський додаток.

3) На канал комунікації.

Далі надається дослідження складу кожної з цих категорій шляхом розгляду функціональних компонентів атакованого об’єкту, що обумовлюють ті чи інші вразливості. [19]

Вектори атак на серверну частину

ІС Інформаційна система, що передбачає використання мобільного додатку в якості клієнтського інтерфейсу, містить серверну частину, яка, найчастіше, реалізує основну частину функціональності. Така частина створюється за допомогою розповсюджених технологій проектування та, здебільшого, містить можливості для взаємодії з певним накопичувачем даних – в даному дослідженні архітектура і тип сховища даних не є принциповими, тому, з метою спрощення, для позначення серверного сховища даних інформаційної системи буде використовуватись термін «база даних». В даному випадку під процесами серверної частини додатку маються на увазі будь-які процеси в інформаційній системі, що забезпечуються функціоналом, реалізованим серверним ПЗ.

До таких, зокрема мають належати:

– аутентифікація, авторизація та розмежування доступу;

– контроль сеансів;

– бізнес-логіка;

– валідація даних;

– обробка помилок.

Дані елементи є характерними для побудови додатків, що оперують інформацією з обмеженим доступом. Реалізація кожного з цих елементів може містити вразливості, експлуатація яких може призвести до порушення

нормального функціонування системи та порушення параметрів інформації з обмеженим доступом.

У широкому розумінні, експлуатація вразливостей механізмів авторизації та аутентифікації може призвести до горизонтального або вертикального підвищення привілеїв. Вертикальне підвищення привілеїв обумовлює отримання користувачем більших прав, аніж ті, що надаються йому системою за нормального її функціонування. Горизонтальне підвищення привілеїв призводить до отримання прав, аналогічних тим, якими володіє атакуючий, але таких, що дозволяють персоніфікувати себе як іншого користувача системи. Обидві атаки призводять до високого ризику порушення параметрів інформації з обмеженими правами доступу – наприклад, отримання конфіденційної інформації, що належить іншому користувачеві, отримання інформації, що вимагає більш високого рівня прав доступу тощо.

Прикладами вразливостей, які викликають такі проблеми, можуть бути можливість типових SQL-ін’єкцій у механізмі авторизації під час доступу до бази даних, відсутність обмежень на невдалі спроби автентифікації (для запобігання атакам грубої сили) тощо. Уразливість, описана Access, заснована на проблемі в реалізації належної процедури перевірки ідентифікатора користувача (його сеансу та ролей) і наданих йому дозволів. Такі вразливості є критичними для систем, які обробляють інформацію з обмеженим доступом.

Конкретні приклади таких проблем включають IDOR (*InsecureDirectObjectReference*), відсутність або неправильну перевірку маркера доступу тощо. Уразливості контролю сеансу тісно пов’язані з процесами авторизації, оскільки вони в основному діють на одному наборі системних ресурсів. Порушення механізму контролю сеансу може призвести до захоплення сеансу іншого користувача, створюючи умови для такого захоплення та, у деяких випадках, ризик компрометації затримки. До таких уразливостей, насамперед, відносяться відсутність перевірки ідентифікаторів сесії під час аутентифікації (можлива реалізація атак SessionFixation),

використання ідентифікаторів сесії низької складності, тобто ідентифікаторів сесії, які можна вибрати за короткий проміжок часу, і жоден сеанс не прив'язаний до пристрою, і немає (або недостатнього) обмеження часу для дійсності сеансу.

Уразливості в бізнес-логіці здебільшого чітко не класифікуються, оскільки вони являють собою різнорідні проблеми в реалізації певних алгоритмів для даних у програмі і, таким чином, залежать від конкретної реалізації. Діри в бізнес-логіці часто дозволяють використовувати непередбачені функції під час проектування без порушення коректності програми. До таких проблем, зокрема, відносяться недостатня перевірка послідовності дій у процесі та порушення (*Racecondition*) у паралельній обробці процесу. Перевірка даних повинна застосовуватися до всіх даних, отриманих із ненадійних середовищ, де немає запланованої можливості для користувача змінити такі дані. Порушення цього процесу, його відсутність або недоліки можуть дозволити порушникам маніпулювати системою під час її робочого циклу. У свою чергу, це може призвести до незапланованих змін таких процесів. Порушення перевірки може мати необмежений вплив на систему, що є критичним при роботі з даними з обмеженим доступом. Найпоширеніші типи помилок перевірки включають усі типи ін'єкційних атак (*SQL*, команди *OC, XML, JavaScript* тощо). Неадекватна або неправильна обробка помилок може дозволити порушнику змінити правильний курс роботи програми, таким чином отримавши доступ до відкритої технічної інформації або отримавши певну можливість вплинути на систему. Основною метою системи обробки помилок є стабілізація системи за ненормальних умов, особливо ненормальних умов, які можуть бути створені штучно.

Вектори атак на клієнтські додатки Атаки на клієнтські мобільні додатки в рамках досліджуваної архітектури можуть стосуватися переважно тих даних, які обробляються локально. Підвищення привілеїв і доступ до даних інших користувачів

Ймовірність значно менша. Основні функціональні елементи, специфічні для клієнтської програми, включають:

– інтерфейс користувача;

– взаємодія між процесами;

– локальне зберігання даних.

На додаток до цих функціональних елементів, клієнтська програма також відповідає за встановлення безпечного каналу зв'язку з *server*, Проте ця проблема винесена в окрему категорію – канал зв’язку, інтерфейс взаємодії з користувачем в системі *Android* забезпечується за допомогою елементів програми *Activity і WebView*. Активіті — це абстракція робочого екрана програми з елементами інтерфейсу, призначеними для взаємодії з користувачем. Іншими словами, кожен елемент Активіті забезпечує інтерфейс для взаємодії користувача з системними даними.

Компонент *WebView* призначений для відображення веб-сторінок, часто подібних до інших типів веб-браузерів, і таким чином імітує більшість їхніх уразливостей. Уразливості, які можуть виникнути через інтерфейс, з яким взаємодіє користувач, включають вразливості, викликані елементами конфіденційної інформації, вбудованими в код програми, - це створює ризик їх виявлення за допомогою аналізу з використанням методів зворотного проектування. Іншим типом є можливість програмного доступу до елементів інтерфейсу іншими програмами, що працюють в операційній системі.

Взаємодія між процесами в операційній системі *Android* в основному реалізується за допомогою функції системного віртуального пристрою *Binder*. Обмін даними здійснюється за допомогою механізмів BroadcastReceiver і ContentProvider, які відповідно отримують і надають дані іншим процесам. Дані, отримані таким чином, є ненадійними, а отже, ризикують зловживати функціональністю та вносити неавторизовані зміни в робочий процес програми. Залежно від алгоритму, який використовується для обробки або

надання даних, можуть виникнути певні вразливості, головним чином пов’язані з перевіркою або логікою програми. Так, наприклад, помилка в реалізації

ContentProvider може призвести до витоку інформації в інші програми, тоді як помилка в *BroadcastReceiver* – непередбачена функція. Локальне накопичення даних у програмах ОС Android може здійснюватися систематично за допомогою файлів у локальних файлах (зокрема, спеціальних *SharedPreferences*), залежно від використання локальних баз даних (таких як *SQLite*) або системних сховищ (таких як *KeyChain*).

За звичайних обставин іншим процесам без явних дозволів заборонено отримувати доступ до даних конкретної програми, але існує багато вразливостей, які можуть призвести до витоку інформації через локальне сховище. Найпростішим прикладом такої проблеми є додаток, що зберігає дані на SD-карті - архітектура *ОС Android* надає менше гарантій щодо безпеки даних, що зберігаються на SD-карті.

Зовнішні носії, тому доступ до них можна отримати з інших процесів. Більшу можливість для експлуатації можуть мати шкідливі процеси з підвищеними привілеями, які так чи інакше можуть отримати доступ до захищених файлів додатків.

Уразливості клієнтських додатків можуть бути викликані іншими факторами, окрім окреслених функціональних компонентів, такими як уразливості використовуваних бібліотек або класичні недоліки нативного коду (якщо такі є). Однак найчастіше в цьому випадку експлуататорський вектор порушника не змінюється і залишається в межах визначених категорій.

Усі інші вразливості клієнтських програм із різними векторами експлойтів можна класифікувати як уразливості середовища.

Вектори атак на каналах зв'язку

Переважна більшість сучасних клієнт-серверних додатків *ОС Android*, особливо призначених для обробки інформації з обмеженим доступом,

спілкуються за допомогою протоколу *HTTP*. Використання цього протоколу без додаткового захисту саме по собі можна вважати серйозною вразливістю системи, оскільки немає можливості забезпечити конфіденційність і цілісність інформації під час передачі. Для забезпечення цих властивостей використовується протокол безпечного з'єднання - *HTTPS*. Однак, оскільки мобільні пристрої в більшості випадків працюють у ненадійному середовищі, яким може керувати потенційний зловмисник, існує обмежена можливість використання звичайного протоколу *HTTPS* – зловмисник може контролювати мережу DNS-серверів і перехоплювати клієнтські запити. Трафік кінцевої точки, тим самим підриваючи можливість перевірити сертифікат шифрування сервера. Щоб подолати цю проблему, використовується технологія SSLPinning, яка вбудовує набір довірених сертифікатів сервера безпосередньо в код програми, тим самим видаляючи етап перевірки сертифіката сервера через третю сторону з процесу встановлення безпечного з’єднання. Однак цей механізм може бути реалізований погано, знижуючи його захисну силу. На підставі наведених фактів можна виділити чотири типи можливих атак на канали зв’язку в системі досліджуваної архітектури:

1) MitMataka на незахищені канали зв’язку (HTTP).

2) MitMataka на мережу.

3) Використання лазівки у впровадженні технології SSLPinning для атаки на захищений канал.

4) Атака шифрування.

Концепція безпеки та архітектура інформаційної системи досліджуваного незалежного програмного забезпечення в рамках дослідження, як показано на рис. 2.11.

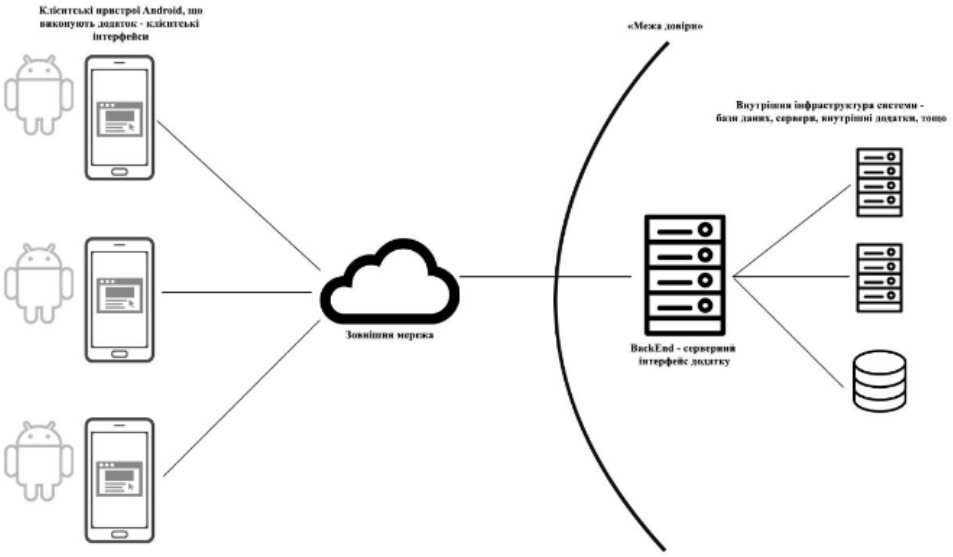


Рис. 2.11. Схема, що відображає типову структуру інформаційної системи клієнт-серверної архітектури з Android-додатками в якості клієнтів

Ця архітектура зберігає всю конфіденційну інформацію системи в сховищі даних на стороні сервера. Надання доступу до даних і логіка обробки реалізована в серверній програмі, яка реалізує API, який підключається до клієнтських програм через канал зв'язку (HTTPS). Слід зазначити, що в реальному середовищі кількість клієнтських програм буде більшою за 1, а архітектура системи та середовище, в якому працюють програми, невідомі, тому їм не можна довіряти. Крім того, слід зазначити, що цілісність програми не може бути гарантована, оскільки її виконуваний код може бути проаналізований і змінений користувачем.

У свою чергу, це означає, що сама програма може не мати відомої архітектури, що також ставить її на межу ненадійного середовища. Таким чином, обмеження довіри в досліджуваній інформаційній системі можна ідентифікувати до точки входу *API* сервера - всі інші елементи належать до ненадійного середовища. Таким чином, інтерфейс клієнта може бути будь-яким, сумісним з *API* сервера та каналом зв'язку.

Оскільки метою є визначення основних проблем безпеки такої архітектури, необхідно визначити відповідну модель порушника. Ці моделі

абстрагуються від мотивації та інструментарію зловмисника, оскільки ці параметри не впливають на архітектуру — несанкціонований доступ до конфіденційних даних вважається метою зловмисника. A. Зловмисник – це сторонній зловмисник, який не має доступу до певних авторизованих клієнтських програм, доступу до клієнтських пристроїв і доступу до серверних частин системи. При цьому зловмисник може проаналізувати алгоритм роботи типового клієнтського додатку та його взаємодію з сервером.

У рамках цієї моделі також розглядається можливість отримання певного контролю над мережевим середовищем. Очевидно, що ця модель є, ймовірно, найбільш загальною, оскільки вона не передбачає будь-яких конкретних умов для її реалізації; B. Зловмисник має певний рівень доступу до клієнтського пристрою, який прихований від самого клієнта, без доступу до сервера частину та прямий доступ до програми.

Ця модель надає порушнику попередній доступ до клієнтського пристрою на рівні програмного забезпечення, наприклад, за допомогою програми троянського коня, що обмежує можливості для таких сценаріїв; C. Порушник представляє клієнтську систему і може змінити клієнта застосування будь-яким способом.

Доступ до серверної частини системи здійснюється на рівні звичайного клієнта. Залежно від мети програми ця модель може бути цілком розумною (відкрита реєстрація в системі) або складною для реалізації, що відносить її до категорії інсайдерів (закрита система, перевірка реєстрації). Розглядаючи систему, призначену для використання широким колом користувачів, припустімо перший варіант;

D. Зловмисник має повний фізичний доступ до пристрою клієнта. Для реалізації цієї моделі необхідне безпосереднє володіння пристроєм, який вимагає від порушника певних дій поза системою інформаційного простору (викрадення, конфіскація тощо), а отже вимагає даних інших користувачів. Зрозуміло, що принципової різниці у векторах експлойту для обраного типу

зловмисника в цьому випадку немає. Зрозуміло, що поверхня атаки зловмисника

А звужує поверхню атаки до серверної інформаційної системи сервера та каналу зв'язку між нею та зловмисником. клієнтський додаток. Завдяки знанню алгоритму клієнтської програми, такий зловмисник має уявлення про алгоритм, який використовується для обробки клієнтських запитів на стороні сервера, і тому може проводити дослідження на предмет уразливості. Атаки Man-in-the-middle можуть бути реалізовані, якщо канал зв'язку відкритий або не захищений належним чином, поверхнею атаки Violator B є передусім інтерфейс для міжпроцесної взаємодії програми, механізм доступу до носія інформації (внутрішній і зовнішні диски), а система ОС називається Android.

До таких внутрішніх частин операційної системи можна отримати доступ, наприклад, через троянську програму, встановлену на пристрої користувача. Також можуть бути реалізовані атаки, такі як Tapjcking, Man-in-the-Disk, запис операцій клієнта (Keylogging). Порушник типу C має привілейований доступ до системи, тобто отримує певний рівень доступу до своїх даних в інформаційній системі. Це означає, що його поверхня атаки — це функціональний простір, доступний авторизованим користувачам, тобто функціональний простір за межами бар'єру безпеки першого рівня. У поєднанні з розумінням алгоритму взаємодії клієнтської програми з серверною частиною цей доступ дозволяє зловмисникам аналізувати безпеку серверної частини системи, минаючи першу лінію механізмів захисту.

Передумова полягає в тому, що пристрій не має блокування на системному рівні. Таким чином, залежно від стану клієнтської програми можливі два випадки:

1) Отримання доступу до авторизованої сесії користувача.

2) Доступ до системи та програми лише без авторизованої сесії. Несанкціонований доступ до конфіденційних даних клієнтів і передача в

Model C. У другому випадку зловмисник отримує доступ до моделі B з більш високими привілеями.

Вплив кожного окресленого вектора експлуатації на інформацію в системі залежить від конкретної архітектури та типу даних, що обробляються, тому в цьому аналізі наводяться лише загальні твердження щодо поверхні атаки для кожного типу порушника.

2.4. Побудова модульної архітектури додатку

Архітектура

Розділивши проблему на менші та легші для вирішення підпроблеми, ми можемо зменшити складність проектування та обслуговування великої системи. Кожен модуль є незалежним збірним блоком, який служить чіткій меті. Ми можемо розглядати кожну функцію як багаторазовий компонент, еквівалент мікросервісу або приватної бібліотеки.

Кожен модуль має чіткий API. Класи, пов’язані з функціями, живуть у різних модулях, і на них не можна посилатися без явної залежності модуля. Ми суворо контролюємо, що доступне для інших частин вашої кодової бази. функції можна розвивати паралельно, напр. різними командами кожна функція може бути розроблена ізольовано, незалежно від інших ознак швидший час складання.

Типи модулів і залежності модулів

У програмі є три види модулів:

− *app module* - це основний модуль. Він містить код, який об’єднує кілька модулів разом (клас, налаштування ін’єкції залежностей, *NavHostActivity* тощо) і фундаментальну конфігурацію програми (конфігурація модернізації, налаштування необхідних дозволів, спеціальний клас програми тощо);

− *module feature_x* — найпоширеніший тип модуля, що містить увесь код, пов’язаний із заданим функціоналом. Ділитися деякими об’єктами або кодом лише між модулями функцій (наразі в додатку немає таких модулів);

− *feature_base* модулі, від яких модулі функціоналу залежать для спільного використання спільного коду.

Дана діаграма представляє залежності між модулями проекту (підпроектами Gradle).

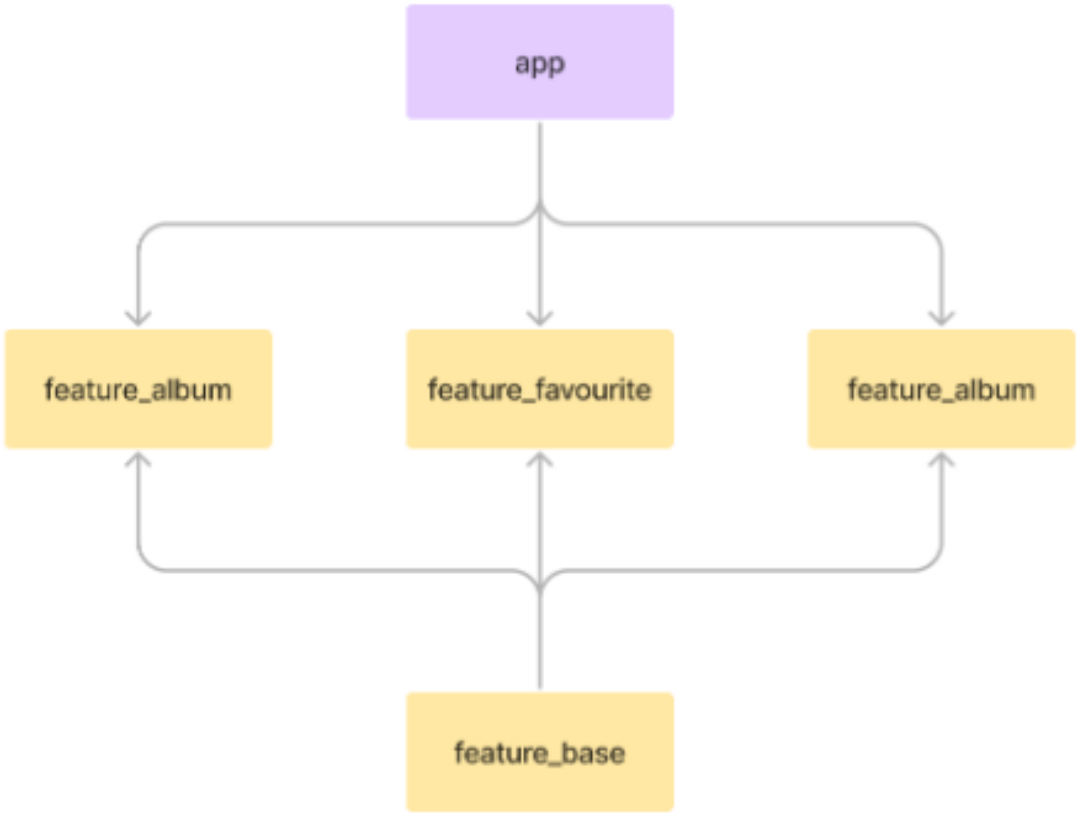


Рис. 2.12. Зображення розділення на модулів додатку

Кожен модуль має чіткий API. Класи, пов’язані з функціями, живуть у різних модулях, і на них не можна посилатися без явної залежності модуля.

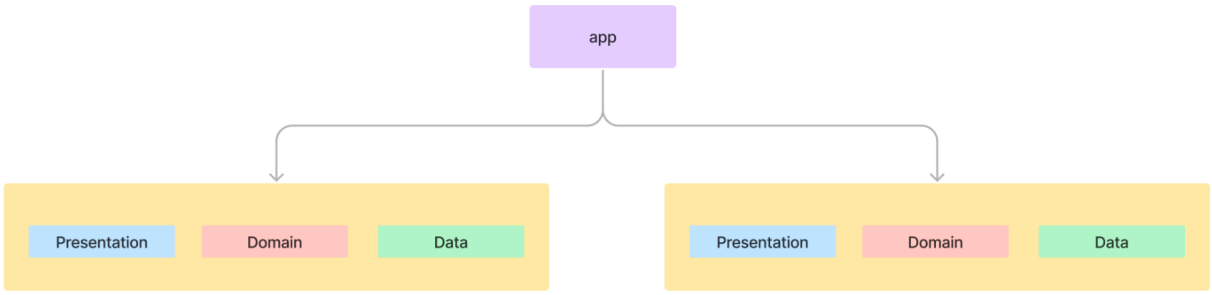


Рис. 2.13. Зображення внутрішньої папкової архітектури модуля

Структура модулів програми та бібліотеки_x дещо відрізняється від структури модуля функцій. Кожен модуль функцій містить нерівневі компоненти та 3 рівні з окремим набором обов'язків.

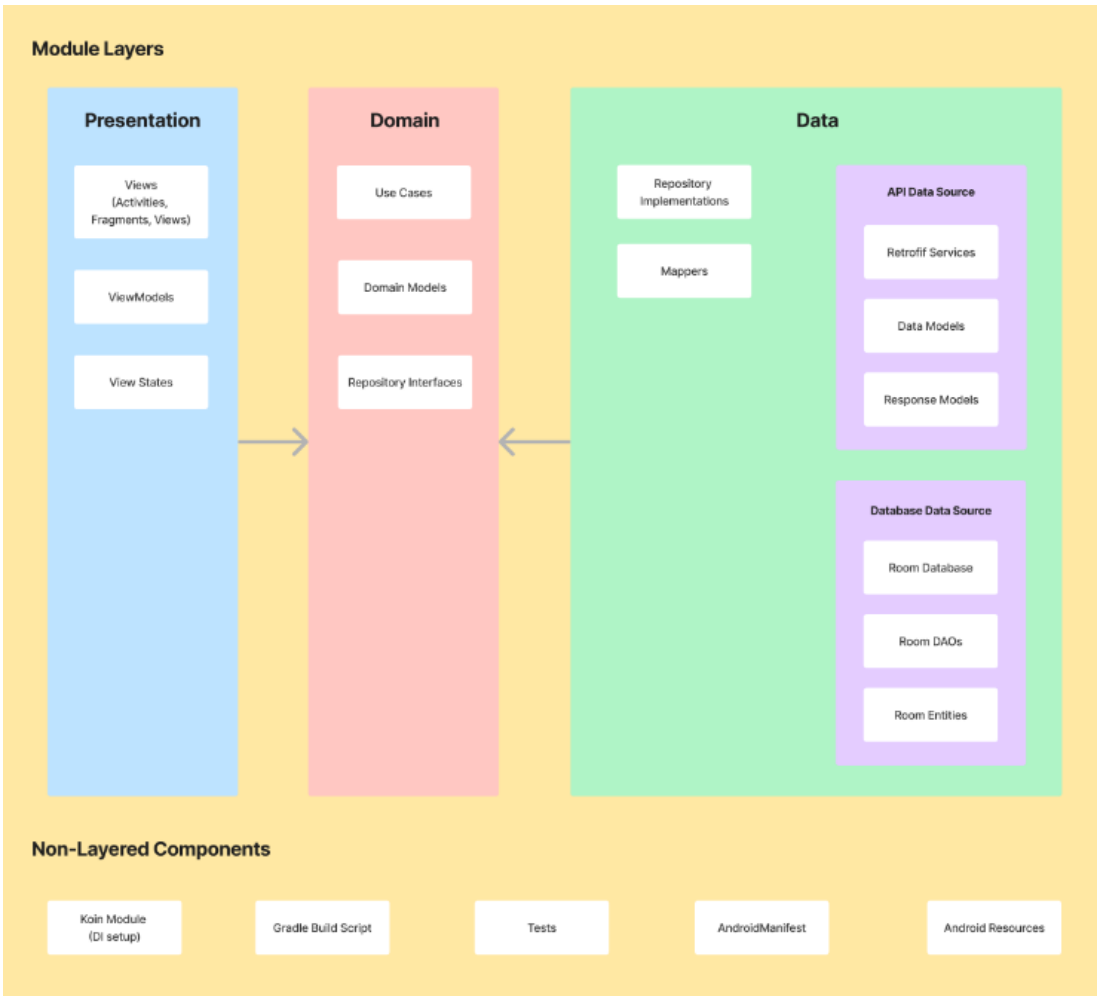


Рис. 2.14. Зображення взаємодії внутршньомодульоної архітектури

Рівень презентації

Цей шар найбільш наближений до того, що бачить користувач на екрані. Презентаційний рівень поєднує шаблони *MVVM* і *MVI*:

− *MVVM* - Jetpack ViewModel використовується для інкапсуляції загального стану інтерфейсу користувача. Він розкриває стан через спостережуваний власник стану (Котлін потік);

− *MVI* — дія змінює загальний стан інтерфейсу користувача та створює новий стан для перегляду через Kotlin Flow.

Загальний стан є єдиним джерелом істини для кожного екрану. Це рішення ґрунтується на принципах односпрямованого потоку даних і *Redux*.

Такий підхід полегшує створення узгоджених станів. Стан збирається за допомогою методу *collectAsUiStateWithLifecycle*. Збір потоків відбувається з урахуванням життєвого циклу, тому ресурси не витрачаються даремно.

Заявлене анотовано незмінною анотацією, яка використовується Jetpack Compose для оптимізації композиції.

Компоненти:

− Екран (фрагмент) — спостерігає за загальним станом перегляду (через *Kotlin Flow*). Стан трансформації *Compose* (випромінюваний *Kotlin Flow*) у інтерфейс програми споживає стан і перетворює його на інтерфейс програми (через *Jetpack Compose*). Передавати взаємодії користувача в *ViewModel*. Перегляди важко перевірити, тому вони повинні бути максимально простими;

− *ViewModel* — передає (через Kotlin Flow) зміни стану перегляду та взаємодію з користувачем (ці моделі перегляду не є просто класами *POJO*);

− *ViewState* - загальний стан для окремого перегляду;

− *StateTimeTravelDebugger* - реєструє дії та переглядає переходи станів для полегшення налагодження;

− *NavManager* — єдиний елемент, який полегшує обробку всіх подій навігації всередині NavHostActivity (замість окремо, у кожному поданні).

Рівень домену

Це основний рівень програми. Рівень домену не залежить від інших рівнів. Це дозволяє зробити моделі домену та бізнес-логіку незалежними від інших рівнів. Іншими словами, зміни в інших рівнях не вплинуть на доменний рівень, наприклад. зміна бази даних (рівень даних) або екранного інтерфейсу користувача (рівень презентації) в ідеалі не призведе до будь-яких змін коду на рівні домену.

Компоненти:

UseCase - містить бізнес-логіку

DomainModel – визначає основну структуру даних, які використовуватимуться в програмі. Це джерело правди щодо даних програми.

Інтерфейс репозиторію – необхідний для підтримки незалежності доменного рівня від рівня даних (інверсія залежностей).

Рівень даних

Керує даними програми. Підключається до джерел даних і надайте дані через репозиторій на рівень домену, наприклад. отримувати дані з Інтернету та зберігати дані в кеші диска (коли пристрій офлайн).

Компоненти

Репозиторій надає дані доменному рівню. Залежно від структури програми та якості зовнішнього репозиторію API також можна об’єднувати, фільтрувати та перетворювати дані. Ці операції спрямовані на створення високоякісного джерела даних для доменного рівня.

Mapper – зіставляє модель даних із моделлю домену (щоб зберегти рівень домену незалежним від рівня даних).

Рівень даних містить неявний рівень під назвою джерело даних, що містить усі компоненти, пов’язані з маніпулюванням даними даного джерела даних. Додаток має два джерела даних - *Retrofit* (мережа) і *Room* (локальне сховище):

- *Retrofit Service* – визначає набір кінцевих точок API

− *Retrofit Response Model* – визначення мережевих об’єктів для заданої кінцевої точки (модель верхнього рівня для даних складається з *ApiModels*)

− *Retrofit Api Data Model* – визначає мережеві об’єкти (підоб’єкти моделі відповіді)

− База даних *Room* – постійна база даних для зберігання даних програми

− *Room DAO* - взаємодія зі збереженими даними

− *Room Entity* - визначення збережених об'єктів

І моделі даних *Retrofit API*, і сутності кімнат містять анотації, тому дана структура розуміє, як розбирати дані в об’єкти.

Мета цього підходу полягала в тому, щоб знайти правильний баланс між надмірною модуляцією відносно невеликої програми та використанням цієї можливості, щоб продемонструвати шаблон модуляції, який підходить для набагато більшої кодової бази, ближче до реальних програм у виробничих середовищах.

Цей підхід обговорювався зі спільнотою Android і розвивався з урахуванням їхніх відгуків. Однак із модуляцією немає однієї правильної відповіді, яка робить усі інші неправильними. Зрештою, існує багато способів і підходів до модульної структури програми, і рідко один підхід відповідає всім цілям, кодовим базам і вподобанням команди. Ось чому попереднє планування та врахування всіх цілей, проблем, які ви намагаєтеся вирішити, майбутньої роботи та прогнозування потенційних сходинок — все це вирішальні кроки для визначення найкращої структури для ваших власних унікальних обставин.

Це загальна інструкція, яку я вважаю найкращою для даного проекту, і пропонуємо її як приклад, який можна надалі змінювати, розширювати та надбудовувати. Одним із способів зробити це було б ще більше підвищити деталізацію кодової бази. Деталізація – це ступінь, до якого ваша кодова база складається з модулів. Якщо ваш рівень даних невеликий, добре зберігати його в одному модулі. Але як тільки кількість сховищ і джерел даних почне

зростати, можливо, варто розглянути можливість розділити їх на окремі модулі.

ВИСНОВКИ ДО РОЗДІЛУ 2

У розділі ми обрали на основі проаналізованого матеріалу найкращий підхід до побудови додатку, встановили базову архітектуру з розділенням на модулі базовані на функціоналі додатку.

Такий спосіб дає неймовірний приріст по швидкості збірки проекту, особливу перевагу можна побачити з наявними бібліотеками, які мають багато кодогенерації, отже не будуть її виконувати, якщо файли у модулі не змінювались.

Функціонал спроектований та декомпозований на частини, отже на проектом легко може працювати декілька чоловік паралельно, розробляючи програмний продукт та маючи свої зони відповідальності, які не залежні одна від одної, що робить командну роботу максимально ефективною.

Обрано тематику додатку то його приблизний вигляд, який потрібно буде розробляти. Описано роботу додатку та взаємозв’язок модулів, їх рощирюваність, архітектурний підхід з розділенням на шари для написання зрозумілого чистого коду, який легко розширювати та тестувати.

На діаграмі нижче показано потік даних програми, коли користувач взаємодіє з екраном списку альбомів:

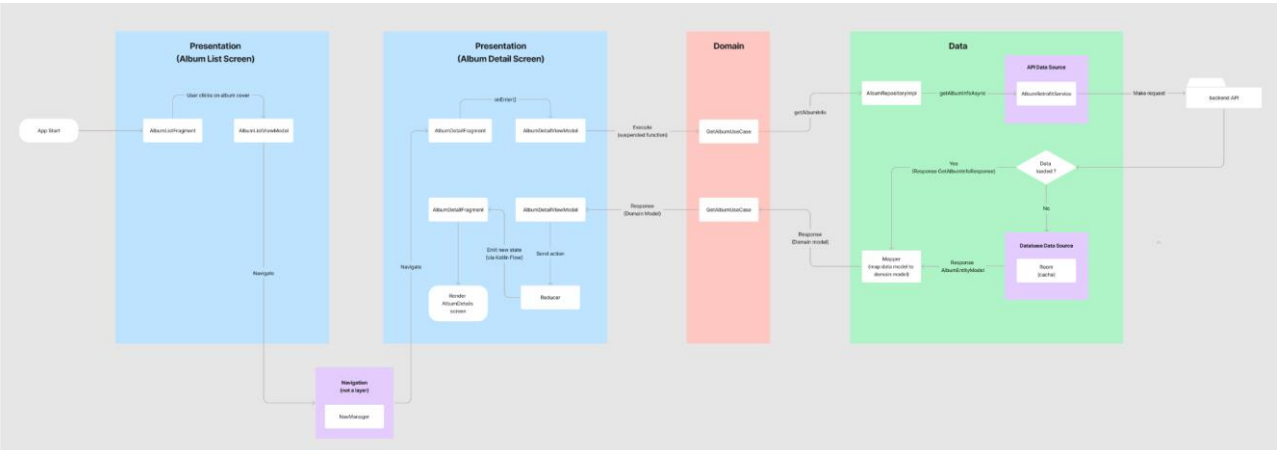


Рис. 2.15. Потік даних програми

РОЗДІЛ 3

РОЗРОБКА ТА ТЕСТУВАННЯ ДОДАТКУ

3.1. Опис роботи додатку додатку

Для розробки додатку прикладу була обрана музична предметна область: андроїд-вітрина, що відображає інформацію про музичні альбоми. Дані завантажуються з *Last.fm Music Discovery API*. Це є публічним інтерфейсом для використання і створення на його основі додатків. [20]

Додаток буде мати кілька екранів, розташованих у кількох модулях функцій:

- екран списку альбомів - відображає список альбомів;
- екран деталей альбому - відображення інформації про вибраний альбом;
- екран профілю - порожній (ще не реалізований);
- екран улюблених пісень - порожній (ще не реалізований).

Створення проекту починаєтсья з головного app модулю, де буде розташована головний контейнер екрану, який буде містити в собі навігацію та подальші екрнани, цей модуль буде корневим.

Також потрібно винести версії бібліотек у окремий файл, так як ці бібліотеки будуть використовуватись у різних модулях, за допомогою одного файлу ми позбудемось повторювання коду і по всьому проекту буде реалізована легка реалізація оновлення версій бібліотек. Для того щоб цей модуль був корневим потрібно реалізувати два основні файли: Application та NavHost

Файли наведені на рис. 3.1 показують структуру нашого першого корневого модуля.

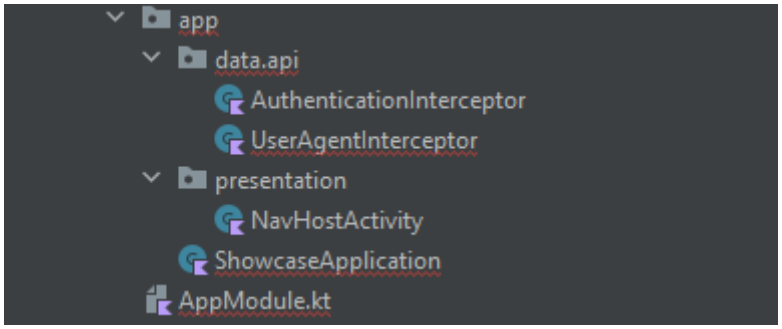


Рис. 3.1. Структура модулю app

Вигляд файлу Manifest який є обов'язковим для кожного модуля показано на рис 3.2, на якому детально видно, що йде запит на дозвіл інтернету користувача та використовується одне активіті у додатку. Також видно назву класу, що реалізує клас Applcation, який є найвищою точкою входу у додаток, створюється першим і існує протягом життя усього додатку.

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

<!--
Data backup:
dataExtractionRules - API 31+
allowBackup and fullBackupContent - API < 31
-->
<application
        android:name=".app.ShowcaseApplication"
        android:allowBackup="false"
        android:fullBackupContent="false"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.Showcase"
        android:usesCleartextTraffic="true">

    <activity
            android:name=".app.presentation.NavHostActivity"
            android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <action android:name="android.intent.action.VIEW" />

            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

</application>
```

Рис. 3.2. Маніфест кореневого модуля app

Відповідно класи *Interceptor* додають у запит на сервер юзера та необхідну метадату для доступу для даних. Клас *NavHost* є класом активіті, що створ. Корінний екран у всьому додатку та відповідає за навігацію між екранами у ньому. Кодова складова класу представлена на рис. 3.3.

```
class ShowcaseApplication : Application() {

    override fun onCreate() {
        super.onCreate()

        initKoin()
        initTimber()
        initDynamicColorScheme()
    }

    private fun initDynamicColorScheme() {
        // Apply dynamic colors to all Activities, Fragments, Views
        // (Material 3 library helper class)
        DynamicColors.applyToActivitiesIfAvailable( application: this)
    }

    private fun initKoin() {
        GlobalContext.startKoin { this: KoinApplication
            androidLogger()
            androidContext(this@ShowcaseApplication)

            modules(appModule)
            modules(baseModule)
            modules(featureFavouriteModules)
            modules(featureAlbumModules)
            modules(featureProfilesModules)
        }
    }

    private fun initTimber() {
        if (BuildConfig.DEBUG) {
            Timber.plant(Timber.DebugTree())
        }
    }
}
```

Рис. 3.3. Кореневий клас Application

З малюнку видно, що у класі відбувається уся необхідна ініціалізація бібліотек, що є важливими і використовуються у модулі. Це є критично важливе місце, адже дуже багато помилок відбувається саме при неправильній ініціалізації бібліотек. Дотримання цього шаблону нівелює шанси помилок, адже все відбуєтсья одразу при запуску додатку, все розміщуєтсья в окремих факлах і здійснюється у одному потоці в результаті чого, нічого не може

завадити проініціалізувати класи та механізм є зручним для додавання нових ініціалізацій.

Наступний по черзі створення але не менш важливий модуль базових реалізацій у проекті. Так як використовувати основні переваги ООП є критично важливим при проектванні будь якої програми, наслідування є одним із ключових при написанні цільної розширюваної системи, що надає багато можливостей у подальшому маніпулюванні об’єктами, покращує зручність їх написання та зменшує кількість коду, який би повторювався без цих базових копонентів. Пакетна структура модуля зображена на рис. 3.4.

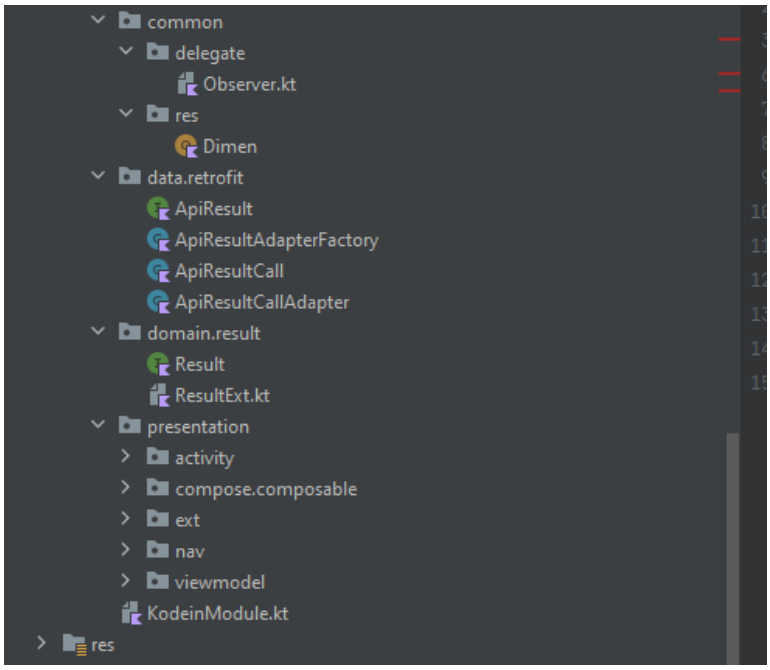


Рис. 3.2. Пакетна структура модуля базових реалізацій

Як можна побачити з малюнків основним розділенням у модулі є папки: common, *data/retrofit, domain/result, presentation*. Це є базові класи, які кожен наступний створений функціонал буде використовувати реалізуючи ці класи чи інтерфейси.

Папка *common* містить дві підпапки, у папці *delegates* миможемо покласти будь який базовий функціонал делегаційних класів. Делегаційні класи, це

класи яким ми перенаправляєм якусь роботу, щоб її виконав інший клас. Зв’язок між класами тримається за допомогою інтерфейсів.

Папка *res* містить в собі поки один файл з базовими розмірами які є у проекті

Папка retrofit містить у собі базові класи для роботи з запитами на сервер, тому є дуже важливим компонентом системи. На рис. 3.5 приведено базову повенку класу для роботи з сервером:

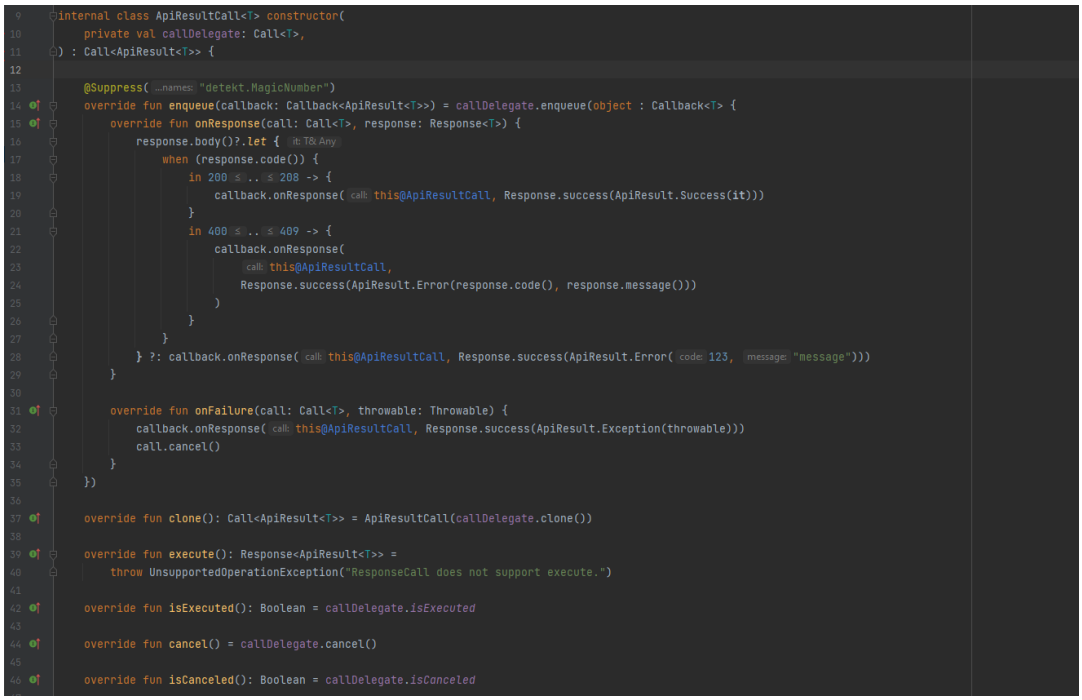


Рис. 3.5. Базова реалізація поведінки додатку з сервером

Для побудови такого архітектурного рішення використовуються два шаблони проектування: Адаптер та Фабрика, які є надпопулярними при роботі з цими класами.

У папці *domain* міститься результат запиту, який уже є зрозумілим для розробника, що нам потрібна буде відображати. Клас зображений на рис. 3.6. дозволяє поділити результат запиту на вдалий та помилку. При вдалому ми покажемо результат запиту, при невдалому – помилку.

```
sealed interface Result<out T> {
    data class Success<T>(val value: T) : Result<T>
    data class Failure(val throwable: Throwable? = null) : Result<Nothing>
}
```

Рис. 3.6. Базовий клас результату запиту

Папка *presentation* містить в собі базові класи презентації, тобто нашого *UI* та архітектурним компонентом *Viewmodel*. Ці реалізації дають змогу при використанні використовувати одні і ті ж налаштування для нашого інтерфейсу користувача, що є безпечним та безпомилковим способом написання коду, адже всі проблеми пов'язані з відомими Android проблемами можна вирішити у базових класах і в реалізаціях залишиться лише конкретний зміст того чи іншого екрану. Зміст класу базової активіті на рис. 3.7.

```
abstract class BaseActivity(@LayoutRes contentLayoutId: Int) : AppCompatActivity(contentLayoutId) {

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

        supportActionBar?.hide()

        // The window will not be resized when virtual keyboard is shown (bottom navigation bar will be
        // hidden under virtual keyboard)
        window.setSoftInputMode(WindowManager.LayoutParams.SOFT_INPUT_ADJUST_NOTHING)

        Timber.d( message: "onCreate ${javaClass.simpleName}")
    }
}
```

Рис. 3.7. Базовий клас активіті

З точки зору дизайну дотримувалися рекомендацій Android Material Design, вичерпний посібник щодо візуального дизайну, дизайну руху та взаємодії між платформами та пристроями. Надання проекту таким чином чудового досвіду користувача (UX) та інтерфейсу користувача (UI). Щоб дізнатися більше про найкращі практики UX, перейдіть за посиланням.

Крім того, реалізовано підтримку темної теми з наступними перевагами:

Може значно зменшити енергоспоживання (залежно від технології екрану пристрою).

Покращує видимість для користувачів зі слабким зором і тих, хто чутливий до яскравого світла.

Також сюди винесено корисні додаткові функції, такі як використання картинки за замовчуванням під час завантаження картинки з серверу, що зображена на рис. 3.8.

```
private val PLACEHOLDER_IMAGES = listOf(
    R.drawable.image_placeholder_1,
    R.drawable.image_placeholder_2,
    R.drawable.image_placeholder_3
)

@Composable
fun PlaceholderImage(
    url: Any?,
    contentDescription: String?,
    modifier: Modifier = Modifier,
) {
    Surface(modifier = modifier) {
        val randomPlaceHolder by rememberSaveable {
            mutableStateOf(PLACEHOLDER_IMAGES.random())
        }

        val model = ImageRequest.Builder(LocalContext.current).data(url).crossfade( enable: true).build()

        AsyncImage(
            model = model,
            contentDescription = contentDescription,
            placeholder = painterResource(randomPlaceHolder)
        )
    }
}

```

Рис. 3.8. Релізація завантаження картинок

3.2. Розробка функціональних модулів додатку

Наступний крок буде уже перехід до самого цікавого – розробки функціоналу у додатку. Додаток складатиметься з меню вибора 3 екранів. А саме: екран профілю, екран улюблених альбомів та екран усіх доступних альбомів.

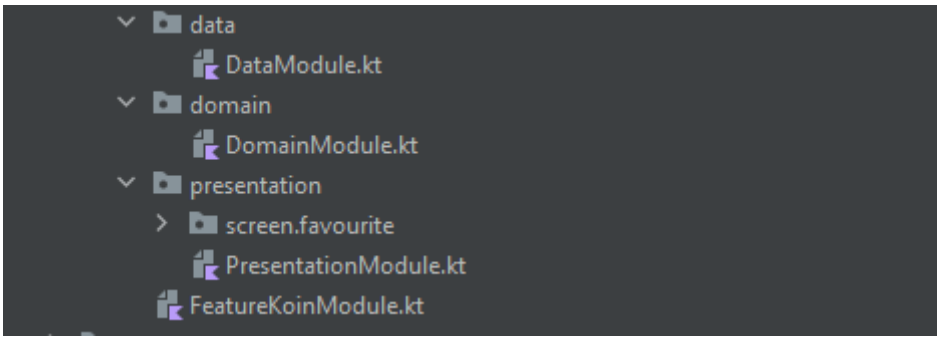


Рис. 3.9 Модуль улюблених альбомів

Цей функціонал буде у розробці і поки ми залишимо лише базові реалізації, які будуть повідомляти користувачу, що функціонало ще не в доступі, а саме використаємо екран «заглушку», який представлений на рис. 3.10 .

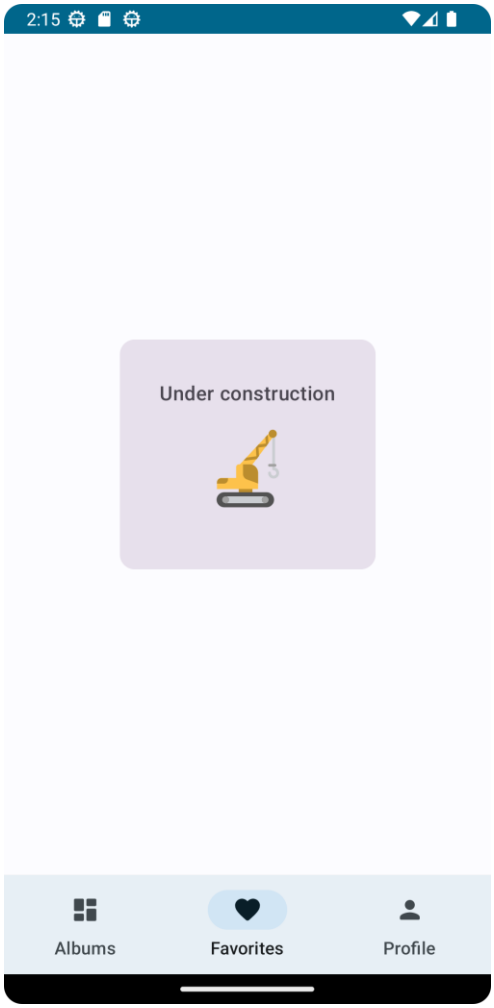


Рис. 3.10 Інтерфейс кеористувача для функціоналу, який ще не реалізований у програмі

Відповідно функціонал профілю користувача також у розробці, для створено схожий модуль у якому такого є екрна у розробці. Цей зручний механізм дозволяє повідомляти користувачу, що в майбтньому буде такий функціонал, хоча він може бути реалізований уже і в релізі у додатку. За допомогою реалізації *feature flag* на сервері можна досягти додання цього екрану без нового релізу, а коли захоче сам автор, що є дуже зручним для проведення тестів між контрольними групами користувачів для визначення актуальності доданого функціоналу.

Найбільш розроблений функціонал буде саме огляду усіх музичних альбомів. Структура цього екрану представлена на рис. 3.11 .

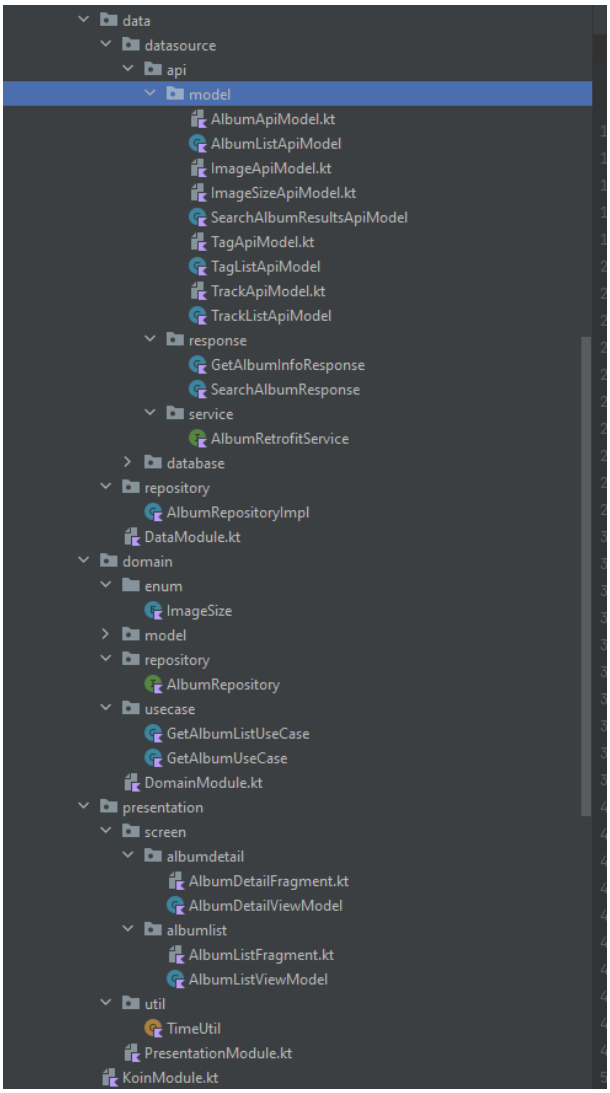


Рис. 3.11. Структура функціонального модулю «Огляд альбомів»

У папці *api* ми маємо ще три папки, вміст кожної з яких відповідає за зав'язок із сервером. Папка *model* містить у собі класи моделі, що передає нам сервер у відповіді на запит, що наавередено на рисунку Рис. 3.12 .

```kotlin
@Serializable
internal data class AlbumApiModel(
    @SerialName("mbid") val mbId: String? = null,
    @SerialName("name") val name: String,
    @SerialName("artist") val artist: String,
    @SerialName("image") val images: List<ImageApiModel>? = null,
    @SerialName("tracks") val tracks: TrackListApiModel? = null,
    @SerialName("tags") val tags: TagListApiModel? = null,
)

internal fun AlbumApiModel.toEntityModel() =
    AlbumEntityModel(
        mbId = this.mbId ?: "",
        name = this.name,
        artist = this.artist,
        images = this.images?.mapNotNull { it.toEntityModel() } ?: listOf(),
        tracks = this.tracks?.track?.map { it.toEntityModel() },
        tags = this.tags?.tag?.map { it.toEntityModel() }
    )

internal fun AlbumApiModel.toDomainModel(): Album {
    val images = this.images
        ?.filterNot { it.size == ImageSizeApiModel.UNKNOWN || it.url.isBlank() }
        ?.map { it.toDomainModel() }

    return Album(
        mbId = this.mbId,
        name = this.name,
        artist = this.artist,
        images = images ?: listOf(),
        tracks = this.tracks?.track?.map { it.toDomainModel() },
        tags = this.tags?.tag?.map { it.toDomainModel() }
    )
}
```

Рис. 3.12. Модель альбома для зв'язку с сервером

Тут видно що клас ми помічаємо анотацією *@Serializable* та використовуємо спеціальну аннотацію *@SerializeName* для помітки полів класу, значення яких повинно відповідати іменам полів на сервері. Такж клас

містить локальні функції перетворювачі у нашу модель інтерфейсу користувача, що є зручним та читабильним механізмом виклику у коді.

Так як телефони різних виробників і різні типи телефонів мають різну густину пікселів на екрані, створено клас, який запитує з сервера розмір картинки, відповідно до густини пікселів користувача, щоб вона не займала забагато місця або не була мильною, що зображено на Рис.3.13 .

```
@Serializable
internal enum class ImageSizeApiModel {

    @SerialName("medium")
    MEDIUM,

    @SerialName("small")
    SMALL,

    @SerialName("large")
    LARGE,

    @SerialName("extralarge")
    EXTRA_LARGE,

    @SerialName("mega")
    MEGA,

    @SerialName("")
    UNKNOWN
}

internal fun ImageSizeApiModel.toDomainModel() = ImageSize.valueOf(this.name)

internal fun ImageSizeApiModel.toEntityModel() =
    ImageSizeEntityModel.values().firstOrNull { it.ordinal == this.ordinal }
```

Рис. 3.13. Клас, що відповідає за завантаження розміру картинок

Запит на сервер відбувається за допомогою спеціальної бібліотеки, і часто може містити в собі якісь параметри, які ми будемо передавати та повертає нам якийсь результат, який потрібно отримати у тому вигляді, у якому об’єкти існують на серверній частині, тобто згідно з контракту.

Функції обов’язково повинні бути помічені ключовим словом *suspend* для використання механізму корутин, що дозволить використовувати багатопотоковість телефона і виконувати запити у іншому потоці, щоб не

зупиняти головний, на якому відбувається рендер інтерфейсу користувача. Вигляд формування запитів на сервер зображено на Рис. 3.14:

```
internal interface AlbumRetrofitService {

    @POST("./?method=album.search")
    suspend fun searchAlbumAsync(
        @Query("album") phrase: String,
        @Query("limit") limit: Int = 60,
    ): ApiResult<SearchAlbumResponse>

    @POST("./?method=album.getInfo")
    suspend fun getAlbumInfoAsync(
        @Query("artist") artistName: String,
        @Query("album") albumName: String,
        @Query("mbid") mbId: String?,
    ): ApiResult<GetAlbumInfoResponse>
}
```

Рис. 3.14. Клас запитів на сервер з параметрами та результатами

Також у проекті реалізовано кешування даних прийнятих з сервера, отже стрворено реляційну базу даних, яка зберігає відповіді, та одна з таблиць представлена на рис. 3.15:

```
@Entity(tableName = "albums")
@TypeConverters(
    AlbumImageEntityTypeConverter::class,
    AlbumTrackEntityTypeConverter::class,
    AlbumTagEntityTypeConverter::class
)
internal data class AlbumEntityModel(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val mbId: String,
    val name: String,
    val artist: String,
    val images: List<ImageEntityModel> = listOf(),
    val tracks: List<TrackEntityModel>?,
    val tags: List<TagEntityModel>?,
)
```

Рис. 3.15. Таблиця реляційної бази даних «Альбоми»

Для відображення з кешу даних, потрібно їх дістати з таблиць, саме запити до БД представлені на рисунку Рис. 3.16:

```
@Dao
internal interface AlbumDao {

    @Query("SELECT * FROM albums")
    suspend fun getAll(): List<AlbumEntityModel>

    @Query("SELECT * FROM albums where artist = :artistName and name = :albumName and mbId = :mbId")
    suspend fun getAlbum(artistName: String, albumName: String, mbId: String?): AlbumEntityModel

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertAlbums(albums: List<AlbumEntityModel>)
}
```

Рис. 3.16. Запити у баду даних

За допомогою цих запитів можна отримати альбоми, зберегти їх або отримати конкретний альбом, саме ці методи і потрібні для реалізованого нами функціоналу. Реалізація запиту на сервер, збереження результату у БД

```
override suspend fun searchAlbum(phrase: String): Result<List<Album>> =
    when (val apiResult = albumRetrofitService.searchAlbumAsync(phrase)) {
        is ApiResult.Success -> {
            val albums = apiResult
                .data SearchAlbumResponse
                .results SearchAlbumResultsApiModel
                .albumMatches AlbumListApiModel
                .album List<AlbumApiModel>
                .also { albumsApiModels ->
                    val albumsEntityModels = albumsApiModels.map { it.toEntityModel() }
                    albumDao.insertAlbums(albumsEntityModels)
                }
                .map { it.toDomainModel() }

            Result.Success(albums)
        }
        is ApiResult.Error -> {
            Result.Failure()
        }
        is ApiResult.Exception -> {
            Timber.e(apiResult.throwable)

            val albums = albumDao
                .getAll()
                .map { it.toDomainModel() }

            Result.Success(albums)
        }
```

Рис. 3.17. Реалізація запиту до конкретного альбому

При ситуації коли ми хочемо відкрити деталі якогось альбому, напишемо таку реалізацію, де у випадку відсутності альбому будемо запитувати його у сервера, а як альбом присутній – діставати його з БД. Реалізація зображена на рисунку рис. 3.18 :

```
override suspend fun getAlbumInfo(artistName: String, albumName: String, mbId: String?): Result<Alb
    when (val apiResult = albumRetrofitService.getAlbumInfoAsync(artistName, albumName, mbId)) {
        is ApiResult.Success -> {
            val album = apiResult
                .data GetAlbumInfoResponse
                .album AlbumApiModel
                .toDomainModel()

            Result.Success(album)
        }
        is ApiResult.Error -> {
            Result.Failure()
        }
        is ApiResult.Exception -> {
            Timber.e(apiResult.throwable)

            val album = albumDao
                .getAlbum(artistName, albumName, mbId)
                .toDomainModel()

            Result.Success(album)
        }
    }
}
```

Рис. 3.18. Відкриття деталей альбому

У результаті можна сформувати два use case для користувача, який буде користуватись цим функціоналом. Знаходячись на спільному екрні зі всіма альбомами він буде бачити перед собою сітку-список, де може листати її і обирати альбом. Обравши альбом і клацнувши на нього, повинен відкриватись новий екран та робитись запит у базу даних цього альбому. Можна узагальнити до двух *use case*. Перший – лістати список альбомів, другий – деталі обраного альбому. Реалізація діставання списку зображена на рис. 3.19:

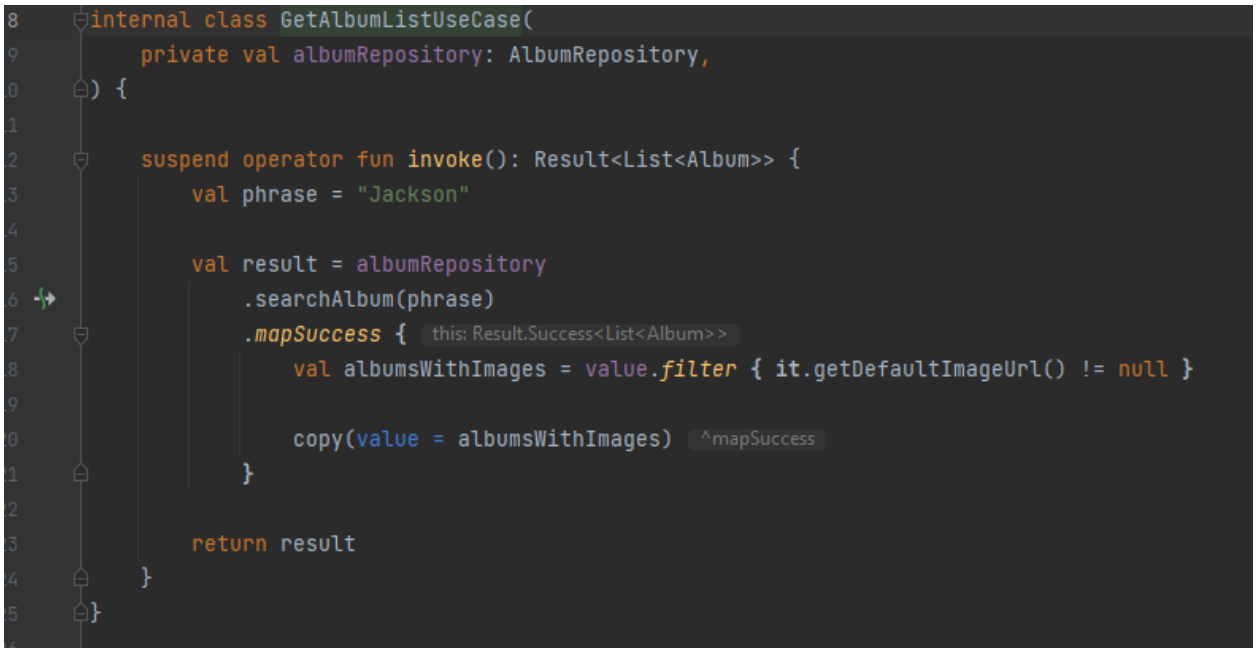


Рис. 3.19. Запит списку альбомів

Другий відповідно дістає деталі та зображений на рисунку на рис. 3.20:

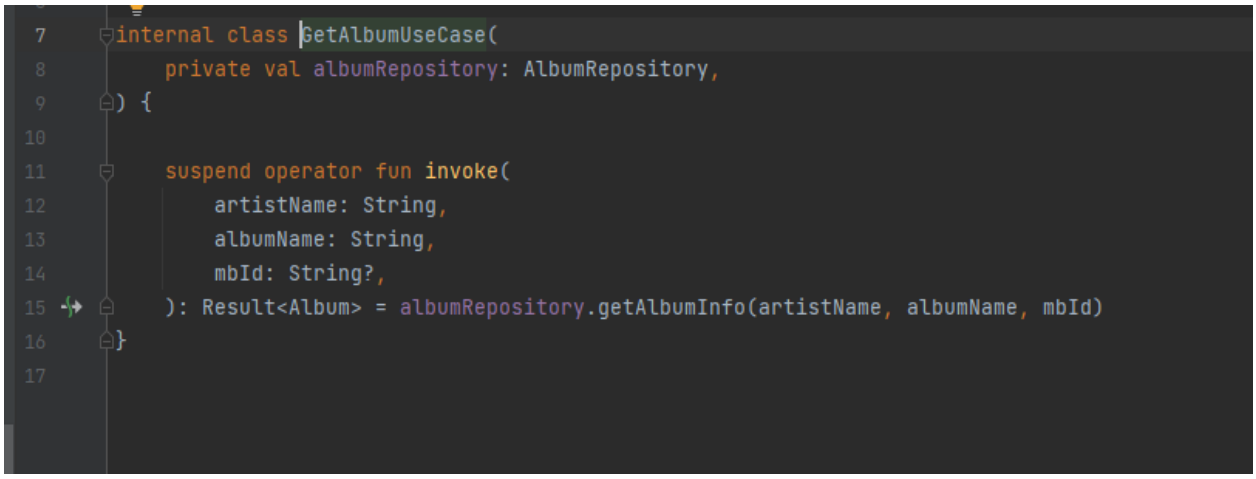
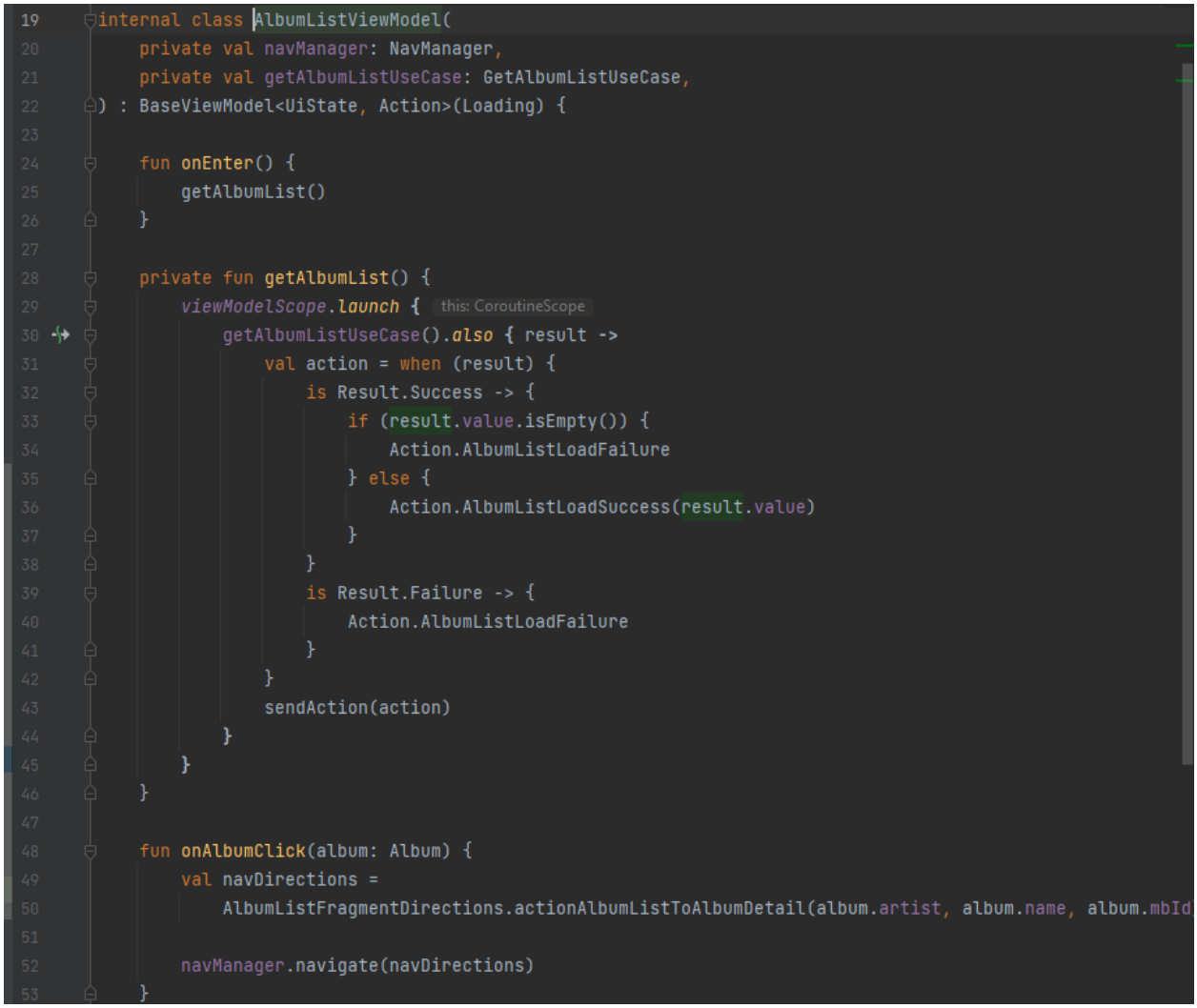


Рис. 3.20. Запит деталів альбому

Тепер для списку альбомів і деталей потрібно створити два екрана з його *viewmodel*, яка згідно нашої архітектури буде спілкуватись з *data* шаром архітектури. Створення viewmodel для списку наведено на рис. 3.21:



```
internal class AlbumListViewModel(
    private val navManager: NavManager,
    private val getAlbumListUseCase: GetAlbumListUseCase,
) : BaseViewModel<UiState, Action>(Loading) {

    fun onEnter() {
        getAlbumList()
    }

    private fun getAlbumList() {
        viewModelScope.launch { this: CoroutineScope
            getAlbumListUseCase().also { result ->
                val action = when (result) {
                    is Result.Success -> {
                        if (result.value.isEmpty()) {
                            Action.AlbumListLoadFailure
                        } else {
                            Action.AlbumListLoadSuccess(result.value)
                        }
                    }
                    is Result.Failure -> {
                        Action.AlbumListLoadFailure
                    }
                }
                sendAction(action)
            }
        }
    }

    fun onAlbumClick(album: Album) {
        val navDirections =
            AlbumListFragmentDirections.actionAlbumListToAlbumDetail(album.artist, album.name, album.mbId

        navManager.navigate(navDirections)
    }
```

Рис. 3.21. Viewmodel списку альбомів

Як видно з рис. 3.21, реалізований механізм діставання альбомів при відкритті екрану, сам метод який викликає наш use case, та звісно наша реалізація кліку користувача на альбом, що дозволяє нам відслідковувати альбом та навігувати його до потрібного за допомогою id.

Для створення UI використовуємо бібліотеку Compose, що дозволяє функціональ писати наш UI. Це допомагає швидко і зручно реалізувати список екранів лише за допомогою декількох функцій. Сама концепція такої побудови інтерфейсу користувача дуже вдало поєднується з використовуваною архітектурою. У такий спосіб потрібно будти дуже уважним при написанні екрану, так як бібліотека доволі нова і має багато застережень у використанні,

але вона легко впорається з таким простим функціоналом. Реалізація екрану списку альбомів наведена на рис. 3.22:

```
@Composable
private fun AlbumListScreen(viewModel: AlbumListViewModel) {
    val uiState: UiState by viewModel.uiStateFlow.collectAsStateWithLifecycle()

    uiState.let {  it: UiState
        when (it) {
            Error -> DataNotFoundAnim()
            Loading -> ProgressIndicator()
            is Content -> PhotoGrid(albums = it.albums, viewModel)
        }
    }
}

@OptIn(ExperimentalMaterial3Api::class)
@Composable
private fun PhotoGrid(albums: List<Album>, viewModel: AlbumListViewModel) {
    LazyVerticalGrid(
        columns = GridCells.Adaptive(Dimen.imageSize),
        contentPadding = PaddingValues(Dimen.screenContentPadding)
    ) {  this: LazyGridScope
        items(albums.size) { index ->
            val album = albums[index]

            ElevatedCard(
                modifier = Modifier
                    .padding(Dimen.spaceS)
                    .wrapContentSize(),
                onClick = { viewModel.onAlbumClick(album) }
            ) {  this: ColumnScope
                PlaceholderImage(
                    url = album.getDefaultImageUrl(),
                    contentDescription = stringResource(id = "Album Cover"),
                    modifier = Modifier.size(Dimen.imageSize)
                )
            }
        }
    }
}
```

Рис. 3.22. Екран списку альбомів

Так як основні компоненти уже створено, можна перевірити їх взаємодію та правильність написання коду, запустивши його і перевіривши чи все працює так як потрібно. Результат написання екрану списку альбомів представлений на рис. 3.23:

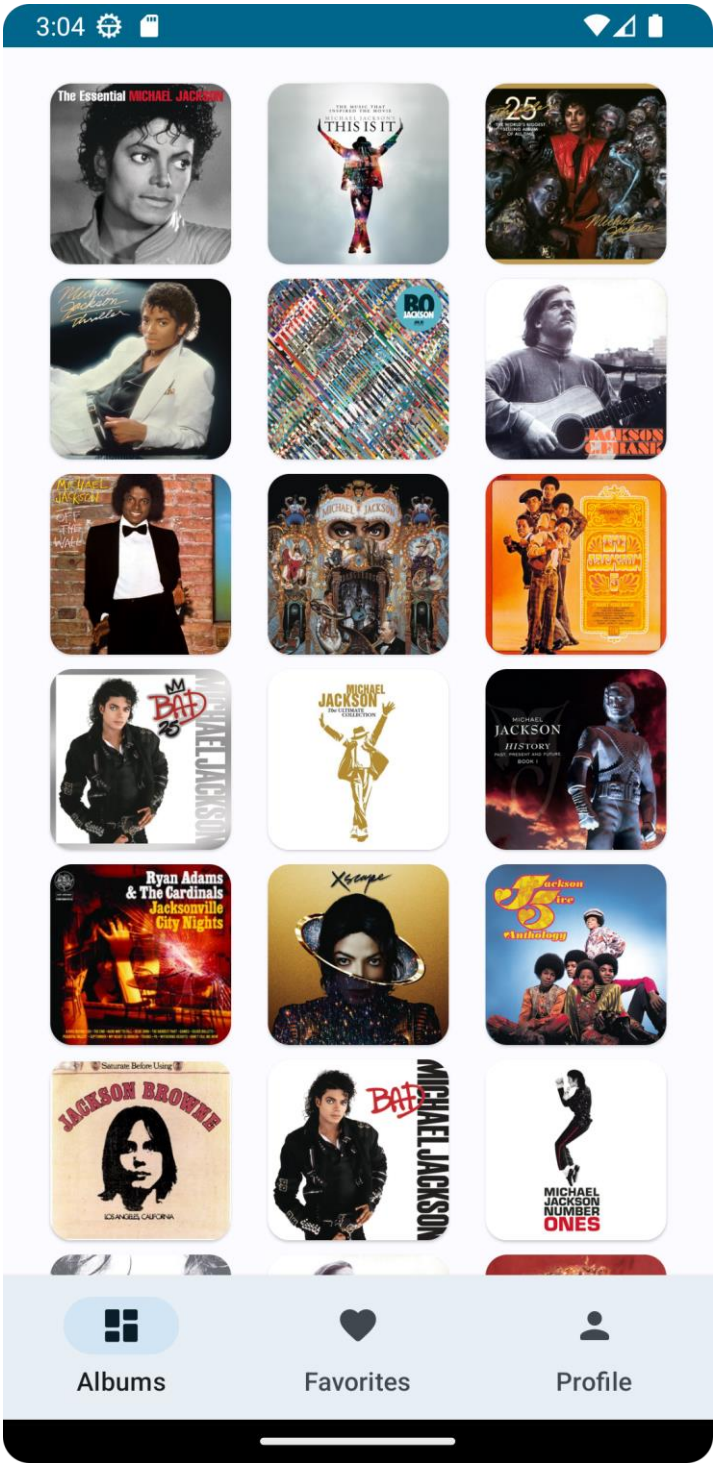


Рис. 3.23. Результат екрану списку альбомів

Результат вийшов чудовим, відступи між альбомами присутні та достатні для естетичного вигляду, при скроллі вниз підвантажуються нові альбоми, а старі знищуються, що не дає засмічувати оперативну пам’ять телефона та працює енергоефективно і повністю задовльняє наші вимоги.

Наступним кроком є по аналогії відтворити всі ці кроки для деталів альбому. Як можна зрозуміти звідси випливає і основна перевага такого підходу у розробці. Щоб створити новий екран з новим функціоналом нам потрбіно додати лише декілька файлів і не переживати за конкретну роботу кожного, адже базвий функціонал присутній у всіх. Приклад створення viewmodel на рисунку рис. 3.24:

```
internal class AlbumDetailViewModel(
    private val getAlbumUseCase: GetAlbumUseCase,
) : BaseViewModel<UiState, Action>(Loading) {

    fun onEnter(args: AlbumDetailFragmentArgs) {
        getAlbum(args)
    }

    private fun getAlbum(args: AlbumDetailFragmentArgs) {
        viewModelScope.launch { this: CoroutineScope
            getAlbumUseCase(args.artistName, args.albumName, args.mbId).also { it: Result<Album>
                when (it) {
                    is Result.Success -> {
                        sendAction(AlbumLoadSuccess(it.value))
                    }
                    is Result.Failure -> sendAction(AlbumLoadFailure)
                }
            }
        }
    }

    internal sealed interface Action : BaseAction<UiState> {
        class AlbumLoadSuccess(private val album: Album) : Action {
            override fun reduce(state: UiState) = Content(
                artistName = album.artist,
                albumName = album.name,
                coverImageUrl = album.getDefaultImageUrl() ?: "",
                tracks = album.tracks,
                tags = album.tags
            )
        }

        object AlbumLoadFailure : Action {
            override fun reduce(state: UiState) = UiState.Error
        }
    }
```

Рис. 3.24. Viewmodel деталів альбому

Viewmodel деталей пістить лише одну функцію отримання деталей альбому або з сервера, або з бази даних. Так як всі шари архітектруні відділені одни від одного, можна побачити, що клас не знає звідки отримає інформацію, а просто виконує методи та передає результат далі.

Use case для деталей виглядає так, як зображено на рисунку рис. 3.25:

```
internal class GetAlbumUseCase(
    private val albumRepository: AlbumRepository,
) {

    suspend operator fun invoke(
        artistName: String,
        albumName: String,
        mbId: String?,
    ): Result<Album> = albumRepository.getAlbumInfo(artistName, albumName, mbId)
}
```

Рис. 3.25. *Use case* деталів альбому

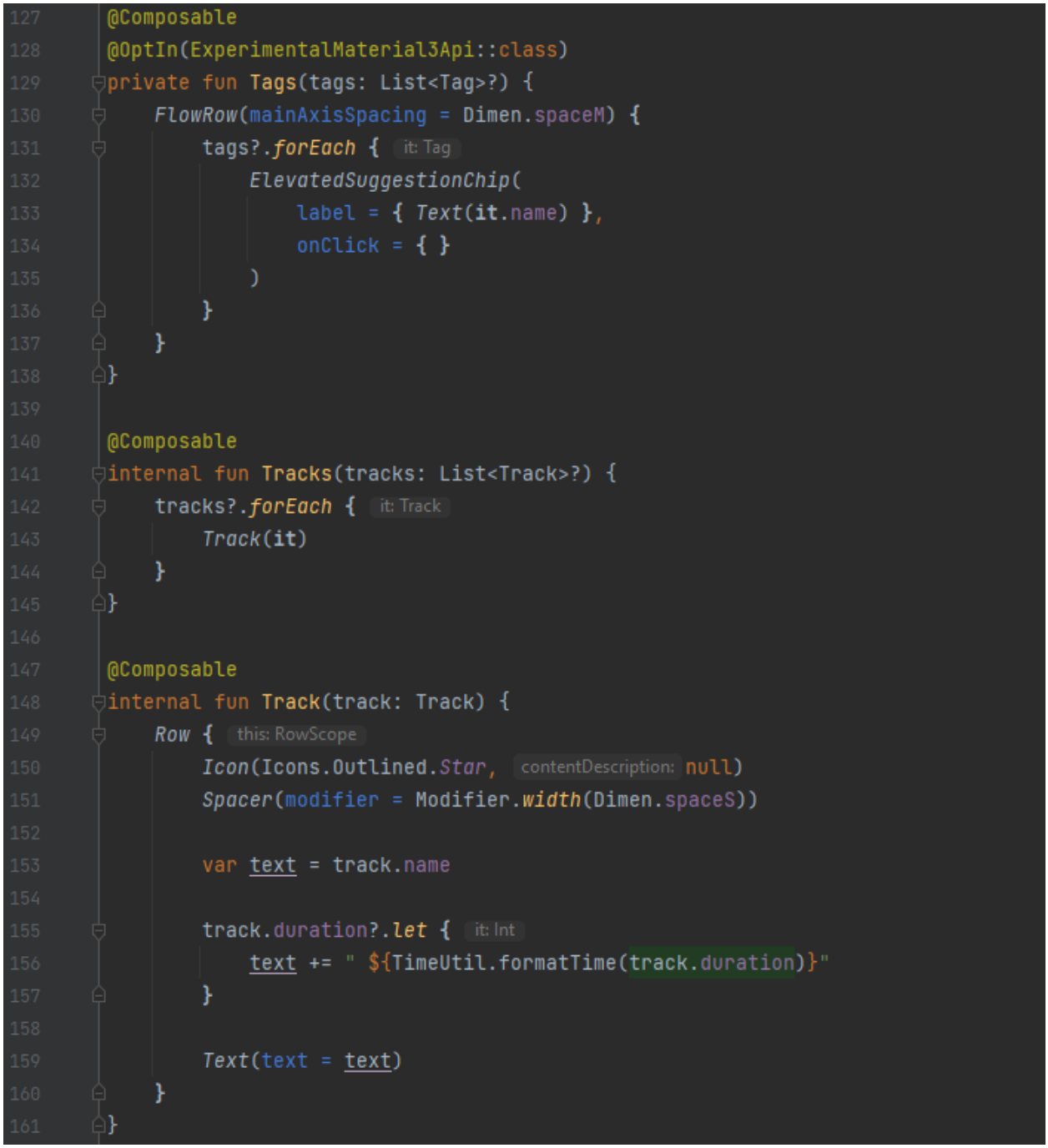
Створюємо екран по аналогії за своїм дизайном за допомогою бібліотеки *Compose*. Створення зображено на рис. 3.26, рис. 3.27 та рис. 3.28:

```
internal class AlbumDetailFragment : BaseFragment() {
    private val args: AlbumDetailFragmentArgs by navArgs()
    private val model: AlbumDetailViewModel by koinNavGraphViewModel(R.id.albumNavGraph)

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?,
    ): View {
        model.onEnter(args)

        return ComposeView(requireContext()).apply { this: ComposeView
            setContent {
                AlbumDetailScreen(uiStateFlow = model.uiStateFlow)
            }
        }
    }
}

@OptIn(ExperimentalLifecycleComposeApi::class)
@Composable
private fun AlbumDetailScreen(uiStateFlow: StateFlow<UiState>) {
    val uiState: UiState by uiStateFlow.collectAsStateWithLifecycle()

    uiState.let { it: UiState
        when (it) {
            Error -> DataNotFoundAnim()
            Loading -> ProgressIndicator()
            is Content -> PhotoDetails(it)
        }
    }
}
```

Рис. 3.26. Екран списку альбомів

```
88  private fun PhotoDetails(content: Content) {
89      Column(
90          modifier = Modifier
91              .padding(Dimen.screenContentPadding)
92              .verticalScroll(rememberScrollState())
93
94      ) { this: ColumnScope
95          ElevatedCard(
96              modifier = Modifier
97                  .padding(Dimen.spaceM)
98                  .wrapContentSize()
99                  .size(320.dp)
00                  .align(CenterHorizontally)
01          ) { this: ColumnScope
02              PlaceholderImage(
03                  url = content.coverImageUrl,
04                  contentDescription = stringResource(id = "Album Cover"),
05                  modifier = Modifier
06                      .fillMaxWidth()
07              )
08          }
09          Spacer(modifier = Modifier.height(Dimen.spaceL))
10          TextTitleLarge(text = content.albumName)
11          TextTitleMedium(text = content.artistName)
12          Spacer(modifier = Modifier.height(Dimen.spaceL))
13
14          if (content.tags?.isNotEmpty() == true) {
15              Tags(content.tags)
16              Spacer(modifier = Modifier.height(Dimen.spaceL))
17          }
18
19          if (content.tracks?.isNotEmpty() == true) {
20              TextTitleMedium(text = stringResource(id = "Tracks"))
21              Spacer(modifier = Modifier.height(Dimen.spaceS))
22              Tracks(content.tracks)
23          }
24      }
25  }
```

Рис. 3.27. Екран списку альбомів



```
@Composable
@OptIn(ExperimentalMaterial3Api::class)
private fun Tags(tags: List<Tag>?) {
    FlowRow(mainAxisSpacing = Dimen.spaceM) {
        tags?.forEach { it: Tag
            ElevatedSuggestionChip(
                label = { Text(it.name) },
                onClick = { }
            )
        }
    }
}

@Composable
internal fun Tracks(tracks: List<Track>?) {
    tracks?.forEach { it: Track
        Track(it)
    }
}

@Composable
internal fun Track(track: Track) {
    Row { this: RowScope
        Icon(Icons.Outlined.Star, contentDescription: null)
        Spacer(modifier = Modifier.width(Dimen.spaceS))

        var text = track.name

        track.duration?.let { it: Int
            text += " ${TimeUtil.formatTime(track.duration)}"
        }

        Text(text = text)
    }
}
```

Рис. 3.28. Екран списку альбомів

Створивши всі необхідні файли можна подивитись на вигляд деталей і чи працюють вони, правильно дістаючи дані по кліку на альбом. Результат роботи зображено на рис. 3.29:

Рис. 3.29. Екран деталей альбому

ВИСНОВКИ ДО РОЗДІЛУ 3

У даному розділі було описане створення Android додатку на основі проаналізованого матеріалу, використовуючи найкращі із кожних існуючих підходів. Описане поетапне створення застосунка, а саме: створення базових на функціональних модулів, створення маніфесту файлу, створення файлу залежностей бібліотек, повний цикл розробки функціонального модулю.

Наведені вдалі приклади ефективності такого підходу особливо для роботи в команді, так як у проекті є змога паралельно працювати над різними функціональними модулями не конфліктуючи з кодом іншого розробника. Продемонстровано можливості сучасних бібліотек у найпополярнішому для створення функціоналу на ОС Android. Закладено масштабовану архітектуру, яку легко розширювати та змінювати незалежні її модулі. Створено проект додатку, у якому закладені на майбутнє ще два не реалізовані модулі «у розробці». Усі вимоги описані у другому розділі були дотримані, додаток писався чітко по описаній інструкції, дотримуючись якої можна створити будь-який застосунок швидко та якомога правильніше.

Виявлені всі плюси та мінуси багатомодульної архітектури, що дозволяє ефективніше займатись менеджментом розробки.

У результаті був продемонстрований готовий працюючий додаток, з реалізованим функціональним модулем, закладеною чистою архітектурою що відмінно працював, як показало тестування.

ВИСНОВКИ

У теперішніх реаліях важливою частиною є написати програму для Android, яку легко підтримувати, тестувати та змінювати. Це також повинно бути легко зрозуміти — якщо хтось новий приходить до вашої роботи, у нього не повинно виникнути проблем із розумінням потоку даних або структури.

Якщо вони знають, що архітектура чиста, вони можуть бути впевнені, що зміни в інтерфейсі користувача не вплинуть ні на що в моделі, а додавання нової функції не займе більше, ніж передбачено. Навіть якщо є добре структурований додаток, його дуже легко зламати безладними змінами коду «на мить, просто для роботи». Кожен код, який порушує правила, може зберігатися в базі кодів і може стати джерелом майбутніх більших порушень.

Якісна архітектура коду є проблемою не лише на початку проекту — це виклик для будь-якої частини життя програми Android. Безлад у коді програми помститься нам, коли ми найменше цього очікуємо. Основними перевагами використання чистих архітектур для Android проектів є:

1. Вирішує «Поділ проблем» який є передумовою кожного проекту.
2. Тестувальникам додатків Android легше працювати з «Чистою архітектурою». Помітно збільшується охоплення тестами код.
3. Код легше підтримувати.
4. Акуратно розділивши рівень презентації, рівень бізнес-логіки та сутності, команді проекту легше рухатися вперед у своїй роботі.
5. Заняття, в тому числі і тестові, є цілеспрямованими.
6. Зміни в інтерфейсі користувача можуть продовжуватися відповідно до вимог бізнесу. Ці зміни та їх тестування можуть продовжуватися в стабільному середовищі.

Наведені архітектури є шаблонами до який дійшли розробники з досвідом. Повторюючи одні і ті ж вирішення проблем, їх винесли в окремі шаблони, що допомагають структурувати код, та дають безліч переваг для кожного розробника реального проекту.

Такий спосіб дає неймовірний приріст по швидкості збірки проекту, особливу перевагу можна побачити з наявними бібліотеками, які мають багато кодогенерації, отже не будуть її виконувати, якщо файли у модулі не змінювались.

Функціонал спроектований та декомпозований на частини, отже над проектом може легко може працювати декілька людей паралельно, розробляючи програмний продукт та маючи свої зони відповідальності, які не залежні одна від одної, що робить командну роботу максимально ефективною.

Мета цього підходу полягала в тому, щоб знайти правильний баланс між надмірною модуляцією відносно невеликої програми та використанням цієї можливості, щоб продемонструвати шаблон модуляції, який підходить для набагато більшої кодової бази, ближче до реальних програм у виробничих середовищах.

Наведені вдалі приклади ефективності підходу особливо для роботи в команді, так як у проекті є змога паралельно працювати над різними функціональними модулями не конфліктуючи з кодом іншого розробника. Продемонстровано можливості сучасних бібліотек у найпополярнішому для створення функціоналу на ОС Android. Закладено масштабовану архітектуру, яку легко розширювати та змінювати незалежні її модулі. Усі описані вимоги до мети кваліфікаційної роботи виконані, додаток писався чітко по описаній інструкції, дотримуючись якої можна створити будь-який застосунок швидко та якомога правильніше.

Виявлені всі плюси та мінуси багатомодульної архітектури, що дозволяє ефективніше займатись керуванням розробки. На основі проаналізованих підходів інженерів, що писали великі популярні додатки було виявлено та викорінено всі можливі недоліки архітектурних підходів та створено і описано найоптимізованіший.

У результаті було продемонстровано готовий додаток, з реалізованим функціональним модулем, закладеною чистою архітектурою, що безпомилково працював, як показало тестування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Гленфорд Майерс. Мистецтво тестування програм / Дж. Гл. Майерс ; Видавництво «Львів-прес» - Львів, 2016 рік -272 с. : іл. Бібліогр.: с 102 – 108.

2. Джефф Кароло. Як тестують у Google / Дж. Каролло ; Видавництво «Україна» - Дніпро, 2018 рік -315 с. : іл. Бібліогр.: с 92 – 102.

3. Рон Паттон. Мистецтво тестування / Р. Б. Паттон ; Видавництво «U-Books» - Київ, 2013 рік - 511 с. : іл. Бібліогр.: с 178 – 185.

4. Чиста архітектура. Роберт Мартин. Видавництво «Фабула» - Львів, 2019 рік -332 с. : іл. Бібліогр.: с 23 – 159.

5. Принципи забезпечення безпеки архітектури інформаційної системи на базі клієнтських додатків для ОС Android [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://www.csecurity.kubg.edu.ua/index.php/journal/article/view/156/150

6. What is Clean Architecture in Android? [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://www.geeksforgeeks.org/what-is-clean-architecture-in-android/

7. Detailed Guide on Android Clean Architecture. [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://medium.com/android-dev-hacks/detailed-guide-on-android-clean-architecture-9eab262a9011

8. The SOLID Principles of Object-Oriented Programming Explained in Plain English. [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/

9. Guide to app architecture. [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://developer.android.com/topic/architecture

10. A Guide to Google’s Recommended Android Apps Architecture. [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://www.scalablepath.com/android/android-apps-architecture

11. Android Architecture Patterns. [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://www.geeksforgeeks.org/android-architecture-patterns/

12. Modern Android Architecture MVI design pattern. [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://amsterdamstandard.com/en/post/modern-android-architecture-with-mvi-design-pattern

13. Архітектура та проектування програмного забезпечення. [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: http://dspace.wunu.edu.ua/jspui/bitstream/316497/24194/

14. KISS, DRY, S.O.L.I.D, YAGNI — навіщо дотримуватись принципів програмування? [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://senior.ua/articles/kiss-dry-solid-yagni--navscho-dotrimuvatis-principv-programuvannya

15. Everything you need to build on Android. [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://developer.android.com/studio

16. Android Development | Best Practices.. [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://proandroiddev.com/android-development-best-practices-7278e9cdbbe9

17. Найкращі практики розробки програм для Android. [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://www.netguru.com/blog/best-practices-in-android-app-development

18. Clean Architecture Guide (with tested examples). [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://proandroiddev.com/clean-architecture-data-flow-dependency-rule-615ffdd79e29

19. The new guide to Android app modularization. [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://android-developers.googleblog.com/2022/09/announcing-new-guide-to-android-app-modularization.html.

20. Last.fm Music Discovery API. [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://www.last.fm/