

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО»**

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці

ДО ЗАХИСТУ ДОПУЩЕНО

В.о. завідувача кафедри

Олександр КОВАЛЬ

«__» _____ 2026 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інженерія програмного забезпечення

інтелектуальних кібер-фізичних систем в енергетиці»

спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Навчально-методичний комплекс з дисципліни «Розробка

програмного забезпечення мобільних пристроїв» на Kotlin.»

Виконав:

студент IV курсу, групи ТВ-23

Черниш Михайло Андрійович

(прізвище, ім'я, по батькові)

_____ (підпис)

Керівник: професор кафедри ІПЗЕ, доктор
технічних наук, доцент Недашківський О. Л.

(посада, науковий ступінь, вчене звання, прізвище та ініціали)

_____ (підпис)

Рецензент:

(посада, науковий ступінь, вчене звання, прізвище та ініціали)

_____ (підпис)

Засвідчую, що у цій дипломній роботі немає
запозичень із праць інших авторів без
відповідних посилань.

Студент _____
(підпис)

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра інженерії програмного забезпечення в енергетиці

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці»

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри

Олександр КОВАЛЬ

«___» _____ 2026 р.

ЗАВДАННЯ

на дипломну роботу студенту

Чернишу Михайлу Андрійовичу

(прізвище, ім'я, по батькові)

1. Тема роботи: «Навчально-методичний комплекс з дисципліни «Розробка програмного забезпечення мобільних пристроїв» на Kotlin.», керівник роботи Недашківський Олексій Леонідович, затверджені наказом по університету від «___» _____ 2026 р. № _____.
2. Строк подання студентом роботи «___» червня 2026 р.
3. Вихідні дані до роботи: мова програмування Kotlin; декларативний UI-фреймворк Jetpack Compose; середовище розробки Android Studio; система збирання Gradle; бібліотеки Room, Hilt, DataStore Preferences та Kotlin Coroutines; формат збереження контенту JSON.
4. Зміст пояснювальної записки (перелік завдань, які потрібно розробити): техніко-економічне обґрунтування вибору архітектури системи; аналіз предметної області та програмних аналогів; методичне проєктування та структурування навчального курсу; програмно-архітектурна реалізація мобільного застосунку; визначення системних вимог та сценаріїв взаємодії користувача; тестування та валідація програмного забезпечення.
5. Перелік ілюстративного матеріалу: архітектурна схема застосунку, UML-діаграми варіантів використання, лістинги програмного коду, скріншоти інтерфейсу застосунку.
6. Дата видачі завдання «31» жовтня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Строк виконання етапів	Примітка
1	Отримання завдання	31.10.2025	Виконано
2	Аналіз предметної області та аналогів	01.11.2025 – 30.11.2025	Виконано
3	Техніко-економічне обґрунтування рішення	01.12.2025 – 31.12.2025	Виконано
4	Методичне проєктування курсу	01.01.2026 – 31.01.2026	Виконано
5	Програмна реалізація застосунку	01.02.2026 – 03.05.2026	Виконано
6	Тестування та валідація	04.05.2026 – 10.05.2026	Виконано
7	Захист програмного продукту на кафедрі	11.05.2026 – 15.05.2026	Виконано
8	Оформлення дипломної роботи	18.05.2026 – 31.05.2026	Виконано
9	Передзахист	01.06.2026 – 05.06.2026	
10	Захист	15.06.2026 – 19.06.2026	

Студент

(підпис)

Михайло ЧЕРНИШ

(ім'я, ПРІЗВИЩЕ)

Керівник роботи

(підпис)

Олексій НЕДАШКІВСЬКИЙ

(ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Структура та обсяг дипломної роботи. Робота містить 76 сторінок, 33 рисунки, 9 таблиць, 3 додатки та 22 джерела за переліком посилань.

Актуальність теми зумовлена стрімким оновленням промислового технологічного стеку Android-розробки та потребою у сучасних навчально-методичних матеріалах, адаптованих під навчальний план кафедри ІІЗЕ, а також специфічною для України вимогою автономності навчального засобу в умовах нестабільного енергопостачання та мережевого зв'язку.

Метою роботи є удосконалення та розширення навчально-методичного комплексу з дисципліни «Розробка програмного забезпечення мобільних пристроїв» на Kotlin шляхом проєктування методичних матеріалів для двадцяти семи практичних занять та реалізації автономного мобільного застосунку, що надає доступ до цих матеріалів.

Об'єктом дослідження є процес методичного забезпечення дисципліни мобільної розробки на базі сучасних технологій платформи Android.

Предметом дослідження є структура, зміст і форми подання навчально-методичних матеріалів та програмно-архітектурні рішення автономного мобільного застосунку-НМК.

Наукова та практична новизна полягає у проєктуванні автономного двомовного мобільного НМК з кастомною підсистемою динамічної локалізації без перезапуску активності, а також у формалізації навчального контенту моделями даних, що уможливорює його програмну обробку та реактивне подання.

Практичне значення одержаних результатів полягає у готовому до впровадження НМК, що розповсюджується через систему «Кафедра» у вигляді APK-файла та функціонує повністю автономно без мережевого з'єднання. Застосунок може використовуватися при викладанні відповідного освітнього компонента у КПІ ім. Ігоря Сікорського.

Ключові слова: навчально-методичний комплекс, kotlin, jetpack compose, mvvm, room, hilt, datastore, автономний застосунок, локалізація, мобільна розробка, інженерія програмного забезпечення

ABSTRACT

Structure and scope of the thesis. The work contains 76 pages, 33 figures, 9 tables, 3 appendices, and 22 references.

The relevance of the topic is driven by the rapid evolution of the industrial Android technology stack and the need for modern teaching materials adapted to the curriculum of the Software Engineering for Energy department, as well as the Ukraine-specific requirement of full offline autonomy under conditions of unstable power supply and network connectivity.

The aim of the work is the enhancement and extension of the educational and methodological complex for the discipline “Mobile Application Software Development” in Kotlin through the design of teaching materials for twenty-seven practical sessions and the implementation of an autonomous mobile application providing access to these materials.

The object of research is the process of methodological support for a mobile development discipline based on modern Android platform technologies.

The subject of research is the structure, content, and presentation forms of teaching materials, as well as the software-architectural solutions of the autonomous mobile educational application.

The practical significance of the results lies in a deployment-ready educational complex distributed through the “eKafedra” system as an APK file that operates fully offline without any network connection, suitable for teaching the corresponding course at Igor Sikorsky Kyiv Polytechnic Institute.

Keywords: educational complex, kotlin, jetpack compose, mvvm, room, hilt, datastore, offline application, localization, mobile development, software engineering

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ	8
ВСТУП.....	9
1 ТЕХНІКО-ЕКОНОМІЧНЕ ОБҐРУНТУВАННЯ.....	12
1.1 Постановка задачі розробки навчально-методичного комплексу	12
1.2 Аналіз альтернативних варіантів реалізації системи	14
1.3 Критерії порівняльного аналізу та їх вагові коефіцієнти	16
1.4 Обґрунтування вибору фінального архітектурного рішення.....	17
Висновки до розділу 1.....	20
2 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА АНАЛОГІВ	22
2.1 Детальний аналіз існуючих освітніх платформ	22
2.2 Дослідження парадигм побудови інтерфейсу: декларативна та імперативна	24
2.3 Аналіз технологій локального збереження даних на пристрої	26
Висновки до розділу 2.....	29
3 МЕТОДИЧНЕ ПРОЄКТУВАННЯ ТА СТРУКТУРУВАННЯ КУРСУ	30
3.1 Архітектура навчального контенту	30
3.2 Таксономія Блума та концепція конструктивного вирівнювання	33
3.3 Інженерно-енергетична специфіка практичних завдань	35
Висновки до розділу 3.....	38
4 ПРОГРАМНО-АРХІТЕКТУРНА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ.....	39
4.1 Опис глобальної архітектурної схеми застосунку.....	39
4.2 Реалізація шару даних (Data Layer)	41
4.3 Реалізація презентаційного шару (Presentation Layer).....	48

4.4 Механізм динамічної безсерверної локалізації.....	55
4.5 Впровадження залежностей (Dependency Injection)	57
Висновки до розділу 4.....	58
5 ВЗАЄМОДІЯ КОРИСТУВАЧА ІЗ СИСТЕМОЮ ТА СЦЕНАРІЇ ВИКОРИСТАННЯ	59
5.1 Системні вимоги програмного продукту	59
5.2 Інструкція з розгортання та встановлення	60
5.3 Проектування людино-машинного інтерфейсу та сценарії взаємодії	61
Висновки до розділу 5.....	64
6 ТЕСТУВАННЯ ТА ВАЛІДАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....	65
6.1 Стратегія тестування застосунку	65
6.2 Модульне тестування бізнес-логіки	66
6.3 Інструментальне тестування інтерфейсу.....	68
Висновки до розділу 6.....	69
ЗАГАЛЬНІ ВИСНОВКИ.....	70
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	72
ДОДАТОК А	75
ДОДАТОК Б.....	95
ДОДАТОК В.....	108

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Android	мобільна операційна система компанії Google
API	Application Programming Interface — інтерфейс програмування застосунків
APK	Android Package Kit — інсталяційний пакет Android-застосунку
DAO	Data Access Object — об'єкт доступу до даних
DI	Dependency Injection — впровадження залежностей
DataStore	реактивне сховище налаштувань платформи Android Jetpack
Gson	бібліотека серіалізації та десеріалізації JSON від Google
Hilt	бібліотека впровадження залежностей на основі Dagger 2
HMI	Human-Machine Interface — людино-машинний інтерфейс
IDE	Integrated Development Environment — інтегроване середовище розробки
JSON	JavaScript Object Notation — формат обміну даними
MVVM	Model-View-ViewModel — архітектурний патерн
ORM	Object-Relational Mapping — об'єктно-реляційне відображення
R8	інструмент скорочення та обфускації коду Android
Room	бібліотека ORM-абстракції над SQLite від Google
SDK	Software Development Kit — набір інструментів розробника
UDF	Unidirectional Data Flow — однонаправлений потік даних
UI	User Interface — інтерфейс користувача
UML	Unified Modeling Language — уніфікована мова моделювання
ViewModel	компонент архітектури Android Jetpack для зберігання стану
ЗВО	заклад вищої освіти
КПІ	Київський політехнічний інститут імені Ігоря Сікорського
НМК	навчально-методичний комплекс
ОК	освітній компонент
ОС	операційна система

ВСТУП

Стрімкий розвиток мобільних технологій та цифровізація суспільства формують нові вимоги до підготовки фахівців у галузі програмної інженерії. Станом на 2025 рік переважна більшість нових застосунків для платформи Android розробляється мовою Kotlin — сучасною, лаконічною та безпечною альтернативою Java, офіційно підтриманою компанією Google. Поряд із Kotlin фреймворк Jetpack Compose кардинально змінив підхід до побудови інтерфейсів, запровадивши декларативну парадигму програмування, яка сьогодні є галузевим стандартом [11, 17].

Підготовка студентів спеціальності 121 «Інженерія програмного забезпечення» вимагає постійного оновлення навчально-методичних матеріалів відповідно до актуального стану індустрії. Наявні навчальні курси нерідко залишаються зосередженими на застарілих підходах або не охоплюють повний цикл розробки — від проєктування архітектури до тестування та публікації застосунку [22].

Актуальність теми полягає в потребі розроблення комплексного, сучасного та практично орієнтованого навчально-методичного комплексу (НМК), що охоплює ключові технології Android-розробки: Kotlin, Jetpack Compose, архітектуру MVVM, Room Database, Kotlin Coroutines та Hilt. Відповідно до курсу кафедри на діджиталізацію, матеріали представлено у вигляді мобільного застосунку, який здобувачі завантажують через систему «ЄКафедра» та використовують як автономний цифровий довідник. Специфічною для України вимогою є повна автономність такого засобу в умовах нестабільного енергопостачання та мережевого зв'язку [22].

Об'єктом дослідження є процес методичного забезпечення дисципліни «Розробка програмного забезпечення мобільних пристроїв» на базі сучасних технологій Android-розробки.

Предметом дослідження є структура, зміст та форми подання навчально-методичних матеріалів, а також програмно-архітектурні рішення автономного

мобільного застосунку для ефективного опанування здобувачами практичних навичок розробки мобільних застосунків на Kotlin.

Метою роботи є удосконалення та розширення функціоналу навчально-методичного комплексу з дисципліни «Розробка програмного забезпечення мобільних пристроїв» на Kotlin шляхом розроблення структурованих методичних матеріалів для двадцяти семи практичних занять та реалізації мобільного застосунку, що надає доступ до цих матеріалів у зручному форматі.

Для досягнення мети поставлено такі завдання:

- проаналізувати сучасні тенденції розробки мобільних застосунків на Kotlin та дослідити існуючі онлайн-платформи навчання;
- обґрунтувати вибір архітектури системи серед альтернативних варіантів за визначеними критеріями;
- розробити структуру та зміст двадцяти семи практичних занять із детальними методичними матеріалами;
- реалізувати автономний мобільний застосунок-НМК на Kotlin та Jetpack Compose зі вбудованими навчальними матеріалами;
- провести тестування та валідацію розробленого програмного забезпечення.

Практичне значення одержаних результатів полягає у впровадженні сучасного НМК, що використовуватиметься при викладанні освітнього компонента «Розробка програмного забезпечення мобільних пристроїв». Застосунок розповсюджується у вигляді APK-файла через систему «єКафедра» та функціонує повністю автономно — без потреби в підключенні до Інтернету.

Під час підготовки тексту дипломної роботи для редагування, перевірки граматики та стилістичного покращення викладу використано допоміжні інструменти на основі штучного інтелекту. Усі змістовні рішення, технічну реалізацію застосунку, проведення тестування та аналіз отриманих результатів виконано автором особисто; згадані інструменти застосовувалися виключно для роботи з текстовою частиною без впливу на її інженерний зміст.

Структура роботи. У першому розділі наведено техніко-економічне обґрунтування вибору архітектури системи. У другому розділі проаналізовано предметну область та програмні аналоги, обґрунтовано ключові технічні рішення. Третій розділ присвячено методичному проєктуванню курсу. У четвертому розділі детально розкрито програмно-архітектурну реалізацію застосунку. П'ятий розділ описує системні вимоги та сценарії взаємодії користувача. У шостому розділі викладено стратегію та результати тестування. У висновках узагальнено отримані результати та окреслено перспективи подальшого розвитку.

1 ТЕХНІКО-ЕКОНОМІЧНЕ ОБҐРУНТУВАННЯ

1.1 Постановка задачі розробки навчально-методичного комплексу

Сучасний стан викладання дисциплін мобільної розробки на кафедрі інженерії програмного забезпечення в енергетиці (ІПЗЕ) формує суперечність між темпами оновлення промислового технологічного стеку та інерційністю традиційних паперових і веб-орієнтованих методичних матеріалів. Платформа Android щороку зазнає істотних змін: декларативний фреймворк Jetpack Compose витіснив імперативну систему View/XML, корутини Kotlin замінили колбеки та RxJava, а реактивне сховище DataStore прийшло на зміну синхронному SharedPreferences. Методичне забезпечення, прив'язане до статичних документів, об'єктивно не встигає за цією динамікою, унаслідок чого здобувач опановує застарілі парадигми проєктування [22].

Додатковим обмежувальним чинником, специфічним для українського контексту, є нестабільність енергопостачання та мережевої інфраструктури. Тривалі планові та аварійні відключення електроенергії, перебої мобільного зв'язку та відсутність стабільного Wi-Fi у навчальних аудиторіях роблять будь-яке хмарне або клієнт-серверне рішення вразливим: у момент відключення живлення серверної частини або мережевого каналу здобувач повністю втрачає доступ до навчальних матеріалів. Для дисципліни, орієнтованої на галузь енергетики, така залежність є методологічно неприйнятною.

З огляду на зазначене, сформульовано інженерну задачу: спроєктувати та реалізувати автономний мобільний застосунок-НМК, який інкапсулює повний обсяг навчального контенту дисципліни «Розробка програмного забезпечення мобільних пристроїв» (дев'ять тематичних модулів, двадцять сім практичних занять, по дванадцять обов'язкових інформаційних блоків у кожному) безпосередньо у складі інсталяційного пакета APK та забезпечує до нього доступ без жодних мережових запитів. Розв'язання задачі передбачає одночасне

дотримання сукупності вимог, які доцільно розмежувати на функційні та нефункційні.

До функційних вимог віднесено:

- відображення ієрархічної структури курсу (модуль → заняття → інформаційний блок) із підтримкою згортання та розгортання вузлів;
- повне подання змісту кожного практичного заняття: огляд, теоретичні відомості, варіанти індивідуальних завдань, критерії оцінювання, контрольні питання, посилання на зразкові рішення;
- збереження обраних занять для швидкого доступу із застосуванням локальної реляційної бази даних;
- двомовність інтерфейсу (українська та англійська) з перемиканням мови у режимі реального часу без перезавантаження застосунку.

До нефункційних (системних) вимог віднесено:

- повну автономність — працездатність застосунку за відсутності мережевого з'єднання та електроживлення зовнішньої інфраструктури;
- мінімальну вартість володіння системою (Total Cost of Ownership) — відсутність витрат на оренду серверів та хмарних сервісів;
- високу швидкодію перекомпонування складних декларативних інтерфейсів навіть на бюджетних пристроях;
- захищеність і приватність даних здобувача — відсутність передавання будь-якої інформації за межі пристрою;
- реалістичну трудомісткість реалізації однією особою у межах строків переддипломної практики та дипломного проєктування.

Оскільки наведені вимоги частково конкурують між собою (зокрема, динамічність оновлення контенту суперечить вимозі автономності), вибір архітектурного рішення не може бути зроблено інтуїтивно. Відповідно до нормативних вимог кафедри ІІЗЕ та критеріїв рейтингової системи оцінювання якості дипломних робіт, обґрунтування основних технічних рішень виконується шляхом розгляду не менш ніж трьох альтернативних варіантів із подальшим

вибором оптимального за обґрунтовано обраним критерієм. Цей аналіз наведено у підрозділах 1.2–1.4.

1.2 Аналіз альтернативних варіантів реалізації системи

Для розв’язання поставленої задачі розглянуто три принципово відмінні архітектурні стратегії, що репрезентують увесь спектр сучасних підходів до дистрибуції освітнього мобільного контенту: повністю автономну нативну, клієнт-серверну та кросплатформну.

Варіант А — нативний автономний Android-застосунок

Стратегія передбачає реалізацію застосунку засобами мови Kotlin та фреймворку Jetpack Compose, у якому весь навчальний контент зберігається статично у вигляді ресурсних JSON-файлів (`modules_uk.json`, `modules_en.json`, `practice_N_uk.json`, `practice_N_en.json`), вбудованих у пакет APK на етапі збирання. Десеріалізація ієрархічних структур виконується локально утилітами `ModuleUtil` та `PracticeUtil`, реляційне сховище `Room` застосовується виключно для збереження обраних занять, а налаштування мови інтерфейсу зберігаються у `DataStore Preferences`. Доступ до даних інкапсульовано єдиним репозиторієм `PracticalAssignmentRepository`, а граф залежностей формується контейнером `Hilt`. Перевагою варіанта є абсолютна незалежність від зовнішньої інфраструктури та нативна швидкодія. Обмеженням — статичність контенту: оновлення матеріалів вимагає перезбирання та повторної публікації APK.

Варіант Б — клієнт-серверна архітектура з централізованим сховищем

Стратегія передбачає винесення навчального контенту на бекенд (наприклад, серверний застосунок на фреймворку `Ktor`) або у хмарну платформу (`Firebase Firestore`, `AWS`), звідки мобільний клієнт динамічно підвантажує матеріали через `REST API`. Беззаперечною перевагою є миттєвість оновлення контенту: викладач

змінює дані на сервері, і всі здобувачі негайно отримують актуальну версію без перевстановлення застосунку. Натомість варіант породжує низку критичних недоліків: повну непрацездатність під час відключень електроенергії та відсутності мережі, постійні експлуатаційні витрати на оренду серверів і трафік, розширення поверхні атаки (передавання даних, автентифікація, керування токенами) та суттєве зростання трудомісткості — потребу проєктувати, реалізовувати й супроводжувати ще й серверний компонент та схему API.

Варіант В — кросплатформне рішення (Kotlin Multiplatform або Flutter)

Стратегія орієнтована на охоплення водночас платформ Android та iOS із єдиної кодової бази засобами Kotlin Multiplatform (KMP) із Compose Multiplatform або фреймворку Flutter. Потенційною перевагою є розширення аудиторії на власників пристроїв Apple. Проте у конкретному контексті кафедри ця перевага значною мірою нівелюється: основний канал дистрибуції — система «єКафедра» — передбачає поширення інсталяційного пакета прямим завантаженням (sideloading), що технічно можливе на Android, але заборонене політикою iOS, де встановлення поза App Store практично недоступне. Отже, для покриття iOS однаково знадобилися б публікація в App Store, обліковий запис Apple Developer (\$99 на рік) та проходження модерації, що суперечить моделі автономної безсерверної дистрибуції. Додатково KMP-стек із Compose Multiplatform на iOS на момент розробки перебуває у стадії дозрівання, а Flutter використовує власний рушій рендерингу замість нативних компонентів Material You, що ускладнює відтворення цільового дизайну [17].

1.3 Критерії порівняльного аналізу та їх вагові коефіцієнти

Для об'єктивного зіставлення варіантів застосовано метод зваженого оцінювання (weighted scoring model). Сформовано чотири критерії, що безпосередньо впливають із нефункційних вимог підрозділу 1.1. Кожному критерію присвоєно ваговий коефіцієнт, який відображає його відносну значущість для умов експлуатації НМК у закладі вищої освіти України; сума вагових коефіцієнтів дорівнює одиниці.

- **Автономність та офлайн-стійкість** (вага 0,35) — здатність системи функціонувати без мережевого з'єднання та зовнішнього електроживлення. Критерію присвоєно найбільшу вагу, оскільки безперебійність доступу до матеріалів під час блекаутів є визначальною характеристикою НМК для енергетичної галузі.

- **Вартість розробки та володіння** (вага 0,25) — сукупні витрати, що охоплюють трудомісткість розробки в людино-годинах та експлуатаційні витрати на інфраструктуру (оренда серверів, облікові записи, трафік) протягом життєвого циклу системи.

- **Безпека та приватність локальних даних** (вага 0,25) — ступінь захищеності даних здобувача, що визначається обсягом поверхні атаки та наявністю каналів передавання інформації за межі пристрою.

- **Швидкодія рендерингу інтерфейсу** (вага 0,15) — продуктивність перекомпонування складних декларативних інтерфейсів, зокрема довгих списків модулів та згортуваних карток, на пристроях середнього й нижнього цінових сегментів.

Оцінювання кожного варіанта за кожним критерієм виконується за п'ятибальною шкалою, де 5 балів відповідає повній відповідності критерію, а 1 бал — суттєвій невідповідності. Перелік критеріїв та їх вагові коефіцієнти узагальнено в таблиці 1.1.

Таблиця 1.1 – Критерії порівняльного аналізу та їх вагові коефіцієнти

Поз.	Критерій	Позначення	Вага
1	Автономність та офлайн-стійкість	K1	0,35
2	Вартість розробки та володіння	K2	0,25
3	Безпека та приватність локальних даних	K3	0,25
4	Швидкодія рендерингу інтерфейсу	K4	0,15
	Разом		1,00

Розподіл вагових коефіцієнтів критеріїв порівняльного аналізу унаочнено на рисунку 1.1.

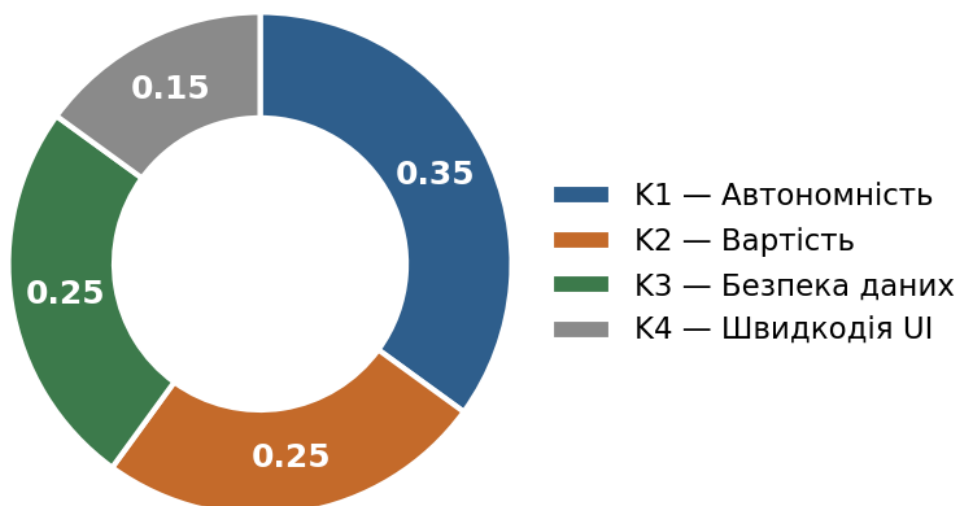


Рисунок 1.1 – Вагові коефіцієнти критеріїв порівняльного аналізу

1.4 Обґрунтування вибору фінального архітектурного рішення

Результати експертного оцінювання трьох варіантів за чотирма критеріями та обчислені зважені інтегральні показники наведено в таблиці 1.2. Зважений бал за кожним критерієм отримано множенням балу варіанта на ваговий коефіцієнт критерію, а інтегральний показник — підсумовуванням зважених балів.

Таблиця 1.2 – Матриця зваженого порівняльного аналізу варіантів

Критерій (ваговий коефіцієнт)	Вар. А	Вар. Б	Вар. В
K1 — автономність (0,35)	5	1	4
K2 — вартість (0,25)	5	2	3
K3 — безпека даних (0,25)	5	2	4
K4 — швидкодія UI (0,15)	5	5	4
Інтегральний показник	5,00	2,10	3,75

Інтегральні показники трьох варіантів реалізації наочно зіставлено на рисунку 1.2.

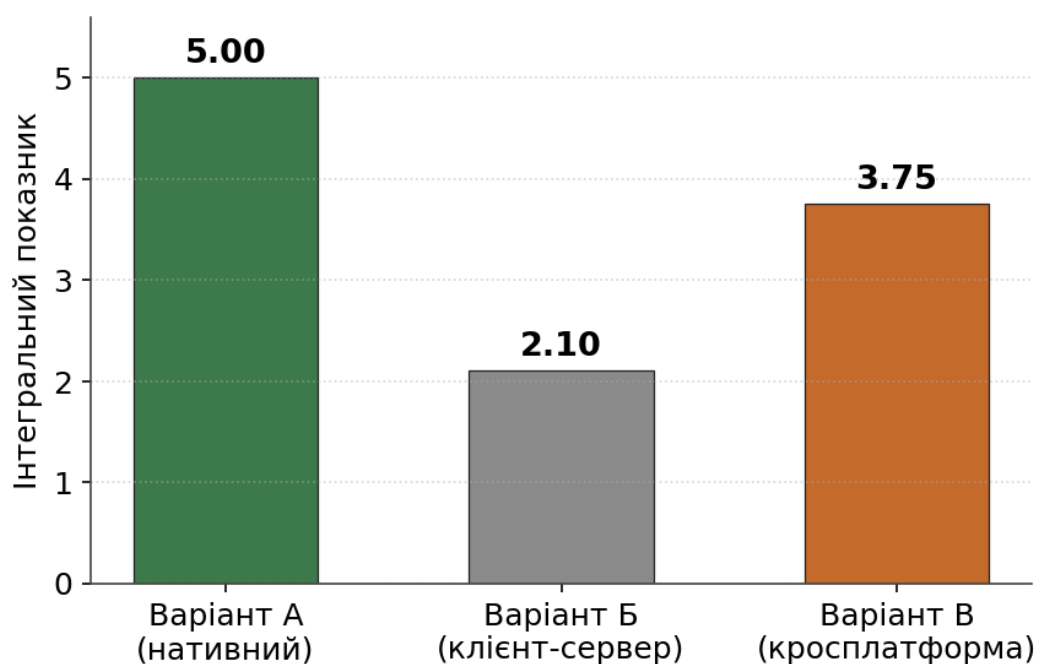


Рисунок 1.2 – Інтегральні показники варіантів реалізації системи

Як свідчить матриця, Варіант А отримав максимальний інтегральний показник 5,00, тоді як кроссплатформний Варіант В — 3,75, а клієнт-серверний Варіант Б — лише 2,10. Перевага Варіанта А формується насамперед за рахунок критеріїв із найбільшою вагою: повної автономності (K1) та захищеності локальних даних (K3), де нативне безсерверне рішення є еталонним. Варіант Б програє передусім через залежність від мережі та інфраструктури (втрата працездатності під час блекаутів) і постійні експлуатаційні витрати. Варіант В

посідає проміжну позицію, проте його ключова перевага — охоплення iOS — практично не реалізується через несумісність платформи Apple з обраною моделлю дистрибуції через «єКафедру».

Економічний аспект обґрунтування підтверджує цей висновок. Орієнтовну порівняльну оцінку сукупної вартості володіння системою протягом трирічного циклу експлуатації наведено в таблиці 1.3.

Таблиця 1.3 – Орієнтовна оцінка сукупної вартості володіння (3 роки)

Стаття витрат	Вар. А	Вар. Б	Вар. В
Трудомісткість розробки (відносна)	1,0	≈1,6	≈1,5
Експлуатація інфраструктури	0	оренда серверів / хмари	Apple Dev (\$99/рік)
Залежність від мережі	відсутня	критична	часткова
Канал дистрибуції	єКафедра (APK)	App Store / сервер	App Store + єКафедра

Варіант А не потребує жодних експлуатаційних витрат: дистрибуція здійснюється наявною системою «єКафедра», а оновлення контенту між семестрами зводиться до перезбирання APK. Варіанти Б та В, навпаки, генерують періодичні витрати (оренда серверних потужностей або щорічна вартість облікового запису розробника Apple) та збільшують трудомісткість розробки приблизно в півтора рази через потребу реалізувати додатковий серверний компонент або адаптувати кодову базу під другу платформу.

На підставі результатів зваженого аналізу та оцінки вартості володіння як оптимальне обрано Варіант А — нативний автономний Android-застосунок. Відповідно до цього рішення сформовано фінальний технологічний стек, кожен елемент якого розв’язує конкретну підзадачу в межах обраної архітектури:

- **Kotlin 2.0+** із Kotlin Symbol Processing — основна мова розробки з лаконічним синтаксисом, безпекою щодо null на рівні системи типів та повною офіційною підтримкою Google [17];

- **Jetpack Compose** (compose-bom:2025.09.01) — декларативний фреймворк побудови інтерфейсу, що забезпечує нативну швидкодію перекомпонування за критерієм K4 [11];
- **архітектура MVVM з елементами Clean Architecture** — чіткий поділ на шари даних, домену та подання, що відповідає принципу «навчання через приклад» [9, 18];
- **Room** (room-runtime:2.8.1) — реляційне ORM-сховище для функції обраних занять, що працює повністю локально [14];
- **Kotlin Coroutines** (1.10.2) та **StateFlow** — реактивне керування асинхронними потоками даних і станами інтерфейсу [16];
- **Hilt** (hilt-android:2.57.2) — контейнер впровадження залежностей, що формує граф об'єктів і керує їхніми життєвими циклами [10];
- **DataStore Preferences** (1.1.7) — асинхронне неблокуюче сховище налаштувань для збереження обраної мови інтерфейсу [8].

Сукупність зазначених технологій повністю задовольняє всі функційні та нефункційні вимоги, сформульовані у підрозділі 1.1, забезпечуючи автономність, нульову вартість володіння інфраструктурою, захищеність локальних даних та нативну швидкодію інтерфейсу.

Висновки до розділу 1

У розділі сформульовано інженерну задачу розробки автономного мобільного навчально-методичного комплексу та визначено сукупність функційних і нефункційних вимог до системи, з-поміж яких визначальними для умов вищої освіти України визнано автономність, захищеність локальних даних та мінімальну вартість володіння.

Виконано порівняльний аналіз трьох альтернативних архітектурних варіантів — нативного автономного, клієнт-серверного та кросплатформного — за чотирма зваженими критеріями із застосуванням методу зваженого оцінювання. За результатами аналізу нативний автономний Android-застосунок (Варіант А)

отримав найвищий інтегральний показник 5,00 проти 3,75 у кроссплатформного та 2,10 у клієнт-серверного варіантів, що додатково підтверджено оцінкою сукупної вартості володіння протягом трирічного циклу.

На підставі обґрунтованого вибору сформовано фінальний технологічний стек (Kotlin, Jetpack Compose, MVVM, Room, Kotlin Coroutines, Hilt, DataStore), детальну програмно-архітектурну реалізацію якого розглянуто в наступних розділах.

2 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА АНАЛОГІВ

Розроблення навчально-методичного комплексу як програмного продукту потребує попереднього аналізу предметної області у двох взаємодоповнювальних площинах. Перша — методично-дистрибутивна: вивчення наявних освітніх платформ, що розв’язують подібну задачу доставлення навчального контенту, та виявлення їхніх технічних обмежень. Друга — інженерно-технологічна: обґрунтування ключових технічних рішень обраної у розділі 1 нативної архітектури шляхом порівняння конкурентних технологій побудови інтерфейсу та локального зберігання даних. Обидві площини розглянуто послідовно у підрозділах 2.1–2.3.

2.1 Детальний аналіз існуючих освітніх платформ

Світовий ринок мобільного навчання (mobile learning) демонструє стале зростання, а підготовка Android-розробників представлена низкою провідних міжнародних платформ. Для формування методологічного підґрунтя НМК детально проаналізовано три репрезентативні платформи, що охоплюють академічний, професійно-комерційний та офіційний (вендорський) сегменти ринку: Coursera, Udacity та Google Developers Training.

Coursera репрезентує академічний сегмент. Платформа у партнерстві з Мічиганським університетом та Google пропонує спеціалізації «Android App Development» та «Google UX Design Professional Certificate». Сильними сторонами є академічна глибина викладу, рецензування завдань іншими слухачами (peer-review) та визнана роботодавцями сертифікація. Технічними та методичними обмеженнями у контексті кафедри ІПЗЕ є: зміщення акценту на загальні концепції замість конкретного промислового стеку, неможливість адаптації до навчального плану дисципліни та критеріїв оцінювання кафедри, а також обов’язкова прив’язка до мережевого з’єднання — увесь контент розміщено на серверах платформи й недоступний офлайн [4].

Udacity репрезентує професійно-комерційний сегмент. Програма Nanodegree «Android Kotlin Developer» охоплює повний цикл розробки із застосуванням Kotlin, реальними проєктами та менторським супроводом. Проте платформа має високу вартість (понад 300 USD на місяць), повну відсутність прив'язки до університетського навчального плану та, як і Coursera, потребує постійного доступу до Інтернету для перегляду відеолекцій і виконання завдань [20].

Google Developers Training репрезентує офіційний вендорський сегмент. Ресурс надає безкоштовні матеріали (зокрема курс «Android Basics with Compose»), практичні codelab-завдання та сертифікацію Associate Android Developer. Перевагою є прямий зв'язок з командою розробників Android та максимальна актуальність. Обмеженнями є орієнтація переважно на самонавчання, відсутність структурування під формат університетської дисципліни з банком індивідуальних завдань і чіткими критеріями оцінювання, а також, знову ж таки, мережева залежність доступу до матеріалів [5, 6].

Узагальнення аналізу дозволяє виокремити три системні технічні обмеження, спільні для всіх розглянутих платформ і критичні саме для умов кафедри ІПЗЕ:

- **Жорстка прив'язка до мережі.** Увесь навчальний контент платформ розміщено на віддалених серверах, унаслідок чого доступ до матеріалів унеможлиблюється за відсутності стабільного Інтернет-з'єднання — ситуація, типова для умов відключень електроенергії в Україні.

- **Відсутність кастомізації під навчальний план кафедри.** Жодна платформа не передбачає прив'язки до галузевої специфіки (енергетика), банку з тридцяти предметно-орієнтованих варіантів індивідуальних завдань та деталізованих критеріїв оцінювання, ухвалених кафедрою.

- **Відсутність вбудованих офлайн-прикладів та зразкових рішень.** Платформи не надають здобувачеві компактного автономного довідника зі зразковими реалізаціями (Варіант №1), доступними безпосередньо на пристрої без переходу до зовнішніх сервісів.

Зведене порівняння платформ за ключовими характеристиками наведено в таблиці 2.1.

Таблиця 2.1 – Порівняльний аналіз платформ навчання Android-розробки

Платформа	Вартість	Технологічний стек	Критерії оцінювання	Офлайн-доступ
Coursera	Аудит / \$49+/міс	Kotlin, XML, SDK	Peer-review, тести	Немає
Udacity	≈\$300+/міс	Kotlin, Jetpack, MVVM	Менторська перевірка	Немає
Google Developers	Безкоштовно	Kotlin, Compose, Jetpack	Автотести (обмежено)	Немає
Розроблений НМК	Безкоштовно (APK)	Kotlin, Compose, MVVM, Room, Hilt	Деталізовані критерії кафедри	Повний

Як свідчить таблиця, жодна з міжнародних платформ не задовольняє водночас вимог університетської дисципліни та автономності доступу, що підтверджує доцільність розроблення власного НМК у форматі автономного мобільного застосунку.

2.2 Дослідження парадигм побудови інтерфейсу: декларативна та імперативна

Вибір способу побудови користувацького інтерфейсу є фундаментальним архітектурним рішенням, що визначає супроводжуваність, передбачуваність і швидкодію застосунку. Сучасна Android-розробка пропонує дві конкурентні парадигми: імперативну, засновану на системі View та XML-розмітці, і декларативну, реалізовану фреймворком Jetpack Compose. У розробленому НМК застосовано виключно декларативний підхід; нижче наведено обґрунтування цього вибору.

Імперативна парадигма (View/XML). За класичного підходу структуру екрана описують у XML-файлі, після чого програмно отримують посилання на елементи (`findViewById`) і вручну змінюють їхній стан у відповідь на події. Спрощений фрагмент такого підходу наведено на рисунку 2.1.

```
// 1. Отримання посилання на елемент розмітки
val titleView = findViewById<TextView>(R.id.title)
val counterView = findViewById<TextView>(R.id.counter)
var count = 0
// 2. Ручна мутація стану елемента у відповідь на подію
button.setOnClickListener {
    count++
    counterView.text = count.toString() // оновлення UI вручну
}
```

Рисунок 2.1 – Імперативне оновлення інтерфейсу засобами View/XML

У наведеному фрагменті розробник самотійно відповідає за синхронізацію стану застосунку (`count`) із його візуальним відображенням (`counterView.text`). За зростання кількості елементів та станів така ручна синхронізація стає джерелом помилок: окремі гілки коду можуть забути оновити певний віджет, унаслідок чого інтерфейс відображає неузгоджений стан.

Декларативна парадигма (Jetpack Compose). Compose ґрунтується на принципі «єдиного джерела істини» (Single Source of Truth): інтерфейс є чистою функцією від стану, а не послідовністю команд його зміни. Розробник описує, **яким** має бути UI для заданого стану, а фреймворк самотійно визначає, які саме фрагменти потрібно оновити. Еквівалент попереднього прикладу у декларативному стилі наведено на рисунку 2.2 [11].

```
@Composable
fun Counter() {
    // Стан є єдиним джерелом істини
    var count by remember { mutableStateOf(0) }
    // UI декларативно описує себе через стан
    Button(onClick = { count++ }) {
        Text(text = count.toString())
    }
}
```

Рисунок 2.2 – Декларативний опис інтерфейсу засобами Jetpack Compose

Тут зміна значення `count` автоматично спричиняє перекомпонування (recomposition) лише тих функцій, що читають цей стан. Розробник позбавлений потреби вручну синхронізувати інтерфейс, що усуває цілий клас помилок неузгодженого стану.

Ключовим механізмом ефективності Compose є **інтелектуальне перекомпонування** (smart recomposition). Фреймворк не перемальовує екран повністю, а відстежує області читання стану — області перекомпонування

(recomposition scopes) — і повторно виконує лише ті composable-функції, вхідні дані яких змінилися. Функцію, вхідні параметри якої не змінилися й мають стабільний тип, фреймворк пропускає (skipping). Таким чином продуктивність прямо залежить від коректного визначення **стабільності** типів [7].

Compose автоматично вважає тип стабільним, якщо його екземпляри незмінні (immutable) або повідомляють про зміни свого стану. Примітивні типи та String стабільні за визначенням, тоді як інтерфейс List фреймворк трактує як потенційно нестабільний, оскільки за посиланням List може приховуватися змінна реалізація. Для явного декларування стабільності застосовують анотації-контракти:

- **@Immutable** — гарантує, що всі публічні властивості типу не змінюються протягом життя екземпляра; дозволяє фреймворку безпечно пропускати перекомпонування;
- **@Stable** — слабша гарантія: властивості можуть змінюватися, але про будь-яку зміну фреймворк буде повідомлено через систему стану.

Моделі даних розробленого НМК спроектовано як незмінні data class (наприклад, ModuleData з полями id, title та списком практичних занять), що відповідає принципам незмінності та створює сприятливі умови для пропускання зайвих перекомпонувань. Застосування анотацій @Immutable/@Stable до колекційних полів моделей є визнаним напрямом подальшої оптимізації швидкодії інтерфейсу.

2.3 Аналіз технологій локального збереження даних на пристрої

Обрана автономна архітектура потребує двох незалежних механізмів локального зберігання: структурованого реляційного сховища для переліку обраних занять та сховища пар «ключ–значення» для налаштувань інтерфейсу. Для кожного з них розглянуто конкурентні технології.

Реляційне зберігання: сирий SQLite, Room та Realm. Платформа Android містить вбудовану СКБД SQLite, доступ до якої можливий трьома способами.

Прямий доступ через `SQLiteOpenHelper` потребує ручного написання SQL-рядків, обробки об'єктів `Cursor` та відсутній контроль коректності запитів на етапі компіляції, що є джерелом помилок виконання. Бібліотека `Room` є об'єктно-реляційним відображенням (ORM) поверх `SQLite`: вона зберігає прозорість SQL, але додає перевірку синтаксису запитів під час компіляції, автоматичне відображення рядків таблиці на об'єкти `Kotlin` та інтеграцію з `Flow` для реактивного відстеження змін. Альтернатива у вигляді NoSQL-бібліотеки `Realm` пропонує об'єктну модель із «живими» об'єктами, проте накладає обмеження на роботу з потоками виконання, збільшує розмір APK та відходить від звичної реляційної моделі, яку студенти опановують у межах суміжних дисциплін.

Для розробленого НМК обрано `Room` як оптимальний компроміс між безпекою, прозорістю та продуктивністю. Реалізацію сутності та об'єкта доступу до даних наведено на рисунку 2.3 [14].

```
@Entity(tableName = "starred_practices")
data class StarredAssignmentEntity(
    @PrimaryKey(autoGenerate = false) val id: Int,
)

@Dao
interface StarredAssignmentDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun starAssignment(assignment: StarredAssignmentEntity)
    @Query("DELETE FROM starred_practices WHERE id = :id")
    suspend fun unstarAssignment(id: Int)
    @Query("SELECT * FROM starred_practices")
    fun getAllStarredPractices(): Flow<List<StarredAssignmentEntity>>
}
```

Рисунок 2.3 – Сутність та DAO реляційного сховища `Room`

Анотація `@Entity(tableName = "starred_practices")` оголошує таблицю; `@PrimaryKey(autoGenerate = false)` визначає поле `id` первинним ключем без автогенерації (ідентифікатор заняття відомий заздалегідь). Анотація `@Insert(onConflict = OnConflictStrategy.REPLACE)` задає стратегію заміщення за повторного додавання, що робить операцію ідемпотентною. Модифікатор `suspend` інтегрує операції з корутинами, виконуючи їх поза головним потоком, а метод `getAllStarredPractices()` повертає `Flow`, завдяки чому інтерфейс автоматично реагує на зміни таблиці.

Зберігання налаштувань: SharedPreferences та DataStore. Традиційний механізм SharedPreferences зберігає пари «ключ–значення», проте має суттєві вади: синхронний характер операцій читання (ризик блокування головного потоку), відсутність типобезпечного інтерфейсу та механізму сигналізації про помилки введення-виведення. Сучасна заміна — DataStore Preferences — усуває ці недоліки: операції є асинхронними та транзакційними, виконуються на корутинах, а дані надаються у вигляді реактивного потоку Flow. У розробленому НМК DataStore застосовано для збереження обраної мови інтерфейсу; відповідну реалізацію наведено на рисунку 2.4 [8].

```
private val Context.dataStore by preferencesDataStore(name = "settings")

class LanguagePreferenceManager @Inject constructor(
    @ApplicationContext private val context: Context
) {
    private val LANGUAGE_CODE = stringPreferencesKey("language_code")
    suspend fun saveLanguage(code: String) {
        context.dataStore.edit { it[LANGUAGE_CODE] = code }
    }
    val languageCode: Flow<String> = context.dataStore.data
        .catch { if (it is IOException) emit(emptyPreferences()) else throw it }
        .map { it[LANGUAGE_CODE] ?: "uk" }
}
```

Рисунок 2.4 – Реактивне зберігання налаштувань засобами DataStore

Делегат preferencesDataStore створює єдиний екземпляр сховища з ім'ям settings. Метод saveLanguage асинхронно та транзакційно записує код мови через блок edit. Властивість languageCode надає мову як Flow: оператор catch коректно опрацьовує можливі помилки введення-виведення, повертаючи порожні налаштування замість аварійного завершення, а оператор map повертає українську мову (uk) як значення за замовчуванням. Реактивна природа Flow забезпечує перемикання мови інтерфейсу без перезапуску активності, що детально розглянуто в розділі 4.

Висновки до розділу 2

За результатами аналізу трьох провідних освітніх платформ (Coursera, Udacity, Google Developers Training) виявлено три системні технічні обмеження, спільні для всіх них: жорстку прив'язку до мережі, відсутність кастомізації під навчальний план кафедри та відсутність вбудованих офлайн-прикладів, що підтверджує доцільність розроблення автономного НМК.

Обґрунтовано вибір декларативної парадигми Jetpack Compose над імперативною системою View/XML на основі принципу єдиного джерела істини та механізму інтелектуального перекомпонування; розглянуто роль контрактів стабільності `@Immutable` і `@Stable` для оптимізації швидкодії інтерфейсу.

Виконано порівняння технологій локального зберігання: для структурованих даних обрано Room як ORM із компіляційною перевіркою запитів та реактивним Flow, для налаштувань — DataStore як асинхронну типобезпечну заміну SharedPreferences. Кожне рішення проілюстровано фрагментами реальної кодової бази застосунку.

3 МЕТОДИЧНЕ ПРОЄКТУВАННЯ ТА СТРУКТУРУВАННЯ КУРСУ

Якість навчально-методичного комплексу визначається не лише технічною досконалістю програмної реалізації, а й педагогічною обґрунтованістю структури навчального контенту. У цьому розділі розкрито методичне проєктування курсу: архітектуру навчального контенту та його ієрархічну організацію, застосування таксономії Блума й концепції конструктивного вирівнювання для побудови рівнів складності завдань, а також інженерно-енергетичну специфіку практичних завдань на прикладі коду зразкового рішення Лабораторної роботи №7.

3.1 Архітектура навчального контенту

Концептуально розроблений НМК є інтерактивною електронною книгою — цифровим підручником, у якому навчальний матеріал організовано як строго формалізовану ієрархічну структуру даних, а не як суцільний текст. Така формалізація є передумовою програмної обробки контенту: кожен елемент курсу відображається на відповідну модель даних мовою Kotlin, що уможливлює його десеріалізацію, кешування та реактивне подання в інтерфейсі.

Ієрархія навчального контенту має три рівні вкладеності: курс поділяється на дев'ять тематичних **модулів**, кожен модуль містить три практичні **заняття** (разом двадцять сім занять), а кожне заняття складається з дванадцяти обов'язкових **інформаційних блоків**. Перелік модулів та їх тематичне наповнення наведено в таблиці 3.1.

Таблиця 3.1 – Тематична структура дисципліни (9 модулів, 27 занять)

№	Назва модуля	Ключові технології занять
1	Вступ і підготовка середовища	Android Studio, Kotlin, Compose, Gradle
2	Основи UI/UX мобільного застосунку	Екрани, навігація, форми, валідація
3	Робота з даними та архітектура	MVVM, StateFlow, Room, Repository
4	Інтеграція з зовнішніми системами	REST API, Retrofit, JWT, обробка помилок
5	Робота з живими даними	Flow, Coroutines, LazyColumn, пошук
6	Взаємодія з пристроями та сенсорами	GPS, карти, сенсори, сповіщення
7	Візуалізація даних	Графіки, кастомні компоненти, анімації
8	Робота з файлами та мультимедіа	CSV/JSON, фото, генерація PDF
9	Завершення проєкту	Unit/UI-тести, моніторинг, реліз APK

Ієрархічну структуру навчального контенту унаочнено на рисунку 3.1.



Рисунок 3.1 – Ієрархічна структура навчального контенту

Структуру окремого практичного заняття формалізовано моделлю даних `PracticalAssignmentData`, наведеною на рисунку 3.2. Поля цієї моделі і є дванадцятьма обов'язковими інформаційними блоками заняття.

```
data class PracticalAssignmentData(  
    val id: Int, // ідентифікатор заняття  
    val theme: String, // 1. тема  
    val goal: String, // 2. мета (результати навчання)  
    val intro: String, // 3. огляд заняття  
    val theory: String, // 4. теоретична частина  
    var individualAssignments: List<String>, // 5. варіанти (30)  
    val additionalAssignments: List<AdditionalAssignmentData>, // 6. рівні  
    val reviewCriteria: List<ReviewCriteriaData>, // 7. критерії  
    val reviewQuestions: List<ReviewQuestionData>, // 8. контрольні питання  
    val homework: String, // 9. домашнє завдання  
    var resourcesUsedData: List<ResourcesData>, // 10. використані ресурси  
    var additionalResourcesData: List<ResourcesData>, // 11. додаткові ресурси  
    val assignmentFullCode: AssignmentFullCode, // 12. зразковий код  
)
```

Рисунок 3.2 – Модель даних практичного заняття як дванадцять інформаційних блоків

Така формалізація забезпечує методичну узгодженість: усі двадцять сім занять мають ідентичний набір блоків, що полегшує як викладання, так і самостійне опанування. Дванадцятий блок — зразковий код — структуровано окремою

моделлю `AssignmentFullCode`, що містить ілюстрації реалізацій трьох рівнів складності (`basic_level_photos`, `medium_level_photos`, `advanced_level_photos`), схему структури проєкту (`structure_photo`) та узагальнення `AssignmentSummary` із шести аналітичних полів: загальна ідея, структура та компоненти, застосовані підходи, взаємодія у застосунку, підсумок і посилання на репозиторій GitHub.

Концепція інтерактивної книги виявляється і в способі подання інформаційних блоків: дванадцять блоків заняття згруповано у три тематичні вкладки інтерфейсу — «Огляд» (тема, мета, огляд і теоретична частина), «Практика» (варіанти індивідуальних завдань та три рівні додаткових завдань) і «Ресурси» (використані й додаткові ресурси, посилання на GitHub). Такий поділ узгоджує структуру даних з логікою навчальної діяльності: спочатку здобувач опановує теорію, потім переходить до завдань, а наприкінці — до джерел поглибленого вивчення.

Блок варіантів індивідуальних завдань (`individualAssignments`) містить тридцять предметно-орієнтованих формулювань, єдиних для всіх практичних занять. Принцип технічної однорідності за тематичної різноманітності означає, що всі варіанти одного заняття потребують ідентичного технологічного стеку, проте відрізняються прикладною галуззю (енергетика, транспорт, медицина, фінанси тощо). Це забезпечує справедливість оцінювання та водночас унеможлиблює пряме копіювання рішень між здобувачами.

3.2 Таксономія Блума та концепція конструктивного вирівнювання

Методичне проєктування завдань ґрунтується на двох взаємопов'язаних педагогічних концепціях: таксономії освітніх цілей Блума та принципі конструктивного вирівнювання (`constructive alignment`). Таксономія Блума впорядковує когнітивні навички за зростанням складності — від відтворення знань до творення нового, — а конструктивне вирівнювання вимагає узгодженості між

трьома компонентами навчання: сформульованою метою, навчальною діяльністю та критеріями оцінювання [1, 2].

У розробленому НМК таксономію Блума реалізовано алгоритмічно: для кожного заняття додаткові завдання поділено на три рівні складності, формалізовані переліком `AdditionalAssignmentLevel`, наведеним на рисунку 3.3.

```
enum class AdditionalAssignmentLevel { BASIC, MEDIUM, ADVANCED }

data class AdditionalAssignmentData(
    val level: AdditionalAssignmentLevel,
    val assignment: String,
)

data class ReviewCriteriaData(
    val level: AdditionalAssignmentLevel,
    val reviewCriteria: String,
)
```

Рисунок 3.3 – Формалізація рівнів складності та прив’язаних до них критеріїв

Принциповим є те, що той самий перелік `AdditionalAssignmentLevel` застосовано одночасно у двох моделях — `AdditionalAssignmentData` (саме завдання) та `ReviewCriteriaData` (критерій його оцінювання). Цим на рівні структури даних реалізовано конструктивне вирівнювання: кожному рівню завдання однозначно відповідає рівень критерію оцінювання, унаслідок чого діяльність здобувача та її оцінювання залишаються узгодженими. Відповідність рівнів складності когнітивним рівням таксономії Блума узагальнено в таблиці 3.2 [1].

Таблиця 3.2 – Відповідність рівнів складності таксономії Блума

Рівень (enum)	Когнітивний рівень за Блумом	Характер діяльності здобувача
BASIC	Запам’ятовування, розуміння, застосування	Відтворення зразкового рішення за наданим прикладом
MEDIUM	Аналіз	Адаптація рішення: додавання нової логіки та умов
ADVANCED	Оцінювання, творення	Розширення та дослідження: нові підсистеми, оптимізація

Відповідність рівнів складності когнітивним рівням таксономії Блума унаочнено на рисунку 3.4.



Рисунок 3.4 – Піраміда таксономії Блума та рівні складності завдань

Така триступенева структура формує стійкі когнітивні навички проєктування кіберфізичних систем: здобувач послідовно переходить від відтворення готового зразка до самостійного аналізу й, зрештою, до творчого розширення системи. Перехід між рівнями є поступовим, що знижує когнітивне навантаження та відповідає принципу диференційованого навчання.

3.3 Інженерно-енергетична специфіка практичних завдань

Прив'язка тематики завдань до предметної галузі енергетики відповідає науковому профілю кафедри інженерії програмного забезпечення в енергетиці та формує у здобувачів розуміння специфіки ІТ-розробки для кіберфізичних енергетичних систем. Розглянемо реалізацію цього принципу на прикладі зразкового рішення Лабораторної роботи №7 (модуль 7 — «Візуалізація даних»), предметом якої є моніторинг потужності сонячної електростанції.

Зразкове рішення демонструє триступеневу прогресію складності засобами трьох окремих компонентів `ViewModel`, що безпосередньо реалізують рівні таксономії Блума з підрозділу 3.2. На **базовому рівні** `BasicViewModel` зберігає єдиний реактивний стан — поточну потужність станції — і надає метод його оновлення. Здобувач лише відтворює патерн «стан → подія → оновлення» без додаткової логіки.

На **середньому рівні** завдання вимагає аналізу: `MediumViewModel` розширює базову модель, обчислюючи експлуатаційний статус підстанції на основі порогових значень потужності. Відповідний фрагмент наведено на рисунку 3.5.

```
class MediumViewModel : ViewModel() {
    private val _currentPower = MutableStateFlow(0)
    val currentPower: StateFlow<Int> = _currentPower.asStateFlow()
    private val _status = MutableStateFlow("Норма")
    val status: StateFlow<String> = _status.asStateFlow()

    fun updatePower(newPower: Int) {
        _currentPower.value = newPower
        _status.value = when {
            newPower < 30 -> "Низька потужність"
            newPower > 70 -> "Висока потужність"
            else -> "Норма"
        }
    }
}
```

Рисунок 3.5 – Визначення статусу підстанції за пороговими значеннями потужності (середній рівень)

Тут реалізовано просту, але показову модель моніторингу промислового енергетичного об’єкта: умовний вираз `when` класифікує поточну потужність за трьома станами — «Низька потужність» (нижче 30 кВт), «Висока потужність» (понад 70 кВт) та «Норма». Два незалежні потоки стану (`currentPower` і `status`) демонструють, як одна подія оновлення синхронно змінює кілька похідних характеристик системи — типовий патерн для систем телеметрії.

На **поглибленому рівні** `AdvancedViewModel` додає асинхронне отримання даних, що моделює реальну взаємодію з віддаленим контролером станції. Відповідний фрагмент наведено на рисунку 3.6.

```

fun refreshPower() {
    viewModelScope.launch {           // корутина у скоупі ViewModel
        _isLoading.value = true       // індикація стану завантаження
        delay(1000)                   // імітація мережевої затримки
        val newPower = (0..100).random()
        _power.value = newPower
        _isLoading.value = false
    }
}

```

Рисунок 3.6 – Асинхронне оновлення показників станції через корутину (поглиблений рівень)

Запуск корутини у `viewModelScope` гарантує її автоматичне скасування при знищенні компонента `ViewModel`, унеможливаючи витoki ресурсів. Оператор `delay(1000)` імітує затримку опитування контролера, а додатковий стан `isLoading` дозволяє інтерфейсу відобразити індикатор завантаження. Таким чином, поглиблений рівень знайомить здобувача з повним життєвим циклом асинхронного запиту до енергетичного об'єкта.

Наведена прогресія засвідчує, що інженерно-енергетична специфіка завдань не є формальним декоруванням: кожен рівень складності спирається на реальні розрахункові та моніторингові сценарії (прогнозування генерації сонячних панелей, класифікація стану підстанції, асинхронне опитування контролера), що формують у здобувачів практичні навички розроблення програмного забезпечення для енергетичної галузі.

Прикметно, що зразкове рішення побудовано на тому самому архітектурному фундаменті, що й сам застосунок-НМК: реактивні стани оголошено через `MutableStateFlow/StateFlow`, а інтерфейс підписується на них оператором `collectAsState`. Цим реалізовано наскрізний педагогічний принцип «навчання через приклад» — здобувач вивчає патерн MVVM на компактному енергетичному застосунку, тоді як той самий патерн у промисловому масштабі застосовано у застосунку, яким він користується для навчання. Детальний розбір цього патерну в межах самого НМК наведено в розділі 4.

Висновки до розділу 3

Спроектовано трирівневу ієрархічну архітектуру навчального контенту (9 модулів → 27 занять → 12 інформаційних блоків), формалізовану моделями даних мовою Kotlin, що уможливлює програмну обробку, кешування та реактивне подання матеріалу.

Реалізовано алгоритмічний поділ завдань на три рівні складності (BASIC, MEDIUM, ADVANCED) у відповідності до таксономії Блума; застосуванням єдиного переліку рівнів водночас у моделях завдання та критерію оцінювання на структурному рівні забезпечено конструктивне вирівнювання навчання.

На прикладі коду зразкового рішення Лабораторної роботи №7 продемонстровано інженерно-енергетичну специфіку завдань: моніторинг потужності сонячної станції реалізовано як прогресію від відтворення базового патерну стану через порогову класифікацію статусу підстанції до асинхронного опитування контролера, що формує у здобувачів навички проєктування кіберфізичних енергетичних систем.

4 ПРОГРАМНО-АРХІТЕКТУРНА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ

Цей розділ є центральним у роботі та присвячений детальному розбору програмно-архітектурної реалізації мобільного застосунку-НМК. Виклад побудовано «згори донизу»: спершу розкрито глобальну архітектурну схему та життєвий цикл даних, після чого пошарово, на рівні окремих класів та функцій, проаналізовано реалізацію шару даних і презентаційного шару, підсистему безсерверної локалізації та механізм впровадження залежностей. Аналіз спирається безпосередньо на кодову базу застосунку, кожен суттєвий фрагмент якої оформлено окремим рисунком із поясненням прийнятих інженерних рішень.

4.1 Опис глобальної архітектурної схеми застосунку

Архітектуру застосунку побудовано за патерном MVVM з елементами Clean Architecture та поділено на три логічні шари: презентаційний (Presentation), доменний (Domain) і шар даних (Data). Наскрізним інфраструктурним механізмом виступає контейнер впровадження залежностей Hilt. Узагальнену архітектурну схему застосунку наведено на рисунку 4.1 [9, 18].

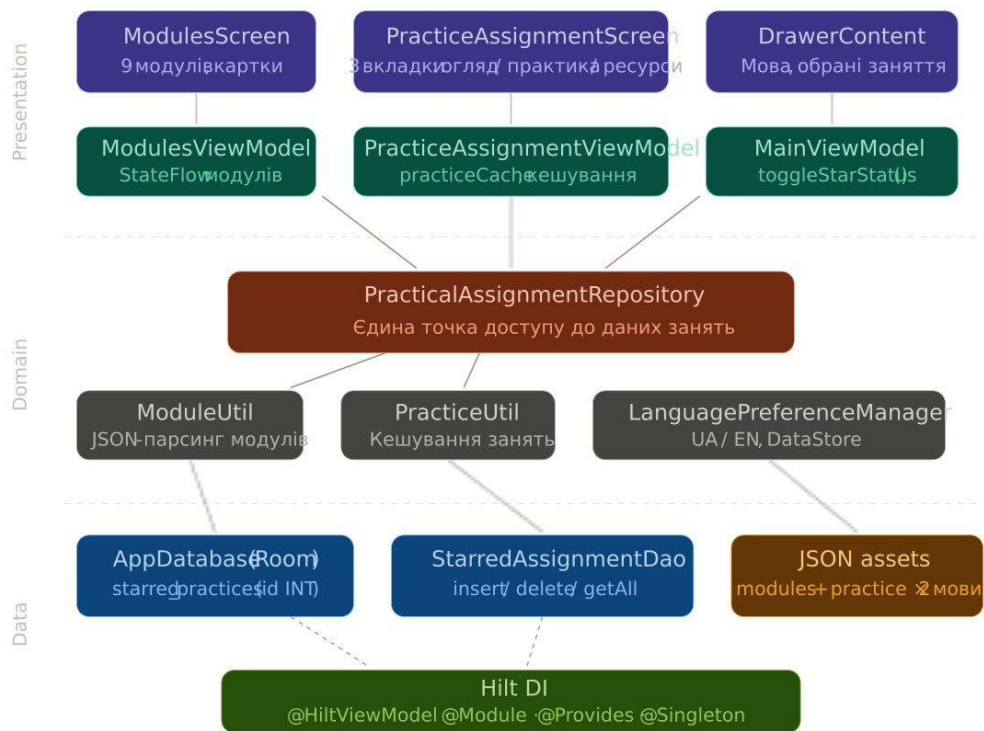


Рисунок 4.1 – Глобальна архітектурна схема застосунку

Презентаційний шар утворюють три екрани з прив'язаними до них компонентами ViewModel. Екран ModulesScreen відображає дев'ять модулів у вигляді карток і отримує дані від ModulesViewModel. Екран PracticeAssignmentScreen із трьома вкладками (огляд, практика, ресурси) обслуговується PracticeAssignmentViewModel, який містить кеш занять practiceCache. Бічне меню DrawerContent, що відповідає за перемикання мови та перелік обраних занять, керується MainViewModel через метод toggleStarStatus.

Доменний шар виступає посередником між презентацією та джерелами даних. Його ядром є репозиторій PracticalAssignmentRepository — єдина точка доступу до даних занять. Допоміжні компоненти доменного шару: утиліта ModuleUtil (парсинг JSON модулів), утиліта PracticeUtil (завантаження та кешування занять) і LanguagePreferenceManager (керування обраною мовою через DataStore).

Шар даних містить локальну реляційну базу `AppDatabase` на основі `Room` з об'єктом доступу `StarredAssignmentDao` та статичні JSON-ресурси (`modules` та `practice` у двох мовних варіантах). Контейнер `Hilt` наскрізно зв'язує всі шари, надаючи синглтон-екземпляри через анотації `@HiltViewModel`, `@Module`, `@Provides` та `@Singleton`.

Життєвий цикл даних реалізує однонаправлений потік (`Unidirectional Data Flow`): стан рухається від шару даних через домен до презентації, а події користувача — у зворотному напрямку. Після запуску активності `ModulesActivity` компонент `ModulesViewModel` звертається до `ModuleUtil`, який зчитує та десеріалізує файл модулів; список відображається у `LazyColumn`. Вибір заняття ініціює навігацію через `Navigation Compose`; `PracticeAssignmentViewModel` повертає дані заняття з кешу або делегує завантаження `PracticeUtil`. Позначення зіркою спричиняє оновлення таблиці `Room` через репозиторій, а оскільки `DAO` повертає `Flow`, перелік обраних занять оновлюється автоматично. Перемикання мови записує новий код у `DataStore`, що реактивно перебудовує локалізований контекст. Деталі реалізації кожного шару розкрито у підрозділах 4.2–4.5.

4.2 Реалізація шару даних (Data Layer)

Шар даних розв'язує дві незалежні задачі: десеріалізацію статичного навчального контенту з локальних ресурсів та збереження динамічного користувацького стану (обраних занять) у реляційній базі. Реалізацію цих механізмів розглянуто у трьох підпунктах.

4.2.1 Десеріалізація локальних JSON-ресурсів

Навчальний контент зберігається у каталозі ресурсів `res/raw` у вигляді JSON-файлів, а його перетворення на об'єкти `Kotlin` виконує бібліотека `Gson`. Завантаження списку модулів реалізує утиліта `ModuleUtil`, наведена на рисунку 4.2.

```

@Singleton
class ModuleUtil @Inject constructor(
    @ApplicationContext private val appContext: Context
) {
    private val gson = Gson()
    private val cache = mutableMapOf<String, ModulesListData>()

    fun getModules(languageCode: String): List<ModuleData> =
        getOrLoad(languageCode).modules

    private fun getOrLoad(languageCode: String) =
        cache.getOrPut(languageCode) { loadModules(languageCode) }

    private fun loadModules(languageCode: String): ModulesListData {
        val name = "modules_$languageCode"
        var resId = appContext.resources.getIdentifier(name, "raw",
appContext.packageName)
        if (resId == 0) resId = appContext.resources
            .getIdentifier("modules_uk", "raw", appContext.packageName)
        appContext.resources.openRawResource(resId).use { stream ->
            return gson.fromJson(InputStreamReader(stream),
ModulesListData::class.java)
        }
    }
}

```

Рисунок 4.2 – Завантаження та десеріалізація модулів утилітою ModuleUtil

Анотація `@Singleton` гарантує єдиний екземпляр утиліти на весь застосунок, а конструкторне впровадження `@Inject` із кваліфікатором `@ApplicationContext` надає контекст рівня застосунку. Метод `getModules` повертає список модулів для заданого коду мови, делегуючи роботу методу `getOrLoad`, який реалізує кешування ідіомою `getOrPut`. Метод `loadModules` формує ім'я ресурсу за шаблоном `modules_{мова}` та визначає його числовий ідентифікатор функцією `getIdentifier`; за відсутності перекладу спрацьовує запасний український варіант (`modules_uk`), що убезпечує застосунок від збою за неповного перекладу. Конструкція `openRawResource(...).use { }` гарантує автоматичне закриття потоку навіть у разі винятку, після чого `gson.fromJson` десеріалізує вміст у кореневу модель `ModulesListData`.

Модель навчального контенту утворює трирівневу ієрархію незмінних `data class`, наведену на рисунку 4.3.

```
data class ModulesListData(val modules: List<ModuleData>)

data class ModuleData(
    val id: Int,
    val title: String,
    val practicals: List<PracticalListData>
)

data class PracticalListData(val id: Int, val title: String)
```

Рисунок 4.3 – Ієрархія моделей навчального контенту

Кореневий контейнер `ModulesListData` містить список модулів `ModuleData`, кожен з яких має ідентифікатор, заголовок та перелік занять `PracticalListData` з мінімальними метаданими (ідентифікатор і заголовок). Така мінімалізація моделі списку є свідомим рішенням: на головному екрані відображаються лише назви занять, тоді як повний зміст заняття завантажується окремо й лише за потреби. Повний зміст конкретного заняття описує модель `PracticalAssignmentData` (дванадцять інформаційних блоків, розглянутих у підрозділі 3.1), яку завантажує утиліта `PracticeUtil`, наведена на рисунку 4.4.

```

@Singleton
class PracticeUtil @Inject constructor(
    @ApplicationContext private val context: Context,
) {
    private val gson: Gson = Gson()

    fun getPractice(practiceId: Int, languageCode: String): PracticalAssignmentData {
        val fileName = "practice_${practiceId}_${languageCode}"
        val resId = context.resources.getIdentifier(fileName, "raw",
context.packageName)
        if (resId == 0) throw IllegalArgumentException("practice $practiceId not
found")
        context.resources.openRawResource(resId).use { stream ->
            return gson.fromJson(InputStreamReader(stream),
PracticalAssignmentData::class.java)
        }
    }

    fun getTranslatedPractices(practiceIds: List<Int>, languageCode: String)
: List<Pair<Int, String>> = practiceIds.mapNotNull { id ->
        val resId = context.resources.getIdentifier(
            "practice_${id}_${languageCode}", "raw", context.packageName)
        if (resId == 0) return@mapNotNull null
        context.resources.openRawResource(resId).use { stream ->
            val item = gson.fromJson(InputStreamReader(stream),
                PracticalAssignmentData::class.java)
            item.id to item.theme
        }
    }
}

```

Рисунок 4.4 – Завантаження занять утилітою PracticeUtil

Метод `getPractice` формує ім'я файлу за шаблоном `practice_{ідентифікатор}_{мова}`. На відміну від `ModuleUtil`, тут за відсутності ресурсу збуджується виняток `IllegalArgumentException`: оскільки звернення відбувається за відомим ідентифікатором з уже завантаженого списку, його відсутність свідчить про порушення цілісності даних, яке доцільно виявляти негайно. Метод `getTranslatedPractices` обслуговує бічне меню: для переліку ідентифікаторів обраних занять він оператором `mapNotNull` повертає пари «ідентифікатор – тема», безпечно пропускаючи відсутні ресурси. Завдяки цьому методу заголовки обраних занять відображаються коректною мовою інтерфейсу.

Вибір бібліотеки `Gson` для десеріалізації обумовлено зрілістю, нульовою вартістю інтеграції та відсутністю потреби в окремому плагіні компілятора, на відміну від `kotlinx.serialization`. Оскільки контент є статичним і повністю контролюється розробником, рефлексивний підхід `Gson` не створює ризиків

несподіваних структур, а незначні втрати продуктивності рефлексії повністю нівелюються описаною нижче стратегією кешування, що виконує парсинг кожного ресурсу не більше одного разу за сесію.

4.2.2 Стратегія дворівневого кешування контенту

Десеріалізація JSON є відносно дорогою операцією, тому для унеможливлення повторного зчитування та парсингу застосовано дворівневу стратегію кешування. Перший рівень — внутрішній кеш `ModuleUtil` (`cache: Map<String, ModulesListData>`), що зберігає завантажені списки модулів за кодом мови; оскільки модулів небагато й вони незмінні, такий кеш повністю усуває повторний парсинг списку. Другий рівень реалізовано безпосередньо в `PracticeAssignmentViewModel` структурою `practiceCache`, наведеною на рисунку 4.5.

```
private val practiceCache =
    mutableMapOf<Pair<Int, String>, PracticalAssignmentData>()

fun getPractice(practiceId: Int, languageCode: String): PracticalAssignmentData {
    val key = practiceId to languageCode
    return practiceCache.getOrPut(key) {
        practiceUtil.getPractice(practiceId, languageCode)
    }
}
```

Рисунок 4.5 – Кеш занять practiceCache у PracticeAssignmentViewModel

Ключем кешу слугує пара `Pair<Int, String>` — ідентифікатор заняття та код мови. Такий складений ключ дозволяє незалежно кешувати україно- та англomовну версії одного заняття: за повторного перемикання мови вже завантажена версія повертається миттєво. Ідіома `getOrPut` забезпечує лінійне завантаження — фактичне зчитування з диска та десеріалізація відбуваються лише за першого звернення до конкретної пари, а всі подальші переходи між трьома вкладками одного заняття обслуговуються з оперативної пам'яті. Оскільки `PracticeAssignmentViewModel` має область видимості `ViewModel`, кеш зберігається протягом усього перебування на екрані заняття та переживає зміну конфігурації пристрою, проте автоматично звільняється при знищенні компонента, не спричиняючи витоків пам'яті.

4.2.3 Реляційне сховище Room та репозиторій

На відміну від статичного контенту, перелік обраних занять є динамічним користувацьким станом, тому зберігається у локальній реляційній базі Room. Сутність таблиці та об'єкт доступу до даних наведено на рисунку 4.6.

```

@Database(entities = [StarredAssignmentEntity::class], version = 1, exportSchema =
false)
abstract class AppDatabase : RoomDatabase() {
    abstract fun practicalAssignmentMinimalDao(): StarredAssignmentDao
}

@Entity(tableName = "starred_practices")
data class StarredAssignmentEntity(
    @PrimaryKey(autoGenerate = false) val id: Int,
)

@Dao
interface StarredAssignmentDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun starAssignment(assignment: StarredAssignmentEntity)
    @Query("DELETE FROM starred_practices WHERE id = :id")
    suspend fun unstarAssignment(id: Int)
    @Query("SELECT EXISTS(SELECT 1 FROM starred_practices WHERE id = :id LIMIT 1)")
    suspend fun isAssignmentStarred(id: Int): Boolean
    @Query("SELECT * FROM starred_practices")
    fun getAllStarredPractices(): Flow<List<StarredAssignmentEntity>>
}

```

Рисунок 4.6 – База даних AppDatabase, сутність та об'єкт доступу

Клас AppDatabase з анотацією @Database декларує перелік сутностей (єдину StarredAssignmentEntity), версію схеми та абстрактний метод доступу до DAO; параметр exportSchema = false вимикає експорт схеми, недоцільний для єдиної незмінної таблиці. Сутність StarredAssignmentEntity навмисно мінімальна — вона містить єдине поле id із анотацією @PrimaryKey(autoGenerate = false), оскільки ідентифікатор заняття відомий заздалегідь і не потребує автогенерації. Інтерфейс @Dao декларує чотири операції. Метод starAssignment з анотацією @Insert(onConflict = OnConflictStrategy.REPLACE) додає запис із стратегією заміщення, що робить операцію ідемпотентною. Метод unstarAssignment виконує параметризований запит DELETE. Метод isAssignmentStarred застосовує оптимальний запит SELECT EXISTS, який повертає булеве значення без вибирання самого рядка. Метод getAllStarredPractices повертає Flow<List<...>>: Room автоматично повторно збуджує цей потік за будь-якої зміни таблиці, завдяки чому інтерфейс реактивно відображає актуальний перелік обраних. Модифікатор suspend у методах модифікації виконує запити поза головним потоком засобами корутин.

Доступ до DAO інкапсульовано репозиторієм `PracticalAssignmentRepository`, наведеним на рисунку 4.7, що відповідає принципу єдиної точки доступу до даних.

```
class PracticalAssignmentRepository @Inject constructor(  
    private val starredDao: StarredAssignmentDao  
) {  
    fun getAllStarredPractices(): Flow<List<StarredAssignmentEntity>> =  
        starredDao.getAllStarredPractices()  
    suspend fun getStarStatus(id: Int): Boolean = starredDao.isAssignmentStarred(id)  
    suspend fun starAssignment(id: Int) =  
        starredDao.starAssignment(StarredAssignmentEntity(id))  
    suspend fun unstarAssignment(id: Int) = starredDao.unstarAssignment(id)  
}
```

Рисунок 4.7 – Репозиторій як єдина точка доступу до даних занять

Репозиторій є тонким шаром абстракції над DAO, що приховує деталі сховища від презентаційного шару та полегшує тестування (підставлення імітаційного DAO, розглянуте в розділі 6). Залежність `starredDao` впроваджується конструктором. Принципово, що Room застосовано виключно для зберігання обраних занять — увесь навчальний контент залишається статичним у JSON-ресурсах, що мінімізує розмір бази, спрощує схему до однієї таблиці та зберігає повну автономність застосунку.

4.3 Реалізація презентаційного шару (Presentation Layer)

Презентаційний шар реалізовано за патерном MVVM: компоненти `ViewModel` зберігають і трансформують стан, а `composable`-функції декларативно його відображають. Реалізацію розглянуто у чотирьох підпунктах: керування станом, розмежування реактивних примітивів, навігація та структура компонентів інтерфейсу.

4.3.1 Керування станом у компонентах `ViewModel`

Найпростішим компонентом є `ModulesViewModel` (рисунок 4.8), що керує станом розгортання карток модулів на головному екрані.

```

@HiltViewModel
class ModulesViewModel @Inject constructor(
    private val moduleUtil: ModuleUtil,
) : ViewModel() {
    private val _selectedModules = mutableStateListOf<Int>()
    fun addModuleToSelected(moduleId: Int) {
        if (!_selectedModules.contains(moduleId)) _selectedModules.add(moduleId)
    }
    fun removeModuleToSelected(moduleId: Int) = _selectedModules.remove(moduleId)
    fun isModuleSelected(moduleId: Int): Boolean = _selectedModules.contains(moduleId)
    fun getModules(languageCode: String): List<ModuleData> =
        moduleUtil.getModules(languageCode)
}

```

Рисунок 4.8 – Компонент ModulesViewModel

Анотація `@HiltViewModel` реєструє компонент у графі залежностей Hilt, а утиліта `ModuleUtil` впроваджується конструктором. Перелік розгорнутих модулів зберігається у `mutableStateListOf<Int>` — спеціальній спостережуваній колекції Compose, зміни якої автоматично спричиняють перекомпонування відповідних карток. Методи додавання, видалення та перевірки інкапсулюють логіку перемикання стану картки.

Застосування `mutableStateListOf` замість `StateFlow` у цьому компоненті є свідомим рішенням: стан розгортання карток є суто екранним, не потребує перетворень чи об'єднання потоків і споживається безпосередньо у композиції. Спостережувана колекція Compose інтегрується з механізмом перекомпонування природніше, ніж потік, і не вимагає оператора `collectAsState`, що спрощує код там, де реактивний потік був би надмірним.

Центральним компонентом є `MainViewModel` (рисунок 4.9), що керує глобальним станом: обраною мовою, заголовком екрана, ідентифікатором та статусом «зірки» поточного заняття.

```

@HiltViewModel
class MainViewModel @Inject constructor(
    private val repository: PracticalAssignmentRepository,
    private val practiceUtil: PracticeUtil,
    private val prefsManager: LanguagePreferenceManager
) : ViewModel() {
    val languageCode: StateFlow<String> = prefsManager.languageCode
        .stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), "uk")
    private val _isStarred = MutableStateFlow(false)
    val isStarred: StateFlow<Boolean> = _isStarred.asStateFlow()

    fun setCurrentAssignment(id: Int, title: String) {
        _currentPracticeId.value = id; setTitle("$id. $title")
        viewModelScope.launch { _isStarred.value = repository.getStarStatus(id) }
    }
    fun toggleStarStatus() {
        val id = _currentPracticeId.value ?: return
        val newStatus = !_isStarred.value
        _isStarred.value = newStatus // оптимістичне оновлення
        viewModelScope.launch {
            try {
                if (newStatus) repository.starAssignment(id)
                else repository.unstarAssignment(id)
            } catch (e: Exception) { _isStarred.value = !newStatus } // відкат
        }
    }
}

```

Рисунок 4.9 – Керування станом та оптимістичне оновлення у MainViewModel

Властивість `languageCode` перетворює холодний потік `DataStore` на гарячий `StateFlow` оператором `stateIn`. Політика `SharingStarted.WhileSubscribed(5000)` підтримує висхідний потік активним ще п'ять секунд після зникнення останнього підписника, що дозволяє пережити короткочасну зміну конфігурації без повторного зчитування з диска, а значення `"uk"` слугує початковим до завершення першого зчитування. Метод `setCurrentAssignment` фіксує активне заняття та асинхронно запитує його статус «зірки» з репозиторію. Метод `toggleStarStatus` демонструє патерн оптимістичного оновлення: спершу негайно змінюється стан `_isStarred`, унаслідок чого інтерфейс реагує миттєво, а вже потім у корутині виконується персистентний запис; за винятку стан відкочується до попереднього значення, чим гарантується узгодженість відображення та бази даних.

4.3.2 Розмежування StateFlow та SharedFlow

Презентаційний шар чітко розрізняє два типи реактивних примітивів. StateFlow застосовано для **стану**, що завжди має поточне значення та має відновлюватися після перекомпонування (мова, заголовок, статус зірки). SharedFlow застосовано для **одноразових подій** (навігація), які не повинні повторно спрацьовувати під час перекомпонування чи зміни конфігурації. Типи подій формалізовано запечатаним інтерфейсом UiEvent, а їх збудження реалізує метод setTab (рисунок 4.10).

```
sealed interface UiEvent {
    data class Navigate(val route: String) : UiEvent
    object Back : UiEvent
}

private val _uiEvent = MutableSharedFlow<UiEvent>()
val uiEvent = _uiEvent.asSharedFlow()

fun setTab(tab: PracticeAssignmentTabDestination) {
    if (_selectedTab.value == tab) return
    _selectedTab.value = tab
    viewModelScope.launch { _uiEvent.emit(UiEvent.Navigate(/* маршрут вкладки */)) }
}
```

Рисунок 4.10 – Одноразові навігаційні події через SharedFlow

Запечатаний інтерфейс `UiEvent` типобезпечно описує дві можливі події — перехід за маршрутом та повернення назад. Метод `setTab` ігнорує повторний вибір тієї самої вкладки та збуджує подію у `SharedFlow`. Споживання подій відбувається в кореневому composable `MainView` (рисуюнок 4.11).

```
LaunchedEffect(Unit) {
    mainViewModel.uiEvent.collect { event ->
        when (event) {
            is UiEvent.Back -> navController.navigateUp()
            is UiEvent.Navigate -> navController.navigate(event.route)
        }
    }
}
```

Рисуюнок 4.11 – Збирання навігаційних подій у `MainView`

Блок `LaunchedEffect(Unit)` запускає збирач подій один раз на весь час життя композиції. Оператор `collect` отримує події з `SharedFlow` і трансліує їх у виклики `navController`. Завдяки одноразовій (а не утримувальній) природі `SharedFlow` подія навігації не відтворюється повторно при поверненні на екран чи переороту пристрою, що усуває класичну вадку випадкової повторної навігації.

4.3.3 Навігація засобами `Navigation Compose`

Навігаційний граф оголошено декларативно; маршрути типобезпечно описано запечатаним класом `Screen` (рисуюнок 4.12) [12].

```
sealed class Screen(val route: String) {
    object ModulesScreen : Screen("modules")
    object PracticeAssignmentScreen : Screen("practice/{practiceId}") {
        fun createRoute(practiceId: Int) = "practice/$practiceId"
    }
}

NavHost(navController, startDestination = Screen.ModulesScreen.route) {
    composable(Screen.ModulesScreen.route) { ModuleList(mainViewModel) }
    composable(
        route = Screen.PracticeAssignmentScreen.route,
        arguments = listOf(navArgument("practiceId") { type = NavType.IntType })
    ) { entry ->
        val practiceId = entry.arguments?.getInt("practiceId") ?: -1
        PracticeAssignmentContainer(practiceId, mainViewModel = mainViewModel)
    }
}
```

Рисуюнок 4.12 – Навігаційний граф та передача аргументів

Метод `createRoute` формує конкретний маршрут із підставленим ідентифікатором, що забезпечує типобезпечну передачу аргументів. Аргумент `practiceId` оголошено з типом `NavType.IntType`, а на приймальній стороні його видобувають із `backStackEntry`. Усередині екрана заняття вкладки реалізовано не повторною навігацією, а горизонтальним пейджером (рисунок 4.13).

```
val pagerState = rememberPagerState(
    initialPage = vm.selectedTab.value.ordinal,
    pageCount = { PracticeAssignmentTabDestination.entries.size })
Column {
    PracticeAssignmentTabs(selectedTab = vm.selectedTab.value, onTabClick = { tab ->
        vm.setTab(tab)
        coroutineScope.launch { pagerState.animateScrollToPage(tab.ordinal) }
    })
    HorizontalPager(state = pagerState, modifier = Modifier.fillMaxSize()) { page ->
        when (PracticeAssignmentTabDestination.entries[page]) {
            Overview -> PracticeAssignmentOverviewScreen(practice,
vm.selectedTab.value)
            Practice -> PracticeAssignmentPracticeScreen(practice,
vm.selectedTab.value)
            Resources -> PracticeAssignmentResourcesScreen(practice,
vm.selectedTab.value)
        }
    }
}
```

Рисунок 4.13 – Контейнер вкладок заняття на основі `HorizontalPager`

Компонент `rememberPagerState` зберігає поточну сторінку пейджера, синхронізовану з обраною вкладкою через властивість `ordinal` переліку. Дотик до вкладки одночасно оновлює стан у `ViewModel` (`setTab`) та плавно прокручує пейджер (`animateScrollToPage`); горизонтальне гортання, навпаки, змінює вкладку. Вираз `when` обирає відповідний екран для кожної сторінки. Таке поєднання забезпечує і дотик до вкладки, і природний жест гортання між «Оглядом», «Практикою» та «Ресурсами».

4.3.4 Stateful- та stateless-компоненти інтерфейсу

Бібліотеку `composable`-компонентів спроектовано за принципом піднесення стану (`state hoisting`): локальний стан інкапсулюють лише там, де він справді потрібен. Прикладом `stateful`-компонента є `CollapsibleIndexedCard` (рисунок 4.14).

```

@Composable
fun CollapsibleIndexedCard(title: String, text: String, index: Int, canExpand: Boolean)
{
    var isExpanded by rememberSaveable { mutableStateOf(false) }
    Card(
        modifier = Modifier.fillMaxWidth()
            .clip(MaterialTheme.shapes.medium)
            .clickable(enabled = canExpand) { isExpanded = !isExpanded },
        shape = RoundedCornerShape(12.dp)
    ) { /* заголовок + анімоване розкриття вмісту за isExpanded */ }
}

```

Рисунок 4.14 – Stateful-компонент CollapsibleIndexedCard

Компонент інкапсулює власний стан розгортання `isExpanded`, оголошений через `rememberSaveable`, завдяки чому стан зберігається не лише між перекомпонуваннями, а й при зміні конфігурації пристрою. Дотик перемикає стан лише за умови `canExpand`, а зміна `isExpanded` спричиняє перекомпонування з анімованим розкриттям вмісту. Натомість екранні компоненти є переважно `stateless`. Прикладом є екран огляду заняття (рисунок 4.15).

```

@Composable
fun PracticeAssignmentOverviewScreen(practice: PracticalAssignmentData, resetTrigger:
Any?) {
    val listState = rememberLazyListState()
    LaunchedEffect(resetTrigger) { listState.scrollToItem(0) }
    LazyColumn(state = listState, contentPadding = PaddingValues(16.dp),
        verticalArrangement = Arrangement.spacedBy(16.dp)) {
        item { TitledCard(title = LocalizedString(R.string.theme), text =
practice.theme) }
        item { TitledCard(title = LocalizedString(R.string.goal), text =
practice.goal) }
        item { TitledCard(title = LocalizedString(R.string.intro), text =
practice.intro) }
        item { TitledCard(title = LocalizedString(R.string.theory), text =
practice.theory) }
    }
}

```

Рисунок 4.15 – Stateless-екран огляду заняття

Екран не зберігає власної бізнес-логіки — він лише відображає передану модель `practice` у `LazyColumn`, що рендерить елементи лінійно й перевикористовує перевикористовувані компоненти `TitledCard`. Блок `LaunchedEffect(resetTrigger)` прокручує список до початку при перемиканні вкладки. Заголовки секцій отримуються через `composable LocalizedString`, що забезпечує їх двомовність (підрозділ 4.4). Загалом застосунок містить понад двадцять п'ять унікальних `composable`-компонентів, оформлених у єдиному стилі

Material Design 3; типографіку побудовано на гарнітурі WixMadeforDisplay у кількох варіантах накреслення через об'єкт AppTypography.

4.4 Механізм динамічної безсерверної локалізації

Стандартний механізм локалізації Android передбачає перезапуск активності при зміні мови, що погіршує користувацький досвід. У застосунку реалізовано кастомну підсистему, яка перемикає мову реактивно, без перезапуску Activity. Її ядром є composable-обгортка ProvideLocalizedContext та функція-розширення updateLocale (рисунок 4.16) [6].

```
@Composable
fun ProvideLocalizedContext(mainViewModel: MainViewModel = hiltViewModel(),
                           content: @Composable () -> Unit) {
    val language by mainViewModel.languageCode.collectAsState()
    val baseContext = LocalContext.current
    val localizedContext = remember(language, baseContext) {
        baseContext.updateLocale(language)
    }
    CompositionLocalProvider(LocalLocalizedContext provides localizedContext) {
        content()
    }
}

val LocalLocalizedContext = staticCompositionLocalOf<Context> {
    error("Localized Context not provided.")
}

fun Context.updateLocale(languageCode: String): Context {
    val locale = Locale(languageCode)
    Locale.setDefault(locale)
    val config = resources.configuration
    config.setLocale(locale)
    config.setLayoutDirection(locale)
    return createConfigurationContext(config)
}
```

Рисунок 4.16 – Підсистема локалізації: ProvideLocalizedContext та updateLocale

Обгортка ProvideLocalizedContext підписується на потік languageCode оператором collectAsState. За зміни мови блок remember(language, baseContext) повторно обчислюється, викликаючи updateLocale. Ця функція-розширення створює новий конфігураційний контекст: установлює локаль за замовчуванням, копіює поточну конфігурацію ресурсів, задає в ній нову локаль і напрямок розкладки, після чого функцією createConfigurationContext повертає окремий Context з оновленими ресурсами. Отриманий контекст надається вглиб

дерева композиції через `CompositionLocalProvider` та `staticCompositionLocalOf`; застосування саме статичного варіанта `CompositionLocal` є оптимальним, оскільки значення змінюється рідко. Зчитування локалізованих рядків виконує допоміжна `composable`-функція `LocalizedString` (рисунок 4.17).

```
@Composable
fun LocalizedString(@StringRes resId: Int, vararg args: Any,
                   mainViewModel: MainViewModel = hiltViewModel(),
                   localizedContext: Context = LocalizedContext.current): String
{
    val languageCode by mainViewModel.languageCode.collectAsState()
    return remember(resId, languageCode, localizedContext, *args) {
        localizedContext.getString(resId, *args)
    }
}
```

Рисунок 4.17 – Зчитування локалізованого рядка функцією `LocalizedString`

Функція `LocalizedString` отримує локалізований контекст із `CompositionLocal` та зчитує рядок методом `getString`. Блок `remember`, ключами якого є ідентифікатор ресурсу та код мови, кешує результат і перераховує його лише при зміні мови. Повний реактивний ланцюг перемикавання мови має такий вигляд: користувач обирає мову в бічному меню; `MainViewModel` через `LanguagePreferenceManager` записує код у `DataStore`; реактивний потік `languageCode` збуджує нове значення; `ProvideLocalizedContext` перекомпоновується, формуючи новий локалізований контекст; усі похідні виклики `LocalizedString` отримують оновлені рядки. Унаслідок цього інтерфейс повністю перемальовується новою мовою без перезапуску активності та без жодного мережевого запиту, що повністю відповідає вимозі автономності.

Окремої уваги заслуговують два інженерні рішення цієї підсистеми. По-перше, виклик `config.setLayoutDirection(locale)` додатково встановлює напрямок розкладки відповідно до локалі, що закладає підтримку мов із письмом справа наліво на майбутнє без змін архітектури. По-друге, застосування `staticCompositionLocalOf` замість звичайного `compositionLocalOf` є оптимізаційним: статичний варіант не відстежує читання значення й не спричиняє точкових перекомпонувальних споживачів при кожному доступі, що доцільно саме для

рідкозмінного контексту локалізації — повне перемальовування дерева відбувається лише у момент фактичної зміни мови.

4.5 Впровадження залежностей (Dependency Injection)

Для керування залежностями застосовано бібліотеку Hilt, побудовану на Dagger 2. Hilt автоматично будує граф об'єктів та керує їхніми життєвими циклами, усуваючи ручне створення залежностей і підвищуючи тестованість коду. Залежності, які неможливо впровадити через конструктор (як-от база даних), надаються модулями. Конфігурацію модуля бази даних наведено на рисунку 4.18 [10].

```
@Module
@InstallIn(SingletonComponent::class)
object DatabaseModule {
    @Provides @Singleton
    fun provideAppDatabase(@ApplicationContext context: Context): AppDatabase =
        Room.databaseBuilder(context, AppDatabase::class.java, "your_db_name").build()

    @Provides @Singleton
    fun provideDao(db: AppDatabase): StarredAssignmentDao =
        db.practicalAssignmentMinimalDao()
}
```

Рисунок 4.18 – Hilt-модуль надання бази даних та DAO

Анотація `@Module` у поєднанні з `@InstallIn(SingletonComponent::class)` встановлює модуль у глобальну область видимості застосунку. Методи з анотацією `@Provides` навчають Hilt створювати екземпляри `AppDatabase` (через `Room.databaseBuilder`) та `StarredAssignmentDao`. Анотація `@Singleton` гарантує єдиний екземпляр бази даних і DAO на весь застосунок, що оптимізує споживання оперативної пам'яті, оскільки повторне відкриття з'єднання з базою є дорогою операцією. Аналогічно побудований модуль `PreferencesModule` надає синглтон `LanguagePreferenceManager`.

Споживачами графу залежностей є компоненти `ViewModel`, позначені `@HiltViewModel` із конструкторним впровадженням `@Inject`, та активність `ModulesActivity`, позначена `@AndroidEntryPoint`. Кваліфікатор `@ApplicationContext` надає контекст рівня застосунку, що унеможливорює витoki

пам'яті, пов'язані з контекстом активності. Розмежування областей видимості — синглтон-залежності для інфраструктури (база, налаштування, утиліти) та обмежені життєвим циклом `ViewModel` стани — забезпечує раціональне використання ресурсів пристрою. Додатковою перевагою є тестованість: оскільки залежності впроваджуються ззовні, у модульних тестах репозиторій отримує підставний DAO (розділ 6).

Висновки до розділу 4

Детально розглянуто програмно-архітектурну реалізацію застосунку-НМК на рівні окремих класів та функцій. Описано глобальну архітектурну схему з поділом на презентаційний, доменний шари та шар даних, а також життєвий цикл даних за принципом однонаправленого потоку.

Розкрито реалізацію шару даних: десеріалізацію локальних JSON-ресурсів бібліотекою `Gson` із запасною мовою, трирівневу ієрархію моделей контенту, дворівневу стратегію кешування для усунення повторного парсингу та реляційне сховище обраних занять на основі `Room` з повним набором операцій DAO та доступом через єдиний репозиторій.

Проаналізовано презентаційний шар: керування станом у трьох компонентах `ViewModel` за патерном `MVVM` з оптимістичним оновленням, розмежування `StateFlow` (стан) і `SharedFlow` (одноразові події), типобезпечну навігацію `Navigation Compose` у поєднанні з горизонтальним пейджером вкладок та поділ компонентів на `stateful` і `stateless` із піднесенням стану.

Описано кастомну підсистему динамічної безсерверної локалізації на основі `ProvideLocalizedContext`, `updateLocale` та `createConfigurationContext`, що реактивно перемикає мову без перезапуску активності, а також конфігурацію впровадження залежностей засобами `Hilt` із розмежуванням областей видимості, що оптимізує споживання пам'яті та підвищує тестованість коду.

5 ВЗАЄМОДІЯ КОРИСТУВАЧА ІЗ СИСТЕМОЮ ТА СЦЕНАРІЙ ВИКОРИСТАННЯ

Цінність програмного продукту визначається не лише його внутрішньою архітектурою, а й зручністю розгортання та взаємодії з кінцевим користувачем. У цьому розділі визначено системні вимоги до застосунку, описано процедуру його дистрибуції через систему «ЄКафедра» та спроектовано людино-машинний інтерфейс із типовими сценаріями використання.

5.1 Системні вимоги програмного продукту

Застосунок-НМК розраховано на широкий парк пристроїв здобувачів, тому апаратні та програмні вимоги свідомо мінімізовано. Мінімальною підтримуваною версією платформи визначено Android 7.0 (рівень API 24, кодова назва Nougat), що охоплює переважну більшість активних пристроїв і водночас дозволяє застосовувати сучасні можливості Jetpack Compose. Цільовий та компіляційний рівні API встановлено на значення 36, що відповідає найновішим вимогам безпеки магазину застосунків. Зведені системні вимоги наведено в таблиці 5.1.

Таблиця 5.1 – Системні вимоги до програмного продукту

Характеристика	Значення
Операційна система	Android 7.0 (API 24) або новіша
Архітектура процесора	ARM64 (arm64-v8a), armeabi-v7a, x86_64
Оперативна пам'ять	від 2 ГБ (рекомендовано)
Вільний дисковий простір	до 100 МБ (застосунок та локальна БД)
Мережеве з'єднання	не потрібне
Дозволи застосунку	не запитуються

Принципово важливою системною характеристикою є повна відсутність мережевих дозволів. Файл маніфесту застосунку не містить дозволу `android.permission.INTERNET`, унаслідок чого застосунок на рівні операційної

системи позбавлений технічної можливості здійснювати мережеві запити. Відповідний фрагмент маніфесту наведено на рисунку 5.1.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
  <!-- Жодного <uses-permission> не оголошено -->
  <application
    android:name=".KotlinBasicsApp"
    android:allowBackup="true"
    android:supportsRtl="true" ... >
    <activity android:name=".activities.ModulesActivity"
      android:exported="true">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Рисунок 5.1 – Маніфест застосунку без мережевих дозволів

Така конфігурація є водночас архітектурним підтвердженням автономності (підрозділ 1.2) та гарантією приватності: оскільки застосунок не має дозволу на доступ до мережі, він фізично не здатний передавати будь-які дані користувача за межі пристрою. Локальна база даних Room зберігає лише цілочислові ідентифікатори обраних занять і має незначний обсяг, а навчальний контент у форматі JSON входить до складу пакета, тому вимоги до оперативної пам'яті та дискового простору залишаються помірними.

Вибір рівня API 24 як мінімального є компромісом між охопленням пристроїв та доступністю сучасних можливостей платформи. Нижчі рівні API підтримують ширший парк застарілих пристроїв, проте позбавлені низки можливостей середовища виконання, на які спирається Jetpack Compose. Рівень 24, навпаки, охоплює переважну більшість активних пристроїв здобувачів і водночас гарантує коректну роботу декларативного інтерфейсу, реактивних потоків та бібліотек Jetpack без потреби у додаткових шарах сумісності.

5.2 Інструкція з розгортання та встановлення

Дистрибуція застосунку здійснюється в обхід магазину Google Play — через внутрішню систему кафедри «Кафедра» у вигляді інсталяційного пакета APK. Така модель усуває залежність від облікових записів магазину та процедур

модерації, забезпечуючи контрольоване розповсюдження серед здобувачів.

Процедура встановлення для здобувача складається з таких кроків:

- увійти до системи «Кафедра» та відкрити розділ дисципліни «Розробка програмного забезпечення мобільних пристроїв»;
- завантажити інсталяційний файл застосунку (APK) на власний Android-пристрій;
- у системних налаштуваннях надати дозвіл на встановлення застосунків із цього джерела (починаючи з Android 8.0 дозвіл надається індивідуально для конкретного застосунку-джерела, а не глобально, що підвищує безпеку);
- запустити завантажений файл та підтвердити встановлення;
- відкрити застосунок — він одразу готовий до роботи без додаткового налаштування чи реєстрації.

Розмір інсталяційного пакета оптимізовано на етапі збирання. У конфігурації збирання активовано параметр `isMinifyEnabled = true`, що вмикає інструмент R8: він виконує усунення невикористаного коду (code shrinking), оптимізацію та обфускацію, а також скорочення невикористаних ресурсів. Правила збереження критичних класів задано у файлі `proguard-rules.pro`. Завдяки цьому розмір APK залишається компактним для зручності завантаження [13].

Оновлення навчального контенту відбувається за моделлю версіонування пакета: при зміні матеріалів модифікуються відповідні JSON-ресурси, збирається нова версія APK (з інкрементом `versionCode` та `versionName`), яка публікується на «Кафедрі». Здобувачі завантажують та встановлюють оновлений пакет поверх попереднього. Така процедура аналогічна виходу нового видання друкованого підручника з чіткою версіонованістю та контрольованим розповсюдженням.

5.3 Проектування людино-машинного інтерфейсу та сценарії взаємодії

Модель взаємодії формалізовано засобами діаграм варіантів використання (Use Case) нотації UML. У системі визначено двох акторів — Здобувача та

Викладача; сам застосунок є пасивним інформаційним інструментом і не виступає самостійним актором. Перелік основних варіантів використання наведено в таблиці 5.2.

Таблиця 5.2 – Варіанти використання застосунку-НМК

Актор	Варіант використання
Здобувач	Завантажити та встановити застосунок із «єКафедри»
Здобувач	Переглядати модулі та обирати практичне заняття
Здобувач	Вивчати теорію, завдання, критерії та ресурси заняття
Здобувач	Перемикаати мову інтерфейсу (українська / англійська)
Здобувач	Копіювати посилання на зразкове рішення GitHub
Здобувач	Позначати заняття зіркою для швидкого доступу
Викладач	Публікувати АРК на «єКафедрі», оцінювати роботи за критеріями

Діаграму варіантів використання застосунку наведено на рисунку 5.2.



Рисунок 5.2 – Діаграма варіантів використання застосунку-НМК

Інтерфейс організовано за принципом ієрархічної навігації, що відповідає структурі навчального контенту. Логіку основних екранів описано нижче (скріншоти інтерфейсу винесено в додатки).

Головний екран модулів (`ModulesScreen`). Відображає дев'ять навчальних модулів у вигляді згорнутих карток. Дотик до картки розгортає перелік практичних занять модуля; повторний дотик згортає його. Стан розгортання керується локально, а сам список рендериться у `LazyColumn` для ефективного відображення.

Екран практичного заняття (`PracticeAssignmentScreen`). Відкривається після вибору заняття та містить три вкладки. Вкладка «Огляд» подає тему, мету, вступ і теоретичну частину через компоненти `TitledCard`. Вкладка «Практика» відображає варіанти індивідуальних завдань та три рівні додаткових завдань. Вкладка «Ресурси» містить використані й додаткові джерела та посилання на репозиторій GitHub зі зразковим рішенням. Перемикання вкладок обробляється через одноразові навігаційні події (підрозділ 4.3) і супроводжується автоматичним прокручуванням списку до початку.

Бічне меню (`DrawerContent`). Надає доступ до перемикання мови інтерфейсу (з відображенням прапорців) та до переліку обраних занять. Вибір мови реактивно перебудовує інтерфейс без перезапуску застосунку (підрозділ 4.4).

Типовий наскрізний сценарій підготовки здобувача до практичної роботи має такий вигляд: здобувач відкриває головний екран, розгортає потрібний модуль і обирає заняття; на вкладці «Огляд» ознайомлюється з теорією, за потреби перемикає мову на англійську; переходить на вкладку «Практика», де вивчає своє індивідуальне завдання та рівні складності; відкриває вкладку «Ресурси» й копіює посилання на зразкове рішення GitHub; позначає заняття зіркою для швидкого повернення через бічне меню. Цей сценарій охоплює повний цикл — від ознайомлення з темою до отримання завдання та зразка реалізації — і підтверджує роль застосунку як «розумного підручника», завжди доступного на пристрої здобувача.

З погляду діаграми послідовності внутрішня взаємодія компонентів у межах цього сценарію розгортається так: дотик до заняття ініціює навігацію, після чого `PracticeAssignmentViewModel` повертає дані заняття з кешу `practiceCache` або делегує їх завантаження `PracticeUtil`; екран підписується на стан і відображає вкладки; позначення зіркою спричиняє виклик `MainViewModel.toggleStarStatus`, який через репозиторій оновлює таблицю `Room`, а реактивний `Flow` негайно оновлює перелік обраних у бічному меню. Важливо, що жодна дія в цьому сценарії не залучає передавання робіт чи оцінювання — усі комунікаційні та оцінювальні дії між здобувачем і викладачем відбуваються поза межами застосунку, що узгоджується з його роллю суто інформаційного інструмента.

Висновки до розділу 5

Визначено системні вимоги до програмного продукту з мінімальним рівнем API 24, що забезпечує широке охоплення пристроїв. Підтверджено повну автономність та приватність застосунку відсутністю дозволу на доступ до мережі у файлі маніфесту.

Описано модель розгортання застосунку через систему «єКафедра» шляхом встановлення APK-пакета, наведено покрокову інструкцію для здобувача та розглянуто оптимізацію розміру пакета інструментом R8 і модель версіонування контенту.

Спроектовано людино-машинний інтерфейс: формалізовано варіанти використання для акторів «Здобувач» і «Викладач», описано логіку основних екранів та наведено типовий наскрізний сценарій підготовки до практичної роботи.

6 ТЕСТУВАННЯ ТА ВАЛІДАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Тестування є невід’ємною складовою розробки якісного програмного забезпечення та водночас невід’ємним елементом навчального контенту НМК, що демонструє здобувачам правильний підхід до верифікації коду. У цьому розділі обґрунтовано стратегію тестування застосунку, описано модульне та інструментальне тестування і наведено фрагменти реальних тестів з кодової бази.

6.1 Стратегія тестування застосунку

Стратегію тестування побудовано за класичною моделлю піраміди тестування. В основі піраміди розташовано численні модульні (unit) тести, що виконуються швидко на віртуальній машині JVM без участі пристрою; середній рівень утворюють інтеграційні тести шару даних із застосуванням бази Room в режимі in-memo; вершину становлять менш численні, але важливі інструментальні UI-тести, що виконуються на емуляторі чи реальному пристрої. Така форма забезпечує оптимальне співвідношення між повнотою покриття та швидкістю виконання набору тестів [15, 19].

Для реалізації стратегії застосовано сучасний інструментарій автоматизованого тестування. Кількісні показники тестового покриття та використаний інструментарій узагальнено в таблиці 6.1.

Таблиця 6.1 – Тестове покриття застосунку

Рівень тестування	Файлів	Перевірок	Інструментарій
Модульні (JVM)	8	111	JUnit 4, MockK, Turbine
Інструментальні (емулятор)	9	49	Compose Testing API, Hilt

Форму піраміди тестування з кількісними показниками покриття унаочнено на рисунку 6.1.



Рисунок 6.1 – Піраміда тестування застосунку

Загальна кількість автоматизованих перевірок перевищує сто шістдесят. Переважання модульних тестів над інструментальними відповідає правильній формі піраміди. Інструментальні тести застосовують середовище Hilt для тестування (кастомний запускар `HiltTestRunner`), що забезпечує впровадження залежностей у тестовому контексті.

6.2 Модульне тестування бізнес-логіки

Модульні тести верифікують логіку компонентів в ізоляції від платформи Android. Залежності компонентів підмінюються імітаційними об'єктами (mock) бібліотекою MockK, а асинхронні потоки Flow перевіряються бібліотекою Turbine. Приклад тесту репозиторію наведено на рисунку 6.2 [3, 19].

```
@Test
fun `getAllStarredPractices returns flow from dao`() = runTest {
    val entities = listOf(StarredAssignmentEntity(1), StarredAssignmentEntity(2))
    every { dao.getAllStarredPractices() } returns flowOf(entities)

    repository.getAllStarredPractices().test { // Turbine
        val result = awaitItem()
        assertEquals(entities, result)
        cancelAndIgnoreRemainingEvents()
    }
}
```

Рисунок 6.2 – Модульний тест репозиторію з MockK та Turbine

Конструкція `every { dao.getAllStarredPractices() } returns flowOf(entities)` задає поведінку імітаційного об'єкта DAO: за виклику методу повертається заздалегідь визначений потік. Блок `repository.getAllStarredPractices().test { },` наданий `Turbine`, підписується на потік і дозволяє послідовно перевіряти збуджені значення оператором `awaitItem`. Обгортка `runTest` керує віртуальним часом корутин, забезпечуючи детермінованість тесту. Так перевірено, що репозиторій коректно делегує дані з DAO, не вносячи спотворень.

Окрім репозиторію, модульними тестами покрито компоненти `ViewModel` (зокрема `MainViewModelTest`, `ModulesViewModelTest`, `PracticeAssignmentViewModelTest`), моделі даних (`ModelsTest`), навігаційні маршрути (`ScreenTest`) та типи подій (`UiEventTest`). Окрему групу утворює інтеграційний тест об'єкта доступу до даних `StarredAssignmentDaoTest`, що працює з реальною базою `Room` у режимі `in-memory`; його фрагмент наведено на рисунку 6.3.

```
@Before fun setUp() {
    database = Room.inMemoryDatabaseBuilder(context, AppDatabase::class.java)
        .allowMainThreadQueries().build()
    dao = database.practicalAssignmentMinimalDao()
}

@Test fun starAssignment_duplicateIdReplacesExisting() = runTest {
    dao.starAssignment(StarredAssignmentEntity(5))
    dao.starAssignment(StarredAssignmentEntity(5)) // стратегія REPLACE
    val all = dao.getAllStarredPractices().first()
    assertEquals(1, all.size)
}
```

Рисунок 6.3 – Інтеграційний тест DAO з базою `Room` у режимі `in-memory`

База створюється методом `Room.inMemoryDatabaseBuilder`, що тримає дані в оперативній пам'яті та автоматично очищає їх після завершення тесту. Наведений тест перевіряє коректність стратегії конфлікту `OnConflictStrategy.REPLACE` (підрозділ 2.3): повторне додавання заняття з тим самим ідентифікатором не створює дублікату, а замінює наявний запис, тому підсумковий розмір вибірки дорівнює одиниці.

6.3 Інструментальне тестування інтерфейсу

Інструментальні тести верифікують поведінку інтерфейсу та виконуються на емуляторі засобами Compose Testing API. Тестове середовище конфігурується правилами JUnit: `HiltAndroidRule` забезпечує впровадження залежностей, а `createAndroidComposeRule<HiltTestActivity>()` надає керування деревом композиції. Тестований компонент розміщують у методі `setContent`, обгорнутим темою застосунку та провайдером локалізації. Приклад тесту перемикання вкладок наведено на рисунку 6.4 [15].

```
@Test fun clickingResourcesTab_selectsIt() {
    var clicked: PracticeAssignmentTabDestination? = null
    setContent(selected = Overview, onTabClick = { clicked = it })

    composeRule.onNodeWithText("Ресурси").performClick()
    assertEquals(PracticeAssignmentTabDestination.Resources, clicked)
}
```

Рисунок 6.4 – Інструментальний тест перемикання вкладок засобами Compose Testing API

Пошук елементів у дереві семантичних вузлів інтерфейсу виконується фінерами `onNodeWithText` та `onNodeWithContentDescription`; взаємодія імітується дією `performClick`, а перевірки стану — твердженнями `assertIsDisplayed` та `assertIsSelected`. У наведеному тесті симулюється дотик до вкладки «Ресурси» та перевіряється, що відповідний зворотний виклик отримав очікуване значення. За аналогічним принципом покрито інші компоненти: `TitledCard` (відображення заголовка, підзаголовка та тексту), `LanguageCard` (стан вибору мови), елементи списків модулів і занять, а також інтеграційний запуск активності у `ModulesActivityTest`. Для `stateful`-компонента `CollapsibleIndexedCard` перевіряється зміна видимості вмісту після дотику, що підтверджує коректність керування локальним станом розгортання.

Усі модульні та інструментальні тести проходять успішно, що підтверджує функційну коректність застосунку. Додатково проведено ручне функційне тестування навчального контенту: перевірено навігацію між екранами, коректність відображення всіх двадцяти семи занять обома мовами, роботу функції обраних занять та збереження вибору мови між сесіями.

Висновки до розділу 6

Обґрунтовано стратегію тестування застосунку за моделлю піраміди тестування з переважанням швидких модульних тестів над інструментальними; загальна кількість автоматизованих перевірок перевищує сто шістдесят.

Реалізовано модульне тестування бізнес-логіки із застосуванням бібліотек MockK та Turbine для ізоляції залежностей і перевірки реактивних потоків, а також інтеграційне тестування шару даних з базою Room у режимі in-memo, що підтвердило коректність стратегії конфлікту записів.

Реалізовано інструментальне тестування інтерфейсу засобами Compose Testing API з пошуком семантичних вузлів, симуляцією взаємодії та перевіркою станів компонентів. Сукупність автоматизованих та ручних тестів підтвердила функційну коректність і стабільність розробленого застосунку-НМК.

ВИСНОВКИ

У ході виконання бакалаврської дипломної роботи розроблено та впроваджено навчально-методичний комплекс з дисципліни «Розробка програмного забезпечення мобільних пристроїв» на Kotlin у форматі автономного мобільного застосунку. Поставлену мету досягнуто, а сукупність розв’язаних інженерних і методичних завдань дозволяє сформулювати такі висновки.

- За результатами техніко-економічного обґрунтування з-поміж трьох альтернативних архітектурних варіантів (нативного автономного, клієнт-серверного та кросплатформного) методом зваженого оцінювання обрано оптимальний — нативний автономний Android-застосунок, що отримав найвищий інтегральний показник за критеріями автономності, вартості володіння, безпеки локальних даних та швидкодії інтерфейсу.

- Проаналізовано предметну область: виявлено системні обмеження провідних освітніх платформ (мережева залежність, відсутність кастомізації під навчальний план кафедри, відсутність офлайн-прикладів) та обґрунтовано ключові технічні рішення — вибір декларативної парадигми Jetpack Compose, реляційного сховища Room і реактивного DataStore.

- Спроектовано методичну архітектуру курсу: ієрархію з дев’яти модулів, двадцяти семи занять та дванадцяти інформаційних блоків, формалізовану моделями даних. Алгоритмічно реалізовано поділ завдань на три рівні складності відповідно до таксономії Блума, а застосуванням єдиного переліку рівнів у моделях завдання та критерію оцінювання на структурному рівні забезпечено конструктивне вирівнювання навчання. Інженерно-енергетичну специфіку завдань підтверджено на прикладі зразкового рішення Лабораторної роботи №7.

- Реалізовано програмну архітектуру застосунку за патерном MVVM з елементами Clean Architecture: шар даних з десеріалізацією локальних JSON-ресурсів бібліотекою Gson, дворівневим кешуванням та сховищем Room; презентаційний шар з керуванням станом через StateFlow і SharedFlow, типобезпечною навігацією Navigation Compose та поділом компонентів на stateful і

stateless; кастомну підсистему динамічної локалізації без перезапуску активності; впровадження залежностей засобами Hilt.

- Визначено системні вимоги (мінімальний рівень API 24) та підтверджено повну автономність і приватність застосунку відсутністю мережових дозволів у маніфесті. Описано модель розгортання через систему «Кафедра» з оптимізацією розміру пакета інструментом R8 та спроектовано людино-машинний інтерфейс із типовими сценаріями використання.

- Проведено комплексне тестування за моделлю піраміди: реалізовано понад сто шістдесят автоматизованих перевірок (модульні тести з MockK і Turbine, інтеграційні тести Room та інструментальні UI-тести Compose Testing API), доповнених ручним функційним тестуванням. Усі тести пройдено успішно, що підтверджує функційну коректність і стабільність застосунку.

- Практичне значення роботи полягає у готовому до впровадження НМК, що може використовуватися при викладанні освітнього компоненту «Розробка програмного забезпечення мобільних пристроїв» у КПІ ім. Ігоря Сікорського. Перспективами подальшого розвитку є технічна інтеграція з системою «Кафедра» для автоматичного оновлення контенту, застосування контрактів стабільності @Immutable/@Stable для подальшої оптимізації швидкодії та розроблення банку тестових завдань для автоматизованої перевірки теоретичних знань.

Виконання роботи дозволило освоїти повний цикл промислової розробки мобільного застосунку — від техніко-економічного обґрунтування архітектури до тестування та підготовки до дистрибуції — та закріпити компетентності у проектуванні чистих архітектурних рішень, реактивному програмуванні, керуванні станом у декларативній парадигмі та автоматизованому тестуванні програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Biggs J., Tang C. Teaching for Quality Learning at University. 4th ed. Open University Press, 2011. 389 p.
2. Bloom B. S. Taxonomy of Educational Objectives. Handbook I: Cognitive Domain. New York: McKay, 1956. 207 p.
3. Cash App. Turbine — testing library for kotlinx.coroutines Flow [Електронний ресурс]. URL: <https://github.com/cashapp/turbine> (дата звернення: 05.05.2026).
4. Coursera. Android App Development Specialization [Електронний ресурс]. URL: <https://www.coursera.org/specializations/android-app-development> (дата звернення: 10.04.2026).
5. Google LLC. Android Basics with Compose Course [Електронний ресурс]. URL: <https://developer.android.com/courses/android-basics-compose/course> (дата звернення: 10.04.2026).
6. Google LLC. Android Developers Documentation [Електронний ресурс]. URL: <https://developer.android.com/docs> (дата звернення: 01.05.2026).
7. Google LLC. Compose Recomposition and Stability [Електронний ресурс]. URL: <https://developer.android.com/develop/ui/compose/performance/stability> (дата звернення: 02.05.2026).
8. Google LLC. DataStore [Електронний ресурс]. URL: <https://developer.android.com/topic/libraries/architecture/datastore> (дата звернення: 04.05.2026).
9. Google LLC. Guide to App Architecture [Електронний ресурс]. URL: <https://developer.android.com/topic/architecture> (дата звернення: 02.05.2026).
10. Google LLC. Hilt for Android [Електронний ресурс]. URL: <https://developer.android.com/training/dependency-injection/hilt-android> (дата звернення: 03.05.2026).
11. Google LLC. Jetpack Compose [Електронний ресурс]. URL: <https://developer.android.com/compose> (дата звернення: 01.05.2026).

12. Google LLC. Navigation Compose [Електронний ресурс]. URL: <https://developer.android.com/develop/ui/compose/navigation> (дата звернення: 04.05.2026).
13. Google LLC. R8 code shrinker [Електронний ресурс]. URL: <https://developer.android.com/build/shrink-code> (дата звернення: 06.05.2026).
14. Google LLC. Room Persistence Library [Електронний ресурс]. URL: <https://developer.android.com/training/data-storage/room> (дата звернення: 02.05.2026).
15. Google LLC. Testing in Jetpack Compose [Електронний ресурс]. URL: <https://developer.android.com/develop/ui/compose/testing> (дата звернення: 04.05.2026).
16. JetBrains. Kotlin Coroutines Guide [Електронний ресурс]. URL: <https://kotlinlang.org/docs/coroutines-guide.html> (дата звернення: 03.05.2026).
17. JetBrains. Kotlin Programming Language [Електронний ресурс]. URL: <https://kotlinlang.org/> (дата звернення: 01.05.2026).
18. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017. 432 p.
19. MockK. Mocking library for Kotlin [Електронний ресурс]. URL: <https://mockk.io/> (дата звернення: 05.05.2026).
20. Udacity. Android Kotlin Developer Nanodegree [Електронний ресурс]. URL: <https://www.udacity.com/course/android-kotlin-developer-nanodegree--nd940> (дата звернення: 10.04.2026).
21. ДСТУ 3008:2015. Інформація та документація. Звіти у сфері науки і техніки. Структура та правила оформлювання. Київ: ДП «УкрНДНЦ», 2016. 26 с.
22. Розробка програмного забезпечення мобільних пристроїв [Електронний ресурс] : навч. посіб. для здобувачів ступеня бакалавра за освіт. програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці» спец. 121 інженерія програмного забезпечення / КПІ ім. Ігоря Сікорського ; уклад.: О. Л. Недашківський, І. І. Гусева, В. О. Голець, М. С. Гнатишин, Т. В. Пироговська, В. Р. Передера. – Електрон. текст. дані (1 файл: 3,6

Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2025. – 316 с. – Назва з екрана. URL:
<https://ela.kpi.ua/handle/123456789/74698> (дата звернення: 01.05.2026).

ДОДАТОК А

Лістинг програмного коду застосунку

У цьому додатку наведено повний лістинг ключових класів програмної реалізації застосунку-НМК: моделей даних, утиліт парсингу, об'єкта доступу до даних, репозиторію, компонентів ViewModel та підсистеми локалізації.

`@AndroidEntryPoint`

```
class ModulesActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            KotlinBasicsAppTheme {
                MainView()
            }
        }
    }
}
```

`@OptIn(ExperimentalMaterial3Api::class)`

`@Composable`

```
fun MainView() {
    val drawerState = rememberDrawerState(initialValue = DrawerValue.Closed)
    val mainViewModel: MainViewModel = hiltViewModel()
    val navController = rememberNavController()
    val scope = rememberCoroutineScope()
    LaunchedEffect(Unit) {
        mainViewModel.uiEvent.collect { event ->
            when (event) {
                is UiEvent.Back -> navController.navigateUp()
                is UiEvent.Navigate -> navController.navigate(event.route)
            }
        }
    }
}
```

```

    }
}
}
ProvideLocalizedContext(mainViewModel = mainViewModel) {

    ModalNavigationDrawer(
        drawerState = drawerState,
        drawerContent = {
            ModalDrawerSheet(
                modifier = Modifier.fillMaxWidth(1f),
                drawerContainerColor = Color.White,
                drawerShape = RoundedCornerShape(0.dp),
            ) {
                DrawerContent(
                    mainViewModel = mainViewModel,
                    onDismiss = {
                        scope.launch {
                            drawerState.close()
                        }
                    }
                )
            }
        }
    )

    Scaffold(
        topBar = {
            AppBarHeader(mainViewModel, onHamburgerClick = {
                scope.launch {
                    drawerState.open()
                }
            })
        }
    )
}

```

```

        })
    },
    content = {
        innerPadding ->
        Navigation(innerPadding, mainViewModel, navController)
    }
)
}
}

@Composable
fun ProvideLocalizedContext(
    mainViewModel: MainViewModel = hiltViewModel(),
    content: @Composable () -> Unit
) {
    val language by mainViewModel.languageCode.collectAsState()
    val baseContext = LocalContext.current

    val localizedContext = remember(language, baseContext) {
        baseContext.updateLocale(language)
    }

    CompositionLocalProvider(LocalLocalizedContext provides localizedContext) {
        content()
    }
}

val LocalLocalizedContext = staticCompositionLocalOf<Context> {
    error("Localized Context not provided. Wrap your composable tree with ProvideLocalizedContext.")
}

```

```

enum class Language {
    English,
    Ukrainian,
}

@Composable
fun DrawerContent(
    onDismiss: () -> Unit,
    mainViewModel: MainViewModel
) {
    val lang by mainViewModel.languageCode.collectAsState()
    val availableLanguages = listOf(
        Pair(LocalizedString(R.string.ukr_lang), Language.Ukrainian),
        Pair(LocalizedString(R.string.english_lang), Language.English),
    )
    val favouritePractices by
mainViewModel.getAllStarredPractices().collectAsState(initial = emptyList())
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(horizontal = 16.dp, vertical = 20.dp)
            .background(color = Color.White),
        horizontalArrangement = Arrangement.SpaceBetween,
        verticalAlignment = Alignment.CenterVertically
    ) {
        IconButton(onClick = {
            onDismiss()
        })
    }
}

```

```

        Icon(
            painter = painterResource(id = R.drawable.x),
            contentDescription = "Back",
            modifier = Modifier.size(24.dp),
            tint = Color.Black
        )
    }

```

```

    Text(
        text = LocalizedString(R.string.app_name),
        color = Color.Black,
        style = MaterialTheme.typography.titleMedium.copy(
            fontSize = 24.sp,
            fontWeight = FontWeight.Normal,
        ),
    )

```

```

        Box(modifier = Modifier.size(24.dp))
    }

```

```
HorizontalDivider()
```

```

LazyColumn(
    modifier = Modifier.background(color = Color.White),
    contentPadding = PaddingValues(all = 16.dp),
    verticalArrangement = Arrangement.spacedBy(24.dp)
) {
    item {
        TitledSectionWithIcon(
            titleRes = R.string.lang,
            iconRes = R.drawable.translator,

```

```

content = {
    Column(
        verticalArrangement = Arrangement.spacedBy(8.dp),
    )
    availableLanguages.forEach { language ->
        LanguageCard(
            iconRes = when (language.second) {
                Language.English -> R.drawable.us_united_states
                Language.Ukrainian -> R.drawable.ua_ukraine
            },
            title = language.first,
            isSelected = when (language.second) {
                Language.English -> lang == "en"
                Language.Ukrainian -> lang == "uk"
            },
            onClick = {
                when (language.second) {
                    Language.English -> mainViewModel.setLanguage("en")
                    Language.Ukrainian -> mainViewModel.setLanguage("uk")
                }
            }
        )
    }
}

}

)

}

item {

```

```

TitledSectionWithIcon(
    titleRes = R.string.favourite,
    iconRes = R.drawable.filled_star,
    content = {
        Column(
            verticalArrangement = Arrangement.spacedBy(8.dp),
        ) {
            favouritePractices.forEach { (id, theme) ->
                PractiseAssignmentListItem(
                    practice = PracticalListData(id,theme),
                    onClick = {
                        onDismiss()
                        mainViewModel.emitEvent(UiEvent.Navigate(
                            Screen.PracticeAssignmentScreen.createRoute(id)
                        ))
                    }
                )
            }
        }
    }
)
}

@Composable
fun AppBarHeader(
    mainViewModel: MainViewModel,
    onHamburgerClick: () -> Unit,

```

```

)
val title by mainViewModel.title.collectAsState()
val isStarred by mainViewModel.isStarred.collectAsState()
val currentPracticeId by mainViewModel.currentPracticeId.collectAsState()
Row(
    modifier = Modifier
        .fillMaxWidth()
        .background(Color.Black)
        .statusBarsPadding()
        .padding(start = 16.dp, end = 16.dp, bottom = 20.dp),
    verticalAlignment = Alignment.CenterVertically
)
IconButton(onClick = {
    if (currentPracticeId != null) {
        mainViewModel.navigateBack()
        mainViewModel.setTitle(null)
    } else {
        onHamburgerClick()
    }
})
if (title == null) {
    Icon(
        painter = painterResource(id = R.drawable.hamburger),
        contentDescription = "Open drawer",
        modifier = Modifier.size(24.dp),
        tint = MaterialTheme.colorScheme.primary
    )
} else {
    Icon(
        painter = painterResource(id = R.drawable.arrow_back),

```

```

        contentDescription = "Back",
        modifier = Modifier.size(24.dp),
        tint = MaterialTheme.colorScheme.primary
    )
}
}

Text(
    modifier = Modifier
        .weight(1f)
        .padding(horizontal = 8.dp),
    text = title ?: LocalizedString(R.string.app_name),
    color = Color.White,
    style = MaterialTheme.typography.titleMedium.copy(
        fontSize = if (title == null) 24.sp else 16.sp,
        fontWeight = if (title == null) FontWeight.Bold else FontWeight.Normal
    ),
    maxLines = 2,
    overflow = TextOverflow.Ellipsis,
    textAlign = if (title == null) TextAlign.Center else TextAlign.Left
)

if (title != null) {
    IconButton(onClick = { mainViewModel.toggleStarStatus() }) {
        val starIcon = if (isStarred) R.drawable.filled_star else R.drawable.star
        Icon(
            painter = painterResource(id = starIcon),
            contentDescription = if (isStarred) "Remove from favorites" else "Add to
favorites",
            modifier = Modifier.size(24.dp),

```

```

        tint = MaterialTheme.colorScheme.primary
    )
}
} else {
    Box(modifier = Modifier.size(24.dp))
}
}
}
}

```

```

@Composable
fun Navigation(innerPadding: PaddingValues,
    mainViewModel: MainViewModel,
    navController: NavHostController) {
    NavHost(
        navController = navController,
        startDestination = Screen.ModulesScreen.route,
        modifier = Modifier.padding(innerPadding),
        enterTransition = {
            slideInHorizontally(
                initialOffsetX = { it },
                animationSpec = tween(300)
            )
        },
        exitTransition = {
            slideOutHorizontally(
                targetOffsetX = { -it },
                animationSpec = tween(300)
            )
        }
    )
}

```

```

        )
    },
    popEnterTransition = {
        slideInHorizontally(
            initialOffsetX = { -it },
            animationSpec = tween(300)
        )
    },
    popExitTransition = {
        slideOutHorizontally(
            targetOffsetX = { it },
            animationSpec = tween(300)
        )
    }
)

composable(Screen.ModulesScreen.route) {
    ModuleList(mainViewModel = mainViewModel)
}

composable(
    route = Screen.PracticeAssignmentScreen.route,
    arguments = listOf(navArgument("practiceId") { type = NavType.IntType }),

) {
    backStackEntry ->
    val practiceId = backStackEntry.arguments?.getInt("practiceId") ?: -1
    PracticeAssignmentContainer(practiceId = practiceId, mainViewModel =
mainViewModel)
}

```

```

    }
}

```

```

        @OptIn(ExperimentalMaterial3Api::class)
@Composable
fun PracticeAssignmentOverviewScreen(
    practice: PracticalAssignmentData,
    resetTrigger: Any? = null
) {
    val listState = rememberLazyListState()

    LaunchedEffect(resetTrigger) {
        listState.scrollToItem(0)
    }

    LazyColumn(
        state = listState,
        modifier = Modifier.fillMaxSize(),
        contentPadding = PaddingValues(16.dp),
        verticalArrangement = Arrangement.spacedBy(16.dp),
    ) {
        item {
            TitledCard(
                title = LocalizedString(R.string.theme),
                text = practice.theme,
            )
        }
        item {
            TitledCard(
                title = LocalizedString(R.string.goal),

```

```

        text = practice.goal,
    )
}
item {
    TitledCard(
        title = LocalizedString(R.string.intro),
        text = practice.intro,
    )
}
item {
    TitledCard(
        title = LocalizedString(R.string.theory),
        text = practice.theory,
    )
}
item {
    TitledSection(
        titleRes = R.string.review_questions,
        content = {
            IndexedList(
                items = practice.reviewQuestions.map { IndexedListItem(it.question,
it.answer) },
                modifier = Modifier.fillMaxWidth(),
                previewCount = 5,
            )
        }
    )
}
}
}

```

```

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun PracticeAssignmentPracticeScreen(
    practice: PracticalAssignmentData,
    resetTrigger: Any? = null
) {

    val listState = rememberLazyListState()

    LaunchedEffect(resetTrigger) {
        listState.scrollToItem(0)
    }

    LazyColumn(
        state = listState,
        modifier = Modifier.fillMaxSize(),
        contentPadding = PaddingValues(16.dp),
        verticalArrangement = Arrangement.spacedBy(16.dp),
    ) {

        item {
            TitledSection(
                titleRes = R.string.individual_assignments,
                content = {
                    IndexedList(
                        items = practice.individualAssignments.map { IndexedListItem(it, "") },
                        modifier = Modifier.fillMaxWidth(),
                    )
                }
            )
        }
    }
}

```

```

        previewCount = 5,
        canItemsExpand = false
    )
}
)
}

item {
    TitledSection(
        titleRes = R.string.additional_assignments,
        content = {
            AdditionalAssignmentList(
                additionalAssignments = practice.additionalAssignments,
                reviewCriteria = practice.reviewCriteria,
            )
        }
    )
}

```

```

item {
    TitledSection(
        titleRes = R.string.home_work,
        content = {
            TitledCard(
                text = practice.homeWork,
            )
        }
    )
}

```

```

    item {
        TitledSection(
            titleRes = R.string.explanation_example_1,
            content = {
                AssignmentLevelsList(practice.assignmentFullCode, practiceId =
practice.id)
            }
        )
    }
}

```

```

    item {
        TitledSection(
            titleRes = R.string.code_description_and_structure,
            content = {
                CodeSummaryAndStructureList(
                    practice.assignmentFullCode.summary
                )
            }
        )
    }
}
}
}

```

```

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun PracticeAssignmentResourcesScreen(
    practice: PracticalAssignmentData,

```

```

        resetTrigger: Any? = null
    ) {

        val listState = rememberLazyListState()

        LaunchedEffect(resetTrigger) {
            listState.scrollToItem(0)
        }

        LazyColumn(
            state = listState,
            modifier = Modifier.fillMaxSize(),
            contentPadding = PaddingValues(16.dp),
            verticalArrangement = Arrangement.spacedBy(16.dp),
        ) {

            item {
                TitledSection(
                    titleRes = R.string.resources_used,
                    content = {
                        IndexedList(
                            items = practice.resourcesUsedData.map { IndexedListItem(it.title,
it.subTitle) },
                            modifier = Modifier.fillMaxWidth(),
                        )
                    }
                )
            }

            item {

```

```

        TitledSection(
            titleRes = R.string.additional_resources,
            content = {
                IndexedList(
                    items = practice.additionalResourcesData.map { IndexedListItem(it.title,
it.subTitle) },
                    modifier = Modifier.fillMaxWidth(),
                )
            }
        )
    }
}

sealed class Screen(val route: String) {
    object ModulesScreen : Screen("modules")

    object PracticeAssignmentScreen : Screen("practice/{practiceId}") {
        fun createRoute(practiceId: Int) = "practice/$practiceId"
    }

    object PracticeAssignmentOverviewScreen :
        Screen("overview")
    object PracticeAssignmentPracticeScreen :
        Screen("practice_tab")
    object PracticeAssignmentResourcesScreen :
        Screen("resources")
}

enum class PracticeAssignmentTabDestination {

```

Overview, Practice, Resources

```
}  
private val Context.dataStore: DataStore<Preferences> by preferencesDataStore(name  
= "settings")  
  
class LanguagePreferenceManager @Inject constructor(  
    @ApplicationContext private val context: Context  
) {  
  
    private object PreferencesKeys {  
        val LANGUAGE_CODE = stringPreferencesKey("language_code")  
    }  
  
    suspend fun saveLanguage(code: String) {  
        context.dataStore.edit { preferences ->  
            preferences[PreferencesKeys.LANGUAGE_CODE] = code  
        }  
    }  
  
    val languageCode: Flow<String> = context.dataStore.data  
        .catch { exception ->  
            if (exception is IOException) {  
                emit(emptyPreferences())  
            } else {  
                throw exception  
            }  
        }  
        .map { preferences ->  
            preferences[PreferencesKeys.LANGUAGE_CODE] ?: "uk"
```

```

    }
}
@Composable
fun LocalizedString(
    @StringRes resId: Int,
    vararg args: Any,
    mainViewModel: MainViewModel = hiltViewModel(),
    localizedContext: Context = LocalLocalizedContext.current
): String {
    val languageCode by mainViewModel.languageCode.collectAsState()

    return remember(resId, languageCode, localizedContext, *args) {
        localizedContext.getString(resId, *args)
    }
}

fun Context.updateLocale(languageCode: String): Context {
    val locale = Locale(languageCode)
    Locale.setDefault(locale)

    val config = resources.configuration
    config.setLocale(locale)
    config.setLayoutDirection(locale)
    return createConfigurationContext(config)
}

```

ДОДАТОК Б

Скріншоти інтерфейсу застосунку

У цьому додатку наведено скріншоти основних екранів застосунку: головного екрана модулів, екрана практичного заняття з трьома вкладками, бічного навігаційного меню з перемиканням мови та переліком обраних занять.

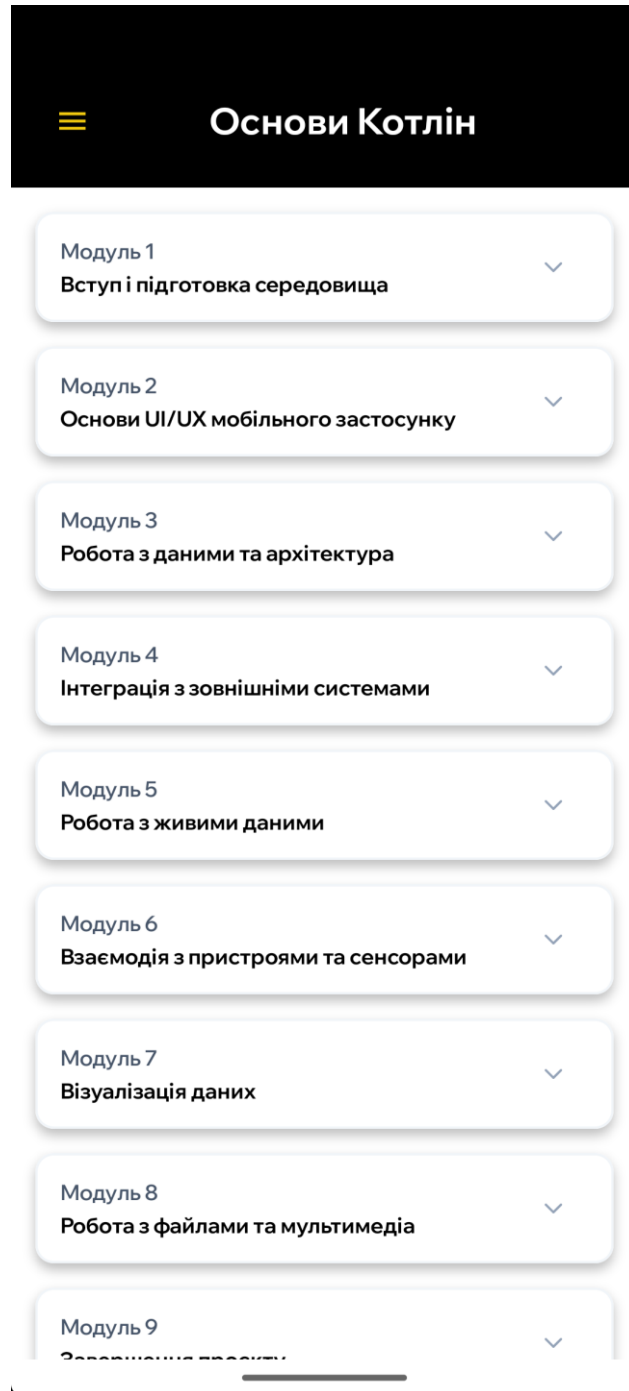


Рисунок Б.1 – Головний екран

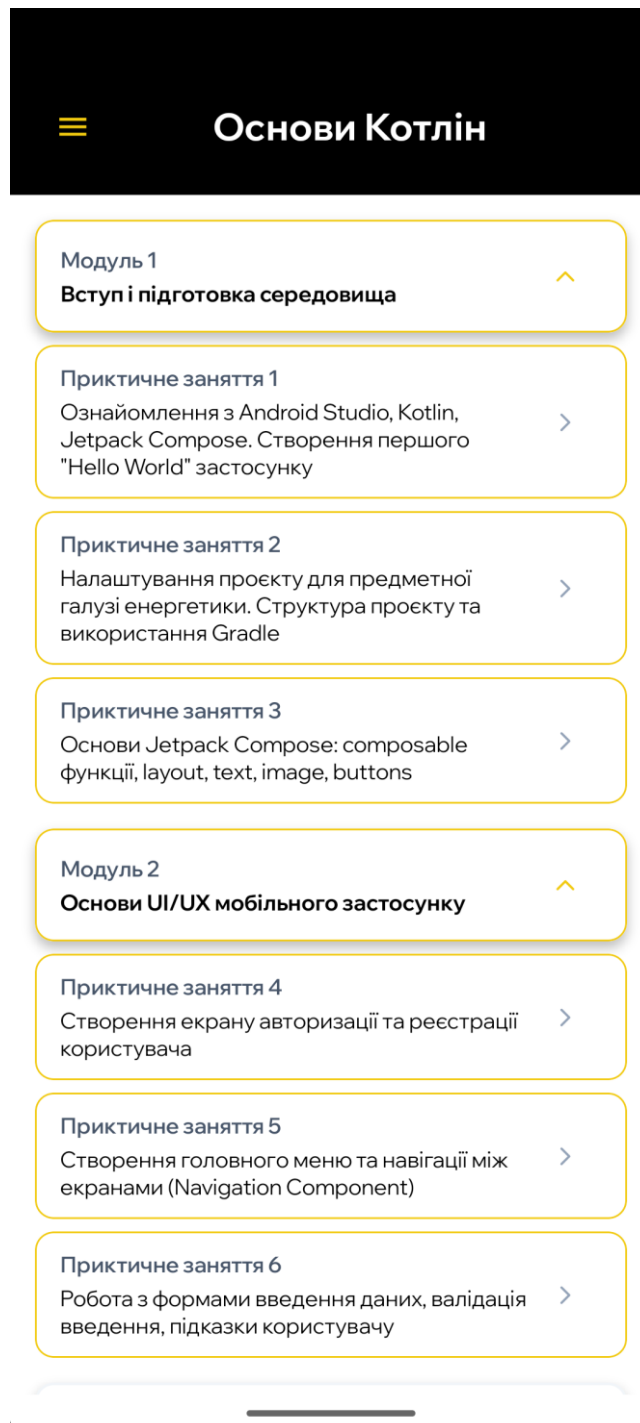


Рисунок Б.2 – Головний екран з випадаюч списком для модулів

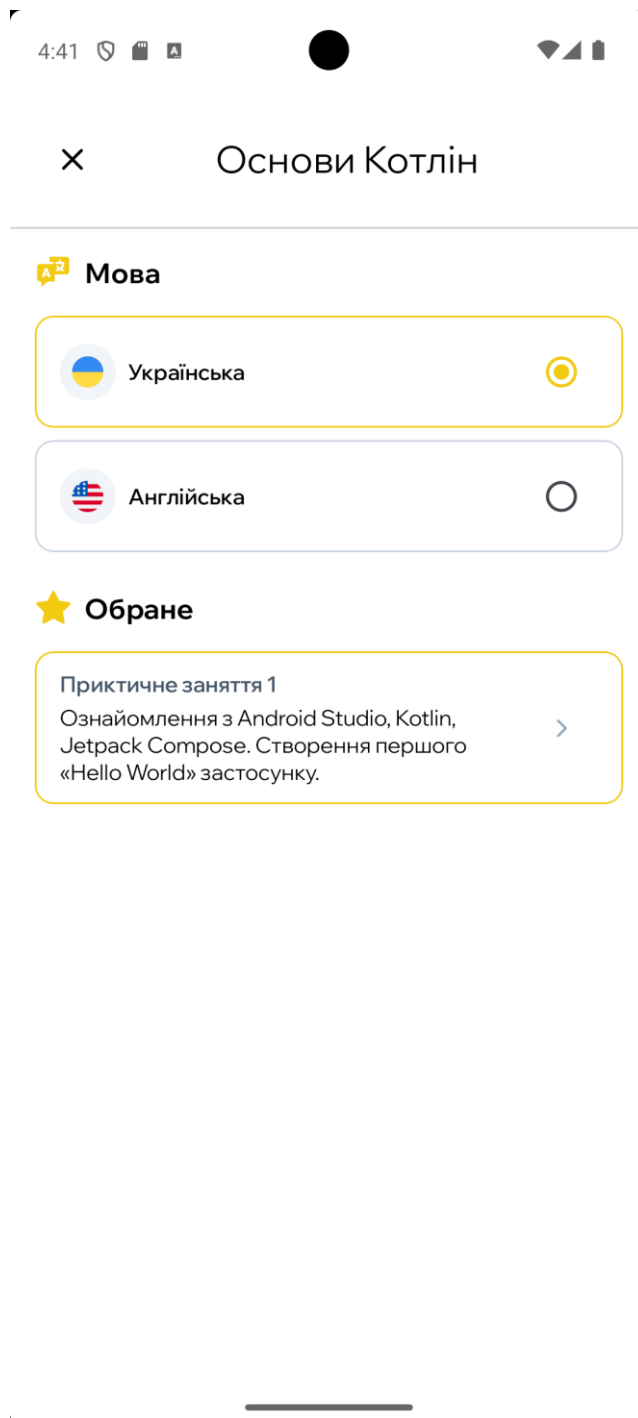


Рисунок Б.3 – Екран з вибором мови і списком обраних практичних

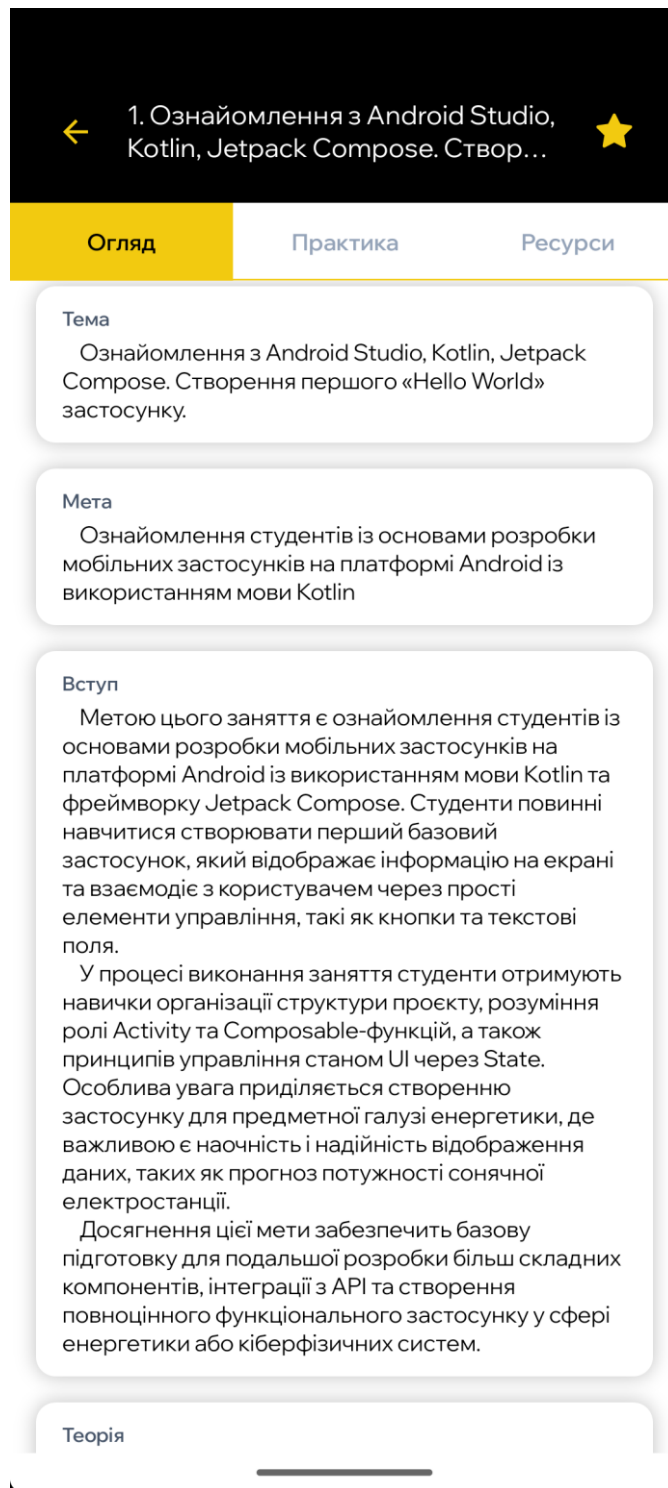


Рисунок Б.4 – Екран практичної роботи вкладка «Огляд»

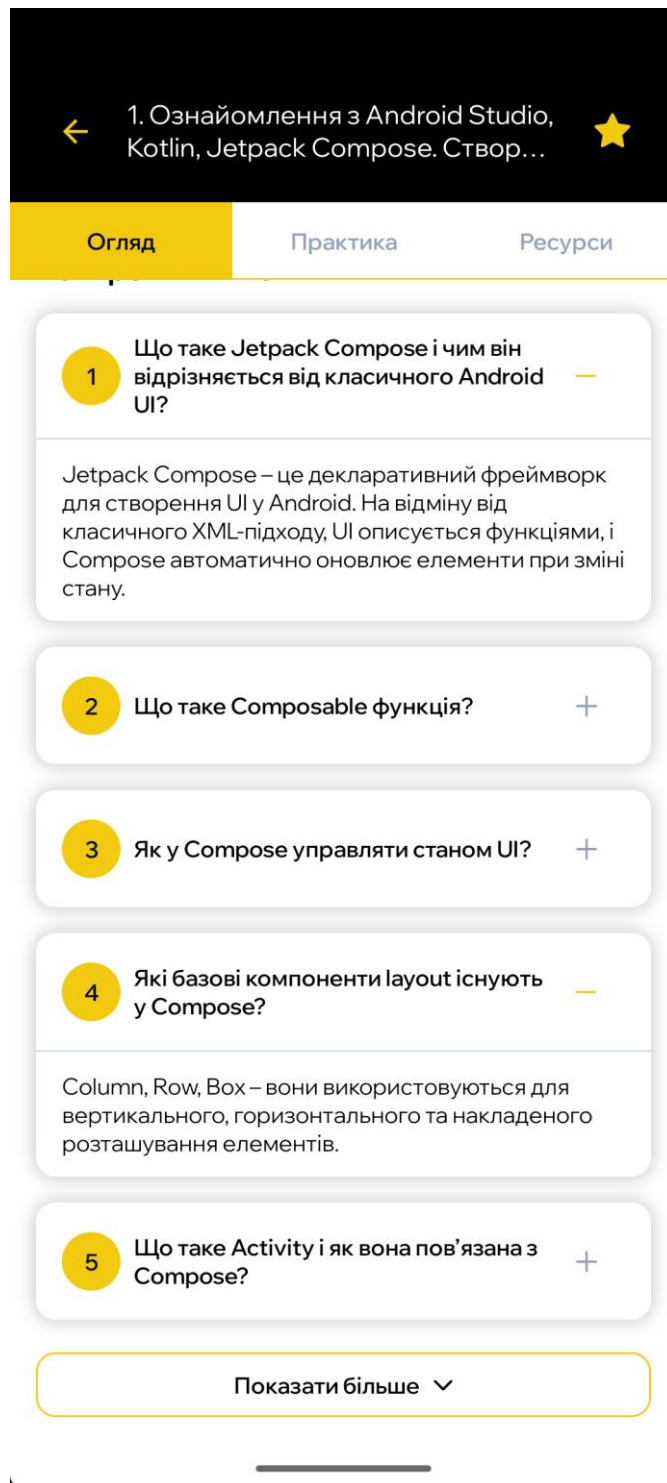


Рисунок Б.5 – Екран практичної роботи вкладка «Огляд»

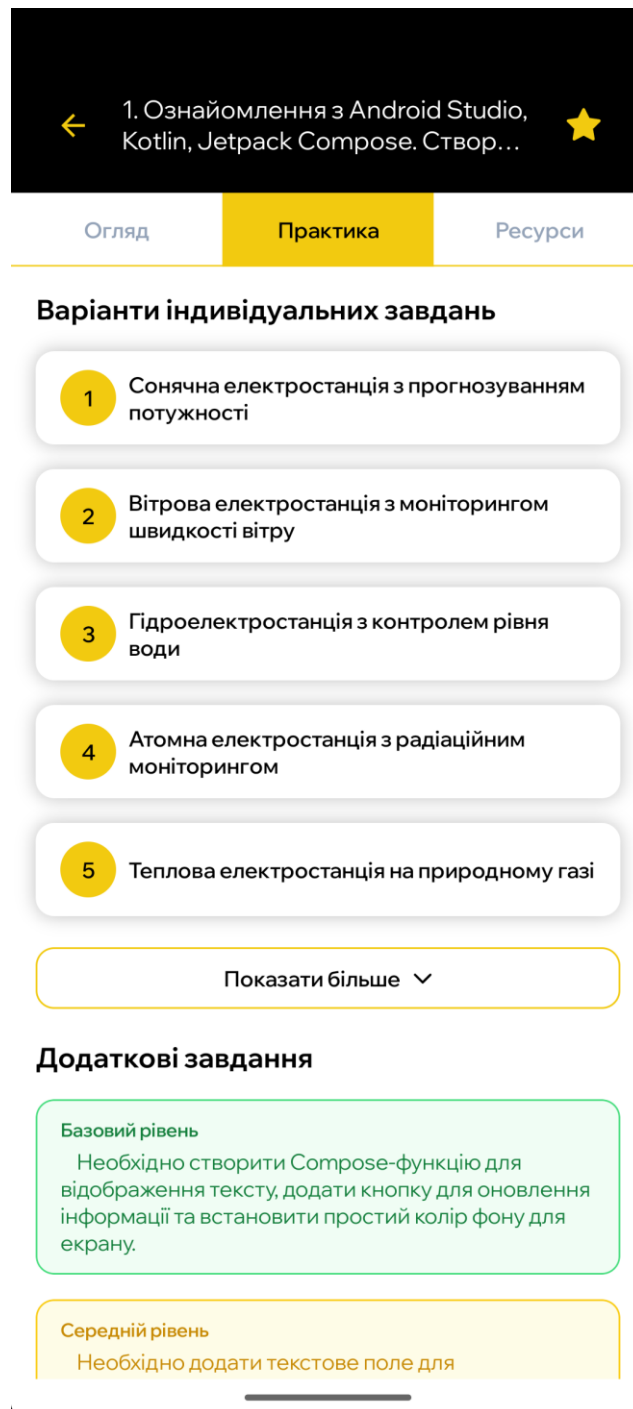


Рисунок Б.6 – Екран практичної роботи вкладка «Практика»



Рисунок Б.7 – Екран практичної роботи вкладки «Практика».

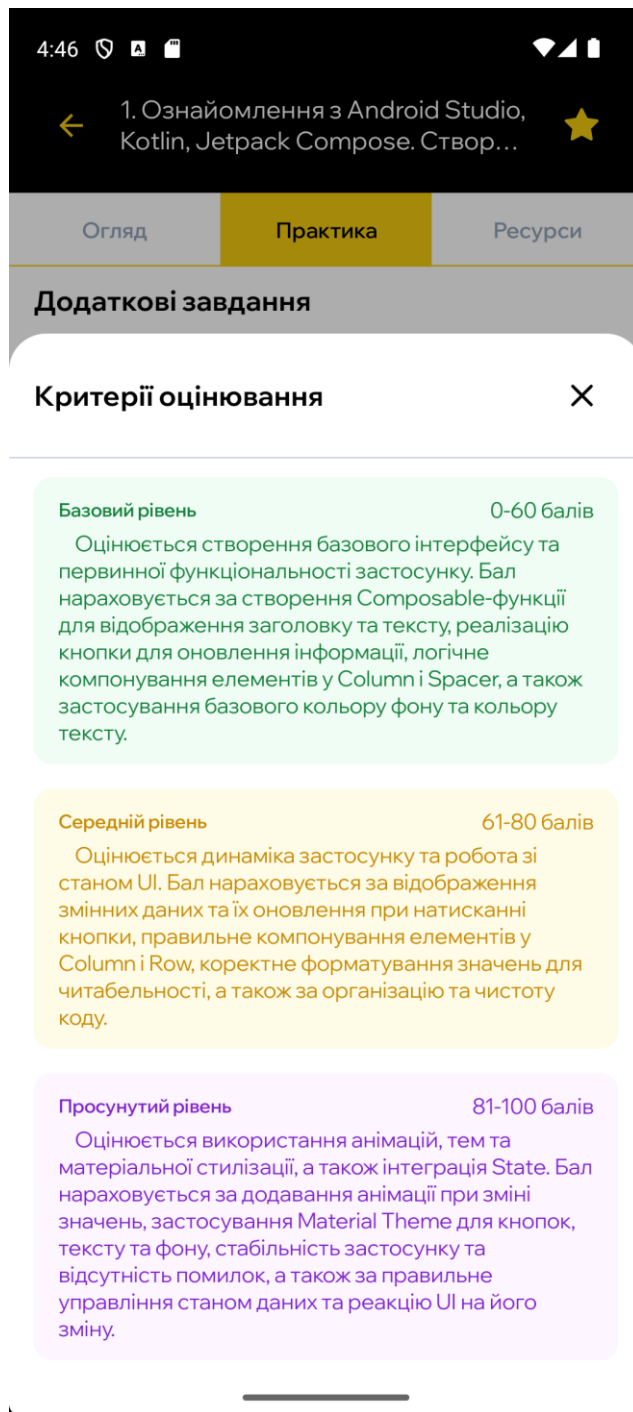


Рисунок Б.8 – Впливаюче меню з критеріями оцінювання

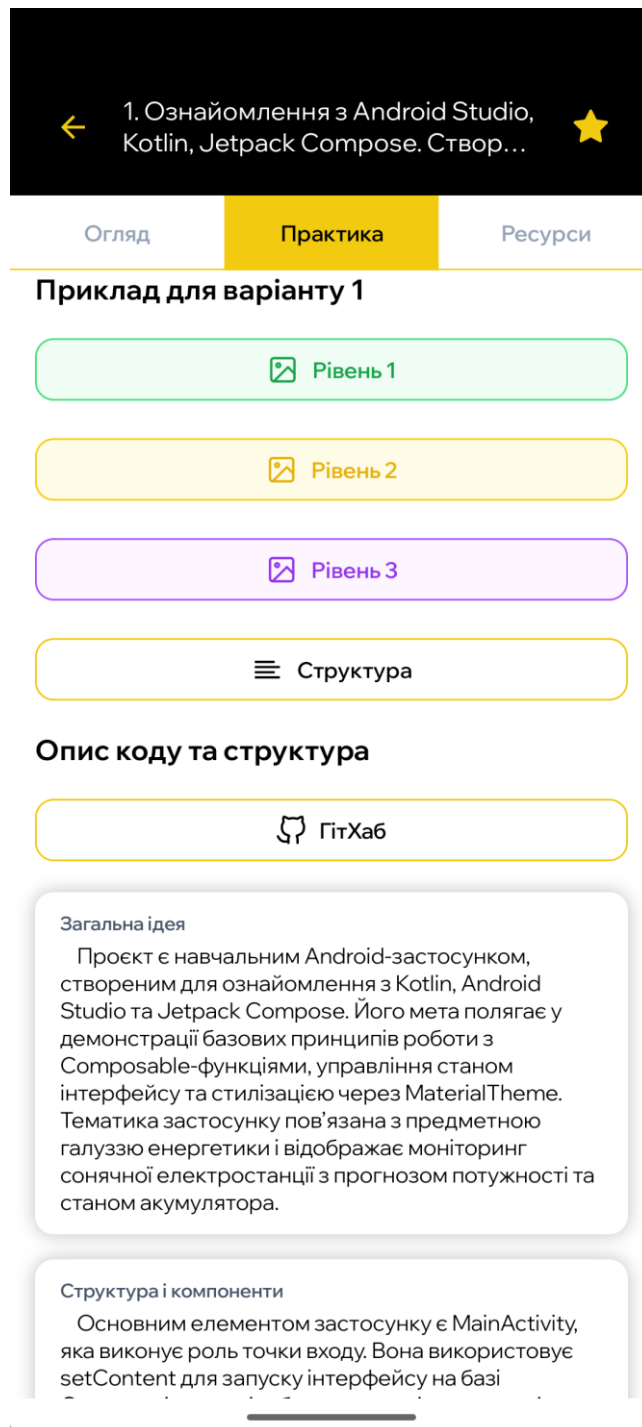


Рисунок Б.9 – Екран практичної роботи вкладка «Практика».

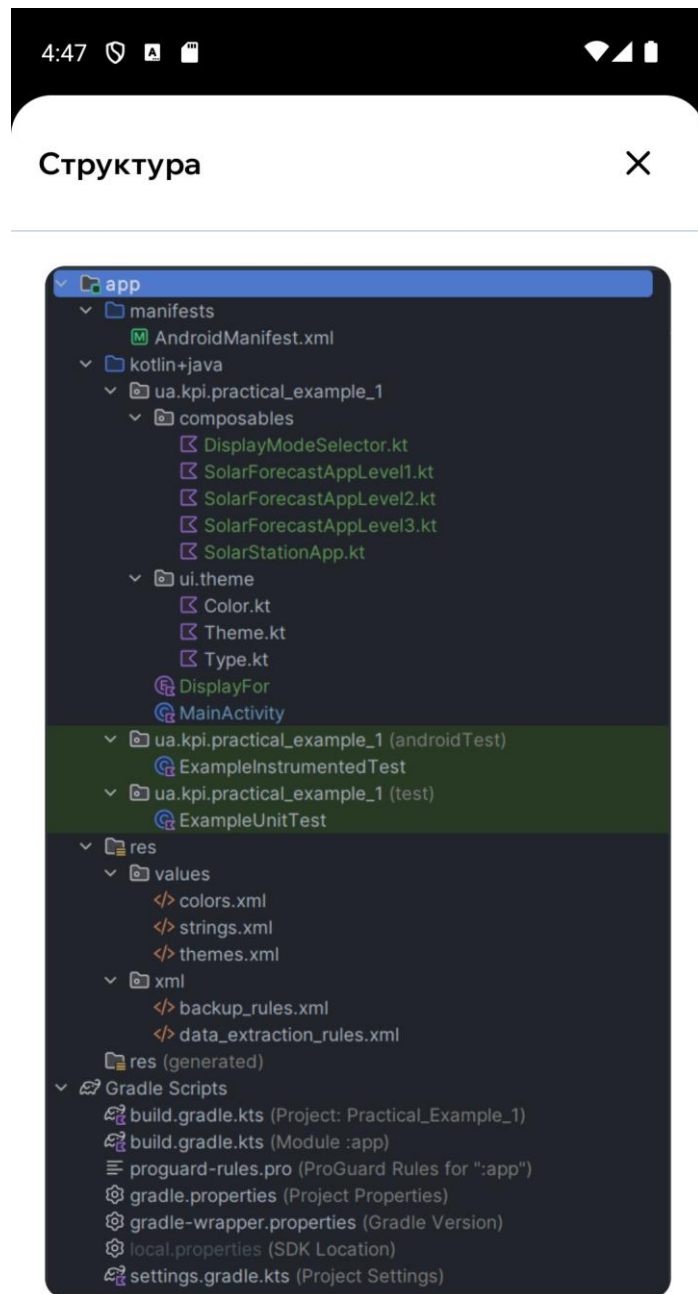


Рисунок Б.10 – Впливаюче меню зі структурою практичної роботи.

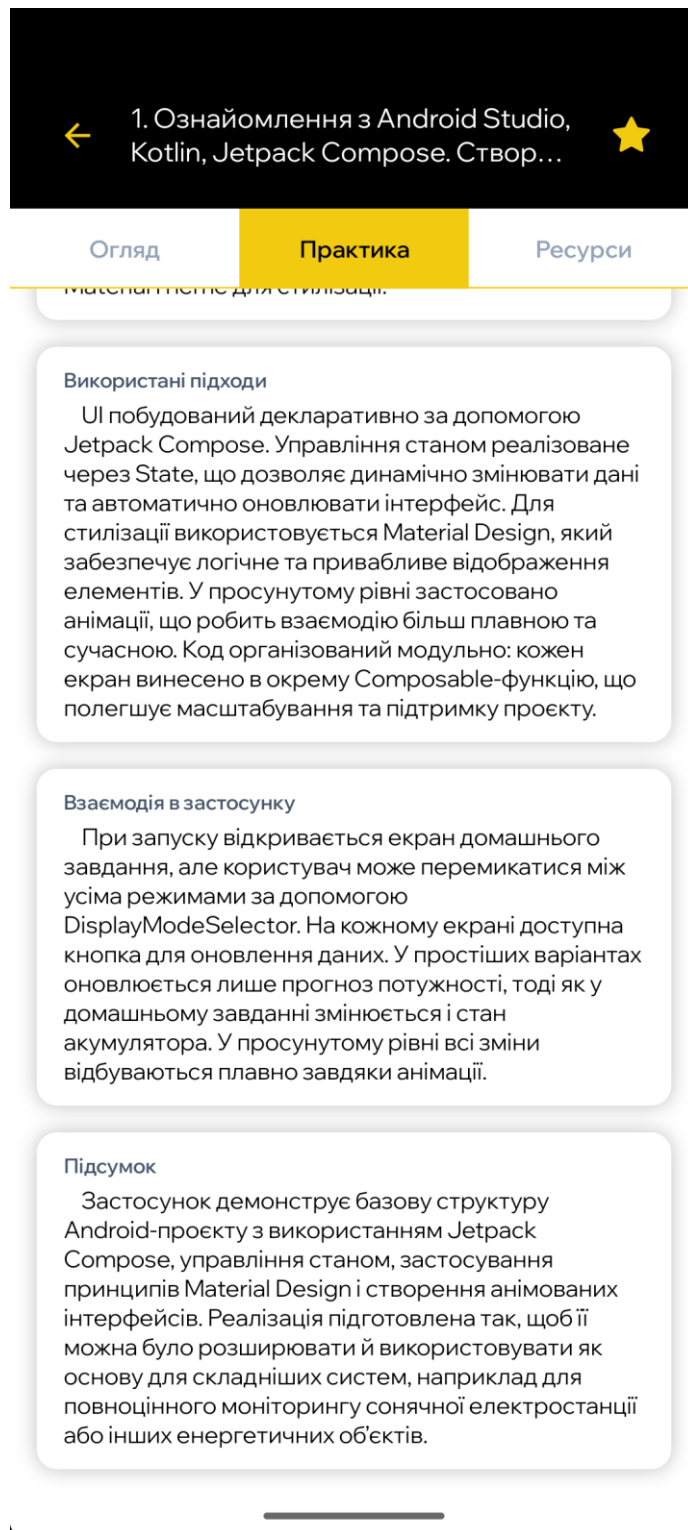


Рисунок Б.11 – Екран практичної роботи вкладки «Практика».

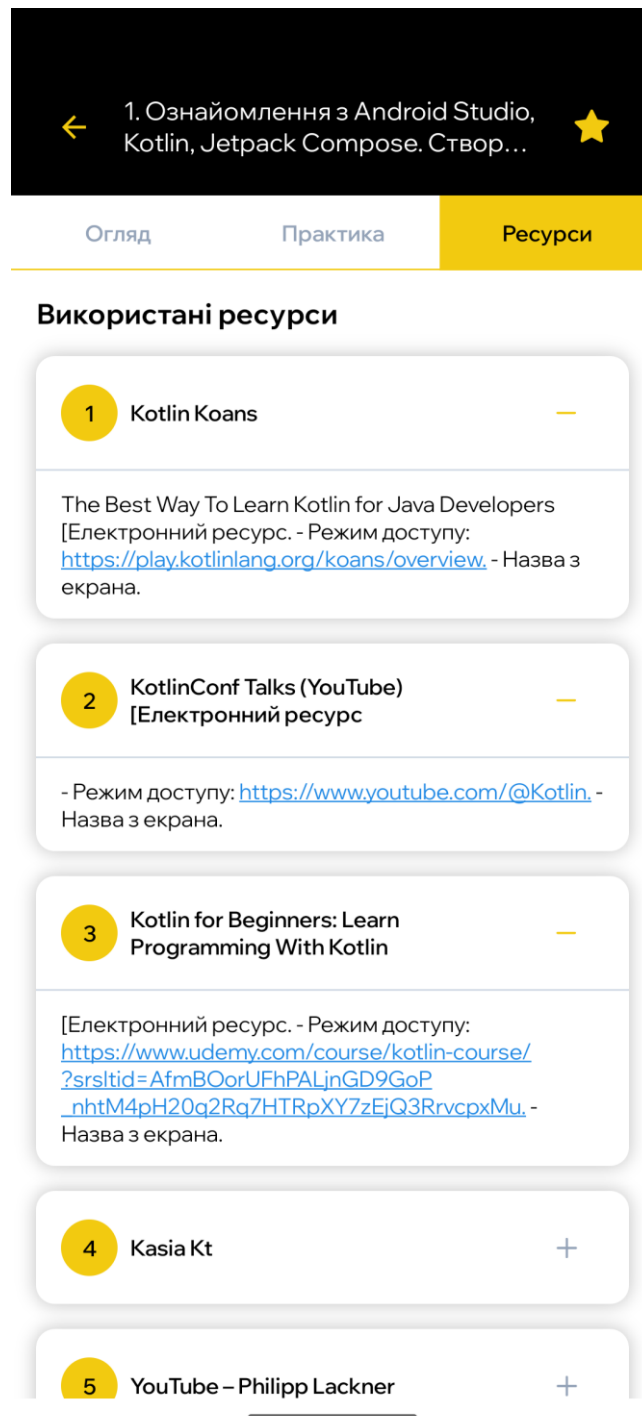


Рисунок Б.12 – Екран практичної роботи вкладка «Ресурси».

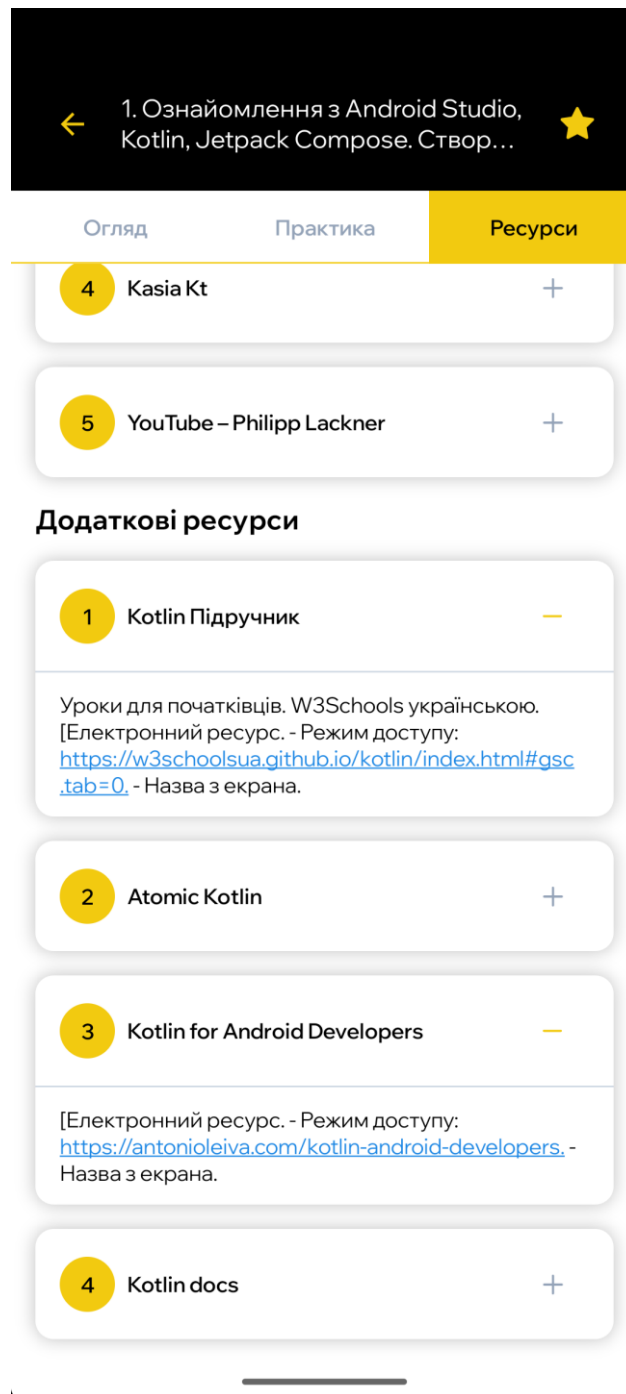


Рисунок Б.13 – Екран практичної роботи вкладка «Ресурси».

ДОДАТОК В

Презентація

У цьому додатку наведено матеріали презентації дипломної роботи для захисту перед екзаменаційною комісією.



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці

**НАВЧАЛЬНО-МЕТОДИЧНИЙ КОМПЛЕКС З ДИСЦИПЛІНИ
“РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ МОБІЛЬНИХ
ПРИСТРОЇВ” НА KOTLIN**

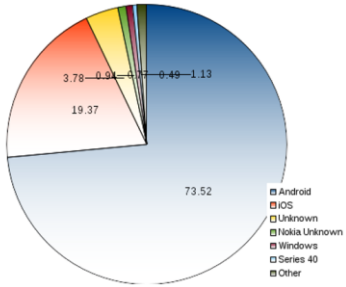
виконав: студент групи ТВ-23 Черниш Михайло Андрійович
керівник: Недашківський Олексій Леонідович

Київ — 2026

Рисунок В.1 - Слайд 1

Актуальність теми

Станом на 2026 рік понад 85% Android-застосунків розробляються на Kotlin (IDC)
Android займає близько 73% світового ринку мобільних ОС (Statista, 2025)
Наявні курси (Coursera, Udacity) не адаптовані під університетський формат з критеріями оцінювання та банком завдань
Мета: розробити сучасний НМК у форматі Android-застосунку для підготовки фахівців з мобільної розробки на Kotlin



ОС	Відсоток
Android	73.52
iOS	19.37
Unknown	3.78
Nokia Unknown	0.94
Windows	0.49
Series 40	0.49
Other	1.13



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

2

Рисунок В.2 - Слайд 2

Постановка задачі

Розробити методичні матеріали для 27 практичних занять (9 тематичних модулів)

Забезпечити кожне заняття: теорією, 30 варіантами завдань (3 рівні складності), критеріями оцінювання, зразковим кодом на GitHub

Реалізувати зразкові Android-застосунки (Варіант №1) для всіх 27 занять: Kotlin, Jetpack Compose, MVVM, Room, Hilt

Розробити мобільний застосунок-НМК KotlinBasicsApp — офлайн цифровий підручник через систему «єКафедра»



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

3

Рисунок В.3 - Слайд 3

Аналіз предметної галузі

Coursera: безкоштовний аудит / \$49+/mic. | Kotlin, XML | Peer-review | Мінімальна інтеграція в ЗВО

Udacity: ~\$300+/mic. | Kotlin, Jetpack, MVVM | Менторська перевірка | Інтеграція відсутня

Google Developers Training: безкоштовно | Kotlin, Compose | Автотести (обмежено) | Інтеграція відсутня

Висновок: жодна платформа не забезпечує чинних критеріїв оцінювання, банку завдань та інтеграції з ЗВО



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

4

Рисунок В.4 - Слайд 4

Структура дисципліни: 9 модулів

Модуль 1: Вступ і підготовка середовища (Android Studio, Kotlin, Gradle)

Модулі 2–3: UI/UX та архітектура (MVVM, Room Database, Repository Pattern)

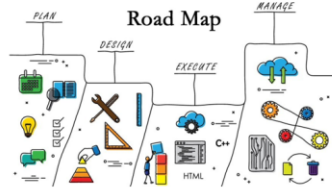
Модулі 4–5: Інтеграція з API (Retrofit, JWT) та живі дані (Coroutines, Flow, LazyColumn)

Модулі 6–7: Пристрої/сенсори (GPS, карти) та візуалізація даних (графіки, анімації)

Модулі 8–9: Файли/мультимедіа та заверш. проєкту (тестування, публікація у Play Store)

Тематика занять: моніторинг сонячних електростанцій, показники електроенергії, енергетичні об'єкти

Mobile App Development



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

5

Рисунок В.5 - Слайд 5

Технічний стек реалізації

Kotlin 2.0+ з KSP — основна мова: null-safety, extension functions, data classes

Jetpack Compose (BOM 2025.09.01) — LazyColumn, Navigation Compose, Material Design 3

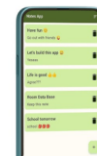
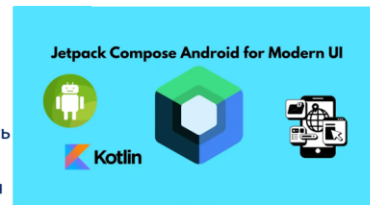
MVVM + Room Database (2.8.1) — збереження обраних занять (starred_practices)

Kotlin Coroutines (1.10.2) + StateFlow — реактивне управління фоновими операціями

Hilt DI (2.57.2) — @HiltViewModel, @Module/@Provides;

DataStore Preferences (1.1.7) — мовні налаштування

Android Studio Meerkat 2024.3.1 — Layout Inspector, Compose Preview, Profiler



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

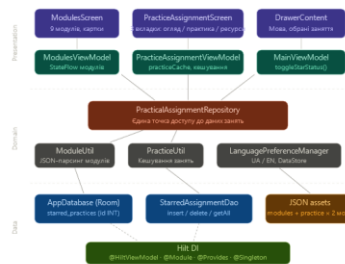
6

Рисунок В.6 - Слайд 6

Архітектура застосунку-HMK

MVVM + Clean Architecture: Data Layer / Utilities / Presentation Layer

Data: AppDatabase (Room), StarredAssignmentDao, PracticalAssignmentRepository, JSON-файли в assets
 Utils: ModuleUtil — парсинг JSON модулів;
 PracticeUtil — кешування занять (practiceCache)
 UI: ModulesScreen (9 розгорнутих карток) + PracticeAssignmentScreen (3 вкладки)
 Двомовність: ProvideLocalizedContext динамічно міняє мовний контекст без перезапуску Activity
 25+ унікальних Composable-компонентів; шрифт Wix Madefor Display у 5 накреслень



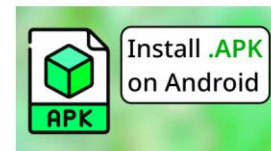
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

7

Рисунок В.7 - Слайд 7

Впровадження та розповсюдження

Система «Кафедра»: студент завантажує APK, встановлює на власний Android-пристрій
 Повна автономність: без Інтернету, без зовнішніх серверів, без передачі даних
 Android API 24+ (Android 7.0 Nougat+) — охоплює понад 97% активних пристроїв
 Оновлення: публікація нової версії APK між семестрами (аналог нового видання підручника)
 GitHub: KotlinBasicsApp + 27 репозиторіїв зразкових рішень (Варіант №1)



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

8

Рисунок В.8 - Слайд 8

Тестування системи

Unit-тести (JUnit 4): ModuleUtil та PracticeUtil — завантаження/парсинг JSON. 5/5 (100%)

Інструментальні тести (Espresso): запуск Activity, Hilt-граф, Room Database. 49/49 (100%)

Unit тести: перевірка роботи функцій, коректність результатів, дієздатність утиліт. 111/111 (100%)

Ручне тестування: 27 UI-сценаріїв + 54 екрани (UA/EN двомовність). 100%

Тестування зразкових рішень: 27 Android-проектів — успішна компіляція + базові тести

✓ 111 tests passed 111 tests total, 1 sec 74 ms

```
Starting 49 tests on Pixel_3_API_34(AVD) - 34
Pixel_3_API_34(AVD) - 34 Tests 7/49 completed. (0 skipped) (0 failed)
Pixel_3_API_34(AVD) - 34 Tests 12/49 completed. (0 skipped) (0 failed)
Pixel_3_API_34(AVD) - 34 Tests 13/49 completed. (0 skipped) (0 failed)
Pixel_3_API_34(AVD) - 34 Tests 17/49 completed. (0 skipped) (0 failed)
Pixel_3_API_34(AVD) - 34 Tests 21/49 completed. (0 skipped) (0 failed)
Pixel_3_API_34(AVD) - 34 Tests 25/49 completed. (0 skipped) (0 failed)
Pixel_3_API_34(AVD) - 34 Tests 30/49 completed. (0 skipped) (0 failed)
Finished 49 tests on Pixel_3_API_34(AVD) - 34
```



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

9

Рисунок В.9 - Слайд 9

Методичне проектування: структура заняття

Кожне з 27 занять містить 12 обов'язкових елементів (принцип конструктивного вирівнювання):

Тема та мета — вимірювані результати у форматі «здобувач вміє / знає / здатний»

Теоретична частина з прикладами Kotlin/Compose та посиланнями на офіційну документацію

30 варіантів індивідуальних завдань + 3 рівні складності: базовий / середній / складний

Критерії оцінювання сумарно 100 балів.

Зразковий код (Варіант №1) + структура проекту + 5–10 контрольних питань + посилання

Практичне завдання №1
Тема: Оптимізація з Android Studio, Kotlin, Jetpack Compose. Створення інтерфейсу «Hello World» за допомогою
Мета: Оптимізація кодування і покращення роботи мобільних застосунків на платформі Android із використанням Kotlin

Вступ

• Назва практичного завдання

• Короткий теоретичний вступ до теми

• Метами виконання практичного завдання, наприклад: завдання для розвитку 1 «Своєчасна оптимізація і покращення продуктивності»

• Визначення індивідуальних завдань

• Завдання за темою

• Короткий підсумок

• Домашнє завдання

• Контрольні питання

• Прикладні коментарі для самонавчання

• Система рекомендаційної структури

• Підсумок заняття



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

10

Рисунок В.10 - Слайд 10

Реалізація UI застосунку-НМК

KotlinBasicsApp: Android 7.0+ (97% пристроїв), 2 мови інтерфейсу (UA / EN)

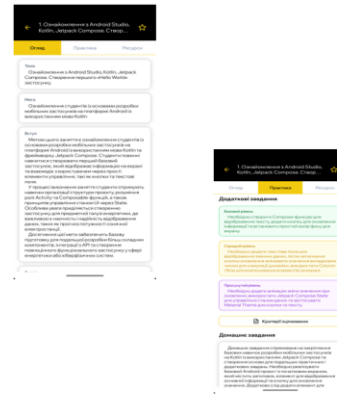
25+ Composable-компонентів: TitledSection, CollapsibleIndexedCard, ActionBarWithIcon та ін.

Головний екран: 9 розгорнутих карток модулів (ModulesScreen + ModuleList Composable)

Екран заняття: вкладки «Огляд», «Практика», «Ресурси» + зразковий код + посилання на GitHub

Типографіка: Wix Madefor Display у 5 накреслень, Material Design 3 компоненти

JSON-активи: modules_uk/en.json + practice_N_uk/en.json (27 занять × 2 мови = 56 файлів)



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

11

Рисунок В.11 - Слайд 11

Результати роботи

27 методичних описів занять + 27 зразкових Android-проектів на GitHub (Варіант №1)

30 варіантів індивідуальних завдань

800+ рядків тестового коду (unit + UI) як навчальний приклад підходу до тестування

25+ унікальних Composable-компонентів у застосунку KotlinBasicsApp

APK розповсюджується через «єКафедру», функціонує повністю автономно без Інтернету

НМК готовий до впровадження у дисципліну «Розробка ПЗ мобільних пристроїв» в КПІ



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

12

Рисунок В.12 - Слайд 12

Висновки

Розроблено сучасний НМК з 27 занять (9 модулів) — перший у кафедрі у форматі Android-застосунку

Реалізовано KotlinBasicsApp на Kotlin 2.0+, Jetpack Compose, MVVM, Room, Hilt — 100% тестів пройдено

Порівняльний аналіз виявив: жодна існуюча платформа не задовольняє потреби ЗВО повністю

НМК вирішує: критерії оцінювання + банк завдань + інтеграція з «єКафедрою» + офлайн-доступ

Перспективи: тісна інтеграція з «єКафедрою», автоматизована перевірка теоретичних знань студентів



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

13

Рисунок В.13 - Слайд 13



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"

Рисунок В.14 - Слайд 14